Human–Computer Interaction Series

Benjamin Weyers
Judy Bowen
Alan Dix
Philippe Palanque *Editors*

The Handbook of Formal Methods in Human-Computer Interaction

Springer

Human–Computer Interaction Series

More information about this series at http://www.springer.com/series/6033

Benjamin Weyers · Judy Bowen
Alan Dix · Philippe Palanque
Editors

The Handbook of Formal Methods in Human-Computer Interaction

Springer

Editors

Benjamin Weyers
Visual Computing Institute—Virtual Reality & Immersive Visualization
RWTH Aachen University
Aachen
Germany

Judy Bowen
Department of Computer Science
The University of Waikato
Hamilton
New Zealand

Alan Dix
School of Computer Science
University of Birmingham
Birmingham
UK

and

Talis Ltd.
Birmingham
UK

Philippe Palanque
IRIT—Interactive Critical Systems Group
University of Toulouse 3—Paul Sabatier
Toulouse
France

ISSN 1571-5035
Human–Computer Interaction Series
ISBN 978-3-319-51837-4 ISBN 978-3-319-51838-1 (eBook)
DOI 10.1007/978-3-319-51838-1

Library of Congress Control Number: 2016963425

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

The aim of this book was to present a snapshot of the use of Formal Methods in Human-Computer Interaction. Through contributed chapters and edited reviews, its goal was to capture the state of the art and suggest potential future topics for theoretical research and practical development.

Formal Methods have a long history in Human-Computer Interaction studies and play a central role in the engineering of interactive systems. Modelling, execution and simulation, as well as analysis, validation and verification, are key aspects for the application of formal methods in the engineering of interactive systems. This Handbook on Formal Methods in Human-Computer Interaction was motivated as an outcome of a workshop conducted at the 7th ACM SIGCHI Symposium on Engineering of Interactive Computing Systems (EICS) 2015 which took place in Duisburg, Germany. The main goal of this workshop was to offer an exchange platform for scientists who are interested in the formal modelling and description of interaction, user interfaces and interactive systems. The workshop further picked up the goal of a workshop organized at ACM SIGCHI conference on Human Factors in Computing Systems (CHI'96) in Vancouver, which was the initiation of a book publication released in 1997 edited by Palanque and Paternò.

Since then, various workshops have been conducted, among these the workshop on Formal Aspects of the Human Computer Interface (FAHCI) in 1996 in Sheffield and the series of workshops on "Formal methods and interactive systems" in 2006 in Macau, 2007 in Lancaster, 2009 in Eindhoven, 2011 in Limerick and 2013 in London. Nevertheless, the last comprehensive collection of work in Formal Methods in Human-Computer Interaction is published by Palanque and Paternò (1997). It succeeded the book published by Harrison and Thimbleby (1990), which presents a wide collection of techniques comparable to that by Palanque and Paternò. In this regard, this book presents an update after nearly two decades of work in the field.

Formal modelling in the description of interaction, user interfaces and interactive systems is still of great interest in current research and modelling strategies. Formal models offer advantages such as computer-based validation and verification, formal modification capabilities and the ability to be executable. In domains such as aeronautics, formal methods are required by standards (such as DO 178-C supplement 333[1]) making the use of both declarative formal approaches for requirements and property's description and procedural state-based formal approaches for systems descriptions mandatory. This use of formal methods is meant to complement (and not replace) testing approaches. Nevertheless, formal modelling has undergone various discussions identifying disadvantages, such as inflexibility, high modelling effort, high learning curve or inefficiency in a wider sense. The EICS 2015 workshop was intended to discuss existing formal modelling methods in this area of conflict. Therefore, participants were asked to present their perspectives, concepts and techniques for formal modelling by means of one or two case studies. The recommended case studies defined a basic structure for the workshop and further dealt as seed for discussions during the workshop. This idea has been picked up for this book, which provides an extended set of case studies that are used by authors throughout all chapters, which makes the presented methods and concepts comparable to a certain degree and offers indications to which focus the presented approaches address. The chapters are comprised of extended versions of the workshop's submissions as well as of chapters provided by invited authors who present their additional perspectives on the topic of Formal Methods in Human-Computer Interaction, complementing the workshop's submissions.

In this regard, the Handbook on Formal Methods in Human-Computer Interaction presents a comprehensive collection of methods and approaches, which address various aspects and perspectives on formal methods and highlight these by means of a compilation of case studies. The latter have been selected as relevant scenarios for formal methods in HCI each focusing on different aspects, such as multi-user versus single-user scenarios, safety critical versus non-critical situations, or WIMP-based versus multi-modal and non-WIMP-based interaction techniques. The book offers an extensive discussion of the state of the art, highlights the relevant topics in HCI for the application of formal methods, presents trends and gaps in the field of study and finally introduces the previously mentioned set of case studies. Furthermore, it contains three thematic sections in its main part.

The first section presents approaches and methods that focus on the use of formal methods for the modelling, execution and simulation of interactive systems. It introduces various methods for the modelling of user interfaces, description of the user's behaviour and a concept for the characterization of physical devices. Furthermore, it presents a concept for the description of low-intention interaction as well as a rule-based formalization of adaptable user interface models.

The second section focuses on the analysis, validation and verification of interactive systems using formal methods. Topics in this section address the verification of interaction through task-centric approaches, e.g. based on task decomposition or methods including models of the user's behaviour. It further presents an

[1] DO-178C/ED-12C, Software Considerations in Airborne Systems and Equipment Certification, published by RTCA and EUROCAE, 2012 and DO-333 Formal Methods Supplement to DO-178C and DO-278A, December 2011.

approach on the use of reasoning for the verification of interactive systems in dynamic situations as well as the specification and analysis of user properties of safety critical systems.

Finally, the third section concentrates on future opportunities and developments emerging in the field of Formal Methods in Human-Computer Interaction. It discusses an approach for enhanced modelling support using domain-specific descriptions, a tool suite for the specification of interactive multi-device applications as well as a method for the modelling of app-ensembles. Furthermore, work on the interactive support for validation and verification is presented as well as a perspective on dealing with faults during operations, describing fault-tolerant mechanisms for interactive systems using formal methods.

With this selection of topics, the book targets scientists interested in and working on formal methods for the description and analysis of interaction, user interfaces and interactive systems as well as practitioners who are interested in employing formal methods in the engineering of interactive systems. Through its thematic organization, the reader can approach the topics either by diving directly into one of these or by letting themselves be inspired by the various case studies. Furthermore, the reader can start with an overview of the past work or with a perspective on possible future investigations. A well-structured start to read is offered by the chapter on topics of Formal Methods in Human-Computer Interaction as it summarizes all chapters and puts them into context.

Finally, we would like to acknowledge all authors and contributors of this handbook as without their enthusiasm, work and large investment of time it would not have been possible to bring together this great compilation of methods, concepts and approaches that characterize the field of Formal Methods in Human-Computer Interaction. Furthermore, we would like to thank our families and supporters who first and foremost made it possible to write this book.

Aachen, Germany Benjamin Weyers
Hamilton, New Zealand Judy Bowen
Birmingham, UK Alan Dix
Toulouse, France Philippe Palanque

References

Harrison MD, Thimbleby HW (1990) Formal methods in human-computer interaction. Cambridge University Press, Cambridge.

Palanque P, Paternò F (1997) Formal methods in human-computer interaction. Springer Science & Business Media, Germany.

The original version of the book was revised: For detailed information please see erratum. The erratum to this book is available at 10.1007/978-3-319-51838-1_21

Contents

Part I Introduction

Part II Modeling, Execution and Simulation

Part III Analysis, Validation and Verification

Part IV Future Opportunities and Developments

Contributors

Eric Barboni ICS-IRIT, University of Toulouse, Toulouse, France

Ellen J. Bass Drexel University, Philadelphia, PA, USA

Matthew L. Bolton University at Buffalo, State University of New York, Buffalo, NY, USA

Judy Bowen University of Waikato, Hamilton, New Zealand

Guillaume Brat NASA Ames Research Center, Moffett Field, CA, USA

José Creissac Campos Dep. Informática/Universidade do Minho & HASLab/INESC TEC, Braga, Portugal

Sébastien Combéfis École Centrale des Arts et Métiers (ECAM), Woluwé-Saint-Lambert, Belgium

Paul Curzon EECS, Queen Mary University of London, Mile End Road, London, UK

Yannick Déléris AIRBUS Operations, Toulouse, France

Alan Dix School of Computer Science, University of Birmingham, Birmingham, UK; Talis Ltd., Birmingham, UK

Jean-Charles Fabre LAAS-CNRS, Toulouse, France

Racim Fahssi ICS-IRIT, University of Toulouse, Toulouse, France

Camille Fayollas ICS-IRIT, University of Toulouse, Toulouse, France; LAAS-CNRS, Toulouse, France

Masitah Ghazali University of Technology Malaysia, Johor, Malaysia

Dimitra Giannakopoulou NASA Ames Research Center, Moffett Field, CA, USA

Arnaud Hamon ICS-IRIT, University of Toulouse, Toulouse, France

Michael D. Harrison School of Computing Science, Newcastle University, Newcastle upon Tyne, UK

Annika Hinze University of Waikato, Hamilton, New Zealand

Marco Manca CNR-ISTI, HIIS Laboratory, Pisa, Italy

Célia Martinie ICS-IRIT, University of Toulouse, Toulouse, France

Paolo M. Masci Dep. Informática/Universidade do Minho & HASLab/INESC TEC, Braga, Portugal

Guillaume Maudoux Université catholique de Louvain, Louvain-la-Neuve, Belgium

Bart Meyers University of Antwerp, Antwerp, Belgium

Raquel Oliveira IRIT—MACAO Group, University of Toulouse 3—Paul Sabatier, Toulouse, France

Philippe Palanque IRIT—Interactive Critical Systems Group, University of Toulouse 3—Paul Sabatier, Toulouse, France

Fabio Paternò CNR-ISTI, HIIS Laboratory, Pisa, Italy

Charles Pecheur Université catholique de Louvain, Louvain-la-Neuve, Belgium

Johannes Pfeffer Technische Universität Dresden, Dresden, Germany

Franco Raimondi Middlesex University, London, UK

Steve Reeves University of Waikato, Hamilton, New Zealand

Rimvydas Rukšėnas EECS, Queen Mary University of London, Mile End Road, London, UK

Neha Rungta NASA Ames Research Center, Moffett Field, CA, USA

Carmen Santoro CNR-ISTI, HIIS Laboratory, Pisa, Italy

Leon Urbas Technische Universität Dresden, Dresden, Germany

Simon Van Mierlo University of Antwerp, Antwerp, Belgium

Yentl Van Tendeloo University of Antwerp, Antwerp, Belgium

Hans Vangheluwe University of Antwerp—Flanders Make, Antwerp, Belgium; McGill University, Montreal, Canada

Benjamin Weyers Visual Computing Institute—Virtual Reality & Immersive Visualization, RWTH Aachen University, Aachen, Germany

Part I
Introduction

Chapter 1
State of the Art on Formal Methods for Interactive Systems

Raquel Oliveira, Philippe Palanque, Benjamin Weyers, Judy Bowen and Alan Dix

Abstract This chapter provides an overview of several formal approaches for the design, specification, and verification of interactive systems. For each approach presented, we describe how they support both modelling and verification activities. We also exemplify their use on a simple example in order to provide the reader with a better understanding of their basic concepts. It is important to note that this chapter is not self-contained and that the interested reader should get more details looking at the references provided. The chapter is organized to provide a historical perspective of the main contributions in the area of formal methods in the field of human–computer interaction. The approaches are presented in a semi-structured way identifying their contributions alongside a set of criteria. The chapter is concluded by a summary section organizing the various approaches in two summary tables reusing the criteria previously derived.

R. Oliveira
IRIT— MACAO Group, University of Toulouse 3—Paul Sabatier, Toulouse, France

P. Palanque
IRIT— Interactive Critical Systems Group, University of Toulouse 3—Paul Sabatier, Toulouse, France
e-mail: palanque@irit.fr

B. Weyers (✉)
Visual Computing Institute—Virtual Reality & Immersive Visualization, RWTH Aachen University, Aachen, Germany
e-mail: weyers@vr.rwth-aachen.de

J. Bowen
Department of Computer Science, The University of Waikato, Hamilton, New Zealand
e-mail: jbowen@waikato.ac.nz

A. Dix
School of Computer Science, University of Birmingham, Birmingham, UK
e-mail: alanjohndix@gmail.com

A. Dix
Talis Ltd., Birmingham, UK

B. Weyers et al. (eds.), *The Handbook of Formal Methods in Human-Computer Interaction*, Human-Computer Interaction Series,
DOI 10.1007/978-3-319-51838-1_1

1.1 Introduction

Building reliable interactive systems has been identified as an important and difficult task from the late 1960s on (Parnas 1969), and methods and techniques developed in computer science have been applied, adapted, or extended to fit the need of interactive systems since then. Those needs have been thoroughly studied over the years, and the complexity of interactive systems has followed or even pre-empted the non-interactive part of computing systems. Such evolution is mainly due to the technological progression of input and output devices and their related interaction techniques.

Another important aspect is related to the intrinsic nature of the interactive systems as clearly identified in Peter Wegner's paper (Wegner 1997) as the input chain is not defined prior to the execution and the output chain is processed (by the users) before the "machine" (in the meaning of Turing machine) halts.

Two books (Harrison and Thimbleby 1990; Palanque and Paternó 1997) have been published to gather contributions related to the adaptation and extension of computer science modelling and verification techniques in the field of interactive systems. Contributions in these books were covering not only the interaction side, the computation side (usually called functional core), but also the human side by presenting modelling techniques applied, for instance, to the description of the user's mental models.

Over the years, the community in Engineering Interactive Computing Systems has been investigating various ways of using Formal Methods for Interactive Systems but has also broadened that scope proposing architectures, processes, or methods addressing the needs of new application domains involving new interaction techniques. Simultaneously, the Formal Methods for Interactive Systems community has been focusing on the use of formal methods in the area of interactive computing systems.

This chapter summarises selected contributions from those two communities over the past years. For each approach presented, we describe how they both support modelling as well as verification activities. We also exemplify their use on a simple example in order to provide the reader with a better understanding of their basic concepts. It is important to note that this chapter is not self-contained and that the interested reader should get more details looking at the references provided. This chapter is organized to provide a historical perspective of the main contributions in the area of formal methods in the field of human–computer interaction. Lastly, the approaches are presented in a semi-structured way identifying their contributions alongside a set of criteria. This chapter is concluded by a summary section organizing the various approaches in two summary tables reusing the criteria previously used.

1.2 Modelling and Formal Modelling

In systems engineering, modelling activity consists of producing a theoretical view of the system under study. This modelling activity takes place using one or several notations. The notation(s) allows engineers to capture some part of the system while ignoring other ones. The resulting artefact is called a model and corresponds to a simplified view of the real system.

In the field of software engineering, modelling is a well-established practice that was very successfully adopted in the area of databases (Chen 1976). More recently, it has been widely advertised by the UML standard (Booch 2005). It is interesting to see that UML originally proposed nine different notations and thus to produce as many different models to capture the essence of software systems. SysML (OMG 2010), the recent extension to UML, proposes two additional notations to capture elements that were overlooked by UML as, for instance, a requirements' notation. Modelling is advocated to be a central part of all the activities that lead up to the production of good software (Booch 2005). It is interesting to note that recent software engineering approaches such as agile processes (Schwaber 2004) and extreme programming (Beck 1999) moved away from modelling considering that on-time delivery of software is a much more important quality than correct functioning, as bugs can always be fixed in the next delivered version.

However, building models in the analysis, specification, design, and implementation of software bring a lot of advantages (Booch 2005; Turchin and Skii 2006):

- to abstract away from low-level details;
- to focus on some aspects while avoiding others (less relevant ones);
- to describe and communicate about the system under design with the various stakeholders;
- to better understand the system under development and the choices that are made; and
- to support the identification of relationships between various components of the system.

Beyond these advantages, modelling (when supported by notations offering structuring mechanisms) helps designers to break complex applications into smaller manageable parts (Navarre et al. 2005). The extent to which a model helps in the development of human understanding is the basis for deciding how good the model is (Hallinger et al. 2000).

When the notation used for building models has rigorous theoretical foundations, these models can be analysed in order to check soundness or detect flaws. Such activity, which goes beyond modelling, is called verification and validation and is detailed in the next section.

1.3 Verification and Validation

The notation used for describing models can be at various levels of formality that can be classified as informal, semi-formal, and formal (Garavel and Graf 2013):

- Informal models are expressed using natural language or loose diagrams, charts, tables, etc. They are genuinely ambiguous, which means that different readers may have different understanding of their meaning. Those models can be parsed and analysed (e.g. spell checkers for natural text in text editors), but their ambiguity will remain and it is thus impossible to guarantee that they do not contain contradictory statements.
- Semi-formal models are expressed in a notation that has a precise syntax but has no formal (i.e. mathematically defined) semantics. Examples of semi-formal notations are UML class diagrams, data flow diagrams, entity relationship graphical notation, UML state diagrams, etc.
- Formal models are written using a notation that has a precisely defined syntax and a formal semantics. Examples of formal specification languages are algebraic data types, synchronous languages, process calculi, automata, Petri nets, etc.

Thus, formal models are built using formal notations and are unambiguous system descriptions. Such formal models can then be analysed to assess the presence or absence of properties, analyse the performance issues (if the formal notation can capture such elements), possibly simulate the models to allow designer checking their behaviour, and generate descriptions in extension (such as state-space or test cases) if the formal notation represents such elements in intention (e.g. set of states represented in intention in Petri nets while represented in extension in an automata).

Formal verification involves techniques that are strongly rooted in mathematics. Defects in models can be detected by formal verification. In such cases, either the model has to be amended (to remove the defect) or the system under analysis has to be modified, for instance, by adding barriers (Basnyat et al. 2007). Such a modified system can then be modelled and analysed again to demonstrate that the modifications have not introduced other (unexpected) problems. This cycle (presented in Fig. 1.1) is repeated until the analysis results match the expectations. Examples of formal verification techniques are model checking, equivalence checking, and theorem proving.

Theorem proving is a deductive approach for the verification of systems (Boyer and Moore 1983). Proofs are performed in the traditional mathematical style, using some formal deductive system. Both the system under verification and the properties that have to be verified are modelled usually using different types of formal notations. Properties are usually expressed using declarative formal notations (e.g. temporal logics Clarke et al. 1986) while system behaviours are usually represented using procedural formal notations such as automata. Checking that the properties are true on a formal model of the systems is done as a theorem demonstration using the deductive proof calculus (see, for instance, verification of temporal logic

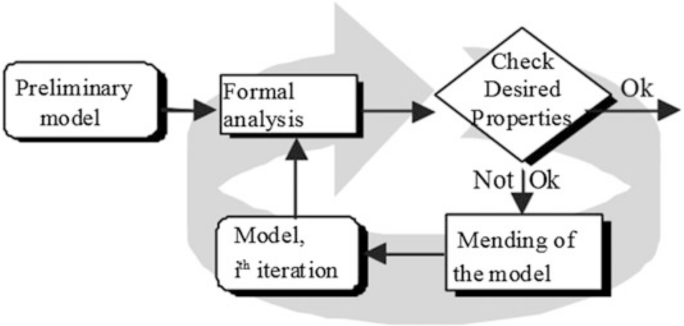


Fig. 1.1 Iterative cycle of models' construction and analysis from Palanque et al. (2009)

formulas over Petri nets Sifakis 1979). Proofs progress by transforming a set of premises into a desired conclusion, using axioms and deduction rules and possibly integrating previously demonstrated theorems. Such a proof production process is usually not fully automated: analyst guidance is required, for instance, regarding the proof strategy to be followed. Good user interfaces for theorem provers can significantly reduce the burden of the users as argued in Merriam and Harrison (1996). Some formal methods have been adapted to address the specificities of interactive systems and if they are specifically supporting theorem proving results, they have been tuned to address interactive systems properties such as the adaptation of B presented in Aït-Ameur et al. (2003a).

Model checking (Fig. 1.2) allows verification of whether a model satisfies a set of specified properties. A property is a general statement expressing an expected behaviour of the system. In model checking, a formal model of the system under analysis must be created, which is afterwards represented as a finite-state machine (FSM). This FSM is then subject to exhaustive analysis of its entire state space to determine whether the properties hold or not. The analysis can be fully automated and the validity of a property is always decidable (Cofer 2010). Even though it is easier for a human being to express properties in natural language, it can result in imprecise, unclear, and ambiguous properties. Expected properties should, thus, be also formalized by means of, for instance, a temporal logic. The analysis is mainly supported by the generation of counterexamples when a property is not satisfied. A counterexample can be a sequence of state changes that, when followed, leads to a state in which the property is false.

Since the introduction of model checking in the early 1980s, it has advanced significantly. The development of algorithmic techniques (e.g. partial-order reduction and compositional verification) and data structures (e.g. binary decision diagrams) allows for automatic and exhaustive analysis of finite-state models with several thousands of state variables (Aït-Ameur et al. 2010). For this reason, model checking has been used in the past years to verify interactive systems in

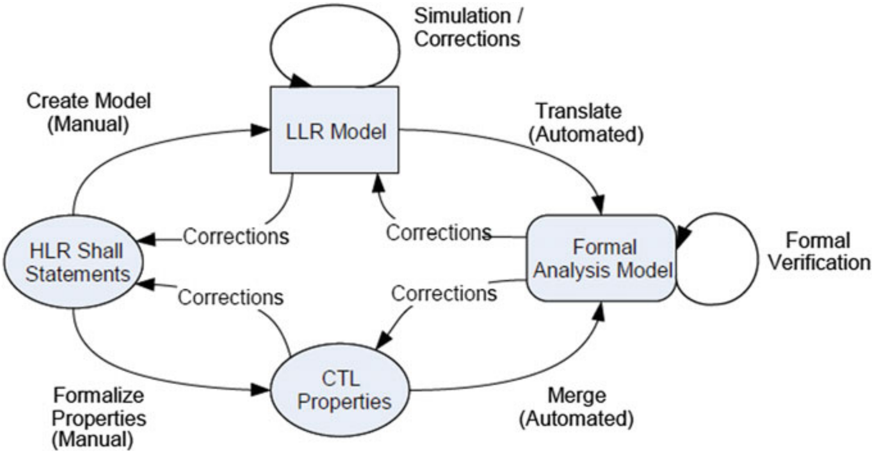


Fig. 1.2 Principle of model checking as defined in DO-178C aeronautics standard—*HLR* stands for high-level requirements, *LLR* stands for low-level requirements (and correspond to procedural systems descriptions)

safety-critical systems of several domains, such as avionics (Degani and Heymann 2002), radiation therapy (Turner 1993), and health care (Thimbleby 2010). In the field of interactive systems, several model-checking approaches have been proposed. With respect to mainstream software engineering, such approaches have been focussing on interactive systems-specific properties (such as predictability Dix 1991 or human errors identification Curzon and Blandford 2004).

Rather than verifying the satisfiability of properties, equivalence checking (Figs. 1.2 and 1.3) provides the ability to formally prove whether two representations of the system exhibit exactly the same behaviour or not. In order to verify whether two systems are equivalent or not, a model of each system should also be created, and then both models are compared in the light of a given equivalence relation. Several equivalence relations are available in the literature (e.g. strong bisimulation Park 1981 and branching bisimulation van Glabbeek and Weijland 1996). Which relation to choose depends on the level of details of the model and the verification goals. As for model checking and theorem proving, results of the analysis are exploited to identify where the models have to be amended in order to ensure their behavioural equivalence. In the field of interactive systems, this can be done for checking that two versions of interactive software exhibit the same behaviour or to check that the descriptions of user tasks are equivalent to the behaviour of the system (Palanque et al. 1995).

These three different approaches to formal verification have been applied to interactive systems in various works. In Sect. 1.6, we present those approaches by describing how formal models are described and how verification is addressed.

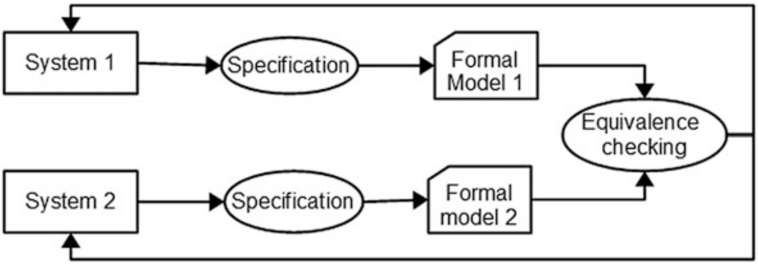


Fig. 1.3 Equivalence checking

1.4 Criteria to Describe and Analyse the State of the Art

Each approach is presented with respect to the following structure: after a brief introduction of the approach, it is unfolded step by step, identifying which language/formalism is used to model the interactive system. Then, the properties addressed by the approach are listed, together with the language/formalism used to describe them, the verification technique employed and whether the approach is tool supported or not. It is important to note that the approaches might be more powerful than presented here. Indeed, we only gather information that the authors were demonstrating in their publications. Thus, instead of presenting what an approach can do, we present what the authors have been doing with it.

After the description of each approach, an analysis is performed according to the following criteria:

- **Modelling coverage**: the verification of the system relies on the system model. For this reason, the model coverage should be large enough for the verification to be useful. It is analysed whether the studied approach covers aspects of the functional core and the user interfaces or not. The functional core of a system implements the domain-dependent concepts and functions, and the user interfaces implement the look and feel of the interactive system (Bass et al. 1991). We call a "User Interface" (UI) the information that is presented to the user with which the users can interact. In addition, it is also analysed if aspects of the users are included in the model, in order to take into account user behaviours. The more sophisticated the interaction techniques used to interact with these user interfaces, the more expressive power is required for the formal description technique. For instance, in multi-modal interactions, fusion of events is usually based on a temporal window in which the events have been received. If the events are too far away (in quantitative time), then they will not be fused. In order to describe such behaviour, the formal methods must allow engineers to describe quantitative temporal information (such as timed automata or temporal Petri nets).
- **Kinds of properties**: one kind of analysis that can be performed over a system model is property verification. In the context of safety-critical interactive

systems, we believe that the focus should be directed both towards dependability (to ensure that the functioning of the system is correct), to usability (to ensure that the system is usable; effective, efficient, and satisfactory), and to prevent users from making errors. For each author, the kinds of properties that have been demonstrated as verifiable using their approach are analysed.

- **Application to safety-critical systems**: whether each approach is applied to safety-critical domains or not. We provide here examples of the domains addressed, e.g. health care, air traffic management, avionics, or nuclear power.
- **Scalability**: while a lot of work has been performed on simple examples, we will identify approaches that have been applied to industrial applications or at least have demonstrated means (e.g. structuring mechanisms) for dealing with real-life systems.

1.5 Modelling and Verification

Interactive systems models can deal with the various aspects of interactive systems. Low-fidelity prototypes usually deal with their presentation part (i.e. how they look and how information is presented to users) while behavioural models usually address interaction or dialogue descriptions. While a model at specification level would describe **what** the system is supposed to do, models at design levels would describe **how** the system is supposed to behave. In the area of interactive systems, formal models have been proposed at different levels. Properties are closer to the specification level as they express constraints on system presentation or behaviour. A presentation property would require, for instance, that all the user interface buttons have the same size. A behavioural property, for instance, could require that all buttons are always available. Verification activity aims at assessing whether or not a property holds on a given system as discussed above.

Several authors propose different categories of properties. For instance, three kinds of properties are identified in Campos and Harrison (1997): **visibility** properties, which concern the users' perception, i.e. what is shown on the user interface and how it is shown; **reachability** properties, which concern the user interfaces, and deal with what can be done at the user interface and how it can be done (in the users' perspective); and **reliability** properties, which concern the underlying system, i.e. the behaviour of the interactive system.

1.6 Succinct Presentation of the Approaches

This section will briefly describe the approaches reviewed for this chapter. The current FoMHCI community and many of the strands of work in this review largely owe their origin to a number of projects funded by the Alvey Programme in the UK in the 1980s, and particularly the "York Approach" (see Dix et al. below). However, it is possible to trace the roots deeper, in particular Reisner's (1981) use of BNF to describe the "action language" (what we would now call dialogue) of an interactive graphics programme, and Sufrin's (1982) use of the Z specification language to specify a simple display editor (see d'Ausbourg 1998 for an early review).

Both Reisner's and Sufrin's work used existing formal notations. This use of existing notations or creation of specialized notations or methods for interactive systems has always been one of the main strands of FoMHCI research. Many of the approaches below and in this book use existing notations (e.g. LOTOS, Petri nets); however, these often either need to extend or create new notations in order to be able to effectively specify behaviours and properties of interactive systems.

While this has emerged as the dominant strand of work in the area and the main focus of this review, there are a number of other strands that have influenced the field, elements of which can be seen in various chapters of this book (see also Dix 2012).

- **Abstract models**: this used a variety of notations, but with the aim of describing classes of systems to define generic properties and prove generic properties (see Dix et al. below). The main legacy of this approach is the formulation of properties including variations of predictability and observability that are adopted by many system modelling approaches, which can be seen in many of the approaches below.
- **Architectural models**: early user interface implementers reflected on their experience. The MVC (Model–View–Controller) paradigm grew out of the Smalltalk programming environment (Kieras and Polson 1985), and a workshop of those developing User Interface Management Systems (UIMS) led to the Seeheim Model (Pfaff and Hagen 1985). The former has been particularly influential in subsequent practical UI development, and the latter in framing a language for interaction architecture, especially the formulation of the presentation–dialogue-functionality distinction. Within the formal modelling community, this work was especially strongly associated with the work of Coutaz, Nigay, and others at Grenoble including the development of the PAC model (Coutaz 1987), which itself fed into the ARCH/Slinky metamodel (Bass et al. 1991). The main legacy of this work has been in its inputs into modelling of multi-modal systems and plasticity. Oddly, many current systems that describe themselves as MVC are actually unintentionally following PAC model (Dey 2011).
- **User and task modelling**: the cognitive modelling and task analysis communities have often used models that have a formal nature, although come from different roots and have had different concerns to those adopting a more computer science formal modelling approach. However, there have been many overlaps including CCT (Cognitive Complexity Theory), which used a dual system and cognitive model (Kieras and Polson 1985), and TAG (Task Action Grammar), which expressed system descriptions in ways that made

inconsistencies obvious (Payne and Green 1986). Of course, the CTT task modelling approach (see Paterno et al. below) has been very influential in the FoMHCI community, and, while having its roots in the LOTOS specification notation, it is very similar to pure task analysis notations such as HTA (Shepherd 1989).

1.6.1 Abowd et al. (USA 1991–1995)

Early approaches to applying formal notations to the study of human–machine interaction and the modelling of interactive systems paved the way for other researchers to explore different alternatives to assess the quality of such systems. In Abowd (1991), a framework for the formal description of users, systems, and user interfaces is proposed.

1.6.1.1 Modelling

In Abowd (1991), the interactive system is modelled as a collection of agents. The language to describe the agents borrows notations from several formal languages, such as Z, VDM, CSP, and CSS. Such agent language contains identifiers to describe internal (*types, attributes, invariants, initially*, and *operations*) and external specifications of agents, as well as communication between agents (input/output). Therefore, data, states, and events can be modelled in this language. When describing the *operations* agents can execute, it is possible to define *pre*- and *post-conditions* for each operation, which may be used to define a given ordering of actions, allowing *qualitative time* to be represented. The external specification of the language allows description of synchronous parallel composition, which can express *concurrent behaviour.* Finally, multi-touch interactions can be implicitly modelled by agents: each finger could be represented by a given agent.

Alternatively, another approach is proposed in Abowd et al. (1995), Wang and Abowd (1994), in which interactive systems are described by means of a tabular interface using Action Simulator, a tool for describing PPS (propositional production system) specifications. In PPS, the dialogue model is specified as a number of production rules using pre- and post-conditions. Action Simulator permits such PPS specification to be represented in a tabular format, in which the columns are the system states, and the production rules are expressed at the crossings of lines and columns. It is possible to represent *multi-modality* using this approach by identifying the states related to each modality, and how they relate to each other using the production rules. However, it does not allow *concurrent behaviour* to be expressed: production rules are executed sequentially. The approach covers the modelling of the functional core and the UIs, and to some extent the modelling of the users, by describing the user actions that “fire” the system state changes.

1.6.1.2 Verification

The verification of specifications written using the agent language described in Abowd (1991) can be tool supported, for instance, by using ProZ for Z specifications. However, such verification is not described in Abowd (1991).

A translation from the tabular specification of the interactive system proposed in (Abowd et al. 1995; Wang and Abowd 1994) into SMV input language is described in Wang and Abowd (1994). The CTL temporal language is used to formalize the properties, allowing the following usability properties to be verified:

- **Reversibility (Abowd et al**. 1995): Can the effect of a given action be reversed in a single action?
- **Deadlock freedom (Abowd et al.** 1995): From an initial state, is it true that the dialogue will never get into a state in which no actions can be taken?
- **Undo within N steps (Wang and Abowd** 1994): From any state of a given state set, if the next step leads the system out of the state set, can a user go back to the given state set within N steps?

In addition, the following functional properties can be verified:

- **Rule set connectedness (Abowd et al.** 1995): From an initial state, can an action be enabled?
- **State avoidability (Wang and Abowd** 1994): Can a user go from one state to another without entering some undesired state?
- **Accessibility (Wang and Abowd** 1994): From any reachable state, can the user find some way to reach some critical state set (such as the help system)?
- **Event constraint (Wang and Abowd** 1994): Does the dialogue model ensure/prohibit a particular user action for a given state set?
- **Feature assurance (Wang and Abowd** 1994): Does the dialogue model guarantee a desired feature in a given state set?
- **Weak task completeness (Abowd et al.** 1995): Can a user find some way to accomplish a goal from initialization?
- **Strong task completeness (Abowd et al.** 1995): Does the dialogue model ensure that a user can always accomplish a goal?
- **State inevitability (Abowd et al.** 1995): From any state in the dialogue, will the model always allow the user to get to some critical state?
- **Strong task connectedness (Abowd et al.** 1995): From any state, can the user find some way to get to a goal state via a particular action?

The automatic translation of the tabular format of the system states into the SMV input language is an advantage of the approach, since it allows model checking of properties to be performed. The tabular format of the system states and the actions that trigger state changes provide a reasonable compact representation in a comprehensible form. However, it looks like the approach does not scale well to larger specifications, unless an alternative way to store a large sparse matrix is provided. Besides, no application to safety-critical systems is reported.

1.6.2 Dix et al. (United Kingdom 1985–1995)

1.6.2.1 Modelling

The PIE model (Dix et al. 1987) considers interactive systems as a “black-box” entity that receives a sequence of inputs (keystrokes, clicks, etc.) and produces a sequence of perceivable effects (displays, LEDs, printed documents, etc.). The main idea is to describe the user interfaces in terms of the possible inputs and their effects (Dix 1991). Such practice is called surface philosophy (Dix 1988) and aims at omitting parts of the system that are not apparent to the user (the internal details of systems, such as hardware characteristics, languages used, or specification notations). The domain of input sequences is called P (standing for programs), the domain of effects is called E, and both are related by an interpretation function I that determines the effects of every possible command sequence (Fig. 1.4). In this sense, the interpretation function I can be seen as a means to represent *events* of the modelled system, *data* cannot be represented, and internal *states* of the system are inferred by what is called *observable effects* (Dix 1991).

The effects E can be divided into permanent results (e.g. printout) and ephemeral displays (the actual UI image). Such specialization of the effects constitutes another version of the PIE model, called the Red-PIE model (Dix 1991).

The PIE model is a single-user single-machine model and does not describe interleaving and the timing of the input/output events (Dix 1991). However, extensions of the basic PIE model dealt with *multi-user* behaviour (including the first formulations of collaborative undo Abowd and Dix 1992); the first formal work on *real-time* interactive behaviours (Dix 1991); continuous interaction (such as mouse dragging) through status-event analysis (Dix 1991); and non-deterministic external behaviours (e.g. due to concurrency or race conditions) (Dix 1991). *Multi-modality* can be expressed by describing the input/output and interpretation function for each modality, the status–event analysis extensions would allow multi-touch applications.

The PIE model is focused on the external behaviour of the system as perceived by the user; it does not model the users themselves, nor more than minimal internal details. Because of this, the external effects of internal behaviour such as *concurrency behaviour* or *dynamic instantiation* can be modelled, but not their internal mechanisms.

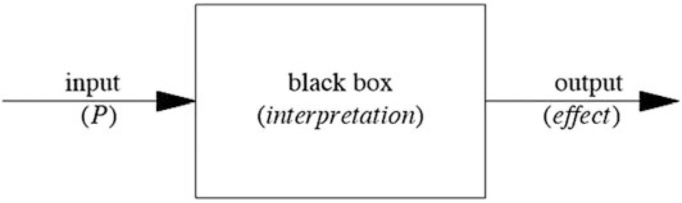


Fig. 1.4 The PIE model (Dix 1991)

1.6.2.2 Verification

The PIE model provides a generic way of modelling interactive systems and permits the following usability properties to be formalized:

- **Predictability (Dix 1995)**: The UI shall be predictable, i.e., from the current effect, it should be possible to predict the effect of future commands.
- **Simple reachability (Dix 1991)**: All system effects can be obtained by applying some sequences of commands;
- **Strong reachability (Dix 1988)**: One can get anywhere from anywhere;
- **Undoability (Dix et al. 1987)**: For every command sequence, there is a function "undo" which reverses the effect of any command sequence;
- **Result commutativity (Dix et al. 1987)**: Irrespective of the order in which different UIs are used, the result is the same.

The PIE and Red-PIE models are one of the first approaches that used formal notations for the modelling of interactive systems and desired properties. As abstract models, their role in respect of verification is therefore more in formulating the user interaction properties that subsequent system modelling and specification approaches (such as Abowd et al., above and Paterno et al., below) seek to verify for specific systems.

Some proofs and reasoning about PIEs are quite extensive, notably Mancini's category theoretical proof of the universality of stack-and-toggle-based undo (Mancini 1997). However, the mathematical notations are very abstract, and no tool support is provided; instead, proofs follow a more traditional mathematical form.

1.6.3 Paternò et al. (Italy 1990–2003)

1.6.3.1 Modelling

Interactive systems can be formally described as a composition of interactors (Hardin et al. 2009). Interactors are more concrete than the agent model described in section (Abowd 1991 above), in that they introduce more structure to the specification by describing an interactive system as a composition of independent entities (Markopoulos 1997).

The interactors of CNUCE (Paternó and Faconti 1992) provide a communication means between the user and the system. Data manipulated by the interactors can be sent and received through events in both directions: towards the system and towards the user (Paternó 1994), which are both abstracted in the model by a description of the possible system and user actions.

The CNUCE interactors are specified using LOTOS (ISO 1989), which has concurrent constructs. However, since LOTOS is a language with action-based semantics, the system states cannot be represented. Besides, only *qualitative time* can be modelled, *dynamic instantiation* cannot be modelled, neither *multi-modality*.

Multi-touch interactions can be modelled by defining one interactor for each finger, and by integrating these interactors to other interactors of the system. Despite the fact that the approach covers mainly the modelling of user interfaces, a mathematical framework is provided to illustrate how to model the user and the functional core too (Paternó 1994).

1.6.3.2 Verification

Figure 1.5 illustrates how a formal model using CNUCE interactors can be used afterwards for verification (Paternó 1997).

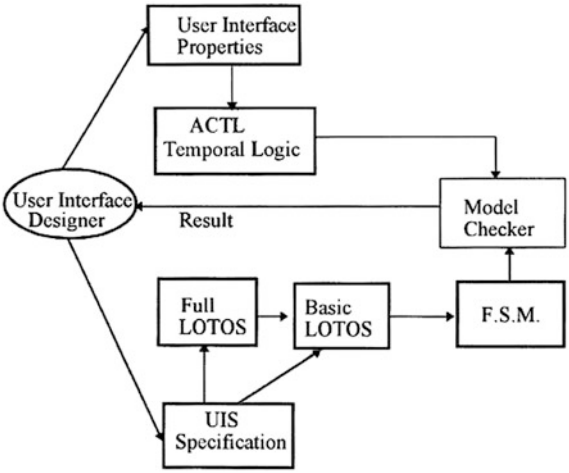
This approach has been used to verify the following usability properties:

- **Visibility (Paternó and Mezzanotte** 1994): Each user action is associated with a modification of the presentation of the user interface to give feedback on the user input;
- **Continuous feedback (Paternó and Mezzanotte** 1994): This property is stronger than visibility; besides requiring a feedback associated with all possible user actions, this has to occur before any new user action is performed;
- **Reversibility (Paternó and Mezzanotte** 1994): This property is a generalization of the undo concept. It means that users can perform parts of the actions needed to fulfil a task and then perform them again, if necessary, before the task is completed in order to modify its result;
- **Existence of messages explaining user errors (Paternó and Mezzanotte** 1994): Whenever there is a specific error event, a help window will appear.

In addition, the following functional property can be verified:

- **Reachability (Paternó and Mezzanotte** 1994): This property verifies that a user interaction can generate an effect on a specific part of the user interface.

Fig. 1.5 The TLIM (tasks, LOTOS, interactors modelling) approach (Paternó 1997)



The approach has been applied to several case studies of safety-critical systems in the avionics domain (Navarre et al. 2001; Paternó and Mezzanotte 1994, 1996; Paternó 1997; Paternó and Santoro 2001, 2003). These examples show that the approach scales well to real-life applications. Large formal specifications are obtained, which describe the behaviour of the system, permitting meaningful properties to be verified.

1.6.4 Markopoulos et al. (United Kingdom 1995–1998)

1.6.4.1 Modelling

ADC (Abstraction–Display–Controller) (Markopoulos 1995) is an interactor model that also uses LOTOS to specify the interactive system (specifically, the UIs). In addition, the expression of properties is facilitated by templates.

The ADC interactor handles two types of data: display data, which come (and are sent to) either directly from the UI or indirectly through other interactors, and abstraction data, which are sustained by the interactor to provide input to the application or to other interactors (Markopoulos et al. 1998). A UI can be modelled as a composition of ADC interactors. Once formalized in LOTOS, the ADC interactors can be used to perform formal verification of usability properties using model checking.

The ADC approach concerns mostly the formal representation of the interactor model. Regarding the coverage of the model, the focus is to provide an architectural model for user interface software. The functional core and the user modelling are not covered. ADC emphasizes the architectural elements of the interactor: its gates, their role, their grouping to sides, the separate treatment of dialogue and data modelling and the composition of interactors to form complex interface specifications (Markopoulos et al. 1998). When connected to each other, ADC interactors exchange data through gates. Connection types (*aout, dout*), (*dout, dout*), and (*aout, aout*) concern pairs of interactors which synchronize over common output gates. These can be useful for modelling multi-modal output where different output modalities synchronize, e.g. sound and video output (Markopoulos et al. 1998). Touch interactions can also be modelled by the combination of such interactors.

1.6.4.2 Verification

The properties to be verified over the formal model are specified in the ACTL temporal logic. For example, the following properties can be verified:

- **Determinism (Markopoulos** 1997): A user action, in a given context, has only one possible outcome;

- **Restartability (Markopoulos 1995)**: A command sequence is restartable if it is possible to extend it so that it returns to the initial state;
- **Undoability (Markopoulos 1995)**: Any command followed by undo should leave the system in the same state as before the command (single step undo);
- **Eventual feedback (Markopoulos et al. 1998)**: A user-input action shall eventually generate a feedback.

In addition, the following functional properties can be verified:

- **Completeness (Markopoulos 1997)**: The specification has defined all intended and plausible interactions of the user with the interface;
- **Reachability (Markopoulos 1997)**: It qualifies the possibility and ease of reaching a target state, or a set of states, from an initial state, or a set of states.

In this approach, the CADP (Garavel et al. 2013) toolbox is used to verify properties by model checking (Markopoulos et al. 1996). Specific tools to support the formal specification of ADC interactors are not provided (Markopoulos et al. 1998).

No case study applying the approach to the verification of critical systems is reported. In fact, the approach is applied to several example systems (Markopoulos 1995) and to a case study on a graphical interface of Simple Player for playing movies (Markopoulos et al. 1996), which makes it difficult to measure whether it can scale up to realistic applications or not.

1.6.5 Duke and Harrison et al. (United Kingdom 1993–1995)

1.6.5.1 Modelling

Another interactor model is proposed by the University of York (Duke and Harrison 1995) to represent interactive systems. Compared to the CNUCE interactor model (below), the main enhancement brought by the interactors of York is an explicit representation of the state of the interactor.

The York interactor (Fig. 1.6) has an internal state and a rendering function (i.e. rho in Fig. 1.6) that provides the environment with a perceivable representation (P) of the interactor internal state. The interactor communicates with the environment by means of events. Two kinds of events are modelled:

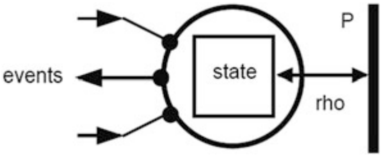


Fig. 1.6 The York interactor (Harrison and Duke 1995)

- **Stimuli events**: They come from either the user or the environment and modify the internal state of the interactor. Such state changes are then reflected to the external presentation through the rendering function.
- **Response events**: These events are generated by the interactor and sent to the user or to the environment.

York interactors are described using the Z notation (Spivey 1989). This notation facilitates the modelling of the state and operations of a system, by specifying it as a partially ordered set of events in first-order logic (Duke and Harrison 1993). Multi-modality can be represented (Duke and Harrison 1995) as each action has an associated modality from the given set *[modality]*. Besides, the authors describe two approaches to define a notion of interaction: the *one-level model* and the *two-level model*, which bound several interactors (Duke and Harrison 1993). This allows multi-touch interactions to be represented by this approach. In addition, the York interactor model provides an abstract framework for structuring the description of interactive systems in terms of layers. It encapsulates two specific system layers: the state and the display (Harrison and Duke 1995), thus covering both the functional core and the UIs in the modelling. However, *concurrent behaviour* cannot be expressed, neither does it support *dynamic instantiation* (even though instantiation is proposed to compose interactors (Duke and Harrison 1993), it seems that the interactors are not dynamically instantiated).

1.6.5.2 Verification

This approach permits usability properties expressed in first-order logic formulas to be verified. Unlike the previous approaches, the York interactor model uses theorem proving as formal verification technique. Examples of properties that can be verified are (Duke and Harrison 1995):

- **Honesty**: The effects of a command are intermediately made visible to the user;
- **Weak reachability**: It is possible to reach any state through some interaction;
- **Strong reachability**: Each state can be reached after any interaction p; and
- **Restartability**: Any interaction p is a prefix of another q such that q can achieve any of the states that p initially achieves.

The approach is applied to a case study in a safety-critical system, an aircraft's fuel system (Fields et al. 1995) in which the pilot's behaviour is modelled, thus showing that the approach also covers the modelling of users. No further case studies applying the approach were found in the literature, which makes it difficult to tell whether the approach scales up to larger interactive systems or not.

1.6.6 Campos et al. (Portugal 1997–2015)

1.6.6.1 Modelling

The York interactor model is the basis of the work proposed in Bumbulis et al. (1995b). Here, Campos chooses MAL (Modal Action Logic) language to implement the York interactor model, since MAL's structure facilitates the modelling of the interactor behaviour. The use of MAL allows *data* to be represented, since *attributes* can be expressed in the language. In Campos and Harrison (2001), the authors propose the MAL interactor language to describe interactors that are based on MAL, propositional logic is augmented with the notion of action, and deontic operators allows ordering of actions to be expressed.

The approach covers the three aspects we are considering: in Campos and Harrison (2011), the approach is used to model the functional core and user interfaces of an infusion pump; and assumptions about user behaviours are covered in Campos and Harrison (2007), by strengthening the preconditions on the actions the user might execute.

A tool called i2smv is proposed in Campos and Harrison (2001) to translate MAL specifications into the input language of the SMV model checker. However, *concurrent behaviour* cannot be modelled. Although the stuttering in the SMV modules allows interactors to evolve independently, a SMV module will engage in an event while another module does nothing (Campos and Harrison 2001).

1.6.6.2 Verification

The approach is applied to several case studies. An application of both model checking and theorem proving to a common case study is described. Further, deeper investigations are performed (and tools developed) into the usage of model checking (only), in order to verify interactive systems.

To support the whole process, a toolbox called IVY is developed (Campos and Harrison 2009). In this framework, the properties are specified using the CTL (computational tree logic) temporal logic, allowing the verification of usability and functional properties (Campos and Harrison 2008). Particularly, the following usability properties can be expressed:

- **Feedback (Campos and Harrison** 2008**)**: A given action provides a response;
- **Behavioural consistency (Campos and Harrison** 2008**)**: A given action causes consistent effect;
- **Reversibility (Campos and Harrison** 2008**)**: The effect of an action can be eventually reversed/undone;
- **Completeness (Campos and Harrison** 2009**)**: One can reach all possible states with one action.

The approach is applied to several case studies (Campos and Harrison 2001; Harrison et al. 2013), specifically in safety-critical systems (e.g. healthcare systems Campos 1999; Campos and Harrison 2009, 2011; Harrison et al. 2015 and avionics systems Campos and Harrison 2007; Doherty et al. 1998; Sousa et al. 2014), showing that the approach scales well to real-life applications.

1.6.7 d'Ausbourg et al. (France 1996–2002)

1.6.7.1 Modelling

Another approach based on the York interactor model is proposed in d'Ausbourg (1998) and d'Ausbourg et al. (1998). These authors push further the modelling of an interactive system by events and states initially proposed by the York approach.

Their interactor model is called CERT. It also contains an internal state, and the interface between an interactor and its environment consists of a set of input and output events. Both internal state and events are described as Boolean flows. Such representation of interactors by flows allows their specification using the LUSTRE data flow language. A system described in LUSTRE is represented as a network of nodes acting in parallel, which allows *concurrent behaviour* to be represented. Each node transforms input flows into output flows at each clock tick.

The approach can handle *data* in the system modelling. However, a drawback is that it does not handle sophisticated data types. The representation of the internal system state and events by Boolean flows considerably limits the modelling capabilities of the approach. In LUSTRE, a flow variable is a function of time, denoting the sequence of values that it takes at each instant (d'Ausbourg et al. 1998). Specifically, two LUSTRE operators allow *qualitative time* to be represented: the "previous" operator *pre* and the "followed-by" operator $\rightarrow$. Besides, *quantitative time* can also be represented: the expression *occur-from-to(a, b, c)* is a temporal operator whose output is true when "*a*" occurs at least once in the time interval [b…c] (d'Ausbourg et al. 1998).

1.6.7.2 Verification

The LUSTRE formal model is then verified by model checking. Verification is achieved by augmenting the system model with LUSTRE nodes describing the intended properties, and using the Lesar tool to traverse the state space generated from this new system. The properties can be either specific or generic properties.

Specific properties deal with how presentations, states, and events are dynamically linked into the UIs, and they are automatically generated from the UIL file (they correspond to functional properties). Generic properties might be checked on any user interface system, and they are manually specified (they correspond to usability properties). The verification process allows the generation of test cases, using the behaviour traces that lead to particular configurations of the UI where the properties are satisfied.

In particular, the following usability properties are verified (d'Ausbourg et al. 1998):

- **Reactivity**: The UI emits a feedback on each user action;
- **Conformity**: The presentation of an interactor is modified when its internal state changes;
- **Deadlock freedom**: The impossibility for a user to get into a state where no actions can be taken;
- **Unavoidable interactor**: The user must interact with the interactor at least once in any interactive session of the UIs.

As well as the following functional property:

- **Rule set connectedness**: An interactor is reachable from any initial state.

The approach was applied to the avionics field (d'Ausbourg 2002). In this case study, the interactions of the pilot with the system and the behaviour of the functional core are modelled. Unfortunately, no evidence is given that the approach scales well to real-life applications.

1.6.8 Bumbulis et al. (Canada 1995–1996)

1.6.8.1 Modelling

Similar to the interactor models (Campos and Harrison 2001; d'Ausbourg et al. 1998; Duke and Harrison 1995), user interfaces can be described by a set of interconnected primitive components (Brat et al. 2013; Bumbulis et al. 1995a). The notion of component is similar to that of interactor, but a component is more closely related to the widgets of the UI. Such component-based approach allows both rapid prototyping and formal verification of user interfaces from a single UI specification.

In Bumbulis et al.'s approach, user interfaces are described as a hierarchy of interconnected component instances using the Interconnection Language (IL). Investigations have been conducted into the automatic generation of IL specifications by re-engineering the UIs (Bumbulis et al. 1995a). However, such automatic generation is not described in the paper. From such component-based IL specification of the UI, a Tcl/Tk code is mechanically generated, in order to provide a UI prototype for experimentation, as well as a HOL (higher-order logic) specification for formal reasoning using theorem proving (Bumbulis et al. 1995a).

The approach covers only the modelling and verification of user interfaces. The user and the functional core are not modelled. Besides, the Interconnection Language does not provide means to represent *multi-modality*, *multi-touch* interactions, *concurrent behaviour*, or *time.*

1.6.8.2 Verification

Properties are specified as predicates in Hoare logic, a formal system with a set of logical rules for reasoning about the correctness of computer programs. Proofs are constructed manually, even though investigations to mechanize the process have been conducted (Bumbulis et al. 1995a). No usability properties are verified in this approach. Instead, the approach permits functional properties to be verified, which are directly related to the expected behaviour of the modelled UI.

No application to safety-critical systems was found in the literature. Besides, it is not clear how to model more complex UIs in this approach, since UI components are not always bound to each other. In addition, it is not clear how multiple UIs could be modelled, neither the navigation modelling between such UIs. All these aspects indicate that the approach does not scale well for larger applications.

1.6.9 Oliveira et al. (France 2012–2015)

1.6.9.1 Modelling

In Oliveira et al. (2015a), a generic approach to verifying interactive systems is proposed, but instead of using interactors, interactive systems are described as a composition of *modules*. Each system component is described as such a *module*, which communicates and exchanges information through channels. This approach allows *plastic* UIs to be analysed. *Plasticity* is the capacity of a UI to withstand variations in its *context of use* (environment, user, platform) while preserving usability (Thevenin and Coutaz 1999). In this approach, interactive systems are modelled according to the principles of the ARCH architecture (Bass et al. 1991), and using LNT (Champelovier 2010), a formal specification language derived from the ELOTOS standard (ISO 2001). LNT improves LOTOS (ISO 1989) and can be translated to LOTOS automatically. LOTOS and LNT are equivalent with respect to expressiveness, but have a different syntax. In Paternó (1997), the authors point out how difficult it is to model a system using LOTOS, when quite simple UI behaviours can easily generate complex LOTOS expressions. The use of LNT alleviates this difficulty.

The approach enhances standard LTS to model interactive systems. An LTS represents a system by a graph composed of states and transitions between states. Transitions between states are triggered by actions, which are represented in LTS transitions as labels. Intuitively, an LTS represents all possible evolutions of a system modelled by a formal model. The approach enhances LTS by proposing the ISLTS (Interactive System LTS) (Oliveira 2015), in which two new sets are added: a set C of UI components and a set L of action names. In addition, the set A of actions of standard LTS is enhanced to carry a list of UI components, representing the UI appearance after the action is performed.

The approach covers aspects of the users, the user interfaces, and the functional core of the system. *Data* and *events* of the system can be modelled by the ISLTS, but not the *state* of the system. The ordering of transitions of the LTS can represent *qualitative time* between two consecutive model elements, and LNT contains operators that allow *concurrent behaviour* to be modelled. Models described with this approach were of WIMP-type, and no evidence was given about the ability of the approach to deal with more complex interaction techniques such as multi-touch or multi-modal UIs (especially, dealing with quantitative time required for fusion engines).

1.6.9.2 Verification

The approach is twofold, allowing: *usability* and *functional* properties to be verified over the system model (Oliveira et al. 2014). Using *model checking,* usability properties verify whether the system follows ergonomic properties to ensure a good usability. Functional properties verify whether the system follows the requirements that specify its expected behaviour. These properties are formalized using the MCL property language (Mateescu and Thivolle 2008). MCL is an enhancement of the modal mu-calculus, a fixed point-based logic that subsumes many other temporal logics, aiming at improving the expressiveness and conciseness of formulas.

Besides, different versions of UIs can be compared (Oliveira et al. 2015b). Using *equivalence checking*, the approach verifies to which extent UIs present the same interaction capabilities and appearance, showing whether two UI models are equivalent or not. When they are not equivalent, the UI divergences are listed, providing the possibility of leaving them out of the analysis (Oliveira et al. 2015c). In this case, the two UIs are equivalent less such divergences. Furthermore, the approach shows that one UI can contain at least all interaction capabilities of another (UI inclusion). Three abstraction techniques support the comparison: *omission*, *generalization*, and *elimination*. This part of the approach can be used to reason of *multi-modal* user interfaces, by verifying the level of equivalence between them.

The approach is supported by CADP (Garavel et al. 2013), a toolbox for verifying asynchronous concurrent systems: systems whose components may operate at different speeds, without a global clock to synchronize them. Asynchronous systems suit the modelling of human–computer interactions well: the modules that describe the users, the functional core, and the user interfaces can evolve in time at different speeds, which reflects well the unordered sequence of events that take place in human–machine interactions. Both parts of the approach can be used either independently or in an integrated way, and it has been validated in three industrial case studies in the nuclear power plant domain, which indicates the potential of the approach with respect to scalability (Oliveira 2015).

1.6.10 Knight et al. (USA 1992–2010)

1.6.10.1 Modelling

Another example of the application of formal methods to safety-critical systems, specifically, to the nuclear power plant domain, can be found in Knight and Brilliant (1997). The authors propose the modelling of user interfaces in three levels: *lexical*, *syntactic*, and *semantic* levels. Different formalisms are used to describe each level. For instance, the lexical level is defined using Borland's OWL (*Object Windows Library)*, allowing *data* and *events* to be represented. The syntactic level in the approach is documented with a set of context-free grammars with one grammar for each of the concurrent, asynchronous dialogues that might be taking place. Such syntactic level imposes the required temporal ordering on user actions and system responses (Knight and Brilliant 1997). Finally, Z is used to define the semantic level. The notion of user interfaces as a dialogue between the operator and the computer system consisting of three components (lexical, syntactic, and semantic levels) is proposed by Foley and Wallace (1974).

Each of these three levels is specified separately. Since different notations are used, the communication between these levels is defined by a set of tokens (Knight and Brilliant 1997). The concept of a multi-party grammar is appropriate for representing grammars in which tokens are generated by more than one source (Knight and Brilliant 1997). Such representation could allow multi-modality to be covered by the approach. However, the authors have elected to use a conventional context-free grammar representation together with a naming convention to distinguish sources of tokens (Knight and Brilliant 1997).

Following this view of user interface structure, the authors develop a formal specification of a research reactor used in the University of Virginia Reactor (UVAR) for training nuclear engineering students, radiation damage studies, and other studies (Loer and Harrison 2000). In order to illustrate the specification layers, the authors focus on the safety control rod system, one of the reactor subsystems. They give in the paper the three specifications for this subsystem.

The approach is also applied to other safety-critical systems, such as the Magnetic Stereotaxis System (MSS), a healthcare application for performing human neurosurgery (Elder and Knight 1995; Knight and Kienzle 1992). UIs, users, and the functional core of systems are covered by this approach. The UI syntactic level in their approach defines valid sequences of user inputs on the UIs, which is to some extent the modelling of the users, and the cypher system case study described in Yin et al. (2008) verifies the correctness of the functional core. Finally, the approach covers the representation of *dynamic reconfiguration* (Knight and Brilliant 1997).

1.6.10.2 Verification

However, the formal specification is not used to perform formal verification. According to the authors, the main goal is to develop a formal specification approach for user interfaces of safety-critical systems. Concerning verifiability, the authors claim that the verification of a UI specification using this approach is simplified by the use of an executable specification for the lexical level and by the use of a notation from which an implementation can be synthesized for the syntactic level. For the semantic level, they argue that all the tools and techniques developed for Z can be applied (Knight and Brilliant 1997).

Later, a toolbox called Zeus is proposed to support the Z notation (Knight et al. 1999). The tool permits the creation and analysis of Z documents, including syntax and type checking, schema expansion, precondition calculation, domain checking, and general theorem proving. The tool is evaluated in a development of a relatively large specification of an international maritime software standard, showing that Zeus meets the expected requirements (Knight et al. 1999).

Following such a separation of concerns in three levels, the authors propose another approach called Echo (Strunk et al. 2005), and this time applied to a case study in the avionics domain. In order to decrease complexity with traditional correctness proofs, the Echo approach is based on the refactoring of the formal specification (Yin et al. 2009a, b), reducing the verification burden by distributing it over separate tools and techniques. The system model to be verified (written in PVS) is mechanically refactored. It is refined into an implementable specification in Spark Ada by removing any un-implementable semantics. After refactoring, the model is documented with low-level annotations, and a specification in PVS is extracted mechanically (Yin et al. 2008). Proofs that the semantics of the refactored model is equivalent to that of the original system model, that the code conforms to the annotations, and that the extracted specification implies the original system model constitute the verification argument (Yin et al. 2009a).

An extension of the approach is proposed in Yin and Knight (2010), aiming at facilitating formal verification of large software systems by a technique called proof by parts, which improve the scalability of the approach for larger case studies.

The authors did not clearly define the kinds of properties they can verify over interactive systems with their approach. The case studies to which the approach is applied mainly focused on the benefits of modelling UIs in three layers using formal notation.

1.6.11 Miller et al. (USA 1995–2013)

1.6.11.1 Modelling

Also in the safety-critical domain, but in avionics, deep investigation has been conducted at Rockwell Collins of the usage of formal methods for industrial realistic case studies. Preliminary usage of formal methods aimed at creating consistent and verifiable system specifications (Hamilton et al. 1995), paving the way to the usage of formal methods at Rockwell Collins. Another preliminary use of formal methods was the usage of a synchronous language called RSML (Requirements State Machine Language) to specify requirements of a flight guidance system. RSML is a *state-based* specification language developed by Leveson's group at the University of California at Irvine as a language for specifying the behaviour of process control systems (Miller et al. 2006). Algorithms to translate specifications from this language to the input languages of the NuSMV model checker and the PVS theorem prover have been proposed (Miller et al. 2006), enabling one to perform verification of safety properties and functional requirements expressed in the CTL temporal logic (i.e. functional properties). Afterwards, deeper investigations are conducted to further facilitate the usage of formal methods.

According to Miller (2009), relatively few case studies of model checking to industrial problems outside the field of engineering equipment are reported. One of the reasons is the gap between the descriptive notations most widely used by software developers and the notations required by formal methods (Lutz 2000). To alleviate the difficulties, as part of NASA's Aviation Safety Program (AvSP), Rockwell Collins and the research group on critical systems of the University of Minnesota (USA) develop the Rockwell Collins Gryphon Translator Framework (Hardin et al. 2009), providing a bridge between some commercial modelling languages and various model checkers and theorem provers (Miller et al. 2010). The translation framework supports Simulink, Stateflow, and SCADE models, and it generates specifications for the NuSMV, Prover, and SAL model checkers, the ACL2 and PVS theorem provers, and generates C and Ada code (Miller et al. 2010) (BAT and Kind are also included as target model checkers in Cofer et al. 2012). Alternatively, Z specifications are also covered by the approach as an input language, since Simulink and Stateflow models can be derived from Z specifications (Hardin et al. 2009).

Algorithms to deal with the time dependencies were implemented in the translator, allowing multiple input events arriving at the same time to be handled (Miller et al. 2006). Concerning the modelling coverage, the approach covers only the functional core of the avionics interactive systems that were analysed (Combéfis 2013; Miller 2009; Miller et al. 2010), but not the user interfaces nor the user behaviour.

Tools were also developed to translate the counterexamples produced by the model checkers back to Simulink and Stateflow models (Cofer 2012), since for large systems it can be difficult to determine the cause of the violation of the property only by examining counterexamples (Whalen et al. 2008).

1.6.11.2 Verification

The technique is applied to several case studies in avionics (Cofer 2012; Combéfis 2013; Miller 2009; Miller et al. 2010; Whalen et al. 2008). The first application of the NuSMV model checker to an actual product at Rockwell Collins is the mode logic of the FCS 5000 Flight Control System (Miller 2009): 26 errors are found in the mode logic.

The largest and most successful application is the Rockwell Collins ADGS-2100 (Adaptive Display and Guidance System Window Manager), a cockpit system that provides displays and display management software for commercial aircraft (Miller et al. 2010). The Window Manager (WM) ensures that data from different applications are displayed correctly on the display panel. A set of properties that formally expresses the WM requirements (i.e. functional properties) is developed in the CTL and LTL temporal logic: 563 properties are developed and verified, and 98 design errors are found and corrected.

The approach is also applied to an adaptive flight control system prototype for unmanned aircraft modelled in Simulink (Cofer 2012; Whalen et al. 2008). During the analysis, over 60 functional properties are verified, and 10 model errors and 2 requirement errors are found in relatively mature models.

These applications to the avionics domain demonstrate that the approach scales well. Even if the approach does not take user interfaces into account, it is a good example of formal methods applied to safety-critical systems. In addition, further investigations of the usage of compositional verification are conducted (Cofer et al. 2008; Murugesan et al. 2013), to enhance the proposed techniques.

1.6.12 Loer and Harrison et al. (Germany 2000–2006)

1.6.12.1 Modelling

Another approach to verifying interactive systems is proposed in Loer and Harrison (2002, 2006), also with the goal of making model checking more accessible to software engineers. The authors claim that in the avionics and automotive domains, requirements are often expressed as statechart models (Loer and Harrison 2002). With statecharts, a complex system can be specified as a number of potentially hierarchical state machines that describe functional or physical subsystems and run in parallel (Loer and Harrison 2000). Such parallelism could represent *concurrent behaviour*. The ordering of events which change the machine from one state to another can be used to represent *qualitative time*. Furthermore, in the statechart semantics, *time* is represented by a number of execution steps, allowing to express the formulation “within n steps from the current state…” (Loer and Harrison 2000).

To introduce formal verification in the process, they propose an automatic translation from statechart models (created with the Statemate toolkit) to the input language of the SMV model checker, which is relatively robust and well supported

(Loer and Harrison 2006). Such translation is part of the IFADIS toolbox, which also provides guided process of property specifications and a trace visualization to facilitate the result analysis of the model checker.

Concerning the modelling coverage of the approach, the authors describe five pre-defined elements in which the formal model is structured (Loer and Harrison 2000):

- **Control elements**: Description of the widgets of the UIs;
- **Control mechanism**: Description of the system functionality;
- **Displays**: Description of the output elements;
- **Environment**: Description of relevant environmental properties;
- **User tasks**: Sequence of user actions that are required to accomplish a certain task.

Therefore, their model covers the three aspects we are analysing: the user, UIs, and the functional core. However, aspects such as *multi-modality* and *multi-touch* interactions are not covered.

1.6.12.2 Verification

The properties can be verified using Cadence SMV or NuSMV model-checking tools. Depending on the type of property, the model checker can output traces that demonstrate why a property holds or not (Loer and Harrison 2006).

The property editor helps designers to construct temporal-logic properties by making patterns available and helping the process of instantiation (Loer and Harrison 2006). Temporal-logic properties can be specified either in LTL (linear temporal logic) or in CTL (computational tree logic). The following usability properties can be verified:

- **Reachability (Loer and Harrison** 2000**)**: Are all the states reachable or not?
- **Robustness (Loer and Harrison** 2000**)**: Does the system provide fallback alternatives in the case of a failure? or, alternatively, are the guards for unsafe states foolproof?
- **Recoverability (Loer and Harrison** 2000**)**: Does the system support undo and redo?
- **Visibility of system status (Loer and Harrison** 2000**)**: Does the system always keep the users informed about what is going on, through appropriate feedback within reasonable time?
- **Recognition rather than recall (Loer and Harrison** 2000**)**: Is the user forced to remember information from one part of the dialogue to another?
- **Behavioural consistency (Loer and Harrison** 2006**)**: Does the same input always yield the same effect?

In particular, the reachability property here is classified as a usability property because it is defined as generic property, which can be applied to any interactive system (i.e. “are all the states reachable or not?”). This is in contrast to the classification of the reachability property, for instance, where it is classified as a functional property because it expresses what can be done at the UI, and how can it be done, which is something that is usually defined in the system requirements.

Although the approach is not applied to many case studies (i.e. only to the avionics domain Loer and Harrison 2006), several reasons indicate that the approach scales well to real-life applications. The approach is supported by a tool that provides a translation from engineering models (statecharts) to formal models (SMV specifications), a set of property patterns to facilitate the specification of properties, and a trace visualizer to interpret the counterexamples generated by the model checker. It is used in the case study described in Loer and Harrison (2006), and an evaluation shows that the tool improves the usability of model checking for non-experts (Loer and Harrison 2006).

1.6.13 Thimbleby et al. (United Kingdom 1987–2015)

1.6.13.1 Modelling

In the healthcare domain, several investigations of medical device user interfaces have been conducted at Swansea University and Queen Mary University of London. Specifically, investigations are conducted on interactive hospital beds (Acharya et al. 2010), for user interfaces of drug infusion pumps (Cauchi et al. 2012a; Masci et al. 2014a, 2015; Thimbleby and Gow 2008), and interaction issues that can lead to serious clinical consequences.

Infusion pumps are medical devices used to deliver drugs to patients. Deep investigation has been done of the data entry systems of such devices (Cauchi et al. 2012b, 2014; Gimblett and Thimbleby 2013; Li et al. 2015; Masci et al. 2011; Oladimeji et al. 2011, 2013; Thimbleby 2010; Thimbleby and Gimblett 2011; Tu et al. 2014). If a nurse makes an error in setting up an infusion (for instance, a number ten times larger than the necessary for the patient’s therapy), the patient may die. Under-dosing is also a problem: if a patient receives too little of a drug, recovery may be delayed or the patient may suffer unnecessary pain (Masci et al. 2011).

The authors report several issues with the data entry system of such pumps (Masci et al. 2014a). Several issues are detected (Masci et al. 2014a) using the approach depicted in Fig. 1.7. In this approach, the C++ source code of the infusion pump is manually translated into a specification in the PVS formal language ([a] in Fig. 1.7).

Concerning the modelling coverage, the approach deals with the display and functionality of the devices, but does not cover the modelling of the users

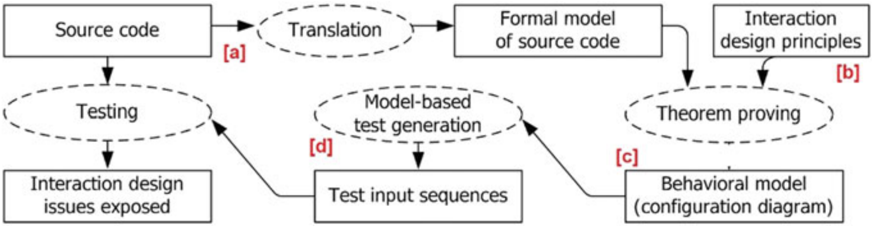

Source code
[a]
Translation
Formal model of source code
Interaction design principles
[b]
Testing
Model-based test generation
[d]
Theorem proving
[c]
Interaction design issues exposed
Test input sequences
Behavioral model (configuration diagram)

Fig. 1.7 Verification approach using PVS, adapted from Masci et al. (2014a)

interacting with such devices. In addition, no references were found describing the modelling of *concurrent behaviour, multi-modality,* nor *multi-touch interactions.*

1.6.13.2 Verification

Usability properties such as consistency of actions and feedback are formalized ([b] in Fig. 1.7) as invariants to be established using theorem proving:

- **Consistency of actions**: The same user actions should produce the same results in logically equivalent situations;
- **Feedback**: It ensures that the user is provided with sufficient information on what actions have been done and what result has been achieved.

A behavioural model is then extracted ([c] in Fig. 1.7), in a mechanized manner, from the PVS formal specification. This model captures the control structure and behaviour of the software related to the handling user interactions. Theorem proving is used to verify that the behavioural model satisfies the usability properties. Lastly, the behavioural model is exhaustively explored to generate a suite of test sequences ([d] in Fig. 1.7) (Masci et al. 2014a).

A similar approach is described in Masci et al. (2013a), in which the PVS specification is automatically discovered (Gimblett and Thimbleby 2010; Thimbleby 2007a) from reversely engineering the infusion pump software. Besides, functional properties are extracted from the safety requirements provided by the US medical device regulator FDA (Food and Drug Administration), to make sure that the medical device is reasonably safe before entering the market (Masci et al. 2013a). This approach allows *quantitative time* to be modelled and property such as “The pump shall issue an alert if paused for more than t minutes” to be verified (Masci et al. 2013a).

The same FDA safety requirements are used to verify a PVS formal model of another device, the *Generic Patient Controlled Analgesia* (GPCA) infusion pump (Masci et al. 2013a). In this work, the authors propose the usage of formal methods for rapid prototyping of user interfaces. Once verified, the formal model of the infusion pump is automatically translated into executable code through the PVS code generator, providing a prototype of the GPCA user interface from a verified model of the infusion pump.

An approach to integrating PVS executable specifications and Stateflow models is proposed in Masci et al. (2014b), aiming at reducing the barriers that prevent non-experts from using formal methods. It permits Stateflow models to be verified, avoiding the hazards of translating design models created in different tools.

All the work mentioned in this subsection is based on the PVS theorem prover. Nevertheless, model checking can also be used in order to formally verify medical devices (Masci et al. 2011, 2015; Thimbleby 2007b). For example, the authors model the Alaris GP in Masci et al. (2015), and the B-Braun Infusomat Space infusion pumps in the higher-order logic specification language SAL (Symbolic Analysis Laboratory) (De Moura et al. 2004). Afterwards, model checking is applied to verify the predictability of user interfaces, a usability property expressed in the LTL temporal logic. Predictability is defined in Masci et al. (2011) as "if users look at the device and see that it is in a particular display state, then they can predict the next display state of the device after a user interaction".

The maturity of the approach described here and its applications to numerous case studies are evidence that the approach scales well to real-life applications.

1.6.14 Palanque et al. (France 1990–2015)

1.6.14.1 Modelling

In Palanque and Bastide (1995), another approach is proposed to modelling and verifying interactive systems with a different formalism: Petri nets (Petri 1962). Being a graphical model, Petri nets might be easier to understand than textual descriptions.

Originally, the work was targeting at modelling, implementation, and simulation of the dialogue part of event-driven interfaces (Bastide and Palanque 1990); it nowadays covers the modelling of the entire interactive system. Early notation was called Petri nets with Objects (Bastide and Palanque 1990) (which belongs to the high-level Petri nets class) and was an extension of Petri nets in order to manipulate tokens which were references to objects (in the meaning the object-oriented paradigm). This has been further extended over the years to the ICO (Interactive Cooperative Object) formalism (Navarre et al. 2009) which permits applications to be prototyped and tested but also can be fully implemented by integrating Java code in the models.

A system described using ICOs is modelled as a set of objects that cooperate to perform the system tasks. ICO uses concepts borrowed from the object-oriented formalism (such as inheritance, polymorphism, encapsulation, and dynamic instantiation) to describe the structural or static aspects of systems, such as its attributes and the operations it provides to its environment.

Through the Petri nets with Objects formalism, *data, state,* and *events*, as well as *concurrent behaviour,* can be modelled. *Dynamic reconfiguration* of user interfaces are proposed in Navarre et al. (2008), allowing operators to continue interacting with the interactive system even though part of the hardware side of the user interface is failing. Besides, the formalism also allows the modelling of low-level behaviour of interaction techniques including *multi-touch interactions* (Hamon et al. 2013). In the multi-touch context, *new fingers* are detected at execution time. Thus, the description language must be able to receive dynamically created objects. In Petri nets, this can be represented by the creation/destruction of tokens associated with the objects (Hamon et al. 2013). The notation has been used to model WIMP interfaces (Bastide and Palanque 1990) and post-WIMP ones including multi-modal interaction with speech (Bastide et al. 2004), virtual reality (Navarre et al. 2005), or bimanual interactions (Navarre et al. 2009).

Once the system is modelled using the ICO formalism, it is possible to apply model checking to verify usability and functional properties. How modelling, verification, and execution are performed using Petshop (for Petri net Workshop) (Bastide et al. 2002, 2004) (the CASE tool supporting the ICO formal description technique) is illustrated in Chap. 20 of this book.

This work allows the integration of two different representations: tasks models and interactive systems models as described in Palanque et al. (1996). This integration was first done by describing task models constructs as Petri net structures. This was first done using UAN notation (Hix and Hartson 1993) as demonstrated in Palanque et al. (1996), allowing for the verification of compatibility between the two representations using Petri net-based bisimulation (Bourguet-Rouger 1988). Then, connection with CTT notation (Paternó et al. 1997) was made using scenarios as a connection artefact (Navarre et al. 2001). Such work has been extended by developing integration of models at the syntactic level allowing for co-simulation of models (Barboni et al. 2010) and more recently using the HAMSTERS notation that allows structuring of models (Martinie et al. 2011a) and enables explicit representation of data, errors, and knowledge in task models (Fahssi et al. 2015). This work focusses on the use of multiple models jointly as presented in Martinie et al. (2014).

1.6.14.2 Verification

For instance, the following usability properties can be verified (Palanque and Bastide 1995):

- **Predictability**: The user is able to foresee the effects of a command.
- **Deadlock freedom**: The impossibility for a user to get into a state where no actions can be taken.
- **Reinitiability**: This is the ability for the user to reach the initial state of the system.

- **Exclusion of commands**: This means the commands which must never be offered at the same time (or, on the contrary, must always be offered simultaneously).
- **Succession of commands**: The proper order in which commands may be issued; for instance, a given command must or must not be followed by another one, immediately after or with some other commands in between.
- **Availability**: A command is offered all the time, regardless of the state of the system (e.g. a help command).

The specification is verified using Petri net property analysis tools (Palanque et al. 1996). In order to automate the process of property verification, the ACTL can be used to express the properties, which are then proved by model checking the Petri net marking graph (Palanque et al. 1999).

The ICO approach also permits user's cognitive behaviour to be modelled by a common Petri net for system, device, and user (Moher et al. 1996). As previously mentioned, Petshop supports the design of interactive systems according to the ICO methodology. Alternatively, the Java PathFinder model checker is used to verify a set of properties on a safety-critical application in the interactive cockpit systems modelled with ICO (Boyer and Moore 1983).

The approach has been applied to case studies and real applications in safety-critical systems not only in the space domain (see, for instance, Bastide et al. 2003, 2004; Boyer and Moore 1983; Palanque et al. 1997) but also in the air traffic management and more recently aircraft cockpits. These case studies and the maturity of the tools show that the approach scales well to real-life applications.

However, the verification based on the Petri net properties has limitations exposed in Navarre et al. (2009). The analysis is usually performed on the underlying Petri net (a simplified version of the original Petri net). A drawback is that properties verified on the underlying Petri net are not necessarily true on the original Petri net. Thus, the results of the analysis are essentially indicators of potential problems in the original Petri net. This is due to the fact that the team involved in the ICO notation has not extended work for properties verifications in Petri nets to encompass extensions (e.g. the use of preconditions).

1.6.15 Aït-Ameur et al. (France 1998–2014)

1.6.15.1 Modelling

An alternative approach is proposed in Aït-Ameur et al. (1998b, 1999, 2003a), this time relying on theorem proving using the B method (Abrial 1996) to specify the interactive system. The approach permits task models to be validated. Task models can be used to describe a system in terms of tasks, subtasks, and their temporal relationships. A task has an initial state and a final state and is decomposed in a sequence of several subtasks.

The approach uses the B method for representing, verifying, and refining specifications. The authors use the set of events to define a transition system that permits the dialogue controller of the interactive system to be represented. The CTT (Concur Task Tree) notation (Paternó et al. 1997) is used to represent task models. In Aït-Ameur et al. (2003a), only the CTT operator called "sequence" between tasks is covered. In further work (Aït-Ameur et al. 2005a, b, 2009), the authors describe how every CTT construction can be formally described in Event B (including the *concurrency* operator) allowing to translate, with generic translation rules, every CTT construction in Event B, which is the event-based definition of B method.

The case study described in Aït-Ameur et al. (1998a) shows that the approach covers the modelling of users, UIs, and the functional core. In this approach, *qualitative time* can be modelled using the PRE THEN substitution, which allows one to order operations (Aït-Ameur et al. 1998b). *Multi-touch interactions* are also covered by modelling each finger as a machine, and by integrating these machines (using the EXTENDS clause) in another machine that would represent the whole hand used for interaction (Aït-Ameur et al. 1998b). However, there is no possibility to account for quantitative time which is needed for the description of fine grain interaction in multi-modal interactions.

1.6.15.2 Verification

This usage of Event B to encode CTT task models is described in several case studies (Aït-Ameur et al. 2006; Aït-Ameur and Baron 2004, 2006; Cortier et al. 2007). In particular, the approach is used to verify Java/Swing user interfaces (Cortier et al. 2007), from which Event B models are obtained. Such Event B models encapsulate the UI behaviour of the application. Validation is achieved with respect to a task model that can be viewed as a specification. The task model is encoded in Event B, and assertions ensure that suitable interaction scenario is accepted by the CTT task model. Demonstrating that the Event B formal model behaves as intended comes to demonstrate that it is a correct refinement of the CTT task model.

Moreover, the following usability properties can be verified:

- **Robustness (Aït-Ameur et al.** 2003b): These properties are related to system dependability.
- **Visibility (Aït-Ameur et al.** 1999): It relates to feedback and information delivered to the user.
- **Reachability (Aït-Ameur et al.** 1999): These properties express what can be done at the user interface and how can it be done.
- **Reliability (Aït-Ameur et al.** 1999): It concerns the way the interface works with the underlying system.
- **Behavioural properties (Aït-Ameur and Baron** 2006): They characterize the behaviour of the UI suited by the user.

The proof of these properties is done using the invariant and assertion clauses of the B method, together with the validation of specific aspects of the task model (i.e. functional properties), thus permitting a full system task model to be validated. The Atelier B tool is used for an automatic proof obligation generation and proof obligation checking (Aït-Ameur 2000).

In order to compare this theorem proving-based approach to model-checking-based approaches, the authors show how the same case study is tackled using both theorem proving (with Event B) and model checking (with Promela/SPIN) (Aït-Ameur et al. 2003b). The authors conclude that both techniques permit the case study to be fully described and that both permit robustness and reachability properties to be verified. The proof process of the Event B-based approach is not fully automatic, but it does not suffer from the state-space explosion of model-checking techniques. The Promela-SPIN-based technique is fully automatic, but limited to finite-state systems on which exhaustive exploration can be performed. The authors conclude that a combined usage of both techniques would strengthen the verification of interactive systems.

An integration of the approach with testing is also presented in Aït-Ameur et al. (2004). Here, the informal requirements are expressed using the semi-formal notation UAN (Hix and Hartson 1993) (instead of CTT), and the B specifications are manually derived from this notation. To validate the formal specification, the authors use a data-oriented modelling language, named EXPRESS, to represent validation scenarios. The B specifications are translated into EXPRESS code (the B2EXPRESS tool Aït-Sadoune and Aït-Ameur 2008). This translation gives data models that represent specification tests and permits Event B models to be animated.

The approach is applied to several case studies in the avionics domain (Aït-Ameur et al. 2014; Jambon et al. 2001). Specifically, the authors illustrate how to explicitly introduce the context of the systems in the formal modelling (Aït-Ameur et al. 2014). The approach is also applied to the design and validation of *multimodal* interactive systems (Aït-Ameur et al. 2006, 2010a; Aït-Ameur and Kamel 2004). The numerous case studies and the maturity of the approach suggest that it might scale to real-life applications even though no evidence was given in the papers.

1.6.16 Bowen and Reeves (New Zealand 2005–2015)

1.6.16.1 Modelling

The main focus of the approach proposed by Bowen and Reeves is the use of lightweight models of interfaces and interactions in conjunction with formal models of systems in order to bridge the gap between the typical informal UI design process and formal methods (Bowen and Reeves 2007a). UIs are formally described using a presentation model (PM) and a presentation and interaction model (PIM), while the underlying system behaviour is specified using the Z language (ISO 2002). The lightweight models allow UI and interaction designers to work with their typical artefacts (prototypes, storyboards, etc.), and a relation (PMR) is created between the behaviours described in the UI models and the formal specification in Z which gives a formal meaning to (and therefore the semantics of) the presentation models. The formal semantics also then enables a description of refinement to be given which can guide the transformation from model to implementation. Again, this is given at both formal and informal levels so can be used within both design and formal processes (Bowen and Reeves 2006).

While the primary use of the models is during the design process (assuming a specification-first approach), it is also possible to reverse-engineer existing systems into the set of models. Most examples given of this rely on a manual approach, which can be error-prone or lead to incomplete models. Some work has been done on supporting an automated approach. Using a combination of dynamic and static analysis of code and Java implementations has been investigated, and this allows UI widgets and some behaviours to be identified which can be used to partially construct models and support their completion (Bowen 2015).

The presentation model uses a simple syntax of tuples of labels, and the semantics of the model is based on set theory. It is used to formally capture the meaning of an informal design artefact such as a scenario, a storyboard, or a UI prototype by describing all possible behaviours of the windows, dialogues, or modes of the interactive system (Bowen and Reeves 2007a). In order to extend this to represent dynamic UI behaviour, a PIM (presentation and interaction model) is used. This is essentially a finite-state machine where each state represents one window, dialogue, or mode (i.e. individual presentation model) of the system with the transitions representing navigational behaviours. The PIM can be visualized using the μCharts language (Reeve and Reeves 2000) which has its own Z semantics (and which is based on statecharts) and which therefore enables a complete model of all parts of the system to be created and ultimately translated into Z.

Although the approach mainly focuses on modelling the user interfaces, the presentation model can also be used to model the user operations. The main goal is to ensure that all user operations described in the formal specification have been described in the UI design, and again, the PMR is used to support this. The models can also be used post-implementation to support testing. A set of abstract tests can be generated from the presentation models which describe the requirements of the UI widgets and behaviours, and these can then be concretized into testing frameworks such as JUnit and UISpec4 J (Bowen and Reeves 2013).

1.6.16.2 Verification

The approach has been applied to several case studies. In Bowen and Reeves (2007b), the authors use the models in the design process of UIs for the PIMed tool, a software editor for presentation models and PIMs. However, in this work, no automated verification of the presentation model and the PIMs of the editor is proposed. The formal models are manually inspected. For example, in order to verify the deadlock freedom property, the authors use a manual walk-through procedure in the PIMs. In later work (e.g. Bowen and Reeves 2012), the verification is automated by the ProZ tool, allowing both usability properties and functional properties to be verified using model checking. The kinds of usability properties that can be verified are:

- **Total reachability**: One can get to any state from any other state.
- **Deadlock freedom**: A user cannot get into a state where no action can be taken.
- **Behavioural consistency**: Controls with the same behaviour have the same name.
- **Minimum memory load on user**: Does not have to remember long sequences of actions to navigate through the UI.

Another case study to which these formal models have been applied relates to a safety-critical system in the healthcare domain (Bowen and Reeves 2012). Again, the verification is supported using ProZ. The authors model a syringe pump, a device commonly used to deliver pain-relief medication in hospitals and respite care homes. The device has ten widgets, which include the display screen, eight soft keys, and an audible alarm (*multi-modality)*. Temporal safety properties and invariants (to check boundary values) are verified against the formal models using ProZ and LTL.

Typically, the generated PIMs are relatively small. This is because the number of states and transitions are bounded by the number of windows/dialogues/modes of the system rather than individual behaviours as seen in other uses of finite-state machines. This abstraction of presentation models into states of PIMs prevents state explosion and enables a combination of both manual and automatic verification (as appropriate) with reasonably low overhead. The presentation models and system specification are created manually as part of the development and specification process. While the creation of models of systems can be seen as an additional overhead to a development process, the benefits provided by both the creation and the use of the models more than compensates for this later in the process. Once the presentation models and PMR have been created, the PIM can be automatically generated and translation between the models, μCharts, and Z is automated using several tools. The individual models can be used independently or in combination to verify different aspects of the system under consideration. Most recently, this work has focussed on safety-critical systems, and an example of this is given in Chap. 6.

1.6.17 Weyers et al. (Germany 2009–2015)

1.6.17.1 Modelling

In Weyers (2012) and Weyers et al. (2012a), a formal modelling approach has been proposed, which is based on a combination of the use of a domain-specific and visual modelling language called FILL with an accompanied transformation algorithm mapping FILL models onto a well-established formal description concept: Petri nets. In contrast to the works by Bastide and Palanque (1990) who extended basic coloured Petri nets to the ICO formalism, the modelling is done in a domain-specific description, which is not used directly for execution or verification. Instead, it is transformed into an existing formalism called reference net (Kummer 2002) (a special type of Petri net) providing a formal semantic definition for FILL, making FILL models executable using existing simulators (e.g. Renew Kummer et al. 2000) and offering the possibility for the application of verification and validation techniques for Petri net-based formalisms. As a modelling approach, FILL has been used in research on collaborative learning systems in which students created an interface for simulating a cryptographic algorithm (Weyers et al. 2009, 2010b).

As the work focuses on the model-based creation of interactive systems and less on the creation of formal models used for their verification as done in various related works, Weyers (2015) extended the basic approach with concepts from software development. The main extension, which is described in Chap. 5 of this book, addresses a component-based modelling as it offers reusability of certain components and capabilities to structure the overall model by means of functional and conceptual entities. The latter enables the modeller to create more complex (scalable) models and to be able to split the model into semantically meaningful parts. It further offers the description of complex user interfaces, which are not restricted to basic graphical user interfaces but include multi-user interfaces as well as mobile and other interaction devices. This is covered by a software infrastructure that offers capabilities to run mobile devices with models generated using FILL and its associated transferred reference nets. All is embedded into a coherent software tool called UIEditor (Weyers 2012).

An application context in which the component-based model description plays a central role is that of gaze guiding as a job aid for the control of technical processes (Kluge et al. 2014; Weyers et al. 2015). Gaze guiding as a method refers to a technique for visualizing context-dependent visual aids in the form of gaze guiding tools into graphical user interfaces that guide the user's attention and support her or him during execution of a control task. Gaze guiding tools are visual artefacts that are added to a graphical user interface in the case the user is expected to apply a certain operation (according to a previously defined standard operating procedure), but the system does not recognize this awaited input. This work facilitates the component-based description as the model describing the behaviour of the gaze guiding is embedded as a component into the user interface description. The model specifies in which cases or context gaze guiding tools are faded into the user interface.

1.6.17.2 Model Reconfiguration and Formal Rewriting

As the work by Weyers et al. does not focus on the verification of interactive systems but on the modelling and creation of flexible and executable model-based descriptions of such systems, a formal approach for the reconfiguration and rewriting of these models has been developed. This enables the specification of formal adaptation of the models according to user input or algorithmic specifications and makes the models flexible. In this regard, the main goal is to maintain the formal integrity of a model during an adaptation. Formal integrity refers to the requirement that an adaptation approach needs to be formally well defined as well and keeps the degree of formality on the same level with that of the model description. This should prevent any gaps in the formalization during adaptation of a model and thus prevent the compromise of any following verification, testing, or debugging of the rewritten model. Therefore, Weyers et al. (2010a, 2014) developed a reconfiguration concept based on pushouts, a concept known from category theory for the rewriting of reference nets. Together with Stückrath, Weyers extended a basic approach for the rewriting of Petri nets based on the so-called double-pushout approach to a method for coloured Petri nets equipped with a rewriting of XML-based specification of inscriptions (Stückrath and Weyers 2014).

The application of formal rewriting that is driven by the user has been investigated in the context of the monitoring and control of complex technical and safety-critical systems (Burkolter et al. 2014; Weyers et al. 2012a, b). In these works, the reconfiguration of a given user interface for controlling a simplified nuclear power plant was reconfigured by the user according to his or her own needs as well as to the standard operating procedures which were presented. These adaptations of the user interface include a change not only in the visual layout of widgets but also in the functional behaviour of the interface using the rewriting of the underlying reference net model. Operations were offered to the user, e.g., to generate a new widget which triggers a combination of two existing operations. For example, it was possible to combine the opening and closing of different valves into one new operation, which was accessible through a new single widget, e.g. a button. By pressing this button, the user was able to simultaneously open and close the dedicated valves with one click instead of two. Weyers et al. were able to show that this individualization of a user interface reduces errors in the accompanied control task. A detailed introduction into the rewriting concept is presented in Chap. 10 of this book.

1.6.18 Combéfis et al. (Belgium 2009–2013)

1.6.18.1 Modelling

In Combéfis (2013), a formal framework for reasoning over system and user models is proposed, and the user models can be extracted from user manuals. Furthermore, this work proposes the automatic generation of user models. Using his technique, “adequate” user models can be generated from a given initial user model. Adequate user models capture the knowledge that the user must have about the system, i.e. the knowledge needed to control the system, using all its functionality and avoiding surprises. This generated user model can be used, for instance, to improve training manuals and courses (Combefis and Pecheur 2009).

In order to compare the system and the user model and to verify whether the user model is adequate to the system model, both models should be provided. With this goal, in this approach system and user are modelled with enriched labelled transition systems called HMI LTS (Combéfis et al. 2011a). In HMI LTS, three kinds of actions are defined (Combéfis et al. 2011a):

- **Commands**: Actions triggered by the user on the system;
- **Observations**: Actions triggered by the system, but that the user can observe; and
- **Internal actions**: Actions that are neither controlled nor observed by the user.

To be considered “adequate”, user models are expected to follow two specific properties: full control and mode-preserving. Intuitively, a user model allows full control of a system if at any time, when using the system according to the user model (Combefis and Pecheur 2009): the commands that the user model allows are exactly those available on the system; and the user model allows at least all the observations that can be produced by the system (Combefis and Pecheur 2009). A user model is said to be mode-preserving according to a system, if and only if, for all possible executions of the system the users can perform with their user model, given the observation they make, the mode predicted by the user model is the same as the mode of the system (Combefis and Pecheur 2009). Model checking is used to verify both properties over the user model.

Concerning the approach's coverage regarding modelling, the users, the user interfaces, and the functional core are modelled and compared to each other, the user model being extracted from the user manual describing the system. However, there is no indication that the approach supports *concurrent behaviour, multi-modality,* or *multi-touch* interactions.

1.6.18.2 Verification

The verification goal of this approach is to verify whether the user model is adequate to the system model, rather than to verify properties over the system model. In order to automatically generate adequate user models, the authors propose a technique based on a derivative of weak bisimulation, in which equivalence checking is used (Milner 1980). This is called "minimization of a model modulo an equivalence relation". Intuitively, using equivalence checking, they generate a user model U_2 from the initial user model U_1; that is, U_2 is equivalent to U_1 with respect to specific equivalence relations introduced by the authors.

Two equivalence relations are proposed: full-control equivalence and mode-preserving equivalence. Full-control equivalence distinguishes commands and observations: two equivalent states must allow the same set of commands, but may permit different sets of observations. Minimization modulo this equivalence produces a minimal user model that permits full control of the system. A mode-preserving equivalence is then derived from the full-control equivalence, by adding an additional constraint that the modes of two equivalent states must be the same (Combefis and Pecheur 2009). Using these equivalence relations, the authors can generate mode-preserving fully controlled user models, which can then be used to design user interfaces and/or training manuals. Both properties (i.e. mode-preserving and full control) and their combination are interesting because they propose that different levels of equivalence can be shown between system models.

A tool named jpf-hmi has been implemented in Java and uses the Java Pathfinder model checker (Combéfis et al. 2011a), to analyse and generate user models. The tool produces an LTS corresponding to one minimal fully controlled mental model, or it reports that no such model exists by providing a problematic sequence from the system (Combéfis et al. 2011a).

The approach is applied to several examples that are relatively large (Combéfis et al. 2011b). In the healthcare domain, a machine that treats patients by administering X-ray or electron beams is analysed with the approach, which detects several issues in the system. In the avionics domain, the approach is applied to an autopilot system of a Boeing airplane (Combéfis 2013), and a potential-mode confusion is identified. These are evidence that the approach scales well to real-life applications.

1.6.19 Synthesis

This section presents a representative list of approaches to verifying interactive systems with respect to the specifications, i.e. general statements about the behaviour of the system, which are represented here as desired properties, and analysed afterwards using formal methods. The approaches diverge on the formalisms they use for the description of interactive systems and for the specification of properties.

Some authors use theorem proving to perform verification, which is a technique that can handle infinite-state systems. Even though a proof done by a theorem prover is ensured to be correct, it can quickly become a hard process (Bumbulis et al. 1995b): the process is not fully automated, and user guidance is needed regarding the proof strategy to follow. Simulation can also be used to assess the quality of interactive systems. Simulation provides an environment for training the staff before starting their daily activities (Martinie et al. 2011b). However, simulated environments are limited in terms of training, since it is impossible to drive operators into severe and stressful conditions even using a full-scale simulator (Niwa et al. 2001). Simulation explores a part of the system state space and can be used for disproving certain properties by showing examples of incorrect behaviours. To the contrary, formal techniques such as model checking and equivalence checking consider the entire state space and can thus prove or disprove properties for all possible behaviours (Garavel and Graf 2013).

The presented approaches allow either *usability* or *dependability* properties to be verified over the system models. We believe that in the case of safety-critical systems, the verification approach should cover both such properties, due to the ergonomic aspects covered by the former and the safety aspects covered by the latter. Some approaches cover the modelling of the users, the user interfaces, and the functional core.

1.6.20 Summary

A representative list of approaches for describing and assessing the quality of interactive systems has been presented in this chapter.

Different formalisms are used in the system modelling (and property modelling, when applied). Numerous case studies have shown that each formalism has its strengths. The criteria to choose one over another would be more related with the knowledge and experience of the designers in the formalisms. Different formal techniques are employed, such as model checking, equivalence checking, and theorem proving. Most of the works presented here are tool supported, even though some authors still use manual inspection of the models to perform verification.

Table 1.1 Modelling coverage of the approaches

		Agent language (Abowd et al.)	PPS-based (Abowd et al.)	Red-PIE (Dix et al.)	TLIM (Paterno et al.)	ADC (Markopoulos et al.)	York (Duke et al.)	MAL (Campos et al.)	Lustre-based (d'Ausbourg et al.)	IL-based (Bumbulis et al.)	LNT-based (Oliveira et al.)	ECHO (Knight et al.)	Gryphon (Miller et al.)	IFADIS (Loer et al.)	PVS-based (Thimbleby et al.)	ICO (Palanque et al.)	B-based (Aït-Ameur et al.)	PM, PIM (Bowen et al.)	HMI LTS (Combéfis et al.)	FILL (Weyers et al.)
Language	Underlying language used for modelling (most of the time extended)	Z	SMV	Math	CTT, Lotos	Lotos	Z	MAL, SMV, PVS	Lustre	IL, HOL	LNT	Z, PVS, Spark ADA	RSML, Lustre	SMV	PVS, SAL	Petri nets and OO	B, Event B, Express	Z, mu Charts, FSM	HMI LTS	Reference nets
	Language for describing properties	-	CTL	Math	ACTL	ACTL	First-order logic	CTL	Lustre	Hoare logic	MCL	-	CTL, LTL	CTL, LTL	LTL, invariants	ACTL CTL*	B	LTL, invariants	-	-
Coverage	users	■	🟧	□	■	□	■	■	■	□	■	■	□	■	□	■	■	■	■	□
	user interfaces	□	■	■	■	■	■	■	■	■	■	■	□	■	■	■	■	■	■	■
	functional core	■	■	■	■	□	■	■	■	□	■	■	■	■	■	■	■	■	■	🟧
Data Description		■	■	□	■	■	□	■	🟧	■	■	■	■	■	■	■	■	■	■	■
State Representation		■	■	🟧	□	□	■	■	■	□	□	■	■	■	■	■	■	■	□	■
Event Representation		■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■
Time	Qualitative between two consecutive model elements	■	■	□	■	■	□	■	■	□	■	■	■	■	□	■	■	□	■	□
	Quantitative between two consecutive model elements	□	□	□	□	□	□	□	■	□	□	□	□	■	□	■	□	□	□	■
	Quantitative over non consecutive elements	□	□	□	□	□	□	□	□	□	□	□	□	■	■	■	□	□	□	■
Concurrent Behavior		■	□	□	■	■	□	□	■	□	■	■	□	■	□	■	■	□	□	■
Dynamic Instantiation	Widgets	□	□	□	□	□	□	□	□	□	□	■	□	□	□	■	□	■	□	🟧
	Input devices	□	□	□	□	□	□	□	□	□	□	□	□	□	□	■	□	□	□	🟧
	Reconfiguration of Interaction technique	□	□	□	□	□	□	□	□	□	□	■	□	□	□	■	□	■	□	■
	Reconfiguration of low level events	□	□	□	□	□	□	□	□	□	□	□	□	□	□	■	□	□	□	□
Multimodality: fusion of several modalities		□	■	■	□	■	■	□	□	□	■	□	□	□	□	■	■	■	□	🟧
Dynamic finger clustering		□	□	□	□	□	□	□	□	□	□	□	□	□	□	■	□	□	□	□
Capability to deal with multi-touch interactions	Implicit	□	□	□	□	□	□	□	□	□	□	□	□	□	□	■	■	■	□	■
	Explicit	■	□	□	■	■	■	□	□	□	□	□	□	□	□	■	□	□	□	□

■ Yes (presented in papers) 🟧 Some (possible) □ No

Tables 1.1 and 1.2 below summarize these approaches: the former gives an overview of the modelling coverage of the approaches, and the latter an overview of their verification capabilities. It is important to note that those tables are only meant at summarizing what has been presented. The cells of Table 1.1 are coloured according to the information found in papers. An empty cell does not mean that this aspect cannot be addressed by the notation; it only means that no paper has made that information explicit. Indeed, the presentation in the sections usually contains more details than the summary table.

Table 1.2 Verification capabilities of the approaches

	Agent language (Abowd et al.)	PPS-based (Abowd et al.)	Red-PIE (Dix et al.)	TLIM (Paterno et al.)	ADC (Markopoulos et al.)	York (Duke et al.)	MAL (Campos et al.)	Lustre-based (d'Ausbourg et al.)	IL-based (Bumbulis et al.)	LNT-based (Oliveira et al.)	ECHO (Knight et al.)	Gryphon (Miller et al.)	IFADIS (Loer et al.)	PVS-based (Thimbleby et al.)	ICO (Palanque et al.)	B-based (Aït-Ameur et al.)	PM, PIM (Bowen et al.)	HMI LTS (Combéfis et al.)	FILL (Weyers et al.)
Analysis	□	▨	□	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨
Technique	-	model checking	-	model checking	model checking	theorem proving	model checking, theorem proving	model checking	theorem proving	model checking, equivalence checking	theorem proving	model checking, theorem proving	model checking	model checking, theorem proving	model checking theorem proving (invariants)	theorem proving	model checking	model checking, equivalence checking	model checking
Tool support	-	SMV, Action Simulator	-	CTTE, LITE, CADP	CADP	Z	i2smv, IVY, SMV, PVS	UIM/X, Centaur, Lesar	HOL system	CADP	Z, Zeus, Echo	NuSMV, PVS, Reactis, Gryphon	Statemate, IFADIS, Cadence SMV, NuSMV	PVS, SAL, Stateflow	PetShop, Java PathFinder	Atelier B, B2EXPRESS, Promela-SPIN	PIMed, ProZ, Z/EVES	jpf-hmi, Java Pathfinder	SAT Solver
Types of Properties	-	usability, functiona	usability	usability, functional	usability, functional	usability	usability, functional	usability, functional	functional	usability, functional	-	functional	usability	usability, functional	usability, functional	usability, functional	usability, functional	-	functional
Application	non-critical	non-critical	non-critical	avionics	non-critical	avionics	avionics, healthcare	avionics	non-critical	nuclear	avionics, nuclear, healthcare	avionics	avionics	healthcare	avionics, space, Air Traffic Managment	avionics	safety-critical	avionics, healthcare	ciritcal, non-critical
Scalability	□	□	□	▨	□	□	▨	▨	□	▨	▨	▨	▨	▨	▨	▨	□	▨	▨

▨ Yes (presented in pa| 🟧 Some (possible) □ Not demonstrated

References

Abowd GD (1991) Formal aspects of human-computer interaction. Dissertation, University of Oxford

Abowd GD, Dix AJ (1992) Giving undo attention. Interact Comput 4(3):317–342

Abowd GD, Wang H-M, Monk AF (1995) A formal technique for automated dialogue development. In: Proceedings of the 1st conference on designing interactive systems: processes, practices, methods, & techniques. ACM, pp 219–226

Abrial J-R (1996) The B-book: Assigning Programs to Meanings. Cambridge University Press, New York

Acharya C, Thimbleby HW, Oladimeji P (2010) Human computer interaction and medical devices. In: Proceedings of the 2010 British computer society conference on human-computer interaction, pp 168–176

Aït-Ameur Y, Girard P, Jambon F (1998a) A uniform approach for specification and design of interactive systems: the B method. In: Proceedings of the fifth international eurographics workshop on design, specification and verification of interactive systems, pp 51–67

Aït-Ameur Y, Girard P, Jambon F (1998b) A uniform approach for the specification and design of interactive systems: the B method. In: Eurographics workshop on design, specification, and verification of interactive systems, pp 333–352

Aït-Ameur Y, Girard P, Jambon F (1999) Using the B formal approach for incremental specification design of interactive systems. In: Engineering for human-computer interaction. Springer, pp 91–109

Aït-Ameur Y (2000) Cooperation of formal methods in an engineering based software development process. In: Proceedings of integrated formal methods, second international conference, pp 136–155

Aït-Ameur Y, Baron M, Girard P (2003a) Formal validation of HCI user tasks. In: Software engineering research and practice, pp 732–738

Aït-Ameur Y, Baron M, Kamel N (2003b) Utilisation de Techniques Formelles dans la Modelisation d'Interfaces Homme-machine. Une Experience Comparative entre B et Promela/SPIN. In: 6th International Symposium on Programming and Systems, pp 57–66

Aït-Ameur Y, Kamel N (2004) A generic formal specification of fusion of modalities in a multimodal HCI. In: Building the information society. Springer, pp 415–420

Aït-Ameur Y, Baron M (2004) Bridging the gap between formal and experimental validation approaches in HCI systems design: use of the event B proof based technique. In: International symposium on leveraging applications of formal methods, pp 74–80

Aït-Ameur Y, Breholee B, Girard P, Guittet L, Jambon F (2004) Formal verification and validation of interactive systems specifications. In: Human error, safety and systems development. Springer, pp 61–76

Aït-Ameur Y, Baron M, Kamel N (2005a) Encoding a process algebra using the event B method. Application to the validation of user interfaces. In: Proceedings of 2nd IEEE international symposium on leveraging applications of formal methods, pp 1–17

Aït-Ameur Y, Idir A-S, Mickael B (2005b) Modelisation et Validation formelles d'IHM: LOT 1 (LISI/ENSMA). Technical report. LISI/ENSMA

Aït-Ameur Y, Aït-Sadoune I, Mota J-M, Baron M (2006) Validation et Verification Formelles de Systemes Interactifs Multi-modaux Fondees sur la Preuve. In: Proceedings of the 18th International Conference of the Association Francophone d'Interaction Homme-Machine, pp 123–130

Aït-Ameur Y, Baron M (2006) Formal and experimental validation approaches in HCI systems design based on a shared event B model. Int J Softw Tools Technol Transf 8(6):547–563

Aït-Sadoune I, Aït-Ameur Y (2008) Animating event B models by formal data models. In: Proceedings of leveraging applications of formal methods, verification and validation, pp 37–55

Aït-Ameur Y, Baron M, Kamel N, Mota J-M (2009) Encoding a process algebra using the event B method. Int J Softw Tools Technol Transfer 11(3):239–253

Aït-Ameur Y, Boniol F, Wiels V (2010a) Toward a wider use of formal methods for aerospace systems design and verification. Int J Softw Tools Technol Transf 12(1):1–7

Aït-Ameur Y, Aït-Sadoune I, Baron M, Mota J-M (2010b) Verification et Validation Formelles de Systemes Interactifs Fondees sur la Preuve: Application aux Systemes Multi-Modaux. J Interact Pers Syst 1(1):1–30

Aït-Ameur Y, Gibson JP, Mery D (2014) On implicit and explicit semantics: integration issues in proof-based development of systems. In: Leveraging applications of formal methods, verification and validation. Specialized techniques and applications. Springer, pp 604–618

Barboni E, Ladry J-F, Navarre D, Palanque PA, Winckler M (2010) Beyond modelling: an integrated environment supporting co-execution of tasks and systems models. In: Proceedings of ACM EICS conference, pp 165–174

Basnyat S, Palanque PA, Schupp B, Wright P (2007) Formal socio-technical barrier modelling for safety-critical interactive systems design. Saf Sci 45(5):545–565

Bass L, Little R, Pellegrino R, Reed S, Seacord R, Sheppard S, Szezur MR (1991) The ARCH model: seeheim revisited. In: User interface developpers' workshop

Bastide R, Palanque PA (1990) Petri net objects for the design, validation and prototyping of user-driven interfaces. In: IFIP INTERACT 1990 conference, pp 625–631

Bastide R, Navarre D, Palanque PA (2002) A model-based tool for interactive prototyping of highly interactive applications. In: CHI extended abstracts, pp 516–517

Bastide R, Navarre D, Palanque PA (2003) A tool-supported design framework for safety critical interactive systems. Interact Comput 15(3):309–328

Bastide R, Navarre D, Palanque PA, Schyn A, Dragicevic P (2004) A model-based approach for real-time embedded multimodal systems in military aircrafts. In: Proceedings of the 6th international conference on multimodal interfaces, pp 243–250

Beck K (1999) Extreme programming explained: embrace change. Addison-Wesley

Booch G (2005) The unified modelling language user guide. Pearson Education, India

Bourguet-Rouger A (1988) External behaviour equivalence between two petri nets. In: Proceedings of concurrency. Lecture notes in computer science, vol 335. Springer, pp 237–256

Bowen J, Reeves S (2006) Formal refinement of informal GUI design artefacts. In: Proceedings of 17th Australian software engineering conference, pp 221–230

Bowen J, Reeves S (2007a) Formal models for informal GUI designs. Electr Notes Theor Comput Sci 183:57–72

Bowen J, Reeves S (2007b) using formal models to design user interfaces: a case study. In: Proceedings of the 21st British HCI group annual conference on HCI, pp 159–166

Bowen J, Reeves S (2012) Modelling user manuals of modal medical devices and learning from the experience. In: Proceedings of ACM SIGCHI symposium on engineering interactive computing systems, pp 121–130

Bowen J, Reeves S (2013) UI-design driven model-based testing. Innov Syst Softw Eng 9(3):201–215

Bowen J (2015) Creating models of interactive systems with the support of lightweight reverse-engineering tools. In: Proceedings of the 7th ACM SIGCHI symposium on engineering interactive computing systems, pp 110–119

Boyer RS, Moore JS (1983) Proof-checking, theorem proving, and program verification. Technical report, DTIC document

Brat G, Martinie C, Palanque P (2013) V&V of lexical, syntactic and semantic properties for interactive systems through model checking of formal description of dialog. In: Proceedings of the 15th international conference on human-computer interaction, pp 290–299

Bumbulis P, Alencar PSC, Cowan DD, Lucena CJP (1995a) Combining formal techniques and prototyping in user interface construction and verification. In: Proceedings of 2nd eurographics workshop on design, specification, verification of interactive systems, pp 7–9

Bumbulis P, Alencar PSC, Cowan DD, de Lucena CJP (1995b) A framework for machine-assisted user interface verification. In: Proceedings of the 4th international conference on algebraic methodology and software technology, pp 461–474

Burkolter D, Weyers B, Kluge A, Luther W (2014) Customization of user interfaces to reduce errors and enhance user acceptance. Appl Ergon 45(2):346–353

Campos JC, Harrison MD (1997) Formally verifying interactive systems: a review. In: Proceedings of design, specification and verification of interactive systems, pp 109–124

Campos JC (1999) Automated deduction and usability reasoning. Dissertatoin, University of York

Campos JC, Harrison MD (2001) Model checking interactor specifications. Autom Softw Eng 8 (3–4):275–310

Campos JC, Harrison MD (2007) Considering context and users in interactive systems analysis. In: Proceedings of the joint working conferences on engineering interactive systems, pp 193–209

Campos JC, Harrison MD (2008) Systematic analysis of control panel interfaces using formal tools. In: Interactive systems. Design, specification, and verification. Springer, pp 72–85

Campos JC, Harrison MD (2009) Interaction engineering using the IVY tool. In: Proceedings of the 1st ACM SIGCHI symposium on engineering interactive computing systems, pp 35–44

Campos JC, Harrison MD (2011) Modelling and analysing the interactive behaviour of an infusion pump. Electron Commun EASST 45

Cauchi A, Gimblett A, Thimbleby HW, Curzon P, Masci P (2012a) Safer “5-key” number entry user interfaces using differential formal analysis. In: Proceedings of the 26th annual BCS interaction specialist group conference on people and computers, pp 29–38

Cauchi A, Gimblett A, Thimbleby HW, Curzon P, Masci P (2012b) Safer “5-key” number entry user interfaces using differential formal analysis. In: Proceedings of the 26th annual BCS interaction specialist group conference on people and computers, pp 29–38

Cauchi A, Oladimeji P, Niezen G, Thimbleby HW (2014) Triangulating empirical and analytic techniques for improving number entry user interfaces. In: Proceedings of ACM SIGCHI symposium on engineering interactive computing systems, pp 243–252

Champelovier D, Clerc X, Garavel H, Guerte Y, Lang F, Serwe W, Smeding G (2010) Reference manual of the LOTOS NT to LOTOS translator (version 5.0). INRIA/VASY

Chen P (1976) The entity-relationship model—toward a unified view of data. ACM Trans Database Syst 1(1):9–36

Clarke EM, Emerson EA, Sistla AP (1986) Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans Program Lang Syst 8(2):244–263

Cofer D, Whalen M, Miller S (2008) Software model checking for avionics systems. In: Digital avionics systems conference, pp 1–8

Cofer D (2010) Model checking: cleared for take-off. In: Model checking software. Springer, pp 76–87

Cofer D (2012) Formal methods in the aerospace industry: follow the money. In: Proceedings of the 14th international conference on formal engineering methods: formal methods and software engineering. Springer, pp 2–3

Cofer D, Gacek A, Miller S, Whalen MW, LaValley B, Sha L (2012) Compositional verification of architectural models. In: Proceedings of the 4th international conference on NASA formal methods. Springer, pp 126–140

Combefis S, Pecheur C (2009) A bisimulation-based approach to the analysis of human-computer interaction. In: Proceedings of the 1st ACM SIGCHI symposium on engineering interactive computing systems, pp 101–110

Combéfis S, Giannakopoulou D, Pecheur C, Feary M (2011a) A formal framework for design and analysis of human-machine interaction. In: Proceedings of the IEEE international conference on systems, man and cybernetics, pp 1801–1808

Combéfis S, Giannakopoulou D, Pecheur C, Feary M (2011b) Learning system abstractions for human operators. In: Proceedings of the international workshop on machine learning technologies in software engineering, pp 3–10

Combéfis S (2013) A formal framework for the analysis of human-machine interactions. Dissertation, Universite catholique de Louvain

Cortier A, d’Ausbourg B, Aït-Ameur Y (2007) Formal validation of java/swing user interfaces with the event B method. In: Human-computer interaction. Interaction design and usability. Springer, pp 1062–1071

Coutaz J (1987) PAC, an object oriented model for dialogue design. In: Bullinger H-J, Shackel B (eds) Human computer interaction INTERACT’87, pp 431–436

Curzon P, Blandford A (2004) Formally justifying user-centred design rules: a case study on post-completion errors. Proc IFM 2004:461–480

d’Ausbourg B (1998) Using model checking for the automatic validation of user interface systems. In: Proceedings of the fifth international eurographics workshop on the design, specification and verification of interactive systems, pp 242–260

d’Ausbourg B, Seguin C, Durrieu G, Roche P (1998) Helping the automated validation process of user interfaces systems. In: Proceedings of the 20th international conference on software engineering, pp 219–228

d’Ausbourg B (2002) Synthetiser I’Intention d’un Pilote pour Definir de Nouveaux Équipements de Bord. In: Proceedings of the 14th French-speaking conference on human-computer Interaction, pp 145–152

De Moura L, Owre S, Rueß H, Rushby J, Shankar N, Sorea M, Tiwari A (2004) SAL 2. In: Proceeidngs of computer aided verification, pp 496–500

Degani A, Heymann M (2002) Formal verification of human-automation interaction. Hum Fact J Hum Fact Ergon Soc 44(1):28–43

Dey T (2011) A comparative analysis on modelling and implementing with MVC architecture. In: Proceedings of the international conference on web services computing, vol 1, pp 44–49

Dix AJ (1988) Abstract, generic models of interactive systems. In: Proceedings of fourth conference of the British computer society human-computer interaction specialist group. University of Manchester, pp 63–77

Dix AJ, Harrison MD, Cunciman R, Thimbleby HW (1987) Interaction models and the principled design of interactive systems. In: Proceedings of the 1st European software engineering conference, pp 118–126

Dix AJ (1991) Formal methods for interactive systems. Academic Press, London

Dix AJ (1995) Formal Methods. In: Monk A, Gilbert N (eds) Perspectives on HCI: diverse approaches. Academic Press, London, pp 9–43

Dix AJ (2012) Formal methods. In: Soegaard M, Dam R (eds) Encyclopedia of human-computer interaction

Doherty G, Campos JC, Harrison MD (1998) Representational reasoning and verification. Formal Aspects Comput 12:260–277

Duke DJ, Harrison MD (1993) Abstract interaction objects. Comput Graph Forum 12:25–36

Duke DJ, Harrison MD (1995) Event model of human-system interaction. Softw Eng J 10(1):3–12

Elder MC, Knight J (1995) Specifying user interfaces for safety-critical medical systems. In: Proceedings of the 2nd annual international symposium on medical robotics and computer assisted surgery, pp 148–155

Fahssi R, Martinie C, Palanque PA (2015) Enhanced task modelling for systematic identification and explicit representation of human errors. In: Proceedings of INTERACT, pp 192–212

Fields B, Wright P, Harrison M (1995) Applying formal methods for human error tolerant design. In: Software engineering and human-computer interaction. Springer, pp 185–195

Foley JD, Wallace VL (1974) The art of natural graphic man-machine conversation. Comput Graph 8(3):87

Garavel H, Graf S (2013) formal methods for safe and secure computer systems. Federal office for information security

Garavel H, Lang F, Mateescu R, Serwe W (2013) CADP 2011: a toolbox for the construction and analysis of distributed processes. Int J Softw Tools Technol Transf 15(2):89–107

Gimblett A, Thimbleby HW (2010) User interface model discovery: towards a generic approach. In: Proceedings of the 2nd ACM SIGCHI symposium on engineering interactive computing system, EICS 2010, Berlin, Germany, pp 145–154

Gimblett A, Thimbleby HW (2013) Applying theorem discovery to automatically find and check usability heuristics. In: Proceedings of the 5th ACM SIGCHI symposium on engineering interactive computing systems, pp 101–106

Hallinger P, Crandall DP, Seong DNF (2000) Systems thinking/systems changing & a computer simulation for learning how to make school smarter. Adv Res Theor School Manag Educ Policy 1(4):15–24

Hamilton D, Covington R, Kelly J, Kirkwood C, Thomas M, Flora-Holmquist AR, Staskauskas MG, Miller SP, Srivas MK, Cleland G, MacKenzie D (1995) Experiences in applying formal methods to the analysis of software and system requirements. In: Workshop on industrial-strength formal specification techniques, pp 30–43

Hamon A, Palanque PA, Silva JL, Deleris Y, Barboni E (2013) Formal description of multi-touch interactions. In: Proceedings of the 5th ACM SIGCHI symposium on engineering interactive computing systems, pp 207–216

Hardin DS, Hiratzka TD, Johnson DR, Wagner L, Whalen MW (2009) Development of security software: a high assurance methodology. In: Proceedings of 11th international conference on formal engineering methods, pp 266–285

Harrison M, Thimbleby HW (eds) (1990) Formal methods in HCI. Cambridge University Press

Harrison MD, Duke DJ (1995) A review of formalisms for describing interactive behaviour. In: software engineering and human-computer interaction. Springer, pp 49–75

Harrison MD, Masci P, Campos JC, Curzon P (2013) Automated theorem proving for the systematic analysis of an infusion pump. Electron Commun EASST69

Harrison MD, Campos JC, Masci P (2015) Reusing models and properties in the analysis of similar interactive devices, pp 95–111

Hix D, Hartson RH (1993) Developing user interfaces: ensuring usability through product process. Wiley, New York

ISO/IEC (1989) LOTOS—a formal description technique based on the temporal ordering of observational behaviour. International Standard 8807

IEC ISO (2001) Enhancements to LOTOS (E-LOTOS). International Standard 15437

ISO/IEC (2002) 13568, Information technology—Z formal specification notation—syntax, type system and semantics

Jambon F, Girard P, Aït-Ameur Y (2001) Interactive system safety and usability enforced with the development process. In: Proceedings of 8th IFIP international conference on engineering for human-computer interaction, pp 39–56

Kieras DE, Polson PG (1985) An approach to the formal analysis of user complexity. Int J Man Mach Stud 22:94–365

Kluge A, Greve J, Borisov N, Weyers B (2014) Exploring the usefulness of two variants of gaze-guiding-based dynamic job aid for performing a fixed sequence start up procedure after longer periods of non-use. Hum Fact Ergon 3(2):148–169

Knight JC, Kienzle DM (1992) Preliminary experience using Z to specify a safety-critical system. In: Proceedings of Z user workshop, pp 109–118

Knight JC, Brilliant SS (1997) Preliminary evaluation of a formal approach to user interface specification. In: The Z formal specification notation. Springer, pp 329–346

Knight JC, Fletcher PT, Hicks BR (1999) Tool support for production use of formal techniques. In: Proceedings of the world congress on formal methods in the development of computing systems, pp 242–251

Kummer O, Wienberg F, Duvigneau M, Köhler M, Moldt D, Rölke H (2000) Renew—the reference net workshop. In: Proceedings of 21st international conference on application and theory of Petri nets-tool demonstrations, pp 87–89

Kummer O (2002) Referenznetze. Universität Hamburg, Disseration

Li K-Y, Oladimeji P, Thimbleby HW (2015) Exploring the effect of pre-operational priming intervention on number entry errors. In: Proceedings of the 33rd annual ACM conference on human factors in computing systems, pp 1335–1344

Loer K, Harrison MD (2000) Formal interactive systems analysis and usability inspection methods: two incompatible worlds? In: Interactive systems: design, specification, and verification, pp 169–190

Loer K, Harrison MD (2002) Towards usable and relevant model checking techniques for the analysis of dependable interactive systems. In: Proceedings of automated software engineering, pp 223–226

Loer K, Harrison MD (2006) An integrated framework for the analysis of dependable interactive systems (IFADIS): its tool support and evaluation. Autom Softw Eng 13(4):469–496

Lutz RR (2000) Software engineering for safety: a roadmap. In: Proceedings of the conference on the future of software engineering, pp 213–226

Mancini R (1997) Modelling interactive computing by exploiting the undo. Dissertation, University of Rome

Markopoulos P (1995) On the expression of interaction properties within an interactor model. In: Interactive systems: design, specification, and verification, pp 294–310

Markopoulos P, Rowson J, Johnson P (1996) Dialogue modelling in the framework of an interactor model. In: Pre-conference proceedings of design specification and verification of interactive systems, vol 44, Namur, Belgium

Markopoulos P (1997) A compositional model for the formal specification of user interface software. Dissertation, University of London

Markopoulos P, Johnson P, Rowson J (1998) Formal architectural abstractions for interactive software. Int J Hum Comput Stud 49(5):675–715

Martinie C, Palanque PA, Navarre D, Winckler M, Poupart E (2011) Model-based training: an approach supporting operability of critical interactive systems. In: Proceedings of ACM EICS conference, pp 53–62

Martinie C, Palanque PA, Winckler M (2011) Structuring and composition mechanisms to address scalability issues in task models. In: Proceedings of IFIP TC 13 INTERACT, pp 589–609

Martinie C, Navarre D, Palanque PA (2014) A multi-formalism approach for model-based dynamic distribution of user interfaces of critical interactive systems. Int J Hum Comput Stud 72(1):77–99

Masci P, Ruksenas R, Oladimeji P, Cauchi A, Gimblett A, Li Y, Curzon P, Thimbleby H (2011) On formalising interactive number entry on infusion pumps. Electron Commun EASST 45

Masci P, Ayoub A, Curzon P, Harrison MD, Lee I, Thimbleby H (2013a) Verification of interactive software for medical devices: PCA infusion pumps and FDA regulation as an example. In: Proceedings of the 5th ACM SIGCHI symposium on engineering interactive computing systems, pp 81–90

Masci P, Zhang Y, Jones PL, Oladimeji P, D'Urso E, Bernardeschi C, Curzon P, Thimbleby H (2014a) Formal verification of medical device user interfaces using PVS. In: Proceedings of the 17th international conference on fundamental approaches to software engineering. Springer, pp 200–214

Masci P, Zhang Y, Jones PL, Oladimeji P, D'Urso E, Bernardeschi C, Curzon P, Thimbleby H (2014b) Combining PVSio with stateflow. In: Proceedings of NASA formal methods—6th international symposium, pp 209–214

Masci P, Ruksenas R, Oladimeji P, Cauchi A, Gimblett A, Li AY, Curzon P, Thimbleby HW (2015) The benefits of formalising design guidelines: a case study on the predictability of drug infusion pumps. Innov Syst Softw Eng 11(2):73–93

Mateescu R, Thivolle D (2008) A model checking language for concurrent value-passing systems. In: Cuellar J, Maibaum T, Sere K (eds) Proceedings of the 15th international symposium on formal methods. Springer, pp 148–164

Merriam NA, Harrison MD (1996) Evaluating the interfaces of three theorem proving assistants. In: Proceedings of DSV-IS conference. Springer, pp 330–346

Miller SP, Tribble AC, Whalen MW, Heimdahl MPE (2006) Proving the shalls. Int J Softw Tools Technol Transf 8(4–5):303–319

Miller SP (2009) Bridging the gap between model-based development and model checking. In: Tools and algorithms for the construction and analysis of systems. Springer, pp 443–453

Miller SP, Whalen MW, Cofer DD (2010) Software model checking takes off. Commun ACM 53 (2):58–64

Milner R (1980) A calculus of communicating systems. Springer

Moher T, Dirda V, Bastide R (1996) A bridging framework for the modelling of devices, users, and interfaces. Technical report

Murugesan A, Whalen MW, Rayadurgam S, Heimdahl MPE (2013) Compositional verification of a medical device system. In: Proceedings of the 2013 ACM SIGAda annual conference on high integrity language technology, pp 51–64

Navarre D, Palanque PA, Paterno F, Santoro C, Bastide R (2001) A tool suite for integrating task and system models through scenarios. In: Proceedings of the 8th international workshop on interactive systems: design, specification, and verification-revised papers. Springer, pp 88–113

Navarre D, Palanque PA, Bastide R, Schyn A, Winckler M, Nedel LP, Freitas CMDS (2005) A formal description of multimodal interaction techniques for immersive virtual reality applications. In: Proceedings of the 2005 IFIP TC13 international conference on human-computer interaction. Springer, pp 170–183

Navarre D, Palanque PA, Basnyat S (2008) A formal approach for user interaction reconfiguration of safety critical interactive systems. In: Computer safety, reliability, and security. Springer, pp 373–386

Navarre D, Palanque PA, Ladry J-F, Barboni E (2009) ICOs: a model-based user interface description technique dedicated to interactive systems addressing usability, reliability and scalability. ACM Trans Comput Hum Interact 16(4):1–56

Niwa Y, Takahashi M, Kitamura M (2001) The design of human-machine interface for accident support in nuclear power plants. Cogn Technol Work 3(3):161–176

Oladimeji P, Thimbleby HW, Cox AL (2011) Number entry interfaces and their effects on error detection. In: Proceedings of human-computer interaction—INTERACT 2011—13th IFIP TC 13 international conference, pp 178–185

Oladimeji P, Thimbleby HW, Cox AL (2013) A performance review of number entry interfaces. In: Proceedings of human-computer interaction—INTERACT 2013—14th IFIP TC 13 international conference, pp 365–382

Oliveira R, Dupuy-Chessa S, Calvary G (2014) Formal verification of UI using the power of a recent tool suite. In: Proceedings of the ACM SIGCHI symposium on engineering interactive computing systems, pp 235–240

Oliveira R (2015) Formal specification and verification of interactive systems with plasticity: applications to nuclear-plant supervision. Dissertation, Université Grenoble Alpes

Oliveira R, Dupuy-Chessa S, Calvary G (2015a) Verification of plastic interactive systems. i-com 14(3):192–204

Oliveira R, Dupuy-Chessa S, Calvary G (2015b) Equivalence checking for comparing user interfaces. In: Proceedings of the 7th ACM SIGCHI symposium on engineering interactive computing systems, pp 266–275

Oliveira R, Dupuy-Chessa S, Calvary G (2015c) Plasticity of user interfaces: formal verification of consistency. In: Proceedings of the 7th ACM SIGCHI symposium on engineering interactive computing systems, pp 260–265

OMG (2010) Systems modelling language (OMG SysML™), version 1.2

Palanque PA, Bastide R, Sengès V (1995) Validating interactive system design through the verification of formal task and system models. In: Proceedings of IFIP WG 2.7 conference on engineering human computer interaction, pp 189–212

Palanque PA, Bastide R (1995) Petri net based design of user-driven interfaces using the interactive cooperative objects formalism. In: Interactive systems: design, specification, and verification. Springer, pp 383–400

Palanque PA, Bastide R, Senges V (1996) Validating interactive system design through the verification of formal task and system models. In: Proceedings of the IFIP TC2/WG2.7 working conference on engineering for human-computer interaction, pp 189–212

Palanque PA, Paternó F (eds) (1997) Formal methods in HCI. Springer

Palanque PA, Bastide R, Paternó F (1997) Formal specification as a tool for objective assessment of safety-critical interactive systems. In: Proceedings of the IFIP TC13 international conference on human-computer interaction, pp 323–330

Palanque PA, Farenc C, Bastide R (1999) Embedding ergonomic rules as generic requirements in a formal development process of interactive software. In: Human-computer interaction INTERACT'99: IFIP TC13 international conference on human-computer interaction, pp 408–416

Palanque PA, Winckler M, Ladry J-F, ter Beek M, Faconti G, Massink M (2009) A formal approach supporting the comparative predictive assessment of the interruption-tolerance of interactive systems. In: Proceedings of ACM EICS 2009 conference, pp 46–55

Payne SJ, Green TRG (1986) Task-action grammars: a model of mental representation of task languages. Hum Comput Interact 2(2):93–133

Park D (1981) Concurrency and automata on infinite sequences. In: Proceedings of the 5th GI-conference on theoretical computer science. Springer, pp 167–183

Parnas DL (1969) On the use of transition diagrams in the design of a user interface for an interactive computer system. In: Proceedings of the 24th national ACM conference, pp 379–385

Paternó F, Faconti G (1992) On the use of LOTOS to describe graphical interaction. People and computers VII. Cambridge University Press, pp 155–155

Paternó F (1994) A Theory of user-interaction objects. J Vis Lang Comput 5(3):227–249

Paternó F, Mezzanotte M (1994) Analysing MATIS by interactors and ACTL. Technical report

Paternó F, Mezzanotte M (1996) Formal verification of undesired behaviours in the CERD case study. In: Proceedings of the IFIP TC2/WG2.7 working conference on engineering for human-computer interaction, pp 213–226

Paternó F (1997) Formal reasoning about dialogue properties with automatic support. Interact Comput 9(2):173–196

Paternó F, Mancini C, Meniconi S (1997) ConcurTaskTrees: a diagrammatic notation for specifying task models. In: Proceedings of the IFIP TC13 international conference on human-computer interaction, pp 362–369

Paternó F, Santoro C (2001). Integrating model checking and HCI tools to help designers verify user interface properties. In: Proceedings of the 7th international conference on design, specification, and verification of interactive systems. Springer, pp 135–150

Paternó F, Santoro C (2003) Support for reasoning about interactive systems through human-computer interaction designers' representations. Comput J 46(4):340–357

Petri CA (1962) Kommunikation mit Automaten. Dissertation, University of Bonn

Pfaff G, Hagen P (eds) (1985) Seeheim workshop on user interface management systems. Springer, Berlin

Reeve G, Reeves S (2000) μCharts and Z: Hows, whys, and wherefores. In: Integrated formal methods: second international conference. Springer, Berlin

Reisner P (1981) Formal grammar and human factors design of an interactive graphics system. IEEE Trans Softw Eng 7(2):229–240

Schwaber K (2004) Agile project management with scrum. Microsoft Press

Sifakis J (1979) Use of petri nets for performance evaluation. In: Beilner H, Gelenbe E (eds), Proceedings of 3rd international symposium on modelling and performance of computer systems

Shepherd A (1989) Analysis and training in information technology tasks. In Diaper D (ed) Task analysis for human-computer interaction, pp 15–55

Sufrin B (1982) Formal specification of a display-oriented text editor. Sci Comput Program 1:157–202

Sousa M, Campos J, Alves M, Harrison M, (2014) Formal verification of safety-critical user interfaces: a space system case study. In: Proceedings of the AAAI spring symposium on formal verification and modelling in human machine systems, pp 62–67

Spivey MJ (1989) The Z notation: a reference manual. Prentice-Hall, Upper Saddle River

Strunk EA, Yin X, Knight JC (2005) ECHO: a practical approach to formal verification. In: Proceedings of the 10th international workshop on formal methods for industrial critical systems, pp 44–53

Stückrath J, Weyers, B (2014) Lattice-extended coloured petri net rewriting for adaptable user interface models. Electron Commun EASST 67(13):13 pages. http://journal.ub.tu-berlin.de/eceasst/article/view/941/929

Thevenin D, Coutaz J (1999) Plasticity of user interfaces: framework and research agenda. In: Sasse A, Johnson C (eds) Proceedings of interact, pp 110–117

Thimbleby H (2007a) Interaction walkthrough: evaluation of safety critical interactive systems. In: Interactive systems. Design, specification, and verification. Springer, pp 52–66

Thimbleby H (2007b) User-centered methods are insufficient for safety critical systems. In: Proceedings of the 3rd human-computer interaction and usability engineering of the Austrian computer society conference on HCI and usability for medicine and health care. Springer, pp 1–20

Thimbleby H, Gow J (2008) Applying graph theory to interaction design. In: Gulliksen J, Harning MB, Palanque P, Veer GC, Wesson J (eds) Engineering interactive systems. Springer, pp 501–519

Thimbleby H (2010) Think! interactive systems need safety locks. J Comput Inf Technol 18 (4):349–360

Thimbleby HW, Gimblett A (2011) Dependable keyed data entry for interactive systems. Electron Commun EASST 45

Tu H, Oladimeji P, Li KY, Thimbleby HW, Vincent C (2014) The effects of number-related factors on entry performance. In: Proceedings of the 28th international BCS human computer interaction conference, pp 246–251

Turchin P, Skii R (2006) History and mathematics. URSS

Turner CS (1993) An investigation of the therac-25 accidents. Computer 18:9I62/93, 0700–001830300

van Glabbeek RJ, Weijland WP (1996) Branching time and abstraction in bisimulation semantics. J ACM 43(3):555–600

Wang H-W, Abowd G (1994) A tabular interface for automated verification of event-based dialogs. Technical report. DTIC Document

Wegner P (1997) Why interaction is more powerful than algorithms. Commun ACM 40(5):80–91

Weyers B, Baloian N, Luther W (2009) Cooperative creation of concept keyboards in distributed learning environments. In: Borges MRS, Shen W, Pino JA, Barthès J-P, Lou J, Ochoa SF, Yong J (eds) Proceedings of 13th international conference on CSCW in design, pp 534–539

Weyers B, Luther W, Baloian N (2010a) Interface creation and redesign techniques in collaborative learning scenarios. J Futur Gener Comput Syst 27(1):127–138

Weyers B, Burkolter D, Kluge A, Luther W (2010) User-centered interface reconfiguration for error reduction in human-computer interaction. In: Proceedings of the third international conference on advances in human-oriented and personalized mechanisms, technologies and services, pp 52–55

Weyers B, Luther W, Baloian N (2012a) Cooperative reconfiguration of user interfaces for learning cryptographic algorithms. J Inf Technol Decis Mak 11(6):1127–1154

Weyers B (2012) Reconfiguration of user interface models for monitoring and control of human-computer systems. Dissertation, University of Duisburg-Essen. Dr. Hut, Berlin

Weyers B, Burkolter D, Luther W, Kluge A (2012b) Formal modelling and reconfiguration of user interfaces for reduction of human error in failure handling of complex systems. J Hum Comput Interact 28(1):646–665

Weyers B, Borisov N, Luther W (2014) Creation of adaptive user interfaces through reconfiguration of user interface models using an algorithmic rule generation approach. Int J Adv Intell Syst 7(1&2):302–336

Weyers B (2015) FILL: formal description of executable and reconfigurable models of interactive systems. In: Proceedings of the workshop on formal methods in human computer interaction, pp 1–6

Weyers B, Frank B, Bischof K, Kluge A (2015) Gaze guiding as support for the control of technical systems. Int J Inf Syst Crisis Resp Manag 7(2):59–80

Whalen M, Cofer D, Miller S, Krogh BH, Storm W (2008) Integration of formal analysis into a model-based software development process. In: Formal methods for industrial critical systems. Springer, pp 68–84

Yin X, Knight JC, Nguyen EA, Weimer W (2008) Formal verification by reverse synthesis. In: Proceedings of international conference on computer safety, reliability, and security, pp 305–319

Yin X, Knight J, Weimer W (2009a) Exploiting refactoring in formal verification. In: Proceedings of dependable systems & networks, pp 53–62

Yin X, Knight JC, Weimer W (2009b) Exploiting refactoring in formal verification. In: Proceedings of the 2009 IEEE/IFIP international conference on dependable systems and networks, pp 53–62

Yin X, Knight JC (2010) Formal verification of large software systems. In: Proceedings of second NASA formal methods symposium, pp 192–201

Chapter 2
Topics of Formal Methods in HCI

Judy Bowen, Alan Dix, Philippe Palanque and Benjamin Weyers

Abstract In this chapter, we present an overview of some of the general themes and topics that can be seen in research into formal methods in human–computer interaction. We discuss how the contents of the rest of the book relate to these topics. In particular, we show how themes have evolved into particular branches of research and where the book contents fit with this. We also discuss the areas of research that are relevant, but are not represented within the book chapters.

2.1 Introduction

This chapters of this book are organised into three sections: modelling, execution, and simulation; analysis, validation, and verification; and future opportunities and developments. These represent specific themes of intended use under which we can group the work presented. While these (somewhat) broad groupings provide a particular categorisation, there are wider topics of interest we can describe when we

J. Bowen (✉)
University of Waikato, Hamilton, New Zealand
e-mail: jbowen@waikato.ac.nz

A. Dix
School of Computer Science, University of Birmingham, Birmingham, UK
e-mail: alanjohndix@gmail.com

A. Dix
Talis Ltd. Birmingham, Birmingham, UK

P. Palanque
IRIT—Interactive Critical Systems Group, University of Toulouse 3—Paul Sabatier,
Toulouse, France
e-mail: palanque@irit.fr

B. Weyers
Visual Computing Institute—Virtual Reality & Immersive Visualization,
RWTH Aachen University, Aachen, Germany
e-mail: weyers@vr.rwth-aachen.de

B. Weyers et al. (eds.), *The Handbook of Formal Methods in Human-Computer Interaction*, Human-Computer Interaction Series,
DOI 10.1007/978-3-319-51838-1_2

consider the literature of formal methods in HCI, under which the chapters of this book can also be considered.

HCI is in itself a broad topic and covers aspects of humans and their cognitive and physical capabilities, as well as machines and interface design at varying levels of detail. HCI must also consider the combination of these two things. When we introduce formal methods into the process, they may similarly be used for some, or all, of these considerations. Formal methods are also used in different ways across the HCI topics, and this is typically driven by the rationale behind the method being used and the purpose of its use within the process. While the benefits of using formal methods across all parts of the design process may be more obvious when working in safety-critical domains (health care, finance, transport, power generation, to name but a few), there is much to be gained from specific applications of use in other domains too, particularly when we consider the increasing levels of sophistication of both interfaces and interaction techniques as well as the ubiquity of such systems in the modern world. We can see examples of each of these approaches in the chapters that follow.

While it may appear that the differences between the HCI practitioner and the formal method practitioner are many, this is primarily an artefact of the approaches they use. In fact, both approaches have a common goal: the ability to reason about the software under construction in order to ensure it satisfies some set of requirements. Whether these requirements are expressed formally (as a specification for example) or as a set of user goals and tasks is then irrelevant, the point is to have some mechanism for ensuring these are met, and it is here that the use of formal methods for HCI becomes more obvious.

In the early years of research into formal methods for HCI, several approaches were developed which were based upon the use of existing formal methods and languages which were then applied to the topics of UI design and HCI, for example Jacky's use of Z to describe the interface to a radiation machine (Jacky 1997). Specialised approaches for interactive elements were developed, but still based on existing formalisms, such as the description of interactors in Z (Bramwell et al. 1995), VDM (Doherty and Harrison 1997), Lotos (Paternò et al. 1995), etc. This subsequently led to the development of new formalisms, or extensions to existing languages and notations, which were more suited for considerations of the user interface, interactions, and human users. While the design process for interfaces and interactions typically focuses on creating shared artefacts that can be used to communicate with all stakeholders in the process (as we would see in a typical user-centred design approach), this does not mesh naturally with the notations required for a more formal process.

Research into formal methods for HCI provides ways of supporting the development of interactive systems at differing levels of rigour and formality so that we gain the advantages that come from a more formal approach, while recognising the differences in process required when we must also consider users and interactions. While the nature of interfaces and types of interaction has changed considerably in the intervening years, many of the crucial factors in ensuring reliability, usability, safety, and soundness (e.g.) have not.

In this chapter, we consider topics of formal methods and HCI more generally and discuss how this has influenced work in the domain. We also describe how the chapters presented in the rest of the book contribute to these themes and build on the existing body of work.

2.2 Describing the Human User of Interactive Systems

HCI methods used to understand the human user have many of their roots in the disciplines of psychology, ergonomics, and pedagogy, as they seek to understand human capabilities (cognitive and physical) in order to design interactive systems that are usable and learnable, and which support the required tasks. While the complexity of human thought and behaviour in combination with proposed interactions may seem like a good fit for the use of formal methods, it is not without its challenges. One of the main problems faced when trying to formalise the user and their behaviour is that of unpredictability: we can never be certain what the user will do, and we can only surmise how they might behave. Typically then, most approaches abstract the problem by defining particular aspects of human behaviour rather than trying to capture full details of a user's thought process, understanding, motivations, and memory. These more abstract representations of users can then be used in conjunction with a model of a UI, or proposed set of interactions, to try and find areas of system interaction that may be problematic for a user and therefore more likely to lead to erroneous behaviour.

In Curzon and Blandford (2002), Curzon and Blandford used formal models of user cognition and actions. These are defined as rational actions a user will take to achieve a goal (so abstract away from random or malicious behaviours). The method is concerned with generic user models and examines how users may make mistakes in particular interfaces and how specific design rules would prevent this. This work has been developed over the years (and has its own basis in the earlier programmable user model concepts (Butterworth and Blandford 1997)), forming the foundation for several research approaches to user modelling. An example can be seen in Chap. 8, which builds on these models to develop the notion of user salience and then combines this with activation theory to try and predict likelihood of user error for particular designs. This enables a comparison of different designs so that those less likely to be problematic for users can be selected.

While the approach of Chap. 8 considers the user in terms of actions that are related to their goals and tasks, these are distinct from any model of the interactive system (allowing different combinations to then be considered). In contrast, Chap. 13 shows how human behaviour can be incorporated into larger formal models of the system to support verification. Here, the user behaviour (or the behaviour of several users) is described as a collection of tasks which are composed of a hierarchy of activities and actions. This is an example of how a well-used HCI technique (task analysis) can be enhanced through a formal approach (we will see the use of task analysis in several of the chapters) and then subsequently be used to build larger combined models of the system and interface/interactions (a theme that is revisited in Chap. 6).

Another use of task analysis, in this case task decomposition, is presented in Chap. 11 which describes how to create mental models from task decompositions for the purpose of assisting users in learning and using an interactive system. Again, there is an abstraction of user behaviour, in this case into rational task decomposition. However, this is different from the approach mentioned above in terms of the use of formal methods. Rather than using them to support design decisions or system verification, here the unambiguity supports a structured approach to guiding the use of the system. In common with the work above though, the starting point is the human user and the actions they can perform.

2.3 Formal Methods for Specific Types of Interactive Systems

Although we talk about interactive systems collectively as if they are all essentially the same, there are of course vast differences between these systems, ranging from single-user desktop-based systems to multi-user mobile adaptive systems, and everything in between. These differences, and their inherent complexities, have led to another strand of research in the use of formal methods for interactive systems which is to assist with reasoning about these different types of interface, interaction, and use type. We still see elements of user modelling in this work (particularly where the concern is multi-user systems), but typically the primary focus is on aspects of the interaction design itself.

Adaptive, context-aware, and plastic interfaces are good examples of categories of interface that lend themselves to formal analysis. The designer must be able to reason about how, and when, the interface changes to suit its context of use or platform. These types of UI are the focus of several different approaches. We see some examples of this in Chaps. 5, 7, 12, and 18 which demonstrate the range of different methods which can be used in this area. The notion of transformation of interfaces (either for plasticity or adaptivity) has a long history, much of it related to the use of XML-based languages such as XIML (Puerta and Eisenstein 2002) or USIXML (Limbourg et al. 2004) which have subsequently been incorporated into larger groups of tools (e.g. Michotte and Vanderdonckt 2008) or development frameworks such as TERESA (Correani et al. 2004).

There are other transformation approaches that have been developed, some based on more traditional refinement concepts from formal methods (see, e.g. Bowen and Reeves 2009; Oliveira et al. 2015) and others which combine several approaches, such as that of Chap. 10 which is based on Petri nets, category theory, and graph rewrite rules. There are also considerations beyond the interface which must be reasoned about when we investigate context-aware systems, and works such as Abi-Aad et al. (2003) and Costa et al. (2006) address this by describing models of the context itself, which can then be used in combination with system models. A different approach to formally modelling the context of use is also demonstrated in Chap. 12 where it is the combination of context model with interaction model that is used to determine suitability of a system under a particular set of circumstances.

While consideration of specific types of interactive systems is based on properties of the system itself, we can also consider types of UI under the umbrella of their use-domain, for example safety-critical systems. While these may be intended for use in very different use-scenarios (aircraft cockpits, medical devices, driverless trains, banking systems, etc.), they share common requirements. There are aspects of these systems which can potentially lead to serious loss or harm and as such we apply formal methods to such systems in order to reason about the safety criteria.

Although the focus of the chapters in this book is on the case studies presented in Chap. 4 (two of which are, of course, safety-critical applications), several of the chapters describe work which has been used in other safety-critical domains. Chapters 6, 8, 12, 13, and 14, for example, present work which has been used with medical devices, and the methods described in Chaps. 17 and 20 have been used in aircraft cockpit specifications.

Not all systems and interfaces are used by individuals. In complex systems, it is common for several users to work on individual interfaces which are all part of the same system. Systems which enable collaboration of users (or which provided mechanisms for collaboration) are sometimes termed ‘groupware’ or more commonly now ‘multi-user systems’. These come under the umbrella of ‘computer-supported cooperative work’ (CSCW) where the users, and use, are often considered along dimensions such as synchronous or asynchronous and colocated or remote. This matrix, first described in detail in Baecker et al. (1995), is the basis for several frameworks which were developed to help manage the design of such systems, see Greenberg (1996), Guicking et al. (2005), Antonaya and Santos (2010) for example.

Dealing with several users and multiple interfaces increases the difficulty when trying to ensure the system will behave as expected. As such, it is not surprising that this attracts the attention of formal practitioners. An example of such a system is seen in Chap. 15 where multiple interfaces in an air traffic control domain are modelled as multi-agent systems to consider human–machine, human–human, and machine–machine interactions.

Another way to approach this problem is to develop interface models which are modular and can therefore be composed (to develop the sort of system above) or exchanged and reused to either develop new systems from existing components or update parts of a system and be sure that certain behaviour is preserved. An example is shown in Chap. 5 where the appearance elements of the interface and the interaction logic are modelled separately to enable either (or both) to be considered as components of a larger system.

A factor of these more complex systems is that there are often elements of automation at play, which can themselves be considered as a type of interaction, and must certainly be reasoned about, in conjunction with the human user, in order to fully understand how such a system might behave. Chapter 7 discusses this showing how explicit system automated behaviours can be identified using activity modelling, whereas in Chap. 19, we see how interconnection of small interactive systems (in this case apps) can be described to consider how they might usefully be connected.

2.4 Descriptions of the Modelling Process and Supporting Tools

Not all formal modelling is aimed at specific design purposes as shown above. There is a body of work where the focus is on suitable models for interactive systems (the hows, whys, and wherefores) where the models can be used more generally across the design process. Some of these focus on aspects such as model-checking, verification, and validation with the aim of showing how these can be applied in the domain of interactive systems. These can be used for ensuring safety properties are met (in critical systems, for example Loer and Harrison 2002; Bowen and Reeves 2013) or to explicitly manage error-prone interaction types that may occur in many systems, such as number entry (Thimbleby 2015).

These general modelling approaches have to tackle the problem of separation of concerns between interface and functional elements, while at the same time managing the relationship between the two. This can be done either by explicitly separating the two parts within the modelling (as we see, for example, in Chaps. 6 and 14), or by focussing on properties of the interactive components which can then be considered as a separate entity, as in the approach shown in Chap. 9.

Going beyond the standard formal approaches of model-checking and verification and moving into the domain of code generation has also been considered. In some ways, UIs are well suited for such automation as specific widgets and layout types can be derived from models (see, for example Eisenstein and Puerta 2000). However, there are also known problems with fully automating the interface design process as typically the aesthetics suffer and usability and appearance may also be compromised.

It can be problematic to introduce new languages into the design process as they may not suitable for supporting the necessary collaboration between design team and end-user. As such, visual notations or domain-specific languages may be more suited in these environments. Chapter 16 addresses this problem by discussing domain-specific languages and shows how these can be used in interactive system development as a more intuitive solution for experts in the field (as opposed to experts in formal methods). Similarly, the use of support tools to simplify the inclusion of formal methods into design is another way to try and reduce the gulf of understanding. This is seen in the earlier discussed work on XML-based languages which incorporates a number of different design tools to make such integration easier and is represented here in Chap. 18 which describes tools which support task and interface modelling at different levels of abstraction.

2.5 Summary

Finally, beyond the practical modelling approaches, tools, and methods which are presented, there is also reflective work which seeks to consider some of the larger encompassing challenges that all such work must address. For example, just as the interactive systems we develop must be useful and usable for their intended use-groups, so too the methods and design models we create must similarly be useful within the design process and understandable by designers. The models and formalisms we use also have different types of users (designers, developers, end-users, etc.). We must also ensure that they are appropriate for each user and perhaps provide different views (via visualisations for end-users of specifications for software engineers e.g.) of the models created. Two chapters in this book address this issue, Chap. 17 from the perspective of usability of verification tools, while Chap. 20 considers the gaps which may still be present, however thorough the specification and verification process may be.

The characterisations of the work presented in this book via the topics of the four sections are not necessarily that clear cut. Similarly, the work discussed in this chapter encompasses many different topics of the book chapters rather than being neatly segregated as described. The intention is to show some of the common themes that exist in formal methods and HCI research and provide an overview of how the chapters support these, rather than imply that all of the work fits exactly under just these headings.

Not all of the topics and areas that can be identified are represented in this book. This is hardly surprising as the domains of both HCI and formal methods continue to grow and expand almost as quickly as the systems they describe. Some of the areas not covered here can be found in the proceedings of relevant conferences such as (EICS, INTERACT, CHI, FM, ICFEM, etc.) and are also discussed further in Chap. 3.

References

Abi-Aad R, Sinnig D, Radhakrishnan T, Seffah A (2003) CoU: context of use model for user interface designing. In: Proceedings of HCI international 2003, vol 4, LEA, pp 8–12

Antonaya SL, Santos C (2010) Towards a framework for the development of CSCW systems. In: Luo Y (ed) Cooperative design, visualization, and engineering, vol 6240. Lecture Notes in Computer Science. Springer, Berlin Heidelberg, pp 117–120

Baecker RM, Grudin J, Buxton WAS, Greenberg S (eds) (1995) Readings in human-computer interaction: toward the year 2000, 2nd edn. Morgan Kaufmann

Bowen J, Reeves S (2009) Supporting multi-path UI development with vertical refinement. In: 20th Australian software engineering conference (ASWEC 2009), 14–17 April 2009. Gold Cost, Australia, pp 64–72

Bowen J, Reeves S (2013) Modelling safety properties of interactive medical systems. In: Proceedings of the 5th ACM SIGCHI symposium on engineering interactive computing systems, ACM, New York, NY, USA, EICS '13, pp 91–100. doi:10.1145/2480296.2480314

Bramwell C, Fields RE, Harrison MD (1995) Exploring design options rationally. In: Palanque P, Bastide R (eds) Design, specification and verification of interactive systems '95. Springer, Wien, pp 134–148

Butterworth R, Blandford A (1997) Programmable user models: the story so far. Puma working paper WP8, Middlesex University

Correani F, Mori G, Paternò FM (2004) Supporting flexible development of multi-device interfaces. In: EHCI/DS-VIS, pp 346–362

Costa PD, Guizzardi G, Almeida JPA, Pires LF, van Sinderen M (2006) Situations in conceptual modeling of context. In: EDOC workshops, pp 6

Curzon P, Blandford A (2002) From a formal user model to design rules. In: Goos G, Hartmanis J, van Leeuwen J (eds) Interactive systems: design, specification and verification, no. 2545 in Lecture Notes in Computer Science, Springer Berlin, pp 1–15

Doherty GJ, Harrison MD (1997) A representational approach to the specification of presentations. Eurographics workshop on design specification and verification of interactive systems, DSVIS 97. Granada, Spain, pp 273–290

Eisenstein J, Puerta A (2000) Adaptation in automated user-interface design. In: IUI '00: Proceedings of the 5th international conference on Intelligent user interfaces, ACM Press, pp 74–81

Guicking A, Tandler P, Avgeriou P (2005) Agilo: a highly flexible groupware framework. In: Groupware: design, implementation, and use. In: Proceedings of 11th international workshop, CRIWG 2005, Porto de Galinhas, Brazil, pp 49–56, 25–29 Sept 2005

Jacky J (1997) The Way of Z: practical programming with formal methods. Cambridge University Press

Limbourg Q, Vanderdonckt J, Michotte B, Bouillon L, López-Jaquero V (2004) UsiXML: A language supporting multi-path development of user interfaces. In: Proceedings of 9th IFIP working conference on engineering for human-computer interaction jointly with 11th international workshop on design, specification, and verification of interactive systems, EHCI-DSVIS'2004, Kluwer Academic Press, pp 200–220

Loer K, Harrison MD (2002) Towards usable and relevant model checking techniques for the analysis of dependable interactive systems. In: Emmerich W, Wile D (eds) Proceedings 17th international conference on automated software engineering, IEEE Computer Society, pp 223–226. http://citeseer.ist.psu.edu/loer02towards.html

Michotte B, Vanderdonckt J (2008) Grafixml, a multi-target user interface builder based on usixml. In: ICAS '08: Proceedings of the fourth international conference on autonomic and autonomous systems (ICAS'08), IEEE Computer Society, Washington, DC, USA, pp 15–22. doi:10.1109/ICAS.2008.29

Oliveira R, Dupuy-Chessa S, Calvary G (2015) Plasticity of user interfaces: formal verification of consistency. In: Proceedings of the 7th ACM SIGCHI symposium on engineering interactive computing systems, EICS 2015, Duisburg, Germany, June 23–26, 2015, pp 260–265

Paternò FM, Sciacchitano MS, Lowgren J (1995) A user interface evaluation mapping physical user actions to task-driven formal specification. In: Design, Specificationa nd Verification of Interactive Systems. Springer, pp 155–173

Puerta A, Eisenstein J (2002) XIML: a universal language for user interfaces. In: Intelligent user interfaces (IUI). ACM Press, San Francisco

Roseman M, Greenberg S (1996) Building real-time groupware with groupkit, a groupware toolkit. ACM Trans Comput-Hum Interact 3(1):66–106

Thimbleby H (2015) Safer user interfaces: a case study in improving number entry. IEEE Trans Softw Eng. doi:10.1109/TSE.2014.2383396

Chapter 3
Trends and Gaps

Alan Dix, Benjamin Weyers, Judy Bowen and Philippe Palanque

Abstract This chapter attempts to identify future research directions for formal methods in HCI. It does this using two main approaches. First, we will look at trends within HCI more broadly and the challenges these pose for formal methods. These trends in HCI are often themselves driven by external technical and societal change, for example the growth of maker/hacker culture and the increasing dependence of basic citizenship on digital technology, effectively establishing external requirements for the field. Second, we will look inwards at the FoMHCI literature, the user interaction phenomena it is trying to address and the processes of interaction design it is intended to support. Through this second analysis, we will identify internally generated trends. This does not lead to a single overarching research agenda but does identify a number of critical areas and issues, and hence establishes opportunities for further research to expand the state of the art.

A. Dix (✉)
University of Birmingham, Birmingham, UK
e-mail: alan@hcibook.com

A. Dix
Talis Ltd., Birmingham, UK

B. Weyers
Visual Computing Institute—Virtual Reality & Immersive Visualization, RWTH Aachen University, Aachen, Germany
e-mail: weyers@vr.rwth-aachen.de

J. Bowen
Department of Computer Science, The University of Waikato, Hamilton, New Zealand
e-mail: jbowen@waikato.ac.nz

P. Palanque
IRIT—Interactive Critical Systems Group, University of Toulouse 3—Paul Sabatier, Toulouse, France
e-mail: palanque@irit.fr

B. Weyers et al. (eds.), *The Handbook of Formal Methods in Human-Computer Interaction*, Human-Computer Interaction Series,
DOI 10.1007/978-3-319-51838-1_3

3.1 Introduction

This book attempts to capture a snapshot of the state of the art of research in formal methods in human–computer interaction. In this chapter, we ask where it is going and where it could go. Where are the gaps, the opportunities and the challenges for the next decade?

The first part of the chapter looks outward, at broader changes in HCI, how technology and the role of technology is changing, from big data and the Internet of Things to the pivotal role of information technology in modern society.

The second looks more inward at the role of formalism: which aspects of user interaction are being studied and modelled, how formal methods fit into and are addressing different parts of the design and deployment process, and how the techniques and methods within formal methods in HCI are changing.

For each issue, we attempt to identify the challenges this poses for formal methods research in HCI and so create a road map, especially for those starting in the field looking for open questions and avenues for theoretical and practical research.

3.2 HCI Trends

The roots of HCI can be traced back many years, indeed the first true HCI paper was Brian Shackel's '*Ergonomics for a Computer*' in (1959); however, the discipline formed properly in the 1980s including the foundation of many of the major international conferences. The study of formal methods in HCI can be traced back to this same period including Reisner's (1981) use of BNF, Sufrin's (1982) Z specification of a display editor and the first PIE paper (Dix and Runciman 1985).

This flowering of HCI research of all kinds was closely aligned to the growth of the personal computer, which moved computing from the domain of a few technical specialists behind sealed doors, to ordinary professionals on their desktops. In the intervening thirty-five years, both technology and the use of technology have changed dramatically. Many of the old concerns are still important today, but there are also new directions in HCI and these create new challenges for the use of formal methods.

In this section, we will trace some of those trends based largely on a recent JVLC article that examined future trends of HCI (Dix 2016). We will then use these general HCI trends to highlight some of the formalisation challenges they present. The trends are divided into three kinds: changing user interaction, changing technology, and changing design and development. Of course, these are not independent; indeed, as noted, the whole discipline of HCI effectively grew out of a particular technological change, the introduction of the desktop computer, and this close interplay between technology and use has continued. While available technology does not necessarily determine the use of that technology, it certainly makes new things possible.

3.2.1 Changing User Interaction

The use of computation has evolved from the desktop to virtually every aspect of our day-to-day lives.

Choice and ubiquity—In the early days of HCI, computers were largely used as a part of work, your job and employer largely dictated whether you used a computer and, if so, the hardware or software you used. The justifications for usability were therefore essentially about the efficiency of the workforce. The rise of home computing in the 1990s, and particularly the growth of the Internet in the early 2000s, meant that the range of users was far wider, and furthermore, the users were customers and had choice—if applications were not usable and enjoyable, they would be rapidly dumped! This process has continued as prices and form factors have made computing (in some form) available to ever-wider groups worldwide. However, in recent years, this ubiquity has meant that computer access is assumed. In commerce, Internet shopping is not only commonplace, but it is usually the cheapest way to obtain many items; for example, online check-in can be far cheaper than at the airport. In civic society, e-Government services are not only common, but some countries are seeking to make them the only way to access certain services, for example, in the UK, the so-called universal benefits, which integrate many kinds of different welfare payments, can only be accessed via an Internet portal, despite low levels of digital literacy among precisely the social groups who are likely to be claimants (Citizens Advice Bureau 2013; Sherman 2013). That is computer, and in particular Internet, services are beginning to underpin society, so that digital exclusion becomes social exclusion; use is no longer a choice but a necessity for participation in civic society.

Formal challenges: The change from optional to necessary use of digital services makes it more important to be able to deal with all kinds of people and also limited digital access. One of the strengths of formal methods is that it can help us analyse and design for situations and people we do not naturally experience or understand. The way this works out for people and for devices and contexts is explored in the next two trends.

Diverse people—HCI has always had strands that focus on those who are different from the 'norm': those with varying abilities and disabilities, different cultures or different ages. Of course, this 'norm' has been contested; for example, the tendency to use undergraduate students as the principal subjects in experiments has meant that what is assumed to be universal human behaviour turns out to be very biased towards Western culture (Henrich et al. 2010). Proponents of universal design have long argued that we are all 'disabled' under certain circumstances, for example effectively blind to dials and controls while our visual attention is on driving, and therefore that if design that is good for those with some form of perceptual, motor or cognitive disability is in fact design that is good for all. Although for many years forms of anti-discrimination legislation have made 'design for all' mandatory, it has still remained a marginal area.

This has long been problematic, but now, because computation is becoming essential for day-to-day life, it is impossible to ignore. The necessity of access combined with ageing populations in many countries means that universal access is now essential. This is exacerbated in many countries where ageing populations mean that some levels of perceptual, motor or cognitive impairment are now 'normal' and universally where broadening societal use includes those with low digital literacy and indeed low literacy, including the so-called next billion users in the developing world. Increasingly, we need to consider those at the social, geographic and economic margins, not just the professional 'class A/B' users of the 1980s.

Formal challenges: Within the formal methods community, work on multi-modal systems and various forms of model-based interfaces (e.g. Chap. 18; Coutaz 2010; Meixner et al. 2011), go some way to addressing these issues. In professional development, internationalisation is a normal practice for product delivery, and ability checklists are used to tune technology to individual abilities (Dewsbury and Ballard 2014; Whittington and Dogan 2016). The latter are often 'formal' in the sense that they have codified knowledge, but whereas most work on formal methods is based around relatively complex analysis of relatively simple specifications, practical development has relatively large corpora of codified knowledge, but with very simple, tick-box-style reasoning.

There are clear opportunities to create user models that encompass the wide variations in human abilities, and formal technical challenges to combine the kinds of codified knowledge already available with other forms of formal reasoning.

Diverse devices and contexts—The mobile-first design philosophy has for some time emphasised that for majority users mobile devices may be their principal, or in the case of the 'next billion' possibly first and only, access to computation. Commercially, the growing range of devices commonly used now includes smartphones, tablets, smart TV, game consoles, and public displays as well as various forms of laptop or desktop computers. Perhaps as important, when we look at need for universal access 'at the margins', we have to consider poor network connectivity, 'last generation' technology and in many areas intermittent or total lack of power.

Formal challenges: The challenges of designing for multiple devices are being dealt with fairly well, both in the formal community with work on plasticity and professional practice, notably responsive design. More generally, this suggests we need methods that model both device characteristics, and their environment. Chapter 12 is a good example of the latter, capturing interactions between environmental aspects such as noise, with device modalities, such as audible output. It would be good to see these areas of research expand, in particular to include infrastructure context such as limited networks or power, not just screen size. We often do not even have adequate vocabulary for these: for example, rural networks often experience frequent short glitches, complete drops in connectivity for a few seconds or minutes; with no word or formal metric for these drops, adequate service cannot be specified in the way that bandwidth or latency can. Theoretical work in these areas may have very immediate practical benefits; despite the widespread focus on responsive design, it is common to have failings such as drop-down menus that do not fit on small-screen devices, and even when executed well, responsive design rarely breaks the mould of simple screen size, rather than more radical modification of the interaction style to fit the device and infrastructure context.

Physicality and embodiment—One of the defining features of early user-interface design was the identification of key abstract interaction primitives, not least the windows, icons, menus and pointers of WIMP. Having these abstractions made it possible to easily design applications in the knowledge that while the details of how these primitives appear and behave may vary between specific devices, they can be assumed to exist and in some ways isolate the application from the vagaries of specific devices and even operating systems. On the other hand, as Apple have exploited particularly well, there has always been a close relationship between physical design and software design.

In more recent years, various factors have made the physical, embodied and situated nature of digital technology more significant. Some of this is connected with new interaction modalities such as bodily interaction with Kinect or geo-spatial interaction such as Pokemon Go. Ultrahaptics now means it is even possible to give holodeck-like mid-air haptic feedback (Carter et al. 2013). In addition, as computation has become embedded in everyday objects and the environment, it becomes hard to separate the digital and physical design: this is evident both in research fields such as tangible user interfaces (Ishii 2003) and ubiquitous computing (Weiser 1991), as well as practical design such as screen-based washing machines, or public displays.

Formal challenges: The abstraction offered by WIMP has been helpful in formal specification, which has typically been able to operate well above the physical interaction layer of ARCH/Slinky (UIMS 1992; Gram and Cockton 1996). There is some work that deals with the more physical nature of devices including Eslambolchilar's (2006) work on cybernetic modelling of human and device interactions, Thimbleby's (2007) work on physical control layout, physigrams as described in Chap. 9 in this volume, and the use of space syntax and other formalisms for movement in the environment (Fatah gen Schieck et al. 2006; Pallotta et al. 2008). However, compared with more abstracted user-interface specification, this work is still nascent.

Really invisible—Weiser's (1991) vision of ubiquitous computing has computers becoming 'invisible'; however, this was in the sense that there are displays everywhere at various scales, but we are so used to them, they fade into the background. This is clearly happening, indeed it is an interesting exercise to walk around your house and count the displays. However, not all computers have obvious displays, and yet this computation embedded into the environment is becoming ever more common (e.g. a modern train has many hundreds of computers in each carriage controlling everything from lighting to toilet doors). Sometimes there is an explicit non-visual user interface, such as body interaction to control a public display or Star Trek-style voice commands. Sometimes there may be more implicit sensing leading to apparent effects, such as a door opening or light coming on. Some sensing may operate to facilitate user interactions in ways that are far less apparent, including the low-attention and incidental interactions described in Chap. 7.

Formal challenges: There has been some work in formal methods dealing with the architectural design of this form of environmentally embedded system (e.g.

Bruegger 2011; Wurdel 2011), some (e.g. Chap 7) dealing with non-UI interactions and some including models of the physical environment (e.g. Chaps. 7–9, 12, 15). However, like the related area of physical interactions, this work has nothing like the maturity of more abstracted direct interactions.

Experience and values—The shift in the 2000s from the professional to the domestic domain and the associated shift from employer decision to consumer choice meant that 'satisfaction', the oft-ignored lesser sibling of 'effectiveness, efficiency and satisfaction', began to take centre place. The most obvious sign of this was the job title changes from 'usability' and 'interaction design' to 'user experience'. However, this change in job title represented a more fundamental shift in focus towards the aesthetic and emotional aspects of design (Norman 2005). Furthermore, increasing scrutiny of the ways in which user interfaces permeate commercial and societal life has led to an examination of the ways in which values are purposefully or accidentally embodied in designs (Cockton 2004; Harper et al. 2008).

Formal challenges: While important trends, it is less clear, given the current state of knowledge, how these issues can be dealt with in a more formal way. One potential path might be to enable forms of annotation and argumentation around designs. Notations such as QOC or gIBIS allow the formalisation of argumentation structures, even though the semantic content of the arguments is entirely captured in textual labels and descriptions. As well as offering potential ways to document and track emotional and value aspects of a design, encouraging the user experience designer to create more formal descriptions could allow automated analysis of more workaday aspects of usability.

Social and personal use—Communication has always been a core part of computer use, from simple email to rich collaborative work; however, the growth of social networking has changed the dominant kinds of communication from functional to phatic. Furthermore, individual use of computers is often very personal, not least the collection of data related to health and well-being. From an interaction point of view, the focus is, as in the last issue, more about communicating feelings than information. From a governance point of view, there are increasing worries about the way information and images once shared cannot easily be recalled, the potential for abusive interactions, and the ways in which data analysis can be used by commercial and government bodies in ways which we had never imagined when simply posting a tweet.

Formal challenges: Several of the chapters in this book include multiple actors (e.g. Chaps. 13, 15), but dealt with largely in terms of the functional effects of their interactions. There has also been work formalizing collaborations in terms of beliefs (e.g. Ellis 1994) including the way this could be used to make sense of certain artistic installations (Dix et al. 2005). The most extensive formal work on the actual social aspects of interaction is in social network analysis, but to date this is entirely separate from interface-level analysis.

In the area of personal data, the earliest paper on privacy in HCI included simple formalism (Dix 1990), and there have been multiple works on privacy preserving frameworks and, perhaps most relevant, ways of exposing the implications of data sharing (Langheinrich 2002; Hong and Landay 2004). Issues of authentication, security and provenance are heavily formalized; however, as with the more social aspects, this is currently in ways which are largely disjoint from user-interface specification and analysis.

Notification-based interaction—Most social network applications are strongly oriented around streams and notifications. These shift the locus of control away from the user and to the choices that the system makes of what to show and when to make it known. There is concern that this can lead to loss of focus and efficiency; indeed, a survey of American college students found that more than 90% reported that digital technologies cause some distraction from their studies, and 34% more serious distraction (McCoy 2016), and another study found that in-class cell phone use (presumably for texting) led to a drop of a one-third of a grade point (Duncan et al. 2012).

Formal challenges: The majority of formal user-interface specification techniques are oriented around explicit user-controlled interaction with only a small amount of work in the formal domain on interruptions (Dix et al. 2004) and dealing with interactions at difference paces (Dix 1992b). However, there is extensive non-formal literature on the impacts of interruptions and multitasking (Adamczyk and Bailey 2004; Czerwinski et al. 2004; Bailey and Konstan 2006) and the opportunity to find ways to match the pace and timing of delivery of notifications to the user's tasks (Dix and Leavesley 2015).

Basic HCI—Although we have had over thirty years of 'standard' usability, still we do not get it right! For example, Apple is often seen as the pinnacle of design, yet, when you turn on a MacOS or iOS device, the splash screen invites interaction (password for MacOS, slide to unlock for iOS) well before the system is ready to interpret your actions. Some of this is probably due to the shift of foci towards aesthetic and emotive design; some to do with the very success of user interfaces meaning more and more people are creating web interfaces in particular, but with less intense training and background than was once the case.

Formal challenges: Tool support could help this, and indeed, several of the Chaps. 17 and 18 describe tool suites that aid designers (although in some cases quite engineering-savvy ones) to develop and analyse user interfaces. Work clearly needs to be done still to improve both (i) the level and kinds of analysis so that they can truly be used as expert guidance for the novice and (ii) be targeted so that a designer without a formal/mathematical background can use them. The domain-specific modelling notations described in Chaps. 5 and 16 are one approach to achieving this.

3.2.2 Changing Technology

New and emerging technologies pose fundamental challenges for people and society, with corresponding challenges for formalisation.

Vast assemblies—Smartphone users may have many dozens, even hundreds of apps, albeit the majority of interaction is with only a few. In the physical world, the

Internet of Things (IoT) promises to fill homes and workplaces with large numbers of potentially interacting small devices. Users will need means to manage and configure these devices, understanding how they can work together to solve specific problems. The larger the collection of devices or apps, the harder this will become. Furthermore, these vast assemblies of small items are likely to suffer from feature interactions, that is where two features, each potentially valuable in their own right, interact badly together. For example, imagine that your kitchen smoke detectors are programmed to increase their sensitivity when they detect the house is empty as no cooking is expected; however, if the Internet-enabled kettle turns itself on a few minutes before you arrive back from work, the steam may well set off the fire alarm.

Formal challenges: Dealing with large numbers of simple objects seems like ideal territory for formal methods. On the configuration side, Chap. 16 shows how workflow notations can be used to connect together apps to make larger functionality; this could be combined with techniques to infer or tangibly program connections (Turchi and Malizia 2016). Feature interactions have been studied for many years in telecoms, so there should be knowledge that could be borrowed and modified to deal with other kinds of complex assemblies.

Big data—Various forms of big data have been the subject of government funding, popular press and of course extensive commercial interest; this ranges from social networks, as discussed above, to large-scale science, such as at CERN. User interfaces to analysis tools and visualisation have some novel features, but in some ways not so dissimilar to relatively long-standing work in visualisation, data analysis and visual analytics (Thomas and Cook 2005; Keim et al. 2010). However, the vast scale does introduce new issues: how to get an overview of data that is too big to scan; how to track ownership and provenance; and new opportunities, for example the success of recommender systems.

Formal challenges: As with the discussion of social network analysis, while the tools for much of this are already formal, it is less clear how, or whether it is valuable, for these to directly connect to user-interface models, or whether supporting big data analysis is 'just' another application area. However, there is certainly great opportunity for big data to be used as part of the formal development process; for example, trace data can be mined to propose common interaction patterns and can be used as part of validation or to drive simulations. Also big data about applications domains could be used as part of knowledge-rich methods (see below).

Autonomy and complexity—Large volumes of data have led to a greater focus on complex algorithms to deal with that data including various forms of machine learning, not least 'deep learning' which has recently been used to master Go (Silver et al. 2016). Many problems that used to be thought to require rich symbolic reasoning, such as translation, are now being tackled using shallow but high-volume techniques (Halevy et al. 2009). However, these algorithms are often opaque, an issue that was highlighted concerning the very earliest machine-learning-based user interfaces, warning of the potential for unethical or even illegal discrimination and bias (Dix 1992a). As the use of these algorithms has become more common, these dangers have become more apparent leading to the General

Data Protection Regulation of the Council of the European Union (2016), which mandates that, for certain forms of critical areas, algorithms need to be able to explain their decisions (Goodman and Flaxman 2016).

The successes of machine learning have also led to a general resurgence of interest in the potential and dangers of intelligent algorithms. Sometimes this intelligence is used to aid user-driven interactions and sometimes to act autonomously. The latter have sometimes reached the popular press: for example, when scientists called for a ban on autonomous battlefield robots (Hawking et al. 2015) or when a driver was killed in a self-driving Tesla car (Yadron and Tynan 2016). In the case of Uber, even when there is a driver in the car, the driver's itinerary and fares are driven by computer algorithms, effectively high-level autonomy.

Formal challenges: The knowledge needed for algorithms to be both intelligent and explicable will draw on expertise from or similar to that found in HCI. Indeed, the UK funding body EPSRC (2016) has identified human-like computing as an important research area, deliberately drawing on cognitive science and human factors as well as artificial intelligence research. Some of this will be at a different level than the issues usually studied by those looking at formal methods in HCI, effectively providing application semantics for the user interface. However, the shifts of autonomy do need to be taken into account. There has been work on dynamic function allocation in cockpit and control situations (Hildebrandt and Harrison 2003), and a few chapters (e.g. Chaps. 7, 15) deal either explicitly or implicitly with more autonomous action sometimes simply at the level of opaque internal transitions.

3.2.3 Changing Design and Development

New ways of creating software and physical artefacts are altering the processes and people involved in user-interface design and hence the notations, tools and analysis techniques needed.

Maker/hacker culture and mass customisation—A new digital DIY-culture has emerged, made possible by accessible electronics such as Arduino and RaspberryPi and the availability of digital fabrication from fully equipped FabLabs to hobbyist-budget MakerBots. At an industrial scale, high-budget digital fabrication, such as metal printers, means that the complex and costly spare parts storage and distribution may soon be a thing of the past, replaced by just-in-time printing of everything from spare door handles to gear boxes. Between the two, there is the potential for a new niche of the digital artisan 'modding' consumer products, using open-source 3D-print files, or maybe iTunes-style commercial versions. At both industrial and artisan scale, there will certainly be greater scope for individual configuration and semi-bespoke design at a level beyond even today's ideas of mass customisation.

However, if everyone can make their own TV remote or washing machine facia panel, how do you ensure safety and usability of the resulting interfaces. Furthermore, if things do go wrong, who gets sued? For non-critical devices, it may be that we see the HCI equivalent of house makeover television programmes and DIY-style how to books aimed at mass-market. However, for both legal and brand protection, products may need to limit acceptable adaptations.

Formal challenges: At first it seems that DIY-culture could not feel further from formal methods; but in fact the need for guaranteed properties on highly configurable interfaces is precisely the kind of problem that formal analysis could address. Specification of usability properties goes back to the earliest days of the FoMHCI community (Dix and Runciman 1985; Thimbleby and Harrison 1990; Dix 1991a) and issues of plasticity and model-based design have been studied for many years (e.g. Coutaz 2010) and are represented in this book (Chap. 18).

The level of configuration is different from those that are typically addressed (e.g. changes in physical form) and the design audience is somewhat different. It is likely that at least two kinds of designer-users need to be addressed: those in commercial enterprises or large open-hardware projects determining the properties required and the range of variations that may be possible; and end-users, or those near to the end use (e.g. the digital artisan), who are performing modifications with some sort of tool support. The level of formal expertise needed even to understand the outputs of current tool and reasoning support is still far too high, but both the domain-specific languages in Chap. 16 and the layered approach in Chap. 14 seem like potential approaches.

Agile and test-driven development—Although there are domains where monolithic development processes dominate, agile development methods are common in many areas, especially web-based systems. Rather like maker culture, at first glance the apparent ‘try it and see’ feel of agile systems seems at odds with formal development, but in fact agile methodologies are highly disciplined, typically combining a strong use-case orientation with test-driven development. Furthermore, for massive-scale systems user-interface development is supported by big data, notably A/B testing (Kohavi et al. 2009; Fisher et al. 2012).

Formal challenges: Although these are very different cultures, there are clear areas where formal methods could make a greater input into agile development. First is use cases, which may already use UML or similar formalisms, and could easily be integrated into an appropriate task modelling framework. This could help with a problem of agile development that it is often hard to keep track of the ‘big picture’ when constantly creating and deploying incremental change. On the testing side, while some test-based development includes user interfaces, for example by using headless web browser simulations, this is still a problematic area. Early work has shown that formal property specification could have a place to play, especially in looking at properties that span across individual units of delivery (Bowen and Reeves 2011). Finally, the volume of data available from large-scale traces of user behaviour is perfect input for performance models such as variants of MHP (Card et al. 1980, 1983), as well as other purposes described previously when looking at big data.

To achieve this would not be without theoretical and practical challenges including addressing presenting methods in ways that are accessible to the ordinary developer, and creating specification methods that can more easily be addressed to facets of an evolving system.

3.3 Formalising Interaction: What and How

Having examined the driving forces from changes in HCI, we now turn to formal methods themselves. We look under four headings.

The first two concern the subject of formal methods in HCI, *what* they model. The first of these looks at the various *actors and entities* in the milieu of human interaction with computers, which are currently being modelled and which could be. The second is about different *levels of abstraction*, generalisation and granularity, the kinds of phenomena that we model and reason about.

The third and fourth headings are more about the process and nature of formal modelling. The first of these looks at the development process for interactive systems and asks when in the process our methods are valuable and who in this process is supported. Finally, we look at *how* our models work, the kinds of reasoning and modelling we are currently using and how this may need to change or be augmented, especially if we seek to support practical development.

3.3.1 What—Actors and Entities

The majority of the earliest work on HCI was focused almost entirely on the direct interaction between a single user and desktop computer or other form of computing device (Fig. 3.1). There was always an interest in the varied stakeholders and other context that surrounded such interactions, but most research, and in particular detailed interaction design and engineering, concerned this dyad. Not surprisingly, this has also been an important theme for formal methods in HCI.

Of course this dyadic interaction is important, but not the whole story; even today, students need to be constantly reminded to consider the broader context. Figure 3.2 shows some of the agents and entities in this wider picture. Typically, users do not interact with a single device but several either at different times, or at the same time, for example the mobile phone as 'second screen' while watching

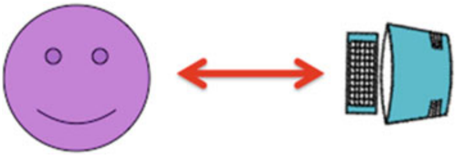


Fig. 3.1 Human–computer interaction: early days—one person, one computer

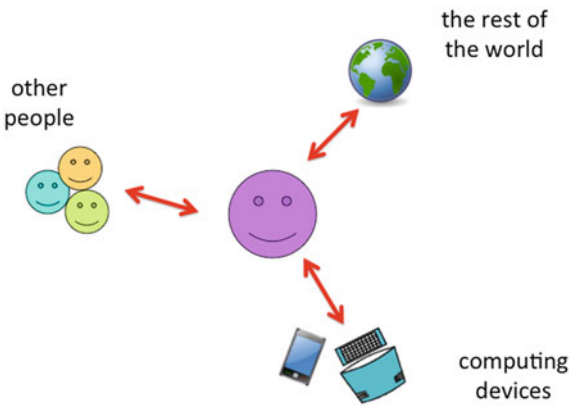


Fig. 3.2 Human–computer interaction: agents and entities

television. Users also interact with people and with other aspects of the world: from cars and cats to central heating systems and chemical plants; these interactions may simply set the context for direct computer interaction (e.g. noise as in Chap. 12), but may also in various ways concern the computer, for example finding your way around a city using a smartphone map.

As well as direct connections, there are indirect interactions: including sensors and actuators on physical objects such as self-driving cars and pervasive computing; computer-mediated communication and social networks; virtual elements overlaid on the physical world in augmented reality such as Pokémon Go; computational interfaces embedded into physical appliances such as washing machines; and even computational devices embedded in the user or other people.

Formal challenges: While there is still a strong concentration on direct interactions, the notations and techniques in the chapters in this book do include different elements in this picture. Several include some element of user or task modelling (Chaps. 8, 11, 13), building on traditions such as MHP (Card et al. 1980, 1983), ICS (Barnard 1985) and CCT (Kieras and Polson 1985), and some include ways to represent multiple actors (Chaps. 13, 15). As noted under *Physicality and embodiment* above, there is some work in modelling direct and indirect interactions with the physical environment (Chaps. 7–9, 15), but these are still rudimentary.

Even the nature of direct interactions has been changing as noted in discussions of notification-based interaction, invisibility and autonomy. There has been a line of work in formal methods dealing with multi-modal interfaces, but now there are fresh challenges, for example dealing with artful interactions such as in crafts or music making and physical movement in the environment.

In general, while formal methods have stepped beyond the single user—single machine dyad, there seems to be substantial room for further work on the richer picture.

3.3.2 What—Levels of Abstraction

Computer scientists, and especially formalists, love abstractions, and the formalisms in this book and in the community at large vary over a number of dimensions of abstraction.

Granularity—We can view user interaction at a very high/coarse level in terms of task analysis or workflows; the basic unit at this level might be quite substantial, such as interacting with an app (Chap. 19). We can also look at a very low/fine level of basic motor actions and physical interactions, such as Fitts' Law or physigrams as in Chap. 9. Between these, there are many intermediate levels, including the architectural levels in the ARCH/Slinky model. The majority of formal modelling work seems to live in this area where the basic unit of interaction is an abstract button press, or similar action (e.g. Chap. 5).

Formal challenges: This suggests two opportunities for future work: first to expand the focus of methods to encompass more of the higher and lower levels; second to fit different levels of models together to enable reasoning across these. This may not be trivial as interactions regarded as atomic at one level have structure at lower levels.

Continuity and time—In some ways, the finest level of interaction involves continuous action in continuous time. Despite early work on status–event analysis (Dix 1991a, b; Dix and Abowd 1996) and the TACIT European project (TACIT 1998; Faconti and Massink 2002), this is still an understudied area.

Formal challenges: Again there are complex formal issues, but in the broader formal methods community hybrid systems have been studied for many years, so there is clear opportunity to improve coverage. However, as with granularity, the ability to link targeted modelling at different levels seems crucial.

Level of generality—In the earliest strands of formal methods in HCI, there was work on notations and methods for specifying specific systems, for example Reisner's (1981) use of BNF and GOMS (Card et al. 1980, 1983); but also work on very abstract models looking at generic properties of all systems, for example the PIE model (Dix and Runciman 1985); and work on properties of undo (Dix 1991a; Mancini 1997). In this book, the majority of work is towards the specification of specific systems, but there is also some work that bridges the extremes of generality including DSL in Chap. 16 and the high-level properties in Chap. 14, which both offer ways to create specifications or properties that apply to a sub-class of systems. While several chapters use variations of generic systems properties such as predictability of actions or visibility of results, there seems to be little current work directed at this generic level.

Formal challenges: The lack of work at a generic level may be because there are very limited things one can say at this level of generality and the early work saturated the area. However, the work in this volume that operates generically over a sub-class of systems suggests a potential path that allows formalists to work alongside domain experts to create intermediate notations, formal properties and tools that can then be used by other designers.

Syntax versus semantics—Some philosophers, such as Searle (1997), would argue that by definition anything done in a computer is merely syntax, symbol juggling, and never true semantics as meaning cannot be reduced to rules. However, it is clear that some aspects of computation are more 'semantic' than others. Formal methods often talks about the syntax–semantics distinction, but it is noteworthy that both the Seeheim model (Pfaff and Hagen 1985) and ARCH/Slinky model (UIMS 1992; Gram and Cockton 1996) stop at adaptors/wrappers for application semantics. As noted under granularity, the majority of formal work is focused on the dialogue level and, hence, largely about the syntax of interaction, the order of actions and observations, but not their 'meaning'. There are exceptions to this. At a generic level, early analysis of undo included use of category theory to model the meaning of undo parameterised over the state semantics of arbitrary systems. At a more specific level, task analysis, cognitive models and domain modelling capture elements of the social, individual and application meaning. The use of ontologies and OWL in Chap. 12 is particularly interesting as these are languages of 'semantics'.

Formal challenges: It is possible that rich models of semantics of systems and the environment could become intractable; indeed, model checking of user-interface specifications already requires some form of abstract interpretation of value domains and faces challenges of combinatorial explosion. However, if formal methods in HCI are to reason about the entire human–computer work system, then there need to be models of each of the elements, so that the physical properties of the world, the human cognition and action, application 'semantics' and dialogue syntax can be verified to all work together to achieve specified goals. Of course, complete models of all of these would be intractable; the greatest challenge will be in determining appropriate levels of detail and abstraction to enable useful results.

3.3.3 Who and When (and Why?)

Just as we looked at the actors and flows involved in interaction itself, we can consider the actors, activities and products involved in the design process for interactive systems (see Fig. 3.3). As noted previously, moves towards agile development methods mean these stages are likely to be highly iterative including deployment itself, where some web-based systems may have many hundreds of releases per week.

Very early, indeed before the start of the design process proper, are the formal experts involved in the formulation of notations, properties and tool creation. As noted, this may also include high-level domain experts helping to create more domain-specific variants.

During the early stages of the design process for a specific system, there are many actors including interaction designers, product designers, user experience specialists; engineers, customers, management, domain experts; and hopefully

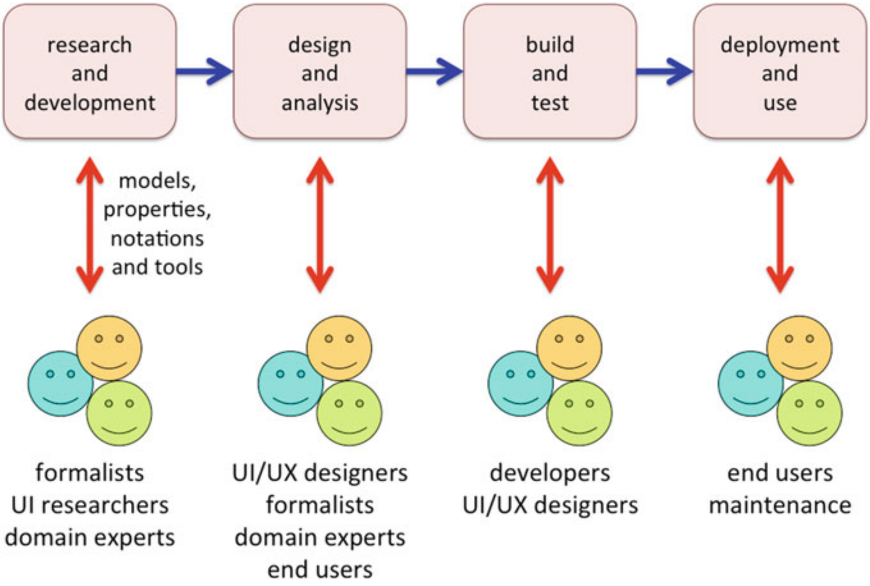


Fig. 3.3 Design process: processes and people

some users! In theory, appropriate notations and tools should help to clarify the design and communicate between stakeholders. The formal outcomes of this stage are detailed specifications, which potentially may be checked using model checkers or other tools.

As the design is turned into code, the formal specification may be translated into executable code; embedded into in-code verification; or used to create test suites.

Finally during use, executable specifications may actually be running in the code, or the user may use manuals that have been in part created, or verified using formal methods (as in Chap. 11). Traces of use may be collected and used as input for analysis tools for future systems. During this, some systems may be open to end-user modifications or appropriation.

Formal challenges: Although there is potential for formal methods to be useful at many stages in this process, in practice there is little real use beyond limited research case studies and some safety critical systems. Indeed, in a 2009 ACM Computer Surveys study of industrial use of formal methods all of the systems surveyed had a critical element and only 10% involved human–computer interaction (Woodcock et al. 2009). The two main barriers are the expertise needed to understand the methods and the time costs of the detailed specification analysis needed even when one has the requisite expertise. Safety critical systems are one of the few areas where the benefits can justify the extreme costs. This leads to three clear challenges: reducing formal expertise, reducing effort and increasing benefit. These are not new, 20 years ago Clarke and Wing (1996) concluded:

> Success in formal specification can be attributed to notations that are accessible to system designers and to new methodologies for applying these notations effectively (Clarke and Wing 1996)

The focus on tool support in several chapters is encouraging as is the use of domain-specific properties and languages (Chaps. 14, 16) as this can help to make the methods more accessible to non-formalists (*reducing formal expertise*). Ensuring any such tools are usable is also critical, as Chap. 17 emphasises (*reducing effort*). Indeed, it is interesting to see work where the design process is the *subject* of formal or structured analysis or modelling, including Chap. 17's use of Norman's model of action to analyse toolset usability and Bowen and Dittmar (2016) semi-formal framework for design spaces.

Using toolsets also means that there is greater opportunity to take the same designer input (in the form of task descriptions, interface specifications, etc.) and use it for different purposes, thus *increasing the benefit*. As several chapters were using ontologies, RDF and OWL, this suggests the potential for offering some form of expert-system-driven guidance (*reducing formal expertise*), perhaps the outcomes of formal analysis could be used to drive more knowledge-rich explanation systems.

As Chap. 6 points out, designers already create many artefacts such as sketches and low-fidelity prototypes; these could be used more effectively in formal methods in the way Denim (Lin et al. 2000) and subsequent systems did for executable prototypes, that is using computational effort to understand the designer's language. These all become even more important when we consider end-user development, although the use of graph transformations for reconfigurable UIs (Chap. 10) may offer one approach to this.

3.3.4 How

The above take us back to the kinds of formalisms we apply and the ways these might already be changing, or perhaps should change in the future based on some of the challenges we have seen.

Types of reasoning—Many of the chapters in this book use notations and methods that are similar in kind to those found in earlier collections (Thimbleby and Harrison 1990; Palanque and Paterno 1997), albeit used in different ways. There is substantial use of textual notations based on sets, functions, predicates and logics, and also more graphical notations including variants of Petri nets, statecharts and hierarchies for task analysis. A major difference from the early years is that many are subject to some form of automated checking or analysis as well as analysis by hand. Another new development is the use of OWL and similar notations and tools. Unfortunately, as already noted there is still very little use of the mathematics of continuous values or time.

Knowledge-rich reasoning—The use of OWL suggests the potential for more knowledge-rich reasoning. Traditional mathematics tends to be based on relatively few rules with relatively complex reasoning, in contrast to expert systems in AI with large rule sets (or knowledge bases) and relatively shallow reasoning. However, user-interface specification, and indeed formal specification in general, tends to already have relatively large specification with relatively simple (possibly automated) analysis. The whole of number theory can be (largely) built from nine Peano axioms, plus a little set theory, even 'toy' interface specifications have more! Furthermore, big data has proved 'unreasonably' effective in using statistical and shallow machine-learning techniques to address problems, such as natural language processing, that had formerly been seen as requiring symbolic artificial intelligence (Halevy et al. 2009). This suggests the potential for using techniques that bring together large data volume knowledge with specifications to address issues such as the inclusion of semantics as well as syntax, and the generation of automated design guidance.

Flexible levels of detail—We have seen how different formal notations and techniques operate at different levels of granularity, from workflows on apps to human–motor system analysis. In professional use, different levels will be appropriate for different aspects of a system; indeed, among the conclusions of an analysis of an early successful formal user-interface specification case study (Dix 2002a, b) was the need to be *useful* (address a real problem) and *appropriate* (no more detailed than needed), both of which may vary depending on which aspect of the system is under consideration.

Imagine a simple map-based system that includes buttons to select options, such as satellite imagery versus schematic, but then uses mouse movement and scroll wheel to drag and zoom the map: when the map is clicked a third-party gazetteer application opens in a pop-up showing additional information about the location. Unless the options buttons are particularly unusual, it will be appropriate to deal with them using a dialogue-level notation such as labelled state transitions (Chaps. 11, 15) or ICO (Chap. 17; Palanque 1992). At this level of specification, the map interactions would be abstracted, possibly to separate zoom and scroll functions, or maybe even to a single 'select location'. Similarly, as the gazetteer is pre-existing, we might abstract it to a single 'view gazetteer' operation and not attempt to model its interface even at the level of buttons and dialogue. However, we may also want to use a more detailed analysis of the map interactions themselves, perhaps making use of multi-scale Fitts' Law (Guiard et al. 2001).

There are two separate challenges here. First is dealing with systems at multiple levels of detail. This is already studied, for example, Chap. 6 on combining models and Chaps. 17 and 18, which have toolsets including notations at different levels; however, this may be more challenging when the notations used involve different paradigms (see below). The second, and perhaps more complex, is when different facets of the system are analysed at different levels so that the specification at certain levels of detail is not 'complete'.

Multiple notations—Many of the chapters in this book use multiple notations. Sometimes this is to deal with different levels as described above and sometimes because different aspects of the system use different notations, for example user and system models (Chaps. 8, 11) or physical and digital models (Chap. 9). Even when considering the same aspect at the same level of detail, different notations are required for different analysis techniques or tools, a translation process which is usually automated; for example, in Chap. 5 a user-interface model is translated into a form of Petri net for execution and in Chap. 16 domain-specific languages are translated into a form of UML and into linear temporal logic.

If the underlying paradigms are very close, then this may be relatively unproblematic; however, typically languages have different strengths, and information is lost in transforming from one model to another, for example, a single entity in one might correspond to several entities in another, or have no correspondence at all. If one notation is semantically richer than the other, then it may be possible to reason across the two by translation, but even then it can be problematic to translate the outputs of analysis back. A trivial example of this was the early days of Java Server Pages (JSP) before the development of 'Source Map' files (Oracle 2003): compiler messages referred to lines in often unintelligible generated Java code. Verifying the connection between notations also requires some sort of explicit or, more often, implicit shared semantics.

Generic descriptions and standards—Whether we are dealing with multiple notations within a single toolset or project, or trying to share artefacts (e.g. UI specifications, traces of user behaviour) between projects, there seems to be a need for different methods, notations and tools to be able to talk to one another. In this book, the nuclear power station case study is specified in some detail in Chap. 4, but how do we know that the formulation of this in other chapters refers to the 'same' thing? In general, as a community do we need some form of standardised means to share?

At a concrete level, this could be ways to specify elements of interaction such as layouts, behaviours or states, so that it is possible to use these to validate whether two specifications really are talking about the 'same' system. Given the differences between notations and models, the likelihood is that this may need to be partial, perhaps instance based: this particular state has this particular visual appearance and after a specific action modifies to a specified new state. Alternatively it may be possible to have a number of agreed ways of sharing more complex behaviours but limited to particular styles of specification.

At a more abstract level, we could seek semantic models that are not necessarily useful for actual specification (e.g. too verbose or complex), but can be used as a basis for giving semantics for other notations, rather like denotational semantics does for programing languages (Stoy 1977). If two notations are given semantics in such a shared semantic model, then it would become possible to assert reliably whether a specification in one is equivalent to one in another, or whether a translation algorithm between the two notations is valid.

One suggestion (Dix 2002a) has been that traces could act as a form of universal semantics, both at a concrete level and as a semantic model, as is used by process algebras. The advantage of this is that while different notations vary in how they abstract and parameterise behaviour, the actual realised behaviours are observable and shared. Traces are not without issues, notably continuous versus discrete time, and even discrete time at different granularities. A similar argument could be made for visual appearance at least for standard control panels with buttons, drop-downs, etc. Richer system or interface behaviour is more difficult, although a number of notations are enhanced versions of simpler ones such as labelled transition systems, or Petri nets; even if it is not possible to have a single interchange model, it may be possible to have a small selection.

3.4 Summary

Tables 3.1 and 3.2 summarise the principal topics and challenges that have emerged in this chapter; they are quite diverse and offer many opportunities for fruitful future research and real societal and economic impact. From these lists, we can bring out a few broader issues.

Table 3.1 HCI trends: summary and formal challenges

Trend	Formal challenge
Changing user interaction	
Choice and ubiquity	Importance of diversity (below)
Diverse people	Model-based interfaces to take into account varying abilities
Diverse devices and contexts	Plastic interfaces beyond screen size
Physicality and embodiment	Modelling beyond syntax layer
Really invisible	Radically new models of interaction
Experience and values	Linking argumentation to formal modelling
Social and personal use	Modelling multiple actors, privacy and provenance
Notification-based interaction	Modelling when uses not in control—matching pace and timing of notifications to user tasks
Basic HCI	Better tool support, especially for non-experts
Changing technology	
Vast assemblies	Application workflows and configuration; feature interactions
Big data	Use in formal development, e.g. mining trace data; knowledge-rich methods (below)
Autonomy and complexity	Human-like computing; dynamic function allocation; autonomous action beyond internal transitions
Changing design and development	
Maker/hacker culture and mass customisation	Guaranteed properties for configurable systems; tools for near end-users
Agile and test-driven development	Formalising use cases; generating tests; trace data

Table 3.2 Formalising interaction: summary and formal challenges

Topic	Formal challenge/issues
What—actors, entities	
Actors and entities	Broadening scope of FoMHCI; modelling users and tasks; physical aspects; artful interactions
What—levels of abstraction	
Granularity	Modelling beyond syntax layer; connecting models at different levels of abstraction
Continuity and time	Moving beyond discrete-event dialogue; hybrid systems
Level of generality	Domain-specific generic models
Syntax versus semantics	Modelling domain semantics
Who and when (and why?)	
Reducing formal expertise	e.g. domain-specific notations, expert-system guidance
Reducing effort	e.g. toolset development
Increasing benefit	e.g. using formal models for multiple purposes
How	
Types of reasoning	Broad range of notations used; increasing use of automatic analysis; limited knowledge-rich methods; continuity and time still poor
Knowledge-rich reasoning	Linking formal specifications and large knowledge bases; application semantics; automated design advice
Flexible levels of detail	Dealing with multiple levels of detail; working with incomplete specifications
Multiple notations	Translating between notations; verifying connections (shared semantics)
Generic descriptions and standards	Interchange models for specifications, physical layout, case studies, etc.; shared semantics (e.g. traces)

We need to extend the kinds of issues we approach, including: users of different abilities; varying devices and infrastructure; physical and semantic interface issues; and privacy. This may include ways to at least partially connect with hard to formalise areas including cognition, emotion and human values.

We need to be able to deal with systems where the computational part is more intelligent, autonomous and proactive; this includes issues such as notification-based systems, Internet of Things and robotics.

Some of these new or understudied uses may offer ‘easy wins’ for FoMHCI, for example in dealing with the complexities of IoT interactions that are hard for human assessment, or applications to agile methodologies.

We need a range of different levels and kinds of model and the ability to connect these: this includes links between existing modelling approaches in FoMHCI, and other approaches such as formal argumentation or large knowledge bases.

Applications in safety critical domains are likely to remain a core area of study for FoMHCI as these justify societally and economically the costs of extensive analysis. However, there are opportunities for FoMHCI to become more mainstream both by tackling the ‘easy win’ areas and by seeking ways for formal analysis to become more cost-effective and more accessible to domain experts, developers and end-users.

References

Adamczyk P, Bailey B (2004) If not now, when? The effects of interruption at different moments within task execution. In: Proceedings of the SIGCHI conference on human factors in computing systems (CHI ’04). ACM, New York, pp 271–278. doi:10.1145/985692.985727

Bailey N, Konstan J (2006) On the need for attention-aware systems: measuring effects of interruption on task performance, error rate, and affective state. Comput Hum Behav 22 (4):685–708. doi:10.1016/j.chb.2005.12.009

Barnard P (1985) Interacting cognitive subsystems: a psycholinguistic approach to short-term memory. In: Ellis A (ed) Progress in the psychology of language, volume 2, chapter 6. Lawrence Erlbaum Associates, Hove

Bowen J. Reeves S (2011) UI-driven test-first development of interactive systems. In: Proceedings of the 3rd ACM SIGCHI symposium on engineering interactive computing systems (EICS ’11). ACM, New York, pp 165–174. doi:10.1145/1996461.1996515

Bowen J, Dittmar A (2016) A semi-formal framework for describing interaction design spaces. In: Proceedings of the 8th ACM SIGCHI symposium on engineering interactive computing systems (EICS ’16). ACM, New York, pp 229–238. doi:10.1145/2933242.2933247

Bruegger P (2011). uMove: a wholistic framework to design and implement ubiquitous computing systems supporting user’s activity and situation. PhD Thesis, University of Fribourg, Switzerland. http://doc.rero.ch/record/24442

Card S, Moran T, Newell A (1980) The keystroke-level model for user performance with interactive systems. Commun ACM 23:396–410

Card S, Moran T, Newell A (1983) The psychology of human computer interaction. Lawrence Erlbaum Associates, Hillsdale, New Jersey

Carter T, Seah S, Long B. Drinkwater B, Subramanian S (2013) UltraHaptics: multi-point mid-air haptic feedback for touch surfaces. In: Proceedings of the 26th annual ACM symposium on user interface software and technology (UIST ’13). ACM, New York, pp 505–514. doi:10.1145/2501988.2502018

Citizens Advice Bureau (2013) 22% don’t have basic banking services needed to deal with Universal Credit. http://www.citizensadvice.org.uk/index/pressoffice/press_index/press_office20131105.htm. Accessed 29 Jan 2014

Clarke E, Wing J (1996) Formal methods: state of the art and future directions. ACM Comput Surv 28(4):626–643 (1996). doi:10.1145/242223.242257

Cockton G (2004) Value-centred HCI. In: Proceedings of the third Nordic conference on human-computer interaction (NordiCHI ’04). ACM, New York, pp 149–160. doi:10.1145/1028014.1028038

Council of the European Union (2016) Position of the council on general data protection regulation. http://www.europarl.europa.eu/sed/doc/news/document/CONS_CONS(2016)05418(REV1)_EN.docx. Accessed 8 April 2016

Coutaz J (2010) User interface plasticity: model driven engineering to the limit! In: Engineering interactive computing systems (EICS 2010). ACM, pp 1–8

Czerwinski M, Horvitz E, Wilhite S (2004) A diary study of task switching and interruptions. In: Proceedings of the SIGCHI conference on human factors in computing systems (CHI ’04). ACM, New York, pp 175–182. doi:10.1145/985692.985715

Dewsbury G, Ballard D (2014) DTA: the dependability telecare assessment tool—the person-centred telecare assessment. Dewsbury, Bourne, UK. http://www.gdewsbury.com/dta/

Dix A, Runciman C (1985) Abstract models of interactive systems. In: Johnson P, Cook S (eds) People and computers: designing the interface. Cambridge University Press, pp 13–22. http://alandix.com/academic/papers/hci88/

Dix A (1990) Information processing, context and privacy. In: Diaper DGD, Cockton G, Shakel B (eds) Human-computer interaction—INTERACT '90, North-Holland, pp 15–20

Dix A (1991a) Formal methods for interactive systems. Academic Press, London. http://www.hiraeth.com/books/formal/

Dix A (1991b) Status and events: static and dynamic properties of interactive systems. In: Duce DA (ed) Proceedings of the eurographics seminar: formal methods in computer graphics, Marina di Carrara, Italy. http://www.comp.lancs.ac.uk/computing/users/dixa/papers/euro91/euro91.html

Dix A (1992a) Human issues in the use of pattern recognition techniques. In: Beale R, Finlay J (eds) Neural networks and pattern recognition in human computer interaction. Ellis Horwood, pp 429–451. http://alandix.com/academic/papers/neuro92/neuro92.html

Dix A (1992b) Pace and interaction. In: Monk A, Diaper D, Harrison M (eds) Proceedings of HCI '92: people and computers VII. Cambridge University Press, pp 193–207. http://alandix.com/academic/papers/pace/

Dix A, Abowd G (1996) Modelling status and event behaviour of interactive systems. Softw Eng J 11(6):334–346. http://alandix.com/academic/papers/euro91/

Dix A (2002a) Towards a ubiquitous semantics of interaction: phenomenology, scenarios and traces. In: Forbrig P, Limbourg Q, Urban B, Vanderdonckt J (eds) Interactive systems. Design, specification, and verification 9th international workshop, DSV-IS 2002, Rostock, Germany, June 2002. Springer, LNCS 2545, pp 238–252. http://alandix.com/academic/papers/dsvis2002/

Dix A (2002b) Formal methods in HCI: a success story—why it works and how to reproduce it. Lancaster University, UK. http://alandix.com/academic/papers/formal-2002/

Dix A, Ramduny-Ellis D, Wilkinson J (2004) Trigger analysis—understanding broken tasks. In: Diaper D, Stanton N (eds) Chapter 19 in The handbook of task analysis for human-computer interaction. Lawrence Erlbaum Associates, pp 381–400. http://www.hcibook.com/alan/papers/triggers2002

Dix A, Sheridan J, Reeves S, Benford S, O'Malley C (2005) Formalising performative interaction. In: Proceedings of DSVIS '2005 (Newcastle, UK, 13–15 July 2005). Springer, LNCS 3941, pp 15–25. http://www.hcibook.com/alan/papers/DSVIS2005-performance/

Dix A, Leavesley J (2015) Learning analytics for the academic: an action perspective. J Univ Comput Sci (JUCS) 21(1):48–65. http://www.hcibook.com/alan/papers/JUCS-action-analytics-2015/

Dix A (2016) Human computer interaction, foundations and new paradigms. J Vis Lang Comput (in press). doi:10.1016/j.jvlc.2016.04.001

Duncan D, Hoekstra A, Wilcox B (2012) Digital devices, distraction, and student performance: does in-class cell phone use reduce learning? Astron Educ Rev 11(1). doi:10.3847/AER2012011

Ellis C (1994) Goal based workflow systems. Int J Collab Comput 1(1):61–86

EPSRC (2016) Human-like computing report of a workshop held on 17 & 18 February 2016, Bristol, UK. https://www.epsrc.ac.uk/newsevents/pubs/humanlikecomputing/

Eslambolchilar P (2006) Making sense of interaction using a model-based approach. PhD thesis, Hamilton Institute, National University of Ireland, NUIM, Ireland

Faconti G, Massink M (2002) A reference framework for continuous interaction. Int J Univ Access Inf Soc 1(4):237–251. Springer

Fatah gen Schieck A, Kostakos V, Penn A, O'Neill E, Kindberg T, Stanton Fraser D, Jones T (2006) Design tools for pervasive computing in urban environments. In: van Leeuwen JPT, Timmermans HJP (eds) Innovations in design and decision support systems in architecture and urban planning. Springer, Dordrecht, Netherlands, pp 467–486

Fisher D, DeLine R, Czerwinski M, Drucker S (2012) Interactions with big data analytics. Interactions 19(3):50–59. doi:10.1145/2168931.2168943

Goodman B, Flaxman S (2016) EU regulations on algorithmic decision-making and a “right to explanation”. In: Presented at 2016 ICML workshop on human interpretability in machine learning (WHI 2016), New York, NY. http://arxiv.org/abs/1606.08813v1

Gram C, Cockton G (eds) (1996) Design principles for interactive software. Chapman & Hall, London

Guiard Y, Bourgeois F, Mottet D, Beaudouin-Lafon M (2001) Beyond the 10-bit barrier: Fitts’ law in multi-scale electronic worlds. In: People and computers XV—interaction without frontiers: joint proceedings of HCI 2001 and IHM 2001. Springer, London, pp 573–587. doi:10.1007/978-1-4471-0353-0_36

Halevy A, Norvig P, Pereira F (2009) The unreasonable effectiveness of data. IEEE Intell Syst 24 (2):8–12. doi:10.1109/MIS.2009.36

Harper R, Rodden T, Rogers Y, Sellen A (eds) (2008) Being human: human-computer interaction in the year 2020. Microsoft Res. http://research.microsoft.com/en-us/um/cambridge/projects/hci2020/

Hawking S, Musk E, Wozniak S et al (2015) Autonomous weapons: an open letter from AI & robotics researchers. Future of Life Institute. http://futureoflife.org/AI/open_letter_autonomous_weapons

Henrich J, Heine S, Norenzayan A (2010) The weirdest people in the world? Behav Brain Sci 33 (2–3):61–83. doi:10.1017/S0140525X0999152X

Hildebrandt M, Harrison M (2003) Putting time (back) into dynamic function allocation. In: Proceedings of the 47th annual meeting of the human factors and ergonomics society

Ishii H (2003) Tangible bits: designing the seamless interface between people, bits, and atoms. In: Proceedings of the 8th international conference on intelligent user interfaces (IUI ’03). ACM, New York, NY, USA, pp 3–3. doi:10.1145/604045.604048

Hong J, Landay J (2004) An architecture for privacy-sensitive ubiquitous computing. In: Proceedings of the 2nd international conference on mobile systems, applications, and services (MobiSys ’04). ACM, New York, NY, USA, pp 177–189. doi:10.1145/990064.990087

Keim D, Kohlhammer J, Ellis G, Mansmann F (2010) Mastering the information age-solving problems with visual analytics. Eurographics Association, Goslar, Germany. http://www.vismaster.eu/wp-content/uploads/2010/11/VisMaster-book-lowres.pdf

Kieras D, Polson P (1985) An approach to the formal analysis of user complexity. Int J Man Mach Stud 22:365–394

Kohavi R, Longbotham R, Sommerfield D, Henne R (2009) Controlled experiments on the web: survey and practical guide. Data Min Knowl Discov 18(1):140–181. doi:10.1007/s10618-008-0114-1

Langheinrich M (2002) A privacy awareness system for ubiquitous computing environments. In: Borriello G, Holmquist LE (eds) Proceedings of the 4th international conference on ubiquitous computing (UbiComp ’02). Springer, London, UK, pp 237–245

Lin J, Newman M, Hong J, Landay J (2000) DENIM: finding a tighter fit between tools and practice for web site design. In: Proceedings of the SIGCHI conference on human factors in computing systems (CHI ’00). ACM, New York, USA, pp 510–517. doi:10.1145/332040.332486

Mancini R (1997) Modelling interactive computing by exploiting the undo. Dottorato di Ricerca in Informatica, IX-97-5, Università degli Studi di Roma “La Sapienza”. http://www.hcibook.net/people/Roberta/

McCoy B (2016) Digital distractions in the classroom phase II: student classroom use of digital devices for non-class related purposes. J Media Educ 7(1):5–32. http://en.calameo.com/read/00009178915b8f5b352ba

Meixner G, Paternó F, Vanderdonckt J (2011) Past, present, and future of model-based user interface development. i-com 10(3):2–11

Norman D (2005) Emotional design. Basic Books

Oracle (2003) JSR-045: debugging support for other languages. JSRs: Java Specification Requests. https://jcp.org/en/jsr/detail?id=45

Palanque P (1992) Modélisation par Objets Coopératifs Interactifs d'interfaces homme-machine dirigées par l'utilisateur. PhD, Toulouse I

Palanque P, Paterno F (eds) (1997) Formal methods in human-computer interaction. Springer, London

Pallotta V, Bruegger P, Hirsbrunner B (2008) Kinetic user interfaces: physical embodied interaction with mobile ubiquitous computing systems. In: Kouadri-Mostéfaoui S, Maamar M, Giaglis P (eds) Advances in ubiquitous computing: future paradigms and directions. IGI Global Publishing, pp 201–228. ISBN: 978-1-599-04840-6

Pfaff G, Hagen P (eds) (1985) Seeheim workshop on user interface management systems. Springer, Berlin

Reisner P (1981) Formal grammar and human factors design of an interactive graphics system. IEEE Trans Softw Eng 7(2):229–240. doi:10.1109/TSE.1981.234520

Searle J (1997) The mystery of consciousness. Granta Books, London

Shackel B (1959) Ergonomics for a computer design 120:36–39. http://www.idemployee.id.tue.nl/g.w.m.rauterberg/presentations/shackel-1959.PDF

Sherman J (2013) Half of benefit claimants lack skill to complete online forms. The Times, London. http://www.thetimes.co.uk/tto/news/uk/article3914355.ece

Silver D et al (2016) Mastering the game of Go with deep neural networks and tree search. Nature 529:484–489. doi:10.1038/nature16961

Stoy J (1977) Denotational semantics: the Scott-Strachey approach to programming language semantics. MIT Press, Cambridge, Massachusetts

Sufrin B (1982) Formal specification of a display-oriented text editor. Sci Comput Program 1:157–202. North Holland Publishing Co.

TACIT (1998–2002) Theory and applications of continuous interaction techniques. EU TMR Network Contract: ERB FMRX CT97 0133. http://www.webalice.it/giorgio.faconti/TACIT/TACITweb/doclist.html

Thimbleby H, Harrison M (eds) (1990) Formal methods in human-computer interaction. Cambridge University Press

Thimbleby H (2007) Using the Fitts law with state transition systems to find optimal task timings. In: Proceedings of second international workshop on formal methods for interactive systems, FMIS2007. http://www.dcs.qmul.ac.uk/research/imc/hum/fmis2007/preproceedings/FMIS2007preproceedings.pdf

Thomas J, Cook K (2005) Illuminating the path: research and development agenda for visual analytics. IEEE Press

Turchi T, Malizia A (2016) A human-centred tangible approach to learning computational thinking. EAI Endorsed Trans Ambient Syst 16(9):e6. doi:10.4108/eai.23-8-2016.151641

UIMS (1992) A metamodel for the runtime architecture of an interactive system. The UIMS developers workshop. SIGCHI Bull 24(1):32–37. ACM. doi:10.1145/142394.142401

Weiser M (1991) The computer of the 21st century. Sci Am 265(3):66–75

Whittington P, Dogan H (2016) A smart disability framework: enhancing user interaction. In: Proceedings of the 2016 British HCI conference

Woodcock J, Larsen P, Bicarregui J, Fitzgerald J (2009) Formal methods: practice and experience. ACM Comput Surv 41(4):19. doi:10.1145/1592434.1592436

Wurdel M (2011) An integrated formal task specification method for smart environments. University of Rostock. ISBN: 978-3-8325-2948-2

Yadron D, Tynan D (2016) Tesla driver dies in first fatal crash while using autopilot mode. Guardian

Chapter 4
Case Studies

Benjamin Weyers, Michael D. Harrison, Judy Bowen, Alan Dix and Philippe Palanque

Abstract This chapter introduces a set of case studies that are used in the rest of the book. They encompass well-known problem domains in human–computer interaction research and provide a practical focus for the approaches presented in this book. The set of case studies includes case studies concerned with the controller interface to a (semiautomated) nuclear power plant; a partly autonomous arrival management interactive system in the domain of air traffic control; a user interface for new interactive cockpits; and an interactive system used in rural and urban areas to maintain wind turbines. The final case study brings an interesting perspective for formal techniques, namely interactive public displays.

B. Weyers (✉)
Visual Computing Institute—Virtual Reality & Immersive Visualization,
RWTH Aachen University, Aachen, Germany
e-mail: weyers@vr.rwth-aachen.de

M.D. Harrison
School of Computing Science, Newcastle University, Newcastle upon Tyne, UK
e-mail: michael.harrison@ncl.ac.uk

J. Bowen
Department of Computer Science, The University of Waikato, Hamilton,

New Zealand
e-mail: jbowen@waikato.ac.nz

A. Dix
School of Computer Science, University of Birmingham, Birmingham, UK
e-mail: alanjohndix@gmail.com

A. Dix
Talis Ltd., Birmingham, UK

P. Palanque
IRIT—Interactive Critical Systems Group, University of Toulouse 3—Paul Sabatier,
Toulouse, France
e-mail: palanque@irit.fr

B. Weyers et al. (eds.), *The Handbook of Formal Methods in Human-Computer Interaction*, Human-Computer Interaction Series,
DOI 10.1007/978-3-319-51838-1_4

4.1 Introduction

This chapter introduces a set of case studies that encompass well-known problem domains in human–computer interaction research. They offer a basic set of case studies for the various approaches to formal methods presented in this book. The set includes case studies concerned with the controller interface to a (semiautomated) nuclear power plant; a partly autonomous arrival management interactive system in the domain of air traffic control; a user interface for new interactive cockpits; and an interactive system used in rural and urban areas to maintain wind turbines. The final case study brings an interesting perspective for formal techniques, namely interactive public displays. The first three case studies are presented in detail, while the other two are presented more briefly. We categorize the case studies according to how many users are involved, whether there is a formal system definition, whether it is safety critical, and what kind of interaction technique it offers (WIMP, post-WIMP, etc.). Additionally, Table 4.1 specifies in which chapters of the book the presented case studies have been used. Beside these categories, some specific characteristics will be given under the category "others" that are specific to the individual case study. Besides aspects such as the number of persons involved or WIMP versus post-WIMP, the case studies were selected according to the challenge they pose to formal methods in human–computer interaction (these are indicated in the rightmost column in Table 4.1).

Case study 1 "nuclear power plant" offers a formal specification of a technical system and defines a set of standard operating procedures, which should be mapped to a user interface design and implementation. It further raises the challenge of implementing interactive systems for semiautomated systems and to answer the question as to how far formal methods can tackle the challenge of automation in the control of safety critical systems.

Case study 2 "air traffic control" picks up the combination of an automated and safety critical system and maps this scenario to a team of 2–3 controllers. Thus, it poses the challenge to formal methods to address multi-user interaction scenarios in a safety-critical context, which includes a high degree of automation and also post-WIMP interaction techniques and concepts.

Errors and faults are a major issue in human–computer interaction research, which is especially relevant for users who control safety-critical systems but also for scenarios which address non-safety-critical systems but for which performance is a concern. Case study 3 "interactive aircraft cockpit" presents the role of hardware and software failures in the context of system certification. The challenge posed here is to which extend formal methods can support the design and development of interactive systems and user interfaces for certified technical systems. For this case study, software and hardware failures are addressed explicitly.

Case study 4 "wind turbine maintenance" and case study 5 "public display" leave the context of safety-critical systems and pose other challenges to formal methods. Case study 4 describes a distributed and multi-device scenario that raise the problem domain of coordination between devices, the distribution of interaction

Table 4.1 Classification of all presented case studies according to four characteristics and a set of specific attributes only addressing each individual case study

	#people involved	Safety critical	Formal specification	Post-WIMP	Used in chapter	Challenge to formal methods
1—Nuclear power plant	1	✓	✓	–[a]	5, 6, 10, 11, 14, 16, 19	• Semiautomated system
2—Air traffic control	2–3	✓	–	✓	13	• Automation • Description of collaborative activities
3—Interactive aircraft cockpit	1–2	✓	–	–[b]	15, 17, 20	• Need for reconfiguration mechanisms • Conformity with certification requirement/need to deal with faults
4—Wind turbine maintenance	1	–	–	–[c]	9, 19	• Distributed system
5—Public display	$n \geq 1$	–	–	✓	7, 12	• Context information

[a] WIMP
[b] WIMP including multi-modal graphical input
[c] WIMP as physical implementation

and information, and synchronous versus asynchronous communication. Case study 5 instead addresses post-WIMP interaction in a complete different spatial context. It further poses the challenge of addressing a completely unknown user base which is heterogeneous regarding preknowledge, interaction behavior, and training status. The question here is how formal methods support the development of such open-space interactive systems.

4.2 Case Study 1—Control of a Nuclear Power Plant

The control of a nuclear power plant involves a high degree of automation required to support the human controllers' tasks. These tasks include the full manual or partial manual starting and shutdown of the reactor, adjusting the produced amount of electrical energy, changing the degree of automation by activating or deactivating the automated management of certain elements of the plant, and handling the exceptional circumstances. In the case of the latter, the reactor operator's primary role is to observe the process as the reactor's safety system gradually suspends the operator from control until the system is returned to its safe state.

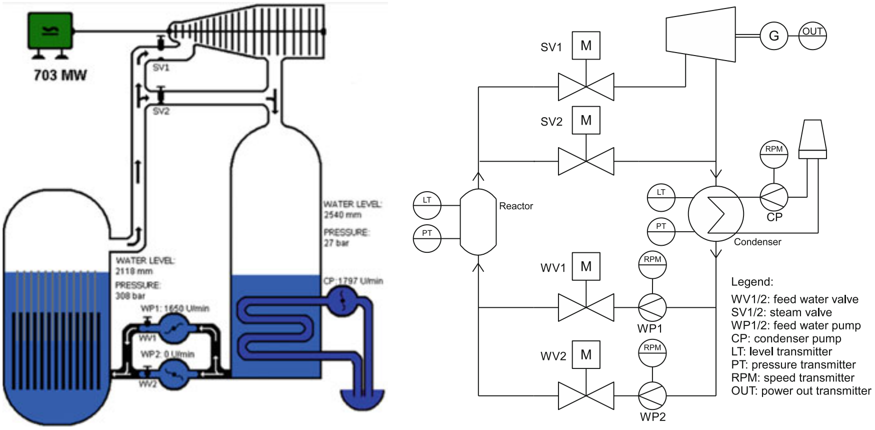


Fig. 4.1 Sketch of a simplified circuit of a boiled water reactor. On the *left* the process is presented as a sketch; on the *right* the process is presented as process flow diagram

Figure 4.1 shows a simplified boiling water reactor (BWR) design (United States Nuclear Regulation Commission 2016) as sketch on the left and as process flow diagram on the right. It comprises of three main components: the reactor core (lower left) containing the fuel elements and control rods, the turbine, which is connected to a generator, and the condenser, which condenses the steam generated by the reactor core back to fluid water. The whole process is driven by water pumps: two pumps pumping feedwater (purified water in contact with the reactor core) from the condenser into the reactor core (WP1 and WP2) and one pump transporting water through the cooling pipes in the condenser (CP). Thus, the latter controls the amount of water returned from steam, which is then transported back into the reactor core where WP1 and WP2 control the amount of water pumped into the reactor core for cooling and steam production.

The reactor (core) is responsible for the production of heat in the system. The amount of produced heat in the core, and thereby the amount of steam producing electrical energy in the turbine, is controlled by two parameters: (a) the amount of water pumped into the core (WP1 and WP2) and (b) the position of the control rods. Control rods are equipped with the material that is able to absorb neutrons, thereby reducing the chain reaction responsible for the emerging heat. Because the feedwater acts as moderator, increasing the amount of water pumped into the core increases the quantity of fissionable neutrons in the core, thereby increasing the heat. A safe state of the reactor core is specified as being in the range of up to 70 bar pressure and up to 286 °C (Gesellschaft für Anlagen- und Reaktorsicherheit (GRS) gGmbH 2016). Further control parameters and observables are the water level in the reactor and the condenser as well as the output power of the generator. Finally, valves can be used to control the flow of water and steam in the system, as shown in Fig. 4.1. WV1 and WV2 are able to cut off the feedwater; SV1 controls the steam for the turbine and SV2 the by-pass of steam which is then sent directly to the condenser.

4.2.1 Formalization of the Simplified BWR Design

The previously introduced design for a BWR can be formalized in various ways and with different types of formalizations as is done in Chap. 6 using Z (O'Regan 2014) or in Chap. 16 using a domain-specific and visual formal language. In this section, formalization is given as a PVS language-based listing (Owre et al. 1992) originating from the work presented in Chap. 14, which has been slightly changed to the specification given in (Gesellschaft für Anlagen- und Reaktorsicherheit (GRS) gGmbH 2016).

Listing 1 specifies the boundary conditions in which the system can be considered as safe. If the system leaves one of these ranges, it has to be considered as unstable or critical.

Listing 1 Maximum values

```
max_vp                 : nat = 2
reactor_max_level      : nat = 4000
condensor_max_level    : nat = 8000
reactor_max_pressure   : nat = 550
condensor_max_pressure : nat = 180
max_pressure           : posnat = 900
control_rods_max_level : nat = 100
max_flow               : posnat = 2000
```

Listing 2 lists the various types used for the specification of the nuclear power plant system. Some types are equipped with restrictions based on the boundaries defined in Listing 1, and some are composed values, e.g., valve_type.

Listing 2 Type definitions

```
itimes          : TYPE = posnat
temperature     : TYPE = nonneg_real
rods_level_type : TYPE = {x: nonneg_real |
                          x <= control_rods_max_level}
pos_type        : TYPE = nonneg_real
volume_type     : TYPE = nonneg_real
press_type      : TYPE = {x: nonneg_real | x <= max_pressure}
speed_type      : TYPE = nonneg_real
flow_val        : TYPE = real
vp_number       : TYPE = {n: upto(max_vp) | n > 0}

valve_type      : TYPE = [# flow: flow_val, on: Boolean #]
valves_type     : TYPE = [ vp_number -> valve_type]
pump_type       : TYPE = [# speed: speed_type, on: Boolean #]
pumps_type      : TYPE = [vp_number -> pump_type]

process_type    : TYPE = [# level : volume_type,
                            pressure: press_type #]
```

The various types define the various components relevant in the process through their characteristic parameters, such as temperature or rods_level_type as one specific characteristic for the reactor. The process that represents the system state of the modeled system has been defined as a complex type (see Listing 3).

Listing 3 System definition as a tuple of system values.

```
IMPORTING definitions_th[]

npp: TYPE = [#
  time                   : itimes,
  sv                     : valves_type,
  wv                     : valves_type,
  wp                     : pumps_type,
  cp                     : pump_type,
  reactor                : process_type,
  pos_rods_reactor       : rods_level_type,
  old_pos_rods_reactor   : rods_level_type,
  time_delta_pos         : itimes,
  condensor              : process_type,
  bw                     : temperature,
  poi_reactor            : pos_type,
  rest_heat              : temperature,
  cooled_water           : temperature,
  boiled_water          : temperature
#]
```

The nuclear power plant and the reactor as a central part are defined as a set of values or parameters, which will be related to each other in the following specifications and descriptions and equipped with dynamic behavior in Listing 5. The first definition is the water flow (flow_val, Listing 4 line 2) for each water valve wv. In case if a water valve is opened and its associated water pump is running (e.g., WV1 with WP1, cf. Fig. 4.1), the water flow is set to the current speed of the pump. Otherwise, the flow is set to 2 or −2 to simulate the effect of communicating vessels if the difference of the water level in the condenser and the reactor tank differs more than 470 mm (see Listing 4, ll. 7–11 and ll. 13–17).

Listing 4 Definition of the flow value for each water valve in the system.

```
flow_update(n: vp_number, st: npp):
 flow_val =
 COND (wv(st)(n)`on AND wp(st)(n)`speed > 0 AND
       condensor(st)`level > 0)
 -> wp(st)(n)`speed,

 (wv(st)(n)`on AND wp(st)(n)`speed = 0 AND
 condensor(st)`level > 0 AND
 ((condensor(st)`level - reactor(st)`level) > 470) AND
 (sv(st)(n)`on OR sv(st)(n)`on))
 -> 2,

 (wv(st)(n)`on AND wp(st)(n)`speed = 0 AND
 condensor(st)`level > 0 AND
 ((condensor(st)`level - reactor(st)`level) < 470) AND
 (sv(st)(n)`on OR sv(st)(n)`on))
 -> -2,

 ELSE -> 0
 ENDCOND
```

The dynamic change of the system is defined as a function tick, which describes the time-dependent evolution of the individual system values specified in Listing 3. Therefore, Listing 5 specifies the update of the various system values for each time step for an instance st of the nuclear power plant process type npp. First, the timer is increased (l.3). In the next step, the steam flow through the two steam valves is updated (ll. 6–13 for SV1 and ll. 15–21 for SV2). Additionally, for both valves, the current on-status is propagated (l.13 and l.21).

During operation, the reactor gets effected by chemical reactions that change the behavior of the reactor, reflected in a poisoning factor (`poi_reactor`, ll. 25–34). The poisoning of the reactor is positively correlated with the change in the control rod position over time (l.26, 27). It influences the produced amount of steam. If the poisoning is high, the reactor generates more heat because the poisoning chemical acts as moderator. As a stopped reactor still produces heat because of fuel decay, a factor of rest heat (`rest_heat`) is calculated if the reactor is shut off (`pos_rods_reactor(st)` = 100, l.37). The amount of boiled water (`boiled_water`, l. 46–58) is calculated most simply using the amount of water boiled by the chain reaction (factor bw, ll. 43,44), the rest heat and the poisoning factor as multiplier. The produced cooled water (`cooled_water`, ll. 60, 61) is calculated depending on the speed of the condenser pump CP and the pressure in the condenser vessel. The water pumped from the condenser into the reactor vessel is determined using the `flow_update` function given in Listing 4 (ll. 63–71). Finally, the reactor (ll. 73–78) and condenser (ll. 80–85) pressure and water-level parameter are updated according to the boiled water in case of the reactor and cooled water in case of the condenser vessel.

Listing 5 Dynamic behaviour of the nuclear power plant as discrete simulation.

```
tick(st: npp):
  npp = st WITH [
    time := time(st) +1,

    sv := LAMBDA (n: vp_number):
      COND n=1
      -> (# flow :=
         COND sv(st)(1)`on
         -> (reactor(st)`pressure - condensor(st)`pressure)/10,

         ELSE -> 0
         ENDCOND,
         on := sv(st)(1)`on #),

      n=2
      -> (# flow :=
         COND sv(st)(2)`on
         -> (reactor(st)`pressure - condensor(st)`pressure)/2.5,

         ELSE -> 0
         ENDCOND,
         on := sv(st)(2)`on #)
      ENDCOND,

    poi_reactor :=
      LET num_reactor =
          (old_pos_rods_reactor(st) - pos_rods_reactor(st))

      IN (
        COND num_reactor >= 0
        -> num_reactor / (time(st) - time_delta_pos(st)),

        ELSE -> -num_reactor / (time(st) - time_delta_pos(st))
        ENDCOND),

    rest_heat :=
      COND (pos_rods_reactor(st) = 100) AND (poi_reactor(st)=0)
      -> rest_heat(st)/1.05,

      ELSE -> 0
      ENDCOND,

    bw := (2*(100 - pos_rods_reactor(st))
          *(900-reactor(st)`pressure))/620,

    boiled_water :=
      COND (pos_rods_reactor(st) = 100 AND poi_reactor(st) = 0)
      -> bw(st) + rest_heat(st),

      (pos_rods_reactor(st) = 100 AND poi_reactor(st) > 0)
      -> (bw(st) + rest_heat(st))*poi_reactor(st),

      (pos_rods_reactor(st) < 100 AND poi_reactor(st) = 0)
      -> bw(st),

      (pos_rods_reactor(st) < 100 AND poi_reactor(st) > 0)
      -> bw(st)*poi_reactor(st)
      ENDCOND,

    cooled_water := 0.003 * cp(st)`speed *
                    SQRT(condensor(st)`pressure),

    wv := LAMBDA (n: vp_number):
      COND n=1
      -> (# flow := flow_update(1, st),
            on   := wv(st)(1)`on #),

      n=2
      -> (# flow := flow_update(2, st),
            on   := wv(st)(2)`on #)
      ENDCOND,

    reactor :=
     (#pressure := 0.25 * (reactor(st)`pressure - sv(st)(1)`flow -
                     sv(st)(2)`flow + boiled_water(st)),

       level    := reactor(st)`level + wv(st)(1)`flow +
                     wv(st)(2)`flow - boiled_water(st)#),

    condensor :=
     (#pressure := condensor(st)`pressure + sv(st)(1)`flow +
                     sv(st)(2)`flow - cooled_water(st),

       level    := condensor(st)`level - wv(st)(1)`flow -
                     wv(st)(2)`flow + 4 * cooled_water(st)#)
  ]
```

4.2.2 Standard Operating Procedures

For the operation of the presented system, three illustrative standard operating procedures (SOPs) are presented below. These procedures explain how the reactor controller is to define (e.g., through a user interface) the various parameters of the system. The *SOP Start-up* defines how to start the reactor to produce electrical energy where the *SOP Shut-down* defines the stopping of the reactor. The *SOP System Failure WP1* is an example of how the reactor controller should react if WP1 has a failure and stops working.

SOP Start-up

Aim

1. Bring output power to 700 MW, i.e., 100% of possible output power.
2. Hold water level in the reactor tank stable at 2100 mm.

SOP

1. Open FV2.
2. Set KP to 1600 U/min.
3. Open WV1.
4. Set WP1 to 200 U/min.
5. Stabilize water level in the reactor tank at 2100 mm by pulling out the control rods.
6. Open FV1.
7. Close FV2.
8. Increase U/min of WP1 and, in parallel, pull out control rods so that the water level in the reactor tank is stable at 2100 mm.
9. At 700 MW power output, stop pulling out the control rods. Water level of the reactor tank has to be stable at 2100 mm.

SOP Shut-Down

Aim

1. Reduce output power to 0 MW.
2. Hold water level in the reactor tank stable at 2100 mm.

SOP

1. Reduce output power to 200 MW by reducing the speed (U/min) of WP1 and, in parallel, push the control rods into the core. Hold water level in the reactor tank stable at 2100 mm.
2. Open FV2.
3. Close FV1.
4. Reduce output power to 0 MW so that:
    a. Control rods are completely pushed into the core,
    b. WP1 is stopped, and
    c. The water level in the reactor tank is stable at 2100 mm.
5. Close WV1.
6. Set KP to 0 U/min.
7. Close FV2.

SOP System Failure of WP1

Aim

1. Prevent a reactor meltdown.
2. Reduce power output to 0 MW in case of breakdown of WP1.
3. Hold water level in the reactor tank stable at 2100 mm.

SOP

1. Discharge control rods immediately into the core.
2. Open WV2.
3. Set WP2 to 800 U/min.
4. Control WP2 in such a way that the water level in the reactor tank stays stable at 2100 mm.
5. After residual heat is completely dissipated and pressure in the reactor tank and the condenser is equal to 0 bar:
    a. Close FV1, FV2, WV1, WV2 and
    b. Set all pumps to 0 U/min.

4.2.3 Automation

Automation is an essential part in controlling a nuclear power plant. In the context of this case study, the simplified BWR, as shown in Fig. 4.1, will be considered in the context of the following description of automation. Therefore, two main aspects of automation can be separated: (a) automation of control of certain components in the BWR and (b) automation of the safety system, which takes the control of the reactor when problems are detected in the system to bring the BWR back to a safe state.

(a) **Automation in Control**

The control of the BWR can be automated as follows:

1. *Feedwater Pumps*: The amount of feedwater pumped into the reactor core can be controlled by means of the water level (to keep it at, e.g., 2100 mm in the reactor tank), the (preselected) output power, and the pressure in the reactor tank. Therefore, the automation controls the pump's speed as variable. The used strategy can be manifold, e.g., using a linear transfer function.
2. *Control Rods*: The position of the control rods can be controlled automatically by keeping the water level and pressure constant in the reactor vessel, to reach the (preselected) output power.

3. *SCRAM*: The system could additionally offer an operation implementing an emergency reactor shutdown that pushes the control rods completely into the reactor core to stop the chain reaction, opens both feedwater valves, sets all pumps to a predefined speed as well as shuts down the steam flow to the turbine, and opens the pass-by channel for keeping the circulation of steam and water open. The constant water flow ensures the ongoing cooling of the reactor core after the shutdown.

(b) **Automation in Error Cases**

In a real nuclear power plant, the safety system excludes the user from the manual control of the system when there is a recognized system failure. This exclusion differentiates between 3 stages:

1. *Abnormal Operation*: This category specifies failures that can be handled, while the reactor is running without risks to the environment and the structure and function of the reactor.
2. *Design Basis Accident*: This category describes failures that endanger the structure and function of the reactor. It is necessary for the system to shut down immediately.
3. *Nuclear Accident*: This category describes failures which endanger the environment. The system is shut down automatically, and the controller is not able to restart it.

In case 1, the system partially excludes the user from the control. If certain system values are exceeded or do not reach certain boundaries, the system regulates itself back into a safe state. In cases 2 and 3, the system excludes the operator completely from the system and shuts down the reactor by executing the SCRAM procedure.

4.2.4 Connection with Formal Methods

The main challenge for an interactive system in scenario (a) (automation in control) is to offer a relevant interaction mechanism to monitor the automated systems as well as to offer relevant interaction operations, which enable the controller of the system to react if necessary. For instance, this is the case if failures occur which cannot be handled by the system but needs intervention by the system controller. An additional research question is in how far the mental model of the controller is influenced by this partial automation of the system control. Both the design of the interactive system and the reaction of the user on the partial automation offer a variety of challenges to be addressed in HCI research from which the use of formal methods can benefit.

As in scenario (a), the perspective on the system's design in scenario (b) (automation in error cases) as well as on the user raises various research questions.

Compared to scenario (a), scenario (b) considers full automation as well as the exclusion of the user in some circumstances. Based on this, the presented system allows the investigation of system designs as well as the user's behavior in cases of error-driven automation of the system, which is a specific type of research on the interaction with automated systems.

4.3 Case Study 2—Arrival Manager Within an Air Traffic Control Workstation

The air traffic control activity in the TMA (terminal maneuvering area) is an intense collaborative activity involving at minimum two air traffic controllers (see Fig. 4.2) communicating with more than one aircraft. The TMA is the area where controlled flights approach and depart in the airspace close to the airport. The planner controller (left-hand side of Fig. 4.2) is in charge of planning clearances (orders) to be sent to pilots by the executive controller (right-hand side of Fig. 4.2) who uses a radar screen.

The AMAN (Arrival MANager) tool is a software planning tool suggesting to the air traffic controller an arrival sequence of aircraft and providing support in establishing the optimal aircraft approach routes. Its main aims are to assist the controller to optimize the runway capacity (by determining the sequence) and/or to regulate/manage (meter) the flow of aircraft entering the airspace, such as a TMA (EUROCONTROL 2010). It helps to achieve more precisely defined flight profiles and to manage traffic flows, to minimize the airborne delay, and to lead to improved efficiency in terms of flight management, fuel consumption, time, and runway capacity utilization. The AMAN tool uses the flight plan data, the radar data, an aircraft performance model, known airspace/flight constraints, and weather information to provide to the traffic controllers, via electronic display, two kinds of information:

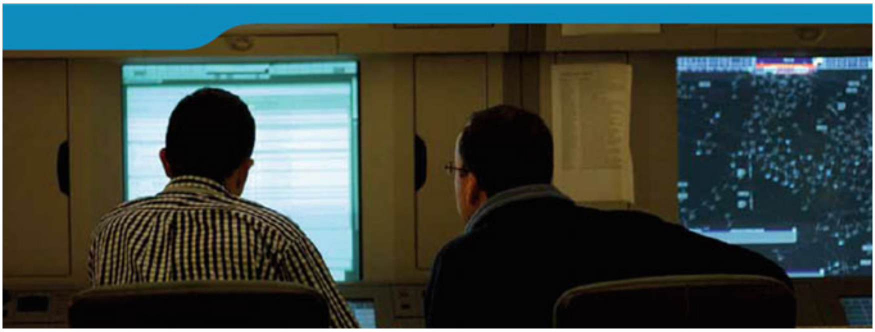


Fig. 4.2 Two TMA controllers working collaboratively

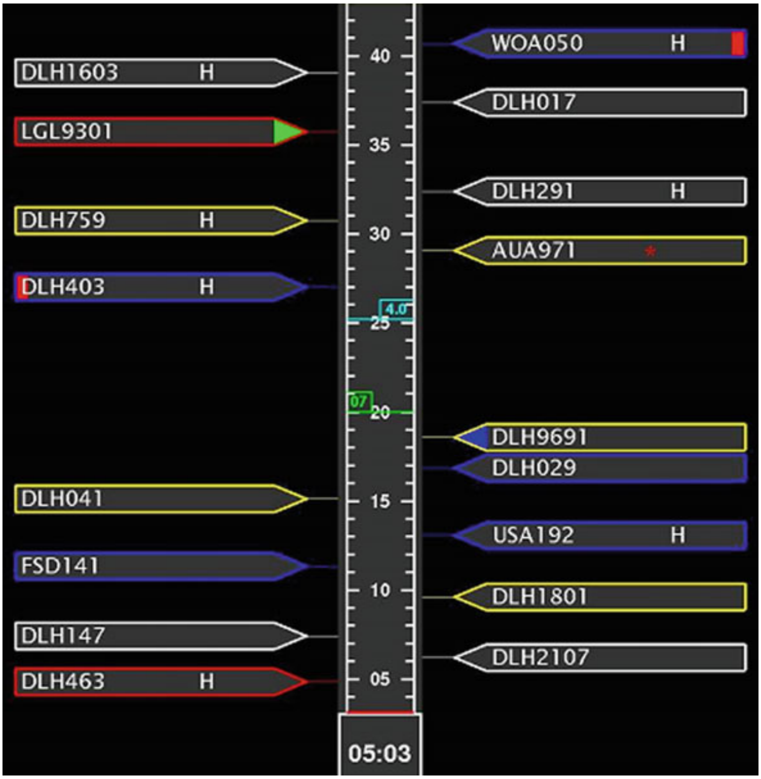


Fig. 4.3 Screenshot of a subpart of an AMAN User Interface (arrival sequence)

- A Sequence List (SEQ_LIST), an arrival sequence that optimizes the efficiency of trajectories and runway throughput (see Fig. 4.3)
- Delay management advisories, for each aircraft in the ATCO's airspace of competence.

The EXC_TMA is the controller delegated to handle the ground/air/ground communications, communication with pilots and releasing clearances to aircraft. He/she has the tactical responsibility of the operations, and he/she executes the AMAN advisories to sequence aircraft according to the sequence list. For the case study scenario, we propose that the pilots assume a passive role, limited to the reception and execution of the clearances. Other more active roles (such as requesting an emergency landing) can be considered but are likely to make things significantly more complicated.

4.3.1 Air Traffic Controller Tasks

Tasks of the EXEC_TMA air traffic controller is described in Fig. 4.4 using the HAMSTERS notation (Martinie et al. 2011; Forbrig et al. 2014). The notation, presented in Martinie et al. (2014), explicitly supports collaborative activities among users. Figure 4.4 details the “Manage aircraft arrivals” task (first row of Fig. 4.4). This task consists of performing concurrently four different tasks (second row of Fig. 4.4): monitoring AMAN advisories, providing clearances to pilots, ensuring distance separation between planes, and ensuring flights’ positions. The “monitor AMAN advisories” abstract task is refined as follows (see row 4 of Fig. 4.4). First, the AMAN system displays the advisories, then the air traffic controller perceives these advisories, and finally, the air traffic controller analyzes these advisories. This task is followed by the “Provide clearance to pilots” task that is further refined in Fig. 4.4. Due to lack of space, “Ensure distance separation” and “Ensure flights’ position” tasks’ representation are collapsed in Fig. 4.4.

4.3.2 User Interface of the Air Traffic Control Radar Screen

An example of an ATC radar screen is shown in Fig. 4.5. In this figure, one can see the labels associated with each aircraft including information such as aircraft callsign and cleared flight level. The line ahead of the aircraft spot is called the speed vector and describes the position of the aircraft in 3-min time. The longer the line, the faster the aircraft. That line does not take into account the change in heading if any; i.e., if the aircraft is changing heading, then it will not be where the speed vector indicates in 3 min. Behind the spot that indicates the position of the aircraft, the set of dots identifies the previous positions of the aircraft (usually 5 of them).

4.3.3 Connection with Formal Methods

This case study addresses various challenges for formal methods in HCI. First, the tasks that have to be performed by the air traffic controllers involve several operators using a complex workstation with multiple screens and multiple input devices. Second, the operators’ tasks are highly collaborative, including collocated and remote collaborations (e.g., communications between the two air traffic controllers and between the air traffic controllers and the aircraft pilots within the sector). It also highlights the integration of partly autonomous support to operator’s tasks (via the AMAN software) that raise the issue of complacency, situation awareness, control, and behavior forecasting of the autonomous systems.

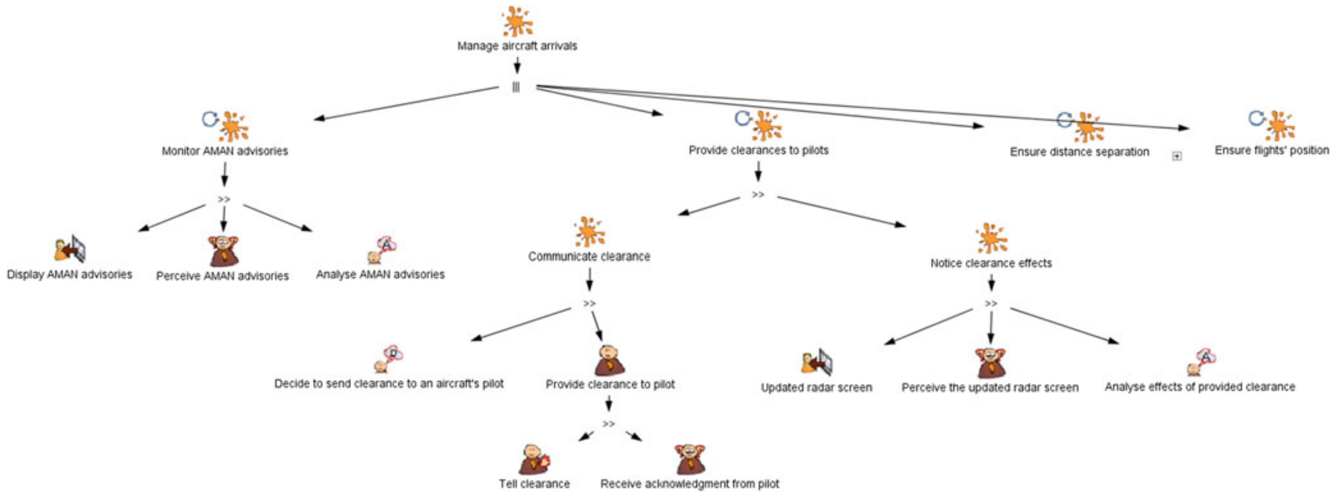


Fig. 4.4 Task model of the management of arrivals in the TMA area

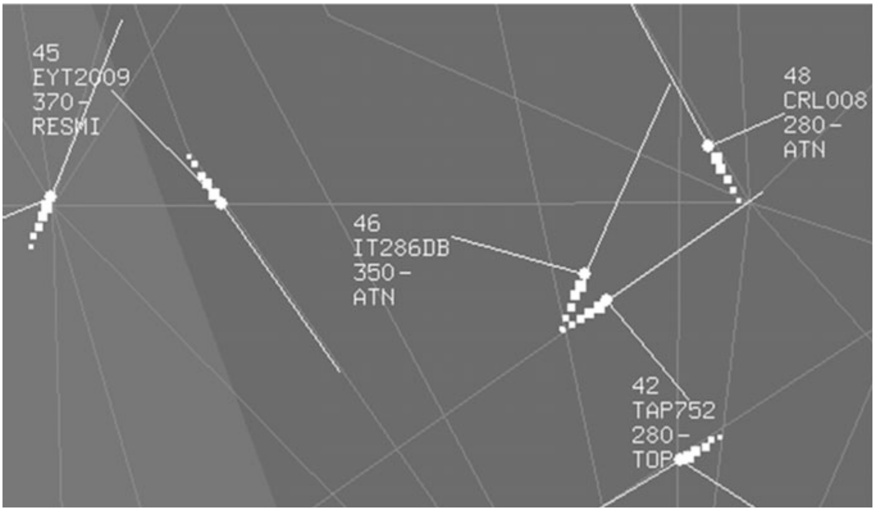


Fig. 4.5 ATC radar screen (each label representing an aircraft)

For many years, interaction with the ATC workstations was limited to zooming on a CRT (cathode ray tube) display. Recently, a lot of effort has been deployed in integrating new technologies such as tactile interactions, which raises more constraints in terms of specification of interactive systems and their related interaction techniques.

4.4 Case Study 3—Interactive Aircraft Cockpits

With the introduction of the ARINC 661 specification (Airlines Electronic Engineering Committee 2002) in the early 2000s, the new generation of aircraft (e.g., Airbus A380, A350 WXB, Boeing 787…) includes graphical interfaces in their cockpits. These graphical interfaces are applications that feature graphical input and output devices and interaction techniques such as you would find in any other digital interactive systems (office and home computers, Web applications…). The set of graphical user interfaces (GUIs) in the cockpit is called the Control and Display System.

As illustration, the example of the Airbus A380 cockpit is shown in Fig. 4.6. The Control and Display System is composed of eight output devices called Display Units (composed of a LCD screen, a graphics processing unit and a central processing unit) and two input devices called Keyboard and Cursor Control Units (KCCUs). The pilots can interact with some of the applications displayed on the Display Unit by using the keyboard and track ball of the KCCU.

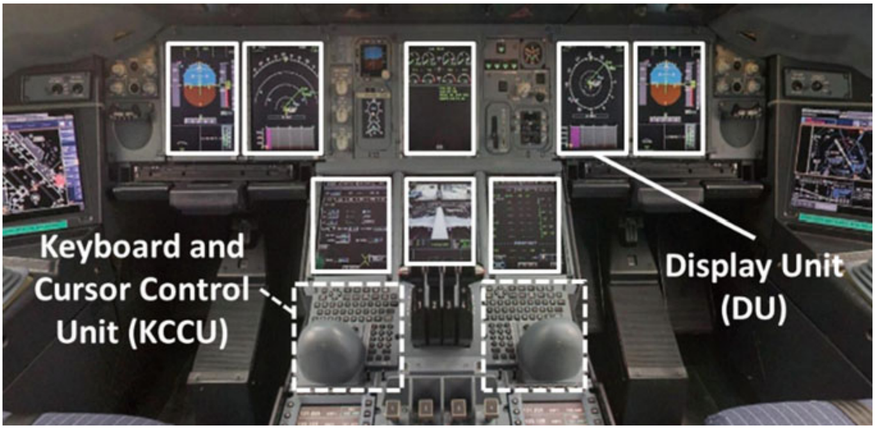


Fig. 4.6 Airbus A380 interactive cockpit

This section first presents an example of an interactive application within interactive cockpits as defined by the Flight Control Unit Software (FCUS). The interactive cockpit architecture (instantiated within the FCUS application) is then detailed, followed by a description of the tasks that have to be performed by the pilots when using this application. Finally, the connection between this case study and formal approaches is discussed.

4.4.1 The FCUS Application

This case study focuses on a single application: the FCUS application (see Fig. 4.8). It is inspired by the FCU Backup application that is designed to allow the crew members to interact with the Auto-Pilot and to configure flying and navigation displays. It is composed of two interactive pages Electronic Flight Information System Control Panel (EFIS CP) (left-hand side of Fig. 4.7) and Auto Flight System Control Panel (AFS CP) (right-hand side of Fig. 4.7) and is displayed on two (one for each flying crew member) of the eight Display Units. The crew members can interact with the application via the Keyboard and Cursor Control Units.

Figure 4.8 details the interactive system architecture in the cockpit. In this figure, we present the display of the FCUS on a single Display Unit. The interactive system architecture in the cockpit and the interactive applications are based on ARINC 661 specification (Airlines Electronic Engineering Committee 2002) as this is the required standard in the area of interactive cockpits for large civil aircraft. More precisely, the ARINC 661 standard specifies firstly the communication protocol between the Control and Display System and the aircraft system and secondly the software interface of interactive objects (namely the widgets).

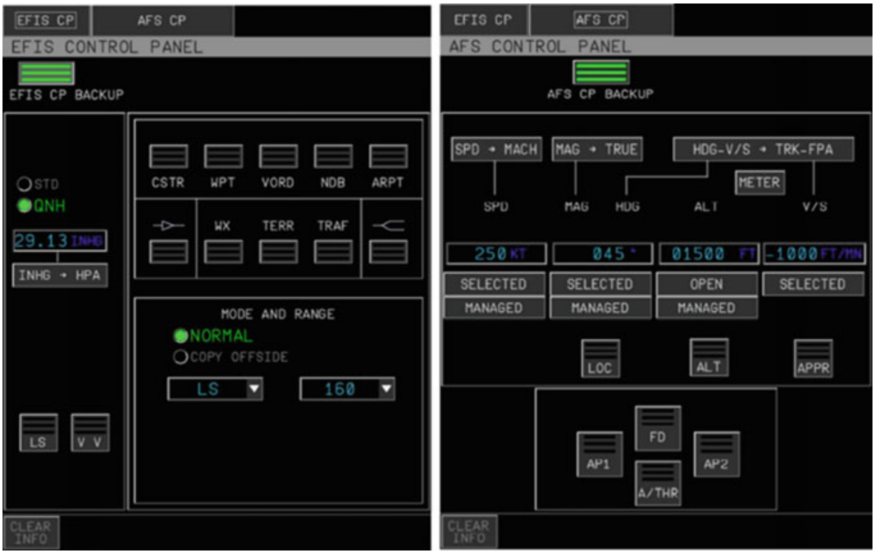


Fig. 4.7 Two windows of the FCUS application, inspired by the A380 FCU Backup

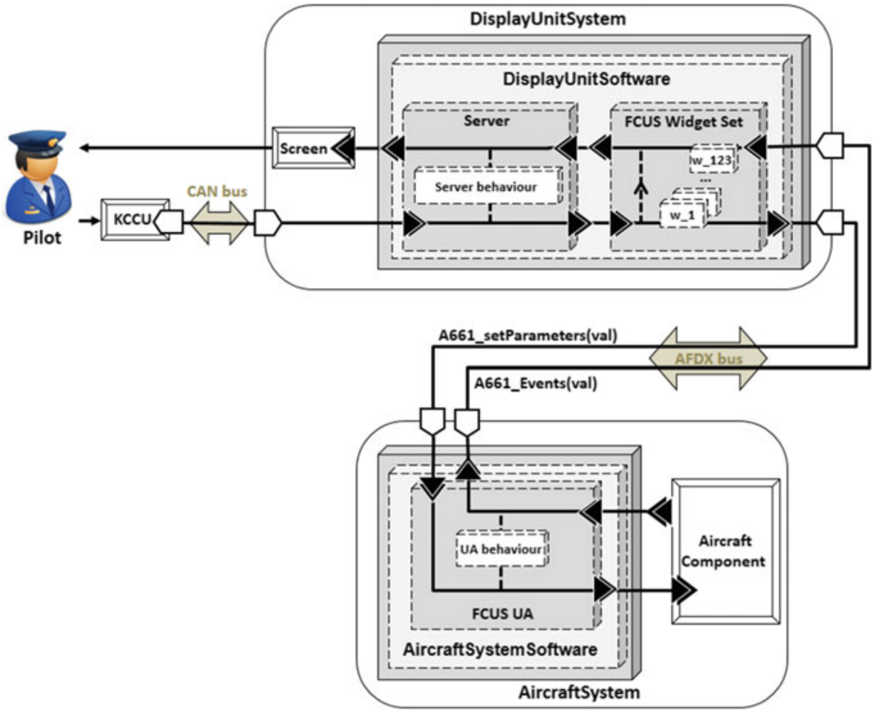


Fig. 4.8 Interactive cockpits architecture exemplified with the FCUS application

The Control and Display System is composed of the following components (as depicted in Fig. 4.8):

- ***FCUS widget set***: It is composed of a set of all the interactive graphical elements called widgets composing the FCUS application. The widgets are organized in a hierarchical way and correspond to the interactive facilities for the FCUS User Application. The hierarchical widget organization is managed by the server.
- ***Server***: It is responsible for the following: (i) the management of the Keyboard and Cursor Control Unit graphical cursor, (ii) the rendering of graphical information on the DU, (iii) the management of the widget hierarchy, and (iv) the dispatching of Keyboard and Cursor Control Units events to the targeted widgets (usually called picking in HCI). The server is also in charge of the management of several widget sets, each one corresponding to one User Application.
- ***Input and output devices***: Keyboard and Cursor Control Unit and LCD screen (Display Unit). They allow the interaction between crew members and the interactive system. They are related to software components (device drivers) that are not pictured here and that will not be considered in this paper.

As shown in Fig. 4.8, following the ARINC661 protocol, the Control and Display System provides information for the following:

- ***Crew members*** (***captain and first officer***): Their role is to fly the aircraft by (i) monitoring all aircraft systems through their associated displays (LCD screens) and (ii) controlling these systems through the associated input devices (Keyboard and Cursor Control Units). Their goal is to maintain the aircraft and to complete the mission, i.e., take the aircraft from the departing airport to its destination.

Aircraft systems: They are composed of two components: The first one called User Application (in this example, the FCUS) is the software interface to the physical aircraft component. The User Application is responsible for managing information to and from the Control and Display System: It processes the *A661_Events(val)* from the Control and Display System and triggers commands related to the physical aircraft components. The User Application can update the application display through the *A661_setParameters(val)* methods (applicable to the graphical widgets) to provide feedback to the crew about the actual state of the aircraft component. Graphically speaking, the User Application consists of one graphical user interface composed of a set of widgets stored in the Control and Display System. Its behavior defines the availability of widgets (enabled, visible, …) that are not presented here for the sake of brevity (more details can be found in Barboni et al. 2006). The flying crew interacts with this user interface to perform their operations by triggering commands and perceiving the status of the aircraft system.

4.4.2 Pilots Tasks

The tasks of the flying crew are rather complex covering both aircraft and mission management as explained above. Providing a full description of these tasks goes beyond the level of details of the description of these case studies. However, Fig. 4.9 provides a description of a subset of these tasks dedicated to the configuration of the barosettings (upper left part of Fig. 4.7). Before landing, crew members may be asked to configure the barometric pressure as relevant to the local airport context. The barometric pressure is used by the altimeter as an atmospheric pressure reference in order to process correctly the plane altitude. When the pilot is asked to enter a new value for the pressure reference, he/she first chooses the QNH mode (interactive input task "Click on QNH"). Then, he/she configures the pressure unit by choosing hPa (interactive input task "Click on InHg to hPa button") or InHg (interactive input task "Click on hPa to InHg button"). He/she can then choose to

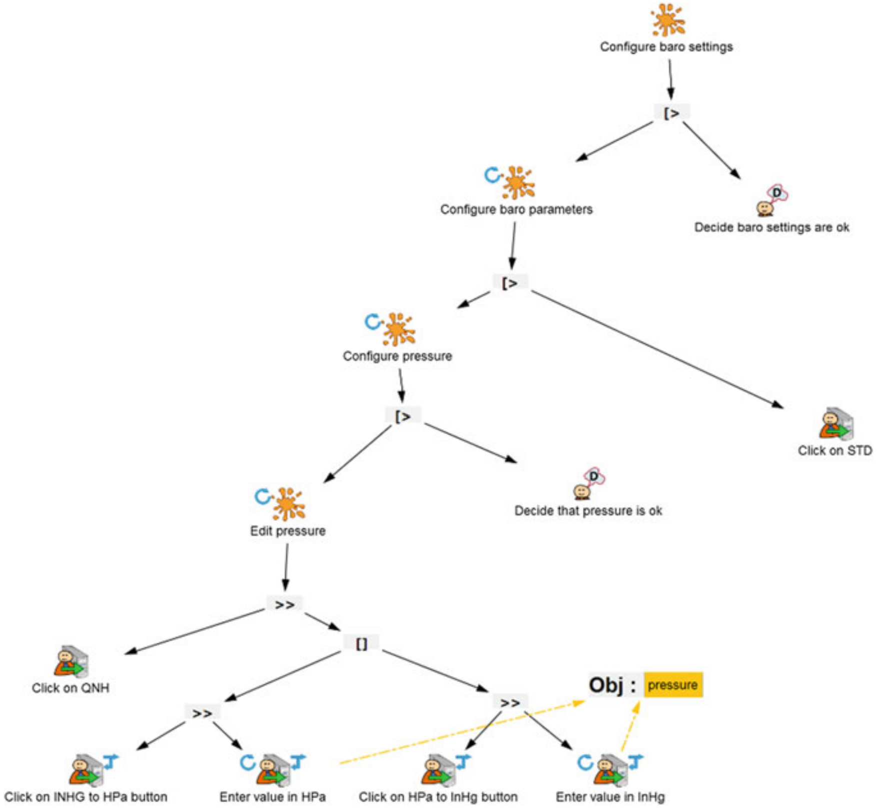


Fig. 4.9 Subset of the tasks of the flying crew interacting with the FCUS

edit the value in hPa (interactive input task “Enter value in hPa”) or in InHg (interactive input task “Enter value in InHg”). The STD mode (interactive input task “Click on STD”) is used during the cruise.

4.4.3 Connection with Formal Methods

This case study addresses various challenges for formal methods in HCI. First, it corresponds to an embedded interactive system used for controlling a safety-critical system. As this presents a public risk, this system must be conformant with certification requirements as expressed in DO-178C (RTCA and EUROCAE 2012) and CS 25 (EASA 2014). While addressing these certification needs is common in airborne software, HCI has so far remained out of the loop and certification of interactive systems remains an agenda for the research community.

On the contrary, the interaction techniques involved in the interactive cockpit context remain rather simple and follow the transposed IBM CUA 89 (IBM 1989) WIMP standard to interactive cockpit (called ARINC 661 specification Airlines Electronic Engineering Committee 2002). However, it is important to note that the presence of two graphical input devices (the KCCU of the captain and the one of the first officer) brings the issue of multi-modal interaction (similar to bimanual interactions for a single user Bier et al. 1993).

Lastly, the requirements for dependability of the interactive cockpits require taking into account the possible failures of hardware and software calling for reconfiguration mechanisms (of the interactive system—moving required information from a faulty display to functioning one). The same holds for interaction technique reconfiguration in case of input devices failures (Navarre et al. 2008).

4.5 Case Study 4—Interactive Systems in Rural Areas—Maintenance of Wind Turbines

Wind turbines as a technical facility to produce electrical energy from wind get more and more important in times of depleting fossil sources and reduced acceptability of atomic energy within the community. Wind turbines are often colocated in huge groups, so-called wind parks, or can be single installations. They are almost exclusively found in rural areas. To keep the reliability of these installations high and reduce technical failures, regular maintenance is necessary. This is applied by trained technicians who follow specified processes offered by the producing company.[1] These involve access to technical information. This information

[1] http://archive.northsearegion.eu/files/repository/20120320111424_PC_Skills-Compendiuminmaintenance.pdf.

includes details of the inspected installation (such as previous maintenance and repairs), as well as the maintenance process. The latter specifies which regular repairs have to be applied to the installation, which inspections have to be done (such as visual inspections of technical parts or temperature inspection using infrared cameras) and what the documentation looks like.

Interactive systems are mainly used in this scenario for leading the maintenance process; providing general information for the maintenance- and process-specific details of the procedure; enabling documentation; and finally planning and applying the maintenance process. This last use comprises the plan of which wind turbines have to be inspected, where to find them, and how different technicians are to coordinate. The information collected during maintenance is further post-processed to report the costs, trigger further repair requests if the inspection reveals problems that could not be fixed during the maintenance, or schedule the next regular maintenance.

4.5.1 Tilley—A Community Wind Turbine

Tilley is a community-owned 900 kW Enercon E44 wind turbine installed in 2009 (TREL 2016). Power from Tilley is fed into the grid, and the income from this is used to fund various island community projects.

The island's electricity supply is connected to its neighboring island Coll via an undersea cable, and this in turn is connected with the larger island of Mull and from there to the mainland and National Electricity Grid (see Fig. 4.10). The island typically consumes between 1MW and 2MW, including the power for the "golf ball," the radar that serves civilian North Atlantic air traffic control.

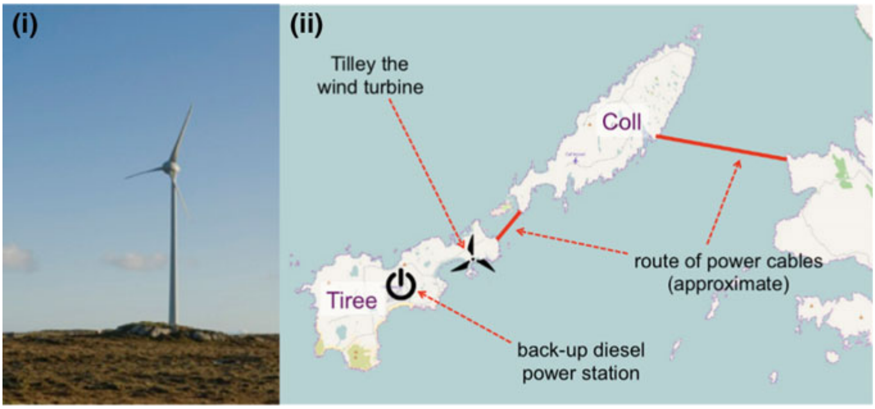

(i)
(ii)
Tilley the wind turbine
Coll
Tiree
route of power cables (approximate)
back-up diesel power station

Fig. 4.10 **a** Tilley the Tiree wind turbine. **b** Power generation and distribution on Coll and Tiree (map OpenStreetMap, CC BY-SA)

As Tilley can provide such a large proportion of the island's electricity, its correct functioning is particularly important as a malfunction could distort the island electricity supply. This is a particular issue when there are problems in the undersea cables and the island relies on the 3.5 MW diesel backup power station. At these times, Tilley has to be braked to "ticking over speeds" as the largely manually controlled power station cannot cope with the potential rapid changes in supply as the wind speed changes.

The large proportion of Tilley's operation is operated remotely from Germany using an industry standard SCADA (Supervisory Control And Data Acquisition) interface. The turbine includes an SMS telemetry device from which operational data can be downloaded and to which commands can be sent. However, there is also a small internal control area within the wind turbine tower for use during on-site maintenance or when the mobile signal fails (a common hazard at a remote location).

The two main control panels are shown in Figs. 4.11 and 4.12. It should be noted that this is by definition in an exposed location on the island, that the control area has no heating, and that the engineer may well have been climbing up further inside, or have come from the outside, potentially in a winter storm. All the controls must therefore be capable of being used with wet, cold, and potentially gloved hands.

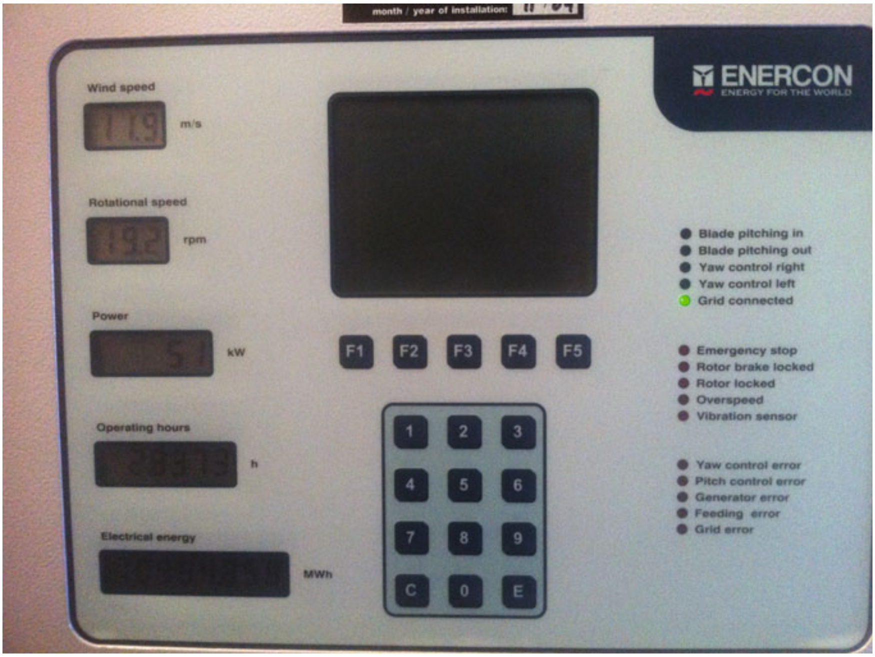


Fig. 4.11 Digital display and control panel in Tilley (photograph © William Simm)

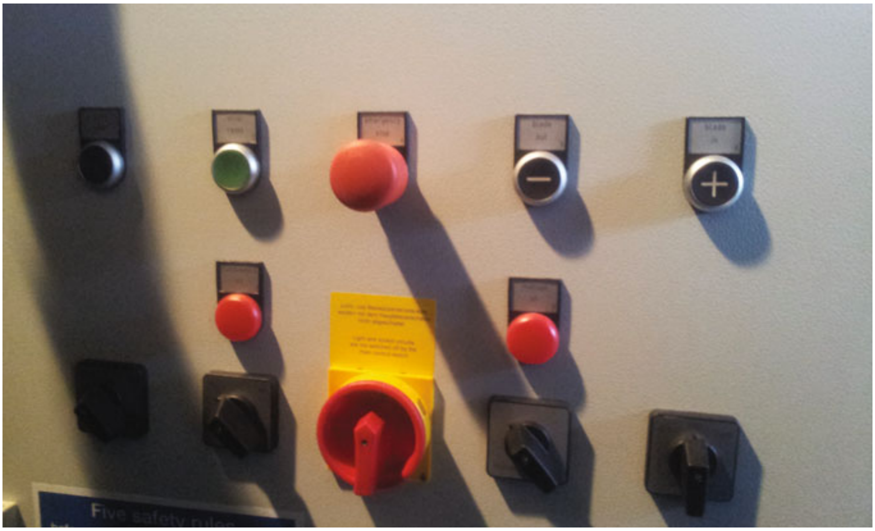


Fig. 4.12 Physical control panel in Tilley (photograph © Maria Angela Ferrario)

Figure 4.11 is the “digital” display and control panel. On the right it has status indicators (small LEDs) and on the left are numeric outputs for wind speed, rotation speed, power output, etc. Between these is an area with a numeric keypad and screen for entering more detailed parameters. Figure 4.12 is the “physical” panel, with large buttons and knobs including an emergency stop button, which is particularly important to be able to operate with gloved hands. Although these are described as “digital” and “physical,” the “digital” panel has physical buttons albeit of the flat membrane type.

4.5.2 Connection with Formal Methods

Formal methods can help to describe and analyze certain aspects of the user interfaces shown in Figs. 4.11 and 4.12. The buttons and display in the digital panel (Fig. 4.11) can be modeled using a range of dialoge-level techniques and standard properties verified such as the predictability of actions or reachability of the state (Dix et al. 2004). However, one of the key features of the turbine controls is their physical nature, in two senses. First, as with the previous three case studies, the majority of controls directly affect some physical process, such as the wind turbine angle of attack. The second is that the physical properties of the buttons on the control panel in Fig. 4.12 is crucial to their operation; for example, the emergency stop button is large enough to be easily pressed with gloved hands, but has enough resistance so that it is not activated accidentally. Chapter 9 of this book gives one example of analyzing the latter using physiograms, and another approach would be to use detailed force–displacement graphs (Zhou et al. 2014). Ideally, one would "unplug" the Tilley control panel to experiment with the feel of physical buttons, or experiment with the live panel to get the "feel" of the buttons, but Tilley is in constant use, so this was not possible for this case study. However, during the design of an interface such as this, detailed physical examination and specification would be possible.

Further work in formal description and analysis could focus on various other types of interactive systems and devices supporting maintenance work of wind turbines. For instance, mobile devices could be used to provide and to collect information in the field during the maintenance procedure, e.g., using the SMS-based communication interface of Tilley, or to implement a cooperative communication infrastructure between various technicians. Alternatively, augmented reality devices could be facilitated to offer installation and maintenance information while looking at the machines and user interfaces to be worked on. Thus, this case study addresses various challenges for formal methods in HCI. First, more than one mobile device can be involved as well as more than one user. Second, it has to be considered that asynchronous and synchronous communication patterns exist. Third, various aspects of information processing and presentation have to be considered: location, workflow, and unknown situations. Finally, the case studies address physical and technical interfaces that have to work in rough environment raising unusual requirements.

4.6 Case Study 5—Interactive Systems in Public Areas—Interactive Public Displays

Nowadays, large interactive displays can be found in various types of public areas either in urban or in rural spaces. Their main purpose is to provide information or to entertain people. More and more relevant are displays that are interactive or even persuasive. They react to a person's input, whether to provide personalized information or to change its content according to the context or the intended information provided. For these interactive displays, various interaction methods and concepts have been developed, such as gesture-based interaction, face detection, or other possible input devices. This interaction with the public display could involve not only a single person but also multiple persons, such as shown in the example of the "Domain Mall Interactive Display"[2] or the "The Magic Carpet" (Paradiso et al. 1997). Another class of public displays is presented by community information displays focusing on the information rather than on the entertainment aspect of public displays (Vogel et al. 2004) .

[2] https://www.youtube.com/watch?v=NRhDpDxTsLA.

4.6.1 Community Information Displays—The Internet-Enabled Shop-Open Sign

A community public display system on the Isle of Tiree has been produced as part of the biannual Tiree Tech Wave, a series of technology/maker meetings. The system is a long-term 24/7 deployment. At several Tiree Tech Wave events, islanders have talked with participants about different aspects of island life. Communication has often emerged as an issue, while the archetypal view of rural life supposes everyone knows everyone else's business; in fact with a widely distributed population, it can be harder to tell people about events and news than in a city.

This has led to a number of community communication interventions including an SMS broadcast system for youth work, a Web-based "Dashboard" (Fig. 4.13), and a public tickertape display in an island café (Fig. 4.14.).

The Dashboard and public display share a common data-oriented architecture (see Fig. 4.15). Raw data sources are gathered using a variety of techniques:

1. APIs and RSS feeds of public services (e.g., BBC local news),
2. scraping of Web-based information,
3. dedicated information entry interfaces (e.g., island events calendar), and
4. sensor data.

This raw data is then sampled in different ways to create parts of the data suitable for the different displays. This data is then passed to the final display device. In the case of the public display, this is converted into plain text for the tickertape, and in the case of the Dashboard, JSON formatted data to be passed into dedicated Dashboard "apps."

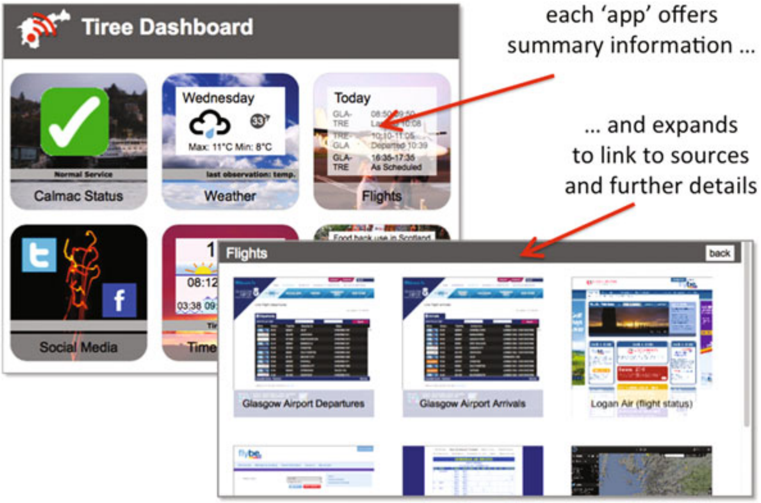


Fig. 4.13 TireeDashboard (http://tireetechwave.org/TireeDashboard)

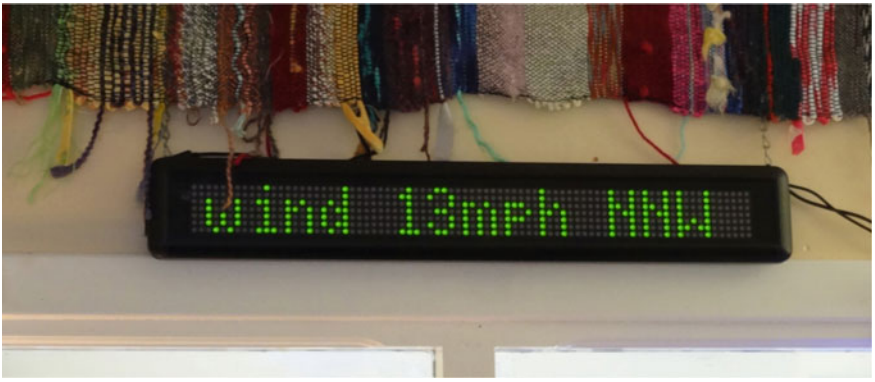


Fig. 4.14 LED public tickertape display

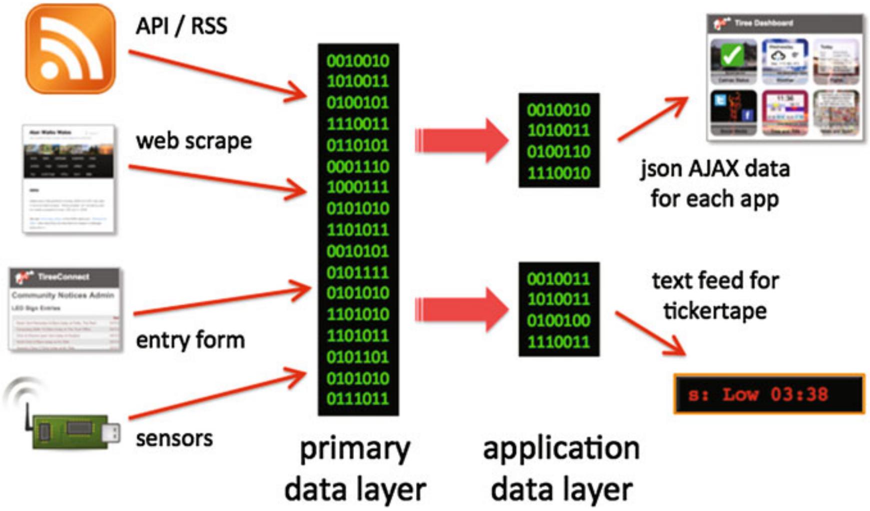


Fig. 4.15 Tiree data architecture

Much of the information is from existing public sources (1 and 2), and some created explicitly (3), and the sensor data (4). Data extracted from Web APIs and Web page scrapping includes weather, tide times, BBC News headlines, local Twitter feeds, and up-to-date travel status for plane and ferry. The last of these is particularly critical as the extreme island weather can often cause transport disruption. In addition, there are dedicated Web forms for the café and island community trust to add messages about upcoming events to compliment Web scrapes of the island "what's on" page.

Many public display projects are in large cities or institutions such as university campuses or airports. In these contexts, it is possible to have staff whose job includes monitoring and keeping information current. In a small community public display, it is crucial that there is a baseline of information that is current and valuable (hence regional news, weather, transport, etc.) so that displays are useful without additional effort. If displays, both Web-based and physical, are actively used, then this makes it more worthwhile for community members to add useful information and hence creates a virtuous spiral.

Live sensor data is used to adapt the information displayed. For example, the island data infrastructure includes an Internet-enabled “open” sign (see Fig. 4.16), which is installed in the Cobbled Cow café’ on the island. The sign constantly broadcasts its status to the Web data infrastructure, and this is used to display whether the shop is open or closed.

However, in general, it cannot be assumed that all sensor data is always reliable. Figure 4.17 shows an analysis of the tasks performed by the café owner during a typical day. Crucially, tasks 4.1 and 6.2, when the sign is turned on and off, are performed as a matter of routine, but could be forgotten, and it is physically possible to open the shop without turning on the sign. Similarly, it is possible to close the shop without turning off the sign, although this is likely to be noticed as the glow of the sign would be visible except in high summer. These issues are described in greater detail in Chap. 7.

The final part of the island public display infrastructure is a projected tabletop display (Dix et al. 2016). So far, this has not been included as part of the data architecture and instead used for stand-alone projects including the launch of Frasan (Fig. 4.18), the island mobile heritage app (Dix 2013). The display consists of an LCD projector and Kinect-style depth camera; these are mounted in a custom-designed enclosure fixed to the ceiling (Fig. 4.19). The enclosure rotates so that the display can be used on either the tabletop (also custom designed for the space) or the wall. The system was designed to enable multi-user, multi-touch interactions

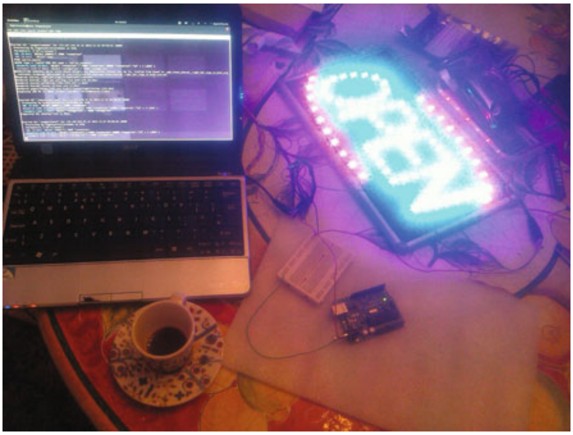
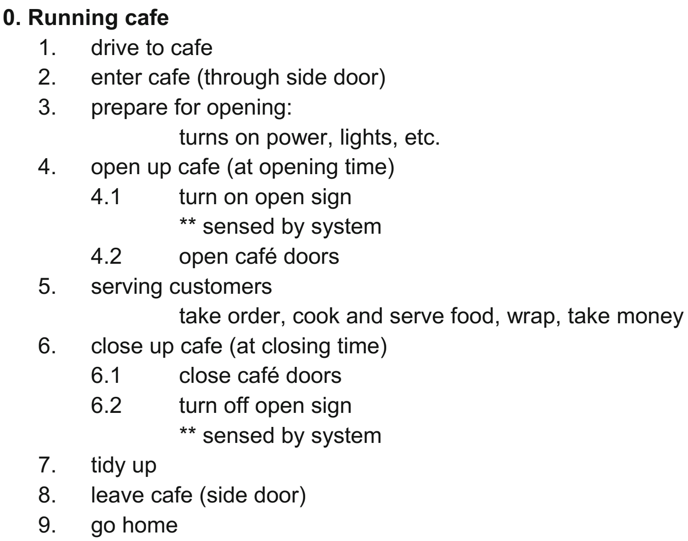


Fig. 4.16 (*Left*) Internet-enabled open sign under development (photograph Rory Gianni); (*right*) electric imp module (photograph http://www.electricimp.com media resources)



0. Running cafe

1. drive to cafe
2. enter cafe (through side door)
3. prepare for opening:
    turns on power, lights, etc.
4. open up cafe (at opening time)
    4.1 turn on open sign
        ** sensed by system
    4.2 open café doors
5. serving customers
    take order, cook and serve food, wrap, take money
6. close up cafe (at closing time)
    6.1 close café doors
    6.2 turn off open sign
        ** sensed by system
7. tidy up
8. leave cafe (side door)
9. go home

Fig. 4.17 Task analysis Cobbled Cow café

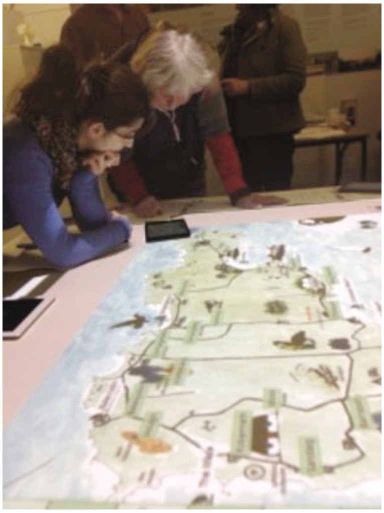


Fig. 4.18 Tabletop display in use at launch of Frasan mobile heritage app

(Bellucci et al. 2011), but could also be used for other depth-based installations, such as shifting sand piles to drive geographic simulations (Kirn 2011), and has already been used for BYOD (bring your own device) tangible interactions (Turchi et al. 2015).

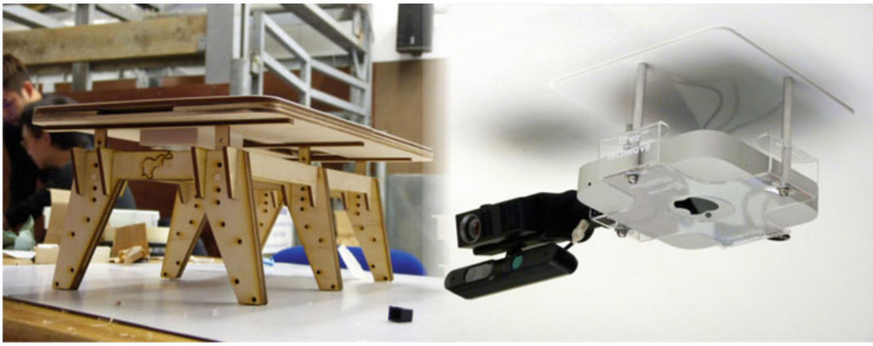


Fig. 4.19 Custom table design (*left*) and ceiling enclosure for projector and depth camera (*right*)

4.6.2 Connection with Formal Methods

The community information display case study is unusual in that insights from formal analysis directly influenced the design of some elements; this is described in detail in Chap. 7. There are also particular issues for formal modeling and analysis that are highlighted by this case study.

The island environment has many technical and social constraints that are different from those in city-based displays, formal descriptions of these could help designers who are not familiar with the context of use. The shop sign showed the importance of understanding these rich contexts and the pitfalls that may arise.

The complex environment also means that a broad range of types of display are included in the island deployment: the simple on–off open sign, the non-interactive LED ticker tape with short text messages, the interactive touch table, and Web-based dashboard. This range is wider than typically considered by standard tools such as responsive designs, ranging from tangible to non-interactive, and graphical to textual displays. Model-based formal methods could help understand the way information is mapped in different ways to these radical different devices. The range of data sources as well as display modalities suggests the potential for rich information models that take into account different timeliness, locality, and dependability of information flows.

A variety of further challenges and research questions can be identified for the use of formal methods in context of public displays in general. First, the design and description of interaction methods for public displays raises various challenges. As the needed interaction methods are not WIMP based as classic input devices are not available, e.g., mouse and keyboard, methods are needed that range from gesture-based interaction to 3D user interfaces or speech-based interaction to only mention a few. Second, both the presented content and the used interaction methods could or should be adaptable to the individual user and the context of use. In this case, the user has to be assumed as completely unknown and the context as completely uncontrolled. Here, the question is how far formal methods could support the gathering process of information to generate user models, to describe such models, and to support the validation of such systems in use. The validation could be of value during runtime as well as beforehand or after the usage phase. The same is true for the context; thus, how far, formal methods enable the description and handling of the dynamic context and the emerging conditions. As public displays influence public areas, it is relevant to investigate the newly created sociotechnical system. This finally is an aspect less discussed in the context of formal methods' research and offers a new perspective in research on formal methods in HCI. The main challenge and opportunity in this regard is the modeling and analysis of these systems using formal methods.

References

Airlines Electronic Engineering Committee (2002) ARINC Specification 661—Cockpit display system interfaces to user systems. Published: April 22, 2002

Barboni E, Conversy S, Navarre D, Palanque PA (2006) Model-based engineering of widgets, user applications and servers compliant with ARINC 661 specification. In: Design specification and verification of interactive systems. Springer, pp 25–38

Bellucci A, Malizia A, Aedo I (2011) TESIS: turn every surface into an interactive surface. In: Proceedings of the ACM international conference on interactive tabletops and surfaces

Bier EA, Stone MC, Pier K, Buxton W, DeRose TD (1993) Toolglass and magic lenses: the see-through interface. In: Proceedings of the 20th annual conference on computer graphics and interactive techniques. ACM, pp 73–80

Dix A, Finlay J, Abowd G, Beale R (2004) Dialogue notations and design. Chapter 16 in Human–computer interaction, 3rd edn. Prentice Hall, Englewood Cliffs. http://www.hcibook.com/e3/chapters/ch16

Dix A (2013) Mental geography, Wonky maps and a long way ahead. GeoHCI, Work-shop on geography and HCI, CHI 2013. http://alandix.com/academic/papers/GeoHCI2013/

Dix A, Malizia A, Turchi T, Gill S, Loudon G, Morris R, Chamberlain A, Bellucci A (2016) Rich digital collaborations in a small rural community. In: Anslow C et al (eds) Chapter 20 in Collaboration meets interactive spaces. Springer. doi:10.1007/978-3-319-45853-3_20

EASA (2014) CS-25. Certification specifications and acceptable means of compliance for large aeroplanes

EUROCONTROL (2010) Arrival manager implementation GUIDELINES and lessons learned, edition 0.1

Forbrig P, Martinie C, Palanque PA, Winckler M, Fahssi R (2014) Rapid task-models development using sub-models, sub-routines and generic components. In: Proceedings of 5th international conference on human-centered software engineering. Springer, pp 144–163

Gesellschaft für Anlagen- und Reaktorsicherheit (GRS) gGmbH (2016) Siedewasserreaktor (SWR) (in german). http://www.grs.de/aktuelles/begriff-des-monats-siedewasserreaktor-swr. Accessed 27 Sep 2016

IBM (1989) Common user access: advanced interface design guide. Document SC26-4582-0

Kirn P (2011) In sand and pixels, playing with worlds virtual and tangible; built with kinect. Create digital media. http://cdm.link/2011/07/in-sand-and-pixels-playing-with-worlds-virtual-and-tangible-built-with-kinect/. Accessed 7 July 2011

Martinie C, Barboni E, Navarre D, Palanque PA, Fahssi R, Poupart E, Cubero-Castan E (2014) Multi-models-based engineering of collaborative systems: application to collision avoidance operations for spacecraft. In: Proceedings of the 6th ACM SIGCHI symposium on engineering interactive systems. ACM, pp 85–94

Martinie C, Palanque PA, Winckler M (2011) Structuring and composition mechanisms to address scalability issues in task models. In: Proceedings of INTERACT. Springer, pp 589–609

Navarre D, Palanque PA, Basnyat S (2008) A formal approach for user interaction reconfiguration of safety critical interactive systems. In: Proceedings of 27th international conference on computer safety, reliability and security. Springer, pp 373–386

O'Regan G (2014) Z formal specification language. In: Introduction to software quality. Springer, pp 311–325

Owre S, Rushby JM, Shankar N (1992) PVS: a prototype verification system. In: International conference on automated deduction. Springer, pp 748–752

Paradiso J, Abler C, Hsiao KY, Reynolds M (1997) The magic carpet: physical sensing for immersive environments. In: CHI'97 extended abstracts on human factors in computing systems. ACM, pp 277–278

RTCA and EUROCAE (2012) DO-178C/ ED-12C, software considerations in airborne systems and equipment certification

Turchi T, Malizia A, Dix A (2015) Fostering the adoption of pervasive displays in public spaces using tangible end-user programming. In: Proceedings of IEEE symposium on visual languages and human-centric computing, pp 37–45

TREL (2016) Tilley, our turbine. Tiree renewable energy limited. http://www.tireerenewableenergy.co.uk/index.php/tilley-our-turbine/. Accessed 24 April 2016

United States Nuclear Regulation Commission (2016) Boiling water reactors. http://www.nrc.gov/reactors/bwrs.html. Accessed 27 Sep 2016

Vogel D, Balakrishnan R (2004) Interactive public ambient displays: transitioning from implicit to explicit, public to personal, interaction with multiple users. In: Proceedings of the 17th annual ACM symposium on user interface software and technology. ACM, pp 137–146

Zhou W, Reisinger J, Peer A, Hirche S (2014) Interaction-based dynamic measurement of haptic characteristics of control elements. In: Auvray M, Duriez C (eds) Haptics: neuroscience, devices, modeling, and applications: Proceedings of the 9th international conference, Eurohaptics 2014, Versailles, France, June 24–26, 2014, Part I. Springer, Berlin, pp 177–184. doi:10.1007/978-3-662-44193-0_23

Part II
Modeling, Execution and Simulation

Chapter 5
Visual and Formal Modeling of Modularized and Executable User Interface Models

Benjamin Weyers

Abstract This chapter presents the visual Formal Interaction Logic Language (FILL) for the description of executable user interface models. This modeling approach is based on an architecture called 3LA, which separates a user interface from its outward appearance as a set of interaction elements called physical representation, and its functional part called interaction logic, which connects the physical representation with the system to be controlled. The latter refers in general terms to a business model of an interactive system, where the interaction logic is further meant to describe dialog-specific aspects, e.g., the availability of widgets depending on the system state or other widgets. FILL is algorithmically transformed to reference nets, a special type of Petri nets, which makes it executable and equips FILL with formal semantics through the formal semantics of reference nets. FILL models are modularization which enables the description of multidevice user interface models as well as the reuse and exchange of parts of the model. For the creation and execution of FILL-based user interface models, the UIEditor tool is presented. It offers editors for modeling the physical representation and the modularized FILL-based interaction logic. It further implements the transformation algorithm for generating reference nets out of a given FILL model and finally the execution environment for the user interface model. The applicability of this modeling approach will be shown by means of a case study for the control of a simplified nuclear power plant. The chapter will conclude with a broader view to future challenges this approach is facing.

5.1 Introduction

Formal methods are a well-known class of approaches used in human–computer interaction research and applications. The major benefits of formal modeling meth-

B. Weyers (✉)
Visual Computing Institute—Virtual Reality & Immersive Visualization, RWTH Aachen University, Aachen, Germany
e-mail: weyers@vr.rwth-aachen.de

B. Weyers et al. (eds.), *The Handbook of Formal Methods in Human-Computer Interaction*, Human-Computer Interaction Series,
DOI 10.1007/978-3-319-51838-1_5

ods are the application of formal verification and analysis methods as well as formal reconfiguration and adaption mechanisms and their description in machine-readable and, in certain cases, executable form. Nevertheless, for general modeling tasks in engineering interactive systems, formal methods are often considered too complex to learn and too inflexible to perform tasks such as fast prototyping (Dix 1991). Their complexity can be reduced by using visual modeling languages, which are easier to learn than textual description concepts (Gaines 1991). A well-known class of formal modeling languages that includes a visual representation and has been used in modeling interactive systems is Petri nets (de Rosis et al. 1998; Jensen and Rozenberg 2012; Navarre et al. 2009; Petri 1962). Petri nets are supported by a visual representation for modeling and a broad variety of simulators, which makes them well suited for interactive systems engineering. However, Petri nets were not specifically developed to model user interfaces, and their use requires a deep understanding of their nature and function. Furthermore, they have to be adapted to the specific purpose of the application domain in question—in this case, interactive systems. To overcome these limitations, a domain specific and formal description must be compatible with the powerful concept of Petri nets but at the same time more specific to the application domain (here, the description of user interfaces) by addressing its specific requirements.

To specify these requirements in modeling user interfaces, it is crucial to take the user's perspective. According to research in human factors on solving tasks (John and Kieras 1996; Rasmussen 1983) and on user and task modeling (Paternò et al. 1997), users follow a hierarchical task decomposition first by means of the task itself and then by means of their mental model of the system to be controlled. The latter is significantly influenced and defined by the functionality of the system provided through a user interface. By providing this functionality, the user interface abstracts from the underlying system, mapping (ideally) the user's needs and expectations. Thus, the underlying system implementation can be treated as a black box and be represented only by a well-defined interface that provides access to its internal state and control variables. Thus, a formal description of user interfaces must not only provide a definition of its surface (buttons, sliders, etc.) but also specify a functional abstraction of the underlying system. Taking into account the technical perspective on this functional abstraction layer reveals a possible interpretation of it as an event-driven data processing layer. Events initiated by the user (e.g., pressing a button) or by the system (e.g., emerging from system state changes) are processed either by generating input to the system interface or by extracting information to be presented to the user. This motivates a domain-specific modeling approach following a process-oriented handling of events (represented below as data objects), which represents the intended functionality through interaction elements presented to the user. In summary, to overcome the aforementioned challenges of formal methods and simultaneously address the boundary conditions presented, a formal modeling approach for user interfaces should

(R1) be formally specified,
(R2) have a (simple-to-learn domain-specific) visual representation,
(R3) be flexible and executable,

(R4) be adapted to the modeling of the functional abstraction layer of a user interface model, and

(R5) follow the concept of a process-oriented handling of events.

Formal Interaction Logic Language (FILL) is a formal modeling approach that addresses these requirements. FILL is a modeling approach for the description of interactive systems with a focus on the modeling of user interfaces. Its goal is, on the one hand, to utilize concepts from object-oriented software development for the formal specification of user interfaces and, on the other hand, to offer a formal reconfiguration concept for the adaptation of existing user interfaces. The latter is the topic of Chap. 10 and will not be further discussed in this chapter.

FILL is intended to describe user interfaces by combining the advantages of a visual modeling language with the advantages of formal methods, such as verification and validation capabilities. Furthermore, a FILL model is executable, which makes the modeled interactive system directly applicable without further implementation efforts. This makes FILL highly suitable for uses such as the prototyping of interactive systems. FILL is based on the concept of component-based modeling, which prevents FILL-based models from being monolithic and makes components reusable and interchangeable. FILL's component-based modeling makes it possible to cope with well-known modeling strategies for engineering interactive systems, such as the need for separate business logic and dialog models. Components in FILL specify functional units with well-defined interfaces, which differs from other definitions in software research, such as those presented in Szyperski (1997). Thus, the term components conforms with how FILL is used—for example, on a more general meta level, in the context of UML component diagrams.[1] Finally, FILL addresses the description of interactive systems using post-WIMP interaction (Beaudouin-Lafon 2000; Van Dam 1997) as well as multidevice user interfaces (Berti et al. 2004). This is accomplished by abstracting FILL from specific types of devices or interaction methodologies, such as WIMP or gesture interaction.

5.2 Overview and Terminology

FILL is equipped with features to describe component-based models of *interaction logic*, as discussed in greater detail in Sect. 5.5. The term interaction logic refers to the aforementioned abstraction layer in a user interface model with functionality that describes the processing of events originating with the physical representation of the user interface or in the underlying (black box) system to be controlled. For these events to be processed, each is mapped to a data object. The *physical representation* is the surface of the user interface—that is, the part of the interface with which the user directly interacts. A physical representation can be defined as a set of interaction elements that are built into a device. *Interaction elements* in this context can be understood as WIMP widgets as well as general artifacts, such as gestures. (These aspects will be detailed in Sect. 5.5.) The system to be controlled by the user

[1] http://agilemodeling.com/artifacts/componentDiagram.htm.

interface is considered a black box system offering a well-defined interface, such as a set of input and output values, also referred to as the *system interface*. Physical representation, interaction logic, and system interface constitute a three-layer architecture (referred to below as *3LA*), which is discussed in detail in Sect. 5.4.

FILL is formalized by (a) an accompanying formal definition of FILL as a graph-based visual language (see Sect. 5.5.1 and Definition 1) and (b) an algorithmic transformation (see Sect. 5.5.3) to reference nets (Kummer 2009), which are a special type of colored Petri nets. The latter provides formal semantics to represent FILL and makes FILL models executable using the companion simulation engine called Renew (Kummer et al. 2004). Reference nets further offer mechanisms that enable the component-based structure of a FILL model to be mapped onto its reference net-based representation, which is discussed in detail in Sect. 5.5.3.1. This mechanism can couple a reference net-based representation of interaction logic to an underlying system.

In conclusion, FILL is a modeling approach based on 3LA (R4) and comprises

- a formal syntax definition (R1),
- a visual modeling language (R2),
- a process-oriented, data-based event handling mechanism (R5),
- a transformation algorithm that transforms a FILL-based user interface model to reference nets, offers formal semantics (R2), and allows model execution (R3),
- a reconfiguration mechanism (R3) based on formal graph rewriting (discussed in Chap. 10), and
- an interactive tool called UIEditor, which is capable of creating, transforming, and executing FILL models (R2, R3).

Thus, FILL meets all the requirements derived above and offers a new domain-specific modeling approach to user interface creation that simplifies the use of formal methods in interactive systems engineering. It also enables the formal adaption of user interfaces, which is implemented as part of the software tool UIEditor.

The following section will introduce related work (Sect. 5.3), highlighting distinct differences between FILL and other modeling approaches, such as BPMN (Chinosi and Trombetta 2012) or ICO (Navarre et al. 2009). Section 5.4 introduces FILL's architecture by comparing it to existing architectures, such as interactors (Duke et al. 1994) or the ARCH model (Bass et al. 1991). Section 5.5 introduces FILL and its transformation to reference nets. This is followed by a description of the UIEditor tool in Sect. 5.6. The applicability of this modeling approach will be demonstrated by means of a case study in Sect. 5.7. The chapter ends with a conclusion in Sect. 5.8.

5.3 Background

There are various formal methods for the description of user interfaces, some of which may be executable and/or are based on Petri nets. For instance, Navarre et al. (2009) present their Interactive Cooperative Objects (ICO) approach, which is based on Petri nets. Using their interpreter, PetShop (Sy et al. 2000), models of an interactive system can be directly executed and validated. Barboni et al. (2013) extended the ICO approach by adding a graphical user interface markup language called UsiXML (Limbourg et al. 2005) to define the physical representation. UsiXML is an XML-based user interface description language that offers a “multipath development” process, enabling the user to describe a user interface on different levels of abstraction based on the CAMELEON framework (Calvary et al. 2003; Florins and Vanderdonckt 2004). Still, UsiXML primarily defines the physical representation and only specifies which sort of functionality is connected to it without describing it explicitly. APEX is a system presented by Silva et al. (2014) that addresses similar goals for the prototyping and analyzing ubiquitous interactive systems by facilitating CPN Tools, a modeling tool for colored Petri nets as specified by Janssen et al. (1993). They present an approach that is conceptually close to the concept presented in this chapter but focuses on ubiquitous systems and does not address the need for an easy-to-use domain-specific visual modeling language like the one presented in this chapter. Further formal modeling approaches can be found, such as the Petri net-based approach described by de Rosis et al. (1998). However, none of these approaches facilitates the use of Petri nets as a modeling method as discussed in the Introduction.

The modeling approach introduced in this chapter focuses on the description of interaction logic as part of a user interface model. As described in the Introduction, the interaction logic describes a process-oriented handling of events, which are represented as data objects. Each type of event is handled by a specific process that deals with the generated data object by such means as converting or redirecting it. To describe processes, various visual languages exist, for instance, BPMN (Chinosi and Trombetta 2012) or UML diagrams such as *sequence diagrams* or *activity diagrams* (Kecher and Salvanos 2015). In general, UML is not equipped with a complete formal semantic definition. The UML standard[2] defines the syntax on inscriptions in the diagram formally using the Backus-Naur Form but keeps the semantic definition on the level of natural language. BPMN is a description language for business processes. It is not fully formally defined, but in the BPMN specification a partial mapping to BPEL is specified (see the OMG standard,[3] Sect. 14). BPEL is a formally defined XML-based notation for the description of executable business processes.[4] Unfortunately, BPEL is not a visual language as argued for in the Introduction. As BPMN is a visual modeling notation for business processes and not fully formalized, it is also not fully applicable for modeling interaction logic. However, BPMN offers concepts that are beneficial for describing interaction logic and have therefore been used in the specification of FILL (see Sect. 5.5). Because of its only partial formalization as well as its relation to business processes, BPMN was not entirely usable for modeling interaction logic. For instance, interaction logic is meant to handle typed data objects, whereas BPMN only describes general information entities without formally specifying the type of each entity.

[2] http://www.omg.org/spec/UML/2.5/PDF/.

[3] http://www.omg.org/spec/BPMN/2.0/PDF/.

[4] https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel.

Besides execution, formal methods permit the validation and verification of user interface models through model checking or other formal verification methods. Brat et al. (2013) discuss an approach using model checking to verify and validate formal descriptions of dialogs. This is a central interest in the modeling of user interfaces in safety critical situations, for example Barboni et al. (2010, 2015), Bastide et al. (2003). Furthermore, Paternò and Santoro (2001) discuss the use of formal verification in the investigation of multiuser interaction.

Various examples of the above-mentioned modeling languages (e.g., ICO) reveal the relevance of modularization addressing aspects such as encapsulation, exchangeability, or abstraction. Modularization has been also identified as beneficial in software engineering, especially in the description of interactive systems, as has been discussed by Bass et al. (1992), among others. Various architectures for interactive systems that follow this notion have been proposed, such as the ARCH model (Bass et al. 1991) and the SEEHEIM model (Pfaff 1985). The main objectives of these developments in the 1980s and 1990s were the reutilization of models or parts of them, the simplification of their evaluation, and the structuring of complex models to make them easier for modelers to understand. We argue that these requirements are no less important for the creation of formal models in user interface creation or, therefore, for the work presented in this chapter. These needs become even more important if multidevice systems are also to be modeled because their description increases the complexity of the resulting models.

Previous versions of FILL have already been presented (Weyers 2012, 2013a; Weyers and Luther 2010). These papers concentrated on defining FILL and its use without addressing multidevice or component-based user interface models in detail, which is the goal here. Furthermore, FILL has been used in various application domains. The papers Weyers et al. (2009, 2011) explain how FILL has been used in collaborative learning systems; students used FILL in a collaborative modeling task to describe cryptographic algorithms as an interaction logic model. In Weyers et al. (2011), it was shown that collaborative modeling supports learning of cryptographic algorithms. FILL has also been used to control technical systems. The papers Burkolter et al. (2014), Weyers et al. (2010, 2012) demonstrated that using FILL to individualize user interfaces reduces user errors as measured during the control of a simplified simulation of a nuclear power plant. Finally, Weyers (2013b) presented a concept for using FILL to model adaptive automation as part of the interaction logic in a user interface model.

5.4 Architecture

This section presents in detail the previously described architecture 3LA. It will illustrate the relation between this architecture and well-known architectural concepts and existing patterns. This description is meant to present and define the terms used and the role of FILL models in this approach and to show how FILL is used to create formal user interface models.

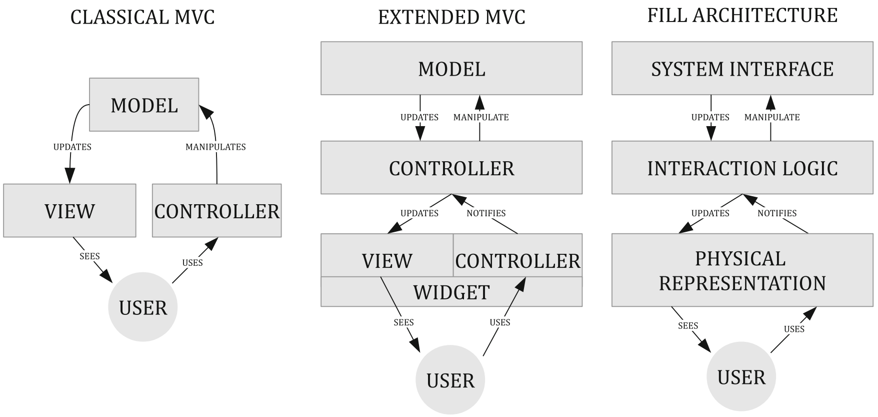


Fig. 5.1 Representation of the model–view–controller architectural pattern from three different perspectives: its classic definition, an extension that describes widgets with their own functionality, and a mapping of this extended version to the intended architecture of FILL

3LA is conceptually related to the model–view–controller (MVC) pattern (Krasner et al. 1988). Figure 5.1 compares the classic MVC pattern and a slightly restructured version (in the middle) that makes interaction elements into the functional elements of a user interface. Here, the MVC's controller is directly associated with the view of an interaction element which can trigger changes in the element's appearance and state; these changes are then encapsulated as an interaction element model. The interaction element controller further encapsulates all functionalities needed to generate events as data objects based on user interaction such as clicking on the representation of the interaction element. Thus, an interaction element can be modeled to conform to the MVC architectural pattern on its own regardless of the concept of interaction logic. Nevertheless, a user interface can be assumed to comprise more than one interaction element, which means that, on the interaction element level, the MVC exists *n* times for *n* interaction elements in one user interface. This set of interaction elements can be mapped to the physical representation layer.

The controller for noninteraction elements (also shown in the middle of Fig. 5.1) comprises all relevant functionalities for updating the (global) model representing the state of the controlled system and processes events emitted from the interaction elements (here, the individual controllers of the interaction element). This global controller relates to the interaction logic in FILL's terminology, where the global model can be associated with the system interface (see Fig. 5.1, right). FILL addresses the modeling of the interaction logic, where the combination of a physical representation description and the interaction logic is referred to as the *user interface model*.

The architecture specifies a macroscopic, horizontal, and layered representation of an interactive system. A microscopic (vertical) view of single-interaction elements, and their associated part of the interaction logic of a user interface model

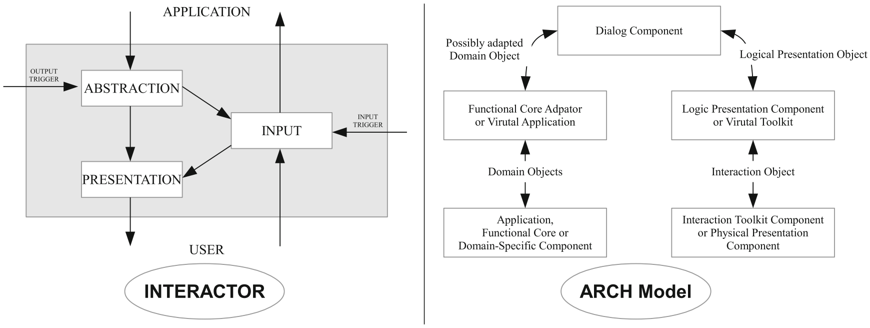

APPLICATION
OUTPUT TRIGGER
ABSTRACTION
INPUT
INPUT TRIGGER
PRESENTATION
USER
INTERACTOR
Dialog Component
Possibly adapted Domain Object
Logical Presentation Object
Functional Core Adaptor or Virutal Application
Logic Presentation Component or Virutal Toolkit
Domain Objects
Interaction Object
Application, Functional Core or Domain-Specific Component
Interaction Toolkit Component or Physical Presentation Component
ARCH Model

Fig. 5.2 Interactors as described by Paternò (1994), defining interaction objects as comprising an abstraction element processing output from outside of the interactor, an input element processing input from the user and a presentation element encapsulating the state of the outward appearance of the interactor. ARCH was introduced by Bass et al. (1991). It structures an interactive system in the components shown here, where the communication is defined by specific types of data objects

obtains a modeling structure similar to that presented by Duke et al. (1994), Paternò (1994), which is shown in Fig. 5.2. These so-called *interactors* not only describe interaction elements including a specific visual representation but also model multilayer networks of interactors through a hierarchical combination of inputs and outputs from interactors. Each interactor is related to a specific (elementary) task the user wants to work on with the system.

As can be seen in Fig. 5.2, an interactor comprises three main elements:

(a) the abstraction element, which contains the description of data to be visualized,
(b) the input element, which processes the input from the user to be redirected to the application or influence the third component, which is
(c) the representation element, which defines the appearance of the interaction object.

A combination of the two views presented (MVC and interactors) on user interface architectures can result in the 3LA presented in Fig. 5.3, which is the architecture used by FILL-based user interface models. As described above, interactors such as smaller components can be modeled (a widget combined with one or more FILL components) as can the more monolithic view of interaction logic as a controller. For the fine-grained structure of interaction logic, FILL offers mechanisms for structuring interaction logic into reusable and exchangeable components. A set of FILL components can be specified through various functional semantics, for instance, components that model the communication of interaction elements with the system to be controlled (as interactors do). FILL components can also be classified as interaction element-specific (also referred to as *interaction process*) if they directly relate to one interaction element or to a set of interaction elements (represented as vertical boxes in Fig. 5.3) or are only related to other components (also referred to as *global components*, represented as horizontal boxes in Fig. 5.3). Consider the ARCH model (Bass et al. 1991, 1992; Coutaz 2001), which specifies various functional components of

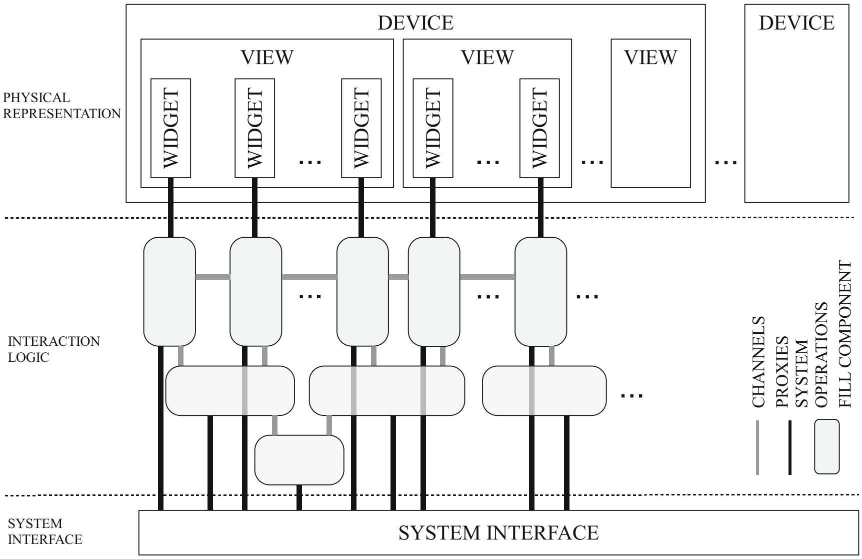


Fig. 5.3 FILL architecture 3LA has three layers: The physical representation, containing a set of interaction elements (widgets) used for direct interaction with the user; the component-based interaction logic, modeling the processing of data emerging from the physical representation or from the third layer; and the system interface, representing the system to be controlled as set of system values

different levels of abstraction (see Fig. 5.2, right). Similar to the proposed architecture, the ARCH model identifies the application or functional core as a set of domain-dependent concepts and functions located in the 3LA system layer. The other extreme in the ARCH model is the interaction toolkit component, or the physical representation component, which has the same functionality as the physical representation layer in 3LA. Both components are connected via three additional elements, whereby the dialog component is the central key element of the ARCH model. Its role can be defined as controlling task-sequencing. In 3LA, the dialog component can be modeled as one or more global components. The two other components in the ARCH model represent adaptors of the dialog component to the application (function core adaptors) and the interaction toolkit component (logical presentation component). The latter can be modeled as an interaction process and thus as a FILL component directly related to an interaction element and connected to a global component in the 3LA representing the dialog component. The function core adaptor can act as another global component in the 3LA related both to the global component representing the dialog component and to the underlying system. This exemplary mapping of 3LA to a well-established architecture demonstrates the applicability of 3LA following this architecture and also shows that 3LA is not limited to this architectural structure. An interaction process is still able to access the system directly, which makes it possible to implement simple user interface models. If the model becomes more complex, 3LA still offers the abstraction layers proposed by the ARCH model and thus enables the engineer to structure the user interface model accordingly. The next section will introduce the various modeling elements of FILL that enable the above-mentioned modeling of component-based interaction logic.

5.5 Modeling—Formalization

The following sections will introduce FILL as modeling language, following a bottom-up approach. In the first step, FILL's visual representation will be introduced by presenting the different types of modeling concepts used (Sect. 5.5.1). In the following, the component-based modeling will be introduced and how components are specified in a FILL model (Sect. 5.5.2). This description includes the modeling of multiview and multidevice user interfaces. These descriptions are followed by an introduction to the transformation of FILL models to a reference net (Sect. 5.5.3) (Kummer 2009), which is a specific type of colored Petri net (Jensen and Rozenberg 2012). Reference nets will be briefly introduced in Sect. 5.5.3.1. The whole modeling concept will be illustrated by means of a case study in Sect. 5.7 and by two smaller examples throughout this section.

5.5.1 Formal Interaction Logic Language—FILL

FILL has been developed as part of 3LA to describe interaction logic. As interaction logic connects the physical representation and the system to be controlled, it is defined to specify data flows that process emerging events from the physical representation or from the system layer. Events are represented by data objects; therefore, they can be handled by formally specified data processing. This data processing is described by a FILL model. Consider the following case: The user presses a button on a user interface. This creates a data object representing the press event. The goal of FILL is to model a data flow that processes this event and generates input data for the underlying system to be controlled. For example, it might specify the opening of a valve in a water flow system. Similarly, state changes in the system can be used to trigger processes in a FILL model, which are then propagated to the physical representation. Thus, FILL specifies event-driven processes for data handling.

5.5.1.1 Informal Introduction to FILL

Every FILL model is described as a graph composed of nodes as defined by FILL's visual representation, and is shown in Fig. 5.4. FILL defines various types of operation nodes. Nodes that set or read system values to or from the system to be controlled are called *system operations*. Nodes that transform data, such as events resulting

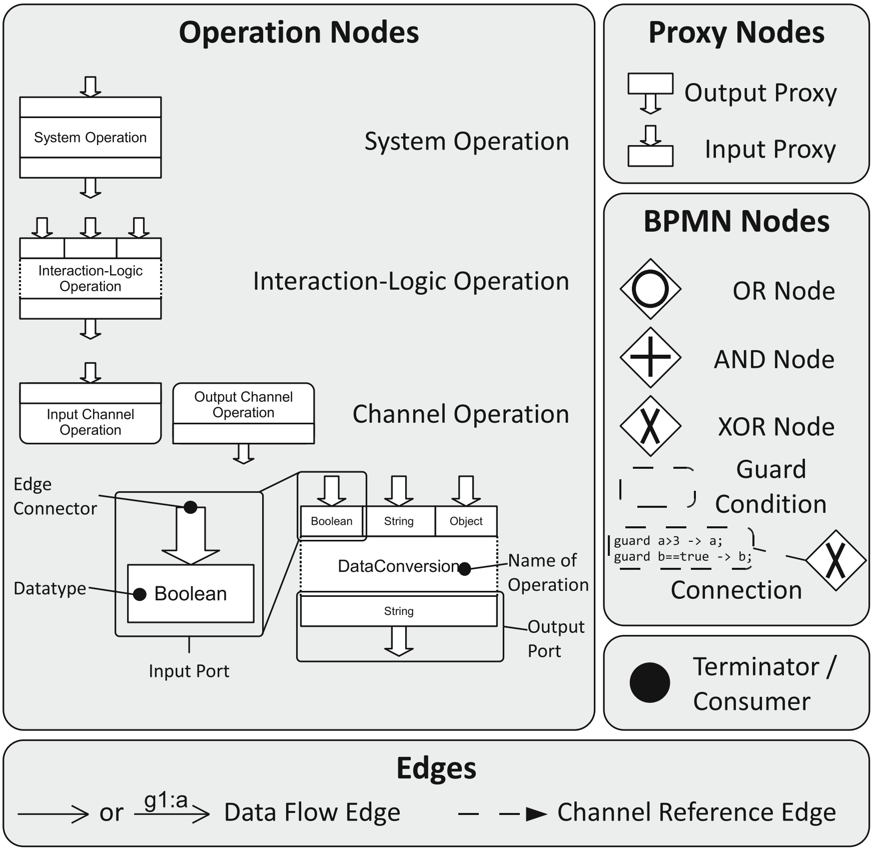


Fig. 5.4 FILL is a graph-based visual language composed of various types of nodes and edges: operation nodes (to process data), proxy nodes (to connect interaction elements to the FILL model), BPMN nodes (to branch and fuse FILL graphs), and two types of edges to define data flow or to couple FILL components

from user interaction, data generated by changes in the system state, or data stemming from other parts of the interaction logic are called *interaction-logic operations*. In Fig. 5.4, system operations are indicated by a solid frame, while interaction-logic operations are indicated by a dashed border. Nodes that connect different FILL components to each other are called *channel operations* and are indicated as gray lines between components in Fig. 5.3. Channel operations are indicated by having only one input or one output port representing the entry (input channel operation) or exit (output channel operation) point of a channel (indicated by a dashed arrow in Fig. 5.4). The concept of components will be described in detail in Sect. 5.5.2.

The ports of operation nodes define the connecting points of incoming or outgoing data objects and are visualized as thick arrows. Therefore, ports are identified according to which the type of data objects can be consumed (input port) or will be emitted (output port) into the FILL graph. All operation nodes and descriptions of the various elements (e.g., ports) of an operation node are shown on the left in Fig. 5.4. Furthermore, FILL defines a terminator, which consumes all incoming data objects and is used to define the sink of a FILL graph.

In addition to operation nodes, which cope with the processing of data objects and the structuring of FILL models, FILL borrows nodes from the BPMN modeling language (White and Miers 2008). BPMN was originally defined for the visual description of business process models. In FILL, nodes called *gateways* in BPMN are used to branch and fuse processes. They can be further equipped with guard conditions, which can be used to define conditions under which incoming data objects are redirected to outgoing connections. These conditions can be further used to control redirection; in the case of branching nodes, for example, gateways determine which outgoing connection will be instantiated with the incoming data object, and in the case of fusion, which of the incoming data objects will be redirected to the outgoing connection. Guard conditions are Boolean expressions based on propositional logic. For instance, consider two incoming data values a and b of type Integer at an AND BPMN node and the guard condition

$$guard\ a > 20 \,\&\, b < 30 \rightarrow a;$$

The part between the key word guard and the arrow sign → is evaluated as a Boolean expression. For instance, if $a = 30$ and $b = 20$, a is redirected to the next process of the BPMN node. The latter information is indicated by the arrow →. For the AND node, the guard condition is only evaluated if a and b are both mapped to a value by an incoming data object. The semantics of the BPMN nodes will be discussed in the context of transformation in Sect. 5.5.3.

To define data flow in a FILL graph, FILL nodes are connected via directed edges, which start and end at ports at the top and the bottom of an operation node or at BPMN nodes, as can be seen in Fig. 5.5 on the right. These so-called *data flow edges* can be further inscribed to make data objects that can be sent to BPMN nodes or referred to in guard conditions. Thus, data objects in the FILL graph can be treated as variables (see description of example in Fig. 5.5).

Another type of edge is defined for the representation of channels in a FILL model. These *channel reference edges* are used for sending data objects from one FILL component to another. They always begin at an input channel operation and end at an output channel operation. These edges are visualized in Fig. 5.4 as a dashed arrow indicating the direction.

Ports that are not part of operation nodes are called *proxy nodes* and represent connections to interaction elements in the physical representation. Output proxy nodes represent events that occur during run-time. If, for example, the user presses a button, an event object is created and sent to the FILL graph via the output proxy node related to the button. This event data object is then passed through the FILL graph along the data flow and channel reference edges and thus processed by operations. To send data objects back to interaction elements, they are passed to input proxy

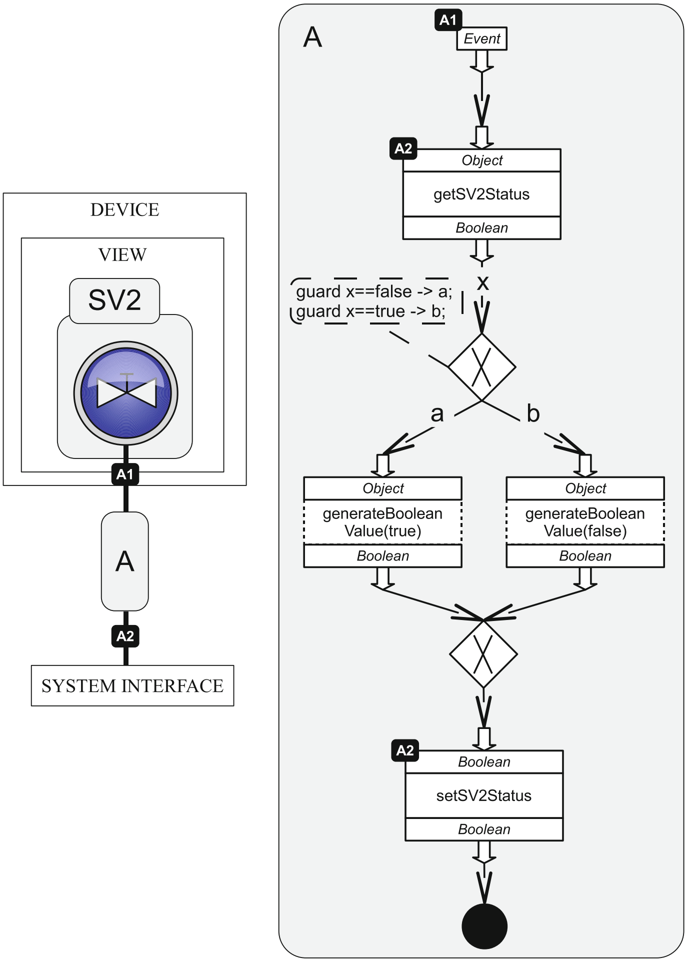


Fig. 5.5 Example of a simple FILL-based user interface model with a physical representation containing one interaction element (a button) related to a single FILL component (shown on the *right* in detail), which is related to an underlying system interface. The FILL graph models the toggling of the value of the system value SV2

nodes associated with interaction elements able to process the specified data type and present it to the user.

5.5.1.2 Formal Definition of FILL

Based on this informal introduction to FILL by means of its visual representation as shown in Fig. 5.4, FILL's syntax can be formally defined as follows:

Definition 1 The *Formal Interaction Logic Language* (FILL) is a 19-tuple

$$(S, I, C_I, C_O, P_I, P_O, X_I, X_O, B, T, P, E, l, g, c, t, \omega, \mathscr{L}, \mathscr{B}),$$

where S is a finite set of system operations and I is a finite set of interaction-logic operations; P_I and P_O are finite sets of input and output ports; X_I and X_O are finite sets of input and output proxies; C_I is a finite set of input channel operations; C_O is a finite set of output channel operations; T is a finite set of data types; B is a subset of BPMN nodes, with

$$B = \{\oplus,\ \otimes,\ \odot\}. \tag{5.1}$$

S, I, C_I, C_O, P_I, P_O, X_I, X_O, T, and B are pairwise disjoint. P is a finite set of pairs

$$\begin{aligned} P = \{(p,o) \mid p_I(p) = o\} \cup \{(p,o) \mid p_O(p) = o\} \cup \\ \{(p,o) | p'_I(p) = o\} \cup \{(p,o) | p'_O(p) = o\}, \end{aligned} \tag{5.2}$$

where $p_I : P_I \to S \cup I$ and $p_O : P_O \to S \cup I$ are functions with

$$\begin{aligned} \forall s \in S : (\exists_1 (p,s) \in P : p_I(p) = s) \wedge \\ (\exists_1 (p,s) \in P : p_O(p) = s),\ \textit{and} \end{aligned} \tag{5.3}$$

$$\forall i \in I : \exists_1 (p,i) \in P : p_O(p) = i, \tag{5.4}$$

and where $p'_I : P_I \to C_I$ and $p'_O : P_O \to C_O$ are functions with

$$\begin{aligned} \forall c \in C_I : (\exists_1 (p,c) \in P' : p'_I(p) = c) \wedge \\ (\nexists (p,c) \in P' : p'_O(p) = c),\ \textit{and} \end{aligned} \tag{5.5}$$

$$\begin{aligned} \forall c \in C_O : (\exists_1 (p,c) \in P' : p'_O(p) = c) \wedge \\ (\nexists (p,c) \in P' : p'_I(p) = c). \end{aligned} \tag{5.6}$$

E is a finite set of pairs, with

$$\begin{aligned} E = \{(p_O, p_I) \mid e(p_O) = p_I\} \cup \\ \{(p,b) \mid e'(p) = b,\ b \in B\} \cup \\ \{(b,p) \mid e'(b) = p,\ b \in B\}, \end{aligned} \tag{5.7}$$

where $e : P_O \cup X_O \to P_I \cup X_I \cup \{\omega\}$ is an injective function, ω is a terminator, and

$$\begin{aligned} \forall (p_O, p_I) \in E : (p_O \in X_O \Rightarrow p_I \notin X_I) \wedge \\ (p_I \in X_I \Rightarrow p_O \notin X_O), \end{aligned} \tag{5.8}$$

and where

$$e' : P_O \cup X_O \cup B \rightarrow P_I \cup X_I \cup B \cup \{\omega\} \tag{5.9}$$

is a function with

$$\begin{aligned} \forall b \in B : (\#\{(p,b)|(p,b) \in E'\} > 1 \Rightarrow \exists_1 (b,p) \in E') \\ \vee(\#\{(b,p)|(b,p) \in E'\} > 1 \Rightarrow \exists_1 (p,b) \in E'). \end{aligned} \tag{5.10}$$

l is a function with

$$l : E' \rightarrow \mathscr{L}, \tag{5.11}$$

where $\mathscr{L}$ is a set of labels.
g is a function with

$$g : B \rightarrow \mathscr{B}, \tag{5.12}$$

where $\mathscr{B}$ is a set of Boolean expressions, also called guard conditions or guard expressions.
c is a relation with

$$c : C_I \rightarrow C_O. \tag{5.13}$$

t is a total function with

$$t : (P_I \cup P_O \cup X_I \cup X_O) \rightarrow T. \tag{5.14}$$

The following example of a simple FILL graph will present the syntactic structure and semantics of a FILL graph in greater detail and complement the descriptions above.

5.5.1.3 FILL Example

This section examines how a button is used to control a valve in a simplified simulation of a nuclear power plant (cf. Chap. 4). The user interface model including the button and the accompanying FILL graph can be seen in Fig. 5.5. On the left is the user interface model from an architectural point of view based on 3LA presented in Sect. 5.4. On the right is one FILL graph, representing the user interface's interaction logic.

When the user presses the button, which is embedded in the physical representation of a classic WIMP-based device (e.g., in a window of a Windows desktop), an event object is sent to the FILL graph by the proxy node shown at the top of the graph. The proxy node is indicated in Fig. 5.5 by (1). This object is passed to a system operation called `getSV2Status`, where it is consumed and triggers a get function on a system value called `SV2Status`. In the simplified simulation of a nuclear power plant (cf. Chap. 4), this system value represents the status of a steam valve (steam valve 2). This system call as well as the data processing function call associated with interaction logic operations will be described in greater detail in Sect. 5.5.3.

The relation between system operations and the underlying system to be controlled is indicated by (2) in Fig. 5.5, which shows the system operation's representation of the system interface in the FILL model. The returned value—a Boolean value indicating whether steam valve 2 is open (true) or closed (false)—is sent to the FILL graph and passed to the succeeding (branching) BPMN `XOR` node. In this case, the guard condition specifies that if the incoming value `x` is `false`, the incoming value is redirected to the edge inscribed with `a`, whereas if `x` is `true`, it is redirected to edge `b`. In both cases, the redirected Boolean value triggers an interaction-logic operation called `createBooleanValue`, which generates a negated Boolean value. This new Boolean value is then passed via a fusing XOR node to a system operation that sets this new value to the same system value `SV2Status`. In this case, the `XOR` node just forwards every incoming data object to its outgoing branch. A terminator node finally consumes the value. Thus, the FILL graph describes the toggling of the state of steam valve 2 in the nuclear power plant simulation.

5.5.2 *Component-Based and Multidevice Models*

The creation of component-based, multidevice models is an extension of the basic FILL definition introduced above. This extension specifies three new types of entities for FILL, namely devices D, views V, and components C, which are used in the descriptions above. Devices represent a combination of software and hardware used as interaction gateways between user and interaction logic that simultaneously provide the physical representation of the modeled user interface. A smart phone, a headset, or a window on a Windows desktop are all examples of such devices. Views represent a specific part of the physical representation of a device, which allows the modeling of more structured physical representations. A device can have more than one view, which means that a physical representation can not only be separated into interaction elements but into views containing interaction elements. This enables the interaction logic model to specify a change between different views of a device. Every view is related to a specific type of physical representation, such as a classical GUI for a smart phone or a set of audio files for a headset. The connection of a FILL graph to the view-based physical representation is defined by proxy nodes. Therefore, Definition 2 specifies the function d, which matches every proxy node to exactly one view. A view can be matched to various proxy nodes since a view might comprise more than one interaction element.

5.5.2.1 Component-Based Extension of the Formal FILL Definition

A formal definition of this extension of FILL models to a component-based description concept can be specified as given below.

Definition 2 A modularized FILL model is a 7-tuple

$$(D, V, C, d, c, v, F),$$

where D is a finite set of devices, V is a finite set of views, C is a finite set of components, and F is a FILL graph. D, C, and all sets of F are pairwise disjoint.

v is a function, such that

$$v : V \to D. \tag{5.15}$$

d is an injective function, such that

$$d : (X_I \cup X_O) \to V, \tag{5.16}$$

where X_I and X_O are input and output proxies from F.
c is an injective function, such that

$$c : (S \cup I \cup C_I \cup C_O \cup X_I \cup X_O) \to C, \tag{5.17}$$

where S, I, C_I, C_O, X_I, and X_O are operation and proxy nodes of F.

However, the above definition does not exclude cases in which a component references proxies of different views. The UIEditor modeling tool (see Sect. 5.6) assumes that a single component is used for all proxies of an interaction element to simplify the visual modeling and prevent overly large graphs from being edited at once. Moreover, components can be modeled without being mapped to proxy nodes. Such components are referred to as *global components* because they are independent and can only be integrated by channel operations into other view-dependent components. This separation of two types of components (view-dependent and global FILL components) will have further implications for the transformation and execution of FILL models that will be discussed in greater detail in Sect. 5.5.3.

For the component-based description of the interaction logic, all operation and proxy nodes are matched to components (elements of set C). Nodes matched to different components should not be connected by data flow edges. This restriction keeps the components exchangeable and internally valid. For instance, by deleting a component from the interaction logic, a direct connection via a data flow edge can change the semantics of the remaining component or even make it nonfunctional. Channels allow a well-defined encapsulation of components to prevent structural dependencies of this nature. Therefore, channel operations should be used to connect components to one another. Channels can also be used internally in a component.

5.5.2.2 Component-Based FILL Example

The example presented in Sect. 5.5.1.3 (see Fig. 5.5) can be extended. As can be seen in Fig. 5.6, a new interaction element has been added: a lamp with four possible states—red/blinking (0), green/blinking (1), red/steady (2), and green/steady (3). The lamp is used if a new value for the steam valve is entered by pressing the button on

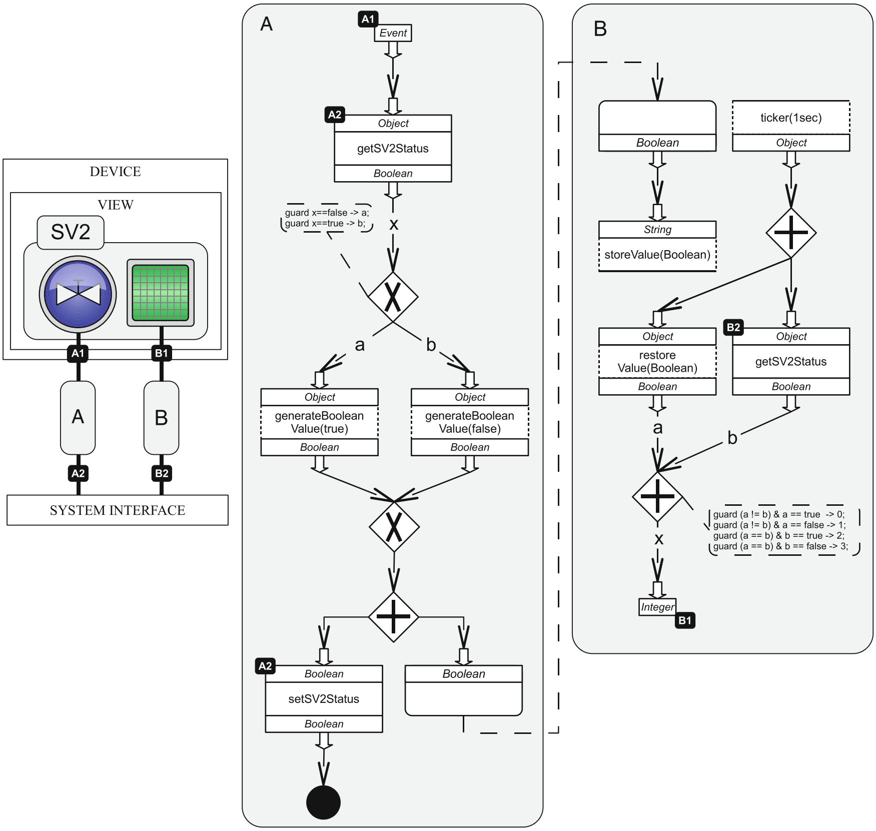


Fig. 5.6 Extended version of the user interface model presented in Fig. 5.5. The model offers a second interaction element (a lamp) associated with a second FILL component that controls the lamps' state. The two components are connected via a channel

the interface. Due to the delay in the system (time needed to mechanically open or close the valve), the new status is not directly applied. During this lag time, the lamp blinks.

To model this functionality, a new component (B) is added to the interaction logic model. The component (A) has been slightly extended, such that the selected value is transferred to component (B) via the channel between the input channel operation in component (A) and the output channel operation in component (B). The transferred value is stored in component (B) (interaction-logic operation `storeValue(Boolean)`) for multiple use. At intervals, an update of the lamp's status is triggered by an interaction-logic operation called `ticker`. This operation periodically generates a data object and pushes it into the FILL graph. This object is duplicated by the succeeding BPMN `AND` node and simultaneously triggers the restoration of the previously stored value and the current status of the valve. Finally, the guard conditions of the second AND node specify which output is generated based on the desired lamp state (0 to 3) in response to the combination of incoming values. The output is then transferred to the lamp via the input proxy node in component (B).

5.5.3 Transformation to Reference Nets

The next step in the modeling process is the transformation of FILL models into an executable representation. This transformation provides formal semantics for FILL. Before the transformation algorithm is presented in Sect. 5.5.3.2, a short introduction to reference nets will be given. Reference nets offer executability and define FILL's formal semantics.

5.5.3.1 Reference Nets

Reference nets were originally described by Kummer (2009) as an extension of classic-colored Petri nets. Reference nets support typed (in the sense of object-oriented data types) tokens and places as well as the inscription of places, transitions, and arcs in the net, including guard conditions for transitions. The latter are especially relevant for the transformation of FILL's BPMN nodes. The following introduction to reference nets assumes basic knowledge of colored Petri nets. For more information on these modeling concepts, please see Jensen and Rozenberg (2012), Priese (2008).

The main difference between reference nets and classic-colored Petri nets is the handling of tokens. Classic-colored Petri nets limit the existence of a token to a singleton. Reference nets lift this restriction by allowing multiple instances of a net, which can be bound to tokens as references on a net instance. For the synchronization of transitions, reference nets support synchronous channels. Synchronous channels synchronize the firing of two or more transitions. To fire, the pre-conditions for all bound transitions have to be fulfilled. If this is the case, the transitions are fired simultaneously. This feature set makes reference nets—compared to other colored Petri net formalisms—a very good candidate for transforming FILL models. By using net patterns and instances, a component-based description of a FILL model can be transferred into a reference net-based model, which preserves the FILL model's structure. This makes it possible to instantiate more than one device of a type meant to use the same interaction logic as has been modeled as a FILL-based description.

Figure 5.7 shows an example of a simple reference net. On the left are the so-called *net patterns*. These define the prototype of a net being instantiated during the reference net simulation. On the right is the situation after the creator net has been instantiated and the upper transition inscribed with the instantiation calls for the worker net has been fired. Two instances of the worker net pattern are created, each referenced by one token in the creator net instance. Both are shown at the bottom-right of Fig. 5.7. In the displayed status of the net, the transitions in the

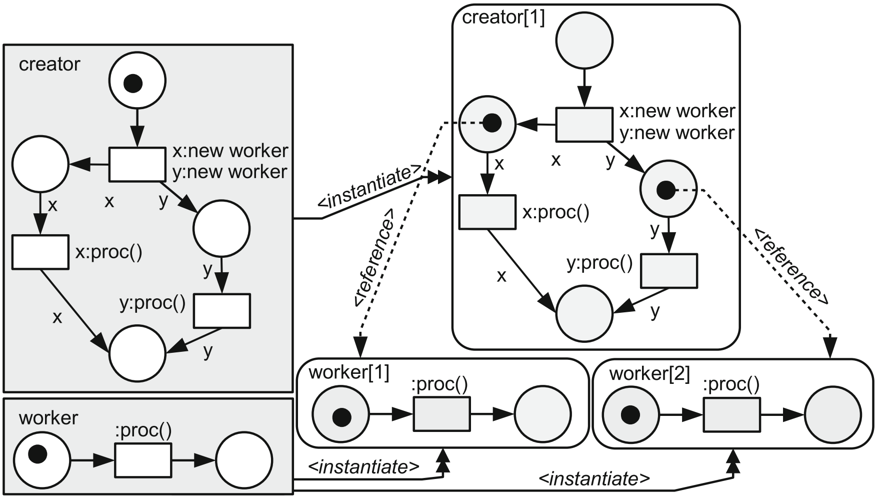


Fig. 5.7 Example of a reference net with two net patterns shown on the *left* and a snapshot of a running simulation of the net on the *right*. The example illustrates the features of reference nets by offering the instantiation of net patterns such that a token is not restricted to a singleton instantiation. The use of synchronous channels is illustrated

`creator` net instance inscribed with `x:proc()` and `y:proc()` are activated and can fire. If transition `x:proc()` is fired, the transitions in `worker[1]` inscribed with `:proc()` are fired at the same time because the inscriptions define a synchronous channel identified by the name `proc`. Synchronous channels are specified using a colon `:` between the name of the instance—here `x`—and the name of the channel—here `proc`. The result is that the token in the incoming place of transition `:proc()` in net `worker[1]` is consumed, and a token in the subsequent place in the same transition is generated.

5.5.3.2 Transformation

The transformation of a given FILL model comprises the following steps:

1. The complete model is examined according to its components and its physical representation (structure of devices and views).
2. The net pattern representing the `controlNet` is generated; this net pattern handles
   - the instantiation of all other net patterns, each representing one FILL component,
   - the FILL channel-based communication between these components, and
   - the management of multidevice user interfaces.
3. Every FILL component is transformed to a net pattern.

4. All net patterns are integrated into one reference net.

An initial examination is conducted to identify the relations between FILL components, views, devices, interaction elements, and the implementations used for the system interface and the interaction-logic operations as defined in the given user interface model. In the case of the UIEditor (see Sect. 5.6), the system interface as well as the interaction-logic operations each has to be implemented as a JAVA class. The language dependency is specific for the UIEditor but not for FILL. In a UIEditor-based implementation of FILL, every system and interaction-logic operation is mapped to one method in the corresponding JAVA class. Ids are used to resolve the mapping of components via channels (channel reference edges) and to interaction elements. Furthermore, ids simplify the control of data flow in the reference net and the algorithmic transformation. The following mapping functions are used in the transformation:

Definition 3 Assume a FILL graph FG. Being FG a graph, the functions f, κ, id, are defined as follows:

f is a function with

$$f : S \cup I \cup C_I \cup C_O \rightarrow \mathscr{F}, \tag{5.18}$$

where $\mathscr{F}$ is a set of function calls and S are system operations, I interaction-logic operations, and C_I and C_O are channel operations that originate from FG. For elements of S and I, $\mathscr{F}$ represent JAVA methods and for C_I and C_O channel names.

κ is a function, with

$$\kappa : X_I \cup X_O \rightarrow \mathscr{I}, \tag{5.19}$$

where $\mathscr{I}$ is a set of references of interaction elements (ids) on the physical representation of the user interface, and X_I and X_O are proxy nodes from FG.

id is a total bijective function, with

$$id : S \cup I \cup C_I \cup C_O \cup P_I \cup P_O \cup X_I \cup X_O \cup B \rightarrow \mathscr{I}\mathscr{D}, \tag{5.20}$$

where $\mathscr{I}\mathscr{D}$ is a set of ids that uniquely identifies any node, port, or proxy in the FILL graph FG.

The following definition further specifies elements and functions relevant for the transformation, which are not directly related to a given FILL graph but necessary for the transformation.

Definition 4 Should be id_s, S', $ieID$, and $procID$ defined as follows:

$id_s : S' \rightarrow \mathscr{I}\mathscr{D}$ is a total bijective function that matches a place in a reference net to an id.

$S' \subseteq S$ is a subset of places in the reference net representing connections to and from a BPMN node. This function is necessary for the transformation of BPMN nodes; it compensates for the fact that a BPMN node does not have ports associated with ids as it is the case for operation nodes.

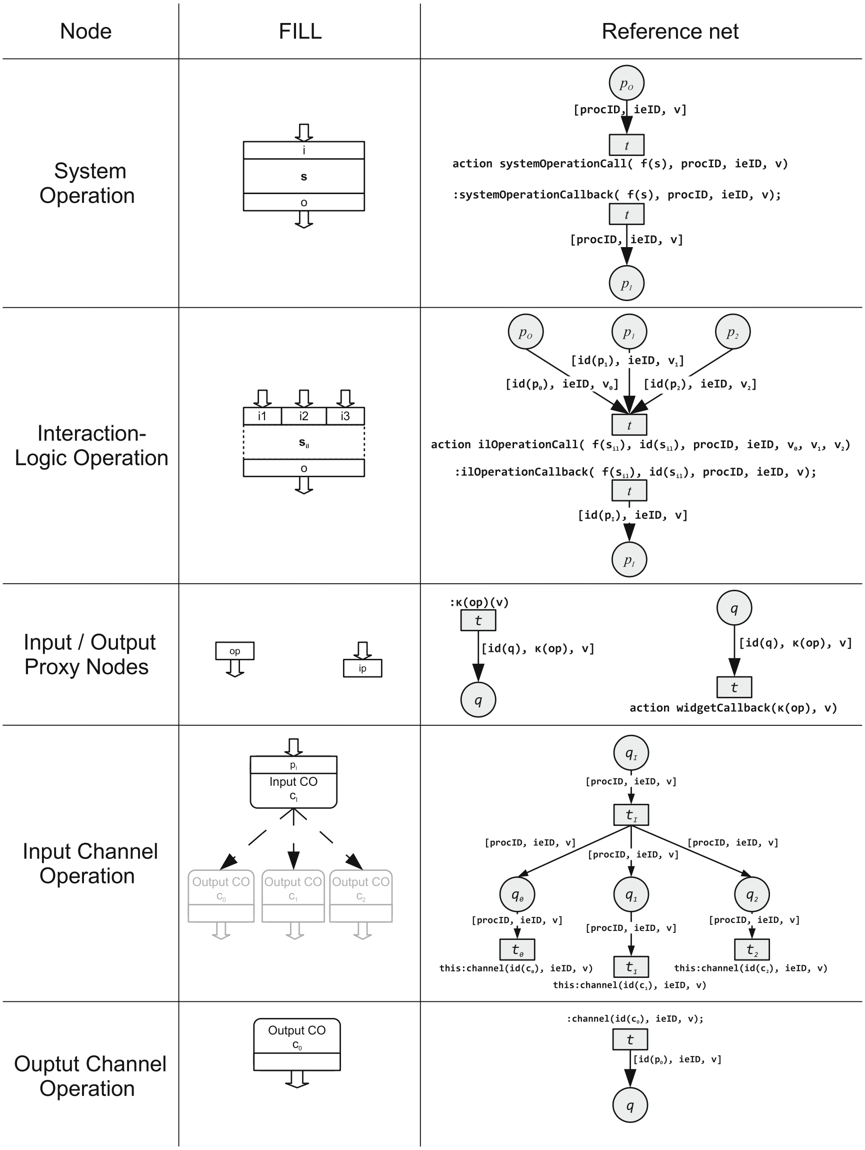


Fig. 5.8 Table presenting the transformation of operation and proxy nodes of FILL to reference nets. The *second column* lists the node type, and the *third column* the resulting reference net. The mapping functions used are defined in Definitions 3 and 4

ieID and *procID* are ids that are generated in the transformation process. *ieID* indicates the id associated with the interaction element that triggers or is triggered by the interaction process. *procID* is used to specify the data flow.

The following introduces the transformation of the different FILL elements.

Transformation of operation nodes—The transformation of interaction logic or system operation nodes results in the generation of two transitions: one for calling an associated (JAVA) method and one for re-entering the net after the method returns. The inscriptions of these transitions differ in the name of the synchronous channel that calls the associated method (`systemOperationCall` versus `ilOperationCall`) and specifies the re-entering point (`systemOperationCallback` versus `ilOperationCallback`). They also differ in the number of variables that are sent to the method, which is indicated by its name $f(op)$, where *op* is the operation node to be transformed to. Data values sent to and from operation nodes are associated with variables, here indicated by v and v_0 to v_2.

The main difference between the transformation of a system operation node (as shown in Fig. 5.8 first row) and the transformation of an interaction-logic operation node (as shown in Fig. 5.8 second row) is the transformation of input ports. According to FILL's syntax definition, every interaction-logic operation has 0 or 1 output port and 0 to n input ports. For system operation nodes, there is exactly 1 input and 1 output port. In general, input and output ports are transformed into an edge/place combination as can be seen in Fig. 5.8, third row.

The transformation of channel operations follows the same concept as system or interaction-logic operations (see Fig. 5.8, fourth and fifth rows). The connection of input to output channel operations by channel reference edges is transformed using the synchronous channel concept of reference nets. If these connections are specified internally in a FILL component, the synchronous channels are specified internally in the net pattern by using the `this` keyword. The net instances that represent connected FILL components need a mechanism to control how nets reference one another. To that end, the `controlNet` can redirect transition firings and simultaneously distribute the reference to the different net instances. For each component, a combination of a transition and a place is added that represents the connection between the `controlNet` and the component net. This is used to forward the data object to the `controlNet` that contains the reference to the corresponding net component. This feature will be discussed in greater detail below.

Proxy nodes—Proxy nodes represent data connectors to and from interaction elements. κ is a function relating a proxy node to its associated interaction element by a unique reference. This reference is a specification of a channel name for an output proxy node (see Fig. 5.8, third row), such that an event can be uniquely redirected to the correct proxy node in the reference net. The callback function from the net to the physical representation and the associated interaction element is specified by a fixed channel name called `widgetCallback`. To identify the correct interaction element for the physical representation, its identifier (given by κ) is passed to the channel as the parameter.

Transformation of BPMN nodes—For BPMN nodes, the transformation into reference nets focuses even more on the structure of the generated net than is the case for operation nodes. Here, the firing semantics of reference nets is actively used to model the semantics of BPMN nodes as used in FILL. Below, the transformations are described per BPMN node for the fusion and branching of interaction processes.

This description includes a detailed presentation of the semantics of each BPMN node type. For the transformation of BPMN nodes, a place is generated in the created reference net for any incoming and outgoing branches which define the entrance or exit point of the BPMN node, as can be seen in Fig. 5.9.

AND(fusion): The outgoing branch is triggered only if all incoming branches provide one or more data objects. This semantic is reflected in the structure of the reference net by defining the places representing the incoming processes as a precondition for the transition *t*. The associated guard condition specifies which data object (here the object associated with the variable *a*) is copied to the outgoing process. The guard condition is obligatory in a FILL graph. The syntax of guard conditions has been specified as compatible with the guard conditions of reference nets (Kummer 2009) (see also Sect. 5.5).

AND(branch): All outgoing branches will be triggered if the incoming branch provides a data object. This semantic is realized by specifying all places representing an outgoing branch as a post-condition of the transition *t* representing the AND node. The guard condition shown in Fig. 5.9 is optional and specifies under which condition the incoming data object is redirected to the outgoing branch.

XOR(fusion): Every incoming data object is redirected to the outgoing edge by copying the data object. Therefore, for every incoming branch, one transition is generated that is optionally inscribed by a guard condition corresponding to the guard condition specified in the FILL graph.

XOR(branch): Only one of the outgoing branches is triggered for an incoming data object. Therefore, any outgoing branch is represented as a transition in the transformed reference net. An outgoing branch must be triggered, but which will be triggered has to be defined by a guard condition.

OR(fusion): Groups of processes can be defined by edge inscriptions in the FILL graph as can be seen in Fig. 5.9: `g1` and `g2` specify one group each. Subsequently, the transformation generates an AND-like subnet for every group and behaves like an `XOR` node between groups. If each of the incoming branches in one group provides a data object, the group's associated transition fires independently from other groups. Guard conditions control which data objects are redirected to the outgoing branch.

OR(branch): Groups of outgoing branches are triggered in accordance with the guard condition. The assignment of the guard conditions to the group is defined by an arrow in the FILL graph's guard condition, as is the case for all guard condition assignments for edges in the above cases of `AND` and `XOR` nodes.

Transformation of data flow edges—Data flow edges are transformed at the very end of the transformation process, after all nodes have been transformed into subnets. Every data flow edge is transformed into one transition, which is connected to the places representing the connected ports or BPMN nodes. The final result is a connected reference net representing the inputted FILL graph.

Generation of the `controlNet`—The `controlNet` implements mainly four functionalities in total:

1. the initialization of devices by instantiating their associated net patterns, which represent the interaction logic components

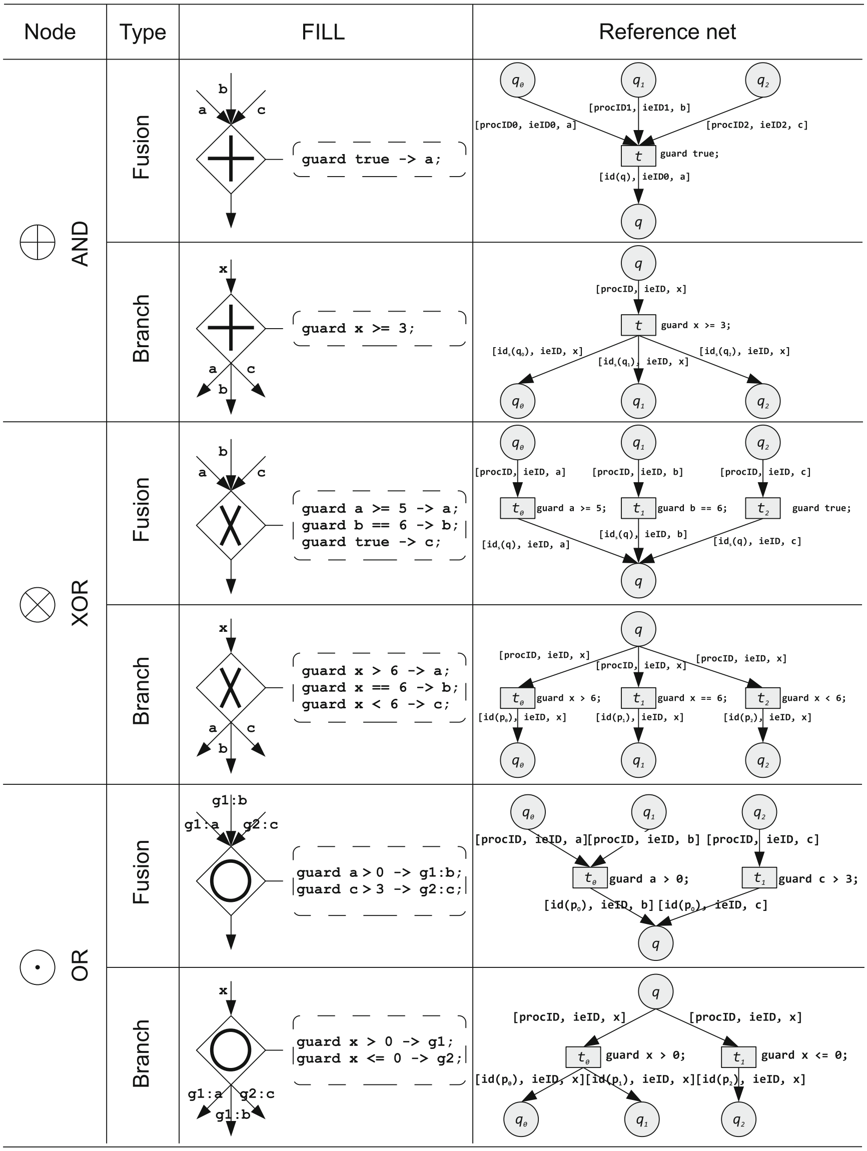


Fig. 5.9 Transformation of BPMN nodes in FILL to reference nets. The *second column* contains the node type (separated into branch and fusion), and the *third column* shows the resulting reference net

2. the initialization of global FILL components
3. the calling of functions on the device sent from an interaction logic component
4. the connection between different net instances representing interaction logic components

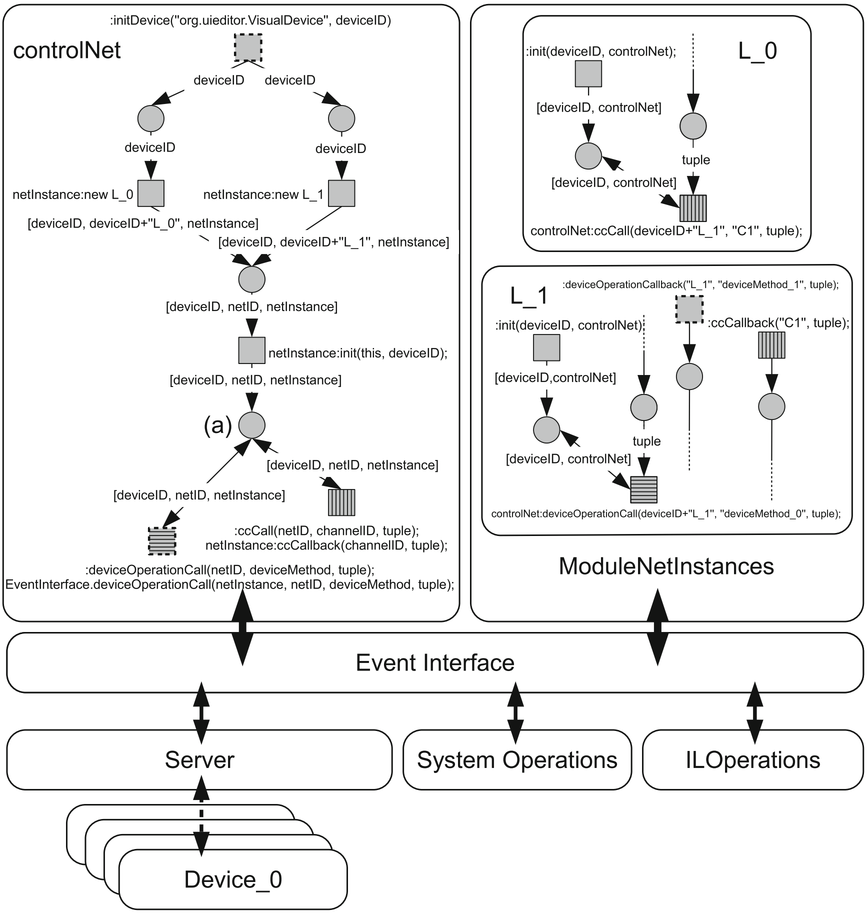

:initDevice("org.uieditor.VisualDevice", deviceID)
controlNet
deviceID
deviceID
deviceID
deviceID
netInstance:new L_0
netInstance:new L_1
[deviceID, deviceID+"L_0", netInstance]
[deviceID, deviceID+"L_1", netInstance]
[deviceID, netID, netInstance]
netInstance:init(this, deviceID);
[deviceID, netID, netInstance]
(a)
[deviceID, netID, netInstance]
[deviceID, netID, netInstance]
:ccCall(netID, channelID, tuple);
netInstance:ccCallback(channelID, tuple);
:deviceOperationCall(netID, deviceMethod, tuple);
EventInterface.deviceOperationCall(netInstance, netID, deviceMethod, tuple);
:init(deviceID, controlNet);
L_0
[deviceID, controlNet]
tuple
[deviceID, controlNet]
controlNet:ccCall(deviceID+"L_1", "C1", tuple);
L_1
:deviceOperationCallback("L_1", "deviceMethod_1", tuple);
:init(deviceID, controlNet);
:ccCallback("C1", tuple);
[deviceID,controlNet]
tuple
[deviceID, controlNet]
controlNet:deviceOperationCall(deviceID+"L_1", "deviceMethod_0", tuple);
ModuleNetInstances
Event Interface
Server
System Operations
ILOperations
Device_0

Fig. 5.10 Example of a `controlNet` showing the result of a possible transformation of a FILL model as represented in Fig. 5.6. Two components and one device are defined. Furthermore, this `controlNet` shows how three global FILL components are handled; they are defined independently from devices

In Fig. 5.10, an exemplary `controlNet` is shown in which all the transitions with a dashed border represent transitions that are fired or called by the `Event Interface` that is a part of the UIEditor execution component (see Sect. 5.6.2). Through a server–client architecture, the `EventInterface` connects the implementations of the system to be controlled, the interaction logic operations, and the devices (see Sect. 5.6.2). All transitions associated with synchronous channels are indicated by the hatching overlay in Fig. 5.10. For reasons of simplification, in the following description, all net patterns that are transformed FILL components will be called *component nets*.

The entire upper part of the controlNet shown in Fig. 5.10 is dedicated to the initialization (see upper left block of Fig. 5.10) of the component nets associated with a device type defined in the model. Information about which component nets are associated with which device type is gathered from the first-generation step by inspecting the FILL model. The relation of views and devices to FILL components can also be gathered by the mapping functions defined in Definition 3.

For each component net associated with a device type, a transition is generated in the controlNet that instantiates one net instance per component net. In Fig. 5.10, the transitions (t1) and (t2) are examples of such instantiation transitions, in this case to instantiate the net components L_1 and L_2. The component net is associated with a specific device instance by mapping the deviceID to the component net instances. The device id is initially passed to the controlNet by the EventInterface by calling the transition (ti). This process generates tokens representing the device id passed and activates the transitions (t1) and (t2). The result of each transition is a tuple associating the net instance of the particular component net with the device id. This makes it possible to instantiate more than one device of the same type, simultaneously making it possible to multiplex the callbacks from the interaction logic to the specific device. The generated tuples are added to place (a).

During the initialization phase of the net instances representing interaction logic components associated with a device, a reference to the controlNet is sent to the newly created net instance by firing the transition inscribed with :init in the component nets. Using this reference as a callback to the controlNet, it is possible to process calls from component nets to the device. The callback from a device to a component net is handled by the EventInterface firing the deviceOperationCallback transition. For examples of these processes, see component net L_2 in Fig. 5.10. Finally, the controlNet handles the communication between component nets as described by channel operations in the FILL model. The transition inscribed with controlNet:ccCall(...) in L_1 fires the associated transition in the controlNet, which matches the component net instances to the passed id. The associated transition inscribed with :ccCallback(...) in L_2 is then fired synchronously.

In the controlNet, for each global component (a component not related to a device or a view) a transition is specified to instantiate the associated net pattern. The main difference for global components is that a global component is only instantiated once and is not mapped to a device id. The instantiation is triggered by firing the transition inscribed with :initGlobalComponents(), which can be seen in Fig. 5.10 on the left in the controlNet. Firing is triggered by the EventInterface. Every generated net instance is initiated by firing the transition inscribed with netInstance:init(...). This sends the controlNet reference to the net instance for channel calls and callbacks. Another difference from device-dependent components is that the net ids of global nets are predefined during the transformation of FILL graphs to reference nets. Thus, the ids do not have to be concatenated with a device id (see inscriptions of transitions (L1) and (L2)) during run-time because the global components exist exactly once and are independent of devices.

An example of a reference to a global component can be seen in L1 (Fig. 5.10, upper-right). The mechanism used is identical with the one for referring to device-dependent components but without the need to include the previously transferred device id.

In the following, we will introduce a software tool that implements the entire architecture, including the event interface, the server–client architecture, and the coupling of all classes with Renew, the reference net simulator.

5.6 Modeling and Editing

The UIEditor is a JAVA-based software tool for modeling, running, and reconfiguring formally described user interface models with FILL. The UIEditor can be used in three different modes: creation, execution, and reconfiguration. Reconfiguration is discussed in Chap. 10. Every mode is combined with a specific interactive editor or visualization. The different modes and their accompanying editors, visualizations, and algorithms will be described in greater detail below. For the execution of a multidevice user interface model, the UIEditor implements a server–client architecture. It enables the initialization of the devices as well as communication between the interaction logic simulation on the server side and the physical representation on the client side. Additionally, it couples the simulator for the reference net-based representation of the component-based interaction logic with the system and implements the interaction-logic operation. Devices are coupled by the implemented EventInterface, as shown in Fig. 5.10.

5.6.1 UIEditor—Creation

Figure 5.11 shows a two-part editor: the FILL canvas for modeling FILL graphs on the left and the physical representation canvas for modeling a WIMP interface on the right. To add an element to a canvas, it has to be dragged from a palette on the left into the canvas on the right. Further positioning and resizing of the operation nodes or interaction elements is performed by clicking on the node using the resize handles or right-clicking on the context menu to set the parameters of the operations, such as the initial parameters or the parameters of the interaction elements (colors, labels, etc.). Other actions—such as deleting operation nodes or interaction elements, duplicating nodes, or adding channel operations to the FILL graph—are supported by toolbars at the top of any canvas. Additional tools—for loading and saving a user interface model as an XML-based description, adding a system interface, and interaction-logic operations—are accessed through the menu bar. System interface operations are read via JAVA's reflection mechanism from their loaded implementation. Here,

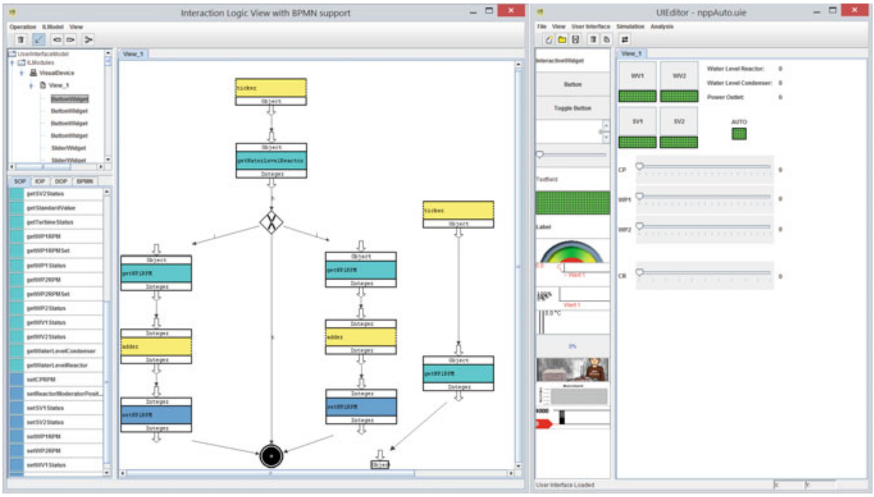


Fig. 5.11 UIEditor in creation mode showing two drag-and-drop-based visual editors for the creation of FILL models on the *left* and a WIMP-based physical representation on the *right*. The editors offer various editing operations in tools bars and menus

each getter and setter method is interpreted and represented as an output or input system operation, respectively. While running the user interface model, these methods are called when the appropriate operation is triggered in the FILL process. The same principal is used for interaction-logic operations. Therefore, data transformation routines represented as interaction-logic operations in a FILL graph are implemented as JAVA methods and called while running the user interface model. This is possible due to the referencing mechanism of reference nets, which synchronizes the firing of transitions in the net and the calling of methods.

In the current version, only a visual editor for WIMPs for the physical representation has been implemented. The UIEditor offers extensibility for adding other types of devices. The serialization mechanism could be used to send the model of the physical representation created with the UIEditor to the devices. This distribution of the physical representation is currently under development. Because it abstracts interaction elements to widgets, the UIEditor can offer a specific set of proxy nodes for the FILL graph description without knowing the device-specific presentation of interaction elements.

5.6.2 UIEditor—Execution

The first step is to load a user interface model for execution. The FILL graph is passed to a converter, which transfers it to a reference net as a PNML file using the transformation described above (see Sect. 5.5.3.2). PNML is a standard markup language for the serialization of Petri nets (Weber and Kindler 2003). The physical representation is passed to a renderer, which generates a JAVA Swing-based container defining interaction elements with their specified parameters, such as position and size. In the case of a multidevice setting, messages are sent to all devices in order to initialize their individual physical representations. All devices have to be registered in advance on the UIEditor's server. The message passing during execution is based on a simple server–client architecture in which the UIEditor represents the server and the devices represent the clients (see Fig. 5.10). After loading the user interface model, the reference net is passed to the execution component of the UIEditor. This component passes the reference net to Renew. By linking Renew with the loaded interaction logic and the `EventInterface`, it is ready to start the execution (see Sect. 5.5.3.2 and Fig. 5.10). To do so, Renew executes the reference net-based interaction logic model. The execution component of the UIEditor loads the system interface and interaction-logic operation implementations and connects them to the `EventInterface`. The `EventInterface` implements the connector with Renew in order to execute the interaction logic, the system interface, and the interaction-logic operation implementations, as well as the server for communication with the distributed devices or the WIMP-based standard implementation. This entire concept is laid out in Fig. 5.10.

To enable the execution of multidevice models, the UIEditor offers a client–server architecture as an extension to the basic execution framework for WIMP user interfaces. The server encapsulates the `EventInterface` and provides a message socket for sending messages to and from the devices. The data protocol between clients and server uses comma-separated values:

$$[Time, Device, View, Method, Content]$$

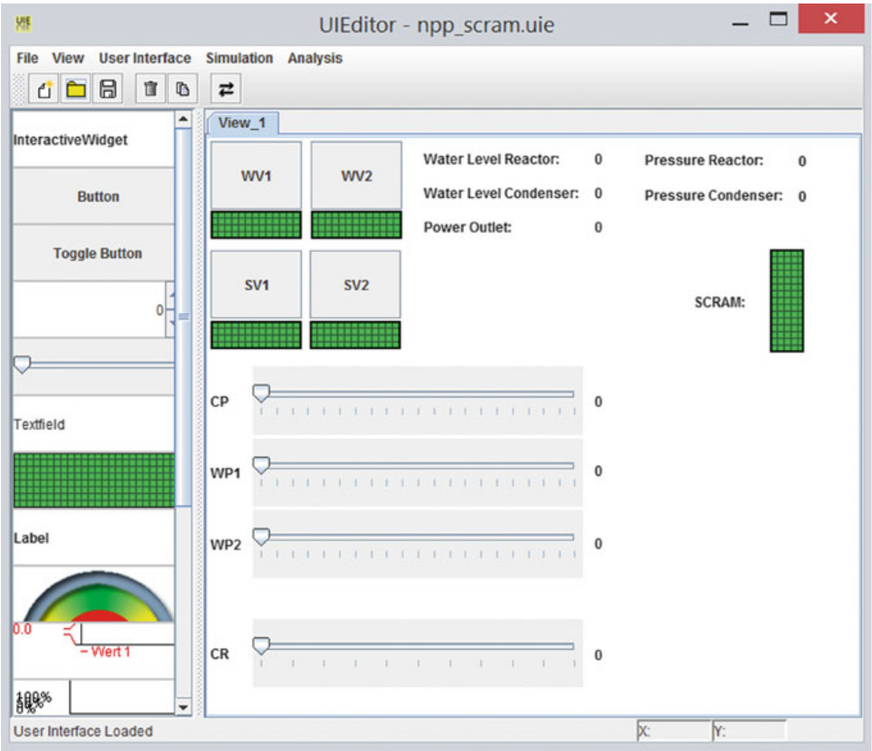
The first value defines the time an event was sent by the interaction device. The second and third values specify the source or target device and target view. This process uses ids that were generated during the instantiation of the reference net and the devices. The *Method* value specifies the entry point to the interaction logic or the physical representation. For instance, the device has to know in which text field to place new content (from the interaction logic executed on the server side). The last parameter represents the content being transferred. The entire architecture is shown at the bottom of Fig. 5.10.

5.7 Case Study

In this section, the nuclear power plant case study as presented in Chap. 4 will be used to exemplify a user interface model incorporating the UIEditor and the presented FILL modeling approach. This model neglects the multidevice aspect of FILL. This is because the multidevice concept only affects the initialization of the reference net model and not the interaction logic model itself.

The physical representation of the modeled user interface can be seen in Fig. 5.12. It has four buttons that are equipped with lamps, representing the current status of the valve (green corresponds to open and red to closed). There are also four sliders—one for each water pump and one to control the position of the control rods responsible for the reactor's thermal output. The physical representation offers a set of output values that show the current selected speed of the pumps or the position of the control rods and the current pressure and water level in the vessels. One of these output values represents the power output of the plant. The physical representation features a lamp showing whether the SCRAM state has been reached or not. SCRAM stands for an emergency shutdown of nuclear power plants in case of system's critical situations, such as an decreasing water level under a certain threshold. SCRAM can be executed automatically by the control system (as presented here) or manually through the human operator.

Figure 5.13 shows the FILL model that corresponds to the physical representation presented. For simplification, only one FILL graph is shown for redundant parts of



Water Level Reactor: 0

Water Level Condenser: 0

Power Outlet: 0

Pressure Reactor: 0

Pressure Condenser: 0

CP 0

WP1 0

WP2 0

CR 0

User Interface Loaded

Fig. 5.12 Physical representation of the user interface model for monitoring and control of a simplified nuclear power plant. It comprises a button panel that controls water and steam valves and sliders that control water pump speed and the control rods in the reactor core. Lamps and numerical labels provide user feedback by indicating the current status of the nuclear power plant

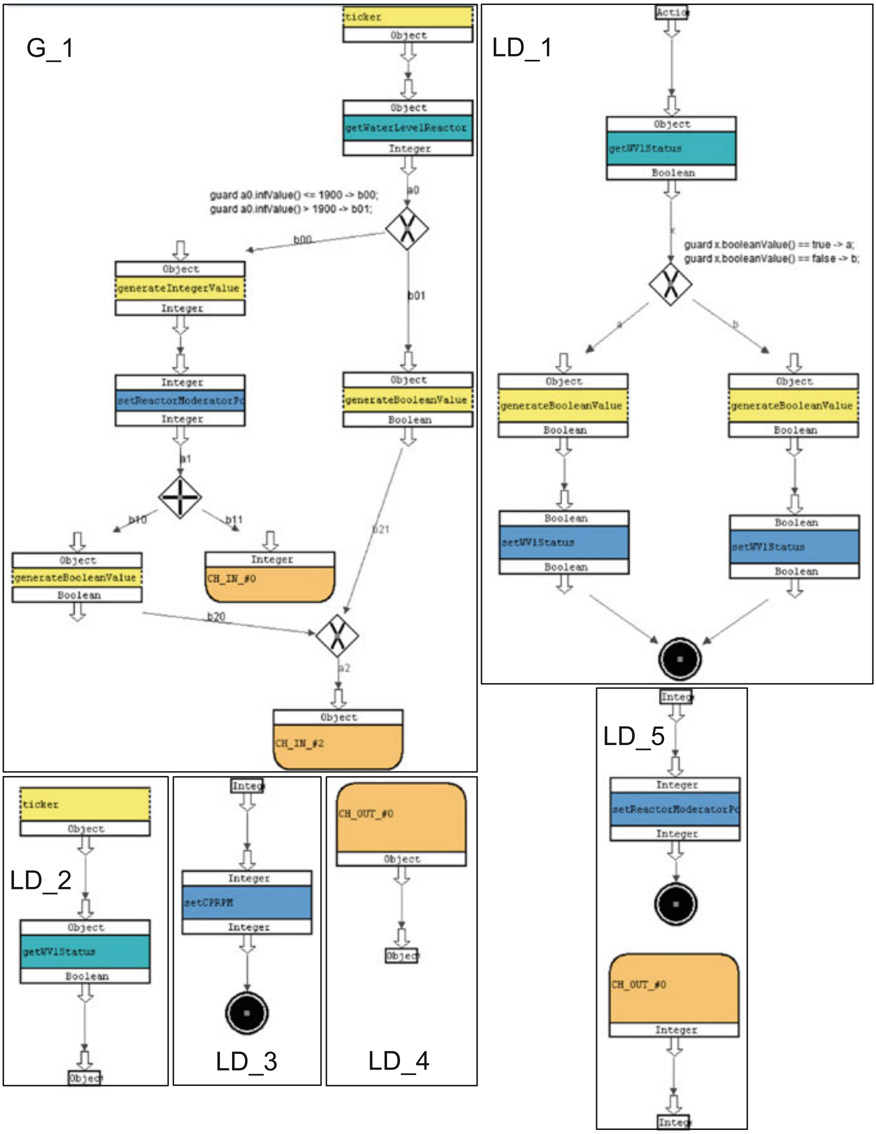


Fig. 5.13 Complete model of the case study: a simple user interface including a model for a simple implementation of the SCRAM operation. The model is comprised of five local components (LD_1 to LD_5), each of which is related to an interaction element. Some of them are redundant, existing in several slightly adapted versions in the interaction logic model; for example, LD_1. G_1 is a global component modeling the SCRAM operation

the interaction logic model, such as the valve buttons (LD_1). In the FILL graph LD_1, the event object sent from the corresponding button (WV1) triggers a system operation, which itself transmits a system value (status of WV1) to the graph. Depending on that value, the left or right branch of the succeeding BPMN node is triggered, which ultimately generates the opposite Boolean value. This value is set to WV1 by triggering the system operation setWV1Status. This component in the model is identical to the one in the example presented in Fig. 5.5.

The LD_2 component represents the corresponding FILL graph for sending the current status of the valve to the lamp, which is green when the valve is open (true) and red when it is closed (false). For each lamp, one instance of this component exists in the model, each triggering a different system operation that returns the status of the corresponding valve. Component LD_3 contains the FILL graph related to the slider setting the speed of the condenser pump. For each slider that sets the speed of a water pump, a similar component is specified in the interaction logic. These components differ in the system operation the value of the slider is sent to. The LD_4 and LD_5 components use channel operations to connect to the global FILL component G_1. LD_4 represents the callback to the lamp inscribed with SCRAM in the physical representation. This lamp indicates whether the SCRAM operation has been executed, which is modeled in G_1. The channel output operation in LD_4 is connected to the channel input operation CH_IN_#2 in G_1. LD_5 shows the FILL graph related to the slider controlling the position of the control rods in the reactor tank. The position in the system is set by changing the position of the slider; the FILL graph is similarly modeled for LD_3. To execute the SCRAM procedure, the new position of the rod position defined in G_1 is also sent to the slider, reflecting the new value in its position through the given output channel operation in LD_5. This channel operation is connected to operation CH_IN_#0 in G_1.

G_1 (partially) implements the SCRAM operation as discussed in Chap. 4. The ticker operation at the top of the FILL graph in component G_1 periodically triggers the getWaterLevelReactor system operation, which transmits the current water level of the reactor vessel to the graph. At the subsequent BPMN XOR node, this value is redirected to one of the subsequent branches in accordance with the evaluation of the inscribed guard condition. If the water level is above 1900 mm, a false Boolean value is generated and sent to a channel operation that redirects this value to LD_4 and thus to the SCRAM lamp. If the water level is under (or equal to) 1900 mm, the left branch is triggered, generating an integer equal to zero. This integer is set as a new control rod position and then sent to a channel and an interaction logic operation by passing a branching AND node. The channel operation transfers the value to LD_5 (operation CH_OUT_#0) and sets the slider position to zero. For the other branch, a true Boolean value is generated and sent over the channel operation CH_IN_#2 to the SCRAM lamp (LD_4), switching it to green.

The SCRAM procedure, which is modeled only by setting the control rod position to zero (completely in the core), can be extended. For instance, valves can be opened or closed, and the pump speed can be set to a certain value to continue cooling the core. This can be done by adding branches to the AND BPMN node in G_1, which currently triggers only FILL components (LD_3 and LD_4). Furthermore, due to the component-based structure of FILL-based interaction-logic models, it is possible to exchange the SCRAM procedure with another model if necessary. Finally, modeling the SCRAM procedure as a global component makes it independent from other parts of the interaction logic. Thus, when interaction elements are added to or deleted from the physical representation, the corresponding parts of the interaction logic have no impact on the model of the SCRAM procedure.

5.8 Conclusion

This chapter introduced a formal modeling concept combined with a visual description language for modeling interaction logic called FILL. The model of a user interface using this description concept is based on the three-layer component architecture 3LA, which has been derived from various well-known architectural concepts in HCI, such as MVC and interactors, and has been mapped to the ARCH model. This architecture divides a user interface into a physical representation, an interaction logic, and the system interface that represents the system to be controlled. The physical representation is described as a set of interaction elements with which the user directly interacts. The interaction-logic model processes the events transmitted by the physical representation (originally triggered by the user through such means as pressing a button) or by the system. The interaction-logic model is structured into components to make the parts of the model exchangeable and reusable. This supports the semantic structuring of interaction logic models separately from one another by such means as defining dialog or business models as part of the interaction logic as a whole (cf. ARCH). Finally, the applicability of the approach introduced was demonstrated through a case study: a user interface for a simplified simulation of a nuclear power plant. This model was discussed in detail to show the various features of FILL in a model of a user interface.

References

Barboni E, Hamon A, Martinie C, Palanque P (2015) A user-centered view on formal methods: interactive support for validation and verification. In: Workshop on formal methods in human computer interaction, Duisburg, 23 June 2015

Barboni E, Ladry JF, Navarre D, Palanque P, Winckler M (2010) Beyond modelling: an integrated environment supporting co-execution of tasks and systems models. In: Proceedings of the 2nd ACM SIGCHI symposium on engineering interactive computing systems, Berlin, 19–23 June 2010

Barboni E, Martinie C, Navarre D, Palanque P, Winckler M (2013) Bridging the gap between a behavioural formal description technique and a user interface description language: enhancing ICO with a graphical user interface markup language. Sci Comput Program 86:3–29

Bass L, Faneuf R, Little R, Mayer N, Pellegrino B, Reed S, Seacord R, Sheppard S, Szczur MR (1992) A metamodel for the runtime architecture of an interactive system. SIGCHI Bull 24(1):32–37

Bass L, Little R, Pellegrino R, Reed S, Seacord R, Sheppard S, Szezur MR (1991) The arch model: Seeheim revisited. In: User interface developers' workshop, 26 April 1991

Bastide R, Navarre D, Palanque P (2003) A tool-supported design framework for safety critical interactive systems. Interact Comput 15(3):309–328

Beaudouin-Lafon M (2000) Instrumental interaction: an interaction model for designing post-wimp user interfaces. In: Proceedings of the SIGCHI conference on human factors in computing systems, The Hague, 1–6 April 2000

Berti S, Correani F, Mori G, Paternò F, Santoro C (2004) Teresa: a transformation-based environment for designing and developing multi-device interfaces. In: Extended abstracts of the SIGCHI conference on human factors in computing systems, Vienna, 24–29 April 2004

Brat G, Martinie C, Palanque P (2013) V&V of lexical, syntactic and semantic properties for interactive systems through model checking of formal description of dialog. In: Human-computer interaction. Human-centered design approaches, methods, tools, and environments. Lecture notes in computer science, vol 8004. Springer, Heidelberg, pp 290–299

Burkolter D, Weyers B, Kluge A, Luther W (2014) Customization of user interfaces to reduce errors and enhance user acceptance. Appl Ergon 45(2):346–353

Calvary G, Coutaz J, Thevenin D, Limbourg Q, Bouillon L, Vanderdonckt J (2003) A unifying reference framework for multi-target user interfaces. Interact Comput 15(3):289–308

Chinosi M, Trombetta A (2012) BPMN: an introduction to the standard. Comput Stand Interfaces 34(1):124–134

Coutaz J (2001) Software architecture modeling for user interfaces. In: Encyclopedia of software engineering. Wiley Online Library

de Rosis F, Pizzutilo S, De Carolis B (1998) Formal description and evaluation of user-adapted interfaces. Int J Hum Comput Stud 49(2):95–120

Dix AJ (1991) Formal methods for interactive systems. Academic Press

Duke D, Faconti G, Harrison M, Paternó F (1994) Unifying views of interactors. In: Proceedings of the workshop on advanced visual interfaces, Bari, 1–4 June 1994

Florins M, Vanderdonckt J (2004) Graceful degradation of user interfaces as a design method for multiplatform systems. IUI 4:140–147

Gaines, BR (1991) An interactive visual language for term subsumption languages. In: Proceedings of the twelfth international joint conference on artificial intelligence, Sydney, 25 August 1991

Janssen C, Weisbecker A, Ziegler, J (1993) Generating user interfaces from data models and dialogue net specifications. In: Proceedings of the INTERACT'93 and CHI'93 conference on human factors in computing systems, Amsterdam, 24–29 April 1993

Jensen K, Rozenberg G (2012) High-level Petri nets: theory and application. Springer

John BE, Kieras DE (1996) The GOMS family of user interface analysis techniques: comparison and contrast. ACM Trans Comput Hum Interact 3(4):320–351

Kecher C, Salvanos A (2015) UML 2.5: das umfassende Handbuch. Rheinwerk Computing

Krasner GE, Pope ST et al (1988) A description of the model-view-controller user interface paradigm in the smalltalk-80 system. J Object Oriented Program 1(3):26–49

Kummer O (2009) Referenznetze. Logos

Kummer O, Wienberg F, Duvigneau M, Schumacher J, Köhler M, Moldt D, Rölke H, Valk R (2004) An extensible editor and simulation engine for Petri nets: renew. In: Applications and theory of Petri nets, Bologna, 21–26 June 2004

Limbourg Q, Vanderdonckt J, Michotte B, Bouillon L, López-Jaquero V (2005) UsiXML: a language supporting multi-path development of user interfaces. In: Proceedings of engineering human computer interaction and interactive systems, Hamburg, 11–13 July 2005

Navarre D, Palanque P, Ladry JF, Barboni E (2009) ICOs: a model-based user interface description technique dedicated to interactive systems addressing usability, reliability and scalability. ACM Trans Comput Hum Interact 16(4):1–18

Paternò F (1994) A theory of user-interaction objects. J Vis Lang Comput 5(3):227–249

Paternò F, Mancini C, Meniconi S (1997) ConcurTaskTrees: a diagrammatic notation for specifying task models. In: IFIP TC13 international conference on human-computer interaction, Sydney, 14–18 July 1997

Paternò F, Santoro C (2001) Integrating model checking and HCI tools to help designers verify user interface properties. In: Interactive systems design, specification, and verification, Glasgow, 13–15 June 2001

Petri CA (1962) Kommunikation mit Automaten. Dissertation, University of Hamburg

Pfaff GE (1985) User interface management systems. Springer, New York

Priese L (2008) Petri-Netze. Springer, Berlin

Rasmussen J (1983) Skills, rules, and knowledge; signals, signs, and symbols, and other distinctions in human performance models. IEEE Trans Syst Man Cybern 3:257–266

Silva JL, Campos JC, Harrison MD (2014) Prototyping and analysing ubiquitous computing environments using multiple layers. Int J Hum Comput Stud 72(5):488–506

Sy O, Bastide R, Palanque P, Le D, Navarre D (2000) PetShop: a case tool for the petri net based specification and prototyping of corba systems. In: Petri nets 2000, Aarhus, 26–30 June 2000

Szyperski C (1997) Component software: beyond OO programming. Addison-Wesley, Boston

Van Dam A (1997) Post-wimp user interfaces. Commun ACM 40(2):63–67

Weber M, Kindler E (2003) The petri net markup language. Petri net technology for communication-based systems. Springer, Berlin, pp 124–144

Weyers B (2012) Reconfiguration of user interface models for monitoring and control of human-computer systems. Dr, Hut, Munich

Weyers B (2013a) FILL: formal description of executable and reconfigurable models of interactive systems. In: Proceedings of the workshop on formal methods in human computer interaction, Duisburg, 23 June 2015

Weyers B (2013b) User-centric adaptive automation through formal reconfiguration of user interface models. In: The sixth international conference on advances in human oriented and personalized mechanisms, technologies, and services, Venice, 27 October–1 November 2013

Weyers B, Burkolter D, Kluge A, Luther W (2010) User-centered interface reconfiguration for error reduction in human-computer interaction. In: 3rd international conference on advances in human-oriented and personalized mechanisms, technologies and services, Nice, 22–27 August 2010

Weyers B, Burkolter D, Kluge A, Luther W (2012) Formal modeling and reconfiguration of user interfaces for reduction of human error in failure handling of complex systems. Int J Hum Comput Interact 28(10):646–665

Weyers B, Luther W (2010) Formal modeling and reconfiguration of user interfaces. In: International conference of the Chilean computer science society, Antofagasta, 15–19 November 2010

Weyers B, Baloian N, Luther W (2009) Cooperative creation of concept keyboards in distributed learning environments. In: International conference on computer supported cooperative work in design, Santiago de Chile, 22–24 April 2009

Weyers B, Luther W, Baloian N (2011) Interface creation and redesign techniques in collaborative learning scenarios. Fut Gener Comput Syst 27(1):127–138

White SA, Miers D (2008) BPMN modeling and reference guide. Future Strategies Inc., Lighthouse Point

Chapter 6
Combining Models for Interactive System Modelling

Judy Bowen and Steve Reeves

Abstract Our approach for modelling interactive systems has been to develop models for the interface and interaction which are lightweight but with an underlying formal semantics. Combined with traditional formal methods to describe functional behaviour, this provides the ability to create a single formal model of interactive systems and consider all parts (functionality, user interface and interaction) with the same rigorous level of formality. The ability to convert the different models we use from one notation to another has given us a set of models which describe an interactive system (or parts of that system) at different levels of abstraction in ways most suitable for the domain but which can be combined into a single model for model checking, theorem proving, etc. There are, however, many benefits to using the individual models for different purposes throughout the development process. In this chapter, we provide examples of this using the nuclear power plant control system as an example.

6.1 Introduction

Safety-critical interactive systems are software or hardware devices (containing software) operating in an environment where incorrect use or failure may lead to loss, serious harm or death, for example banking systems, ATMs, medical devices, aircraft cockpit software, nuclear power plant control systems and factory production cells. Avoiding such errors and harm relies on the systems being developed using robust engineering techniques to ensure that they will behave correctly and also that they can be used successfully.

Developing suitable interfaces for safety-critical systems requires two things. First, they must be usable in their environments by their users—i.e. they must be

J. Bowen (✉) · S. Reeves
University of Waikato, Hamilton, New Zealand
e-mail: jbowen@waikato.ac.nz

S. Reeves
e-mail: stever@waikato.ac.nz

B. Weyers et al. (eds.), *The Handbook of Formal Methods in Human-Computer Interaction*, Human-Computer Interaction Series,
DOI 10.1007/978-3-319-51838-1_6

developed using a sound user-centred design (UCD) process and following known HCI principles. Secondly, we must be able to verify and validate the user interface and interaction with the same rigour as the underlying functionality. While we can (we hope) assume the former, the latter is harder and requires us to develop suitable techniques which not only support these requirements but which will also be useful (and used) by the interface developers of such systems.

We have developed modelling techniques for the user interface (UI) and interactivity of a system which take as a starting point typical informal design artefacts which are produced as part of a UCD process, e.g. prototypes (at any level of fidelity), scenarios and storyboards. In addition to the interface modelling techniques, we also have mechanisms for combining these models with more traditional functional specifications (which deal with the requirements for the system behaviour) in order to be able to reason about the system as a whole.

In the rest of this chapter, we provide details of the models and notations we use to describe the different parts of an interactive system. We also discuss how these can be combined into a single model to give a single, formal 'view' of the entire system. This cohesive model allows us to consider important properties of the system (which generally involves proving safety properties and ensuring the system as specified will behave in known, safe ways at all times) which encompasses aspects of the UI and interaction as well as functional behaviour. At the same time, however, the individual component models used to create this single model have their own benefits. They give us the ability to consider different aspects of the system (either specific parts or different groups of behaviours, for example) using different levels of abstraction or different modes of description to suit the domain. Essentially, they provide us with a set of options from which we can select the most appropriate model for a given use. Because these component models are developed as part of the design process and form part of the overall system model, we essentially get this 'for free' (that is without additional workload).

We use the nuclear power plant control system as an example to show how these models can be used independently, as well as in combination, to consider different properties of interest during a development process.

6.2 Related Work

In early years, formal methods were developed as a way of specifying and reasoning about the functionality of systems which did not have the sorts of rich graphical user interfaces provided by today's software. Some formal methods were used to reason about interaction properties, e.g. (Jacob 1982; Dix and Runciman 1985), but as user interfaces evolved and became more complex, and the importance of their design became increasingly obvious, the disciplines of HCI and UCD evolved to reflect this. However, these two strands of research—formal system development and UI design research—remained primarily separate and the approaches used within them were also very different. On the one hand were formal languages and notations based on mathematical principles used to formally reason about a system's behaviour via specification, proof, theorem proving, model checking, etc., while on the other were design processes targeted at usability based on psychological principles and involving shared, informal design artefacts, understanding end-users and their tasks, evaluation of options and usability, etc.

This gap between the formal and informal has been discussed many times, notably as far back as 1990 by Thimbleby (1990). Numerous approaches have been taken over the years to try and reduce the gap between the two fields, particularly as the need to reason about properties of UIs has become increasingly necessary due to the prevalence of interactive systems in general and the increase in safety-critical interactive systems in particular. We can generalise key works in this area into the following categories:

- development of new formal methods specifically for UIs (Puerta and Eisenstein 2002; Courtney 2003; Limbourg et al. 2004)
- development of hybrid methods from existing formal methods and/or informal design methods. (Duke and Harrison 1995; Paternò et al. 1995)
- the use of existing formal methods to describe UIs and UI behaviour (Harrison and Dix 1990; Thimbleby 2004)
- replacing existing human-centred techniques with formal model-based methods. (Hussey et al. 2000; Paternò 2001; Reichart et al. 2008)

These, and other similar works, constitute a concerted effort and a step forward in bringing formal methods and UI design closer together. However, in many cases, the resulting methods, models and techniques continue to either retain the separation of UI and functionality in all stages of the development process, or seek to integrate them by creating new components within the models which combine elements of both in a new way (Duke et al. 1994, 1999).

When we first began to consider the problem and investigate and develop modelling techniques for interactive systems, we had a number of criteria, including a desire to model at the most natural level of granularity (describe the existing components as they appear) as well as come up with an approach that could fit with both formal and HCI methodologies. In contrast to other approaches, our starting point is that of design artefacts (of both interface and system) which may be developed separately, and perhaps at different times, during the development life cycle. Unlike more recent work such as (Bolton and Bass 2010), we do not include models of user behaviour or consider the UI in terms of the tasks performed or user goals. Rather, we model at a higher level of abstraction which enables us to consider any available behaviours of the system via the UI rather than constraining this to expected actions of the user based on predefined goals.

So, just as the interfaces we design must be suitable for their systems and users, the models we use to reason about these interactive systems must also be usable and useful to their users (designers, formal practitioners, etc.). Rather than having an expectation that UI designers should throw away their usual, and necessarily informal,design processes in favour of a new formal approach, we are of the opinion that these should be incorporated into our approach rather than being replaced by it.

We start with the assumption that traditional, informal user-centred design practice has been used, as usual, to produce many artefacts relating to the ‘look and feel’ of the system. These will include design ideations such as prototypes, scenarios, and storyboards and will be the result of collaboration between designers and end-users, as well as other stakeholders. Keeping this point in mind is very important when it comes to understanding the modelling techniques that we discuss in this chapter.

In the same way that we assume the interface has been developed using well-known and appropriate design methodologies, we similarly assume that the design of the functional behaviour of the system has likewise followed a rigorous development process which leads to the development of formal artefacts such as a system specification or some other formal model of the system’s behaviour. Given that we are interested in safety-critical systems, where erroneous behaviour can lead to injury or even death, this assumption seems a reasonable one.

In order to be able to reason about all parts of the system and not just the functional behaviour, we need a way of considering the informal design artefacts with the same level of formality as the functional specification. To do this, we create formal models of these design artefacts. By creating a formal representation of the informal design artefacts (in essence creating a different abstraction of the UI which happens to be in a formal notation), we gain the ability to combine the two representations (system and UI) into another model which gives a single view of the system as a whole. This then allows us to reason about the interactive system in a way which captures all of the information (both UI and system) and perform activities such as model checking and theorem proving to show that all parts of the system will have the properties we desire (Bowen and Reeves 2008, 2013).

We next give an overview of the different models used in our process and then go on to provide examples of these in use for the nuclear power plant example.

6.3 Background

Our starting point for the modelling process is to consider the two main components of the system (functionality and interactivity—which includes the interface) separately. This separation of concerns is, in some sense, an artificial division. While it is often the case that one group within a design team may focus on appearance and look and feel of the interface while another focusses on core functionality, we are not aiming to reflect the divisions and complexities that exist within design groups. Rather, the separation allows us to consider different parts of the system in different, and appropriate, ways, and provides the basis for our use of different levels of abstraction to describe different components within that system. Such separation is a common approach taken in this type of modelling, although it can be done at differing levels of granularity. For example in Chap. 5, we see functional behaviours described as components which are then associated with related widgets and composed using channels, whereas Chap. 14 uses a layered approach where different parts of the system and model are described in different layers.

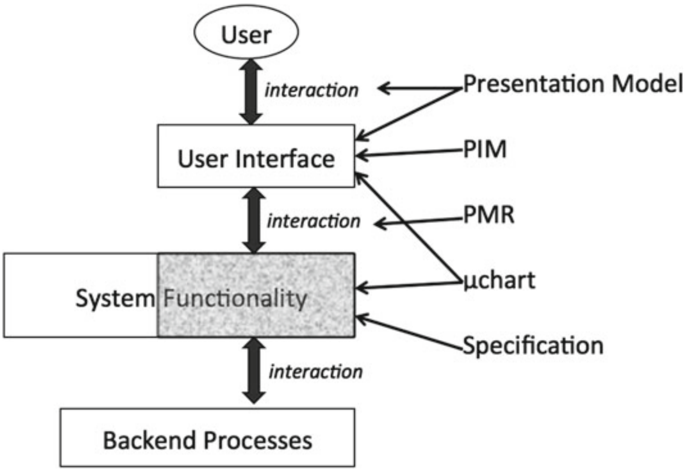


Fig. 6.1 Overview of model components

We rely on a combination of existing languages and models to specify the functionality. Typically, for us, this means creating a Z specification (ISO/IEC 13568 2002; Henson et al. 2008) and/or μcharts (Reeve 2005) to reason about functional and reactive behaviours, although any similar state-based notation could be substituted for Z. The ProZ component of the ProB tool[1] is used for model checking the specification, or we can use Z theorem provers such as Proofpower.[2] These allow us to ensure that the system not only does the right thing (required behaviour) but also does not do the wrong thing (behaviour that is ruled out), irrespective of the circumstances of use. Figure 6.1 gives an overview of how each of the models relates to the system under consideration.

The presentation model describes the user interface elements that are present but not the layout of the interface. It also describes the types of interaction each widget exhibits (which suggests how a user interacts) and labels the behaviour associated with that widget. The presentation interaction model (PIM) also describes the user interface but at a higher level of abstraction which hides the general behaviours and widgets of the interface and focusses on the navigation that is possible through the different modes/windows, etc.

The presentation model relation (PMR) relates some of the labels of widget behaviours to operations in the specification, and as such gives a very high-level view of the interaction between system and UI. Not all functionality is expressed via the user interface (the user can only directly access certain behaviours) and hence the split in the system functionality box where the grey sections are those functions which do relate to user actions.

[1] http://stups.hhu.de/ProB/.

[2] http://www.lemma-one.com/ProofPower/index/index.html.

μCharts is used as an alternative notation for PIMs, and so describes the same properties of the user interface, but can also be used to describe aspects of system functionality to show cause and effect behaviours. The specification on the other hand describes only the system functionality in terms of what can be achieved and abstracts away all details of how it might be achieved.

Finally, the models can be combined into single representation which gives a low-level description of the what and how of the interface and system and their interaction. We do not explicitly model the user at all in our approach nor any back-end processes such as database connectivity and networks. In the next section, we describe each of the models in more detail using examples from the nuclear power plant example.

6.3.1 Presentation Model

This is a behavioural model of the interface of a system described at the level of interactive components (widgets), their types and behaviours. We group the widgets based on which components of the interface (e.g. windows and dialogues) they appear in (or in the case of modal systems, which modes they are enabled in). The presentation model can be derived from early designs of interfaces (such as prototypes and storyboards), final implementations, or anything in between. As such they can be produced from the sorts of artefacts, interface designers are already working with within a user-centred design process. The presentation model does not in any sense replace the design artefacts it describes. Rather, we use it as an accompanying artefact that provides a bridge between informal designs and formal models. So the presentation model describes the 'meaning' of the interface (or interface design) in terms of its component widgets and behaviours, but the layout and aesthetic attributes are contained in the visual artefacts that the presentation model is derived from.

It is important to appreciate that we do not require widgets to be (only) buttons or checkboxes or menu items, etc. Widgets are any 'devices' through which interaction can occur. For example, sensors attached to parts of a physical system would be widgets: as the physical system evolves, the sensors would note this evolution and, in response, the system would move between states. In this sort of system, we might simply describe a collection of sensors together with the behaviours (both functional and interactive) that their triggering or their readings cause in the system. Thus, our idea of 'interface' is very general.

Each window, dialogue, mode or other interactive state of the system is described separately in a component presentation model (*pmodel*) by way of its component widgets which are described using a triple:

```
widget name, widget type, (behaviours)
```

The full interface presentation model is then simply the collection of the *pmodels*, and describes all behaviours of the interface and which widgets provide the behav-

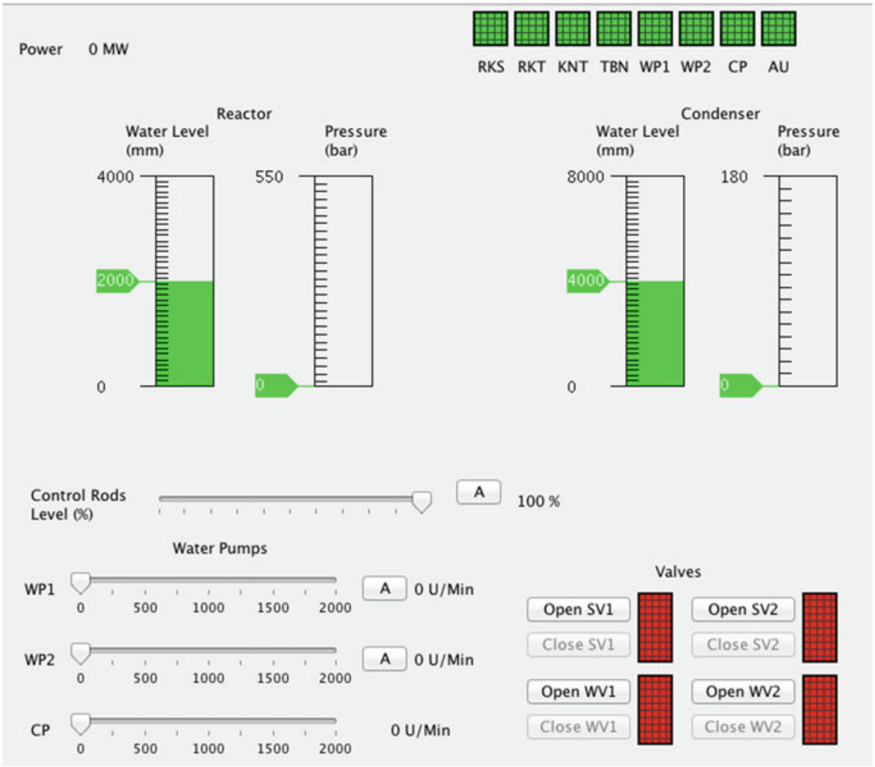


Fig. 6.2 Nuclear plant control example interface

iours. Behaviours are split into two categories, interactive behaviours (I-behaviours) are those which facilitate navigation through the interface (opening and closing new windows, noting the change of a sensor state, etc.) or affect only presentational elements of the interface, whereas system behaviours (S-behaviours) provide access to the underlying functionality of the system (the grey part in Fig. 6.1).

The syntax of the presentation model is essentially a set of labels which we use to meaningfully describe the attributes of the widgets. Consider the interface provided as part of the nuclear power plant example, which we repeat in Fig. 6.2. Each of the widgets is described by a tuple in the presentation model, so for example the power output label at the top left of the display is:

```
PowerDisplay, Responder, (S_OutputPower)
```

The category assigned is that of ‘Responder’ as this widget displays information to the user in response to some inner stored value (which keeps track of the current power level). As such, the behaviour it responds to is a system behaviour which outputs whatever that power level currently is and hence has the label ‘S_OutputPower’. The WP1 slider on the other hand is described as follows:

```
WP1Ctrl, ActionControl, (S_IncWaterPressure1,
                                 S_DecWaterPressure1)
```

The category ‘ActionControl’ indicates it is a widget which causes an action to occur (i.e. the user interacts with it to make something happen), which in this case is to change the value of the water pressure either up or down. Again, these are behaviours of the system and so are labelled as S-behaviours.

Once we have labelled all of the widgets in the UI, we have an overview of all the available behaviours that are presented to a user by that UI. We subsequently give a formal meaning to these labels via the presentation interaction model (PIM) and presentation model relation (PMR) which we describe next.

6.3.2 Presentation Interaction Model

The presentation interaction model (PIM) is essentially a state transition diagram, where *pmodels* are abstracted into states and transitions are labelled with I-behaviours from those *pmodels*. As such, the PIM gives a formal meaning to the I-behaviours as well as provides an abstract transition model of the system’s navigational possibilities. The usual ‘state explosion’ problem associated with using transition systems or finite state automata to model interactive systems is removed by the abstraction of *pmodels* into states, so the size of the model is bounded by the number of individual windows or modes of the system. While the presentation model describes all available behaviours, the PIM describes the availability of these via the user’s navigation. For example, in a UI with multiple windows, the user may be required to navigate through several of these windows to reach a particular behaviour. The nuclear power plant example of Fig. 6.2 has only a single, static UI screen, and as such, the PIM is a single-state automaton. We show later how this changes when we extend the example to have multiple windows constraining behaviour in the case of the emergency scenarios.

A PIM can also be used to consider aspects such as reachability, deadlock and the complexity of the navigational space (via the lengths of navigational sequences). Considered formally, the role of the PIM is to inform us what the allowable sequences of Z operations are for the interactive system that we are modelling. This allows us to make the Z definitions somewhat simpler since we do not have to introduce an elaborate system of flags and consequent preconditions in order to disallow the use of Z operations when the interactivity does not allow them: the PIM handles all of this, and what Z operations are allowed at any point in the interactivity is given by which widgets have behaviours (that are given by the Z operations) at that point.

6.3.3 Presentation Model Relation

Just as the PIM gives meaning to the I-behaviours of the presentation model, the presentation model relation (PMR) does the same for the S-behaviours. These behaviours represent functional behaviours of the system, which are specified in the formal specification. The PMR is a many-to-one relation from all of the S-behaviours in a presentation model to operations in the specification. This reflects the fact that there are often multiple ways for a user to perform a task from the UI, and therefore, there may be several different S-behaviours which relate to a single operation of the specification.

So, in order to understand what an S-behaviour label represents, for example the behaviour label *S_IncWaterPressure1* from the presentation model tuple above, we identify from the PMR the name of the operation in the Z specification that it represents

$$S\_IncWaterPressure1 \mapsto IncreaseWaterPressure$$

This tells us that in the formal specification is an operation called 'IncreaseWater-Pressure' which specifies what effect this operation has on the system. The specified operation then gives the meaning to this behaviour label. We give a larger example of the PMR for the nuclear power plant example later.

6.3.4 Specification

The formal specification of the system provides an unambiguous description of the state of the system and the allowable changes to that state provided by the operations. Many different formal languages exist for such a specification (e.g. VDM, Z, B, Event-B and Object-Z to name but a few), but for our approach, we rely on the Z specification language which is based on set theory and first-order predicate logic. In Z, we typically give a description of the system being modelled which is based on what can be observed of it, and then the operation descriptions show how (the values of) what is observed change. The operations are guarded by preconditions which tell us under what circumstances they are allowed to occur (i.e. based on given values of observations or inputs) and the postcondition defines which observations change and which do not when the operation occurs as well as describing any output values.

The specification of the nuclear power plant example, therefore, is concerned with observations relating to the items such as reactor pressure and water levels, condenser pressure and water levels, power output, speed of the pumps and status of the valves. A Z specification can be used with theorem provers to prove properties over the entire state space of the system (for example to show that a certain set of observations is never possible if it describes an undesirable condition) and can also be used with model checkers to examine the constrained state space for things such as safety conditions.

6.3.5 *μCharts*

In addition to the Z specification, we also use the visual language, μCharts (Reeve 2005; Reeve and Reeves 2000a, b) (a language used to model reactive systems). PIMs can also be represented as μcharts, which provide additional benefits over a simple PIM (including the ability to compose specific sets of behaviours in different charts via a feedback mechanism and embed complex charts into simple states in order to 'hide' complexity) (Bowen and Reeves 2006a).

μCharts is based on Harel Statecharts (Harel 1987) and was developed by Philipps and Scholz (Scholz 1996; Philipps and Scholz 1998). μCharts was subsequently extended by Reeve (2005), and we use his syntax, semantics and refinement theory. μCharts has a simpler syntax than Statecharts (and in some sense can be considered a 'cut-down' version) and it also has a formal semantics. μCharts and Statecharts differ in terms of synchrony of transitions (we imagine a clock ticking and a step happening instantaneously at each tick), step semantics and the nature of the labels on transitions. Labels on transitions in μcharts (note, we refer to the language as μCharts and the visual representations as μcharts) are of the form $guard/action$, where guards are predicates that trigger a transition if they are true, and also cause actions to take place. For example, if a guard is simply a signal s, then the presence of s in the current step makes the guard true (think of the guard as being the predicate 'the signal s is present'). An example of an action is 'emit the signal t in the current step'. Guards are evaluated and actions happen instantaneously in the same single step; thus, the emission of a signal from one transition which is a guard to another transition results in both transitions occurring in the same step.

The μCharts language includes several refinement theories which in turn gives us refinement theories for PIMs. The trace refinement theory for μCharts is particularly useful as it can be abstracted into a much more lightweight refinement theory for interfaces based on contractual utility (Bowen and Reeves 2006b). The semantics of μCharts is given in Z, and there is a direct translation available (via an algorithm and tool) from a μchart to a Z specification (Reeve and Reeves 2000b) and this in turn means we have an algorithm and means to turn a PIM into a Z specification (Bowen and Reeves 2014).

6.3.6 *Combining the Models*

The models of functionality (specification) and interactivity (presentation model and PIM) are already coupled via the PMR. This gives us a model which combines the conventional use of Z to specify functionality together with the more visually appealing use of charts for the interactivity.

However, we can also combine the models in a way that leads to a single model, all in Z, of the entire system. This gives us the ability to, for example, create a single model of all parts of an interactive system (i.e. interactivity and underlying functionality) that can then be used to prove safety properties about that system which might relate to functional constraints, interface constraints or both (Bowen and Reeves 2013). It might also be used as the basis for refinement, ultimately into an implementation, via the usual refinement theories for Z (Derrick and Boiten 2014; Woodcock and Davies 1996).

We do this single-model building by using the Z semantics of μCharts and by expressing the PMR as a Z relation. This is then combined with the formal specification of the functionality of the system, giving a single model where the transitions representing the PIM are used to constrain the availability of the operations. So if an S-behaviour given in the presentation model is only available in one state of the UI, this is represented in the new combined Z specification as a precondition on the related operation. Recently, we have also given a simplified semantics for the creation of a single specification from the models to reflect models of modal devices, where the PIMs typically do not use the full expressiveness of μCharts (Bowen and Reeves 2014).

6.4 The Nuclear Power Plant Case Study

We now consider the case study and give concrete examples of the modelling techniques described above. From the general overview of the functionality of the nuclear power plant given in the case study (along with several assumptions to fill in the gaps), we can generate a Z specification of the desired functionality. This Z specification gives us a description of the observable states of the system (i.e. its state space) as a Z state schema, along with the operations that can change these states (i.e. move us around the state space). As such, it formally specifies behaviours that can be exhibited by the system.[3] Note that we have simplified the value types for pressure and water levels to natural numbers for convenience of specification.

The observable states of the system can be captured by observing all of the parameters that are described in the case study brief along with any known constraints. For example, we know that for the state of the reactor to be considered ‘safe’, then the reactor pressure must be no more than 70 bar while the core temperature remains below the maximum value of 286 °C. So, we would have to be able to observe these values, and ensure that they are within the required limits, in order to be able to say that we are describing an allowed state of the system.

The state space of the system is given by a state schema like *ReactorControl* below. It declares names and types for all the observations that can be made of parts (or parameters) of the system, and it places constraints on some or all of those observations in order to ensure that the state space contains only safe states. Of course, this is a design decision; it is equally valid to have a state space that includes unsafe values for some observations, as long as this aspect is handled elsewhere in the spec-

[3] Of course, in order to ensure this is actually true, we must also consider the preservation of these properties in the final implementation, but we will not go into a discussion about refinement here.

ification (e.g. perhaps there are specialised error-handling operations that come into play once we enter a state which is allowed by the model but which is unsafe according to the requirements).

The Z snippets below show some of the definitions used to model the system.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{ReactorControl}} \\
\hline
\mathit{sv}1 : \mathit{Valve} \\
\mathit{sv}2 : \mathit{Valve} \\
\mathit{cpUMin} : \mathbb{N} \\
\mathit{pressure} : \mathbb{N} \\
\mathit{temp} : \mathbb{N} \\
\mathit{wv}1 : \mathit{Valve} \\
\mathit{wv}2 : \mathit{Valve}\ \mathit{outputMW} : \mathbb{N} \\
\mathit{waterlevelMM} : \mathbb{N} \\
\mathit{wp}1\mathit{UMin} : \mathbb{N} \\
\mathit{rodposition} : \mathbb{N} \\
\hline
\mathit{pressure} \leq 70 \\
\mathit{temp} < 286 \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{LowerRods}} \\
\hline
\Delta \mathit{ReactorControl} \\
\hline
\mathit{rodposition} > 0 \\
\mathit{rodposition}' = \mathit{rodposition} - 1 \\
\mathit{wp}1\mathit{UMin} = \mathit{wp}1\mathit{UMin}' \\
\mathit{sv}1' = \mathit{sv}1 \\
\mathit{sv}2' = \mathit{sv}2 \\
\mathit{wv}1' = \mathit{wv}1 \\
\mathit{wv}2' = \mathit{wv}2 \\
\mathit{cpUMin}' = \mathit{cpUMin} \\
\mathit{waterlevelMM}' = \mathit{waterlevelMM} \\
\mathit{outputMW}' = \mathit{outputMW} \\
\hline
\end{array}
$$

The *ReactorControl* schema is the state schema showing the observable states of the system (via the parameters or observable values within the system), and *LowerRods* is an operation schema that changes a state—in this case, the position of the rods is the only part of the overall state that changes.

Once we have a complete specification of all allowable behaviours, we can then use a model checker such as the ProZ plug-in for ProB to investigate the specification to ensure that it behaves as expected. For example, if we have specified what it means for the system to be in a stable state, then we should be able to show that when these conditions are not satisfied then the system moves into one of the three error control states—abnormal operation, accident or SCRAM.

$$\begin{array}{|l}
\hline
\textit{Stable} \\
\Xi \textit{ReactorControl} \\
\hline
\textit{cpUMin} = 1600 \\
\textit{waterlevelMM} = 2100 \\
\textit{outputMW} = 700 \\
\hline
\end{array}$$

$$\textit{Status} \mathrel{\widehat{=}} \textit{Stable} \lor \textit{Abnormal} \lor \textit{Accident} \lor \textit{SCRAM}$$

In the same way as we describe the system observations for *Stable*, we also describe the *Abnormal, Accident* and *SCRAM* states so that *Status* is then defined as the disjunction of these possible states.

While this high-level view is essential in enabling us to perform the sorts of proofs we require for a safety-critical system, we may wish to consider subsets of behaviour in more detail. We could always, of course, add to the specification to include all of the details needed to consider these, but it is true that the more we reduce the level of abstraction, the more unreadable (and potentially unwieldy) the specification becomes.

Suppose we wish to consider some specific procedures of the power plant which combine several operations. For example, when the plant starts up or shuts down, there are a series of required steps that must occur along with requirements of values of certain parameters (i.e. conditions on observable values) that enable the required steps to proceed. We could investigate the 'start-up' and 'shutdown' steps using model checking with the Z specification as above; however, in order to more easily consider the user inputs to control these procedures (which are not included in the specification), we might instead create a μchart which shows the required input levels and reactions that occur in these processes.

Figure 6.3 shows the μchart of the'start-up' procedure for the power plant. It provides a different (and more visual) abstraction of the system than the Z specification might be able to and is more expressive in terms of the reactive properties (the Z is intended to describe what is and is not possible and abstracts away such reactive and event-driven behaviour deliberately), but at the same time (via its Z semantics) retains all of the useful properties of a formal specification. In particular, the Z specification, while it tells us precisely what the state space is and what operations can move us around the state space, says nothing about the *sequencing* of operations that are allowed or possible. This is, in part, the role of the μchart. For example, the fact that a valve must be open before it can be closed is best handled by the chart. It could formally be handled via setting up flags and preconditions for Z operations in terms of those flags, but this is an example of the unwieldiness of one language over another: picking the right language for each part of a job is part of the skill of a modeller, and part of the elegance that comes from experience. The fact that the modelling *could* all be done in Z, say, is no reason to actually do it all in Z. If there is a better language for certain parts of the model, in this case the use of a chart for the sequencing of operations, then the elegant solution is to do the modelling in that way.

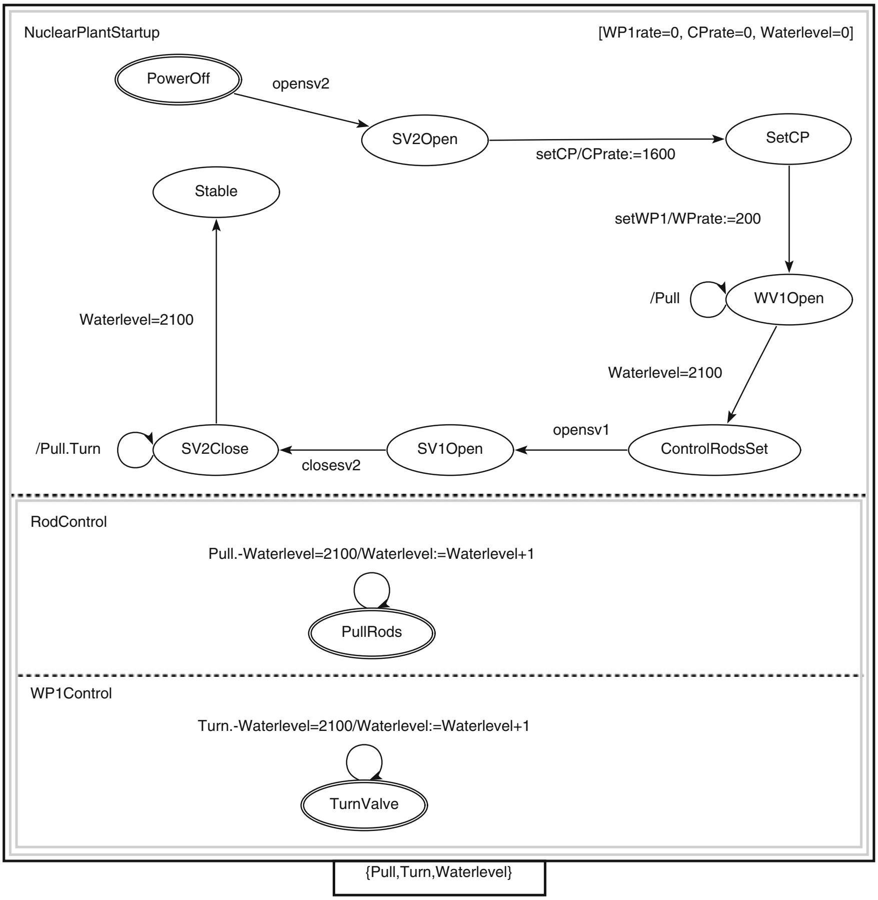


Fig. 6.3 Chart of start-up procedure

The model consists of three atomic μcharts (which are, in this case, just like simple finite-state machines) composed in parallel, which means that they each react in step with each other and signals listed in the feedback box at the bottom of the chart are shared instantaneously between all charts at each step. This modularity is another of the advantages of using μCharts as we can explicitly model relationships between independent components and behaviours. For example, in this chart, we can see that once the system is in the *SV2Close* state, it will remain there until the water level reaches the required value, and at each step, it outputs signals (from the self-loop transition) *Pull* and *Turn*.[4] These signals are then shared with the *RodControl* and

[4] Note that these two transitions happen at *every* clock tick since their guards are true, denoted by convention by the absence of any guard.

WP1Control charts which lead to transitions which ultimately affect the water level until the required value is reached.

The transitions between the various states the system goes through are guarded by required values on key indicators (such as water level and power output) as well as user operations (such as opening and closing valves). In this model, we still do not distinguish between user operations, system controlled operations and functional monitoring of values. In this way, we still abstract from a user's view of the system as we are most interested here in ensuring that the correct outcomes are reached depending on the values and that the components interact properly as shown by the feedback mechanism of the composed charts.

Using the Z semantics of μCharts, and a tool called ZooM[5] which generates the Z specification which expresses the meaning of a μchart, we can go on to model check this component of behaviour to ensure that the system progresses correctly through the start-up procedure (and similarly shutdown) only when the correct preconditions/postconditions are met. Already having these two different, but interrelated, views gives us a consistent mechanism for viewing parts of the system in different ways.

Once we are satisfied that the system will behave correctly as described, we must also ensure that the users can perform the required operations and that at the very least the interface provides the necessary controls (we do not talk about the issue of usability of the interface in this chapter—recall our comment in the introduction about the usual artefacts being available from the UCD process—however, it is of course equally important in ensuring that the system can be used): so we take any design artefacts we have for the user interface to the control system and create presentation models, PIMs and PMR, as described previously.

For the nuclear power plant control system, we start with the initial design shown in Fig. 6.2. The following presentation model and PMR snippet give an example of the models derived from this. There is no PIM at this stage as we are dealing with a single fixed ‘window’ which has no navigational opportunities for the user and so, as mentioned previously, it is trivially a single-state automata.

Presentation Model

PowerDisplay, Responder, (S_OutputPower),
RWaterLevelDisplay, Responder, (S_OutputReactorWaterLevel),
RPressureDisplay, Responder, (S_OutputReactorPressure),
ControlRodCtrl, ActionControl, (S_RaiseControlRods,
    S_LowerControlRods),
WP1Ctrl, ActionControl, (S_IncWaterPressure1,
    S_DecWaterPressure1),
WP2Ctrl, ActionControl, (S_IncWaterPressure2, S_DecWaterPressure2),
CPCtrl, ActionControl, (S_IncCPressure, S_DecCPressure),
SV1Open, ActionControl, (S_OpenSV1),

[5] http://sourceforge.net/projects/pims1/files/?source=directory.

SV1Close, ActionControl, (S_CloseSV1),
SV1Status, Responder, (S_OutputSV1Status)

PMR

S_OutputPower $\mapsto$ OutputPowerLevel
S_OutputReactorWaterLevel $\mapsto$ OutputReactorWaterLevel
S_OutputReactorPressure $\mapsto$ OutputReactorPressure
S_RaiseControlRods $\mapsto$ RaiseRods
S_LowerControlRods $\mapsto$ LowerRods
S_IncWaterPressure1 $\mapsto$ IncreaseWaterPressure
S_DecWaterPressure1 $\mapsto$ DecreaseWaterPressure
S_OpenSV1 $\mapsto$ OpenSV1
S_CloseSV1 $\mapsto$ CloseSV1
S_OutputSV1Status $\mapsto$ OutputSV1Status

For brevity, we do not include all of the status lights and valve controls (e.g. for valves SV2, WV1 and WV2), but the reader can assume they are described in the same manner as the SV1 controls and status display.

The presentation model can be used to ensure that all of the required operations are supported by the user interface, while the PMR ensures that the UI designs are consistent and complete with respect to the functionality of the system. For example, if we have some S-behaviours of the presentation model which do not appear in the PMR, then we know that the interface describes functionality that is not included in the specification and we must therefore address this incompleteness.

We can also use these interface models to help derive alternative (restricted) interfaces for use in error conditions when the user may have only partial control of the system, or when they have no control due to SCRAM mode. Initially, a presentation model of the alternative interfaces provides information about what operations are (and more crucially, are not) available for the user. Subsequently, we can use the refinement theory based on μCharts trace refinement (Bowen and Reeves 2006b) to examine alternatives and prove that they are satisfactory. The visual appearance of the alternative interfaces may be entirely different from the original (although of course we would want as much correspondence between interfaces for the same system as possible to avoid user confusion). The presentation models of the different interfaces allow us to compare behaviours (via the refinement theory), irrespective of the appearances.

The interface in Fig. 6.2 allows the user full control of all aspects of the system, as it occurs when it is in a stable mode. However, if it moves into one of its error states, then the user has a reduced amount of control (or none in SCRAM mode when the system functions in a totally automated fashion). We might propose changes to the interface to support this restricted control and provide feedback to the user about what is happening and what they can, and cannot, do. Figure 6.4 shows a suggested design change to the nuclear power plant for when the control system is in 'Abnormal' mode, and partial automated-only control is in place.

The presentation model for the interface will then differ from that of the original example as the three pump controls are now displays rather than controls (they show

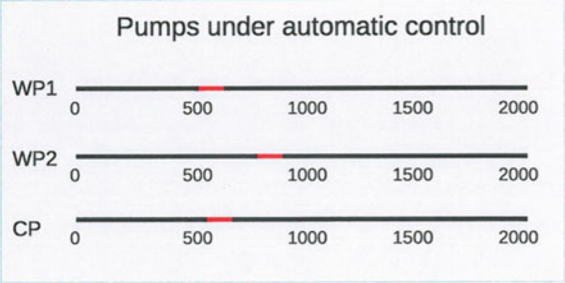


Fig. 6.4 Restricted interface for pump controls

the user what the automated system is doing rather than enabling the user to make changes themselves). The widgets are still associated with the same behaviours, but instead of generating these behaviours (as action controls do), they now respond to them. In addition, we include the automated behaviour (which drives the mode switch), described as a 'SystemControl', which leads to the interface behaviours of changing the display.

WP1Ctrl, Responder, (S_IncWaterPressure1, S_DecWaterPressure1)
WP2Ctrl, Responder, (S_IncWaterPressure2, S_DecWaterPressure2)
CPCtrl, Responder, (S_IncCPressure, S_DecCPressure)
Status, SystemControl, (S_Stable, I_Stabilised)

Similarly we add the automated behaviour to the original presentation model to make explicit the automation which switches into abnormal mode.

Status, SystemControl, (S_Abnormal, I_AbnormalOperation)

It is important to ensure that this new interface provides exactly the right controls and restrictions to the user, and also that they are only present in the relevant error states (i.e. that a user is not restricted when the system is stable).

For each of the possible error states, Abnormal, Accident and SCRAM, we can provide different interfaces which provide only the correct levels of user interaction. Figure 6.5 shows the PIM for the new collection of interfaces, including those of the other error modes (Accident and SCRAM), although we do not discuss their designs here.

The *Stable* state is considered the initial state (indicated by the double ellipse) and the transitions indicate possible movements between states. The *SCRAM* state is a deadlock state, in that there are no possible transitions out of this state. This is correct for this system as the user cannot interact with the system in this mode and the only possible behaviour for the system is to go into a safe shutdown mode.

There are several types of properties we may wish to consider once we have these designs and their associated models. First, we should ensure that the combination of new interface models still adhere to the original requirements. Second, as stated

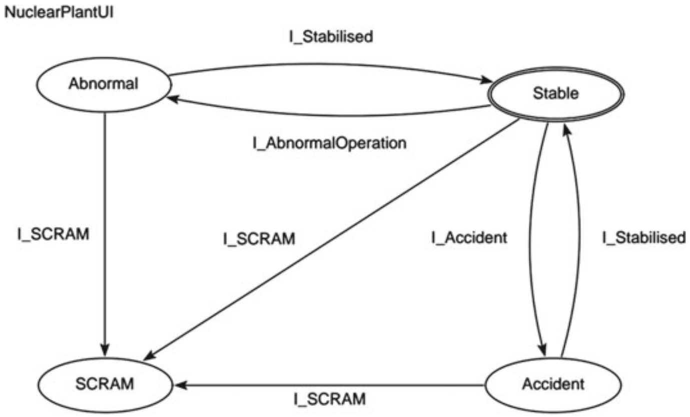


Fig. 6.5 PIM of interface modes

above, we should be sure that the correct level of control/restriction is provided in each instance.

Typically, when we make changes to the interface or interaction possibilities of our system during the modelling stage, we would use refinement to ensure the adherence to original requirements. However, what we are doing with the new interfaces is to restrict behaviour, so it is not the case that each of the different (new) modes refines the original, but rather that the total interface model (the concatenation of the four *pmodels*) refines the original. We reiterate that we retain all of the visual designs of layout, etc. for both the original as well as the new UIs so that we can always refer back to these to understand more about the actual appearance. This will, of course, be crucial when we come to evaluate the usability aspects of the UIs. In terms of the final considerations of refinement, however, we rely on the models alone.

Refinement for interface and interaction properties described in presentation models and PIMs is based on the notion of contractual utility (Bowen and Reeves 2006b) and can be described by relations on the sets of behaviours of the models in a simple way, while being formally underpinned by the trace refinement theory of μCharts. Given two arbitrary interfaces, A and C, the requirements for the two types of behaviours are as follows:

$$\mathrm{UI}_A \equiv_{SBeh} \mathrm{UI}_C$$
$$\mathrm{I\_Beh}[\mathrm{UI}_A] \subseteq \mathrm{I\_Beh}[\mathrm{UI}_C]$$

where I_Beh[P] is a syntactic function that returns identifiers for all I-behaviours in P.

We call the first interface (from Fig. 6.2) ‘Original’ and the new interfaces (a combination of ‘Original’ with the addition of the automation and ‘Abnormal’) ‘New’. We use the syntactic functions to extract behaviours to create the following sets:

I_Beh[Original] = { }
S_Beh[Original] = {S_OutputPower, S_OutputReactorWaterLevel, S_OutputReactorPressure, S_RaiseControlRods, S_LowerControlRods, S_IncWaterPressure1, S_DecWaterPressure1, S_IncWaterPressure2, S_DecWaterPressure2, S_OpenSV1, S_OpenSV2, S_OutputSV1Status}
I_Beh[New] = {I_AbnormalOperation, I_Stabilised}
S_Beh[New] = {S_OutputPower, S_OutputReactorWaterLevel, S_OutputReactorPressure, S_RaiseControlRods, S_LowerControlRods, S_IncWaterPressure1, S_DecWaterPressure1, S_IncWaterPressure2, S_DecWaterPressure2, S_OpenSV1, S_OpenSV2, S_OutputSV1Status, S_Stable, S_Abnormal}

The requirement on the I-behaviours permits the addition of new behaviours, and so, this is satisfied. However, notice the inclusion of the S-behaviours for the automation to switch the system between states of the interface (S_Stable and S_Abnormal). These are now implicit behaviours of the interface and so must also be included, but these additional behaviours break the requirement on equality between sets of S-behaviours of the interfaces. If we consider this further, we can understand why this is a requirement. Our new interface depends on behaviours to switch between modes under different states of the system which were not part of the original description; as such, there is no guarantee that this behaviour is described anywhere in the specification. We can see from the PMR that there is no relation between these operations and the specification, so we have added functionality to the interface that may not (yet) be supported by system functionality. This is an indication that we need to also increase behaviours in the functional specification which will allow these to then be supported, or we need to ensure that we can relate any existing specified operations (via the PMR) to the new S-behaviours. In fact, we have already discussed earlier how we can describe the different states of the system (Stable, Abnormal, SCRAM, etc.) in the specification, so we must now ensure that the identification of these states along with the necessary behaviour is also described.

Now we can investigate the behaviours of the component *pmodels* (of each different mode of use) via translation of the PIM (to Z) and its corresponding specification to ensure that these correspond to permissible user behaviours in each of the system states. We discuss the combination of models which provides this next.

6.4.1 Benefits of Combining the Models

In the previous sections, we have given examples of the use of the individual models to consider some properties of the nuclear power plant control system which might be of interest during the development process. There are some things, however, which require a combined model of both UI and functionality in order to formally consider. Suppose we want to ensure that when the system is in *Abnormal* mode, the user cannot alter the water pressure (as this is one of the elements under automatic control in this mode). Using the ProB model checker, there are several different ways to perform such analysis, and we have found that describing the properties as LTL formulae and then checking these are a useful mechanism as ProB provides counterexamples consisting of a history list of operations performed when a formula check fails (Bowen and Reeves 2014).

However, the functional specification alone cannot be used for this. If we perform some analysis to show that when the status of the system is *Abnormal*, the operations to increase or decrease the water pressure are not enabled we find that this is not true. Of course, this is exactly as it should be; although the user cannot change the water pressure, it can still be changed (via automation) and our functional model correctly describes this.

In order to consider the possible effects of user interactions, therefore, we need to combine the interface models with the specification. We start by declaring types for the states of the PIM and the transition labels.

$$
\begin{array}{l}
State ::= Stable \mid Abnormal \mid Accident \mid SCRAM \\
Signal ::= I\_AbnormalOperation \mid I\_Accident \mid I\_SCRAM \mid I\_Stabilised
\end{array}
$$

Next, we give a description of the UI which consists of a single observation which shows the state the UI is in (one of the states of the PIM) and create an operation schema for each of the transitions which describes the change in state that occurs on a given signal, for example:

$$
\begin{array}{|l}
\hline
\textit{TransitionStableAbnormal} \\
\Delta UI \\
signal? : Signal \\
\hline
signal? = I\_AbnormalOperation \\
currentState = Stable \\
currentState' = Abnormal \\
\hline
\end{array}
$$

This describes how the observation of ‘currentState’ changes when the transition from ‘stable’ to abnormal’ occurs, which requires the ‘I_AbnormalOperation’ input signal to be present.

The final step is to create a combined schema for the system and UI (so we include the schema descriptions for each into a single schema), and then, for each of the operation schemas, we add a precondition which gives the required state of the UI, i.e. shows when the operation is available to a user. Now when we model check the specification and check the condition of whether a user can change the water pressure when the system is in an abnormal state, we find that they cannot, as none of the user operations which change the water pressure are enabled if the system state is abnormal.

In order to create a fuller picture, we should include all of the automation considerations. We can do this by describing automatic control in exactly the same way as we have shown above. That is, we create the alternate set of models including automated interface behaviours (as if they were user controls) and then we can include this with the combined model. This new model then enables us to show that the correct levels of manual (human user) or automatic control occur in all of the different states of the system.

6.5 Conclusion

In this chapter, we have described the different models we use for reasoning about interactive systems. Using the nuclear power plant control system as an example, we have shown how the models can be used independently as the differing expressive natures of the languages involved mean that they are individually suitable for different tasks in the process of verifying and validating safety-critical interactive systems. We have also given an example of combining the models into a single formal description, which allows us to ensure correctness of the interactivity and interaction in combination with the functionality.

Although the type of interface used in the nuclear power plant control example consists of standard desktop system controls (buttons, sliders, etc.), this is not a requirement for our methods. Any type of interaction control (speech, touch, gesture, sensor, etc.) can be modelled in the same way as we can abstract them in the presentation models as event generating (action controls) or responding (responders) as shown here.

References

Bolton ML, Bass EJ (2010) Formally verifying human-automation interaction as part of a system model: limitations and tradeoffs. Innov Syst Softw Eng A NASA J 6(3):219–231

Bowen J, Reeves S (2006a) Formal models for informal GUI designs. In: 1st international workshop on formal methods for interactive systems, Macau SAR China, 31 October 2006. electronic notes in theoretical computer science, Elsevier, vol 183, pp 57–72

Bowen J, Reeves S (2006b) Formal refinement of informal GUI design artefacts. In: Australian software engineering conference (ASWEC'06). IEEE, pp 221–230

Bowen J, Reeves S (2008) Formal models for user interface design artefacts. Innov Syst Softw Eng 4(2):125–141

Bowen J, Reeves S (2013) Modelling safety properties of interactive medical systems. In: 5th ACM SIGCHI symposium on engineering interactive computing systems, EICS'13. ACM, pp 91–100

Bowen J, Reeves S (2014) A simplified Z semantics for presentation interaction models. In: FM 2014: formal methods—19th international symposium, Singapore, pp 148–162

Courtney A (2003) Functionally modeled user interfaces. In: Interactive systems. design, specification, and verification. 10th international workshop DSV-IS 2003. Lecture notes in computer science, LNCS. Springer, pp 107–123

Derrick J, Boiten E (2014) Refinement in Z and object-Z: foundations and advanced applications. Formal approaches to computing and information technology, 2nd edn. Springer

Dix A, Runciman C (1985) Abstract models of interactive systems. Designing the interface, people and computers, pp 13–22

Duke DJ, Harrison MD (1995) Interaction and task requirements. In: Palanque P, Bastide R (eds) Eurographics workshop on design, specification and verification of interactive system (DSV-IS'95). Springer, pp 54–75

Duke DJ, Faconti GP, Harrison MD, Paternò F (1994) Unifying views of interactors. In: Advanced visual interfaces, pp 143–152

Duke DJ, Fields B, Harrison MD (1999) A case study in the specification and analysis of design alternatives for a user interface. Formal Asp Comput 11(2):107–131

Harel D (1987) Statecharts: a visual formalism for complex systems. Sci Comput Program 8(3):231–274

Harrison MD, Dix A (1990) A state model of direct manipulation in interactive systems. In: Formal methods in human-computer interaction. Cambridge University Press, pp 129–151

Henson MC, Deutsch M, Reeves S (2008) Z Logic and its applications. Monographs in theoretical computer science. An EATCS series. Springer, pp 489–596

Hussey A, MacColl I, Carrington D (2000) Assessing usability from formal user-interface designs. Technical report, TR00-15, Software Verification Research Centre, The University of Queensland

ISO, IEC 13568 (2002) Information technology-Z formal specification notation-syntax, type system and semantics. International series in computer science, 1st edn. Prentice-Hall, ISO/IEC

Jacob RJK (1982) Using formal specifications in the design of a human-computer interface. In: 1982 conference on human factors in computing systems. ACM Press, pp 315–321

Limbourg Q, Vanderdonckt J, Michotte B, Bouillon L, López-Jaquero V (2004) UsiXML: a language supporting multi-path development of user interfaces. In: 9th IFIP working conference on engineering for human-computer interaction jointly with 11th international workshop on design, specification, and verification of interactive systems, EHCI-DSVIS'2004, Kluwer Academic Press, pp 200–220

Paternò FM (2001) Task models in interactive software systems. Handbook of software engineering and knowledge engineering

Paternò FM, Sciacchitano MS, Lowgren J (1995) A user interface evaluation mapping physical user actions to task-driven formal specification. In: Design, specification and verification of interactive systems. Springer, pp 155–173

Philipps J, Scholz P (1998) Formal verification and hardware design with statecharts. In: Prospects for hardware foundations, ESPRIT working group 8533. NADA—new hardware design methods, survey chapters, pp 356–389

Puerta A, Eisenstein J (2002) XIML: a universal language for user interfaces. In: Intelligent user interfaces (IUI). ACM Press, San Francisco

Reeve G (2005) A refinement theory for μcharts. PhD thesis, The University of Waikato

Reeve G, Reeves S (2000a) μ-charts and Z: examples and extensions. In: Proceedings of APSEC 2000. IEEE Computer Society, pp 258–265

Reeve G, Reeves S (2000b) μ-charts and Z: hows, whys and wherefores. In: Grieskamp W, Santen T, Stoddart B (eds) Integrated formal methods 2000: proceedings of the 2nd international workshop on integrated formal methods. LNCS, vol 1945. Springer, pp 255–276

Reichart D, Dittmar A, Forbrig P, Wurdel M (2008) Tool support for representing task models, dialog models and user-interface specifications. In: Interactive systems. Design, specification, and verification: 15th international workshop, DSV-IS'08. Springer, Berlin, pp 92–95

Scholz P (1996) An extended version of mini-statecharts. Technical report, TUM-I9628, Technische Univerität München. http://www4.informatik.tu-muenchen.de/reports/TUM-I9628.html

Thimbleby H (1990) Design of interactive systems. In: McDermid JA (ed) The software engineer's reference book. Butterworth-Heineman, Oxford, Chap, p 57

Thimbleby H (2004) User interface design with matrix algebra. ACM Trans Comput Hum Interact 11(2):181–236

Woodcock J, Davies J (1996) Using Z: specification, refinement and proof. Prentice Hall

Chapter 7
Activity Modelling for Low-Intention Interaction

Alan Dix

Abstract When modelling user interactions, we normally assume that the user is acting with intention: some very explicit such as opening a valve in a nuclear power station, others more tacit, hardly needing any thought, for example tipping a tablet to turn a page. However, there are also a range of system behaviours that make use of unintentional user actions, where the user acts, but the system decides that the action has meaning, and how to make use of that meaning. Again, these may operate on a variety of levels from 'incidental interactions', which operate entirely without the user realising, perhaps subtle changes in search results based on past activity, to more 'expected interactions' such as automatic doors that open as you approach. For intentional interaction, there is long-standing advice—making sure that the user can work out what controls do, where information is, interpret the available information, receive feedback on actions—and also long-standing modelling techniques. Low-intention interactions, where the system has more autonomy, require different design strategies and modelling techniques. This chapter presents early steps in this direction. Crucial to this is the notion of two tasks: the sensed task, which the system monitors to gain information and the supported task, which the system augments or aids. First, this chapter demonstrates and develops techniques in the retrospective modelling of a familiar low-intention interaction system, car courtesy lights. These techniques are then applied proactively in the design of a community public display, which is now deployed and in everyday use.

A. Dix (✉)
School of Computer Science, University of Birmingham, Birmingham, UK
e-mail: alan@hcibook.com

A. Dix
Talis Ltd. Birmingham, Birmingham, UK

B. Weyers et al. (eds.), *The Handbook of Formal Methods in Human-Computer Interaction*, Human-Computer Interaction Series,
DOI 10.1007/978-3-319-51838-1_7

7.1 Introduction

Mostly, when modelling user interactions, we assume that the user is acting with intention. Indeed, the presence of a goal is central to Norman's (1990) influential 'seven stages' model. Traditional hierarchical task analysis also starts with a top-level goal, which then leads to a set of tasks to achieve that goal and sub-tasks of those top-level tasks. In a more dynamic fashion, means-end analysis decomposes by creating sub-goals when plans encounter impasses. Sometimes, these intentional actions are very explicit, for example opening a valve in a nuclear power station to alter pressure in the containment vessel; some are 'implicit', hardly needing any thought, for example tipping a tablet to turn a page.

However, there is also a range of system behaviours that make use of unintentional user actions, or to be precise where the use made by the system is not the primary intention of the action. The most extreme are '*incidental interactions*'. In these situations, the user's intention and actions are focused on a primary goal. Because of these actions, the system is able to gather some direct or sensed information, which can then be used to help the user or other agents achieve some secondary goal. Somewhere between these and fully intentional interaction are '*expected interactions*', for example walking into a room and the automatic light coming on. You usually enter the room because you want to be inside, but expect that the light will come on and would be surprised if it did not.

For intentional interaction, there is long-standing advice: make sure that the user can work out what controls do, where information is, interpret the available information and receive feedback on actions (e.g. Nielsen's (1993) ten heuristics or Shneiderman's (1998) golden rules). While some of these design principles and heuristics need interpretation by a graphical, interaction or user experience designer, others can be operationalised in well-known formal models and formalisations of usability principles (e.g. variants of visibility in Dix 1991).

Low-intention interaction is more problematic, as it is not so much a matter of presenting the user with controls and behaviours that are as clear as possible, but instead interpreting the user's actions performed for a variety of other purposes.

This chapter shows how a level of activity modelling can be used in order to ascertain actions or signs that may already be sensed, or where additional sensors can be added, so that this provides sufficient information for informed system action. Context-aware systems do this by creating very explicit system models of user actions. This chapter, in contrast, will focus on designer models of user activity, which can then be analysed to reveal potential triggers for automated action. Crucially, these models need to adopt some level of probabilistic reasoning, although this may be qualitative.

This chapter starts by explaining in more detail what is meant by 'low-intention interaction', using a series of examples, and this includes the two tasks (sensed and supported) mentioned above. This is followed by a short review of related literature including notions of implicit interaction, natural interaction and architectures and tools to support the design and construction of sensor-rich systems. This leads into a further discussion of design issues for low-intention interaction, notably the way user models do not necessarily embody the same notions of transparency and feedback of intentional systems, and the implications for privacy. Finally, this is brought to bear on the explicit design of low-intention systems centred around the separate analysis of the sensed task (in order to determine what can be known of the users' actions) and of the supported task (in order to determine what are desirable interventions). The initial motivating example will be car courtesy lights (previously described in Dix et al. 2004; Dix 2006), as the technique is most well suited for non-safety critical applications. However, it is then applied to the design of a deployed public display system, which has since been in operation for several years.

The underlying methods used in this chapter date back over ten years and parts draw heavily on early work in incidental interaction (Dix 2002), and its incorporation in the author's HCI textbook (Dix et al. 2004) and online material (Dix 2006) on design methods for incidental interaction. Some of the concepts and methods are developed further and made explicit, but the principal contribution of this chapter is the reporting of how the techniques were applied in practice in the design of TireeOpen, the Internet-enabled open sign.

7.2 What Is Low-Intention Interaction?

7.2.1 Intentional and Low-Intention Interaction

As noted, traditional user interaction principles have focused on the user being in control of precisely what the computer system does. This is at the heart of direct manipulation (Shneiderman 1982; Hutchins et al. 1986); the system state is represented faithfully in the user interface, and as the user modifies the representation, the system immediately updates accordingly. Norman's seven-stage model of interaction starts with a goal, and the user translates this into actions on the system and then evaluates the results of the action. Users are assumed to know what they want to do, and the computer's job is to perform the actions requested as reliably and transparently as possible.

Intentional systems may use intelligent algorithms. For example, the Kinect uses complex vision processing to work out your body position and gestures, but this is in order to be able to accurately understand your intended movements, say to swing a golf club. The goal and the intent lie clearly with the user, and the system merely works to enact or interpret the user's intention.

In contrast, some autonomic systems operate at such a level that the user may be unaware that they are doing anything at all. For example, an intelligent heating system may use mobile phone GPS and other sensors to work out when you are due home, whether you have been walking through the rain, doing a workout at the gym or merely driving back. If the heating system is good enough at predicting the right temperature, you may be completely unaware that it has been making adjustments.

In such systems, the user has no little or no explicit intention; if there is intent, it is on the system's part.

7.2.2 *The Intentional Spectrum*

Incidental interaction was coined to describe an extreme form of low-intention interaction:

> Incidental interaction—where actions performed for some other purpose or unconscious signs are interpreted in order to influence/improve/facilitate the actors' future interaction or day-to-day life (Dix 2002).

In fact, there is a spectrum with explicitly intended actions at one end and incidental interaction at the other extreme (see Fig. 7.1). In between are 'expected' system actions, for example if you work in an office building with automatic lights, you expect the lights to turn on as you enter a room, even though you do not explicitly flick a switch.

Near to the intentional end are actions such as tipping an e-book reader to turn a page as well as very explicit actions such as pressing a button. At this intentional end, we find actions that are invoked entirely by the user, but are low awareness as they are automatic or even autonomic; when reading a (physical) book you are rarely explicitly aware of turning the page, similarly when using a tool you are often focused on the work at hand, not the tool itself, Heidegger's (1927) 'Ready at hand'.

7.2.3 *Examples of Low-Intention Interaction*

Many research systems include aspects of low-intention or incidental interaction.

The Pepys system in Xerox EuroPARC used infrared badges to track researchers in the office and then created automatic diaries at the end of each day (Newman et al. 1991). Here, the primary, intentional task was simply to walk to a colleague's office, but, incidentally, the location was tracked and the diary produced.

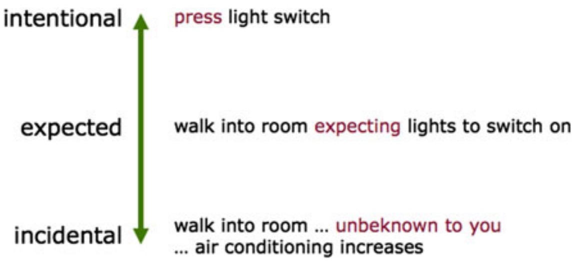

intentional
press light switch
expected
walk into room expecting lights to switch on
incidental
walk into room ... unbeknown to you
... air conditioning increases

Fig. 7.1 Continuum of intentionality (from Dix et al. 2004)

One of the defining ubiquitous computing applications was MediaCup (Beigl et al. 2001; Gellersen et al. 1999). MediaCup added a small sensor pack to the base of ordinary mugs measuring pressure (whether the cup was full or empty), temperature and tip sensors. As the cup owner filled and drank their coffee, the system collected the sensor data and relayed this to others who could then build an idea of the cup owner's activity and availability. Again, the primary (sensed) task was drinking the coffee, but, incidentally, other people were able to improve their interpersonal interactions.

In the author's own work in the late 1990s, onCue provided an intelligent desktop task bar (Dix et al. 2000b). Whenever the user cut or copied things into the clipboard (primary task), onCue analysed the clipboard contents and then suggested possible additional things that could be done using the clipboard content on the Internet or desktop. For example, when you copied a postcode, onCue would suggest Web-based mapping services, or when you copied tabular data, onCue would suggest Web graphing tools or copying into Excel on the desktop.

Even in the age of the Internet of Things (IoT), Internet-enabled mugs are not common. However, incidental interactions are common in day-to-day life.

As you walk towards a building, the doors open, lights go on as you enter a room, toilets flush as you leave the cubicle, and when you get into your car, the courtesy lights go on. In each case, your primary task is simply going in or out of a room, car or toilet cubicle, but the system senses aspects of your primary activity and then performs additional actions aimed to help you. Location-aware apps in phones and wearables use our movements in the environment to enhance secondary goals: telling friends where we are, recording fitness information.

To some extent, this trend is still accelerating as the availability of low-power, small-sized and, crucially, cheap sensors and networking is only just making the visions of 1990s ubiquitous computing commercially possible (Greenfield 2006).

However, in the purely digital domain, where sensing is simply a matter of logging digital interactions, these interactions abound. When we use a shopping site, our primary task may be to buy a particular book or device, but incidentally, the data collected is used to enhance recommendations for other shoppers. When we search, our preferences for clicking through on certain topics (primary task) are often analysed to improve subsequent search result ranking. Possibly less welcome, we also know that our visits to many sites are tracked, sometimes by hundreds of tiny 'beacons', which are then used to gather marketing information and channel particular advertisements to us.

7.2.4 Intentional Shifts

We have seen that there is a continuum between incidental interaction, where users may have no awareness that information is being sensed or the way this is enhancing their interactions, to fully intentional interaction, where users explicitly control a system to fulfil their goals.

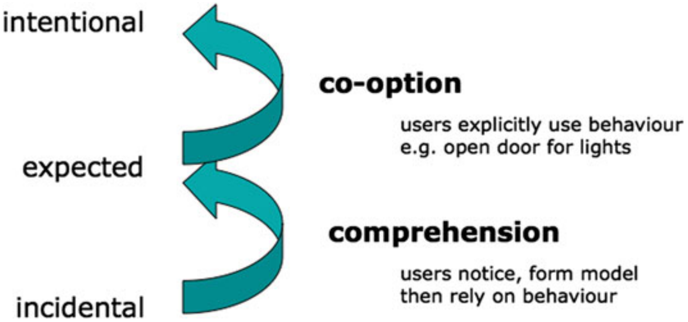


Fig. 7.2 Fluidity of intentionality (from Dix et al. 2004)

Sometimes, parts of a system, or even our own actions of which we are normally unaware, may become more apparent. This might be because some part of the system does not behave seamlessly. Heidegger's (1927) 'breakdowns' occur precisely when more tacit actions are forced into conscious attention, for example if a hammer head works loose. At this point, one shifts into a fully intentional form of interaction.

Even where a system is working correctly, we may become aware of its behaviour. As we begin to understand the rules of sensor-based systems, actions that were entirely below awareness (incidental) may become expected. For example, if the lights in a building operate automatically based on sensors, you may not explicitly intend the lights to switch on when you enter, but you certainly do not expect to be left in darkness.

Once we understand the rules well enough to be 'expected', we may co-opt the system behaviour to exert explicit control over what were originally intended to be incidental interactions. For example, you might open and close a car door to get the courtesy lights to turn on for a longer period, or wave your arms to activate the movement sensor if you want the lights to turn on (see Fig. 7.2).

7.2.5 Two Tasks

As mentioned previously, it is important to recognise that there are two different user tasks to consider:

sensed task—the task related to the user's primary goal during the behaviour that is being sensed by the systems and

supported task—the task that is in some way supported or enhanced by the information inferred from the sensor data.

Sometimes, these are entirely different tasks, for example with the MediaCup (Beigl et al. 2001; Gellersen et al. 1999), the sensed task is about drinking coffee while the supported task is about meeting at appropriate times. However, in other cases, it may be that these are part of the same overall activity (indeed, in the car courtesy light, in the next section, this is precisely the case). Even when the two are actually the same or related tasks, we are using the task analysis in different ways.

In the case of the sensed task, the user's primary goal is not necessarily being supported, but enables us to interpret the sensors, turning raw data into behavioural information. For example, during normal use, if a car door is opened after the car has been stationary for a period, it is likely that someone has got into the vehicle.

Sometimes, this sensing is purely passive, but we may choose to modify the systems around this sensed task. Where the sensed data is insufficient to interpret behaviour, we may choose to add sensors, for example adding an infrared movement sensor to a car interior to be sure that passengers are present. Alternatively, we may choose to modify the actual user interaction on the sensed task.

For example, consider a library website. We have two design alternatives:

(i) show full information as lots of book 'cards' in a large scrollable page, rather like Pinterest (https://about.pinterest.com/).
(ii) show shorter descriptions (title + teaser) but where the full book details appear as a pop-up when the user hovers over or clicks an item.

Let us assume that you decide (i) is more usable, but that the difference is slight. However, from (ii), it is far easier to identify the user's interests. You may then deliberately decide to adopt (ii), even though it is slightly less usable for the sensed task, because you know that by having more information about the user, the system is better able to make suggestions. In other words, you may choose to trade usability between the sensed and supported tasks.

Turning to the supported tasks, there are various ways in which this support can occur.

While the sensing may be low intention, the information gathered by this may be explicitly presented to the user. For example, onCue monitors the user cutting and copying to the clipboard (the sensed task) and infers the kind of thing in the clipboard (e.g. postcode, personal name, table of numbers); this is all automatic, without user attention (Dix et al. 2000b). However, it then alters the toolbar to show actions that the user can perform using the clipboard contents. The user explicitly interacts with the toolbar; that is the supported task is intentional.

In other cases, system modifications to the supported tasks may also be low attention and/or low intention. For example, the order of search results in the library system may be subtly altered based on inferred interest, or the heating in the room may adjust to be slightly warmer when you are sitting and cooler when you are busily moving around.

Note that low intention and low attention are related but different properties. The level of intention is about the extent to which the user controls the initiation of actions, whereas levels of attention are about how much conscious focus is directed towards actions and their results. A system action might be autonomous and unexpected (not the user's intention), but still very obvious and salient (high attention). Likewise, a user's action might be fully intentional but low awareness, for example drinking from a cup of tea while chatting.

7.3 Frameworks and Paradigms

There have been a number of models and frameworks that, in different ways, help understand, analyse, design or construct sensor-rich user interactions. In this brief overview of the most related literature, we will first look at various design concepts in the area, followed by a more in-depth analysis of the notion of 'naturalness' as this is closely related to, but distinct from, low attention. Finally, we look at some of the related architectural frameworks and modelling notations.

7.3.1 Design Concepts

In the 1990s, as ubiquitous computing began to mature, the concept of *context-aware computing* emerged (Schilit et al. 1994; Schmidt 2013). Whereas, in traditional computer applications, user input was interpreted solely in terms of the state of the system, context-aware systems looked to the environment or context. Dey defined this context as:

> any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves (Dey 2001).

Schilit et al.'s (1994) early identification of context-aware applications was inspired by PARCTAB, a deployment of Olivetti infrared tracking badges deployed at EuroPARC (Want et al. 1995). More generally, while other kinds of context were considered, mobility and location were often the defining contexts for early work in the area (Dix et al. 2000a). This in turn led to a number of related interaction-focused design concepts.

Schmidt developed the design concept of *implicit interaction* partly to deal with interactions such as tipping a mobile device to turn a page or move a map.

> Implicit human computer interaction is an action, performed by the user that is not primarily aimed to interact with a computerized system but which such a system understands as input (Schmidt 2000).

In many ways, this can be seen as a precursor to the recent focus on, so-called, *natural user interfaces* or NUIs (Wigdor and Wixon 2011). NUIs are characterised by the detection of ordinary human actions such as gaze direction, body movement, or touch, for example touch tables or Kinect games. The use of the term 'so-called' at the start of this paragraph is because the 'naturalness' of NUIs is often challenged; while the actions are natural ones, the ways in which these are interpreted are often far from natural (Norman 2010).

Ju and Leifer (2008) developed a design framework around Schmidt's notion of *implicit interaction*. The issue of attention is important in their framework, which is particularly focused around two interaction distinctions: attentional demand (foreground vs background) and initiative (reactive vs. proactive).

The concept of *Kinetic User Interfaces* (KUI) emerged during the uMove project (Pallotta et al. 2008) inspired partly by the author's work on *incidental interaction* (Dix 2002). KUIs are where either direct bodily movement, or the user's movement of objects in the environment, is sensed and used for 'unobtrusive' interactions. Like implicit interaction, KUIs embody a particular kind of low-intention interaction.

Wilde et al. (2010) reviewed interaction patterns for pervasive systems based on Alexander et al.'s (1977) architectural pattern language and its uses within software engineering (Gamma et al. 1995). Wilde et al. break down pervasive interaction patterns into three broad classes: interactions with mobile systems, intelligent environments and collaborative work, although many pervasive interactions will of course include elements of each. Forbrig et al. (2013) also consider models and patterns for smart environments building on their concept of *supportive user interfaces*. The patterns are more domain specific than Wilde et al.'s, focusing on smart meeting rooms as an example application area.

Much of the early conceptual work in ubicomp and context-aware computing was primarily descriptive or focused on structuring implementation. The *expected, sensed and desired* framework (ESD) sought to transform much of the practical experience of creating instances of context-aware systems into a more structured ideation and early process (Benford et al. 2005). Expected movements are those that a person might naturally do (e.g. twist a screen, or move in the environment to see something better); sensed movements are those that can be detected by some sort of existing, or easy to add, sensor; and desired movements are the actions you might like to perform (e.g. zoom a display to see more detail).

By making this distinction, it is possible to compare them, for example identifying expected movements that are not sensed suggesting the potential for adding new sensors, or expected movements that can be sensed, which might be used to control desired actions.

Note, in incidental interaction terms, the expected and sensed movements are primarily concerned with the primary goal (sensed task), whereas the desired movements concern the supported task. Also, while the earliest work leading to this chapter predated ESD, aspects of the formulation later in this chapter are influenced very much by ESD.

7.3.2 Low Intention and Naturalness

The concepts of implicit interaction and natural user interfaces (NUIs) are particularly relevant to low-intention interaction. Figure 7.3 shows some of the distinctions they raise.

At the top left are the most artificial actions, for example when you come to a new interface and have to work out which of the new icons or menu choices you need to use. However, with practice, they become second nature (bottom left), and you may not even be aware that you are doing them.

On the far bottom right are the 'most natural' interactions, those that you never have to think about at all. These may be truly instinctive; one particular case of this is where there is a 'natural inverse' (Ghazali and Dix 2006) (e.g. push/pull, twist left/twist right); in such cases, users automatically do the opposite action when they 'overshoot' a target, for example correcting on a steering wheel. Often they are themselves learnt, for example swinging your arm is in a sense an unnatural action to control a computer simulation, but when faced with a Kinect and an image of a golf ball, it recruits already learnt physical actions.

Many NUI interactions are not so obvious. Early research on touch tables found that given free choice, most people used the same dragging, twisting and stretching actions, but beyond these, there was little agreement. However, after a period of use, even the less intuitive gestures become automatic, and perhaps more quickly learnt than abstract icons, as they are able to employ kinesthetic or other forms of sensory memory.

NUIs cover the whole of the right-hand side of the figure, whether immediately obvious and later learnt, or so immediate you never know you are doing them. The early examples of implicit interaction are in this space, notably the turning of pages by tipping a device.

From a low-/high-attention point of view, actions may be very artificial (e.g. shifting gears in a car), but so automatic that you are unaware you are doing them,

However, when we consider the sensed and supported tasks of incidental interaction, in fact, even very explicit and artificial actions may be part of a wider low-intention interaction.

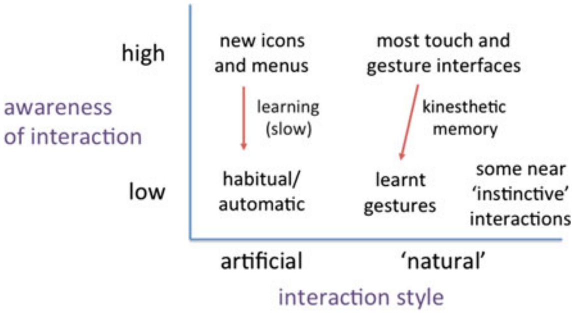


Fig. 7.3 Various forms of natural interaction

Consider an example of a sensed task. When you browse on Amazon, this is an explicit action and may include some high-awareness actions (what shall I search for) and some low-attention learnt interactions (clicking through for more details). However, all of this may be used by Amazon to build its profile of you; the actions are explicit, but the way information is collected about them is not.

Similarly, in the supported task, there may be entirely autonomous actions that the system executes based on your past behaviour (top right of Fig. 7.4, Ju and Leifer's (2008) 'proactive' initiative). For example, an intelligent heating system might automatically set the temperature just right for when you enter your home in the evening. However, you might also execute an explicit action, the effect of which is modified based on previous behaviour or current context (bottom right Fig. 7.4, Ju and Leifer's (2008) 'reactive' initiative). For example, if you hit 'cooler' on the heating system, this might be interpreted as a short blast of cool air, but gradually returning to normal if the system knows (say from phone sensors) that you have just come in after running hard, but behave differently if it is pressed after you have been at home for a while.

That is the key difference for the supported task is whether the action is fixed or contextual based on previous sensed behaviour.

7.3.3 Architecture and Modelling

Dey's definition of 'context' quoted above was influenced by the development of the context toolkit (Salber et al. 1999), which was important in signalling the movement from more specific bespoke developments towards more principled design and reusable architecture. The author's own work in the area included the development of the onCue context-sensitive toolbar, which was built on top of an agent-based framework, *aQtive space*, specifically architected to support context-sensitive interaction (Dix et al. 2000b).

Fig. 7.4 Supported actions

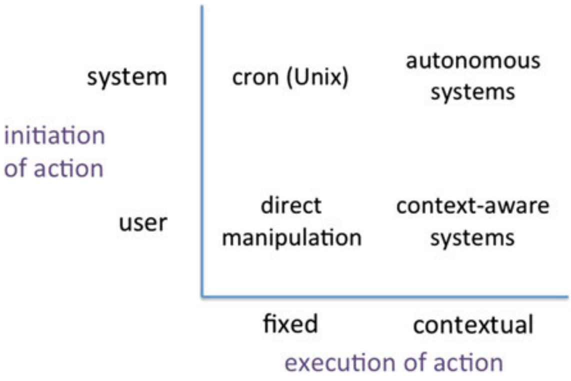


Fig. 7.5 Context description from Schmidt (2000)

```
<context_interaction>
    <context>
        <group match='one'>
            sensor_module.touch
            pilot.on
        </group>
        <group match='none'>
            sensor_module.alone
            pilot.pen_down
        </group>
    </context>
    <action trigger='enter' time='3'>
        pilot.notepad.confidential
    </action>
</context_interaction>
```

The aQtive space framework itself was built directly upon earlier formal modelling concepts of *status–event analysis* (Dix and Abowd 1996). While many intentional user interactions are *event* based (e.g. pressing a key) as are low-level computer implementations, sensor data are more often a *status*, that is there is always an underlying value even if it is only sampled periodically. Work following on from this has included an XML component-based notation (Dix et al. 2007).

Schmidt's work on implicit interaction also included an XML-based specification notation (Fig. 7.5), and other work in the area has included discrete event modelling (Hinze et al. 2006) and context-aware architecture derived from MVC (Rehman et al. 2007).

There have been a number of projects which have created combinations of models, tools and evaluation methods for pervasive systems or smart environments.

In addition to the Kinetic User Interface approach, the uMove project led to the creation of a framework including conceptual modelling, architectural design and implementation tools as well as an evaluation method, IWaT (Interactive Walk-Through) (Bruegger et al. 2010; Bruegger 2011).

Wurdel addressed similar broad goals, but focused more on task modelling using a notation CTML (Wurdel 2011). CTML is an extension to CTT (Paterno 2012) with additional primitives to deal with both actions that are not explicitly addressed to the system (but may be sensed by it), and autonomous system behaviours including those that directly or indirectly affect the user. The resulting task models are used partly to drive model-based system development and partly as input to hidden Markov models.

Another very similar project by Tang et al. (2014) creates a variety of OWL-based notations and integrated design and development tools for pervasive applications including task specification and service design.

7.4 Modelling Low-Intention Interactions

In order to explore ways to formally model these behaviours, we will use a rational reconstruction of the design of car courtesy lights (originally developed in Dix et al. 2004; Dix 2006). These usually include an explicit off/on switch, but also turn on automatically at times.

7.4.1 Modelling Process

We will not create a new notation, but instead augment standard task and state description formalisms. The task descriptions here will be simple scenarios augmented by state-space models, but richer descriptions could be used such as HTA (Shepherd 1989), CTT (Paterno 2012) or CTML (Wurdel 2011).

The critical steps are to:

(i) annotate the supported task to see where augmentation or assistance would be useful. In the examples, this is in the form of + or – to indicate how desirable or undesirable a particular assistance would be.

(ii) annotate the sensed task to see where existing or new sensors could give useful information and to assess the likelihood that sensed state matches the target context in the real world.

Often sensors are attached to objects in the environment; so for (ii), it can be useful to model explicitly the states of physical (or virtual) objects.

Note that the assessment of desirability in (i) is a core design decision. It may be a matter of discussion and it may include more or less desired states as well as hard and fast ones. This is particularly important as sensed input in (ii) is rarely precise. This imprecision may be because of limitations in sensors, because sensed user actions may be typical but not guaranteed, or because the target world state may not be directly able to be monitored, meaning some proxy has to be sensed instead. By having an explicit notion of both desirability and precision, we can ensure that sensors are chosen to ensure the best behaviour in critical situations, with leeway in the less critical ones (Fig. 7.6). For example, we may be able to adjust threshold values to ensure correct behaviour in the critical areas (ticks and crosses) but simply achieve ‘good enough’ results in the less critical areas (grey ‘leeway’).

As we work through the examples adding annotations to the task models, we will develop a form of secondary notation (Green and Petre 1996). In particular, when looking at the supported task, we will use + and – to denote the desirability or otherwise of a particular intervention. However, these are not intended to be a closed set of annotations; indeed, the ‘bomb’ annotation in Fig. 7.7 was added when the example was being used in class and the need arose to highlight particularly problematic situations.

Fig. 7.6 Matching desirability of intervention in the supported task (support, augmentation, autonomous action) with reliability of sensing in the sensed task

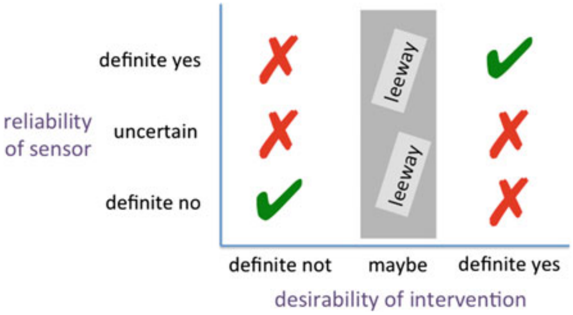


Fig. 7.7 Car courtesy light—getting into the car (from Dix et al. 2004) + signifies would like light on, – signifies would like light off, 'bomb' signifies that safety is a major issue

1.	deactivate alarm	0		
2.	walk up to car	++/– –	💣	is this safe?
3.	key in door	–		
4.	open door & take key	+		
5.	get in	++		
6.	close door	0		
7.	adjust seat	+		
8.	find road map	++		
9.	look up route	+++		
10.	find right key	+		
11.	key in ignition	–		
12.	start car	0		
13.	seat belt light flashes	0		
14.	fasten seat belt	+		
15.	drive off	– –	💣	safe? legal?

7.4.2 Car Courtesy Lights

In the case of the car courtesy light, the sensed task and the supported task are identical. Figure 7.7 shows the task of getting into the car (there would be other tasks such as stopping and getting out, sitting in the car to have a picnic). The main steps are listed and against each one whether or not the courtesy light is wanted.

The pluses mean it would seem good to have it on, the more pluses, the more beneficial. The minus signs show where it would be best to have it off, and the bomb situations where there could be safety issues. Step 15 is obviously problematic and it may be distracting if the courtesy light stays on while driving, and in some places may even be illegal. Step 2 is also marked as potentially unsafe as it might allow a mugger to see which car you are going to, although could also be useful in helping you find your car. Cars vary on whether they choose to turn lights on in these circumstances.

In addition, we ought to look at the scenarios such as when the driver is leaving the car, picking up or dropping off passengers. However, on the whole, the lights want to be (a) on when you are sitting in the car, (b) not left on for long periods

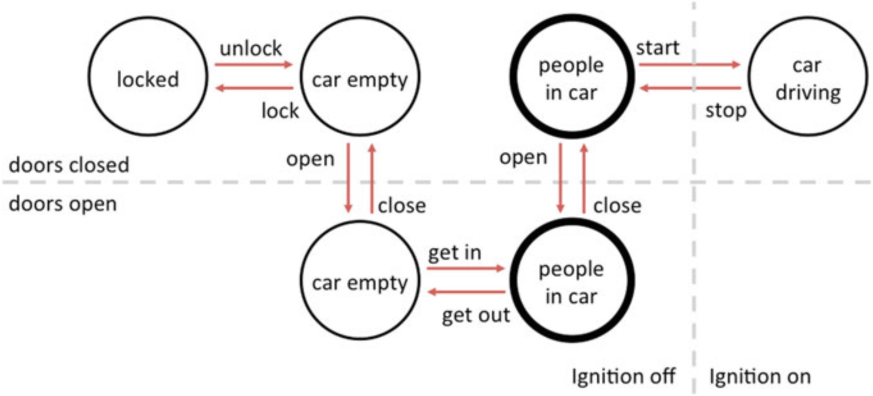


Fig. 7.8 States of the physical car

when you are not in the car (so as not to drain the battery), but (c) not on when you are actually driving (for safety). Of these, (c) is an absolute requirement, (a) is a 'good to have' (you can always turn them on or off explicitly) and (b) is a strong requirement, although the precise time can be quite relaxed.

One could add sensors to the car, for example a PIR (passive infrared) sensor in the car to detect movement so that you can tell if it is occupied. However, cars already have a range of sensors that are reused by the manufacturers to produce automatic lighting. Notably, there is typically a door sensor that triggers a warning if the car is driven without closing the doors properly, and also it is possible to detect the power when the ignition is turned on.

Figure 7.8 shows the main states of the car in terms of people's presence (omitting unusual cases such as people locking themselves inside). These are divided into the states that can be distinguished by the two available sensors.

Note that it is the two states in bold, when people are in the car, which are where we would like the light to be on.

It is possible to tell with certainty when the car is being driven as the ignition is on, so this is a reliable sensor to use to ensure that the lights are not automatically turned on. Most cars gradually dim the lights as soon as the ignition is turned on.

However, it is impossible to tell from these sensors the difference between opening the car door and then realising you have forgotten something and going back into the house to get it, unless you lock the car. Most cars simply use a timer for this, turning the light on when the car doors are first opened, and then off after a period. This is based on the assumption that the sojourn in the 'people in car' state before transitioning to 'car driving' is brief. The timer means that the light is likely to be on most of the time during this state (fulfilling the soft requirement), while also ensuring the light is not on for too long in the 'car empty' state (hard but variable time requirement).

It is clear that having access to the car lock state or a PIR would be useful, as the latter is often fitted as part of the car alarm system. Presumably, the alarm subsystem in most cars is relatively isolated from the rest of the car subsystems for security reasons and may often be fitted by third parties, so cannot be accessed by the courtesy light subsystem. However, it would not be hard to imagine a future in-car IoT framework to allow more creative use of sensors.

7.5 Into Practice: The Internet-Enabled Shop Open Sign

The interaction styles and techniques in this paper were put into practical use in the design of a community public display system on the Isle of Tiree. This was produced largely as part of the biannual Tiree Tech Wave series of technology/maker meetings (tireetechwave.org). The system is a long-term 24/7 deployment, and the overall data and public display architecture are described in detail in Chap. 4.

In this section, we will focus on the design of two particular elements of this, the Internet-enabled open sign and LED ticker-tape display, which provide sensor input to the data infrastructure. We start with an initial ‘provotype’-style (Boer and Donovan 2012) design concept (the fish and chip van that tweets) and then move on to the Internet-enabled open sign, which has been in operation now for several years.

7.5.1 Concept—The Chip Van That Tweets

At the first Tiree Tech Wave, several themes came together around the creation of a concept mini-project, the ‘Chip Van That Tweets’ (Dostal and Dix 2011).

There was a fish and chip van positioned quite close to the location of the event. Many island businesses are run by a single individual and so can be fragile: if the individual or one of their family is ill, or there is some other kind of emergency, the shop or van may be late opening or close early.

On the mainland, if you went to the fish and chip shop, but found it unexpectedly shut, there would be another close by. On the island, you may have driven perhaps up to ten miles (15 km), over bumpy roads, and there is nowhere else to go. Occasionally, people would ring friends who were within sight of the chip van to ask whether it was open before setting off.

As a light-hearted exemplar, the participants created a prototype to address the issue.

A near full-size mock-up of the van front was constructed out of cardboard. On the flap at the front of the van, an Arduino was positioned with a tilt switch. When the van flap was opened, the tilt switch was activated and the Arduino connected to a mobile phone and sent a tweet via SMS ‘#tireechipvanopen’. When the van flap closed, the Arduino sent the tweet ‘#tireechipvanclosed’.

Software that could run on an islander’s computer at home listened using the Twitter API and when it saw ‘#tireechipvanopen’ or ‘#tireechipvanclosed’, it sent a

Fig. 7.9 Tiree chip van—sensed task (owner)

0. Serving chips

1. drive to van
2. enter into van (opens side door)
3. prepare for evening:
   turns on power, lights, deep fat fryer
4. open serving flap (at opening time)
   ** sensed by system – tweets #tireechipvanopen
5. serving customers
   take order, fry food, wrap cooked food, take money
6. close serving flap (at closing time)
   ** sensed by system – tweets #tireechipvanclosed
7. tidy up
8. leave van (close side door)
9. go home

Fig. 7.10 Tiree chip van—supported task (customer)

0. Buy and eat chips

1. decide would like fish and chips
2. check if open
   2.1 check current time and opening hours (old)
   2.2 look at chip van model on mantlepiece (new)
3. drive to chip van
4. buy fish and chips (if open)
   or
5. disappointed and hungry (if closed)
6. drive home

message to another Arduino, which was connected to a model fish and chip van, maybe sitting on the potential customer's mantlepiece. A small motor then opened or closed the model to match the real-world fish and chip van.

This is a form of incidental interaction, and Figs. 7.9 and 7.10 show the sensed and supported tasks. While the chip van owner is opening and serving the focus is always on preparing and opening the van, incidentally this leads to improving the customer's interaction efficiency (at 2.2) and experience (4 rather than 5!).

7.5.2 TireeOpen—The Internet-Enabled Open Sign

The chip van that tweets was created as a concept project. It exemplified many important and very practical issues of island life and also was interesting technologically, using Twitter as middleware and an 'Internet of things' architecture. However, it was playful and probably slightly over the top in terms of technology.

Although the technology for the prototype was largely working, there was a gap as the correct kind of phone was not available, so the step between the Arduino detecting the open serving flap and the tweet SMS being sent was emulated by the Arduino flashing an LED and the tweet being sent by hand.

In one way, it would have been a small step to deploy this as a form of technology probe (Hutchinson et al. 2003), but the patchy mobile signal on the island would probably render even the SMS tweeting unreliable.

This last point is not inconsiderable for the practical application of Internet of Things—it only works if you have the Internet, or at least some data communications.

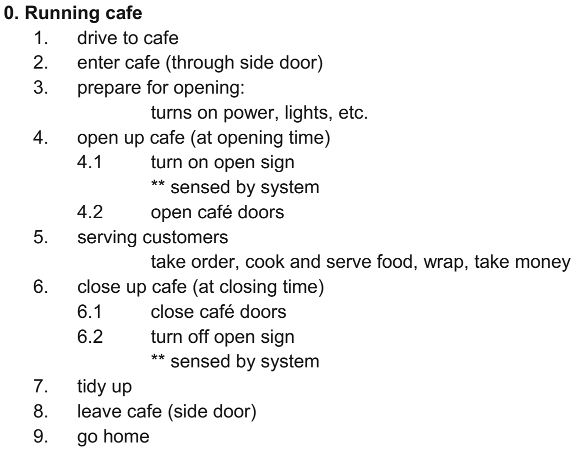
The Cobbled Cow café is based at the Rural Centre on Tiree, the venue of the Tiree Tech Wave. The Cobbled Cow has some of the same issues as the chip van (small family business, illness or other unexpected events can lead to late opening, etc.); however, unlike the chip van, Wi-fi is available.

The closest equivalent to the chip-van serving flap is unlocking the café door, but it is harder to add a sensor to a door lock, than to a flap opening, and it would mean modifying the physical fabric of the café. However, there is also an LED 'open' sign in the window, which is switched on as part of the opening up process (see Fig. 7.11). This already has low-voltage power, hence easy and safe to modify. Modifying the open sign also has the advantage that it could easily be replicated for different kinds of businesses.

The supported task is the customer experience deciding whether or not to visit the café (Fig. 7.12). It is pretty much identical, equivalent to the chip-van customer task (Fig. 7.10).

Rory Gianni created the Internet-enabled open sign (probably the world's first), using an Electric Imp (see Fig. 7.13). The Electric Imp contains a Wi-fi and cloud enabled processor in a package rather like a large SD card. This has been specifically designed for Internet of Things applications and can be connected to various sensors and actuators. It is programmed and runs code through Electric Imp's cloud platform. For this purpose, no explicit sensor was needed as the power for the device was simply connected to the open sign's low-voltage power input, meaning it is only on when the open sign is on. When it powers up, it sends a periodic message back to the cloud platform, which in turn calls a small script on the Tiree Tech Wave site where a

Fig. 7.11 Cobbled Cow café—sensed task (owner)



0. Running cafe

1. drive to cafe
2. enter cafe (through side door)
3. prepare for opening:
   turns on power, lights, etc.
4. open up cafe (at opening time)
   4.1 turn on open sign
   ** sensed by system
   4.2 open café doors
5. serving customers
   take order, cook and serve food, wrap, take money
6. close up cafe (at closing time)
   6.1 close café doors
   6.2 turn off open sign
   ** sensed by system
7. tidy up
8. leave cafe (side door)
9. go home

```
0. Eat out at café
    1.   decide would like food at café
    2.   check if open
         2.1   check current time and opening hours (old)
         2.2   look at open sign on web page (new)
    3.   drive to café
    4.   buy and eat food (if open)
         or
    5.   disappointed and hungry (if closed)
    6.   drive home
```

Fig. 7.12 Cobbled Cow café—supported task (customer)

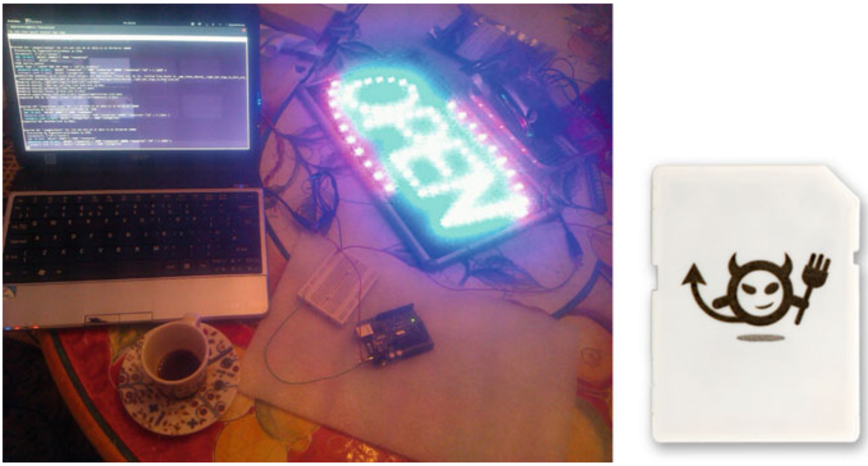


Fig. 7.13 *Left* Internet-enabled open sign under development (photograph Rory Gianni). *Right* Electric Imp module (photograph www.electricimp.com media resources)

log is stored. When the Web-based status page is requested, this checks the logs and can display a Web open sign, which reflects the state of the physical one.

The Internet-enabled open sign was deployed at the Cobbled Cow and has been running now for more than 2 years.

In many ways, the system at the Cobbled Cow resembles the chip van prototype, with a few differences. Technically, rather than Twitter as middleware, the Electric Imp cloud service is being used, and rather than a model chip van on the mantlepiece, a simple Web sign is used for displaying the status. More critically, there are differences in the interaction probabilities.

In the case of the chip van, it is impossible to serve customers without opening the flap. That is for the chip van sensed task (Fig. 7.9), sub-task 4 (open flap) will always happen before sub-task 5 (serve customers). In contrast, for the Cobbled Cow sensed task (Fig. 7.11), sub-task 4.1 (turn on open sign) is a matter of routine, but it is physically possible to open the café (sub-task 5) without turning on the sign; that is sub-task 4.1 is likely but not certain to occur. Similarly, it is possible to close the shop without turning off the sign (sub-task 6.2), although this is likely to be noticed at sub-task 8 (leave shop) as the glow of the sign would be visible except in high summer.

That is the uncertain nature of tasks being sensed means that there is a possibility that the Web sign might be off in the daytime when the café is actually open, or on at night when it is actually closed. The latter is fairly obvious to the viewer, but the former could mean they do not go to the café when it is in fact open. Arguably, this is not as bad as a customer going when it is actually closed, but still a problem.

Figure 7.14 shows these issues using a diagram of the main states of the café, similar to that for the car courtesy light in Fig. 7.8. In Fig. 7.14, the states where we would like the Internet version of the sign to show 'open' (when the shop is open) are shown with thick edges. The states shown dashed are those where the sign is 'wrong' (on when the shop is closed or vice versa). The transitions labelled with roman numerals (i), (ii), (iii), (iv), (v), and (vi) are the normative path corresponding,

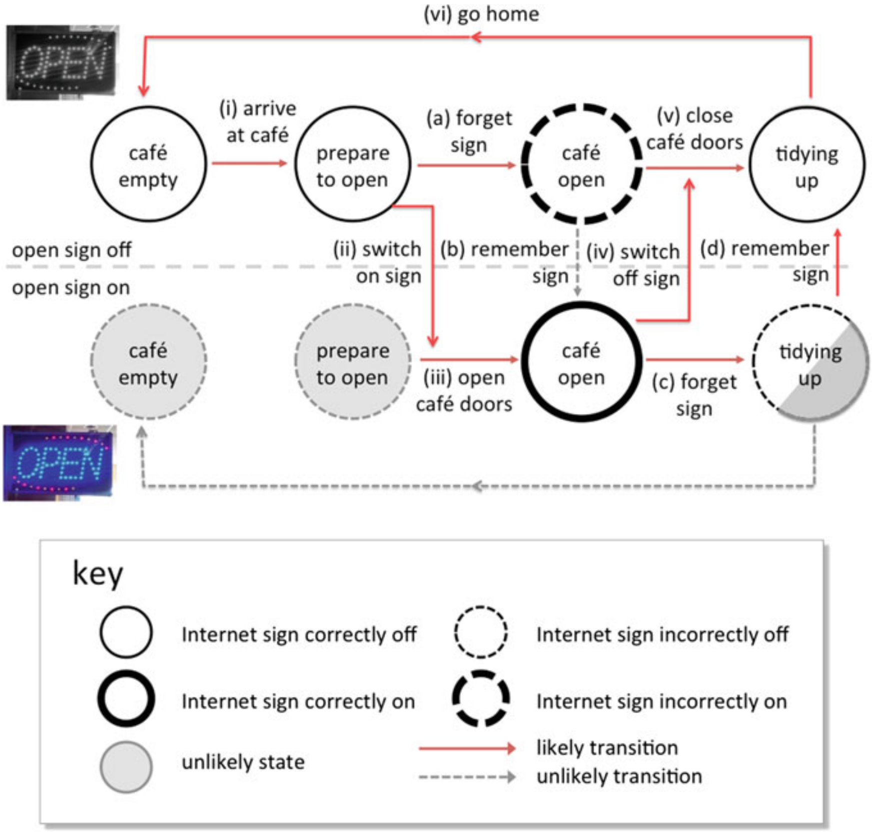


Fig. 7.14 States of the café

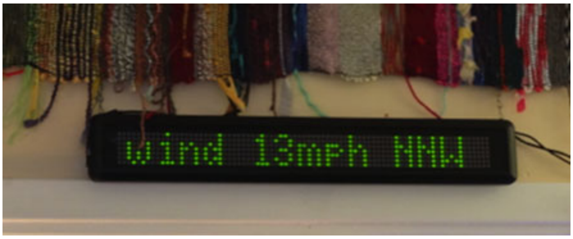


Fig. 7.15 Public LED 'ticker-tape' display showing local weather, news, etc

respectively, to steps in Fig. 7.11: 2, 4.1, 4.2, 6.2, 6.1 and 8. So long as the café owners follow these transitions, the sign will always be correct.

However, some transitions lead from good states to bad states: (a) forgetting to turn on the sign when the café is opened; and (c) forgetting to turn it off when closing. In each case, there is a possible correction: (b) noticing the sign is off when the café is open, and (d) noticing it is on when the café is being tidied up at the end of the day. Transition (b) is marked dashed and in grey to signify that it is less likely to occur, because in the daytime it is not easy to see that the sign is not on from inside the café. In contrast, transition (d) is solid to signify it is more likely to occur, as the glow of the sign is pretty obvious as you turn out lights, although sometimes this could be well after the shop is actually closed. Because of the latter, the transition denoting leaving with the sign on is greyed out to signify it is unlikely. Based on the likelihoods of these transitions, some states have been marked in grey to signify that they are unlikely to occur.

As can be seen, the only state that is both wrong and likely to occur is the sign being off when the café is open. This could be addressed by finding some means to make forgetting to turn on the sign (a) less likely. Alternatively, one could try to make it easier to notice when the sign is not on. As an attempt to partly address this, the LED 'ticker-tape' information display (Fig. 7.15) was wired so that both it and the open sign are turned on together. While the open sign faces outwards through the window, the ticker-tape faces inwards into the café. This can still be forgotten, but it is more likely that it will be noticed, that is making transition (b) more likely to occur at all, and transition (d) likely to occur more promptly after the shop has closed.

7.6 Further Design Considerations for Low Intention

The techniques above have been focused on the 'functional' design of low-intention systems, choosing appropriate sensors from the sensed task to allow suitable interventions to the supported task. However, these are far from the only important design issues. We will look at two of these: user models and privacy.

7.6.1 User Models

This chapter is principally concerned with designer models for low-intention interaction, but of course, users also have (or maybe fail to have) models of these systems. This is well explored for more traditional systems with considerable work on metaphors, and design guidelines related to aspects such as visibility, consistency and feedback, which help the user to understand the system states, the available actions and the effects of these on the system.

Considering context-aware systems, Schmidt (2013) suggests that the user interface should seek to minimise 'awareness mismatch', making it clear to users what sensory information is being used, so that the user can make satisfactory explanations of why system effects occurred and valid inferences about how to create desired outcomes.

For some of the best low-intention and low-attention interfaces, none of this is necessary. If users are unaware that modifications are happening, and any alterations in interaction are sufficiently good, then they do not need to build any sort of model. For example, many users are unaware that search results are tuned to their past click-through behaviour.

However, if the user does become aware, for example that the heating or lighting levels spontaneously change, then the interaction may become 'spooky', as if there are ghosts in the walls changing the very environment. Intentional interactions may sometimes be indirect (e.g. waving your arm to control the character on a video game), but are extensions of the physical actions of day-to-day life. Autonomous behaviour, however, often suggests animate beings at work. This is rather like the 'uncanny valley' (Mori 1970) for human-like robots, the more intelligent and human-like autonomous action becomes, the more like magic, or the supernatural it becomes.

This suggests that, on a moment-to-moment basis, low-intention systems need not follow Schmidt's (2013) 'awareness mismatch' advice. However, some sort of model should be available when users want it. The latter also helps address some of the ethical issues of making modifications that are not apparent to users. In Europe, this issue is likely to become increasingly important following the General Data Protection Regulation of the Council of the European Union (2016), which bans or severely regulates black box automatic decision-making in any legally sensitive area (Goodman and Flaxman 2016).

7.6.2 Privacy

Sensor-rich environments also raise privacy and security issues. The definition of incidental interaction says sensed data is collected '*in order to influence/improve/facilitate the actors' future interaction or day-to-day life*' (Dix 2002)—that is the individual or group being sensed also personally benefits from that sensing. This was phrased deliberately to exclude surveillance where data are collected purely for the benefit of others.

Of course, even where benefits do accrue to the person being sensed, this does not mean that their data may not be deliberately or accidentally misused. The usage data that sites such as Google or Twitter collect are used to enhance your own experience, but also to feed marketing databases. Cloud-based systems can serve to make this worse, and there have been a number of recent news articles worrying about smart TVs and voice interaction dolls, which send anything spoken in their vicinity to central servers to be processed and often stored.

There have been attempts to use similar technology to reverse the balance. Steve Mann and colleagues deliberately use personal wearable technology to record as a visitor in environments, such as shops, where CCTV and other means are usually used to record those coming in, a practice they term '*sousveillance*' (Mann et al. 2003).

Of course, this itself may invade the privacy of others and create a new disparity, potentially leading to antagonism against 'cyborgs', perhaps most notably when Mann's 'EyeTap' digital glasses were removed in a Paris McDonald's restaurant (Biggs 2012; Popper 2012). As this kind of technology becomes commoditised, for example with Google Glass or even ubiquitous mobile phone video, both kinds of privacy issue converge.

7.6.3 Can Task Models Help?

None of the above issues are explicitly dealt with by the task modelling suggested in this chapter, but the two-task view does help to elucidate some of the issues and the way they relate to one another.

The user modelling issues are primarily related to the supported task. Clearly identifying the points at which interventions happen in this can at least help the designer to assess the potential for 'spookiness' and also make it easier to create explanation systems. The uncertainty related to sensors can of course make this worse as system actions are less predictable, for example if your phone battery dies and so the heating system adjusts the house based on the wrong inferred prior activity.

In contrast, privacy issues are primarily related to the sensed task. While such issues are not 'solved' by task modelling, having a clear specification of where and when data are gathered makes it easier both to avoid unintended disclosure and to explain to users what is being gathered and why, thus ensuring informed consent.

Both issues are made more problematic by complex inference algorithms used as either part of sensor fusion or context-sensitive interactions.

Privacy frameworks in the ubicomp literature focus on *restricting* information flows, in the belief that less information means more privacy. However, the earliest work on privacy in HCI showed that on occasions, less information, even highly anonymised information, could be more personally sensitive than full information (Dix 1990); indeed, half-stories are the grist of gossip and the downfall of many a politician.

Just as problematic are actions based on black box algorithms. Again, the author's early work on the HCI implications of pattern matching technology highlighted the danger that algorithms could unintentionally produce sexist or racist outputs (Dix 1992). This issue has taken some time to become apparent, but has now come to public attention with high-profile reporting of apparently racist search results (Gibbs 2015) and photo tagging (BBC 2015; Hern 2015). More mundane examples such as intelligent heating might seem immune from such issues, except that it has been recently argued that office workplace temperature standards are effectively sexist, based on typical male metabolism and hence disadvantaging women (Kingma and van Marken Lichtenbelt 2015). Because the standards are public, this can be exposed, challenged and debated. Quite possibly, a more intelligent office air-conditioning system might avoid this problem, basing temperature on actual occupancy, but equally it could make things worse by introducing new implicit and potentially illegal bias. Most worrying, it would be far harder to tell whether it was discriminatory or, even if it is not, defend against the accusation that it is.

In both cases, provenance of data and perspicuity of algorithms are at the heart of helping to ensure that users of these systems can make sense of the outcomes. The two-task analysis helps in the former, because it factors the gathering of data from the application of that data; but it does not solve the problem of inscrutable algorithms.

7.7 Discussion

We have seen how it is possible to analyse the twin tasks for low-intention interaction to identify the potential, albeit uncertain, sensor data from the sensed task, in order to modify system behaviour for the supported task. In the first example, the car courtesy lights, the two tasks were effectively the same activity, whereas in the second example, the Internet-enabled open sign, the sensed and supported tasks were different.

Critical in both examples has been the assessment of the likelihood that certain sensor measurements will indicate particular user behaviour. In some cases, this can be clear-cut: if the car has started and is moving, then it is occupied. In others, it is not definitive, merely indicative: in the car example, opening the doors may mean a person is getting in the car; when the sign is powered on, it may indicate that the shop is open; but in both cases, there are circumstances when the sensor and user behaviour may not match.

Note also that in some cases there are physical processes at work that constrain user behaviour (e.g. you cannot get into a car normally without opening a door), but in others, we rely on habit, routine or 'typical' user behaviour (e.g. the café owner's opening up rituals).

To some extent, the methods and techniques described in this chapter have become 'second nature' to the author, and so during the design process for TireeOpen, the formal models were mostly 'in the head' rather than on paper; however, the steps described in Sect. 6 accurately reflect the processes and analyses used.

This chapter has not presented a specific notation, but instead has shown how standard notations for describing tasks, scenarios and physical models can be annotated and analysed in order to aid the design of low-intention systems. Also, it has not attempted to connect this design modelling to the more implementation-oriented context-modelling notations. Doing this could enable a level of verification of the implemented system with respect to the design intentions.

As noted, low-intention systems are already ubiquitous in the digital domain, and we encounter mundane examples in day-to-day life. However, as the Internet of Things and ubiquitous computing move from vision to reality, we will live in an increasingly digitally augmented physical environment. The techniques in this chapter are one step in addressing some of the issues that arise as we seek to design systems for this emerging world.

Acknowledgements Many thanks to Rory Gianni who created the Internet-enabled open sign.

References

Alexander A, Ishikawa S, Silverstein M Ingrid K, Angel S, Jacobsen M (1977) A pattern language. towns, buildings, construction. Oxford University Press. https://archive.org/details/APatternLanguage

BBC (2015) Google apologises for Photos app's racist blunder. BBC News, Technology. http://www.bbc.co.uk/news/technology-33347866. Accessed 1 July 2015

Beigl M, Gellersen H, Schmidt A (2001) MediaCups: experience with design and use of computer-augmented everyday objects. Comput Netw 35(4):401–409

Benford S, Schnadelbach H, Koleva B, Gaver B, Schmidt A, Boucher A, Steed A, Anastasi R, Greenhalgh C, Rodden T, Gellersen H (2005) Expected, sensed, and desired: a framework for designing sensing-based interaction. ACM Trans Comput-Hum Interact (TOCHI) 12(1):3–30

Biggs J (2012) Augmented reality explorer Steve Mann assaulted at Parisian McDonald's. TechCrunch. http://techcrunch.com/2012/07/16/augmented-reality-explorer-steve-mann-assaulted-at-parisian-mcdonalds/. Accessed 16 July 2012

Boer L, Donovan J (2012) Provotypes for participatory innovation. In: Proceedings of the DIS '12. ACM, pp 388–397. doi:10.1145/2317956.2318014

Bruegger P, Lisowska A, Lalanne D, Hirsbrunner B (2010) Enriching the design and prototyping loop: a set of tools to support the creation of activity-based pervasive applications. J Mobile Multim 6(4):339–360

Bruegger P (2011) uMove: a wholistic framework to design and implement ubiquitous computing systems supporting user's activity and situation. PhD thesis, University of Fribourg, Switzerland. http://doc.rero.ch/record/24442

Council of the European Union (2016) Position of the council on general data protection regulation. http://www.europarl.europa.eu/sed/doc/news/document/CONS_CONS(2016)05418(REV1)_EN.docx. Accessed 8 April 2016

Dey A (2001) Understanding and using context. Pers Ubiquit Comput J 5(1):4–7

Dix A (1990) Information processing, context and privacy. In: Diaper G, Cockton G, Shakel B (eds) Human-computer interaction—INTERACT'90 North-Holland. pp 15–20. http://alandix.com/academic/papers/int90/

Dix A (1991) Formal methods for interactive systems. Academic Press, London. ISBN 0-12-218315-0. http://www.hiraeth.com/books/formal/

Dix A (1992) Human issues in the use of pattern recognition techniques. In: Beale R, Finlay J (eds) Neural networks and pattern recognition in human computer interaction, Ellis Horwood, pp 429–451. http://alandix.com/academic/papers/neuro92/neuro92.html

Dix A, Abowd G (1996) Modelling status and event behaviour of interactive systems. Softw Eng J 11(6):334–346

Dix A, Rodden T, Davies N, Trevor J, Friday A, Palfreyman K (2000a) Exploiting space and location as a design framework for interactive mobile systems. ACM Trans Comput-Hum Interact (TOCHI) 7(3):285–321

Dix A, Beale R, Wood A (2000b) Architectures to make simple visualisations using simple systems. In: Proceedings of the working conference on advanced visual interfaces (AVI '00). ACM, New York, USA, pp 51–60. doi:http://dx.doi.org/10.1145/345513.345250

Dix A (2002) Beyond Intention—pushing boundaries with incidental interaction. In: Proceedings of building bridges: interdisciplinary context-sensitive computing. Glasgow University. http://alandix.com/academic/papers/beyond-intention-2002/. Accessed 9 Sept 2002

Dix A, Finlay J, Abowd G, Beale R (2004) Modeling rich interaction. Chapter 18 in Human–computer interaction, 3rd edn. Prentice Hall, Englewood Cliffs. http://www.hcibook.com/e3/

Dix A (2006) Car courtesy lights—designing incidental interaction. Case study in HC Book online! 2004/2006. http://www.hcibook.com/e3/casestudy/car-lights/

Dix A, Leite J, Friday A (2007). XSED—XML-based description of status-event components and systems. In: Proceedings of engineering interactive systems 2007, IFIP WG2.7/13.4 conference, Salamanca, March 22–24, 2007, LNCS, vol 4940, pp 210–226

Dostal J, Dix A (2011) Tiree tech wave. Interfaces, Summer 2011, pp 16–17 http://tireetechwave.org/events/ttw-1/interfaces-article/

Forbrig P, Märtin C, Zaki M (2013) Special challenges for models and patterns in smart environments. In: Kurosu M (ed) Proceedings of the 15th international conference on human-computer interaction: human-centred design approaches, methods, tools, and environments—Volume Part I (HCI'13), vol Part I. Springer, Berlin, pp 340–349

Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns: elements of reusable object-oriented software. Addison-Wesley, Longman

Gellersen H, Beigl M, Krull H (1999) The MediaCup: awareness technology embedded in an everyday object. In: International symposium on handheld and ubiquitous computing (HUC99), Karlsruhe, Germany

Ghazali M, Dix A (2006) Natural inverse: physicality, interaction & meaning. In: Let's get physical: tangible interaction and rapid prototyping in, for, and about design workshop at 2nd international conference on design computing & cognition 2006, TU/e Eindhoven. http://alandix.com/academic/papers/DCC-2006-LGP-natural-inverse/. Accessed 8–12 July 2006

Gibbs S (2015) Google says sorry over racist Google Maps White House search results. The Guardian, 20 May 2015. http://www.theguardian.com/technology/2015/may/20/google-apologises-racist-google-maps-white-house-search-results

Goodman B, Flaxman S (2016) EU regulations on algorithmic decision-making and a "right to explanation". Presented at 2016 ICML workshop on human interpretability in machine learning (WHI 2016), New York, NY. http://arxiv.org/abs/1606.08813v1

Green T, Petre M M (1996) Usability analysis of visual programming environments: a 'cognitive dimensions' framework. J Visual Lang Comput 7(2):131–174. doi:10.1006/jvlc.1996.0009

Greenfield A (2006) Everyware: the dawning age of ubiquitous computing. Peachpit Press, Berkeley

Heidegger M (1927) Sein und Zeit. (English translation: Being and Time. Harper, 2008)

Hern A (2015) Flickr faces complaints over ‘offensive’ auto-tagging for photos. The Guardian. http://www.theguardian.com/technology/2015/may/20/flickr-complaints-offensive-auto-tagging-photos. Accessed 20 May 2015

Hinze A, Malik P, Malik R (2006) Interaction design for a mobile context-aware system using discrete event modelling. In: Estivill-Castro V, Dobbie G (eds) Proceedings of the 29th Australasian computer science conference, vol 48 (ACSC ‘06). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, pp 257–266

Hutchins E, Holland J, Norman D (1986) Direct manipulation interfaces. In: Norman DA, Draper SW (eds) User centered system design. Lawrence Erlbaum Associates, Hillsdale, NJ, pp 87–124

Hutchinson H, Mackay W, Westerlund B, Bederson B, Druin A, Plaisant C, Beaudouin-Lafon M. Conversy S, Evans H, Hansen H, Roussel N, Eiderbäck B (2003) Technology probes: inspiring design for and with families. In: CHI’03: Proceedings of the SIGCHI conference on human factors in computing systems. ACM, pp 17–24

Ju W, Leifer L (2008) The design of implicit interactions: making interactive systems less obnoxious. Des Issues 24(3):72–84. doi:10.1162/desi.2008.24.3.72

Kingma B, van Marken Lichtenbelt W (2015) Energy consumption in buildings and female thermal demand. Nat Clim Change 5:1054–1056. doi:10.1038/nclimate2741

Mann S, Nolan J, Wellman B (2003) Sousveillance: inventing and using wearable computing devices for data collection in surveillance environments. Surv Soc 1(3):331–355. http://www.surveillance-and-society.org/articles1(3)/sousveillance.pdf

Mori M (1970). The uncanny valley. Energy 7(4):33–35. (in Japanese). Translated MacDorman K, Kageki N (2012) IEEE Spectrum. http://spectrum.ieee.org/automaton/robotics/humanoids/the-uncanny-valley. Accessed 12 Jun 2012

Newman W, Eldridge M, Lamming M (1991) Pepys: generating autobiographies by automatic tracking. In: Proceedings of the second European conference on computer supported cooperative work—ECSCW ‘91, 25–27 Sept 1991, Kluwer Academic Publishers, Amsterdam, pp 175–188

Nielsen J (1993) Usability engineering. Academic Press, San Diego

Norman D (1990) The design of everyday things. Doubleday, New York

Norman D (2010) Natural user interfaces are not natural. Interactions 17(3):6–10. doi:10.1145/1744161.1744163

Pallotta V, Bruegger P, Hirsbrunner B (2008) Kinetic user interfaces: physical embodied interaction with mobile ubiquitous computing systems. In: Kouadri-Mostéfaoui S, Maamar M, Giaglis P (eds) Advances in Ubiquitous computing: future paradigms and directions. IGI Global Publishing, pp 201–228. ISBN:978-1-599-04840-6

Paterno F (2012) Model-based design and evaluation of interactive applications. Springer

Popper B (2012) New evidence emerges in alleged assault on cyborg at Paris McDonald’s. The Verge. http://www.theverge.com/2012/7/19/3169889/steve-mann-cyborg-assault-mcdonalds-eyetap-paris. Accessed 19 July 2012

Rehman K, Stajano F, Coulouris G (2007) An architecture for interactive context-aware applications. IEEE Perv Comput 6(1):73–80. doi:http://dx.doi.org/10.1109/MPRV.2007.5

Salber D, Dey A, Abowd G (1999) The context toolkit: aiding the development of context-enabled applications. In: Proceedings of the 1999 conference on human factors in computing systems (CHI ‘99), Pittsburgh, PA, 15–20 May 1999, pp 434–441

Schmidt A (2013) Context-aware computing. The encyclopedia of human–computer interaction, 2nd edn. In: Soegaard M, Friis R (eds) Interaction design foundation. https://www.interaction-design.org/literature/book/the-encyclopedia-of-human-computer-interaction-2nd-ed/context-aware-computing-context-awareness-context-aware-user-interfaces-and-implicit-interaction

Schmidt A (2000) Implicit human computer interaction through context. Pers Technol 4(2–3):191–199. doi:10.1007/BF01324126

Schilit B, Adams N, Want R (1994) Context-aware computing applications. In: Proceedings of the 1994 first workshop on mobile computing systems and applications (WMCSA ‘94). IEEE

Computer Society, Washington, DC, USA, pp 85–90. doi:http://dx.doi.org/10.1109/WMCSA.1994.16

Shepherd A (1989) Analysis and training in information technology tasks. In: Diaper D (ed) Task analysis for human-computer interaction, Chapter 1. Ellis Horwood, Chichester, pp 15–55

Shneiderman B (1982) The future of interactive systems and the emergence of direct manipulation. Behav Inf Technol 1(3):237–256

Shneiderman B (1998) Designing the user interface—Strategies for effective human-computer interaction, 3rd edn. Addison Wesley

Tang L, Yu Z, Wang H, Zhou X, Duan Z (2014) Methodology and tools for pervasive application development. Int J Distrib Sens Netw 10(4). http://dx.doi.org/10.1155/2014/516432

Want R, Schilit B, Adams N, Gold R, Petersen K, Goldberg D, Ellis J, Weiser M (1995) An overview of the PARCTAB ubiquitous computing experiment. IEEE Pers Commun 2(6):28–43. doi:10.1109/98.475986

Wigdor D, Wixon D (2011) Brave NUI world: designing natural user interfaces for touch and gesture. Morgan Kaufmann

Wilde A, Pascal B, Hirsbrunner B (2010) An overview of human-computer interaction patterns in pervasive systems. In: International conference on user science and engineering 2010 (i-USEr 2010), Sha-Allam, Malaysia, 13–15 Dec 2010

Wurdel M (2011) An integrated formal task specification method for smart environments. University of Rostock. ISBN:978-3-8325-2948-2

Chapter 8
Modelling the User

Paul Curzon and Rimvydas Rukšėnas

Abstract We overview our research on the formal modelling of user behaviour, *generic user modelling*, as a form of usability evaluation looking for design flaws that lead to systematic human error. This involves formalising principles of cognitively plausible behaviour. We combine a user model with a device model so that the actions of the user model are the inputs of the device, and the outputs of the device are linked to the perceptions of the user model. In doing so, we gain a model of the system as a whole. Rather than modelling erroneous behaviour directly, it emerges from the interactions of the principles and the device model. The system can then be verified against properties such as task completion. This approach can be combined with other analysis methods, such as timing analysis, in a complementary way. It can also be used to make the cognitive assumptions specific that have been made for a design and explore the consequences of different assumptions. It can similarly support human-computer interaction experimental work, giving a way to explore assumptions made in an experiment. In addition, the same approach can be used to verify human-centred security properties.

8.1 Introduction

Traditional formal methods for verifying computer systems often ignore users. People make mistakes, however. Good design processes take this into account as do usability evaluation methods. Formal methods ought to support such processes. There are many ways to take a human perspective into account in the formal modelling of computer systems. For example, the focus can be on modelling the interface and exploring its properties. In other approaches, the focus is on modelling the tasks and processes the user of a system is expected to follow. This can be combined with approaches where specific errors, such as omitting actions, are injected into task

P. Curzon (✉) · R. Rukšėnas
EECS, Queen Mary University of London, Mile End Road, London, UK
e-mail: p.curzon@qmul.ac.uk

R. Rukšėnas
e-mail: r.ruksenas@qmul.ac.uk

B. Weyers et al. (eds.), *The Handbook of Formal Methods in Human-Computer Interaction*, Human-Computer Interaction Series,
DOI 10.1007/978-3-319-51838-1_8

sequences to explore the consequences. In these approaches the human operators are absent from the formal models, however. Some approaches do involve a model of the user, but assume the user is capable of fault-free performance and are essentially task models. Modelling human behaviour within a system explicitly gives an alternative approach where erroneous behaviour emerges from general principles about behaviour.

Furthermore, as the concern of interaction design, and so of modelling and verification, moves out beyond that of a single interface to multiple interfaces with multiple users, or further to socio-technical and ubiquitous systems, it becomes natural to think of modelling each component separately, to explore the, possibly emergent, behaviour of the system as a whole. It then becomes natural to think of modelling the people as components of that system.

In this chapter, we explore the idea of modelling the human and their actions more explicitly, and in particular exploring the fallibility of that behaviour. Rather than modelling erroneous behaviour, we model cognitively plausible behaviour. It is neither right nor wrong behaviour in itself. Human error emerges from acting in this way, in situations where it turns out to be inappropriate. We overview the different strands of our work in developing generic user models based on cognitively plausible behaviour (Butterworth et al. 2000). These models are specialised for any specific situation by providing information about the task and context. The work has shown how generic user models can be used to detect usability problems in interactive systems, as well as used in other ways such as to verify formal statements of requirements or design rules, and combined with other analysis techniques such as timing analysis.

8.1.1 Between Demonic and Angelic Behaviour

While the behaviour of human elements is not explicit in most verification approaches to interactive systems, implicit assumptions about what behaviour is possible are still being made. One can think of this as a continuum (see Fig. 8.1). At one end of the continuum, the user of the system is assumed to potentially do anything. No restrictions at all are placed on their behaviour in the analysis of the system. This is a sensible position to take when safety properties are under consideration. For example, in a nuclear power plant controller, it is important to be sure that the interface does not allow for user error of any kind, or indeed malicious behaviour, to lead to a core meltdown. Similarly, in a security analysis of a system, where the interest is in whether a malicious attacker can subvert it, the aim is likely to be that no such subversion is possible. One must assume in these situations that the user is demonic and will exploit any vulnerability that exists. An analysis in this context needs to be exhaustive over all possible interaction traces. In any safety-critical context, it is important to ascertain that whatever a user does, for whatever reason, their actions cannot lead to identified hazards or unsafe states.

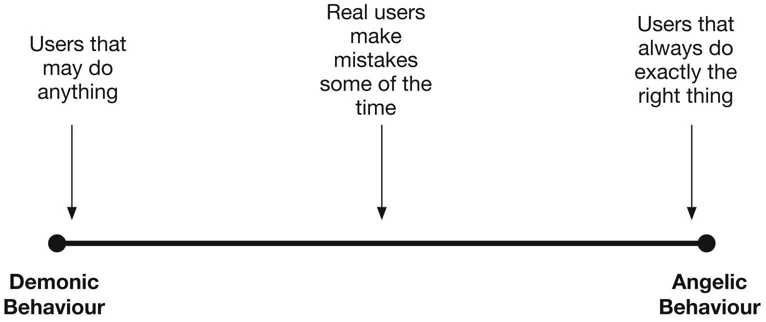

Users that may do anything
Real users make mistakes some of the time
Users that always do exactly the right thing
Demonic Behaviour
Angelic Behaviour

Fig. 8.1 Continuum of user behaviour

At the other end of the spectrum, formal analysis methods may assume that the user does exactly the right thing. This is the stereotypical position being taken when human factors issues are just ignored. The designer of a system devises a 'right' way of interacting with it and assumes that the user will be trained to do exactly that, without error, every time. If they do not, then they are to blame for the consequences: 'The system behaved to specification. Therefore it is not the problem'. This approach, thus, assumes an angelic user who always does the right thing. This kind of analysis also clearly has its place. It is certainly of interest to know, and a bare minimum requirement, that the expected process does always lead to desired outcomes. If it does not, clearly there is a problem. The issue, however, is that the prescribed process may be flawed in other ways too. People do make mistakes, and good design can often prevent it or at least prevent bad consequences resulting.

While the above extreme positions are important forms of analysis, an intermediate position is too. Real people are neither angels nor demons. Their behaviour falls somewhere in between. We have very specific cognitive limitations, and this puts bounds on what it is possible for a human actually to do, even when highly trained. For example, we all have a limited amount of working memory, and when it is overloaded things will be temporarily forgotten. We have one focus of attention, and if it is focussed on one point then we will not see things elsewhere. We do not have access to the designer's conceptual model, only the system image that is presented by the interface, manuals, and so on. This will lead to assumptions being formed about the system that may or may not be correct. So in this sense, it is not only the angelic approach that is unrealistic in general terms, so is the demonic approach. What is appropriate depends, however, on the aim of the analysis.

A question though is where on the spectrum does a 'real' user lie? What behaviour do we model? Humans are not computers. Behaviour varies widely from person to person as well as at different times and in different contexts. We may well be able to behave angelically most of the time, but under conditions of extreme stress or when extremely busy and needing to multitask, our behaviour may be closer to being demonic. This might suggest that modelling users is a lost cause. If anything goes, there is nothing useful that can be modelled. However, our work shows that there is a range of simple but cognitively plausible behaviour that can be modelled in a way that allows a user model-based approach to detect a wide range of usability problems.

In the uses of the above extreme, angelic and demonic, positions, the interest is not on designing a system that is usable. If an analysis aims to support the design of usable systems or of a good user experience, then modelling real users, rather than angelic or demonic ones, is of interest. This matters even in safety-critical contexts. In those contexts, we especially want designs to be as easy to use as possible. We do not want it to be more likely than necessary that mistakes will be made. If we are the one that is about to be infused with a potentially deadly drug, we want the dose set by the nurse to be right today. It is no comfort that the nurse gets it right the vast majority of the time.

In practice, with complex systems, it is often impossible to design them so that nothing bad can possibly happen. The world is too complex, and trade-offs, ethical and practical, may have to be made. So designers of systems that involve humans, who wish to create usable systems, rather than unusable ones, have to take our limitations in to account. User modelling is one way to do this. This matters even for safety-critical systems. Just because you have proved that nothing bad can happen, whatever the user does, does not mean that real users are capable of ensuring that the desired good things do happen.

Another more practical reason why explicitly modelling more realistic user behaviour is of interest is that it gives a way to cut down the search space of behaviour to consider. This may be the difference between an analysis being possible and it being totally impractical. It offers an alternative, or perhaps complementary, way to verify abstractions of the computer system itself. Furthermore, an analysis that throws up large numbers of false positives or negatives with respect to potential usability problems may in actual practice be unhelpful, if the important issues are lost among unimportant ones. User models may provide a way to cut down the number of trivial issues raised, focussing on those that it is most important to fix. Ultimately, usability evaluation is about issues that can be fixed, not in finding as long a list of problems as possible, many of which will never be fixed (Nielsen 2000). Usability evaluation is also as much as anything about persuasion (Furniss et al. 2008). A user modelling approach may ultimately be more persuasive, in that problems detected are potentially linked to an explanation of why they occur in terms of the underlying cognitively plausible behaviour, not just to system traces.

8.2 Verifying Systems with a User Model

8.2.1 Defining Systems Involving User Models

Our approach to modelling users is an extension of hierarchical system verification. It involves the verification of a series of separate subsystems, with each subunit treated as a ‘black box’ represented by its specification. Each black box has inputs and out-

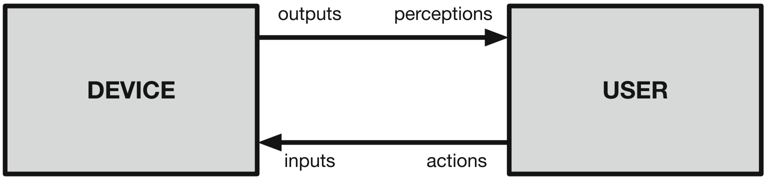


Fig. 8.2 Combining user and Device Model

puts, used to interact with other units. By modelling the way the separate components are connected, we get a description of the implementation of the whole. The outputs of one component are the inputs to others. This combined system can be verified against a specification of the behaviour of the whole.

We can extend this system modelling approach naturally to modelling human-computer systems. In such a system, one or more of the components is a human. We describe their behaviour formally and it just becomes another black box in the system. Now the outputs of the human black box correspond to the actions of the user that can have an effect on other components: pressing buttons, making gestures, issuing commands, etc. Those actions are the inputs to the computer part of the system. The outputs of the computer system, what is on the screen, lights, audible alarms, as well as the actual buttons available, become the perceptions or inputs of the user model (see Fig. 8.2). When the models of the human and computer systems are combined in this way, they give a model of the human-computer system as a whole, with the human component no different to any other.

This can be specified formally, for example, by describing the separate components of the system as relations in higher-order logic. The `USER` relation is defined to give the behaviour of the user with arguments the user's perceptions and actions. The `DEVICE` relation gives that of the device with arguments its inputs and outputs. By equating the actions to the inputs and perceptions to the outputs, we obtain a description of the system (defined as a relation `SYSTEM`)

```
SYSTEM (perceptions, actions, inputs, outputs) =
  USER (perceptions, actions) ∧
  DEVICE (inputs, outputs) ∧
  actions = inputs ∧
  perceptions = outputs
```

The arguments to `SYSTEM` can be represented, for example, as tuples of functions from time to values at that time. We will refer to these as history functions. They represent possible traces of activity over time for a single signal.

For example, one history function within `actions` might represent pushing a physical ENTER button. For each time point, it would specify a boolean to indicate whether it is pushed at that time or not. The definition `USER` specifies the restrictions on what points in time it would be true, i.e. when it would be pushed. It would be equated with the `input` of the device that corresponds to that button being pressed.

The DEVICE specification determines what state changes occur in the device as a consequence of it being pressed and so what outputs result.

The last equalities in our system specification can actually be reasonably seen as part of the user model. They correspond to stages in models of the human action cycle (e.g. Norman (2002)) of taking actions and perceiving results, and could themselves be linked to error. It is easy to press a button but the press not be registered (e.g. on touch screens, this is a common problem). Similarly, just because something is there to be seen does not mean it will be seen immediately, if at all. A slightly more sophisticated user model might, for example, introduce delay between an output and its perception, rather than using equalities. Whatever the relation used to make the link, we now include the machine inputs and outputs as part of the arguments of the user model. The relations linking them become a part of it too. This leads to the system being defined in the form:

```
SYSTEM (perceptions, actions) (inputs, outputs) =
  DEVICE (inputs, outputs) ∧
  USER (perceptions, actions) (inputs, outputs)
```

For example, consider a simple information booth: an *Advice Device*. It has one button marked HAPPINESS. When pressed, Buddhist advice about how to be happy is displayed on the screen: 'Each day, mentally wish things go well for a random person you see' perhaps. The screen is otherwise blank. The device behaviour might be specified formally in a relation which states that the message appears exactly at times t when the button is pressed:

```
ADVICE_DEVICE (happinessButtonPressed,
                happyMessageAppears) =
  ∀t: time.
    happinessButtonPressed t = happyMessageAppears t
```

We need to define a user model relation to work with this device. It should have appropriate values for the perception and action tuples: each a single history function: seeHappinessButton and pressHappinessButton, respectively.

```
ADVICE_DEVICE_USER
      (seeHappyMessage, pressHappinessButton)
      (happinessButtonPressed, happyMessageAppears)
```

Its definition specifies when the button would be pressed by a user. We will explore how that might be defined in subsequent sections. For now, we will assume a trivial version that just equates actions to device inputs and perceptions to device outputs. It states just that the user sees everything that matters and they flawlessly press the button every time they try.

```
ADVICE_DEVICE_USER
      (seeHappyMessage, pressHappinessButton)
      (happinessButtonPressed, happyMessageAppears) =
  ∀t. happyMessageAppears t = seeHappyMessage t ∧
  ∀t. pressHappinessButton t = happinessButtonPressed t
```

Creating a combined system of user and device then just involves putting these together.

```
ADVICE_SYSTEM
      (seeHappyMessage, pressHappinessButton,
       happinessButtonPressed, happyMessageAppears) =
  ADVICE_DEVICE
    (happinessButtonPressed, happyMessageAppears) ∧
  ADVICE_DEVICE_USER
    (seeHappyMessage, pressHappinessButton)
    (happinessButtonPressed, happyMessageAppears)
```

8.2.2 System Verification Involving User Models

Once we have defined a combined system, it can then be verified. In general verification is based around scenarios of particular tasks and whether they would always be successfully completed by users who behave in accordance with the behaviour specified in the user model.

Here, we illustrate the idea in its simplest form, looking at more complex versions in later sections. We first define a usability relation, which we will call TASK_COMPLETION. In a very simple version of our *Advice Device* example, the property might just be that the user gets advice when they press the button.

```
TASK_COMPLETION
      (seeHappyMessage, pressHappinessButton) =
  ∀t: time.
    pressHappinessButton t ⊃ seeHappyMessage t
```

We would then verify the property:

```
∀ seeHappyMessage pressHappinessButton
    happinessButtonPressed happyMessageAppears .
  ADVICE_SYSTEM
    (seeHappyMessage, pressHappinessButton,
     happinessButtonPressed, happyMessageAppears) ⊃
     TASK_COMPLETION
        (seeHappyMessage, pressHappinessButton)
```

This says that the combination of DEVICE and USER restricts the behaviour of the system as a whole in a way that ensures the task is always completed as specified, i.e. the history functions can only hold traces that involve the task being completed.

Of course with our very simple user model described so far, this is a very limited idea of usability.

This kind of usability verification can be done via model checking or theorem proving. The best choice will depend on the specific systems and properties of interest. We have used both, with a variety of technologies including HOL, PVS, SAL and IVY. The style of specification has to vary from that given here depending on the verification technology used of course.

8.2.3 Instantiating a Generic User Model

Just as the device model is created for each verification, a bespoke user model can be too, as we just did for the *Advice Device*. However, that means that the specific behaviours that are possible have to be reinvented each time. Our approach has been to focus on *generic* user models instead. Here, core cognitively plausible behaviour is formalised once and then used with different interactive systems by instantiating the model with just the details specific to that system. This saves work and also means the model needs to be validated just once.

The first step to doing this is to generalise over the user and machine states. It is convenient to use abstract data types to represent these states. They package up all the separate history functions in to a single structure. We use accessor functions in to these states to refer to particular elements.

The generic user model takes this user state (`ustate`) and machine state (`mstate`) as arguments and is defined in terms of them. Critically, each has as its type a type variable (`'u` and `'m`, respectively), meaning that the type is not at this stage specified. It can be later instantiated to any appropriate tuple type, for example, with each element holding a trace. Different devices will have different inputs and outputs. Different tasks to be performed will involve different actions and perceptions. By using type variables, we do not commit to what the actual possibilities are in the generic model.

Following this approach with our user model that just identifies inputs and actions, outputs and possessions, we obtain a user model:

```
USER (actions_inputs, perceptions_outputs)
      (ustate:'u) (mstate:'m) =
  ∀t.
    LINK actions_inputs ustate mstate t ∧
    LINK perceptions_outputs ustate mstate t
```

where `LINK` is given lists of pairs of accessor functions from the two states and equates them at the given time.

The system definition now has the following form, where argument `details` provides information about the actions, perceptions, etc.

```
SYSTEM details ustate mstate =
  DEVICE mstate ∧
  USER details ustate mstate
```

We refer to the process of creating a concrete user model for a particular device and task as *instantiating* the generic user model. To do this for our trivial version to date, we need to give concrete types for the type variables and provide concrete accessor functions that indicate what each element of the type represents. This would be a tuple of traces. Each trace would be of type from time to a value type (such as an integer or boolean type).

For our *Advice Device*, the user state type might be (time → bool × time → bool) to represent the two history functions of seeing and pressing the button, bound into a pair. For brevity, we will refer to the concrete user and machine types, and thus defined as U and M, respectively.

In defining the types, we specify how many things there are to perceive and actions to perform with this system under consideration. The next step of the instantiation is to provide accessor functions into the tuples.

For our *Advice Device*, this means specifying concrete accessor methods for `ustate` and `mstate`. For example, `pressHappinessButton` might be defined to just return the first trace from the user state and `seeHappyMessage` the second.

The concrete user model for the *Advice Device* might then be defined in terms of the generic user model as:

```
ADVICE_DEVICE_USER (ustate: U) (mstate: M) =
  USER([(PressHappinessButton, HappinessButtonPressed)],
       [(SeeHappyMessage, HappyMessageAppears)])
       ustate mstate
```

As we create a more sophisticated generic user model we will need to provide other information too as part of the instantiation.

8.3 A Simple Model of Cognitively Plausible Behaviour

Our work has explored variations on the kind of behaviour that might be treated as cognitively plausible and so limit the range of behaviours assumed. Rather than specifying a user's behaviour from scratch for each new device, we base it on a generic model that embodies such a set of principles. This gives an abstract knowledge-level description (Newell 1990). It is then instantiated with details of a particular task, such as the goals and knowledge the user is assumed to have and that guide their actions internally.

In this section, we discuss a relatively simple form of user model but that puts restrictions on human behaviour. We have shown that even such a simple model can be used to detect a variety of interaction design and usability problems that lead to systematic human error (Curzon and Blandford 2000, 2001). In the next section, we look at more complex forms of plausible behaviour that take into account both internal and external cues that drive actions.

The generic model (in our original HOL version) is essentially just a set of guarded rules of the form:

$$\begin{array}{l} \texttt{guard}_1\ \texttt{t} \land \texttt{NEXT action}_1\ \texttt{actions t} \lor \\ \quad \ldots \\ \texttt{guard}_n\ \texttt{t} \land \texttt{NEXT action}_n\ \texttt{actions t} \end{array}$$

Here, `NEXT` states that, of all the actions available, the next one taken after time t, possibly with some delay, is $\texttt{action}_i$ for each *i*. It only comes in to play if the corresponding guard is true at that time. The guards and actions are determined by the information given to the model when instantiated.

In most of our work, we have specified the models in the various versions of higher-order logic of the different tools we have used. More recent work led by Harrison (Harrison et al. 2016) has used modal action logic. It is a logic that has an explicit notion of actions and includes notions such as ‘permissions’ indicating when actions can occur. It allows state transition models to be expressed in a style similar to that used in industry standard tools such as Simulink statecharts. Details of this approach can be found in Harrison et al. (2016).

In whatever formalism, the core of the user model is based on the set of actions that can be taken to interact with the interactive system. The right-hand side of each guarded rule specifies an action that can be taken, such as pushing a button. The left-hand side is a guard specifying the preconditions for that action to be taken. For example, a button on a touch screen must appear on the screen for it to be pressed. The actions are essentially the outputs of the user model. They link to the inputs of the other components of the system. In a simple system, there would be a single other component: the device interacted with. In general, it could be a range of other devices, people and physical artefacts: all possibilities we have explored (Rukšėnas et al. 2008b, 2013).

8.3.1 Non-determinism

Because the guarded rules in the user model are relational in nature and combined with disjunction, this framework is potentially non-deterministic. If multiple guards are true, then multiple rules are active and corresponding actions could be taken. This reflects the idea that if there are several cognitively plausible behaviours at any time, then any might be chosen. The model thus does not prescribe single behaviour but potentially specifies a range of possible behavioural traces that are plausible. Any may be taken. This may result in correct or erroneous behaviour. Any verification will need to consider all the allowed behaviours.

Depending on how the guards are used, this basic framework could model different kinds of users. If the guards of all actions are set to true, then you have a demonic user, where anything is always possible. If they are set to require specific previous actions to have occurred, the actions must occur in a specified required sequence, or, more generally, exactly one guard corresponding to the ‘correct’ action is guaranteed to be true at all times then you have an essentially angelic one.

The inclusion of non-determinism allows the model to move away from a fully angelic model of behaviour that assumes such a single correct sequence. A user model of this form that specifies alternative expected sequences is essentially a form of task model. If the aim is to explore usability of designs, then this kind of user model already gives scope for some usability issues to be discovered in a design. Consider a verification property of whether a specific user goal is achieved as a result of an interaction. The consequence of non-determinism is that for such a property to be provable, the design cannot rely on any specific action definitely being taken if there are alternatives. The property will only be provable if taking any of the actions specified in the task-based user model at each point leads to the task being completed.

A generic user model lets us go beyond task models, however. Rather than specifying directly the possible behaviour sequences, they emerge from a general specification of plausible behaviour. We explore this idea in the subsequent sections.

8.3.2 Reactive Behaviour

We can start to turn this general framework into a more structured model by providing specific mechanisms for adding action rules. This contrasts with assuming demonically that any action available may be taken, or adding them in an ad hoc way. The simplest way to structure the behaviour of a user is through the process of reacting to sensory prompts. One plausible reason for a person to take an action is that they react to a just-in-time prompt at the point where they are able to take an action. If a light flashes on the start button of an infusion pump, the user might, if the light is noticed, react by pressing the button. If a light flashes on the receipt slot of a cash machine, then that might prompt the person to take the receipt. The generic model can be instantiated by providing a set of prompt–action pairs, that correspond to reactive behaviours. Each pair provided is then turned in to a corresponding guarded rule through a generic rule.

The following generic rule asserts that the signal `prompt` leads to `action` being the next action taken.

```
REACT actions prompt action ustate t =
  (prompt ustate) t ∧ NEXT (action ustate) actions t
```

A collection of such guarded rules for reactive behaviour, combined by disjunction, is defined recursively in terms of `REACT`:

```
(REACTIVE actions [ ] ustate t = FALSE) ∧
(REACTIVE actions (r :: prompt_actions) ustate t =
  ((REACT actions (PromptOf r) (ActionOf r) ustate t))∨
   (REACTIVE actions prompt_actions ustate mstate t))
```

To ensure the user model is fully specified, we add an extra catch-all relation, ABORTION that applies if no other rule fires. It asserts that if none of the other guards is true, so no rule can fire, then the user model takes a special finished action to abort the interaction. As we add new kinds of rules to the user model, this relation is modified to combine their guards too.

If at any time no action at all is cognitively plausible, the user may either give up or take random actions (such as randomly hitting buttons). The finished action is used as an abstraction of any such behaviour. Designs that allow it have usability problems.

We add both REACTIVE and ABORTION relations to the user model:

```
USER (actions, actions_inputs, perceptions_outputs,
      prompt_actions, finished)
      (ustate:'u) (mstate:'m) =
  ∀t.
    REACTIVE actions prompt_actions ustate t ∧
    ABORTION finished actions prompt_actions ustate t ∧
    LINK actions_inputs ustate mstate t ∧
    LINK perceptions_outputs ustate mstate t
```

Having introduced a finished signal, we can make use of it to bind the interaction. We assert in the user model that once finished, the interaction remains finished (at least for the purpose of verification). This overrides the non-deterministic rules, so is placed in an outer if-then-else construct.

```
if (finished (t - 1))
then (finished t)
else non-deterministic rules
```

This leaves the traces of all other actions and perceptions of the model unspecified, so that nothing can be proved about them based on the user model once finished goes true.

To instantiate the model, we need to provide a list of pairs, linking prompts to react to with actions. The prompts can be specified as any relation, though the simplest form would be outputs from a device.

With just this mechanism for including guarded rules, the user model takes actions only when there are specific external prompts to do so. This complements an interaction design approach often used with simple walk-up-and-use machines such as vending machines and cash points, where prompts guide the user through the necessary task sequence.

Let us alter our *Advice Device* so that instead of displaying a message, it releases a capsule containing a printed message, fortune cookie style. The device now has two inputs: one a button to press to get a message, and the other a sensor on the capsule drawer to note the capsule has been taken. The button has a flashing light next to it to show when it should be pressed. The capsule appears in a window with a clunk indicating when it can be taken. Such a device would be specified as a straightforward finite-state machine that specifies restrictions on the history functions. We omit the formal details here as they are straightforward.

To model the user of this version, we just provide as arguments the list of prompt_action pairs, and the finish action, along with the other details.

```
ADVICE_DEVICE_USER (ustate: U) (mstate: M) =
  USER (
    % actions %
    (PressHappinessButton, TakeCapsule),
    % action-input links %
    [(PressHappinessButton, HappinessButtonPressed),
     (TakeCapsule, CapsuleTaken)]
    % perceptions-output links %
    [(HappinessbuttonLight, SeeHappinessbuttonLight),
     (CapsuleReleased, SeeCapsule)]
    % prompt-action pairs %
    [(SeeHappinessbuttonLight, PressHappinessButton),
     (SeeCapsule, TakeCapsule)]
    % finished %
    FinishInteraction)
    ustate mstate
```

This now specifies the reasons for the user of our *Advice Device* taking actions: a behaviour of following prompts. Our verification property might now be that when the user sees the light next to the button, they eventually take a capsule. To prove this, given the way the `finished` signal overrides all else, it will only be guaranteed to happen if the interaction does not abort first:

```
DEVICE_IS_USABLE (HappinessbuttonLight, TakeCapsule)
                  ustate mstate =
  ∀t: time.
    (HappinessbuttonLight mstate) t ⊃
      EVENTUALLY (TakeCapsule ustate) t
```

This form of user model, though simple, would uncover usability problems where a necessary prompt was missing. As we do an exhaustive exploration of the search space, if *any* situation can arise when the device might fail to provide the prompt when needed, then the verification of this property will fail.

Care has to be taken though. The above property is vacuously true if the light next to the happiness button never comes on. The property we verify needs to be extended with a property that it is not achieved vacuously (i.e. that the light is eventually on). That might not be true if the device could be left in some other state by the previous user, for example. The proof might therefore need an assumption added that the light is on at the start of the interaction. One solution to allow that assumption to be discharged would be to add to the design an automated time-out to reset if a capsule was released but not taken.

A situation where a prompt might easily be missed in a design (and often is with real software) is when there is a processing delay. For example, suppose there was a delay between the user of our *Advice Device* pressing the button and the capsule being released. The verification would fail as the user model would have nothing to do. This leads to the need of some kind of 'Please Wait' message added to the device design —a reactive signal with corresponding action to do nothing for a period.

Verification would also uncover some issues where multiple prompts could become active at the same time, suggesting either action was appropriate. If the design under consideration actually required a specific order and ignored actions in the wrong order, this would lead to the user model taking actions that would not result in the achievement of the property being verified. On the other hand, if the device was agnostic about the order, the verification property could still be proved.

8.3.3 Goal-Based Behaviour

A problem with this approach to reactive behaviour is that it does not take into account the goals of the user. People do not completely follow prompts in a Pavlovian manner. They need a reason to do so that ties to the task they are trying to achieve. Actions need to be *relevant*. We can model a simple aspect of this easily. Part of the guard of reactive rules can be that the particular action corresponds to a subgoal needed to achieve the task that is yet to be discharged. One way to do this is to keep a to-do list of outstanding and relevant subgoals. As they are discharged by completing the corresponding action, they are removed from the list for future time instances. The guards now include a check that the goal has not been discharged. This can be done by keeping a history function of actions to be done and removing them from that list once done.

To build this into the generic user model, we add relation, `MANAGE_TODO_LIST`:

```
MANAGE_TODO_LIST actions todolist =
  ∀t. todolist (t+1) = REMOVE actions (todolist t) t
```

This states that at all times the to-do list is the same as at the previous time, except with any actions just taken removed. The to-do list could be added as a new argument to the generic user model needing to be instantiated. Alternatively, we could just use the action list as the initial to-do list. The generic user model can automatically add to the guard of a rule a requirement that the action of the rule is still to be done, i.e. `(ToDo action)` is added.

Suppose our *Advice Device* requires payment: a coin has to be inserted before the happiness button is pressed. In this design, a light next to the coin slot flashes to highlight the coin slot. Now suppose the device design has both the coin slot and happiness buttons flashing at the same time—the designer's intention is perhaps that they indicate that both the button has to be pressed and coin inserted. For this device, our user model is instantiated with extra details about the coin slot. Checking the to-do list is automatically added to the guards so not needed in the instantiation.

```
ADVICE_DEVICE_USER (ustate: U) (mstate: M) =
  USER (
    (PressHappinessButton, InsertCoin, TakeCapsule),
    [(PressHappinessButton, HappinessButtonPressed),
     (InsertCoin, CoinInserted),
     (TakeCapsule, CapsuleTaken)],
    [(HappinessbuttonLight, SeeHappinessbuttonLight),
     (CoinSlotLight, SeeCoinSlotLight),
     (CapsuleReleased, SeeCapsule)]
    [(SeeHappinessbuttonLight, PressHappinessButton),
     (SeeCoinSlotLight, InsertCoin),
     (SeeCapsule, TakeCapsule)]
    FinishInteraction)
    ustate mstate
```

With this enhanced user model, our verification property will only be provable if the light for the button comes on only at times when the action is accepted. If the device appears to prompt for the button to be pressed but ignores any button press until after the coin is inserted, the model will ultimately abort on some paths, because the button will no longer be a candidate to be pressed if pressed too soon. No guard will be active. The verification will therefore fail.

Similarly, verification would fail, highlighting the usability problem, if the device design is such that the prompt comes fractionally too early, before the device accepts the corresponding button press: a common problem in interactive systems.

As we add new classes of rule to the user model as we are doing here, it opens up the possibility of exploring the consequence of different user behaviours. We can explore what happens when a user does just follow prompts blindly—a completely novice user perhaps. Then using the more complex version of the model with more complex guards, explore the consequences of a slightly more sophisticated user who knows something of what they are trying to do.

8.3.4 Termination Behaviour

Another area of behaviour that can be easily modelled in this framework concerns more subtle reasons for when a person might plausibly terminate an activity. We have considered abnormal termination when there is nothing apparently possible. However, there are more positive reasons for termination. In situations where the activity has a well-defined goal, then the obvious reason to terminate is that that goal has been achieved. This can be added as an extra guarded rule with goal achievement as the guard and an action to terminate the interaction as the action.

```
COMPLETION actions finished goalAchieved ustate t =
  goalAchieved ustate t ∧ NEXT actions finished t
```

For our *Advice Device* the goal is that the person has an advice capsule. It can therefore be set (i.e. passed as a new argument to the user model) as `takeCapsule`.

Since the verification property we are proving is also about achieving the goal, we can now create a generic verification property too: that the goal is achieved given an initial state.

```
DEVICE_IS_USABLE initstate goalachieved =
  ∀t: time.
    initstate t ⊃
      EVENTUALLY goalachieved t
```

Termination conditions are more subtle than this though. In many interaction scenarios, the user not only needs to achieve their goal but return the system to some initial state, ready for the next interaction. This may coincide with the goal but it may not. Examples are logging out of a system, retrieving inserted cards, switching off a light on leaving a room, switching off the headlights and locking a car door on arriving at a destination, and so on.

We can address this by adding an 'interaction invariant' property to the verification property. We then prove that, not only should the goal be achieved, but the invariant should also be restored. It is an invariant in the sense that it should always be restored at the end of an interaction (it is equivalent to a loop invariant in programming that must be restored at the end of the loop, where here it is the interaction that is looping).

```
∀t: time.
  initstate t ⊃
    EVENTUALLY (goalachieved ∧ invariant) t
```

This allows us to show that the user model can make another class of usability-based error: the post-completion error (Byrne and Bovair 1997). If the goal itself is achieved before all the tidying up actions, such as those above, that are necessary have been achieved, then those extra actions may be omitted.

For example, suppose our *Advice Device* requires the person to press a *reset* button to indicate they have taken the capsule, before it resets to allow the next customer to get a message. The goal of the user is still to take the capsule. The invariant, however, is that the machine has returned to the start state as indicated by the happiness light being on.

```
happinessbuttonLight mstate t = TRUE
```

We can extend the user model with extra buttons and lights corresponding to the *reset* button. However, we will not be able to prove that eventually the goal is achieved and invariant restored. Once the goal is achieved, the user model can finish the interaction without pressing the button. People would sometimes make the post-completion error. This is poor design, and the versions we have already considered that reset automatically are better. We will return to post-completion error in the next section.

8.3.5 Modelling the Physical World

In many interaction situations, there is more than just a user and a device. Systems are much wider than this, and this wider system and context often has to be taken into account and so modelled too. A simple way to deal with some aspects of the context is by including in the system specification a relation describing physical laws that must always hold and so are universally quantified over time. One practical example is about physical objects. We insert cards and coins, and obtain goods, for example, even in the simplest of walk-up-and-use machines. We therefore include laws about possessions such as that if a user gives up a possession (e.g. by inserting it in a device), then they no longer have that possession.

For each group of possessions, such as a particular kind of coin, we can define a relation over the states HAS_POSSESSION which keeps track over time of that possession: whether a person has a possession, how many they possesses at each time instance, and whether they take a new such possession (e.g. from a device) or give one up. This relation takes as arguments accessor functions in to the states of the user and device. An example law about possessions is that if at a moment in time the action to take a possession occurs and the user does not also give that possession up, then the number of that possession held goes up by 1 at the next time instance.

```
HAS_POSSESSION
    (takepossession, givepossession,
     countpossession, haspossession)
    (ustate:'u) (mstate:'m) =
  (∀t. (takepossession mstate t ∧
        ¬(givepossession mstate t)) =
        (countpossession ustate (t+1) =
          countpossession ustate t + 1 )) ∧
      ...
```

The value of possessions might additionally be recorded. This gives a way of reasoning about, for example, taking cash from a cash machine.

Separate universal laws are then combined in to a single relation to either incorporate in to the user model itself or as a separate context relation included in the system model.

```
UNIVERSAL_LAWS possessions ... ustate mstate =
  HAS_POSSESSION possessions ustate mstate ∧
  ...
```

Instantiating the user model now involves also providing as argument concrete details of specific possessions. We can also use facts about possessions as part of the guards of rules. For example, suppose our *Advice Device*now involves payment by card. The card is inserted at the start and returned at the end after the capsule has been taken. The card returns to its initial state automatically once the card is returned. Flashing lights prompt each step. We extend our instantiated user model accordingly with the extra actions and perceptions. The user has to have a card to insert it, and so this will become part of the guard to the reactive rule.

The goal can now be specified, rather than as the action to take the capsule but, treating it as a possession, that the user has a capsule. The invariant now is not just that the machine resets, but also that the user has the card, also treated as a possession. We need to verify the invariant:

```
happinessbuttonLight mstate t ∧
hasCard ustate t
```

A more general alternative for the invariant would be that the total value of the person's possessions is the same as it was at the outset.

In either case, this verification will fail as the user model achieves the goal and so can terminate before the invariant is restored: before taking the card back. To fix this problem and allow the verification to succeed, we need the device to return the card and only then release the capsule. Then, the only traces allowed by the user model involve the invariant becoming before the goal is achieved.

8.4 Internally Prompted Behaviour

Our model as discussed to date has focussed mainly around variations in reactive behaviour cued by external prompts. External stimuli are not the only kind of prompts that lead us to take action. We also have knowledge and/or experience of the task we are attempting to do, though possibly with other device designs or in other situations. This results in internal prompts. We already made use of this partially in the guards of our reactive rules as the basis of giving a motivation for prompts to be followed. However, we can also use it in more fundamental ways.

8.4.1 Cognitively-Cued Behaviour

As we have already discussed, we often know of actions we must take to achieve a task. The device we are using may change, but particular things must be done whatever the device. For example, to obtain money from a cash machine we must indicate how much money we want in some way, we must indicate the account it is to come from, and so on. These subgoals might concern information that must be communicated to the device, such as the dose of a drug to be infused or items that must be used with the device, such as a card that must be swiped to indicate the account or authenticate the user. Actions related to these subgoals are likely to just spring to mind even without explicit sensory cues, especially if the opportunity to discharge them is obvious. If we are holding a coin, know we need to pay for our happiness advice and see the coin slot, we may insert the coin, even if sensory cues are suggesting some other action, because our attention has focussed on the coin slot.

We can extend our generic user model to allow for this kind of behaviour by adding a new family of non-deterministic rules. We just indicate the list of actions with their preconditions, corresponding to this behaviour. Not all possible actions will be included. For example, some actions may be device specific, such as confirmation buttons after particular actions. Those included are the ones that are necessary for the task, whatever the design of the device. The guards in this family of rules do not need to include explicit sensory cues, as with reactive rules. Instead they correspond to whether the action is available and also whether the person has the information or object needed to discharge the goal.

If the guard is true, suggesting the opportunity is there, people may take the action so as to discharge the subgoals. This could happen even if reactive prompts are present. The two families of rules, sensory-cued and cognitively-cued behaviour, are set up in the generic user model as non-deterministic options.

Including this behaviour in a user model allows usability issues to be detected where internal knowledge leads to external prompts being ignored. If you are holding your identity card, and first see the card reader, you may swipe it first even if a flashing message on the screen is suggesting you must touch the screen first. Unless the design of the interaction is such that alternate actions can be taken in any order and still achieve the task, verification with the user model will fail, identifying the traces that lead to the problems. This leads to a permissive style of interaction design (Thimbleby 2001).

For our *Advice Device*where we must pay using coins, and also press the happiness button (perhaps now one of several choices of advice the device offers us), then take the capsule, the cognitive cued action might be set to correspond to inserting a coin and pressing the button with our choice. The additional argument included in the generic user model that we would need to provide as part of the instantiation would then be

```
[(SeesHappinessButton, PressHappinessButton),
 (SeesCoinSlot ∧ HasCoin, InsertCoin)]
```

The generic rule will include as part of the guard that the action is on the user models to-do list still. Once we have made our choice by pressing the button, we would not expect to do it again. In effect, the first guard, for example, is automatically expanded to:

```
(ToDo PressHappinessButton) ∧ SeesHappinessButton
```

The generic user model could also in fact automatically generate other parts of the cues. In particular, in any rule that involves an action, the generic rule could automatically add the linked history function corresponding to the opportunity for doing it being seen.

With this *Advice Device* user model, the verification will fail if the button and coin have to be inserted in a specific order. This will still be true even if the reactive prompts correctly guide the user through the correct sequence. Why? Because whatever action is required first, the other might spring to mind and be done as the coin slot and button are there to be seen. The user's attention may be focussed on the choices, for example, and so they might not see the flashing light next to the coin slot higher up on the device. Only when they have pressed the button might their attention be freed up to notice the flashing light. For the user model, the rules are non-deterministic: both the behaviour of following the cue and so inserting the coin, and of following the internal cue and pressing the button, satisfy the model. Both are therefore possible traces.

This usability problem could be solved in several ways. One is to make the design permissive with respect to these actions. The device does not need to care in what order they are done, just that they are done. Alternatively, it could ensure that the apparent opportunity to do invalid actions does not arise. For example, a touch screen could be used instead of physical buttons, so that the choices, and corresponding button, only appear once the coins have been inserted. In either case, the device design could be verified: despite all the limitations embodied in the user model, we would be able to prove it always achieved the goal with a good enough design.

8.4.2 Procedurally-Cued Behaviour

Another form of internal prompting that occurs is due to a person having followed a procedure in the past and so internalised a sequence of actions to be done one after the other, leading to habitual behaviour. A linkage builds up between them in the person's head so that one cues the next in that sequence. A user model that includes procedural cues can make capture errors. This corresponds to the situation where a person follows the habitual sequence without consciously intending to in an inappropriate situation.

This can be modelled in a simple way within our framework. Now the guard to each action in a habitual sequence is that the triggering action was done in the previous step. We can extend the generic user model to take as a new argument lists of such actions, turning them in to the appropriate rules as for the previous families of behaviour.

For our running example, suppose a person has been using an *Advice Device* regularly that allows a choice between asking for happiness or asking for sadness advice. As a result, they have built up a habit of inserting the coin, making their choice between happiness or sadness and then taking the capsule. The device uses a screen that displays the labels for adjacent physical buttons, indicating what they do at any particular time. We thus split the buttons from their function, since it changes. When it appears, the happiness button is button 1. As part of the instantiation, we provide an additional argument to the user model of this habitual sequence:

```
[insertCoin, pressButton1, takeCapsule]
```

However, suppose a new version of the software is released. To give more personalised information before the happiness options are given, the user is now asked to press one of the two buttons to indicate whether they are under or over 16. The same buttons are used as for indicating the kind of advice wanted. With procedural cuing, the user model will prevent successful verification of this new version of the device if this scenario is explored. This is because a behaviour of the user model is allowed where an adult user, who has built up the habit, presses button 1 thinking they are choosing happiness but instead indicating they are under 16.

This might be avoided, allowing for successful verification against such a scenario, by using different sets of buttons for the two questions, or asking for the new information later in the interaction when it has no correspondence to something that might be habitual.

For a more serious example, suppose a nurse normally uses one kind of infusion pump, where the sequence of actions is to authenticate, then enter the volume to be infused and then the rate of infusion. Each of those actions will start to act as a cue for the next. If the nurse switches to a new ward where a different pump is used that requires the rate to be input before the volume, then a capture error may result. The procedural cue from authentication may lead to the nurse entering the volume next even if the screen is clearly asking for the rate. If not noticed, this could lead to the patient being given an overdose.

By adding procedural cuing, we are able to explore the ramifications of habits in a scenario-driven way. This could be part of a verification of a specific design or as part of a hazard analysis aiming to identify hazards. Such habits may cause problems on the introduction of new devices, new versions of old devices being installed, or just as a result of a change of context as with the nurse. It may not always be possible to eliminate problems that arise, but such verification might at least mean the risk can be highlighted. This is done when road layouts are changed: new signs are put up indicating there has been a change to help avoid habitual errors. Nurses can at least be trained in the specific risks and warned in situations such as a ward change where the risks are increased. More robust designs that support error detection and recovery might be developed. For example, extra screens might ask the user to review the information, undo buttons might be provided, or error detection software included, for example, that notices dangerously high doses being entered for the drug used.

8.4.3 Mental Commit Actions

Performing an action has two stages: an internal action when a person mentally commits to acting, for example due to the internal goals or as a response to the interface prompts, and a physical action when the actual action is taken in the world. This distinction reflects a delay between the commitment moment and the moment when the physical action is taken. Once a signal has been sent from the brain to the motor system to take an action, it cannot be revoked after a certain point, even if the person becomes aware that it is wrong before the action is taken. We can capture this by pairing guarded rules about pairs of actions. Each physical action has as its guard that the mental action has occurred. This means that it is guaranteed to follow the committing mental action after some delay. The guards of the mental action are those that correspond to taking the action and hold true up until the mental action is taken. After that point, they may or may not hold, but the physical action will be taken either way. This infrastructure for pairing guards can be part of the generic model rather than pairs of actions needing to be instantiated explicitly. A list of mental actions and corresponding physical action can be given to be paired up in this way.

This can be used to highlight situations where an interface changes between a user committing to an action and the action being taken, so that the stimuli they are reacting to and the consequence of an action have changed. For example, suppose the *Advice Device* has periods when it stops accepting input temporally when not being used so as to upload log data to a central server. When it goes offline, a please wait sign is displayed. The user model, with the mental–physical action distinction, could now commit to an action of pressing the button at a point when the light indicating it can be pressed was still flashing, but when it actually pressed the button it was no longer active. Such a usability problem might be avoided by the machine only going offline in this way immediately at the end of an interaction so that the light indicating the button could be pressed would not come on only to go off again.

A more serious example might be an air traffic control system where flights appear in a window in a queue-like manner, with each entry moving down a place each time a new flight arrives. In this system, the act of touching the flight record on the screen performs an action on that flight. A flight controller could make a decision to touch a flight, but after this commitment is made, a new flight arrives and the flight actually touched is a different one to that originally perceived.

8.4.4 Case Studies

We have applied models based on simple rules such as those described in this and the last section to a variety of simple case studies based on various walk-up-and-use machines such as cash points and vending machines, similar to our running example.

Through these simple examples, we have shown that design flaws can be detected that lead to all the various errors discussed including post-completion errors, termination due to no clear guidance as to what to do, order errors due, for example, to a lack of permissiveness of actions, where internal knowledge guides a user to a different action to prompts, and errors due to habitual behaviour interfering with what is required of a new design. We have also shown how soft-key interfaces using touch screens can be verified and related problems identified.

In these examples, we have also shown how these problems can be eliminated with appropriate design changes such as those discussed in the previous sections, verifying successfully modified designs. For example, by changing the order of actions, we have verified that post-completion errors in cash and vending machines with respect to left cards or change have been eliminated without introducing new problems for the user model.

This demonstrates that the approach can be used, even with fairly simple principles and rules, to go beyond assumptions of angelic or demonic users to users who do make mistakes. It can detect a variety of usability issues with interface designs.

In conducting even these simple case studies, we also found problems that were only discovered as a result of the verification. They had not purposefully been designed in to the example so not expected when the case study was created. For example, in one simple walk-up-and-use device design considered the model would terminate early as there was no plausible option according to the rules. To fix this, 'please wait' feedback was needed as discussed earlier. This was only realised when exploring the consequences of the model (Curzon and Blandford 2001). The need for 'please wait' feedback had not been explicitly considered at all at this point. This example of poor design emerged from cognitively plausible principles included for other reasons. Similarly, another possible human error unexpectedly arose in a design with insufficient prompting. This was due to a user model belief that all actions relevant to the task had already been completed (resulting from an order error). This was discovered when using a user model set to explore novice behaviour (Curzon et al. 2007).

8.5 A More Complex Salience Model

The above models are based on a simple set of general principles but despite that can detect a wide range of usability issues, so in a sense provide an interesting compromise between the angelic and demonic extremes. However, for complex interactive systems more complex models may be needed if subtler issues are to be found. Interactive systems often have multiple cues in play of different strengths at any time. We have therefore explored more complex models, with a focus on the different kinds of internal and external cues potentially having differing strengths.

The model was based on ideas from activation theory (Altmann and Trafton 2002) and the results of a series of experiments (discussed in Sect. 8.6.2). The impetus to take a particular action can depend on a mixture of internal and external cues that each contribute to the overall level of salience of the action itself. The idea is that actions which have most activation from the combination of internal and external cues driving them are the ones most likely to be taken.

We consider three kinds of cue salience: corresponding to the families of cues already discussed: two internal (in-the-head) forms of salience: procedural salience and cognitive salience, and external (in-the-world) sensory salience.

We applied variations of models that include salience to a graphical user interface version of a fire-engine dispatch system (Rukšėnas et al. 2008a, 2009), as well as medical interfaces such as infusion pump use as part of the CHI+MED project (Curzon et al. 2015).

Determining the levels of salience for a particular device specification and scenario would require a human-computer interaction specialist, though it might possibly be done, at least in part, automatically in the future using programs modelling the human vision system. An alternative way to view this though is that the user model can be used as a tool to explore potential consequences of different cues having different levels of salience and so recommend where particular cues need to be strong, and where it is less essential.

8.5.1 Different Kinds of Salience

An action has procedural salience as a result of a person forming a habit due to previously taking actions in sequence. A linkage builds up between them in the person's head so that one cues the next in that sequence. This can lead to capture errors. An example of this as we saw might occur where a nurse switched from a ward using one model of infusion pump to a different ward, where the settings are the inputs in a different order. This may lead occasionally to the nurse entering things in the old order into the new device. The salience of the procedural cuing may depend on how practiced the previous habit was.

Cognitive salience refers to the situation where the need to do an action just springs-to-mind due to it being associated with the current activity being performed. This is the salience linked to cognitive cues. For example, when using a cash machine to get cash you know that you are likely to have to indicate the amount of cash you want in some form: that amount is probably on your mind anyway as it is part of the task you are trying to complete. How and when you need to provide this information will differ from device to device, but on seeing an opportunity you may try and provide it. The strength of the salience may depend on how central to the task the action is, for example.

Sensory salience is concerned with cues in the world and corresponds to reactive behaviour. These external cues could be related to any of the senses though typically are in the form of visual or audible cues. The salience strength may depend on the kind of alert: a flashing message will have higher salience than a small light going on. A klaxon sounding will have high salience.

8.5.2 Load

Experiments suggest that the salience of cues depends on the load on the user at the time. We therefore explored generic user models that take into account two kinds of such load: intrinsic load and extraneous load. Intrinsic load is the inherent difficulty of the task at hand. For example, if an operator such as an emergency dispatcher has to do complex mental calculations to decide what to do, then the load for a task might be classified as high. Extraneous load arises from information that is present but that does not contribute directly to the performance of a specific goal. The need to do visual search to find relevant information on the device that in itself does not foster the process of performing a goal can be classified as extraneous.

Load levels can be included in the model as history functions. As with the salience levels, determining load levels to instantiate the user model with would need to be done by an expert in human-computer interaction or automated programs, but the user model could be used to explore where load needed to be reduced to make the device usable.

Experiments suggested that intrinsic load can influence the strength of procedural cues used to perform future task critical actions. Extraneous load also influences the awareness individuals have of sensory cues when intrinsic load is high (Rukšėnas et al. 2009). We modelled rules based on these results. Note that the work described here was developed in SAL, so used a different formalism of the model to our earlier work. In the following, we sketch the approach following the notation we have used for the rest of the paper.

8.5.3 Combining Salience

To model the interactions of different cues requires that each is given a level of salience. To keep things simple, we modelled the strength with a small number of levels. There may be no salience (there is no cue at a particular time), some small level of salience (such as an LED prompt being there to be seen) or they may have higher levels (such as an audible alarm).

We calculate the combined salience of cues ultimately driving the action as a similar series of levels of priorities, though with an additional intermediate level. The salience from the different kinds of cues for an action are combined according to rules about their relative strengths and the effects of the load on the user. Actions of the highest overall saliency level are taken if there are any. If not, then those of the next highest level are taken, with those with the lowest level of saliency only being taken if there are none of the higher levels.

In the simple model, an action might have several rules that cause it to fire, each based on different kinds of cues. In this new model, each action needs only one rule with the separate cues now combining based on their salience to form part of the guard of that single rule.

The generic action rules therefore include an algorithm that calculates whether there are no other actions of higher salience that are otherwise active. This is then included in the guard of the rule so that only the actions cued with the highest salience can be taken. There still may be multiple actions falling in to this highest salience category, so it is still non-deterministic.

To do this, we create a series of predicates to calculate the overall salience of an action: `HighestSalience`, `MediumSalience` and `LowSalience`, each given arguments specifying the level of the different kinds of salience and load. The guards contain an expression of the following form:

```
(HighestSalience( ... ) ∨
(HighSalience( ... ) ∧
    no action is of highest salience) ∨
(LowSalience( ... ) ∧
    no action is of high or highest salience)
```

The levels of the various forms of salience for each action are each computed separately and combined to give the above levels of salience. The highest salience level arises, for example, if the sensory or cognitive cuing are high or if procedural cueing is not low.

```
HighestSalience( ... ) =
  ...
  (¬ (proceduralSaliency t = Low) ∨
  (sensorySaliency t = High) ∨
  (cognitiveSaliency t = High))
```

The following algorithm for procedural saliency shows how we model load as it affects the different cue saliencies:

```
if (procedural_default = High) ∧ (intrinsicLoad t = High)
then proceduralSaliency (t+1) = Medium
else proceduralSaliency (t+1) = procedural_default
```

This states that procedural cue saliency is normally unchanged from its default level set for that action. However, intrinsic load lowers otherwise high procedural cueing if the load is high.

8.6 Alternate Uses of Generic User Models

So far we have considered the basic use of generic user models to evaluate the usability of a system, with a focus on finding design flaws that lead to user error. We have also explored other ways to use the basic models, however. We sketch these ideas in this section.

8.6.1 Combining Error Analysis with Timing Analysis

There are other aspects to usability beyond human error, and human error analysis does not need to be performed in isolation. It can explicitly complement other methods. Timing analysis is, for example, a successful application of cognitive modelling as embodied in the variations of GOMS models (John and Kieras 1996) and in the CogTools toolset (John et al. 2004). We have shown how this kind of modelling can be done formally within the same user modelling framework as we have used to do human error analysis (Rukšėnas et al. 2014).

Since GOMS models are deterministic, these predictions assume and apply to a single, usually ‘expert’, or optimal, sequence of operators. They assume a form of angelic behaviour, though introducing aspects of cognitive limitations. However, such perfect behaviour cannot always be assumed. With walk-up-and-use interactive systems, typical users are not experts. They are not trained to follow optimal procedures and may not even be aware of them. They are liable to choose less cognitively demanding methods given the opportunity. Even the expert users of safety-critical systems when under pressure may choose suboptimal plans of action, either consciously or unconsciously, due to limitations in their cognitive resources. It is therefore worth including a broader set of cognitively plausible behaviours in to such a KLM-GOMS style timing analysis of interactive systems going beyond just optimal paths.

A generic cognitive model, of the form discussed in this chapter, gives a fairly straightforward way to do this. It involves only a simple adaptation of the model. Timing information is attached to the actions. In addition, mental operators are added before cognitive units, though not actions that can be predicted in advance. This follows a general principle common to GOMS analysis. In adding timings in this way, we obtain a way to integrate timing analysis with user error-based analysis. By adding the timings on paths explored, the timing consequences of taking those paths can be determined. Using model checking, bounds can be predicted on the time taken to complete tasks, based on exhaustive exploration of cognitively plausible paths.

We have applied this approach to both simple scenarios involving cash machines (Rukšėnas et al. 2007b) and in a more complex, safety-critical context of programming an infusion pump (Rukšėnas et al. 2014), illustrating how the approach identifies timing and human error issues. In particular, it predicts bounds for task completion times by exhaustive state-space exploration of the behaviours allowed by the cognitive model. At the same time, it supports the detection of user error-related design issues. The performance consequences of users recovering from likely errors can be investigated in this framework. It thus gives a novel way to explore the consequences of cognitive mismatches on both correctness and performance grounds at the same time. Our case studies (Rukšėnas et al. 2014) showed how an initial design can be explored where performance issues are identified, and then how alternate designs aiming to fix the problem can then be analysed. This can highlight how fixes to the performance issues found might then lead to unexpected human error consequences. The variations considered in the case studies were based on features found in real designs.

8.6.2 Supporting Experiments by Exploring Behavioural Assumptions

An issue about modelling user behaviour is whether the assumptions embodied in the model are actually psychologically valid. Is one kind of cue actively weaker than another in all contexts, for example? Cue strength almost certainly depends on context in subtle ways, meaning that just because experiments have shown that rules apply in one situation may not mean they remain valid in others. Our model of salience was based on a combination of theory and experiments, but changing the context could change the validity of the model.

Ultimately, the principles embodied in a user model and then the model itself need to be validated experimentally across a wide range of situations. These are open questions for psychology and experimental human-computer interaction researchers. This issue is an important advantage of a generic user model approach, however. The core principles and model only need to be validated once. If hand-crafted user models are created for each new application, then the behaviour embodied in each would need to be validated. If instead of user modelling, other techniques are used where the behaviour assumed is implicit in the analysis, then there is no inspection at all of the assumptions made.

However, whatever analysis technique is used, even without rigorous experimental validation, the designers of interfaces are making assumptions about such issues, even if implicitly, every time they design. User models therefore have another role to play. They give a way to make the assumptions being made about the user, as well as the device, explicit. The combination of the user model and its instantiation does this. For example, with respect to salience, the assumptions about how actions are cued and the relative importance of those different cues are made explicit when the user model is instantiated.

Furthermore, rather than just treating the analysis as a way to find design flaws, we can treat user modelling as a way to explore the consequences of the assumptions made about user behaviour, for the usability of a particular design. This can be done exhaustively in a way that cannot be done within experiments or simulation. A similar approach can also be followed to support more basic experimental human-computer interaction research, including research aimed at understanding cognitively plausible behaviour. Formal methods give increased analytical power to experimental results, while experimental results inform formal modelling.

We trialled this idea with respect to developing the saliency rules described above (Rukšėnas et al. 2009). Experiments and formal modelling then went hand in hand to inform each other. The results of an initial experiment were turned in to a user model embodying the behaviour assumed. Exhaustive modelling showed that the hypothesis about the relationships between different cues made did not completely explain the behaviour recorded, however. It also suggested that variants might fix the problem. Such variants were then modelled.

This approach can also be used to suggest gaps in knowledge from empirical research, an approach we have trialled in a slightly different context of modelling processes in tandem with field work to support and understand a distributed cognition analysis (Masci et al. 2015a).

To facilitate the general approach of exploring assumptions, making it more flexible, we restructured the architecture of the generic user model in to three layers. The lowest layer captures core assumptions that are unlikely to be modified by the approach. The intermediate layer, also part of the generic model, specifies the set of cognitive assumptions being explored. The third layer is the instantiation of the generic model and captures specific assumptions that relate to the details of the device and the task to be performed on it (Rukšėnas et al. 2013).

We used this restructured model with experiments investigating the task of setting up sets of infusion pumps for a single patient, where the separate tasks corresponding to each pump can be done either sequentially or interleaved (Rukšėnas et al. 2013). In this particular study, modelling was based on a set of assumptions about behaviour derived from the soft constraints hypothesis (Gray et al. 2006) which concerns the way resource allocation affects behaviour. These assumptions formed the basis of the experimental hypothesis. The experimental study appeared to back up this assumption about how people would behave. The exhaustive analysis also predicted that errors would be made that matched those observed in the experiment.

At first sight, this suggests that the experimental results were fully supported by the formal analysis. However, the formal analysis highlighted a mismatch. The specific erroneous traces identified did not actually match the sequence of actions of the participants in the experiment. The formal analysis predicted that a different form of interleaving between the tasks would be followed. This suggests that the modelled assumptions do not fully explain the observed behaviour. Therefore, more experimentation and/or modelling is needed to fully understand the behaviour and why the detail of the actual behaviour deviated from that predicted.

This work shows how formal methods can be integrated with laboratory-based user evaluation and experimentation. It can either give increased confidence in the results of the evaluation or find gaps that might otherwise be missed. It helps the evaluator to consider the validity of the experimental design and so provides insight into the confidence with which the results of the experiment and its interpretation can be considered. It can highlight mismatches between the consequences of the assumptions and experimental results and so lead to suggestions of further experiments as well as ruling out potential explanations for those mismatches.

The approach parallels the use of cognitive modelling to analyse user assumptions (Ritter and Young 2001) using cognitive architectures. However, cognitive architectures provide extremely detailed, though deterministic, models of the cognitive processes, such as visual or auditory perception, memory and learning, whereas the above approach works at a much more abstract level. Cognitive modelling approaches use simulation runs to analyse system properties. This means a relatively small range of behaviour is actually considered. The idea is that each deterministic trace represents average behaviour. This contrasts with the non-determinism of our models which generate a wider range of behaviours. As our approach is based on automated reasoning, not simulation, we can explore the consequences of the assumptions made exhaustively.

8.6.3 Hazards, Requirements and Design Rules

A user model can also be used to justify usability requirements or design rules (Curzon and Blandford 2002, 2004), making explicit the assumptions they are based on and the consequence of applying them. Design rules can be derived informally from the user model. We did this with a variety of design rules based on one of our earlier models of behaviour:

- allow the task to be finished no later than the goal;
- provide information about what to do;
- providing information is not enough;
- use forcing functions so that other options are not available;
- interfaces should be permissive where possible;
- controls should make visible their use;
- give immediate feedback;
- do not change the interface under the user's feet; and
- where possible, determine the user's task early.

Such rules or requirements can also be verified. Design rule correctness statements are formally stated and verified against assumptions embodied in the user model. This involves replacing the device description with a formal statement of the design rule. It then acts as an abstract version of the interactive system. The property verified is then a statement that the aim of the design rule has been achieved. We demonstrated this with respect to the first, post-completion error linked design rule.

An alternative way of thinking of this is that it gives a way to explore the potential hazards that might arise from a user acting according to the cognitively plausible behaviour. Our discussion of each principle added involved noting the consequences of including the behaviour: essentially identifying potential hazards implied by the assumptions of behaviour. This approach could be incorporated in to a more rigorous hazard analysis process when combined with a reference model of a device, for example. This is essentially what is done by Masci et al. (2015b), developing a hazard analysis process, though based on an informal model of human behaviour. A formal generic user model could support a more formal version of this process, with respect to the user's behaviour not just the device, and apply to families of similar devices. By exploring different assumptions in the user model, their effect on hazards could be investigated. As with designs, a user model could be used more generally to explore how different assumptions about plausible behaviour affect design rules, including about how the context affects their effectiveness.

8.6.4 Security Analysis

The focus of most work with user models has concerned usability. However, there are also applications to security. Most security research focuses on the technical aspects of systems. We have considered security from a user-centred point of view using a generic user model, but verifying the system against security rather than usability goals. This gives a focus on the cognitive processes that influence the security of information flow between the user and the computer system, for example. Examples have shown how some confidentiality leaks, caused by a combination of an inappropriate design and certain aspects of human cognition, can be detected within our framework (Rukšėnas et al. 2007a, 2008b).

This kind of analysis has been suggested as a part of a concertina-verification approach (Bella et al. 2015) for verifying security ceremonies: an extension of a security protocol that takes wider issues such as human factors into account. This involves layering the security analysis into a series of models extending out from the security protocol itself to human interaction, and then further outwards to consider the wider socio-technical aspects of the system. These ideas have not been tested in depth, and the possibilities for formal user models to be used in this context need to be investigated.

8.7 Other Forms of User Model

Our approach to modelling user behaviour has been to focus on specific individual aspects of behaviour such as the effect of the salience of cues. Other aspects can be formally modelled, however, and have been done in a variety of work.

8.7.1 Interactive Cognitive Subsystems

Duke et al. (1998) and Bowman and Faconti (2000) use interactive cognitive subsystems (ICS) Barnard and May (1995) as the underlying model of human information processing in their work. This work is based on lower-level cognitive models and has a focus on simulation. Their models deal with information flow between the different cognitive subsystems and constraints on the associated processes that transform the information, including situations where the interaction of different paths blocks some information flows. As a result, their work focusses on reasoning about multimodal interfaces and analyses whether interfaces based on several simultaneous modes of interaction are compatible with the capabilities of human cognition. Su et al. (2009) developed an ICS-based formal cognitive model to explore low-level psychology phenomena, and in particular, attentional blink. This concerns the deployment of temporal attention and how the salience of items is affected by their meaning. They have also adopted the model to simulate interactions in the presence of information overload. This illustrates how user models can support psychology experiments as well as more applied human-computer interaction experiments.

8.7.2 Mental Models

People form mental models as a result of the system image they are working with. One source of usability problem and ultimately human error results from people forming a mental model that does not exactly match the way the system works. Rushby (2001, 2002) formalised plausible mental models of systems, looking for discrepancies between these and actual system behaviour. The analyses were specifically concerned with automation surprises, mode errors and the ability of pilots to track mode changes. The mental models are a form of user model, but simplified system models rather than cognitive models. They can be developed in various ways, including an empirical approach based on questioning pilots about their beliefs about the system. Like interface-oriented approaches, each model is individually hand-crafted for each new device. Bisimulation is used to determine whether any important divergence results from following the abstraction, that is the mental model rather than an accurate model of the system. In the original work, the device and user models were intertwined. Lankenau (2001) and Bredereke and Lankenau (2004) extended the approach, making a clear separation so that non-existent dependencies could not arise in the models. They also include a relation between environment events and mental events that could in principle be lossy. This corresponds to an operator not observing all interface events of a system. However, in practice in their work the relation does no more than renaming of events and so is not lossy. They do not therefore explicitly consider the operator's imperfections. Buth (2004) also explores the use of CSP refinement as a way of analysing the system.

The approach to modelling user behaviour we have described above is essentially a threshold model. All behaviour is possible (and may be seen if a user is malicious, trying to subvert the system); however, the concern is to model cognitively plausible behaviour, i.e. behaviour that is likely to happen for systematic reasons and so passes some threshold of concern that it may occur. An alternative is to create probabilistic models where probabilities are assigned to actions. To be of practical use, empirical work is needed to determine the probabilities of actions in different contexts.

8.8 Future Challenges

We have investigated only a small set of behaviours, and for more complex interactive systems, a wider form of behaviour is likely to be needed. We must integrate more forms of cognitively plausible behaviour in to the models. Multitasking, for example, is a critical feature of much human endeavour in the modern world for which there are strong theories based on decades of experiments. Developing formal models of these theories in a form that could be integrated with this kind of user model is an important area for further research.

Any model if it is to be used in industrial contexts ideally needs to be validated experimentally. The principles embedded in the models have been based on those from the literature. The salience models were also the result of modelling the outcomes of experiments. However, the resulting model as a whole needs to be validated in other situations to those considered. Any model is by necessity an abstraction of the real world, and context matters a lot with respect to usability, so the results of a single set of laboratory-based experiments do not necessarily generalise to other situations. As the issues modelled become more subtle, the validity of the models in the situations analysed becomes more of an issue. At what point should a behaviour that leads to systematic human error be considered sufficiently likely to be included?

The work described has explored the feasibility of a range of possible uses for user models, but has generally only piloted ideas. Much more work is needed across a range of areas. The ideas need to be applied to a wider range of case studies and extended to be able to deal with more complex situations. The modelling described here has mainly been concerned with single user, single-device interactions. The focus of human-computer interaction has moved outwards to a concern with more complex systems of teams working with multiple interfaces and to issues of the wider socio-technical systems. Computing power is increasingly ubiquitous, embedded in objects, and interaction increasingly involves interacting with those objects as well as with more conventional interfaces. User modelling naturally extends to these kinds of situation. Various of our studies have involved such concerns at least in simple ways. The work on user-centred security involved the modelling of physical objects, and our work on infusion pumps involved the programming of multiple devices. However, this work has only touched the surface of the issue. Much more could be done in this area.

If formal, generic user model-based verification is to be adopted in industrial contexts, then work is needed on making the models accessible to non-experts, in terms of instantiating them, in running verification and in presenting the results in terms of the underlying principles responsible for failed behaviour traces. As with other formal techniques, ways of integrating user model-based methods in to existing engineering processes are needed if they are to be taken up widely.

Acknowledgements This work was in collaboration with a team of people. We are particularly grateful to Michael Harrison, Ann Blandford and Jonathan Back. Much of the work referred to was funded by the EPSRC research grants GR/S67494/01, GR/S67500/01 (Human Error Modelling) and EP/G059063/1 (CHI+MED Computer-Human Interaction for Medical Devices).

References

Altmann E, Trafton J (2002) Memory for goals: an activation-based model. Cognit Sci 26(1):39–83

Barnard PJ, May J (1995) Interactions with advanced graphical interfaces and the deployment of latent human knowledge. In: Interactive systems: design, specification, and verification (DSV-IS'95). Springer, pp 15–49

Bella G, Curzon P, Lenzini G (2015) Service security and privacy as a socio-technical problem: literature review, analysis methodology and challenge domains. J Comput Secur 23(5):563–585. doi:10.3233/JCS-150536

Bowman H, Faconti G (2000) Analysing cognitive behaviour using LOTOS and Mexitl. Formal Aspects Comput 11:132–159

Bredereke J, Lankenau A (2004) A rigorous view of mode confusion. In: Computer safety, reliability and security: SAFECOMP 2002. Lecture notes in computer science, vol 2434. Springer, pp 19–31

Buth B (2004) Analysing mode confusion: an approach using FDR2. In: Computer safety, reliability and security: SAFECOMP 2004. Lecture notes in computer science, vol 3219. Springer, pp 101–114

Butterworth RJ, Blandford AE, Duke DJ (2000) Demonstrating the cognitive plausibility of interactive systems. Formal Aspects Comput 237–259

Byrne M, Bovair S (1997) A working memory model of a common procedural error. Cognit Sci 21(1):31–61

Curzon P, Blandford A (2000) Using a verification system to reason about post-completion errors. In: Palanque P, Paternò F (eds) Participants Proc. of DSV-IS 2000: 7th International workshop on design, specification and verification of interactive systems, at the 22nd International conference on software engineerings, pp 292–308

Curzon P, Blandford A (2001) Detecting multiple classes of user errors. In: Little R, Nigay L (eds) Proceedings of the 8th IFIP working conference on engineering for human-computer interaction (EHCI'01). Lecture notes in computer science, vol 2254. Springer, pp 57–71. doi:10.1007/3-540-45348-2_9

Curzon P, Blandford A (2002) From a formal user model to design rules. In: Forbrig P, Urban B, Vanderdonckt J, Limbourg Q (eds) 9th International workshop on interactive systems. Design, specification and verification. Lecture notes in computer science, vol 2545. Springer, pp 1–15. doi:10.1007/3-540-36235-5_1

Curzon P, Blandford A (2004) Formally justifying user-centred design rules: a case study on post-completion errors. In: Boiten E, Derrick J, Smith G (eds) Proceedings of the 4th international conference on integrated formal methods. Lecture notes in computer science, vol 2999. Springer, pp 461–480. doi:10.1007/b96106

Curzon P, Rukšėnas R, Blandford A (2007) An approach to formal verification of human-computer interaction. Formal Aspects Comput 19(4):513–550. doi:10.1007/s00165-007-0035-6

Curzon P, Blandford A, Thimbleby H, Cox A (2015) Safer interactive medical device design: Insights from the CHI+MED project. In: 5th EAI International conference on wireless mobile communication and healthcare—Transforming healthcare through innovations in mobile and wireless technologies. ACM. doi:10.4108/eai.14-10-2015.2261752

Duke DJ, Barnard PJ, Duce DA, May J (1998) Syndetic modelling. Human Comput Interact 13(4):337–393

Furniss D, Blandford A, Curzon P (2008) Usability work in professional website design: Insights from practitioners' perspectives. In: Law E, Hvannberg E, Cockton G, Vanderdonckt J (eds) Maturing usability: quality in software. Springer, Interaction and value, pp 144–167

Gray W, Sims C, Fu W, Schoelles M (2006) The soft constraints hypothesis: a rational analysis approach to resource allocation for interactive behavior. Psychol Rev 113(3):461–482

Harrison M, Campos JC, Rukšėnas R, Curzon P (2016) Modelling information resources and their salience in medical device design. In: Engineering interactive computing systems. ACM, pp 194–203. doi:10.1145/2933242.2933250, honourable Mention Award:top 5% papers

John B, Kieras D (1996) The GOMS family of user interface analysis techniques: comparison and contrast. ACM Trans Comput-Hum Interact 3(4):320–351

John BE, Prevas K, Salvucci DD, Koedinger K (2004) Predictive human performance modeling made easy. In: Proceedings of the SIGCHI conference on human factors in computing systems, CHI '04, ACM, pp 455–462

Lankenau A (2001) Avoiding mode confusion in service-robots. In: Integration of assistive technology in the information age, Proceedings of the 7th international conference on rehabilitation robotics. IOS Press, pp 162–167

Masci P, Curzon P, Furniss D, Blandford A (2015a) Using PVS to support the analysis of distributed cognition systems. Innov Syst Softw Eng 11(2):113–130. doi:10.1007/s11334-013-0202-2

Masci P, Curzon P, Thimbleby H (2015b) Early identification of software causes of use-related hazards in medical devices. In: 5th EAI International conference on wireless mobile communication and healthcare—Transforming healthcare through innovations in mobile and wireless technologies. ACM. doi:10.4108/eai.14-10-2015.2261754

Newell A (1990) Unified theories of cognition. Harvard University Press

Nielsen J (2000) Why you only need to test with 5 users. http://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users/

Norman DA (2002) The design of everyday things. Basic Books

Ritter F, Young R (2001) Embodied models as simulated users: introduction to this special issue on using cognitive models to improve interface design. Int J Hum Comput Stud 55:1–14

Rukšėnas R, Curzon P, Blandford A (2007a) Detecting cognitive causes of confidentiality leaks. In: Curzon P, Cerone A (eds) Proceedings of the 1st International workshop on formal methods for interactive systems. Electronic notes in theoretical computer science, vol 183. Elsevier pp 21–38. doi:10.1016/j.entcs.2007.01.059

Rukšėnas R, Curzon P, Blandford A, Back J (2007b) Combining human error verification and timing analysis. In: Proceedings of engineering interactive systems 2007: Joining three working conferences IFIP WG2.7/13.4 10th conference on engineering human computer interaction, IFIP WG 13.2 1st conference on human centered software engineering, DSVIS—14th conference on design specification and verification of interactive systems. Lecture notes in computer science, vol 4940. Springer, pp 18–35. doi:10.1007/978-3-540-92698-6_2

Rukšėnas R, Curzon P, Back J, Blandford A (2008a) Formal modelling of salience and cognitive load. Electron Notes Theoret Comput Sci 208:57–75. doi:10.1016/j.entcs.2008.03.107

Rukšėnas R, Curzon P, Blandford A (2008b) Modelling and analysing cognitive causes of security breaches. Innov Syst Softw Eng 30(2):143–160. doi:10.1007/s11334-008-0050-7

Rukšėnas R, Curzon P, Blandford A (2009) Verification-guided modelling of salience and cognitive load. Formal Aspects Comput 21:541–569. doi:10.1007/s00165-008-0102-7

Rukšėnas R, Curzon P, Harrison MD (2013) Integrating formal predictions of interactive system behaviour with user evaluation. In: Johnsen EB, Petre L (eds) Proceedings of integrated formal methods. LNCS, vol 7940. Springer, pp 238–252. doi:10.1007/978-3-642-38613-8_17

Rukšėnas R, Curzon P, Blandford A, Back J (2014) Combining human error verification and timing analysis: a case study on an infusion pump. Formal Aspects Comput 26(5):1033–1076. doi:10.1007/s00165-013-0288-1

Rushby J (2001) Analyzing cockpit interfaces using formal methods. Electron Notes Theoret Comput Sci 43:1–14. doi:10.1016/S1571-0661(04)80891-0

Rushby J (2002) Using model checking to help discover mode confusions and other automation suprises. Reliab Eng System Safety 75(2):167–177

Su L, Bowman H, Barnard P, Wyble B (2009) Process algebraic modelling of attentional capture and human electrophysiology in interactive systems. Formal Aspects Comput 21:513–539

Thimbleby H (2001) Permissive user interfaces. Int J Hum-Comput Stud 54(3):333–350. doi:10.1006/ijhc.2000.0442

Chapter 9
Physigrams: Modelling Physical Device Characteristics Interaction

Alan Dix and Masitah Ghazali

Abstract In industrial control rooms, in our living rooms, and in our pockets, the devices that surround us combine physical controls with digital functionality. The use of a device, including its safety, usability and user experience, is a product of the conjoint behaviour of the physical and digital aspects of the device. However, this is often complex; there are multiple feedback pathways, from the look, sound and feel of the physical controls themselves, to digital displays or the effect of computation on physical actuators such as a washing machine or nuclear power station. Physigrams allow us to focus on the first of these, the very direct interaction potential of the controls themselves, initially divorced from any further electronic or digital effects—that is studying the device 'unplugged'. This modelling uses a variant of state transition networks, but customised to deal with physical rather than logical actions. This physical-level model can then be connected to underlying logical action models as are commonly found in formal user interface modelling. This chapter describes the multiple feedback loops between users and systems, highlighting the physical and digital channels and the different effects on the user. It then demonstrates physigrams using a small number of increasingly complex examples. The techniques developed are then applied to the control panel of a wind turbine. Finally, it discusses a number of the open problems in using this kind of framework. This will include practical issues such as level of detail and times when it feels natural to let some of the digital state 'bleed back' into a physigram. It will also include theoretical issues, notably the problem of having a sufficiently rich semantic model to incorporate analogue input/output such as variable finger pressure. The latter connects back to earlier streams of work on status–event analysis.

A. Dix (✉)
School of Computer Science, University of Birmingham, Birmingham, UK
e-mail: alan@hcibook.com

A. Dix
Talis Ltd. Birmingham, Birmingham, UK

M. Ghazali
University of Technology Malaysia, Johor, Malaysia
e-mail: masitah@utm.my

B. Weyers et al. (eds.), *The Handbook of Formal Methods in Human-Computer Interaction*, Human-Computer Interaction Series,
DOI 10.1007/978-3-319-51838-1_9

9.1 Introduction

In industrial control rooms, in our living rooms, and in our pockets, the devices that surround us combine physical controls with digital functionality. The use of any device, including its safety, usability and user experience, involves the conjoint behaviour of its physical and digital aspects; however, this is often complex.

The digital side of this has been the subject of extensive research in the formal user interface modelling community. However, there has been very little work that addresses the physical interaction, and this chapter addresses that gap. We present physigrams, a semi-formal modelling notation for describing the physical interactions of devices and how this links to underlying digital behaviour.

The chapter begins by describing the multiple feedback loops between users and systems, highlighting the physical and digital channels and the different effects on the user with reference to the case studies and other examples. It will then demonstrate physigrams using a small number of increasingly complex examples, largely of domestic appliances for which the methods were first developed.

Having laid the groundwork, we then look at a more practical use where product designers used the notation to describe three potential options for a media-player. This exposes some of the advantages of a semi-formal notation, as the designers were able to extend the notation to deal with emerging issues in the design.

Finally, physigrams are used in a case study of a wind-turbine control panel. This explores the potential for use in real-world industrial control panels and some of the practical problems in studying detailed physical phenomena of critical equipment while in use.

Finally, the chapter will describe some of the open problems in using this kind of framework. This will include practical issues such as level of detail and times when it feels natural to let some of the digital state ‘bleed back’ into a physigram. It will also include theoretical issues, notably the problem of having a sufficiently rich semantic model to incorporate analogue input/output such as variable finger pressure. The latter will connect back to earlier streams of work on status–event analysis.

9.2 Physical and Digital Feedback Loops

The provision of feedback is a core feature in most lists of interaction design heuristics and guidelines. For example, in Shneiderman’s *Eight Golden Rules* number three is “offer informative feedback” (Shneiderman and Plaisant 2010), and Norman’s *Execution–Evaluation Cycle* is as much about interpreting feedback as it is about formulating and performing actions (Norman 1988).

For PC-based systems, the main feedback is visual, the effects of keystrokes and mouse/pointer movements on virtual screen objects. However, when we look at richer digital-physical systems, we find that there are multiple feedback pathways, from the look, sound and feel of the physical controls themselves, to digital displays or the effect of computation on physical actuators such as a washing machine or nuclear power station.

Figure 9.1 looks at a number of different kinds of feedback using the paths of action and communication between the user's body, an input device (e.g. a control knob, or button) and the underlying digital system. The various paths are described in detail elsewhere (Dix et al. 2009), so here we will just work through a few examples.

First consider a public display in a café. The café owner is about to open and so presses the power switch for the display (physical manipulation (a)). The switch moves and the café owner can both feel and see the switch is now on (a directly perceived state of the control switch (b) leading to feedback loop A). In addition, after a few seconds of boot-up, the public display system shows a message (electronic feedback (d), loop C).

As another example, the operator of a nuclear power station wishes to reduce the reactor power output to tick-over level ready for maintenance. The operator twists a 'soft' dial (a). The dial is fitted with a haptic feedback system: when the system detects the dial movement, it produces a small resistance to any movement (physical effects in the device (ii)), but has much stiffer resistance at the point at which the reactor would shut down completely. Because the operator is able to feel this (digital feedback similar to physical feedback (c), feedback loop B), it is easy to move the dial to the minimum point without danger of over-shooting the critical level. In addition, actuators within the reactor begin to lower the control rods (physical effects on the environment (iv), feedback loop D).

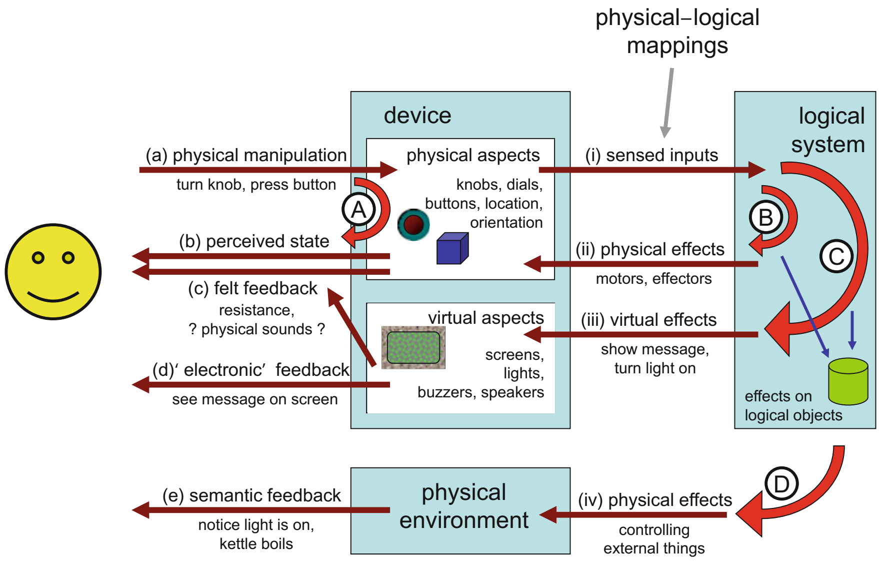

physical–logical mappings
device
logical system
(a) physical manipulation
turn knob, press button
physical aspects
knobs, dials, buttons, location, orientation
(i) sensed inputs
A
B
C
(b) perceived state
(c) felt feedback
resistance, ? physical sounds ?
(ii) physical effects
motors, effectors
virtual aspects
screens, lights, buzzers, speakers
(iii) virtual effects
show message, turn light on
(d)′ electronic′ feedback
see message on screen
effects on logical objects
D
(e) semantic feedback
notice light is on, kettle boils
physical environment
(iv) physical effects
controlling external things

Fig. 9.1 Multiple feedback loops (from Dix et al. 2009)

While loop D is very clearly of a different nature, the distinctions between loops A, B and C are a little more subtle. From a *system* point of view, loops B and C are effectively identical and would appear so in most formal modelling notations. However, *for the user* the opposite is the case, very well designed and executed haptic feedback (loop B) might seem just as if it were direct physical resistance in a device (loop A). Indeed, if the implementation really is perfect, then there is no difference for the user. However, any slight or occasional imperfections can be critical; for example, we describe elsewhere how delays of only a fraction of a second in simulated key-clicks made it almost impossible to triple tap on a mobile phone (Dix et al. 2009).

All of these loops are important; however in this chapter we will focus principally on loop A, the immediately perceived effects of the physical manipulation of the device. That is the things you feel whether or not the digital aspect of the system is actually working. We have called this analysing the 'device unplugged' (Dix et al. 2009).

9.3 The Device Unplugged

Imagine visiting a nuclear power plant that is being decommissioned. All of the wiring and electronics have been removed, but the shell of the control room is still there. You can twist dials, press buttons, pull levers, but nothing does anything. To take a more day-to-day example, it is common to see a small child playing with their parent's phone. If the power has run out (maybe because of the child's playing), would there be things the child could still play with, such as buttons to press?

The idea of considering the device unplugged is to foreground the interaction potential of the physical controls themselves.

Affordances are an important concept in HCI. In Gibson's original definition, "*the affordances of the environment are what it offers the animal, what it provides and furnishes, either for good or for ill*" (Gibson 1979, p. 127). Following adoption in the HCI community, led by Norman and Gaver (Gaver 1991; Norman 1988, 1999), we often think about the apparent affordances of on-screen controls; however, Gibson was focused on more basic physical interactions: a rock affords picking up and throwing if it is of a suitable weight and size for the hand.

Looking at a light-switch, we can consider the whole lighting system of the house and say that the switch affords turning on the light, but before that the fact that the switch looks and feels switch-like means it affords pressing. While later writers have considered other aspects, such as cultural knowledge, when considering simple physical actions Gibson claimed that affordances were immediately perceived, in the sense that our whole perceptual systems are tuned to see the interaction potential of the world. The argument for this is that as a species we are fitted for our environment and optimised for action. Of course, for complex digital systems there is no guarantee that this will be the case.

Many of the problems with pervasive technologies happen where this more immediate affordance is missing. A common example is the water for handbasins in public toilets. The water is often turned on by a proximity sensor, but there is little to indicate this theoretical affordance except the knowledge that something must make the water go on (cultural knowledge). If you imagine the bathroom 'unplumbed', with the water turned off and the pipes empty, a traditional water tap (faucet) can still be twisted, but the sensor-based one has no physical interaction potential at all.

Of course, the complete system is not just the physical controls, we also have to think about the mapping between physical actions and digital effects; even if we know that a switch can be pressed, what happens when it is pressed? However, by initially focusing on the device unplugged we both redress the more abstract action- and functionality-oriented descriptions that are common, and also lay proper emphasis on the first point of contact between human and machine.

9.4 Modelling the Device Unplugged

The vast majority of more formal modelling of interaction is focused on the dialogue level of the Seeheim model (Pfaff and ten Hagen 1985). Many techniques are used, from state transition networks (STNs) to Petri nets, but almost all assume user inputs at the level of abstracted actions, such as 'quit button pressed', rather than the actual physical action of pressing the button.

There are occasional descriptions of issues closer to the physical interface. For example, in even the earliest editions of the Human–Computer Interaction textbook (Dix et al. 2004), a case study is described where a change in the keyboard layout led to errors as it became possible to accidentally press adjacent keys forming an important and critical key sequence (Fig. 9.2).

Thimbleby has also looked at physical placement of buttons on a fascia and analysed how this might affect relevant time costs of button press sequences due to Fitts' Law delays for longer movements (Thimbleby 2007). Also, safety cases and accident reports include all levels of interaction: for example, the placement of a label at Three Mile Island hid an important indicator.

As a way to model the physical aspects of the device unplugged, we use *physigrams*, a variation of *state-transition networks* (STNs) with additional features

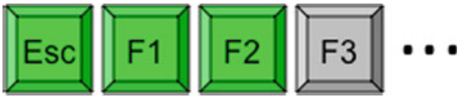


Fig. 9.2 Layout matters (from Dix et al. 2004). The key sequence F1–F2 meant "exit and save if modified", whereas F1–esc–F2 meant "exit without saving". This was fine on keyboards where the function and escape keys were separated, but on the above layout led to accidentally pressing the escape when trying to hit F1–F2

to deal with particular kinds of physical interactions (for example, buttons that 'bounce back' after being pressed). The name 'physigram' was coined by product designers who used the notation to model prototype devices. Crucially the STN-based notation is precise enough to give some analytic traction, whilst readable enough for product designers to use, even though they would normally avoid more formal descriptions.

9.5 Physigrams—Modelling Physical States

Figure 9.3 shows the simplest example of a physigram, the states of a simple on/off switch. It has two states, and the transition between the two is controlled solely by the user.

Of course, just looking at the switch 'unplugged', you cannot tell what it does electrically:

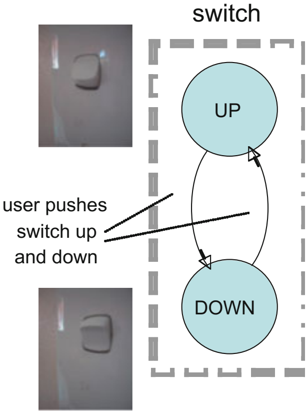
Is it a light switch, or controlling a cooling fan?

If it is a light switch, which light?

However, when you press the switch you can immediately see and feel that you have changed something, even if the ultimate effect is not clear (it may be a fluorescent light that takes a few seconds to warm up, or even an outside light that is not visible from where you are).

Not all switches are like this; Fig. 9.4 shows an example of a *bounce-back button*. This is the kind of button you quite frequently find on computers; you press it and, as soon as you release it, the button immediately bounces back to its original position. Although, like a click switch, it has two main states, there is only one *stable state*, the other is a transient *tension state* that requires constant pressure to be applied.

Fig. 9.3 Physigram states of a switch



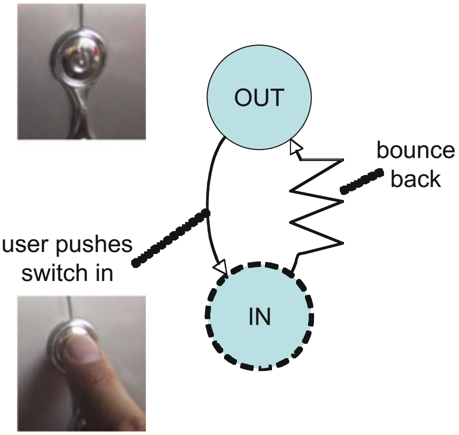


Fig. 9.4 Physigram of a bounce-back button

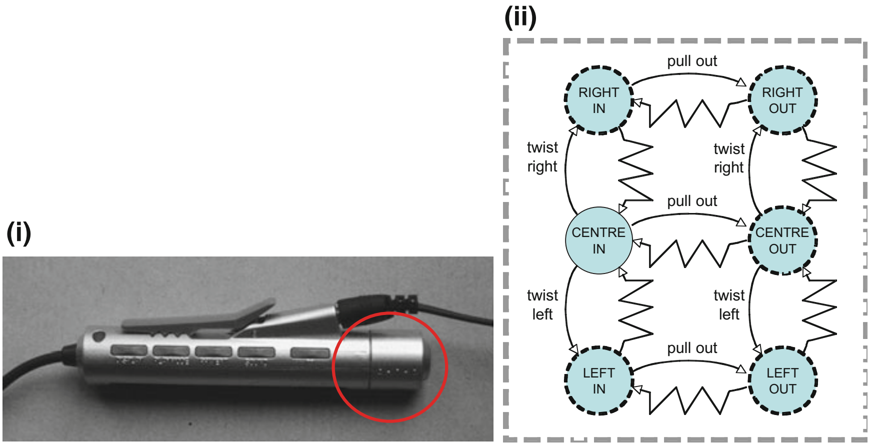


Fig. 9.5 **i** Minidisk controller, **ii** physigram

The physigram denotes this bounce-back using the jagged spring-like arc; whilst the solid transition is triggered by the user's action, the bounce-back is physically autonomous, usually just a spring inside.

These simple transitions can be used to represent more complex controls. Figure 9.5 shows the physigram of an old minidisk controller. The knob at the end can be pulled outwards, but once released it snaps back to the 'IN' state (a *bounce-back*). This is shown in the middle of the physigram (CENTRE IN <-> CENTRE OUT); the pull-out transition is controlled by the user, but it has a bounce-back transition back in. This centre part (turned on its side) is exactly the same as Fig. 9.4, except it is a pull rather than push action for the user.

However, the minidisk controller uses this pull not to trigger some action, but effectively as a mode control and the knob can be twisted as well. Twisting the knob when ‘IN’ changes the volume, but while pulled out skips forward and back through tracks.

While the effect of the twisting is not part of the physical ‘unplugged’ state, the twists themselves are modelled in the full physigram (Fig. 9.5ii). Note that the left and right twists are themselves bounce-backs; the only stable state is when the knob is in the ‘IN’ position and centred; all other states are *tension states* requiring some continued physical force.

As with many formal notations, there is some flexibility in terms of the level of detail you wish to capture with a physigram. Figure 9.3 showed a simple switch, but if you press a switch just lightly it ‘gives’ a little, yet does not snap to the other position until you are more than half way. That is, as well as the up and down states, there are many part-up and part-down states in between.

Figure 9.6i shows this more detailed view of the physical states and transitions including the spring-like bounce-back and the ‘give’ as a lightning-like transition. Note the give is similar to bounce-back as the physical properties create the transition, but whilst the bounce-back only happens when tension is released, the ‘give’ works with the user’s physical force.

Whilst this clearly provides more details about the physical interaction potential than Fig. 9.3, arguably it could be far more detailed still as there is not a single ‘part down’ state, but one where it is down just a little, then down a little more. The choice of level will depend on the issues being analysed. In our previous work on this issue, we suggested that a more detailed status–status mapping view of this

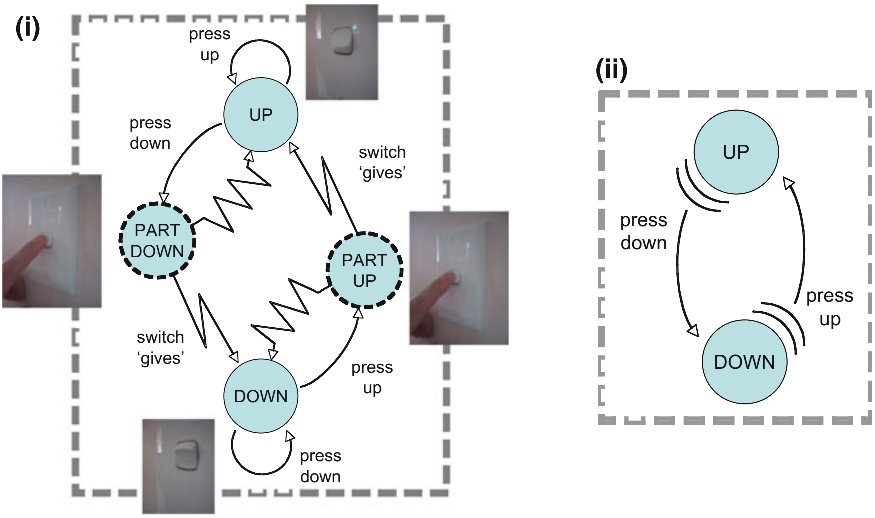


Fig. 9.6 Switch with ‘give’ **i** detailed physigram, **ii** ‘shorthand’ physigram with decorated transition

would be valuable, and since then Zhou et al. (2014) have produced analyses of button pressing using force-displacement graphs and taking into account dynamics such as weight and inertia. One could imagine being able to 'zoom' into the transitions in Fig. 9.6ii and see such a graph for the particular choice of button, or even annotating the transitions in some way to distinguish different levels of resistance or dynamics where this is relevant.

As this slight 'bounce-back then give' is a common phenomenon, a shorthand version is introduced in Fig. 9.6ii. This has much of the clarity of Fig. 9.3, but the transition is annotated to show that it has the initial resistance.

9.6 Plugging in—Mappings to Digital State

While we have emphasised the importance of describing the device unplugged, this is of course only part of the story. In the end we are interested in the whole system, physical and digital. To do this we also model the logical states of the digital or electronic side of the system, and the relationship between these and the physigram. We will use STNs to model the logical state also, but one of the many dialogue or system modelling notations could be used for this.

The idea of natural 'mappings' between controls and effects has been a core concept in interaction design since very early days (Norman 1988). There are many different kinds of mapping; in some cases if the controlled thing is spatial and the controls are laid out spatially we might look for spatial correlation: the light on the right is controlled by the switch on the right; or proximity: the switch to turn on the light is the one closest to it.

The relationship between digital states and physigram states is more about the dynamics of interaction.

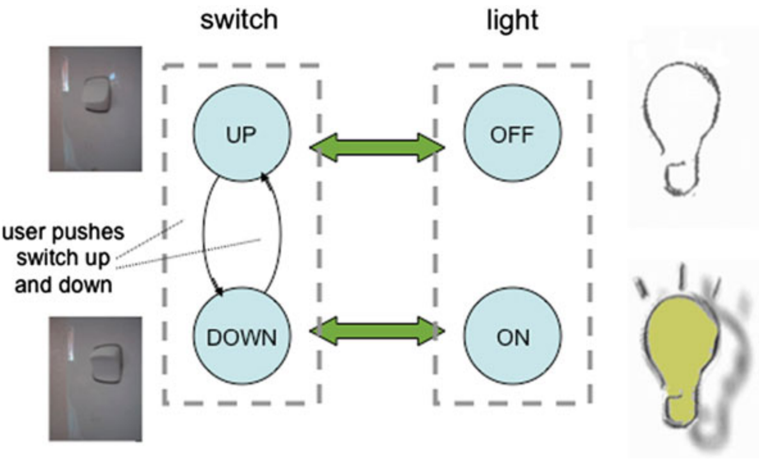


Fig. 9.7 Logical states of an electric light map 1–1 with physigram states

Figure 9.7 shows the physigram of a light switch on the left, and on the right the logical states of the light, whether it is on or off. Strictly there are probably additional states, when the light bulb is defective, there is a power cut, etc., but we are simply modelling the common illumination states.

In this case, the relationship between the physigram states and logical states is 1–1, but of course this is not always the case. Figure 9.8 shows a bounce-back switch. Except maybe for a signalling flashlight, it would be rare to find a bounce-back switch in 1–1 correspondence with the logical state. Most often, a user-determined transition in the physigram triggers a transition in the logical system.

In this example, we have labelled the physigram transition as an abstract action (a) and then shown on the logical system where this abstract action causes transitions. While state–state mappings are easy to represent diagrammatically, transition–transition mappings are harder to portray, especially as they may be many-to-many (as in Fig. 9.8). The use of named actions to link STNs is similar to the technique used in statecharts (Harel 1987), which, due to their adoption in UML, are likely to be reasonably familiar to many in computing.

In the case in Fig. 9.8, we have shown the bounce-back switch as a press on/press off button. In fact this is more likely for a light control, since powering a computer down may cause damage to the system. Quite commonly, the bounce-back switch is used to turn on a computer, but the 'off' is soft, determined by the computer after a software shutdown.

Figure 9.9 shows this alternative. Note how the physigram of the physical control is the same—you cannot tell by playing with the switch alone, when unplugged from the power supply, what action it will have. However, the mapping between physical and logical states allows us to model the different behaviour.

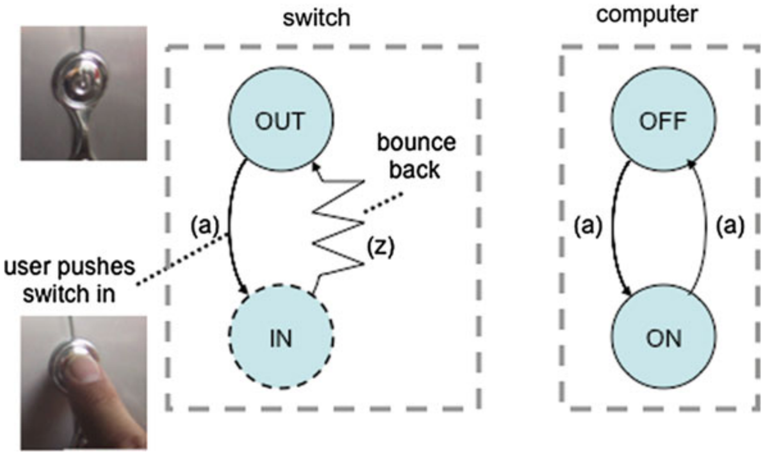


Fig. 9.8 Physical and logical states of a bounce-back switch

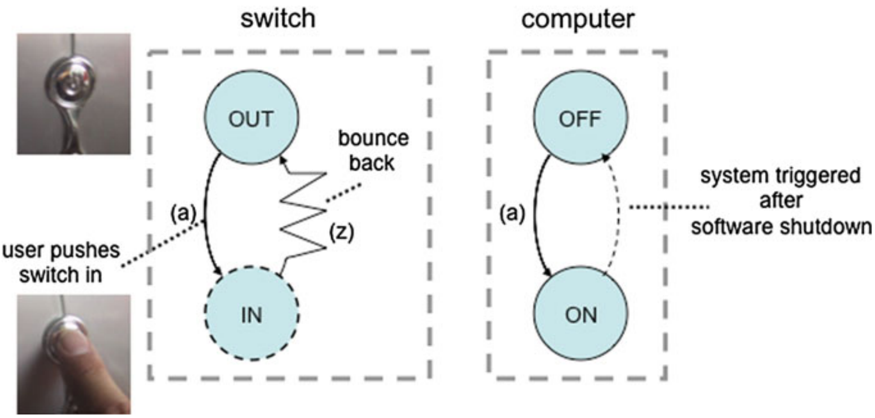


Fig. 9.9 Bounce-back switch only controlling one transition direction

9.7 Properties of Physical Interactions

We are now in a position to talk about some of the design properties of physical–digital devices.

First of all, consider again the *1–1 mapping* between device state and logical state for the light switch in Fig. 9.7. This is an example of *exposed state*; that is where the logical system state can be apprehended directly from some aspect (visible, tactile) of the device. Note this does not necessarily mean that the user can know what that logical state is (for example, whether the switch controls a light out of sight or is in fact turning on the nuclear emergency alarm). However, once this is known, it means that the system state is instantly observable.

The opposite of exposed state is *hidden state*: for example, in Fig. 9.8, the switch does not expose the state of the computer system. Of course, you might see the screen flicker or hear the disk start to spin, that is, in terms of Fig. 9.1, feedback loops C or D. However, there is the danger that if there is a small delay you might think you have not pressed the button properly and press again, accidentally turning the computer back off in mid-boot-up.

Clearly exposed state is a very powerful property when it is possible, but it is not always achievable.

Part of the power of digital systems is that vast amounts of system state can be packaged into very small footprints; if every state of a typical phone were to be made physically accessible in terms of dials, knobs, or switches, the control panel would probably stretch over an area the size of a city.

Bounce-back buttons are most useful where there is a large or unbounded number of logical states, for example the number of tracks on a music player. However, here, as the physical device feedback loop (loop A) is weaker, it is important that there is additional logical feedback. In the case of the minidisk player, increasing/decreasing the volume will be apparent—except when it happens to be during a quiet point. For example, “Funeral for a Friend”, the first track of Elton John’s album “Goodbye Yellow Brick Road”, starts very quietly, with the distant sound of an owl hoot; if you adjust the volume at this point you are likely to be deafened moments later. Some sound controls deliberately make a small sound immediately after the level is set, to help deal with this problem.

Another property, which is evident in the minidisk player, is the *natural inverse.* The volume up is achieved by twisting the knob in one direction and the opposite twist turns it down again.

This property is very important in two ways:

- *Automatic correction*—if you are trying to do something and ‘overshoot’, you automatically do the physically ‘opposite’ action. If this does not map to the logically opposite action you are likely to have problems.
- *Discoverability*—if you know that a pair of buttons always behave as natural inverses, then even if you do not know precisely what they do, you can feel free to experiment knowing you can correct the effect.

One of the authors used to have a phone with an up/down slider button. Sometimes it changed menu selections, and sometimes it controlled the volume. The author never learnt the precise rules for how this changed in each mode, but was still able to use it knowing that it was ‘safe’ to experiment.

Indeed in experiments (Ghazali 2007; Ghazali et al. 2015), we found that if we had completely arbitrary cognitive mappings (that is the user had no idea which controls affected which system property), but a good natural inverse, this performed much better than when there was a good cognitive map (the user knew precisely the effects of any action), but a poor natural inverse. In the first situation, users had to experiment slightly to work out which control to use, but once they did were able to complete tasks. In the second situation, things were fine initially as they knew which control to use, but if they overshot everything fell to pieces!

Give is also very important for discoverability, even to work out what physical manipulation is possible. Most lever-style door handles push down to open. However, when they do the opposite, that is you have to lift the handle to open, few people have serious problems. This is because having pushed slightly down and felt no ‘give’, the user is likely to automatically try to pull up slightly. The slight give tells you which directions are possible manipulations. This is a form of low-level and instinctive epistemic action (things you do in order to discover information).

In Fig. 9.9, we saw an example of the logical state changing autonomously. This is of course normal for computer systems! Less common in computers, but more so in domestic appliances, is when the system makes some change to the state of the controls themselves.

Electric kettles are a good example of this: when the water boils, the switch flicks to the off position.

Figure 9.10 shows a typical electric kettle. On the left is the physigram (ignore for a moment the dotted line). The user can press the switch down or up. There is an

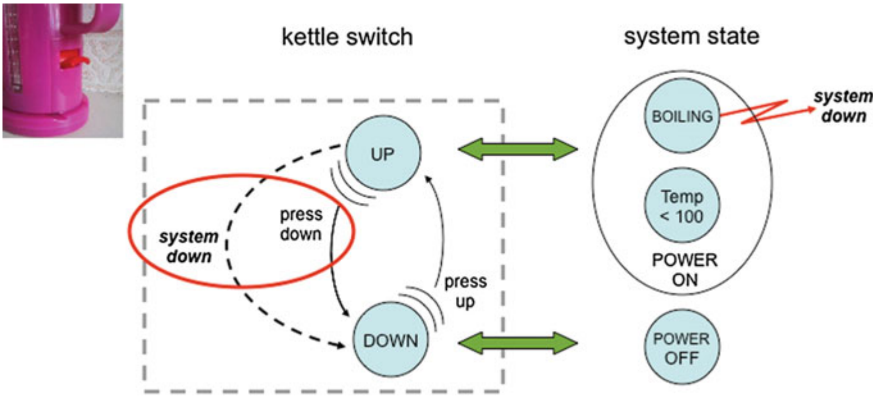


Fig. 9.10 Compliant interaction—matching system and user control

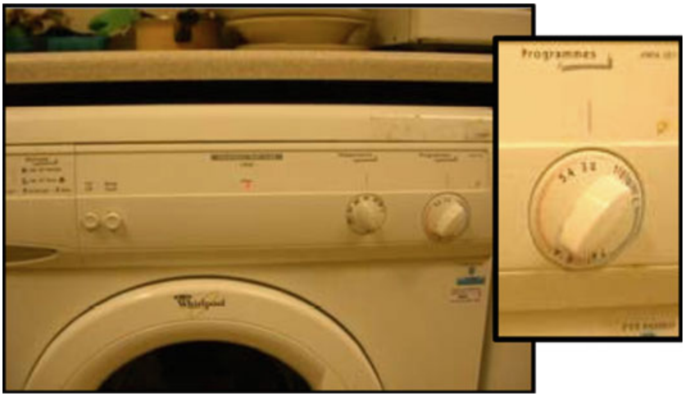


Fig. 9.11 Compliant interaction—the washing machine dial

exposed state relationship between the kettle switch and the power going to the kettle. In this case when the switch is up the kettle power is on, when it is down the power is off.

However, when the power is on and the water is boiling the sensor triggers the switch to flick back down into the power-off state. That is the kettle achieves both system control of the power and *exposed state*.

Not only that, but the way this is achieved means that if you, as the user, want to turn the kettle off, you do exactly the same action as the system does: a property we call *compliant interaction*.

Typically, when system and user control are aligned like this, it is easier to learn how to manipulate potentially complex system states. A good example of this is the traditional washing machine dial (Fig. 9.11). The user sets the desired programme using the dial, and then as the washing machine moves through its cycle it moves the dial. The dial therefore becomes an indicator of progress and also a means for the user to make tweaks, perhaps cutting a stage short. Sadly, this fluid expert behaviour has become more difficult and less common as washing machine control panels have become more digital.

On the other hand, where these properties fail we often see problems. Many electric toasters allow you to push down the handle to let the bread into the heat, but pop the toast up automatically. If you want to stop the toast early, there is often a special button. However, if in a hurry or flustered (maybe when the smoke comes out the top!), it is easy to forget and people often try to lift the toaster handle manually, leading to a few moments' struggle before the frantic search for the stop button. Other toasters effectively make lifting the handle serve this purpose, both a *natural inverse* to pushing it down and also *compliant interaction*: the automatic way to stop toasting and the manual way both involve the same control change.

9.8 Flexibility and Formality

Physigrams are a semi-formal notation. The STN core is amenable to analysis. However, in practice this is usually trivial; it is an STN of the physical device controls and manipulations: physical interactions between controls, and the combinatorial explosion this often causes, are rare. The real power of the physigram lies less in these aspects than in the differing styles of transitions, which enable subtle distinctions in behaviour to be expressed, for example, Figs. 9.3 versus 9.4 and 9.8 versus 9.9.

This more communicative power of physigrams was demonstrated very clearly when product designers used them as part of a design exercise for a media controller. They compared three different designs for a dial: two were physical movable dials, each with slightly different tactile properties, and one was a smooth touchpad-style dial.

Figure 9.12 shows the three physigrams produced by the designers, who had been shown examples of physigrams, but were not aided by the formal team. In some ways, rather like the comparison between Figs. 9.3 and 9.4, these are virtually identical, simply a number of control states that the control cycles between. However, when examined in detail, there are subtle differences: some controls allow full 360° movement (ii and iii), while one has a 'stop' (i), so that it is logically a linear control.

+Crucially, the designers also augmented the notation. One of the dials (Fig. 9.12i, close-up Fig. 9.13i) had hard selections, it clicked from one to another with a small amount of bounce-back when it was in one of them; another (Figs. 9.12ii and 9.13ii) moved fluidly but with tangible feel as the user went over the critical transitions; whilst for the third (Figs. 9.12iii and 9.13iii) there was only virtual feedback.

Strictly, the numbered states in (iii) are logical states only; the physigram of the physical interactions alone would only include the ability to move one's finger

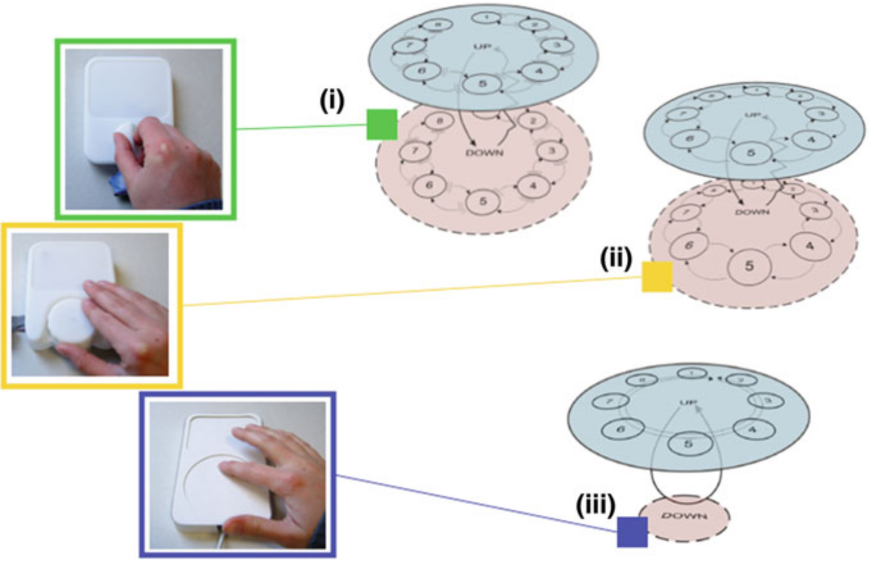


Fig. 9.12 Product designers' use of physigrams

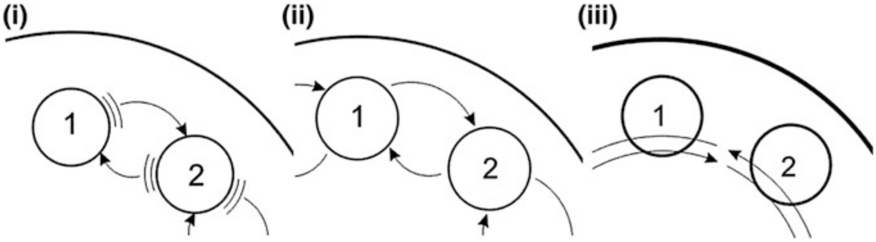


Fig. 9.13 Detail of transitions in Fig. 9.12

smoothly round the pad and press it down. However, neither did the numbered states denote the full controlled digital state, which might represent different menu selections depending on mode; instead, they were somewhere in between, capturing an early design decision that the device would always allow precisely 8 selections, but not the exact binding of these. Note that the line is doubled in Fig. 9.13iii, as the finger can slide over the touchpad in either direction.

This sort of touch-only device is common now with trackpads and smartphone displays. For these devices it is not that there is no tactile feedback, you can feel that you are in contact with the pad, and you can feel the movement through both touch and proprioceptive senses. The control in Fig. 9.12iii is a three-state device in Buxton's three-state model (Buxton 1990):

- State 0 when the finger is not in contact with the touchpad

- State 1 when the finger is in contact and dragging across the surface
- State 2 when the touchpad is pressed down

Figure 9.12iii shows a state 1–2 transition between the UP and DOWN states. In Fig. 9.12i, ii this is a bounce-back transition as the knobs noticeably press down and then click back into place when released (strictly this is a shorthand for lots of bounce-back transitions between the corresponding states as the dial stays in the same orientation when it bounces back). In contrast, the touchpad has a 'press to select' action, but it is not tangible except for the internal feeling of pressing something. The designers denoted this by showing the DOWN state as transient (there is a device 'down'), but with a loop transition drawn going from the UP state *through* the DOWN state and back to the UP state. This denotes, as in Fig. 9.13iii, that the user is not perceptually aware of the device state change.

Although there is tactile feedback through the internal sense of pressure for the touch device, the relation between the felt feedback and the system detection of a touch or movement is less clear than with the physical buttons.

It may be that felt touches are not registered by the system. For example, one of the authors has a TV with on-screen touch buttons; it can take many touches before it registers and turns on. This may be because he misses the 'hot spot' of the button or because the system 'de-bounces' the finger press, treating it as electrical noise.

Alternatively, it may be that the system registers events without the user being aware of it. For example, one of the authors cannot use a laptop with 'tap to select' enabled; when he types, his thumb occasionally makes contact with the touchpad, and although this is too light to feel it registers as a 'select' and the typing appears in apparently random parts of the screen.

For some purposes, the discrete version of the three-state model might be sufficient, but a real physical model to deal with touch would be even more difficult than those discussed for 'give' in Sect. 9.6. We will not attempt to deal with these issues here, but the physigram does act as a communication point with the potential, on the one hand, to drill down into the minutiae of body contact and pressure, and, on the other, to connect to system behaviour.

Note also that the designers drew the states in a form that visually matched the physical shape of the controller. The fact that the physical location of states in an STN does not have a formal interpretation left it open for the designers to create a meaning for this. This *openness to interpretation* is one of the design principles for appropriation identified elsewhere (Dix 2007) and effectively allows secondary notation (annotations that can be added which do not have formal meaning within the notation), which has been identified as an important feature in the study of cognitive dimensions of notations (Green and Petri 1996).

This openness of the notation is essential for physical devices because the range of possible interactions is wider than for screen-based controls. When using semi-formal notation for communication within a design team, it is more important that the meaning is clear to those involved than that they shoe-horn the behaviour into pre-defined but inappropriate categories. This also feeds back into more formal versions of the notation, as it highlights gaps (e.g. the 'half way logical' states).

On the other hand, if we wish to perform more formal analyses, some aspects have to be more strictly defined. In previous work, we have used a more precise meta-model to define the semantics of the various physigram primitives (Dix et al. 2009). A discrete meta-model was possible when describing the physical state physigrams (effectively an STN with coloured states and transitions). However, unsurprisingly, we found limitations when describing the more continuous interactions; a point encountered by the first author previously in studying status–event analysis (Dix 1991; Dix and Abowd 1996) and by others (Massink et al. 1999; Wüthrich 1999; Willans and Harrison 2000; Smith 2006). Ideally, we would also like to be able to model human aspects such as the level of pressure applied, and felt, as a finger presses a switch. We should be able to describe Fig. 9.6i, ii in such a way that we can verifiably say that one is syntactic sugar for the other. The only work that comes close to this is Eslambolchilar's (2006) work on cybernetic modelling of human–device interactions.

9.9 Case Study—Tilley, a Community Wind Turbine

We will look at the control panels described in the community wind turbine case study in Chap. 4, although the other book case studies would also include similar panels. The particular wind turbine, Tilley, is a land-based one, on the island of Tiree (TREL 2016), where, due to the windswept environment, it is one of the most efficient turbines of its type in the world.

Recall there are two control panels described in detail in Chap. 4 and reproduced in Fig. 9.14.

The 'digital' panel in Fig. 9.14a is mostly dedicated to outputs except for the generic function keys and numeric keypad in the middle. This will be used for the most complex, but not the most time critical, interactions. The individual membrane buttons have some *give*, and are each *bounce-back* as in Fig. 9.6. As they are for

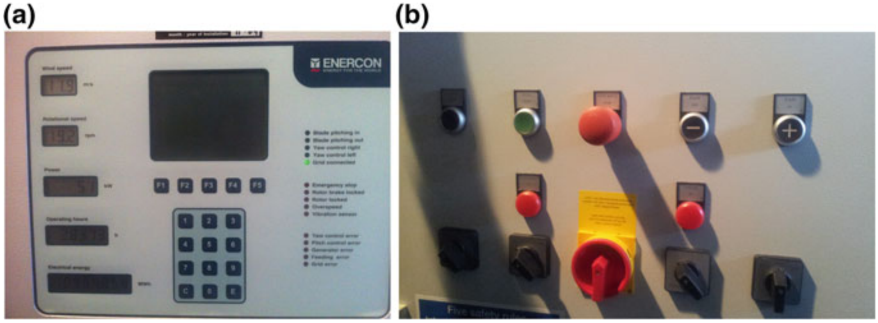


Fig. 9.14 **a** Digital display and control panel in Tilley (photo © William Simm), **b** physical control panel in Tilley (photo © Maria Angela Ferrario)

generic input this is a reasonable choice, so there is little more that physigrams can say about them. However, alternative analyses would be useful, for example the techniques applied in the CHI-MED project to medical numeric input devices (Cauchi et al. 2014). We should note also that the numeric keypad somewhat oddly adopts a phone-style key order (with 1 at the top) rather than the order found in calculators or computer numeric keypads (with 1 at the bottom).

The panel in Fig. 9.14b is more interesting, with a combination of press switches and twist knobs of various kinds. Of course it is not possible to 'unplug' Tilley to experiment in the way described for domestic devices earlier in this chapter; neither is it possible to experiment with the live panel to get the 'feel' of the buttons, so the analysis here is inevitably limited. However, during the design of an interface such as this, detailed physical examination and specification would be possible.

Note that this is an example that could benefit from the kinds of pressure annotations discussed towards the end of Sect. 9.6. The emergency stop button (large red button, top centre in Fig. 9.14b) needs to be firm enough not to be pressed accidentally, but also responsive enough that it can be pressed quickly and that you know you have pressed it successfully from its physical response. Note too that this is large, as it, especially, needs to be operated easily even if the operator is wearing gloves. This button is a *bounce-back button* and this is appropriate from a system control point of view as the restart is expected to be a complex process, not simply pulling the button out again (see Fig. 9.15). However, as a *hidden state* control it does not have feedback of type A (Fig. 9.1). Instead, the operator would either need to look at numeric outputs on the digital display panel or screen (feedback loop C), or more likely simply be aware of the physical effect (loop D) as the system shuts down.

The reset button (green to the left of the emergency stop button in Fig. 9.14b) is also a bounce-back, although of a slightly different design as it does not need to be

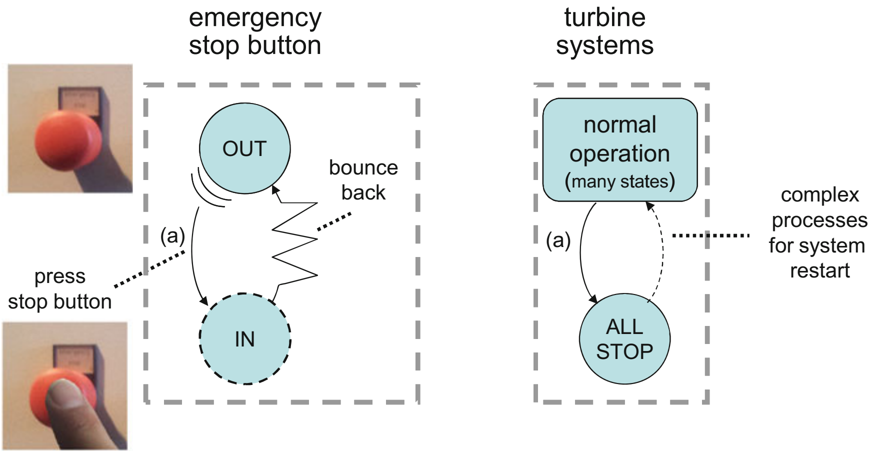


Fig. 9.15 Emergency stop button (*left*) physigram (*right*) system state

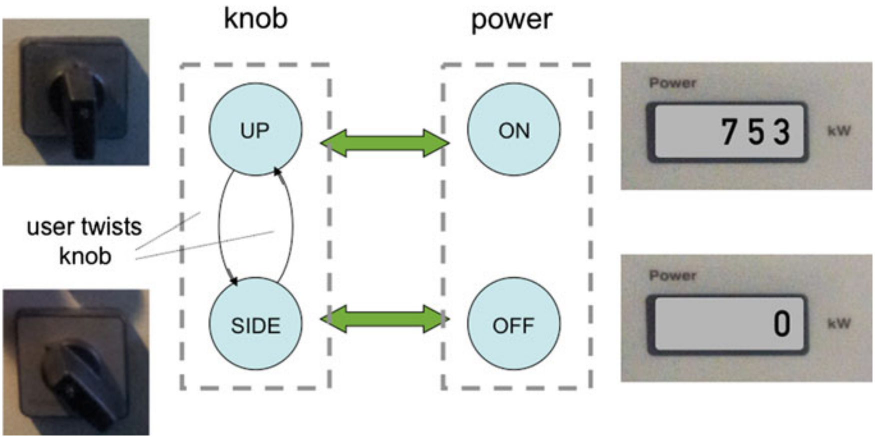


Fig. 9.16 Power isolation knobs (*left*) physigram (*right*) system state

hit quickly like the emergency stop button. It too has hidden state, presumably resetting various electrical systems. The effects of this may be less obvious than the emergency stop and might require explicit checking of the digital display. However, as it is a reset, it is *idempotent*, meaning that if the operator is at all uncertain whether it was pressed, they can simply press it again.

Below the emergency stop button is a large power control knob flanked by several small black knobs to control specific circuits. These are all visible state controls with one-to-one mappings between physical state and controlled system state (see Fig. 9.16). The settings of these are critical: if the engineer wrongly thinks a circuit is off and touches it, they may be injured or killed. From the image we can immediately see that the main control is switched on, as are three of the sub-circuits, but the one on the right is off.

At the top right are two buttons labelled '+' and '–'. These control the angle of attack of the turbine blades. These are each *bounce-back buttons* and serve to increase or decrease the current blade angle (see Fig. 9.17). These are *hidden state* controls and the engineer would either need to go outside to look at the blades (loop D), or more likely simply observe the impact of the change in angle on power output.

One could imagine an alternative control that used a dial to adjust the blade angle. This would lead to a visible state and also be faster to set a particular angle. However, the speed is probably immaterial; the engineer has taken a three-day trip to come to the island, a few seconds pressing ± buttons is not going to make much difference! Also, CHI-MED has shown that often this form of increment/decrement setting is safer as it makes it harder to perform gross errors (Oladimeji et al. 2011).

However, the positioning of the buttons is not optimal. They are clearly placed in a neat grid, but this means there is nothing to suggest physically or visibly that the buttons are linked. Unlike the minidisk volume controls, this does not form a *natural inverse*.

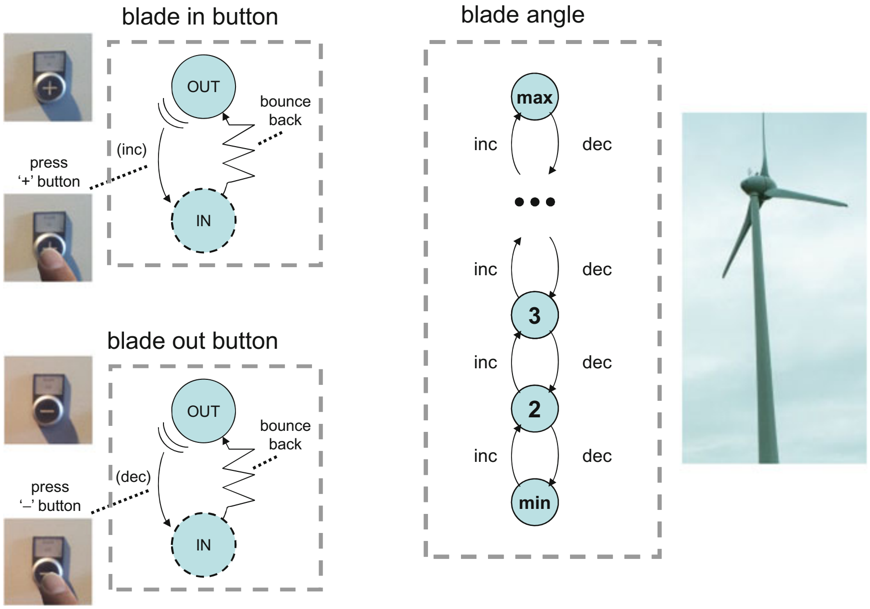


Fig. 9.17 Blade angle control (*left*) physigram (*right*) system state

9.10 Conclusions

We have seen how physigrams allow us:

(i) to describe the behaviour of the device 'unplugged', the purely physical interaction potential of a device, enabling the exploration of sensory-motor affordances of the physical device independent of how it is connected into the wider system.
(ii) to link these subsequently to digital states, exposing a variety of properties of physical–digital hybrid systems, related to but going beyond conventional discussions of representational 'mapping'.

The first of these often exposes subtle differences between controls that otherwise appear superficially similar. The second both allows specific issues to be identified and also offers the potential to create generic advice on matching physical controls and digital state.

In addition to exposing these generic properties, we have seen how physigrams can be used by product designers to describe aspects of their physical designs that would otherwise only be apparent during physical exploration. This can both help their own design activity and also allow clearer discussions with developers or more formal analysis.

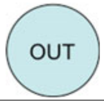
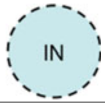
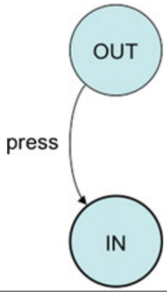
The semi-formal nature of physigrams means that we can specify precise formal semantics for a core set of primitives, whilst still allowing an openness to extend and augment the notation for new circumstances. Marrying these two, flexibility for extension and formal tractability, is still an open issue, not just for physigrams, but for any formal notation.

Both the informal and formal uses of physigrams raise again the still open issue of how to create comprehensible and tractable models of 'hybrid' interactive systems that combine both continuous (status) and discrete (event) behaviours. In particular, these would ideally also be able to talk about the physical actions and sensations of the user as well as more common mental states.

Finally, physigrams were used to explore the specification of the control panel of a medium sized wind turbine. There were limitations to this as the system could not be brought out of operation simply for the purposes of the study; so some aspects of the behaviour had to be guessed. However, the case study gives some idea of how the methods could be applied during the design stages of this kind of industrial control panel.

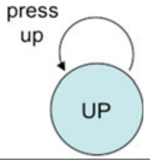
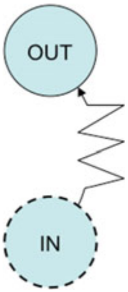
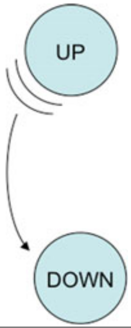
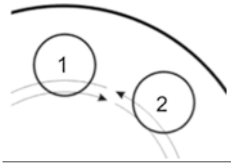
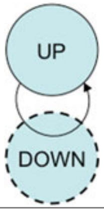
9.11 Key to Notation

Physigrams

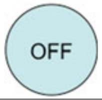
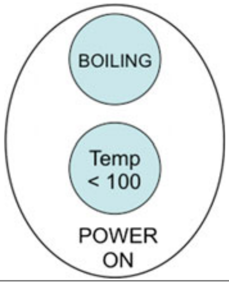
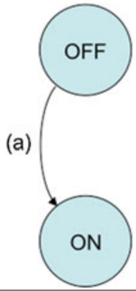
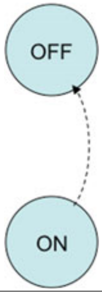
Symbol	Meaning	Used in Figs
	**state**—physical state of the device	3–10, 12, 13, 15–17
	**transient tension state**—physical state which can only be maintained by some sort of continuous user pressure or exertion	4–6, 9, 12, 15, 17
	**transition**—this may be labelled by the user action that causes this, or this may be implicit. It may also have a label to connect it with logical state (see linkage)	3–9, 12, 13, 16

(continued)

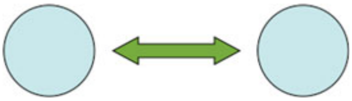
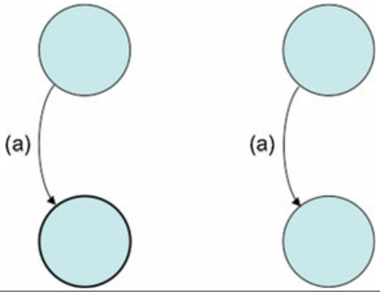
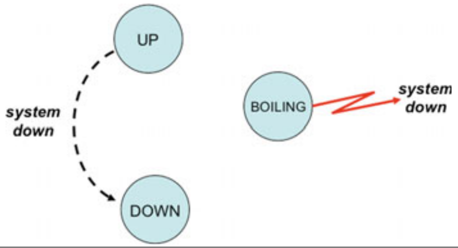
(continued)

Symbol	Meaning	Used in Figs
	**self transition**—special case of the above where a user action has no effect on the system (e.g. attempting to twist a dial beyond its limits)	6
	**bounce-back**—when the device spontaneously returns from a transient tension state to a stable state when user pressure is removed	4–6, 8, 9, 12, 15, 17
	**give**—where a button or other control moves slightly but with resistance before ‘giving’ and causing state change. If the user stops exerting pressure before the ‘give’ point, it will return to the initial state	6, 10, 12, 13, 15, 17
	**slide transition**—designer extension to denote situation when there are device changes that are not perceptible to the user	6, 10, 12, 13
	**unfelt bounce-back**—this is basically a press and bounce-back, but where there is no perceptual feedback	12

Logical State

Symbol	Meaning	Used in figs
	**state**—state of the logical system	7–10, 16, 17
	**group**—where several lower-level states can be thought of as sub-states of a more abstract state. In the example, ‘POWER ON’ is a higher-level state, but within this water may be below or at boiling point	10
	**user–initiated state transition**—logical state transition initiated by external user activity (see also linkage)	8, 9, 17
	**system-initiated transition**—where some internal computation, or environmental event not connected with the interacting user, causes a transition in the logical system	9

Linkage

Symbols physigram logical state	Meaning	Used in figs
	**state–state mapping** –where the physically visible or tangible states in an *exposed state* device correspond precisely to states in the underlying logical system	7, 10, 16
	**user-initiated transition**—where a state transition in the physical device caused by a user action gives rise to a transition in the logical state. The connection is denoted by the label, '(a)' in the example	8, 9
	**system–initiated transition**—where a spontaneous change or event in the system (denoted by the lightning symbol) triggers a physical state change in the device	10

Acknowledgements Early parts of this work were supported by the AHRC/EPSRC funded project DEPtH "Designing for Physicality" (http://www.physicality.org/). We are very grateful for insightful comments from the reviewers.

References

Buxton W (1990) A three-state model of graphical input. In: Proceedings of human–computer interaction—INTERACT'90. Elsevier, Amsterdam, pp 449–456

Cauchi A, Oladimeji P, Niezen G, Thimbleby H (2014) Triangulating empirical and analytic techniques for improving number entry user interfaces. In: Proceedings of the 2014 ACM SIGCHI symposium on engineering interactive computing systems (EICS '14), ACM, NY, USA, 243–252. doi:10.1145/2607023.2607025

Dix A (1991) Formal methods for interactive systems. Academic Press, New York. http://www.hiraeth.com/books/formal/

Dix A, Abowd G (1996) Modelling status and event behaviour of interactive systems. Softw Eng J 11(6):334–346. http://www.hcibook.com/alan/papers/SEJ96-s+e/

Dix A, Finlay J, Abowd G, Beale R (2004) Human–computer interaction, 3rd edn. Prentice Hall, Englewood Cliffs. http://www.hcibook.com/e3/

Dix A (2007) Designing for appropriation. In: Proceedings of BCS HCI 2007, People and computers XXI, vol 2. BCS eWiC. http://www.bcs.org/server.php?show=ConWebDoc.13347

Dix A, Ghazali M, Gill S, Hare J, Ramduny-Ellis S (2009) Physigrams: modelling devices for natural interaction. Formal Aspects Comput, Springer 21(6):613–641

Eslambolchilar P (2006) Making sense of interaction using a model-based approach. PhD thesis, Hamilton Institute, National University of Ireland, NUIM, Ireland

Gaver W (1991) Technology affordances. In: Proceedings of the SIGCHI conference on human factors in computing systems (CHI'91). ACM Press, New York, pp 79–84

Ghazali M (2007) Discovering physical visceral qualities for natural interaction. PhD thesis, Lancaster University, England, UK

Ghazali M, Dix A, Gilleade K (2015) The relationship of physicality and its underlying mapping. ARPN J Eng Appl Sci 10(23):18095–18103

Gibson J (1979) The ecological approach to visual perception. Houghton Mifflin Company, USA

Green T, Petri M (1996) Usability analysis of visual programming environments: a 'cognitive dimensions' framework. J Vis Languages Comput 7:131–174

Harel D (1987) Statecharts: a visual formalism for complex systems. Sci Comput Program 8 (3):231–274. doi:10.1016/0167-6423(87)90035-9

Massink M, Duke D, Smith S (1999) Towards hybrid interface specification for virtual environments. In: DSV-IS 1999 design, specification and verification of interactive systems, Springer, Berlin, pp 30–51

Norman D (1988) The psychology of everyday things. Basic Books, New York

Norman D (1999) Affordance, conventions, and design. Interactions, 6(3):38–43. ACM Press: NY

Oladimeji P, Thimbleby H, Cox A (2011) Number entry interfaces and their effects on error detection. In: IFIP conference on human-computer interaction, Springer, Heidelberg

Pfaff G, ten Hagen P (eds) (1985) Seeheim workshop on user interface management systems. Springer, Berlin

Shneiderman B, Plaisant C (2010) Designing the user interface: strategies for effective human-computer interaction, 5th edn. Addison-Wesley, MA

Smith S (2006) Exploring the specification of haptic interaction. In Interactive systems: design, specification and verification (DSVIS 2006). Lecture notes in computer science, vol 4323. Springer, Berlin, pp 171–184

Thimbleby H (2007) Using the fitts law with state transition systems to find optimal task timings. In: Proceedings of second international workshop on formal methods for interactive systems, FMIS2007. http://www.dcs.qmul.ac.uk/research/imc/hum/fmis2007/preproceedings/FMIS2007preproceedings.pdf

TREL (2016) Tilley, Our turbine, Tiree Renewable Energy Limited. http://www.tireerenewableenergy.co.uk/index.php/tilley-our-turbine/. Accessed 24 Mar 2016

Willans J, Harrison M (2000) Verifying the behaviour of virtual world objects. In: Proceedings of DSV-IS'2000. Springer, Berlin, pp 65–77

Wüthrich C (1999) An analysis and model of 3D interaction methods and devices for virtual reality. In: Proceedings of DSV-IS'99. Springer, Berlin, pp 18–29

Zhou W, Reisinger J, Peer A, Hirche S (2014) Interaction-based dynamic measurement of haptic characteristics of control elements. In: Auvray M, Duriez C (eds) Haptics: neuroscience, devices, modeling, and applications: 9th international conference, EuroHaptics 2014, Versailles, France, June 24–26, 2014, Proceedings, Part I, pp 177–184. Springer, Berlin. doi:10.1007/978-3-662-44193-0_23

Chapter 10
Formal Description of Adaptable Interactive Systems Based on Reconfigurable User Interface Models

Benjamin Weyers

Abstract This chapter presents an approach for the description and implementation of adaptable user interfaces based on reconfigurable formal user interface models. These models are (partially) defined as reference nets, a special type of Petri nets. The reconfiguration approach is based on category theory, specifically on the double pushout approach, a formalism for the rewriting of graphs. In contrast to the related single pushout approach, the double pushout approach allows the definition of reconfiguration rules that assure deterministic results gained from the rewriting process. The double pushout approach is extended to rewrite colored (inscribed) Petri nets in two steps: first, it has already been extended to basic Petri nets and second, the rewriting of inscriptions has been added to the approach in previous work of the author. By means of a case study, this approach is presented for the interactive reconfiguration of a given user interface model that uses a visual editor. This visual editor is equipped with an XML-based rewriting component implemented in the UIEditor tool, which has been introduced as a creation and execution tool for FILL-based user interface models in Chap. 5. This chapter is concluded with a discussion of limitations and a set of future work aspects, which mainly address the rule generation and its application to broader use cases.

10.1 Introduction

Human users of interactive systems are highly individual. Their needs, skills, and preferences vary, as do task and context. To address the applicability of interactive systems to different tasks, concepts such as task modeling and task-driven development have been investigated in previous work (Paternò 2004, 2012). Among other concepts, adaptive and reconfigurable user interfaces can incorporate the individual needs and preferences of a specific user. Reconfiguration of user interfaces has proved beneficial in various ways, such as increased usability and a decreased num-

B. Weyers (✉)
Visual Computing Institute—Virtual Reality & Immersive Visualization, RWTH Aachen University, Aachen, Germany
e-mail: weyers@vr.rwth-aachen.de

B. Weyers et al. (eds.), *The Handbook of Formal Methods in Human-Computer Interaction*, Human-Computer Interaction Series,
DOI 10.1007/978-3-319-51838-1_10

ber of errors in interacting with the user interface (Jameson 2009). Langley defines an adaptive user interface as "…an interactive software artifact that improves its ability to interact with a user based on partial experience with that user" (Langley and Hirsh 1999 p. 358). Here, Langley refers to the ability of a user interface to adapt to the user based on its interactions with that user by facilitating concepts from machine learning and intelligent systems research. These inferred adaptations must be applied to the user interface, which therefore needs to be reconfigurable. A reconfigurable user interface can be changed in its outward appearance and its functional behavior (physical representation and interaction logic, see Chap. 5).

The formal specifications for reconfiguring user interfaces have been also discussed as formal methods in human-computer interaction. Nevertheless, the proposed solutions lack a fully fledged formalization for the creation, execution, and reconfiguration of user interface models based on one coherent concept (see Sect. 10.2). Such a concept would not only offer the opportunity to formally validate a created or reconfigured user interface model but also enable the documentation and analysis of the reconfiguration process itself. This can be of interest in cases where the user (and not an intelligent algorithm) applies changes to the user interface, especially in cases of safety critical systems (such as that described in Sect. 10.5) but also for other kinds of user interfaces, for example, for intelligent house control or ambient intelligent systems. To use intelligent systems to derive the needed reconfigurations, it is also possible to enable the intelligent algorithm (and not the user) to generate the adaptation rule, thus instigating the formalized change. Therefore, the approach presented here addresses system- as well as user-driven reconfiguration of formal user interface models; the latter will be examined in Sect. 10.5 as this has been the topic of earlier research.

This chapter primarily presents an approach to the description and implementation of reconfigurable user interfaces based on formal, executable, and reconfigurable models of user interfaces. A corresponding modeling method for user interface models is presented in Chap. 5 based on a visual modeling language (called FILL) and including a transformation algorithm to reference nets (Kummer 2009), a special type of colored Petri nets (Jensen and Rozenberg 2012). In addition to the formal semantics provided by this transformation, the transformed model is executable, based on the existing simulator for reference nets called Renew (Kummer et al. 2000). Therefore, the reconfiguration approach presented in this chapter assumes models based on reference nets or colored Petri nets in more general terms. The reconfiguration approach is based on category theory (Pierce 1991), specifically on the double pushout approach (Ehrig et al. 1997), which has been developed for the rule-based rewriting of graphs. It offers the definition of rewriting rules such that they can be applied to graph models in a deterministic form, which is not always the case, as with the single pushout approach (Ehrig et al. 1997). The general approach described here, which is applicable for rewriting any graph, was first extended to Petri nets by Ehrig et al. (1997). Then, Weyers (2012) extended the rewriting to inscribed Petri nets on an algorithmic and implementation basis. Stückrath and Weyers (2014b) extend thisrewriting further by formally specifying the rewriting of inscribed Petri nets, which extends the double pushout approach through the application of a concept based on lattice theory for the representation of inscriptions.

Furthermore, this chapter discusses the problem of generating rules for rewriting graphs. Rewriting based on the double pushout approach is only a formal and technical basis for the adaptation of user interface models. It offers mainly a formal concept for the modeling of adaptable and adaptive user interfaces such that not only the initial user interface model but also its adaptation is defined formally, which offers formal verification. By storing the applied rules to an initial user interface model, the adaptation process is made completely reproducible and, within certain boundaries, reversible. Nevertheless, rule generation is by design not formally specified. As mentioned above, there are two main options for rule generation: manual generation by the user and generation by an intelligent algorithm. This chapter will present an algorithmic approach, which includes the user in the adaptation process to define which parts of a given user interface model have to be reconfigured and how. Based on this information, an algorithm generates a rule that is applied by the reconfiguration component of the UIEditor (see Chap. 5) to the user interface model. This component is an implementation of the double pushout approach able to rewrite reference nets provided as a serialization in the Petri Net Markup Language (PNML) (Weber and Kindler 2003). For consistency, the reconfiguration concept will refer to the familiar user interface model used in the nuclear power plant case study, which was described in Chap. 4.

The next section presents related work on the formal reconfiguration of user interface models. Section 10.3 offers a detailed introduction of the formal rewriting approach, which will serve as a basis for the description of reconfigurable user interfaces and the core technique for enabling fully fledged formal description of reconfigurable user interface models. Then, Sect. 10.4 presents an interactive approach to rule generation. Rules define the specific reconfiguration applied to a given user interface model. To demonstrate the feasibility of this approach as a combination of formal rewriting and interactive rule generation, Sect. 10.5 presents a case study involving the creation of new functionality and the change of automation offered by an earlier user interface model. The latter was investigated in a previously published user study (Weyers 2012) that found that individualizing a user interface model through formal reconfiguration led to a decrease in user error. Section 10.6 concludes the chapter.

10.2 Related Work

Adaptive user interfaces are an integral part of human-computer interaction research. Various studies have discussed use case-dependent views of adaptive user interfaces, and all have a similar goal: to make interaction between a user and a system less error-prone and more efficient. Jameson (2009) gives a broad overview of various functions of adaptive user interfaces that support this goal. One function he identifies is "supporting system use". He subdivides this into the functions of "taking over parts of routine tasks", "adapting the interface", and "controlling a dialog", all of which are of interest in the context of this chapter. Lavie and Meyer (2010) identify three categories of data or knowledge needed for the implementation of adaptive user interfaces: task-related, user-related, and situation-related, that is, related to the situation in which the interaction takes place. They characterize situations as either routine or non-routine. They also discuss level of adaptivity, which specifies how much adaptation can be applied to a given user interface. These functions are provided by various implementations of and studies on adaptive user interfaces. A general overview of task and user modeling is provided by Langley and Hirsh (1999) and by Fischer (2001). However, none of these studies address how the data and knowledge is gathered or described; instead, they concentrate on how it can be used for applying changes to a given formal user interface model in an interactive fashion.

Various examples can be found of the successful implementation of adaptive user interfaces that address the aspects discussed above. For instance, Reinecke and Bernstein (2011) described an adaptive user interface implementation that takes the cultural differences of users into consideration. They showed that users were 22% faster using this implementation. Furthermore, they made fewer errors and rated the adapted user interface as significantly easier to use. Cheng and Liu (2012) discussed an adaptive user interface using eye-tracking data to retrieve users' preferences. Kahl et al. (2011) present a system called SmartCart, which provides a technical solution for supporting customers while shopping. It provides context-dependent information and support, such as a personalized shopping list or a navigation service. Furthermore, in the context of ambient intelligent environments, Hervás and Bravo (2011) present their adaptive user interface approach, which is based on Semantic Web technologies. The so-called ViMos framework generates visualization services for context-dependent information. Especially in the context of ambient assisted living, there are various types of adaptive systems that include the user interface. For instance, Miñón and Abscal (2012) described a framework that adapts user interfaces for assisted living by supporting daily life routines. They focus on home supervision and access to ubiquitous systems.

Previous work has shown that formal models of user interfaces can be adapted to change their outward appearance, behavior, or both without necessarily abandoning the formalization that describes the user interface model. Navarre et al. (2008a, b) described the reconfiguration of formal user interface models based on predefined replacements that are used in certain safety-critical application scenarios, such as airplane cockpits. Blumendorf et al. (2010) introduced an approach based on so-called executable models that changes a user interface during runtime by combining design information and the current runtime state of the system. Interconnections between system and user interface are changed appropriately during runtime. Another approach that applies reconfiguration during runtime was introduced by Criado et al. (2010).

Thus, adaptive user interfaces play a central role in human-computer interaction and are still the focus of ongoing research. Formal techniques in their development, creation, and reconfiguration are still discussed in the literature, offering various advantages regarding modeling, execution, and verification. Petri net- and XML-based approaches are already in use in various application scenarios. Nevertheless, none of these approaches presents a full-fledged solution for the creation and reconfiguration of user interface models in one coherent formalization. Furthermore, none of the approaches discusses a closely related concept that enables computer-based systems to generate and apply reconfiguration flexibly and independently of use cases. This chapter introduces a self-contained approach for visual modeling and creation (based on the approach presented in the Chap. 5 on FILL), rule-based reconfiguration, and algorithmic rule generation of user interfaces that builds a formal framework for the creation of adaptive user interfaces.

This approach has previously been explored in various publications. In Burkolter et al. (2014) and Weyers et al. (2012) the interactive reconfiguration of user interfaces models was used to reduce errors in interaction with a simulation of a simplified nuclear reactor. In Weyers et al. (2011), Weyers and Luther (2010), the reconfiguration of a user interface model was used in a collaborative learning scenario. In Weyers (2015), the approach was used to describe adaptive automation as part of a user interface model. Publications specifically focusing on the rule-based adaptation of FILL-based user interface models are Stückrath and Weyers (2014b), Weyers (2012), Weyers et al. (2014), Weyers and Luther (2010).

10.3 Formal Reconfiguration

This section introduces a reconfiguration approach (where reconfiguration refers to the adaptation of interaction logic) that is based on the double pushout approach, which originated with category theory and assumes a formal model of a user interface as has been specified in Chap. 5. Thus, the basic architecture of a user interface model is assumed to differentiate between a physical presentation and an interaction logic. The physical representation comprises a set of interaction elements with which the user directly interacts. Each interaction element is related to the interaction logic, which models the data processing between the physical representation and the system to be controlled. It is further assumed that reconfiguration will be applied to the interaction logic. Nevertheless, in various cases changing the interaction logic also implies changes in the physical representation. This will be the topic of the generation of rules and the case study for the application of reconfiguration described in Sects. 10.4 and 10.5. Before presenting the approach itself, the following description will briefly examine the reasons for using a graph-rewriting approach rather than other means of adapting formal models.

10.3.1 Double Pushout Approach-Based Reconfiguration

As mentioned in the introduction to this section, formal reconfiguration can be differentiated from redesign, where redesign refers to changes in the physical representation of a user interface model, while reconfiguration refers to changes in itsinteraction logic. Here, it is assumed that the interaction logic is modeled using FILL and then transformed to reference nets. Thus, reconfiguration means changing reference nets, necessitating a method that is (a) able to change reference net models and (b) defined formally to prevent reconfigurations from being nondeterministic. Various graph transformations and rewriting approaches can be found in the literature. Shürr and Westfechtel (1992) identify three different types of graph rewriting systems. The logic-oriented approach uses predicate logic expressions to define rules. This approach is not widespread due to the complexity of its implementation. Another approach defines rules based on mathematical set theory, which is flexible and easily applied to various applications. Still, it has been shown that irregularities can occur when applying set-theoretical rules to graph-based structures. The third class of techniques is based on graph grammars. Using graph grammars for reconfiguration means changing production rules instead of defining rules to change an existing graph. At a first glance, this seems uncomfortable and counterintuitive for the reconfiguration of interaction logic.

Thus, graph rewriting based on category theory as forth option seems the best starting point for reconfiguration. First of all, pushouts (see Definition 1) as part of category theory are well-behaved when applied to graphs—especially the double pushout (DPO) approach, as discussed by Ehrig et al. (1997). The DPO approach specifies rules that explicitly define which nodes and edges are deleted in the first step and added to the graph in the second. This is not true of the single-pushout (SPO) approach, which is implementation-dependent or generates results that are unlikely to be valid graphs (Ehrig et al. 1997). To give a simple example, the SPO approach can result in dangling edges, which are edges having only a source or a destination but not both. A further problem can be the implicit fusion of nodes, which could have negative implications for the rewriting of interaction logic. These aspects have been resolved in the DPO approach by deleting and adding of nodes and edges explicit and by defining a condition that prevents rules from being valid if they produce dangling edges.

A further argument supporting the use of the DPO approach for rewriting interaction logic is that it has been extended and discussed in the context of Petri nets as introduced by Ehrig et al. (2006, 2008), who offer a solid basis for the reconfiguration of reference net-based interaction logic. Another argument for choosing the Petri net-based DPO approach as described by Ehrig et al. is that it can be easily extended to colored Petri nets. Within certain boundaries, the semantics of the inscription can also be taken into account, as described in detail by Stückrath and Weyers (2014b). Here, the treelike structure of the XML-based definition of inscription is ambiguous, which is discussed in greater detail in Sect. 10.3.2.

Like the SPO, the DPO relies on the category theory-based concept of pushouts. Assuming a fundamental understanding of category theory (otherwise consider, e.g., Pierce 1991), a pushout is defined as follows.

Definition 1 Given two arrows $f : A \to B$ and $g : A \to C$, the triple $(D, g^* : B \to D, f^* : C \to D)$ is called a *pushout*, D is called the *pushout object* of (f, g), and it is true that

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "g"'] & B \arrow[d, "g*"'] \arrow[ddr, "g'"] & \\
C \arrow[r, "f*"] \arrow[drr, "f'"'] & D \arrow[dr, dotted, "h"] & \\
 & & E
\end{tikzcd}$$

Fig. 10.1 A pushout diagram

1. $g^*\circ f = f^*\circ g$, and
2. for all other objects E with the arrows $f' : C \rightarrow E$ and $g' : B \rightarrow E$ that fulfill the former constraint, there has to be an arrow $h : D \rightarrow E$ with $h\circ g^* = g'$ and $h\circ f^* = f'$.

The first condition specifies that it does not matter how A is mapped to D, whether via B or via C. The second condition guarantees that D is unique, except for isomorphism. Thus, defining (f, g) there is exactly one pushout (f^*, g^*, D) where D is the rewritten result, also called the pushout object. In general, A and B are produced by defining the changes applied to C, the graph to be rewritten. Therefore, a rewriting rule can be specified as a tuple $r = (g, f, A, B)$, such that D is the rewritten result by calculating the pushout (object). This procedure is mainly applied in the SPO approach. The resulting diagram is shown in Fig. 10.1.

To define the DPO approach, the *pushout complement* has to be defined first.

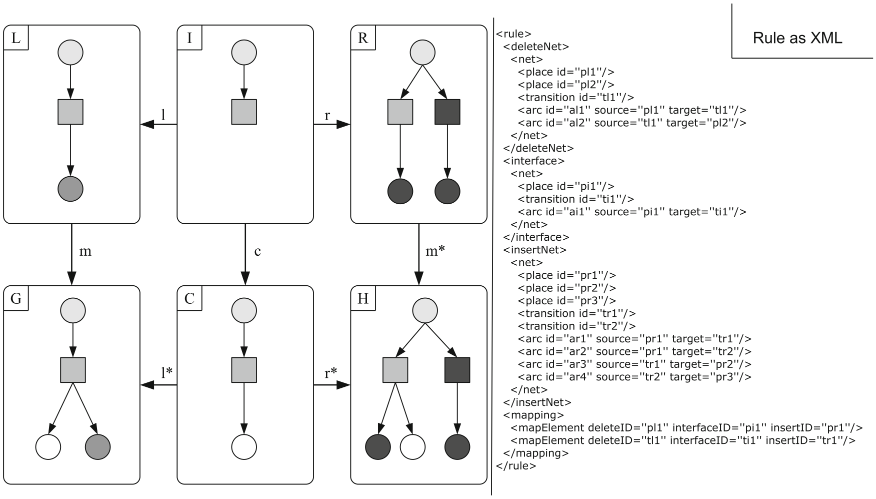
Definition 2 Given two arrows $f : A \rightarrow B$ and $g^* : B \rightarrow D$, the triple $(C, g : A \rightarrow C, f^* : C \rightarrow D)$ is called the *pushout complement* of (f, g^*) if (D, g^*, f^*) is a pushout of (f, g).

A DPO rule is then defined based on the definition of a *production* corresponding to the former discussion of pushouts in category theory.

Definition 3 A *matching* is a mapping $m : L \rightarrow G$; a *production* is a mapping $p : L \rightarrow R$, where L, R, and G are graphs. The corresponding mappings of m and p are defined as mapping $m^* : R \rightarrow H$ and $p^* : G \rightarrow H$, where H is also a graph.

Definition 4 A *DPO rule* s is a tuple $s = (m, (l, r), L, I, R)$ for the transformation of a graph G, with $l : I \rightarrow L$ and $r : I \rightarrow R$, which are two total homomorphisms representing the production of s; $m : L \rightarrow G$ is a total homomorphism matching L to graph G. L is called the *left side* of s, R is called the *right side* of s, and I is called an *interface graph*.

Given a rule s, the pushout complement C can first be calculated using L, I, m, and l with a given graph G to be rewritten. In the DPO approach, this step deletes nodes



```
<rule>
  <deleteNet>
    <net>
      <place id="pl1"/>
      <place id="pl2"/>
      <transition id="tl1"/>
      <arc id="al1" source="pl1" target="tl1"/>
      <arc id="al2" source="tl1" target="pl2"/>
    </net>
  </deleteNet>
  <interface>
    <net>
      <place id="pi1"/>
      <transition id="ti1"/>
      <arc id="ai1" source="pi1" target="ti1"/>
    </net>
  </interface>
  <insertNet>
    <net>
      <place id="pr1"/>
      <place id="pr2"/>
      <place id="pr3"/>
      <transition id="tr1"/>
      <transition id="tr2"/>
      <arc id="ar1" source="pr1" target="tr1"/>
      <arc id="ar2" source="pr1" target="tr2"/>
      <arc id="ar3" source="tr1" target="pr2"/>
      <arc id="ar4" source="tr2" target="pr3"/>
    </net>
  </insertNet>
  <mapping>
    <mapElement deleteID="pl1" interfaceID="pi1" insertID="pr1"/>
    <mapElement deleteID="tl1" interfaceID="ti1" insertID="tr1"/>
  </mapping>
</rule>
```

Fig. 10.2 Example for a DPO rule showing its application to a Petri net *G*

and edges from *G*. Second, the pushout is calculated using *I*, *R*, and *r* applied to *C* resulting in the graph *H*. This step adds nodes and edges to *C*. Finally, the difference between *L* and *I* specifies the part deleted from *G*, where the difference between *I* and *R* defines those elements, which are added to *C* and finally to *G*. The result of applying *s* to *G* is the graph *H* as can be seen in Fig. 10.2.

Nevertheless, the pushout complement is not unique in all cases and probably does not even exist. However, if the total homomorphisms *l* and *m* fulfill the *gluing condition* given below, the pushout complement can be considered to exist. The gluing condition is defined as follows.

Definition 5 There are three graphs $I = (V_I, E_I, s_I, t_I)$, $L = (V_L, E_L, s_L, t_L)$, and $G = (V_G, E_G, s_G, t_G)$. Two graph homomorphisms $l : I \to L$ and $m : L \to G$ fulfill the *gluing condition* if the following assertions are true for both *l* and *m*:

$$\nexists e \in (E_G \setminus m(E_L)) : s_G(e) \in m(V_L \setminus l(V_I)) \vee t_G(e) \in m(V_L \setminus l(V_I)), \qquad (10.1)$$

and

$$\nexists x, y \in (V_L \cup E_L) : x \neq y \wedge m(x) = m(y) \wedge x \notin l(V_I \cup E_I). \qquad (10.2)$$

Condition 10.1 is called *dangling condition*. The homomorphism *l* of a DPO rule that defines which nodes are to be deleted from a graph fulfills the dangling condition if it also defines which edges associated with the node will be removed. Thus, the dangling condition avoids dangling edges; a *dangling edge* is an edge that has only one node associated with it as its source or target. Condition 10.2 is called *identification condition*. The homomorphism *m* fulfills the identification condition if a node in G that should be deleted has no more than one preimage in L. However, if one node in G has more than one preimage in L defined by m and one of these has to be deleted, it is not defined whether the node will still exist in G or must be deleted. This confusion is avoided by applying the identification condition.

The problems of the SPO approach discussed above are mainly solved by the gluing condition being an integral part of the DPO approach. The pushout complement is not unique but exists if the gluing condition is fulfilled. If l and m are injective, the pushout complement will be unique except in the case of isomorphy. This is further discussed by Heumüller et al. (2010) and in Weyers (2012, p. 107).

Finally, rule descriptions have to be serialized in a computer-readable form. This is necessary for the implementation-side use of rewriting rules presented in the next section. Therefore, an existing XML-based description language for Petri nets (PNML Weber and Kindler 2003) has been used with an extension that structures and describes the rule-specific elements, such as indicating L as `deleteNet`, I as `interface`, and R as `insertNet`. Thus, every net (as embedded in a $< net >$ node) is given as a PNML-based description of the respective left, interface, or right graph of the DPO rule. The $< mapping >$ node specifies l and r as a set of $< mappingElements >$ that are representations of tuples of XML ids. Mapping to G is not part of the rule serialization because a rule is first and foremost independent from a graph being rewritten. An example of such an XML-based representation of a rule can be seen in Fig. 10.2 on the side of the DPO diagram, which also shows the rule applied to a Petri net G.

10.3.2 Rewriting Inscriptions

For the rewriting of inscriptoins, the rewriting approach introduced previously involves only two steps. First, the node that carrying the inscription to be rewritten is deleted. Second, a new node carrying the new inscription is added. This new node must have the same connection as the deleted node; otherwise, the rewriting would also change the structure of the graph, which should not be the case since only the inscription is being rewritten. The problem with this approach is that the rule has to map all incoming and outgoing edges, which can increase the effort involved in generating rules. If this is not done carefully, edges will be delete that are not mapped to prevent dangling edges, generating unintended changes in the net. Furthermore, detailed rewriting of inscriptions offers finer-grained changes to be applied to a Petri net. Thus, slight changes can be made in, for example, guard conditions without the need to rewrite the structure of the net.

Therefore, the previous rewriting approach has been extended such that the rule is not only aware of the nodes and edges in the graph but also of the node's inscriptions. The complete definition and proof of this extension can be found in Stückrath and Weyers (2014a, b). The discussion in this section will focus on deletion-focused PNML-based rewriting. It is assumed that the net and the inscriptions are given as

PNML-based serialization, as introduced above. An XML-based inscription can be formally defined as follows.

Definition 6 Let $(Val, \leq)$ be a disjoint union of complete lattices Val_i of values with $\biguplus_{i \in I} Val_i = Val$ and let N be a set of IDs sorted such that it can be partitioned in N_i with $N = \biguplus_{i \in I} N_i$. An XML inscription $xml_{N,Val}$ is a directed rooted tree (V, E, r, γ) of finite height, where V is a set of vertices (nodes), $E \subseteq V \times V$ is a set of edges, $r \in V$ is the root and $\gamma : V \rightarrow \bigcup_{i \in I}(N_i \times Val_i)$ maps properties to each vertex. Additionally, for every two edges $(v_1, v_2), (v_1, v_3) \in E$ with $\gamma(v_i) = (n_i, w_i)$ (for $i \in \{2, 3\}$), it holds that $n_2 \neq n_3$.

For every $v \in V$ we define $v{\downarrow} = (V', E', v, \gamma')$ to be the subtree of $xml_{N,Val}$ with root v, which is an XML inscription itself.

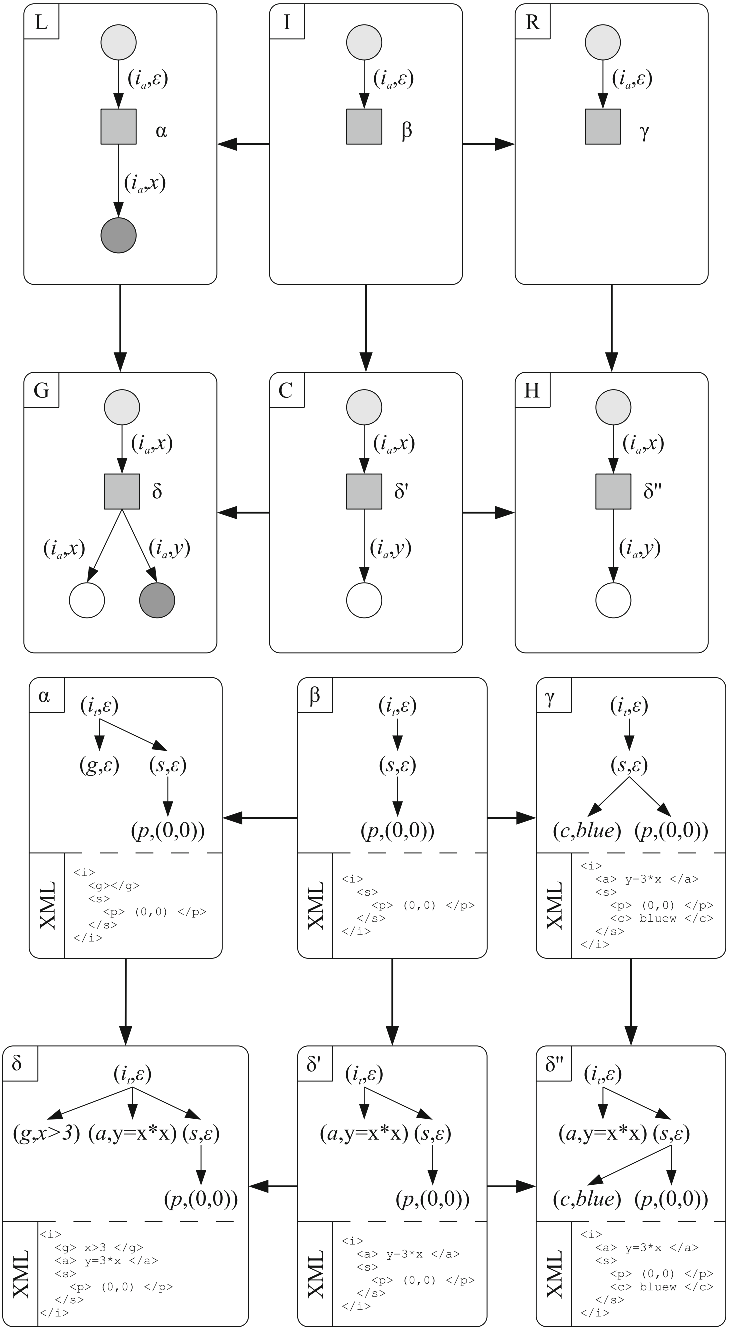
Thus, an XML-based inscription is nothing but a tree structure of nodes and edges, which can be organized into subtrees in a well-defined manner. For the rewriting, it is important to define an order for these trees, so that it is clear whether a subtree is part of another subtree and whether this second subtree is smaller or larger. This is important for deciding what is to be deleted or added and thus for calculating the differences between right side, interface, and left side of the rule. The order is defined on a lattice basis.

Definition 7 Let $XML_{N,Val}$ be the set of all XML inscriptions $xml_{N,Val}$. We define the ordered set $(XML_{N,Val}, \sqsubseteq)$, where for two elements $(V_1, E_1, r_1, \gamma_1) \sqsubseteq (V_2, E_2, r_2, \gamma_2)$ holds if and only if $\gamma_i(r_i) = (n_i, w_i)$ for $i \in \{1, 2\}$. Then $n_1 = n_2$, $w_1 \leq w_2$, and for all $v_1 \in V_1$ with $(r_1, v_1) \in E_1$, there is a $v_2 \in V_2$ with $(r_2, v_2) \in E_2$ such that $v_1{\downarrow} \sqsubseteq v_2{\downarrow}$.

10.3.2.1 Deletion-Focused Rewriting

Deletion-focused rewriting refers to rewriting that prefers the deletion of an inscription and thereby prevents undefined behavior by the rewriting rule. An example can be seen in Fig. 10.3. The rewriting is based on the DPO approach, calculating deletion and adding elements in the inscription by first computing the pushout complement δ' and then computing the pushout object δ''. What is to be deleted or added (as with the DPO net rewriting) is identified based on the definition of the difference between the left side and the interface graph of the rule (here α and β) and between the interface graph and the right side (here β and γ). As defined above, these differences are derived using the lattice-based specification of (XML) tree-structured inscriptions. Furthermore, the pushout complement and the pushout itself are calculated using a strategy called *deletion-focused rewriting*. This strategy is needed to prevent the rewriting of undefined behavior if the pushout complement is not unique (see discussion in the previous section). In deletion-focused rewriting, the pushout complement with the smallest result (defined by the lattice) is chosen.

To make this strategy clearer, Fig. 10.3 shows an example of the deletion-focused rewriting of a net using the rule and rewriting shown in the upper part of the figure. The mapping of the left side to the net to be rewritten is shown as hatching pattern,



```
<i>
  <g></g>
  <s>
    <p> (0,0) </p>
  </s>
</i>
```

```
<i>
  <s>
    <p> (0,0) </p>
  </s>
</i>
```

```
<i>
  <a> y=3*x </a>
  <s>
    <p> (0,0) </p>
    <c> bluew </c>
  </s>
</i>
```

```
<i>
  <g> x>3 </g>
  <a> y=3*x </a>
  <s>
    <p> (0,0) </p>
  </s>
</i>
```

```
<i>
  <a> y=3*x </a>
  <s>
    <p> (0,0) </p>
  </s>
</i>
```

```
<i>
  <a> y=3*x </a>
  <s>
    <p> (0,0) </p>
    <c> bluew </c>
  </s>
</i>
```

Fig. 10.3 Example for rewriting of an XML-based inscription of a petri net. The *upper half* shows the DPO rule for rewriting the net; the *lower half* shows the DPO rule based on lattice-structured inscription for rewriting the inscription of one transition (*middle gray*) in the upper DPO rule (see α, β, γ and δ to δ'')

where a similar pattern defines the mapping of two nodes (in L and G) to one another. in the lower part of Fig. 10.3, the rewriting of the inscription of a transition (vertical hatching) is visible. Various types of XML nodes are involved, such as the inscription node $< i >$, guard condition $< g >$, style node $< s >$ and nodes for color $< c >$ and position $< p >$. The rule assumes that, after the deletion of the node (g, e) (specified as difference between α and β), the resulting inscription node no longer has a guard condition, and it therefore prefers to delete rather than preserve an inscription. One could assume a preservation-focused rewriting, which deletes only nodes that are equal to or smaller than the matched one. In the present case, the matched node is larger and thus contains non-empty content ($x > 3$); therefore, it would not be deleted because the preservation-focused rewriting rule would be inapplicable (Stückrath and Weyers 2014b).

With deletion-focused rewriting, the outcome is different. Nodes are also deleted if they are larger as shown in the example. The rule specifies the deletion of the guard inscription node such that the push-out complement δ', as shown in Fig. 10.3, is derived. In this case, the complete node will be deleted. In addition to the deletion of the guard condition $(g, x > 3)$ from graph δ, the rule specifies adding a color to the style inscription node. Similar to the rewriting of Petri nets, the rules specify a new subnode of the (s, ϵ) node. Applying this adding operation to δ' yields the graph δ''. The result is a style node defining a position $(0, 0)$ and a color *blue* for the inscribed transition.

10.4 Interactive Reconfiguration and Rule Generation

The previous sections introduced the basic theory of rule-based rewriting of colored Petri nets based on the DPO approach including an extension for deletion-focused rewriting of tree-structured inscriptions. This section will introduce an approach for generating rules in a given context including the use of explicit user input and using algorithms to generate rules for rewriting the interaction logic based on user input. It will also explore how rewriting interaction logic influences the physical representation of a user interface model.

The interactive reconfiguration of formally specified user interfaces based on FILL is a component of the UIEditor (see Chap. 5). This component implements algorithms for the generation of rules as well as a visual editor that enables the user to apply these reconfiguration operations to a given user interface model. With this editor, the user can select individual or a group of interaction elements and apply a reconfiguration operation to this selection. Thus, this interactive approach to rule generation uses the physical representation as a front end for the user to define which parts of the underlying interaction logic are to be reconfigured.

The user interface of this visual editor is shown in Fig. 10.4. The workspace presents the physical representation of the user interface to be reconfigured. The tool bar at the top right of the editor window offers a set of reconfiguration operations. For instance, the user can select two buttons and choose a reconfiguration operation

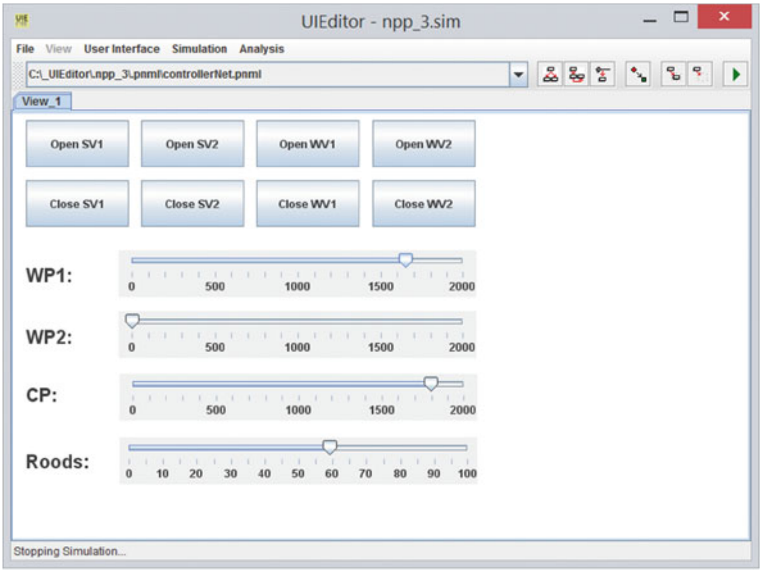


Fig. 10.4 Reconfiguration editor of the UIEditor showing an exemple user interface. In the *upper right corner* is a tool bar with various buttons for the applying reconfiguration operations to a selection of interaction elements

that creates a new button that triggers the operations associated with the two selected buttons in parallel with only one click. The selected buttons indicate which part(s) of the interaction logic must be rewritten. The specific parts of the interaction logic affected are determined by simple graph-traversing algorithms. Selecting the operation (here, `parallelization`) specifies the kind of extension or change to be applied to the interaction logic.

An rule resulting from the aforementioned interactive rule generation can be seen in Fig. 10.5 on the left. Here, the user selected the buttons labeled `Input A` and `Input B`. By applying the parallelization operation to this selection, the underlying algorithm generated the rule shown in the lower half of Fig. 10.5. Applying this rule to the interaction logic creates the net on the right labeled “Reconfiguration”, which is also related to a new interaction element, here a newly added button. This example shows that this kind of reconfiguration always implies a change in the physical representation—here, the addition of a new button that triggers the newly created part of the interaction logic.

Another example of an implemented reconfiguration operation is shown in the middle of Fig. 10.5, where the `discretization` of a continuous interaction element is shown. The term *continuous* refers to the types of values that are generated via an interaction element—here, a slider is used to generate values in a certain range. In contrast to a slider, which can generate any of a range of values, a button is a discrete interaction element and can only generate one specific value: an event. Thus, `discretization` is a reconfiguration operation that maps a continuous

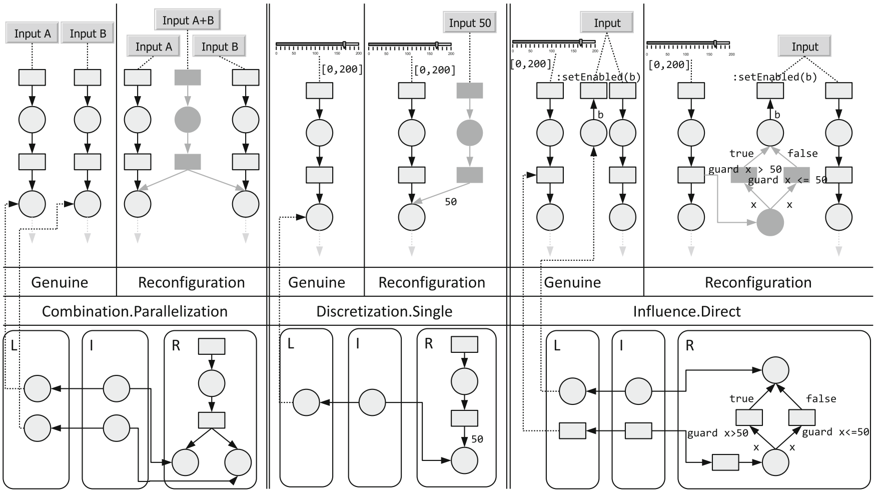


Fig. 10.5 Three examples of reconfiguration operations applied to a user interface. *Left* application of a `parallelization` operation to two buttons, creating a new button that combines the functions of the original two; *middle* application of a `discretization` operation to a slider generating a button that sets a preset value to the system parameter controlled by the slider; *right* application of a `direct` reconfiguration operation that influences the behavior of two interaction elements by making them dependent on each other

interaction element to a discrete one. In Fig. 10.5, a slider is mapped to a button. The button generates a single value from the slider's interval—here, the value 50.

The last example presented in Fig. 10.5 is of a type of rule that cannot be generated interactively. Here, a certain part of the underlying interaction logic of an interaction element is changed. The added subnet changes the status of the button `Input`. Based on the value selected with the slider, the button is disabled or enabled: If the value is ≤50, it will be disabled; if the value is >50, it will be enabled. This type of reconfiguration operation can be pre-defined or determined by using extended methods for rule generation, as discussed in Weyers et al. (2014) in detail.

10.5 Case Study

To demonstrate how interaction logic can be rewritten, this section discusses its use in the nuclear power plant case study (see Chap. 4). This user interface model (as partially described in Chap. 5) includes a simple interaction logic that triggers the operations as indicated by the interaction elements' labels (see Fig. 10.6). However, the user interface used in the study offers no complex operations such as the SCRAM operation for emergency shutdown the reactor. Such an operation can be added to the user interface model as an extension of its interaction logic; this process will be

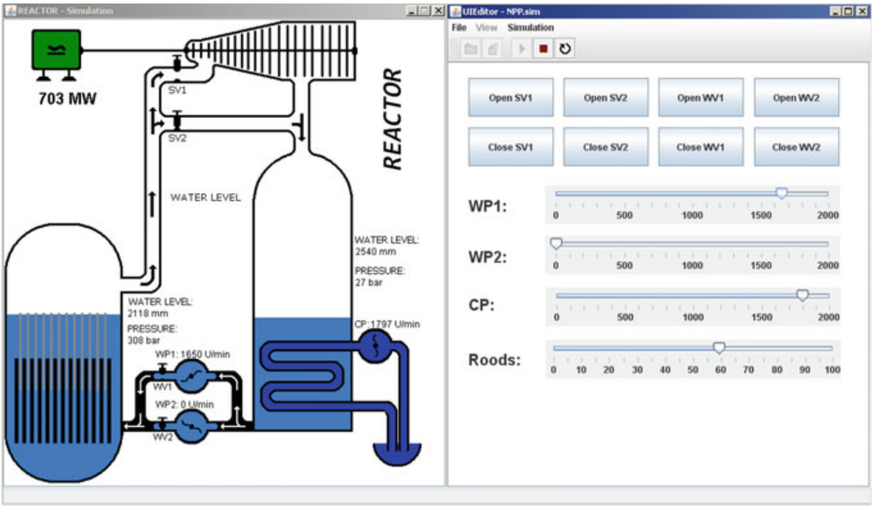


Fig. 10.6 The initial user interface for the control of a simplified simulation of a nuclear power plant running in UIEditor

explained in the next subsection. The following subsection examines the effect of a user-driven reconfiguration on the basis of a user study. A final subsection discusses these results and various aspects of future work.

10.5.1 Case Study: SCRAM Operation

The so-called SCRAM operation is currently missing as part of the user interface. As described in Chap. 4, the SCRAM operation shuts down the reactor immediately and returns it to a safe state. This includes the complete insertion of the control rods into the reactor, which controls the thermal output of the core and stops the chain reaction. The pumps have to continue to cool the core, and the turbine should be shut off from the steam circuit. Finally, the condenser pump has to continue working to condense the steam still being produced by the reactor and thus cool the core.

To implement the SCRAM operation into the given user interface model, users could apply a limited number of reconfigurations to the user interface. Firstly, they could select a slider to define the position of the control rods and apply the `discretization` operation. Here, selecting `0` created a button that set the position of the control rods to 0, such that they were completely inserted in the core. Thus, by pressing the newly created button, the chain reaction would be stopped and the core would produce only residual heat.

Second, the Buttons to open SV2, close SV1, open WV1 and close WV2 should be combined to one button by applying the `parallelization` operation. Thus, the

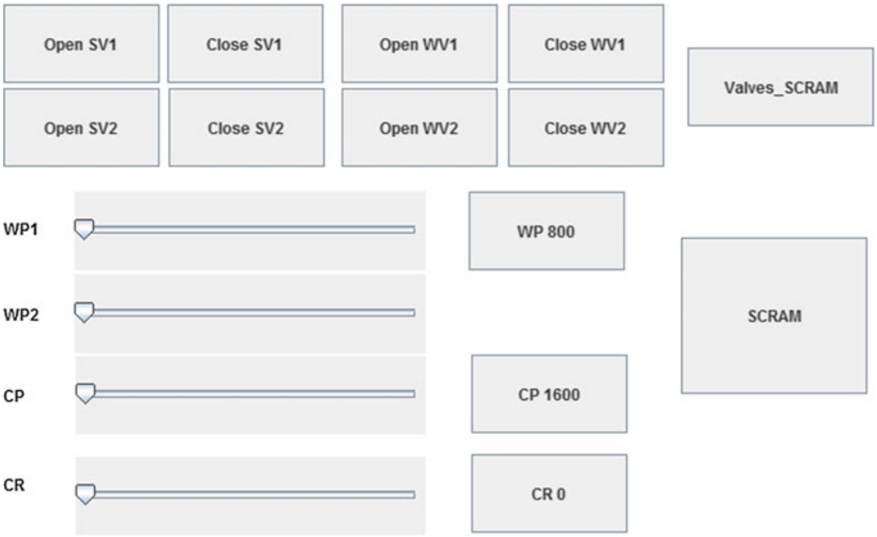


Fig. 10.7 A user interface showing the various operations needed for a SCRAM operation, which has been also added. The user interface shows the result of the application of various reconfiguration operations to an initial user interface through interactive and sequential application of reconfiguration operations

new Button subsumes all relevant valve operations needed for the SCRAM operation. Third, the water pumps WP1 and CP have to be set to a certain speed: WP1 to 800 rpm and CP to 1600 rpm. Therefore, the user selects first the slider for WP1 and applies the `discretization` operation as she did for the control rod position. She repeats this with the slider for CP. The final step is to fuse all the newly created Buttons into one using the `parallelization` operation. The resulting physical representation of the created user interface can be seen in Fig. 10.7.

10.5.2 User Study: Error Reduction Through Individualization

In addition to the extension of the user interface's functionality by creating new interaction elements extending the interaction logic, the effect of such extension on the interaction process between the human user and the system must also be analyzed. To do this, a user study was conducted to measure the effect of the individualization of a given user interface model on the number of errors users committed while working with that interface. For the user study, participants had to control the simplified simulation of the nuclear power plant. The nuclear power plant simulation had been selected to keep the attention of the participants as high as possible (because they all know about the criticality of faulty control of nuclear plants) and to be as flexible as possible regarding the degree of complexity for the control task. The control process was simple to adapt for the study in that it could be learned in a limited time but was complex enough to make any effects visible. The results of the study indicated a reduction of errors in controlling the power plant when participants used their individually reconfigured user interfaces (Burkolter et al. 2014; Weyers et al. 2012).

In total 72 students of computer science (38 in the test group) participated in the study. First, all participants responded to a pre-knowledge questionnaire to ensure that the groups were balanced regarding knowledge about nuclear power plants. Next, all participants were introduced to the function of a nuclear power plant and its control. This introduction was a slide-based presentation that was read aloud to keep content and time equal among all groups of participants. Afterwards, participants participated in an initial training session on the controls for the nuclear power plant simulation; the instructions were presented on paper. In total three procedures were practiced: starting the reactor, shutting it down, and returning the reactor to a safe state after one of the feedwater pumps failed. Participants were allowed enough time that every participant was able to complete each procedure at least once. After the training in each procedure, the participants in the test group had the chance to apply reconfiguration operations to the initial state of the user interface and test their changes. For this, they had a specific amount of time after each training run. To keep time on task equal for both the test and the control group, the control group watched a video on simulation in general (not of nuclear plants) for the same amount of time as the test group had to apply reconfigurations to their user interface. To apply changes to the initial user interface, they used the interactive rule generation procedure as implemented in UIEditor. An example of an outcome of this kind of individualization can be seen in Fig. 10.8.

After the training and reconfiguration phases, the participants had to start up and shut down the reactor several times, simulating real use of the system. In the last runthrough, a feedwater pump broke down. Participants had been trained in the response to this event, but had not been informed in advance that it would occur.

During the trials, errors were measured in the sense of wrongly applied operations. An operation was classified as wrongly applied if it did not match the expected operation for the startup or shutdown of the reactor or for the response to the water pump breakdown in the final runthrough. Various types of errors were detected, based on the classification defined by Hollnagel (1998). All error types identified are shown in Fig. 10.9, which also shows how the log files were evaluated by manually looking for patterns and mismatches.

The results show that the control group using the individualized user interface models made fewer errors in interacting with the reactor than did the control group, which used the initial user interface without individualization. Furthermore, the test group was able to respond to the system failure more effectively than the control group. These results are shown in Table 10.1. This study shows the potential for this type of adaption of user interface models. It has been previously published in Burkolter et al. (2014) and Weyers et al. (2012).

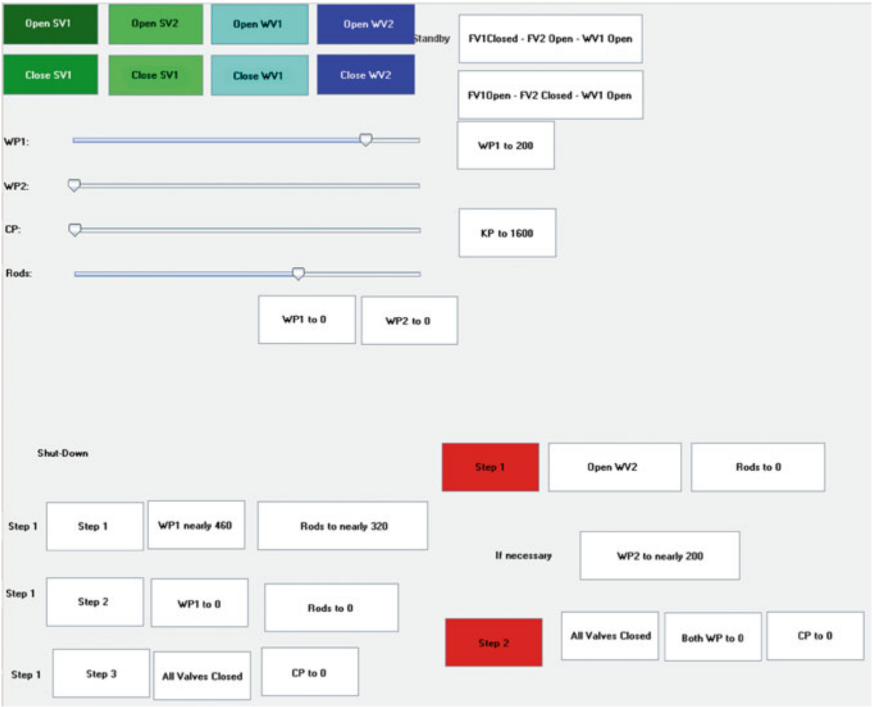
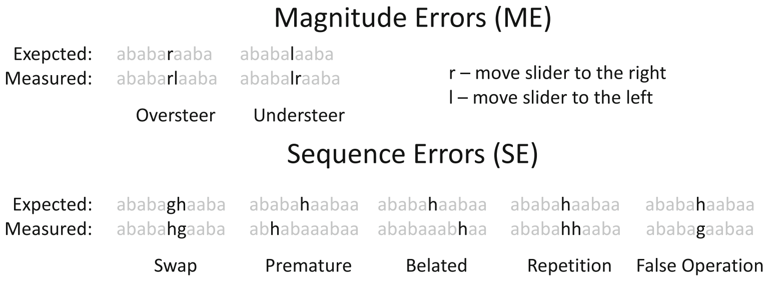


Fig. 10.8 An example of a reconfigured user interface for the control of a simplified simulation of a nuclear power plant



Magnitude Errors (ME)

Exepcted: ababaraaba ababalaaba

Measured: ababarlaaba ababalraaba

Oversteer Understeer

r – move slider to the right

l – move slider to the left

Sequence Errors (SE)

Expected: ababaghaaba ababahaabaa ababahaabaa ababahaabaa ababahaabaa

Measured: ababahgaaba abhabaaabaa ababaaabhaa ababahhaaba ababagaabaa

Swap Premature Belated Repetition False Operation

Fig. 10.9 Different error types evaluated. The errors were identified by comparing the interaction logs gathered during the use of the system with the expected operation sequences, which were provided to the participants as control operation sequences on paper

10.5.3 Discussion

In addition to the benefits of a user-driven reconfiguration of a user interface model as shown above, there are various side conditions to be considered. First, by adding more abstract operations to the user interface, the degree of automation is increased

Table 10.1 Startup of reactor and practiced system fault: mean, standard deviation (in parentheses), and results of the applied t-test for comparing the means of the group using the initial user interface (NonRec-group) and the grouping using their individualized/reconfigured user interface (Rec-group)

Error types	Rec-group	NonRec-group	t(df), p (One-tailed), ES r
Magnitude error			
Oversteering	0.38 (0.70)	2.38 (1.72)	t(45.86) = −6.13, **p = 0.000**, r = 0.67
Understeering	0.15 (0.37)	0.15 (0.36)	t(58) = 0.07, p = 0.472, r = 0.01
Sequence error			
Swap	0.04 (0.20)	0.47 (0.86)	t(37.40) = −2.83, **p = 0.004**, r = 0.42
Premature	0.23 (0.43)	0.24 (0.50)	t(58) = −0.04, p = 0.486, r = 0.01
Belated	0.42 (0.58)	0.38 (0.70)	t(58) = 0.24, p = 0.405, r = 0.03
Repetition	0.38 (0.70)	0.09 (0.29)	t(31.54) = 2.04, **p = 0.003**, r = 0.34
False operation	0.96 (1.25)	1.82 (1.90)	t(56.87) = −2.12, **p = 0.020**, r = 0.27
Total	2.48 (1.93)	5.53 (2.71)	t(59) = −4.93, **p = 0.000**, r = 0.54

which is known to have a potentially negative influence on task performance. This is especially true in critical situations (Parasuraman et al. 2000). However, it is questionable at which point this also becomes true for automation added to the user interface model by the users themselves. The study described argues for a higher task performance through user-driven increases in automation. However, there is no data on whether this is still true if the user works with the reconfigured user interface over a longer period of time. The interactive process of creating more abstract operations could also have a certain effect on training and user understanding of the interactive system controls. The effects of increased automation and the enhancement of training through interactive reconfiguration are aspects to be considered in future work.

A related problem is the creation of erroneous operations which a user could implement while individualizing the user interface. Erroneous operations could result in unexpected system operation or faulty system states. The scenario discussed above does not prevent users from adding erroneous operations to the user interface model. However, the underlying seamless formalization of the user interface model and the reconfiguration process constitute the basis for a formal validation of the assumed outcome of a reconfiguration. Thus, the user can be informed of potential problems arising from the application of a certain reconfiguration rule. Such a validation approach is also of interest for automatic non-user-driven reconfiguration. Nevertheless, the development of "on-the-fly" validation is not trivial, especially if the underlying system is not known. Initial studies on this subject have been conducted by facilitating a SMT solver (Nieuwenhuis et al. 2006) for the validation of interaction logic. Here, the generated reference net was transformed algorithmically into an expression of first-order logic. This expression was composed by the pre- and post-conditions for the transitions in the reference net and is being tested by the SMT solver for satisfactory performance. It could be extended by certain pre-conditions relevant to the use case, making it possible to check whether an extended interaction logic generates input values to the system that are not valid or specified for the system to be controlled.

The approach shown offers other benefits related to creation rather than the use of such user interface models. By formalizing the user interface model and its reconfiguration, both models and reconfiguration rules can be reused and facilitated as provenance information describing the reconfiguration process. If the underlying system is replaced by, for example, a newer version or a similar system, the user interface model can be reused as a whole—assuming, of course, that there are no changes in the system interface providing the access to the system values to be controlled. However, if changes have been applied to that interface, these can simply be adopted in the user interface model as well without the need to touch any code. It is imaginable that such changes could be applied automatically by applying reconfiguration rules gathered from a (hopefully formalized) description of the API changes.

The user study presented addresses only the use of reconfigured user interfaces by using the interactive rule generation approach discussed in Sect. 10.4. No deeper evaluation of UIEditor has been conducted, especially addressing the creation of user interface models. The exception is results gathered from items in the post-study questionnaire of the study described above, which indicated no problems with the reconfiguration concept. However, an important goal of future work is to intensify the research and development effort on measuring and enhancing the usability and user experience of UIEditor. The use of UIEditor by non-experts in HCI and user interface modeling is also of great interest. This user group, which would include mechanical engineers among others, would benefit from the simplified creation and rapid prototyping of user interfaces without having to resign the benefits of formal validation and reuse of such models.

As these case studies show, the reconfiguration approach is well-suited for the individualization of user interface models. Nevertheless, the approach offers the algorithmic generation of reconfiguration rules by, for example, intelligent algorithms, as is the case with adaptive or intelligent interactive systems (Langley and Hirsh 1999). An initial approach to the automatic generation of reconfiguration rules was presented in Weyers et al. (2014). Such an extension to the current approach could also enable its integration into frameworks like CAMELEON (Calvary et al. 2003; Pleuss et al. 2013). The CAMELEON framework focuses mainly on the application of abstract user interface models to specific contexts and systems in final user interface models. Nevertheless, it does not address the underlying functionality, that is, the interaction logic of a user interface. The approach presented here could embed interaction logic into the transformation from abstract to concrete user interface models. Starting with an abstract interaction logic and reconfiguring it step-wise using rule-based rewriting and applying the transformation to the physical representation makes that interaction logic specific to a certain context, user, task, or device.

10.6 Conclusion

This chapter presents a solution for the reconfiguration of formal user interface models based on reference nets, a special type of Petri nets. This reconfiguration is based on a graph rewriting approach called the double pushout approach, which has been extended to inscribed/colored Petri nets. This extension was based on lattice theory to define a formal structure for XML node and edge inscription, making it possible to rewrite inscriptions in such a way that it is not necessary to apply any changes to the net structure. The applicability of this approach was demonstrated in a case study in which new operations were added to an initial user interface for the control of a simplified nuclear power plant simulation. For this reconfiguration, an interactive rule generation approach was facilitated which enabled the user to specify the relevant parts of the user interface to be reconfigured as well as the type of reconfiguration to be applied. This tool is part of UIEditor as introduced in Chap. 5.

References

Blumendorf M, Lehmann G, Albayrak S (2010) Bridging models and systems at runtime to build adaptive user interfaces. In: Proceedings of the 2nd ACM SIGCHI symposium on engineering interactive computing systems, Berlin

Burkolter D, Weyers B, Kluge A, Luther W (2014) Customization of user interfaces to reduce errors and enhance user acceptance. Appl Ergon 45(2):346–353

Calvary G, Coutaz J, Thevenin D, Limbourg Q, Bouillon L, Vanderdonckt J (2003) A unifying reference framework for multi-target user interfaces. Interact Comput 15(3):289–308

Cheng S, Liu Y (2012) Eye-tracking based adaptive user interface: implicit human-computer interaction for preference indication. J Multimodal User Interface 5(1–2):77–84

Criado J, Vicente Chicote C, Iribarne L, Padilla N (2010) A model-driven approach to graphical user interface runtime adaptation. In: Proceedings of the MODELS conference, Oslo

Ehrig H, Heckel R, Korff M, Löwe M, Ribeiro L, Wagner A, Corradini A (1997) Algebraic approaches to graph transformation. Part II: single pushout approach and comparison with double pushout approach. In: Rozenberg G (ed) Handbook of graph grammars and computing by graph transformation. World Scientific Publishing, Singapore

Ehrig H, Hoffmann K, Padberg J (2006) Transformation of Petri nets. Electron Notes Theor Comput Sci 148(1):151–172

Ehrig H, Hoffmann K, Padberg J, Ermel C, Prange U, Biermann E, Modica T (2008) Petri net transformation. In: Kordic V (ed) Petri net, theory and applications. InTech Education and Publishing, Rijeka

Fischer G (2001) User modeling in human-computer interaction. User Model User-Adap Interact 11(1–2):65–86

Hervás R, Bravo J (2011) Towards the ubiquitous visualization: adaptive user-interfaces based on the semantic web. Interact Comput 23(1):40–56

Heumüller M, Joshi S, König B, Stückrath J (2010) Construction of pushout complements in the category of hypergraphs. In: Proceedings of the workshop on graph computation models, Enschede

Hollnagel E (1998) Cognitive reliability and error analysis method (CREAM). Elsevier, Amsterdam

Jameson A (2009) Adaptive interfaces and agents. Hum Comput Interact Des Issues Solut Appl 105:105–130

Jensen K, Rozenberg G (2012) High-level petri nets: theory and application. Springer, Berlin

Kahl G, Spassova L, Schöning J, Gehring S, Krüger A (2011) Irl smartcart-a user-adaptive context-aware interface for shopping assistance. In: Proceedings of the 16th international conference on intelligent user interfaces, Palo Alto

Kummer O (2009) Referenznetze. Logos, Berlin

Kummer O, Wienberg F, Duvigneau M, Köhler M, Moldt D, Rölke H (2000) Renew—the reference net workshop. In: Tool demonstrations, 21st international conference on application and theory of Petri nets. Computer Science Department, Aarhus

Langley P, Hirsh H (1999) User modeling in adaptive interfaces. In: Proceedings of user modeling, Banff

Lavie T, Meyer J (2010) Benefits and costs of adaptive user interfaces. Int J Hum Comput Stud 68(8):508–524

Miñón R, Abascal J (2012) Supportive adaptive user interfaces inside and outside the home. In: Proccedings of user modeling, adaption and personalization workshop, Girona

Navarre D, Palanque P, Basnyat S (2008a) A formal approach for user interaction reconfiguration of safety critical interactive systems. In: Computer safety, reliability, and security. Tyne

Navarre D, Palanque P, Ladry JF, Basnyat S (2008b) An architecture and a formal description technique for the design and implementation of reconfigurable user interfaces. In: Interactive systems. Design, specification, and verification, Kingston

Nieuwenhuis R, Oliveras A, Tinelli C (2006) Solving sat and sat modulo theories: from an abstract davis-putnam-logemann-loveland procedure to dpll (t). J ACM 53(6):937–977

Parasuraman R, Sheridan T, Wickens C (2000) A model for types and levels of human interaction with automation. IEEE Trans Syst Man Cybern Syst Hum 30(3):286–297

Paternò F (2004) Concurtasktrees: an engineered notation for task models. In: The handbook of task analysis for human-computer interaction. CRC Press, Boca Raton

Paternò F (2012) Model-based design and evaluation of interactive applications. Springer, Berlin

Pierce BC (1991) Basic category theory for computer scientists. MIT press, Cambridge

Pleuss A, Wollny S, Botterweck G (2013) Model-driven development and evolution of customized user interfaces. In: Proceedings of the 5th ACM SIGCHI symposium on engineering interactive computing systems, London

Reinecke K, Bernstein A (2011) Improving performance, perceived usability, and aesthetics with culturally adaptive user interfaces. ACM Transact Comput Hum Interact 18(2):1–29

Schürr A, Westfechtel B (1992) Graph grammars and graph rewriting systems. Technical report AIB 92-15, RWTH Aachen, Aachen

Stückrath J, Weyers B (2014a) 2014-01: lattice-extended coloured petri net rewriting for adaptable user interface models. Technical report, University of Duisburg-Essen, Duisburg

Stückrath J, Weyers B (2014b) Lattice-extended cpn rewriting for adaptable ui models. In: Proceedings of GT-VMT 2014 workshop, Grenoble

Weber M, Kindler E (2003) The Petri net markup language. In: Petri net technology for communication-based systems. Springer, Berlin

Weyers B (2012) Reconfiguration of user interface models for monitoring and control of human-computer systems. Dr, Hut, Munich

Weyers B (2015) Fill: formal description of executable and reconfigurable models of interactive systems. In: FoMHCI workshop in conjunction with EICS 2015, Aachen

Weyers B, Borisov N, Luther W (2014) Creation of adaptive user interfaces through reconfiguration of user interface models using an algorithmic rule generation approach. Int J Adv Intell Syst 7(1&2):302–336

Weyers B, Burkolter D, Kluge A, Luther W (2012) Formal modeling and reconfiguration of user interfaces for reduction of human error in failure handling of complex systems. Int J Hum Comput Interact 28(10):646–665

Weyers B, Luther W (2010) Formal modeling and reconfiguration of user interfaces. In: Proceedings of the international conference of the Chilean computer science society, Antofagasta

Weyers B, Luther W, Baloian N (2011) Interface creation and redesign techniques in collaborative learning scenarios. Future Gener Comput Syst 27(1):127–138

Part III
Analysis, Validation and Verification

Chapter 11
Learning Safe Interactions and Full-Control

Guillaume Maudoux, Charles Pecheur and Sébastien Combéfis

Abstract This chapter is concerned with the problem of learning how to interact safely with complex automated systems. With large systems, human–machine interaction errors like automation surprises are more likely to happen. Full-control mental models are formal system abstractions embedding the required information to completely control a system and avoid interaction surprises. They represent the internal system understanding that should be achieved by perfect operators. However, this concept provides no information about how operators should reach that level of competence. This work investigates the problem of splitting the teaching of full-control mental models into smaller independent learning units. These units each allow to control a subset of the system and can be learned incrementally to control more and more features of the system. This chapter explains how to formalize the learning process based on an operator that merges mental models. On that basis, we show how to generate a set of learning units with the required properties.

11.1 Introduction

The field of human–computer interaction (HCI) studies how humans interact with automated systems and seeks to improve the quality of these interactions. In HCI, formal methods allow to unambiguously describe behaviours of both humans and interactive systems. Given a formal description of humans and/or computer systems, it is possible to mechanically check properties on their interactions such as the *control* property, which we will describe later on. The automated verification of properties is called model checking. Model checking techniques can also be used to generate

G. Maudoux (✉) · C. Pecheur
Université catholique de Louvain, Louvain-la-Neuve, Belgium
e-mail: guillaume.maudoux@uclouvain.be

C. Pecheur
e-mail: charles.pecheur@uclouvain.be

S. Combéfis
École Centrale des Arts et Métiers, Brussels, Belgium
e-mail: s.combefis@ecam.be

B. Weyers et al. (eds.), *The Handbook of Formal Methods in Human-Computer Interaction*, Human-Computer Interaction Series,
DOI 10.1007/978-3-319-51838-1_11

models that satisfy desirable properties. In this chapter, we will use that capability to produce formal descriptions of the knowledge required to interact safely with a formally defined system.

Safe interactions are achieved when a system behaves in accordance with the user expectations. Any discrepancy between these expectations and the actual behaviour of the system is called an *automation surprise* (Sarter et al. 1997; Palmer 1995). To formalize our analysis of automation surprises, two models are used: one model describes the behaviour of the system and the other describes how the system is assumed to behave from the point of view of the user. The latter is called a *mental model* (Rushby 2002). A user can build a mental model of a system from training, by experimenting with the system or by reusing knowledge applicable to other similar systems. In this work, a user is assumed to behave according to her mental model. This assumption does not always hold as users may get distracted or make errors when manipulating a system, but we are not considering these kinds of faults here.

The *control* property describes the relation between a mental model and a system such that their interactions cannot lead to surprises. It is important to ensure that the mental model of the user allows control of the system in use. Or, less formally, that the user knows enough about the features she uses to avoid any surprise. To interact safely with a system, a user need not know all of its features. With most systems, knowing nothing about their behaviour is sufficient to never get surprised. These devices remain idle or turned off while waiting for a human to initiate the interaction. In the case of a plane at rest, for example, not interacting with it ensures that it will not behave unexpectedly. Some systems however require some knowledge in order to avoid surprises. Phones are an excellent example, as one could get surprised by an incoming ringtone if not warned that it may happen.

Generating mental models that ensure proper control provides a mean to train operators of critical systems. If the learning process ensures that their mental model controls the system, we can avoid automation surprises. In particular, new system features should be taught in such a way that the new mental model of the operators still ensures proper control of the system. This also means that operators must not interact with the system until the end of a learning phase and cannot perform their duties during that period.

Operators that have learnt all the possible behaviours of a system have built a full-control mental model. Such models have been defined by Combéfis and Pecheur (2009), and techniques to build minimal ones have been described in Combéfis and Pecheur (2011a, b). These mental models allow users to safely control all the features of a system. However, teaching a full-control mental model is impractical as it is equivalent to teaching all the features of the system at once, in one big step. For example, newly hired operators would be useless until they master the full complexity of the system. Large systems might even be too complex for one operator to manage. In that case, the operation of the system must be split in tasks dedicated to different operators and the generated full-control mental model cannot be used at all.

To be practical, learning processes should provide a set of learning units in the form of small, compatible mental models that can be combined incrementally into bigger models. Each intermediate mental model should ensure safe interactions with the system without necessarily describing all of its features. Learned sequentially, these units should increase the mental model of the operator until her mental model reaches full-control over the system. We will see that it is possible to decompose a full-control mental model into learning units units with such properties.

In this chapter, we first summarize the necessary concepts. Section 11.2 will describe *Labelled Transition Systems for Human Machine Interactions* (HMI-LTSs) as a specialization of *Labelled Transition Systems* (LTSs) and use these to formally define mental models, control, and full-control. In Sect. 11.3, we introduce a *merge* operator that describes the mental model resulting of learning two other mental models, and we argue that it is coherent with the intuition of learning a system. In Sect. 11.4, we explore the decomposition induced by the *merge* operator on HMI-LTSs. We show that some HMI-LTSs are too basic to be worth further decomposing, and that decomposing mental models into such basic elements is a finite process. We also provide an algorithm to automate this decomposition. Finally, we demonstrate that it is possible to generate a set of learning units with the desired properties by decomposing full-control mental models. This is presented with many examples in Sect. 11.5.

11.2 Background

In this section, we define HMI-LTSs, mental models, and the (full-)control property. We also formally introduce full-control mental models, a concept that lies at the intersection of these three notions. A deeper discussion of the background information provided in this section can be found in Combéfis (2013).

To formalize systems and mental models, we use *Labelled Transition Systems for Human–Machine Interactions* (HMI-LTSs) which are slightly modified *Labelled Transition Systems* (LTSs). An LTS is a state transition system where each transition has a *label*, also called *action*. LTSs interact with their environment based on this set of actions. Additionally, LTSs can have an internal τ action that cannot be observed by the environment. The representations of three different LTSs are shown in Figs. 11.1 and 11.2.

Definition 1 (*Labelled Transition System*) A labelled transition system (LTS) is a tuple $\langle S, \mathscr{L}, s_0, \rightarrow \rangle$ where S is a finite set of states, $\mathscr{L}$ is a finite set of labels representing visible actions, $s_0 \in S$ is the initial state and $\rightarrow \subseteq S \times (\mathscr{L} \cup \{\tau\}) \times S$ is the transition relation, where $\tau \notin \mathscr{L}$ is the label for the internal action.

The executions of LTSs can be observed from the environment via traces. An *execution* of an LTS is a sequence of transitions $s_0 \overset{a_1}{\rightarrow} s_1 \ldots s_{n-1} \overset{a_n}{\rightarrow} s_n$ where each $(s_{i-1}, a_i, s_i) \in \rightarrow$. It represents the system moving from state to state by firing transitions. A *trace* of an LTS is a sequence $\sigma = a_1, a_2, \ldots, a_n \in \mathscr{L}^{\omega}$ such that there exists an execution $s_0 \xrightarrow{\tau^* a_1 \tau^*} s_1 \cdots s_{n-1} \xrightarrow{\tau^* a_n \tau^*} s_n$. The notation $s \xrightarrow{\tau^* a \tau^*} s'$ represents a transitionlabelled a preceded and followed by any number of invisible τ transitions. Its

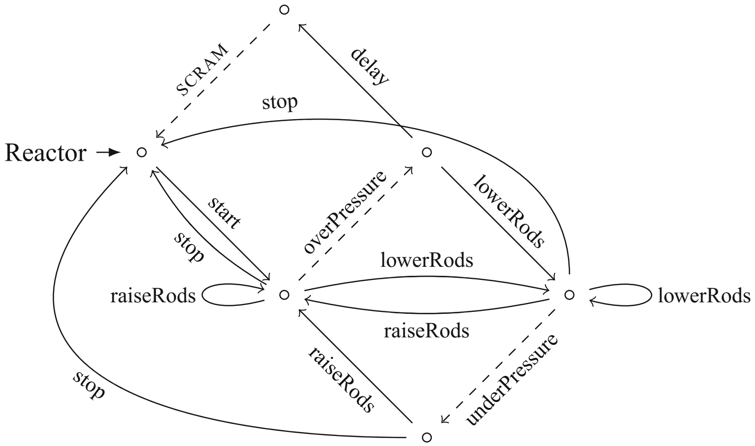


Fig. 11.1 HMI-LTS model of a reactor inspired from the nuclear power plant example (cf. Chap. 4). Observations are represented with a *dashed* edge. This model will be used as the main example through the remainder of this chapter

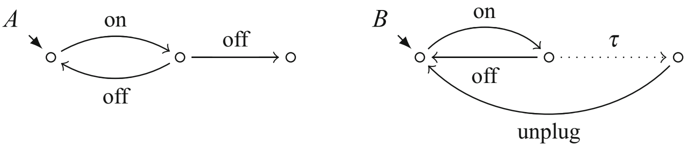


Fig. 11.2 Two examples of nondeterministic systems. *A* can be turned on then off at least once, but it is impossible to determine if it can be turned on again. *B* can be turned on and off, but it can also unobservably change to a state where the only way to restart it is to unplug it

only observable action is *a*. A trace is a sequence of visible actions that the system may produce in one of its executions. For example, the Reactor system of Fig. 11.1 can produce the trace “start, lowerRods, underPressure” and the trace “start, overPressure, delay, SCRAM” among infinitely many others.

To model interactions, we need to distinguish inputs from outputs. HMI-LTSs refine LTSs by distinguishing two kinds of actions, *commands* and *observations*. Like any I/O transition system, observations are uncontrollable outputs generated by the system and commands are controllable inputs. HMI-LTSs are very similar to LTS/IOs described by Tretmans (2008).

Definition 2 (*Human-Machine Interaction LTS*) A human-machine interaction labelled transition system (HMI-LTS) is a tuple $\langle S, \mathscr{L}^c, \mathscr{L}^o, s_0, \rightarrow\rangle$ where $\langle S, \mathscr{L}^c \cup \mathscr{L}^o, s_0, \rightarrow\rangle$ is a labelled transition system, $\mathscr{L}^c$ is a finite set of command labels and $\mathscr{L}^o$ is a finite set of observation labels. The two sets $\mathscr{L}^c$ and $\mathscr{L}^o$ are disjoint and the set of visible actions is $\mathscr{L} = \mathscr{L}^c \cup \mathscr{L}^o$.

HMI-LTSs are used to describe both systems and mental models. Mental models represent the knowledge an operator has about the system she controls. It is important to note that mental models do not represent the behaviour of a user, but the behaviour of a system as seen by a user. Therefore, a command in a mental model corresponds exactly to the same command on the system. The interactions between a system $\mathscr{S}$ and an operator behaving according to its mental model $\mathscr{M}$ are defined by the synchronous parallel composition $\mathscr{S} \parallel \mathscr{M}$. This distinguishes HMI-LTSs from LTS/IOs where inputs of the system must be synchronized on the outputs of the user and vice versa.

Notions on LTSs can be easily lifted to HMI-LTSs due to their high similarity. The set of states that can be reached from state s with an observable trace σ is represented as s **after** σ. This definition applies as is to LTSs and HMI-LTSs. We also use notations specific to HMI-LTSs. $A^c(s)$ (resp. $A^o(s)$) is the set of possible commands (resp. observations) of s. An action is possible in s if it is the first action of some trace starting at s.

An LTS is deterministic if $|s\,\textbf{after}\,\sigma| \leq 1$ for any σ. For example, the HMI-LTS A from Fig. 11.2 can be in two states after the trace "on, off" and is therefore not deterministic. Also, the HMI-LTS B has two possible actions in its middle state ('off' and 'unplug') because unplug can be the next visible action in executions firing the τ transition. It is of course also nondeterministic because after the "on" trace the system can be in two different states.

We want mental models to control systems without surprises. In particular, we want to avoid mental models that contain commands that are impossible on the system and to ignore observations that the system could produce. This motivates the introduction of the control property.

Definition 3 (*Control Property*) Given two HMI-LTSs $\mathscr{S} = \langle S_{\mathscr{S}}, \mathscr{L}^c, \mathscr{L}^o, s_{0\mathscr{S}}, \rightarrow_{\mathscr{S}} \rangle$ and $\mathscr{M} = \langle S_{\mathscr{M}}, \mathscr{L}^c, \mathscr{L}^o, s_{0\mathscr{M}}, \rightarrow_{\mathscr{M}} \rangle$, $\mathscr{M}$ controls $\mathscr{S}$ if $\mathscr{M}$ is deterministic and for all traces $\sigma \in \mathscr{L}^*$ such that $s_{\mathscr{S}} \in s_{0\mathscr{S}}$ **after** σ and $\{s_{\mathscr{M}}\} = s_{0\mathscr{M}}$ **after** σ:

$$A^c(s_{\mathscr{S}}) \supseteq A^c(s_{\mathscr{M}}) \text{ and } A^o(s_{\mathscr{S}}) \subseteq A^o(s_{\mathscr{M}}).$$

This definition is symmetric because it allows the mental model not to know the full set of available commands, and it also allows the system to produce fewer observations than expected by the mental model. From now on, this is the formal definition we refer to when we say that a mental model controls a system.

For a given system, there always exists a mental model that contains no commands and still allows control of the system. That mental model contains only the traces of observations available from the initial state and corresponds to the mental model needed by an agent to avoid surprises when not interacting with a system. Think back to the example of the phone given in the introduction. You need to know that your desk phone may ring even when you do not want to interact with it. Someone who ignores that fact will be surprised whenever the phone rings.

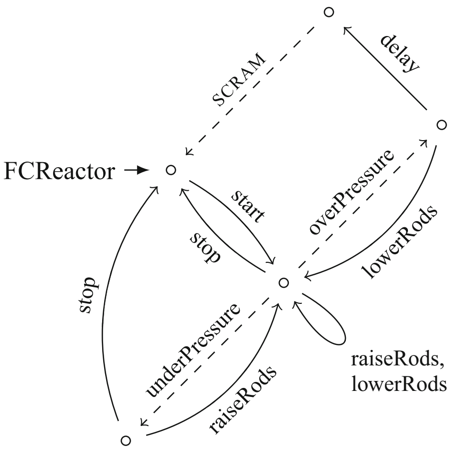


Fig. 11.3 The only minimal full-control mental model of the reactor system

We see that a mental model that controls a system does not necessarily explore the full range of possible behaviours of that system. When a mental model ensures control over a system and allows access to all the available commands of the system, we say that the model fully controls the system.

Definition 4 (*Full-Control Property*) Given two HMI-LTSs $\mathscr{S} = \langle S_{\mathscr{S}}, \mathscr{L}^c, \mathscr{L}^o, s_{0\mathscr{S}}, \rightarrow_{\mathscr{S}} \rangle$ and $\mathscr{M} = \langle S_{\mathscr{M}}, \mathscr{L}^c, \mathscr{L}^o, s_{0\mathscr{M}}, \rightarrow_{\mathscr{M}} \rangle$, $\mathscr{M}$ is a full-control mental model for $\mathscr{S}$, which is denoted $\mathscr{M}\ \mathbf{fc}\ \mathscr{S}$, if $\mathscr{M}$ is deterministic and for all traces $\sigma \in \mathscr{L}^*$ and for all $s_{\mathscr{S}} \in (s_{0\mathscr{S}}\ \mathbf{after}\ \sigma)$ we have

$$A^c(s_{\mathscr{S}}) = A^c(s_{\mathscr{M}}) \text{ and } A^o(s_{\mathscr{S}}) \subseteq A^o(s_{\mathscr{M}}).$$

where $\{s_{\mathscr{M}}\} = (s_{0\mathscr{M}}\ \mathbf{after}\ \sigma)$ is the only state reached in $\mathscr{M}$ by the trace σ.

A full-control mental model is therefore a deterministic HMI-LTS representing the required information for an operator to interact with a system to the full extent of its possibilities, and without surprises. Full-control mental models are minimal if they have a minimal number of states compared to other full-control mental models of the same system. Also, being *full-control deterministic* is a property shared by all the systems for which there exists a full-control mental model (Combéfis and Pecheur 2009). The property states that nondeterminism in the system does not impact controllability. Different algorithms exist to generate such models (Combéfis and Pecheur 2009; Combéfis et al. 2011a; Delp et al. 2013).

Figure 11.3 presents FCReactor, the only minimal full-control mental model of the Reactor system from Fig. 11.1. The two active states with no pressure warning have been merged as they are undistinguishable from a control point of view: they allow exactly the same commands, and their observations are compatible.

Minimal full-control mental models are important because they represent the minimal knowledge that a perfect operator should master. Compact training material and user guides should therefore describe a minimal full-control mental model.

11.3 Modelling the Learning Process with the Merge Operator

While minimal full-control mental models are perfect in terms of control, they are inefficient when training operators as they require to be completely mastered before using a system. But to optimize this process, we need to describe how the mental model of a user can be augmented with a learning unit. In this section, we define the new merge operator that combines two mental models into a broader one, and we claim that this operator is a natural way to encode the learning process.

The merge of two HMI-LTSs is obtained by superimposing their graphs and merging identical paths. This is a kind of lazy choice as the final behaviour does not commit to behave like the first or the second operand until a decision is required. The result may even alternate between the behaviour of its two operands. As the definition of the merge operator does not rely on observations and commands, it can easily be generalized to LTSs. An example of the action of the merge operator is given in Fig. 11.4.

Definition 5 (*Merge*)
The *merge* of two deterministic HMI-LTSs $A = \langle S_A, \mathscr{L}^c_A, \mathscr{L}^o_A, s_{0A}, \rightarrow_A \rangle$ and $B = \langle S_B, \mathscr{L}^c_B, \mathscr{L}^o_B, s_{0B}, \rightarrow_B \rangle$, denoted $A \oplus B$, is an HMI-LTS $\langle S, \mathscr{L}^c_A \cup \mathscr{L}^c_B, \mathscr{L}^o_A \cup \mathscr{L}^o_B, s_0, \rightarrow \rangle$ where

1. $S \in \mathscr{P}(S_A \uplus S_B)$ is the set partition defined by the equivalence relation $\sim$.
2. $\sim = \cup_{i=0}^{\infty} \sim_i$ is the equivalence relation on $S_A \uplus S_B$ such that

   a. s_{0A} and s_{0B} are the only equivalent nodes in $\sim_0$;
   b. $\sim_k \subset \sim_{k+1}$ and

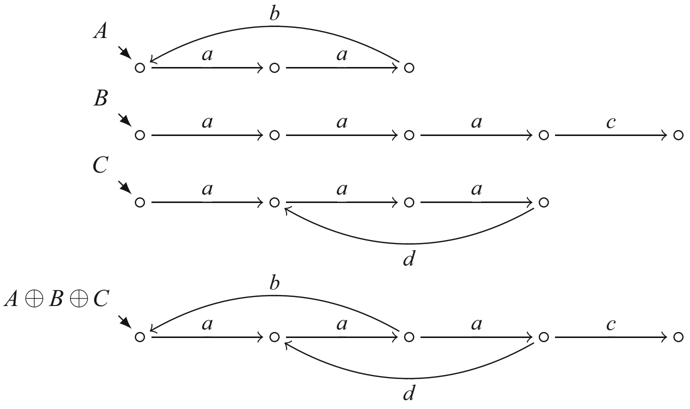


Fig. 11.4 Example of the merge operation on three HMI-LTSs

c. $\sim_{k+1}$ is the finest equivalence relation such that $m' \sim_{k+1} n'$ if there exists $m \sim_k n$ and a such that $m \xrightarrow{a}_A m'$ and $n \xrightarrow{a}_B n'$.

3. $\rightarrow$ is the set of transitions $[s]_\sim \xrightarrow{a} [s']_\sim$ such that either $s \xrightarrow{a}_A s'$ or $s \xrightarrow{a}_B s'$.

In this definition, S is always well defined. While the equivalence relation $\sim$ is an infinite union, we can see that the intermediate equivalence relations are monotonically increasing as they embed the previous one and can only add new equivalent states, not remove them. If no states are added at some step, then a fixpoint is reached. And this must happen within a finite number of steps as it must end when all the states are equivalent (in the worst case).

From this definition, we can also see that the merge of two deterministic HMI-LTS is unique. Were it not the case, there would be two different ways to build $\sim_{k+1}$ for some k. But each $\sim_{k+1}$ is uniquely defined with respect to $\sim_k$, and $\sim_0$ is uniquely defined. Therefore $\sim$ must be unique.

The example in Fig. 11.4 uses the fact that the merge operator is associative to write $A \oplus B \oplus C$ instead of $(A \oplus B) \oplus C$ or $A \oplus (B \oplus C)$. Associativity is tedious to prove because we need to show that $(A \oplus B) \oplus C$ is isomorphic to $A \oplus (B \oplus C)$. Instead, we draw attention to the extensibility of the definition to more than two models. It only requires minor adjustments to define the merge of three or even n HMI-LTSs. The operator is also commutative. This property may be assumed from the symmetry of the definition.

We can show that the result of merging two deterministic HMI-LTSs is deterministic. Indeed, as the two operands of the merge are deterministic, they cannot contain τ transitions and so their merge is free of τ transitions too. Neither can the result contain two transitions with the same label leaving the same state. Let us assume that the result contains two transitions outgoing from the same state, with the same label, and leading to different states. By the definition of $\rightarrow$, this means that there exists $(m, a, m') \in \rightarrow_A$ and $(n, a, n') \in \rightarrow_B$ such that $m \sim n$ and $m' \nsim n'$, which violates the recursive property on $\sim$. Also, the resulting HMI-LTS can contain no τ transitions and no fork where a transition with the same label leads to two different states. These two conditions are sufficient to prove that the merge is deterministic.

The HMI-LTS $A \oplus B$ can switch its behaviour from A to B provided A can reach a state that was merged with a state of B. This conversely holds from B to A. If the HMI-LTS can switch from A to B and from B to A, then it can alternate its behaviour arbitrarily often. We can see that this operator is different from the traditional choice operator because it is more than the union of the traces. It can build complex behaviours from two simple models. In Fig. 11.4, we can see that the trace $a, a, a, d, a, b, a, a, a, c$ is not possible on the different models but is valid on their merge.

The merge operator is useful because the set of traces of a merge is always larger than or equal to the union of the traces of the merged transition systems. This means that the possible behaviours of a merge can be richer than the union of the behaviours of its operands. This is needed to ensure that the decomposition of a big system is a small set of small systems. Indeed, if the behaviour of a merge was exactly the sum

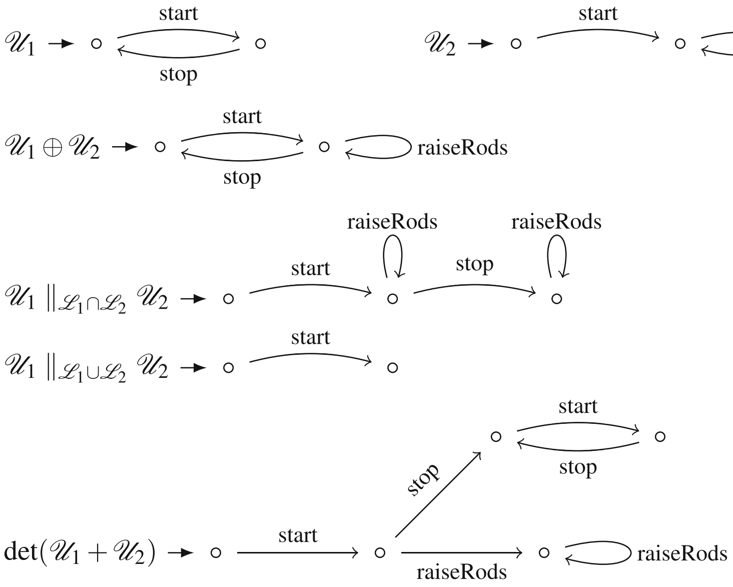

$\mathscr{U}_1$
start
stop
$\mathscr{U}_2$
start
raiseRods
$\mathscr{U}_1 \oplus \mathscr{U}_2$
start
stop
raiseRods
$\mathscr{U}_1 \parallel_{\mathscr{L}_1 \cap \mathscr{L}_2} \mathscr{U}_2$
raiseRods
raiseRods
start
stop
$\mathscr{U}_1 \parallel_{\mathscr{L}_1 \cup \mathscr{L}_2} \mathscr{U}_2$
start
start
stop
stop
$\det(\mathscr{U}_1 + \mathscr{U}_2)$
start
raiseRods
raiseRods

Fig. 11.5 The merge operation compared to the parallel synchronization (||) and the choice (+) operators on (HMI-)LTSs. The result of the choice between $\mathscr{U}_1$ and $\mathscr{U}_2$ has been determinised for readability

of the behaviour of its operands, then learning a system would reduce to enumerating all the possible traces on it.

Figure 11.5 shows the effect of three operators on two simple models. We can see that the merged model can perform the actions "stop" and "raiseRods" infinitely often. This corresponds to the combined knowledge that the system can be started and stopped infinitely often, and that when started, the rods can be raised as many times as wished.

By comparison, the synchronous parallel composition requires the two operands to be executed in parallel, and to be synchronized on some actions. Usually, systems are synchronized on their common alphabet. This means that $\mathscr{U}_1 \parallel_{\mathscr{L}_1 \cap \mathscr{L}_2} \mathscr{U}_2$ cannot execute "start" more than once because $\mathscr{U}_2$ can only do it once. Unsynchronized actions like "raiseRods" can however fire at any time, even when the reactor is shut down. These two aspects are not desirable when modelling combined knowledge.

If we synchronize the two models on the unions of their alphabets, then $\mathscr{U}_1 \parallel_{\mathscr{L}_1 \cup \mathscr{L}_2} \mathscr{U}_2$ cannot perform any "stop" action because that action $\mathscr{U}_2$. This case is even worse for modelling knowledge increase as it removes existing known behaviours from their combined learning.

Finally, the choice operator does not allow to knowledge of multiple mental models to be combined either. It forms a new model that can use the knowledge of either operand, but that prevents any interaction between the two. This is like learning addition and multiplication but not being able to use both in the same calculation.

With these examples, we have shown that existing operators are inappropriate for our mathematical model of learning. The new merge operator remedies to these shortcomings. If two HMI-LTSs each represent some knowledge about a system, then their merge represents the combined knowledge of an operator who knows these two facts.

Furthermore, the merge operator is consistent with the interpretation of HMI-LTSs as scenarios. When a scenario loops, the system is assumed to have returned in a state strictly equivalent to the initial one. In particular, the scenario is assumed to be repeatable infinitely often unless explicitly stated. When a learning unit loops to a given state, it means that state is completely equivalent to the initial one for controllability purposes.

While there is no way to prove that the merge operator is perfect, we have provided examples and intuition on why it is a good way to encode how mental models grow during the learning of a system.

11.4 Basic Learning Units

Within our formal theory of knowledge and learning, we are now able to split a learning objective into smaller elements that can be learned independently. Decomposing a full-control mental model into independent elements amounts to find a set of mental models such that their merge by the merge operator is exactly that full-control mental model.

To define a good decomposition, we first need an order relation to tell if some sub-elements form a valid decomposition. We need our decomposition to split a model into elements that can be merged back into the original model, but we also need the elements to be smaller than the original model. Were it not the case, the decomposition would not be well-founded: systems could be decomposed forever into other systems. This goes against the idea of a decomposition into simpler elements.

In this section, we first explore the order induced by the merge operator on HMI-LTSs. We then show that this order is not well-founded and how an order based on the size of the graphs fixes it. Finally, we introduce *basic* learning units, small learning units that cannot be decomposed into smaller elements.

The merge operator naturally defines a partial order on the HMI-LTSs. The merge order is such that $A \leq_{\oplus} B$ if and only if $A \oplus B$ is isomorphic to B, which we denote $A \oplus B \simeq B$. The strict partial order relation also requires A to be different from B (i.e. not isomorphic to B). Figure 11.6 shows four HMI-LTSs ordered according to $<_{\oplus}$. The merge order captures the idea that B has more behaviours than A because $A \leq_{\oplus} B$ implies that Traces(A) $\subseteq$ Traces(B).

Furthermore, due to the definition of the merge order, the set of deterministic HMI-LTSs forms a join-semilattice: any two HMI-LTSs A and B are (upper) bounded by $A \oplus B$. In a such a lattice, the decomposition is performed by finding two elements strictly smaller than an HMI-LTS and such that their upper bound is exactly that HMI-LTS. However, this lattice has the undesirable property of allowing infinite

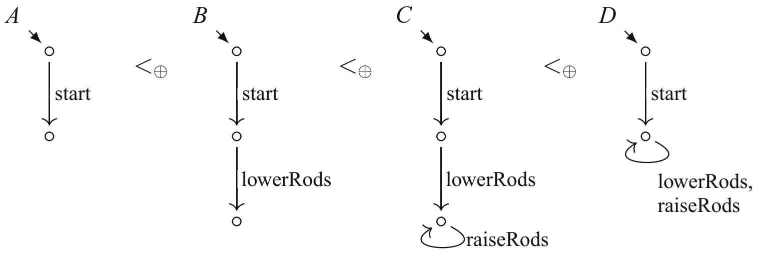


Fig. 11.6 Illustration of the merge-order relation on HMI-LTSs. For example, we have $C <_\oplus D$ because $C \oplus D = D$ and $C \neq D$

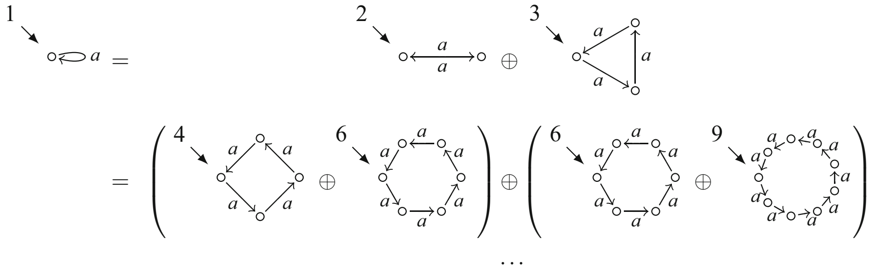


Fig. 11.7 A decomposition based on the merge order alone can lead to infinite decompositions into HMI-LTSs with increasing size

decomposition chains. As illustrated in Fig. 11.7, we can see that a simple loop can be decomposed into the merge of a 2- and a 3-loop, which can in turn be decomposed into the merge of a 4- and 6-loop, and a 6- and 9-loop, respectively, and so forth.

To encode the fact that a decomposition should produce simpler models than the model it comes from, we define the learning order. It restricts the merge order with the constraint that smaller models must have a lower number of states. Ties are broken based on the number of edges. This structural size order is denoted $\leq$.

Definition 6 (*Learning order*) A deterministic HMI-LTS A is said to be smaller than an HMI-LTS B according to the learning order if and only if $A \leq B$ and $A \leq_\oplus B$. This is denoted $A \leq_{\text{learn}} B$.

A decomposition will stop when we reach a model not worth splitting, which happens when we cannot find two strictly smaller models (in the sense of the learning order) that merge into the current model. We say that such HMI-LTSs are *basic*.

Definition 7 (*Basic HMI-LTS*) A deterministic HMI-LTS M is *basic* if there does not exist two HMI-LTSs A and B such that $A <_{\text{learn}} M$, $B <_{\text{learn}} M$ and $A \oplus B = M$.

It turns out that such basic HMI-LTS take the form of single sequences, single loops, lassos or tulips as shown in Fig. 11.8. Loops and sequences can be seen as

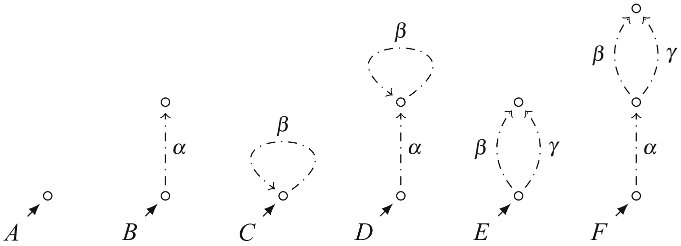


Fig. 11.8 Different shapes of basic HMI-LTSs. They can be *A* empty, *B* sequences, *C* loops, *D* lassos and *E*, *F* tulips with and without stem. *Dotted lines* represent any oriented sequence of states and transitions. All these shapes are degenerated tulips where action sequences α, β and γ can be empty

degenerated lassos with no stem or no loop. The fully degenerated lasso is the HMI-LTS with no transitions at all. Finally, a tulip is a branching HMI-LTS where the two branches reunite in the last state. Like lassos, they may have no stem. When they are comparable, lassos and tulips are always strictly greater than sequences. Lassos and tulips are never comparable.

Any finite deterministic HMI-LTS can be decomposed into a finite set of basic HMI-LTS. This arises from the fact that any HMI-LTS is the merge of a basic HMI-LTS and another HMI-LTS strictly smaller than the previous one. Were it not the case, that HMI-LTS would be basic itself. By induction on the remaining HMI-LTS, we show that it eventually reduces to a basic HMI-LTS after a finite number of basic HMI-LTS removal. All these basic elements form a set that we call the decomposition of the HMI-LTS. This algorithm is shown with the FCReactor model on Fig. 11.9 and formally defined hereafter.

Algorithm 1 formalizes the enumeration of the basic units forming the decomposition of a mental model. At line 5, this algorithm performs an unspecified exploration of the graph, edge by edge. It could be a depth-first search, a breadth-first search or any other exploration strategy. The *pre* mapping is used to build a spanning tree. The algorithm looks for edges that are outside the spanning tree and removes them permanently by calling the "extract_unit" procedure defined in Algorithm 2. For each such edge, a basic HMI-LTS is extracted. A special case is added in line 13 to handle states with only one adjacent edge. Such states are part of a basic sequence that does not contain edges outside of the spanning tree and a basic sequence must be extracted at that place. That way, all the paths in the graph are covered by some basic HMI-LTS, and basic HMI-LTSs are disjoint because they contain at least one edge that no other unit contains.

Algorithm 2 describes "extract_unit," which splits a model into a basic HMI-LTS and a smaller model by removing a given edge from the model. For correctness and efficiency, it also removes all the edges that are completely described by the extracted basic unit. That way, the resulting model $\mathcal{M}$' is the minimal (size-wise) model such that $\mathcal{M}' \oplus \mathcal{U} = \mathcal{M}$. Note that *pre* is guaranteed to be defined for the nodes used at line 6 because it forms a partial spanning tree spanning at least to x

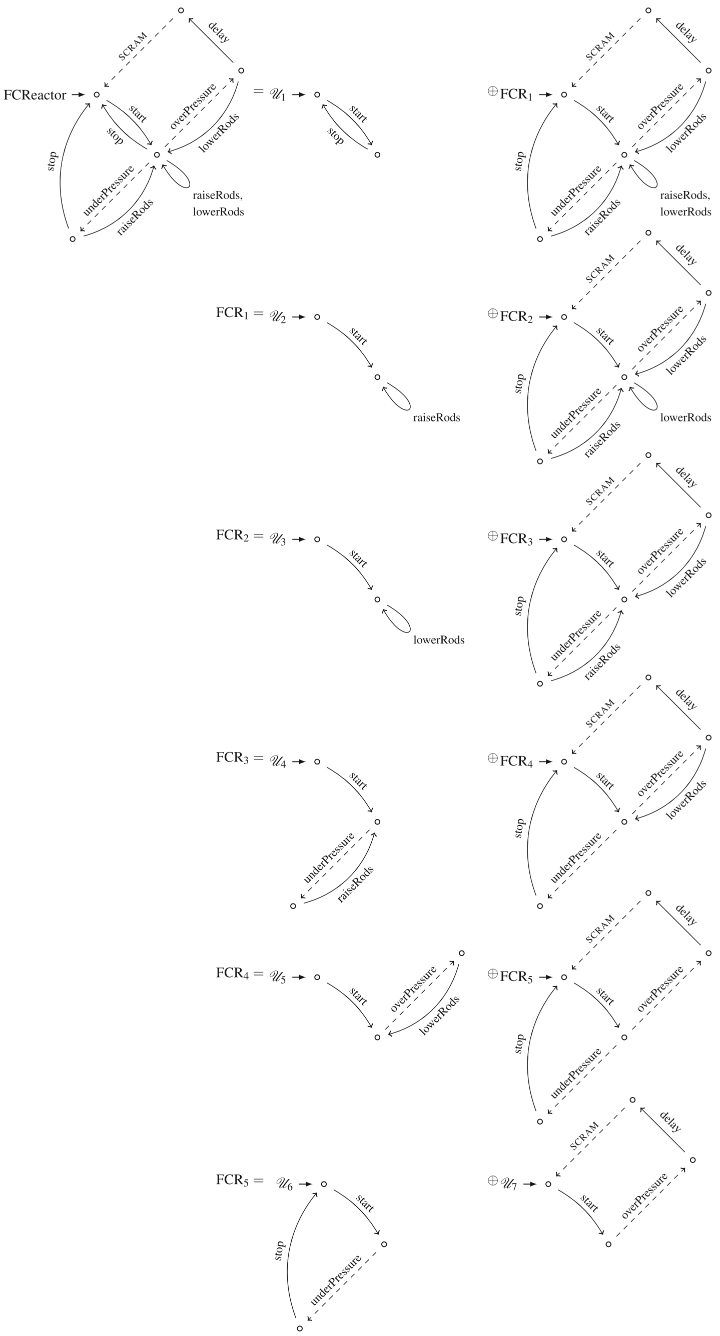


Fig. 11.9 Illustration of the decomposition algorithm, where basic HMI-LTSs are extracted one by one until there is nothing more to extract

and x' by construction in Algorithm 1. As “extract_unit” traverses that tree upwards from x and x', it cannot encounter undefined values of *pre*. The units extracted by the algorithm are subgraphs of the initial model. They contain a subset of the edges of that model and the corresponding subset of nodes.

Algorithm 1 Basic HMI-LTSs enumeration

Require: $\mathscr{M} = \langle S, \mathscr{L}^c, \mathscr{L}^o, s_0, \rightarrow \rangle$, a minimal deterministic HMI-LTS.
Ensure: $U = \{u_1, u_2, \ldots, u_n\}$, a minimal decomposition of $\mathscr{M}$ into basic HMI-LTSs.

```
$U := \emptyset$                                          ▷ Initialise the decomposition with an empty set
$E := \{(m, v, m') \in \rightarrow \mid m = s_0\}$                  ▷ The set of edges yet to explore,
                                                          ▷ initialised with the edges starting at $s_0$
$pre := [\bot, \ldots, \bot]$
$pre[s_0] := \top$

while $\exists (n, a, n') \in E \cap \rightarrow$ do
    $E := E \setminus \{(n, a, n')\}$
    if $pre[n'] \neq \bot$ then                            ▷ Extract a tulip or a lasso
        $(\mathscr{M}, u) :=$ extract_unit($\mathscr{M}, pre, (n, a, n')$)
        $U := U \cup \{u\}$
    else
        $pre[n'] := (n, a, n')$
        $E := E \cup \{(m, v, m') \in \rightarrow \mid m = n'\}$
        if deg($n', \rightarrow$) = 1 then                     ▷ Extract a single path
            $(\mathscr{M}, u) :=$ extract_unit($\mathscr{M}, pre, (n, a, n')$)
            $U := U \cup \{u\}$
        end if
    end if
end while
return $U$
```

A decomposition is nonredundant if it does not contain two comparable elements. If it was the case, one of these elements could be further decomposed. The decomposition algorithm sketched in Fig. 11.9 always produces a nonredundant decomposition because each basic HMI-LTS contains actions that were not part of the previously removed basic elements, and that are removed with it. For example, the decomposition of the FCReactor system into $\{\mathscr{U}_1, \mathscr{U}_2, \mathscr{U}_3, \mathscr{U}_4, \mathscr{U}_5, \mathscr{U}_6, \mathscr{U}_7\}$ as shown in Fig. 11.9 is nonredundant.

A decomposition is minimal if no other decomposition of the same HMI-LTS contains fewer basic elements. The size of a minimal decomposition is called the *complexity* of an HMI-LTS. Minimal decompositions of the FCReactor system contain exactly seven elements, so the complexity of that model is 7.

We now know how to decompose an HMI-LTS into basic elements, and that decomposition gives us a measure of the complexity of that HMI-LTS.

Algorithm 2 Basic units extraction

```
Require: $\mathscr{M}$, a minimal deterministic HMI-LTS;
Require: $pre : S \to S \times (\mathscr{L}^c \cup \mathscr{L}^o) \times S : n \mapsto pre[n]$, mapping nodes to preceding edges;
Require: $(n, a, n')$ an edge to extract.
Ensure: $\mathscr{U}$, a basic HMI-LTS;
Ensure: $\mathscr{M}'$, an HMI-LTS strictly smaller than $\mathscr{M}$ and such that $u \oplus \mathscr{M}' = \mathscr{M}$.

  function EXTRACT_UNIT($\mathscr{M} = \langle S, \mathscr{L}^c, \mathscr{L}^o, s_0, \to \rangle, pre, (n, a, n')$)
      $(\to_{\mathscr{U}}) := \{(n, a, n')\}$
      $(\to) := (\to) \setminus \{(n, a, n')\}$

      for $x := n, n'$ do
          while $pre[x] \neq \top$ do
              $(x, v, x') := pre[x]$
              $(\to_{\mathscr{U}}) := (\to_{\mathscr{U}}) \cup \{(x, v, x')\}$
              if $\deg(x', \to) = 1$ then
                  $(\to) := (\to) \setminus \{(x, v, x')\}$
              end if
          end while
      end for

      $S_{\mathscr{U}} := \{n \in S \mid \exists v, x. (n, v, x) \in \to_{\mathscr{U}} \vee (x, v, n) \in \to_{\mathscr{U}}\}$
      $\mathscr{U} := \langle S_u, \mathscr{L}^c, \mathscr{L}^o, s_0, \to_{\mathscr{U}} \rangle$
      $S' := \{s_0\} \cup \{n \in S \mid \exists v, x. (n, v, x) \in \to \vee (x, v, n) \in \to\}$
      $\mathscr{M}' := \langle S', \mathscr{L}^c, \mathscr{L}^o, s_0, \to \rangle$
      return $(\mathscr{M}', u)$
  end function
```

11.5 How to Teach Full-Control

In this section, we show that it is possible to build a set of learning units such that each unit controls a given model, and such that all units can be combined into a full-control mental model.

The main idea is to decompose a full-control mental model of the system into basic subgraphs. It appears that basic subgraphs can be completed to form learning units that can control the system. This means that the completed basic subgraphs of a full-control mental model of a system form a set of independent, compatible mental models that can be merged to reproduce the behaviour of the full-control mental model.

Given two deterministic subgraphs $\mathscr{U}$ and $\mathscr{U}'$ of a deterministic HMI-LTS $\mathscr{M}$, we have the property that their merge $\mathscr{U} \oplus \mathscr{U}'$ is isomorphic to a subgraph of $\mathscr{M}$. This can be seen from the fact that $\oplus$ merges states that can be reached with the same traces, and that these states must correspond to exactly one state of $\mathscr{M}$, as $\mathscr{M}$ is deterministic.

Starting from a full-control mental model $\mathscr{M}$ of a system $\mathscr{S}$, we can decompose it into a set of basic HMI-LTSs. However, these basic HMI-LTSs do not necessarily control $\mathscr{S}$. To achieve this property, they need to be completed with respect to observations. This is sufficient because a mental model that controls a system must accept all the observations of that system but is allowed to ignore commands.

If we call $\rightarrow^o_S$ the transition relation of $\mathscr{S}$ restricted to observations, then any subgraph of a full-control mental model can be completed with $\rightarrow^o_S$ in order to control $\mathscr{S}$. Of course, only the connected component reachable from the initial state should be kept after the completion. In particular, any *basic* subgraph of a mental full-control mental model can be completed in order to control $\mathscr{S}$.

Definition 8 (*Observation Completion*)
Given an HMI-LTS $\mathscr{M} = \langle S, \mathscr{L}^c, \mathscr{L}^o, s_0, \rightarrow^c \cup \rightarrow^o \rangle$ and one subgraph $\mathscr{U} = \langle S_{\mathscr{U}}, \mathscr{L}^c, \mathscr{L}^o, s_0, \rightarrow_{\mathscr{U}} \rangle$ of $\mathscr{M}$, the *observation completion* of $\mathscr{U}$ is an HMI-LTS $\mathscr{U}'$ such that $\mathscr{U}'$ is the connected component of $\langle S, \mathscr{L}^c, \mathscr{L}^o, s_0, \rightarrow_{\mathscr{U}} \cup \rightarrow^o \rangle$ reachable from s_0.

Figure 11.10 shows the completion of the basic HMI-LTSs from Fig. 11.9. Each model is completed with reachable observations from FCReactor. The interpretation

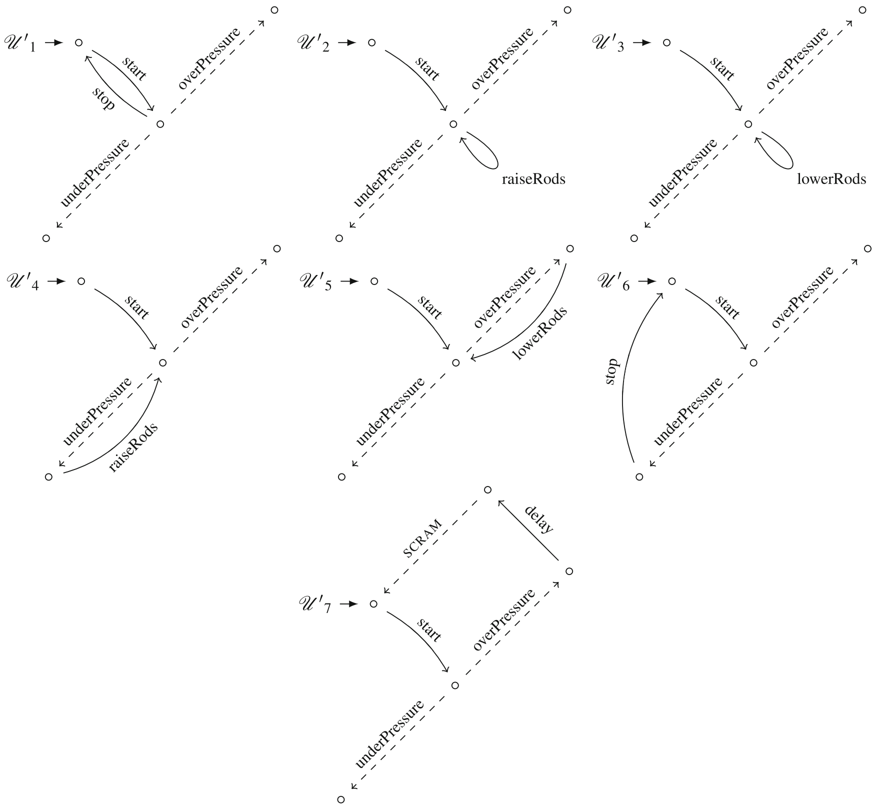


Fig. 11.10 The observation completion of the basic HMI-LTSs from Fig. 11.9 with respect to the only minimal full-control mental model of reactor, which is FCReactor

of $\mathscr{U}_1'$ is that the reactor can be turned on and off again, but that the operator needs to know that, when turned on, the reactor may emit warnings about the pressure. If a pressure warning event happens, the operator will not be surprised. She would however be unable to further operate the system. To unblock the situation, she could, for example, read the user manual to improve her knowledge of the system or ask a more experienced user.

The astute reader will have noticed that users need not to know about the delay transition. This is because the delay has been modelled as a command. If modelled as the observation that some time has elapsed, then no operator would be able to prevent the system to enter SCRAM. More advanced techniques need to be used to model passing time as a partially (un)controllable event on HMI-LTSs.

The observation completion of any subgraph of a full-control mental model $\mathscr{M}$ controls the intended system. Indeed, such a completed subgraph cannot prevent observations from occurring as the full-control mental model does not, and the completed graph has all the observations from the system. In particular, the observation completion of basic subgraphs of full-control mental models of a system $\mathscr{S}$ control that system $\mathscr{S}$. These elements also have the nice property of merging into completed subgraphs of $\mathscr{M}$ that themselves have control over $\mathscr{S}$.

Definition 9 (*Basic Learning Unit*)
Given a full-control mental model $\mathscr{M} = \langle S, \mathscr{L}^c, \mathscr{L}^o, s_0, \rightarrow \rangle$ where $\rightarrow = \rightarrow^c_{\mathscr{M}} \cup \rightarrow^o_{\mathscr{M}}$, a *basic learning unit* is a mental model $\mathscr{U} = \langle S_{\mathscr{U}}, \mathscr{L}^c, \mathscr{L}^o, s_0, \rightarrow_T \rangle$ where $\rightarrow_T$ is the connected component of $\rightarrow^o_{\mathscr{M}} \cup \rightarrow_b$ containing s_0 and $\langle S_{\mathscr{U}}, \mathscr{L}^c, \mathscr{L}^o, s_0, \rightarrow_b \rangle$ is a basic subgraph of $\mathscr{M}$.

With this definition, we can state that any full-control, deterministic (fc-deterministic) system is fully controlled by the merge of a set of basic learning units.

Theorem 1 (Decomposition) *Any finite fc-deterministic HMI-LTS $\mathscr{S}$ can be decomposed into a finite set $T = \{\mathscr{U}_1, \mathscr{U}_2, \dots \mathscr{U}_n\}$ of basic learning units such that*

- *each $\mathscr{U}_i$ controls $\mathscr{S}$;*
- *for each subset $I \subset \{1, 2, \dots n\}$ of indices, the partial merge $\bigoplus_{i \in I} \mathscr{U}_i$ of elements of T controls $\mathscr{S}$; and*
- *the complete merge $\bigoplus_{i=1}^{n} \mathscr{U}_i$ has full-control over $\mathscr{S}$.*

Proof By definition, any fc-deterministic HMI-LTS $\mathscr{S}$ has at least one minimal full-control mental model $\mathscr{M}$. We have shown that such a full-control mental model can be decomposed into a finite set of basic learning units that are completed basic subgraphs. Because they are completed subgraphs, these elements and any partial merge of these elements have control over $\mathscr{S}$. As the elements are the completion of the decomposition of $\mathscr{M}$ into basic HMI-LTS, their full merge will be exactly $\mathscr{M}$, and therefore fully controls $\mathscr{S}$. This proves that there exists a decomposition of $\mathscr{S}$ meeting the required properties.

The decomposition of an fc-deterministic system into a set of basic learning units may not be unique. Indeed, there may exist more than one minimal full-control mental model, and each full-control mental model may have multiple decompositions into basic learning units. Nevertheless, the minimal number of basic units required to decompose a mental model gives a measure of its complexity. We define the learning complexity of a system as the size of the smallest set of basic learning units that can be merged into a full-control mental model of that system. This metric measures the number of small learning units that an operator needs to learn before being able to control all the features of the system. This is different from the complexity of a system as defined at the end of Sect. 11.3 because the observation completion of two different basic units may turn to be the same basic learning unit. This metric is different from both the number of states and the number of transitions, which are the most common measures of complexity for transition systems.

11.6 Related Work

The idea of generating user manuals from formal specifications has been widely explored. Thimbleby and Ladkin (1995, 1996) provided a way to derive a complete description of system features by enumerating the sequences of actions to reach each state. Based on a formal description of a fax machine in the Prolog language, they generated a complete user manual describing how to perform every possible action. The fax machine is described as a tree of possible commands annotated with the information displayed in each node. A skeleton user manual is then generated by enumerating traces to each state of the system. The trace is separated into a sequence of actions and another sequence of observations. Finally, a technical author is required to turn the skeleton into a natural language. By comparison, our work gives hints on which elements should be described. Where Thimbleby and Ladkin describe one trace to each state, we propose to split the learning into stand-alone compatible units.

More recently, Delp et al. (2013) have implemented a system to generate a complete description of a system's formal model. Spichkova et al. (2014) presented a tool to maintain and update the technical documentation of a system based on its formal model. All of these authors have focussed their efforts in generating a complete description of the system. This work derives learning units from an ideal mental model and therefore tries to teach users a good mental model and not the full system itself.

Interestingly, user tasks model has been used by Kieras and Polson (1985) to compute the user complexity of a system. Each task is formally defined as a generalized transition network, and the complexity is measured by the number of states, the number of keystroke (their system's commands) and other direct metrics on the transition network. Further investigations are required to compare these approaches to the learning complexity defined here.

In this work, we used HMI-LTSs to model both systems and user mental models. This formalism is the result of research performed by Combéfis et al. (2011a) to detect and avoid surprises during interaction with a system. These tools are then used to define and verify the full-control property. Many authors have provided a formal description of systems and verified properties on it, but few model the mental model of users explicitly. Among them are Blandford et al. (2011) and Buth (2004). Buth checks the trace equivalence of a system and a mental model to ensure that both agree on the possible sequences of events. In contrast, the full-control property does not require the system to accept all the observations expected by the user. Buth's works can be considered an extension of Rushby (2002), where the user and the system are modelled separately but properties are verified on the single model of their interactions. The framework described by Bolton et al. (2008) uses a similar setup where erroneous mental models are derived from a correct one and combined with a model of the interface. The resulting model is then checked for errors that outline a vulnerability of the system to the simulated error. Finally, Campos et al. (2004) use mental models in the form of user tasks to check advanced properties on their systems.

The idea to generate mental models from system models has been explored separately and conjointly by Heymann and Degani (2007), Combéfis and Pecheur (2009). In both cases, the generation is constrained by a validity property to ensure that the mental model preserves desirable properties when interacting with the system and by a minimality property to ensure that the resulting models are efficient. Various properties can be checked on formal models in addition to full-control. Campos and Harrison (2008) define generic usability properties that could be used to generate better mental models.

The concept of mental model itself is not new. Carroll and Olson (1987) already proposed generalized transition networks as a formalism for mental models. They outlined the difference between prescriptive and descriptive mental models. Descriptive mental models represent actual users and can therefore only be checked against properties. Normative mental models are generated to verify these properties and can be used as a description of how users should behave. This is the kind of mental models that we need to use to describe what we want to teach to operators. Staggers and Norcio (1993) make a clear distinction between the conceptual model of the target system, the interface (or image) of that system, the actual mental model of the user and the scientist's conceptualization of that mental model. The power of HMI-LTSs is that we can use them to model the actual system restricted to its interface and the normative mental models of the users.

11.7 Conclusion

In Sect. 11.2, we described HMI-LTS and how they can be used to model interactive systems. We formally defined the control property and its full-control extension. These properties are defined on two HMI-LTSs representing a system interface and a descriptive mental model of a user. These properties are important because they are used to guide the generation of ideal, normative mental models for human operators.

Based on these tools, we then defined the merge operation that represents how a human augments its mental model by learning new mental models. We have provided some evidence that this operator is a natural way to formalize the learning process. We have also outlined the properties of the merge operator and the lattice structure it induces on HLI-LTSs.

Finally, we have shown how full-control mental models can be decomposed into basic learning units. These basic units have the desired properties of independence and minimality, and each has proper control over the full-control mental model and hence over the system. With this decomposition, we have defined a measure of the complexity of learning an interactive system.

Of course, there remains a lot of work to show how this theory relates to existing training material. For example, this works could be used to verify the modularity of system design by detecting irreducible large components. We could also investigate how the structure of existing user manual relates with sets of basic learning units, and how basic learning units can help generating such manuals. This theory opens the way towards formal analysis of training material.

References

Blandford A, Cauchi A, Curzon P, Eslambolchilar P, Furniss D, Gimblett A, Huang H, Lee P, Li Y, Masci P, Oladimeji P, Rajkomar A, Rukšėnas R, Thimbleby H (2011) Comparing actual practice and user manuals: a case study based on programmable infusion pumps. In: Proceedings of the 1st international workshop on engineering interactive computing systems for medecine and health care (EICS4Med 2011)

Bolton ML, Bass EJ, Siminiceanu RI (2008) Using formal methods to predict human error and system failures. In: Proceedings of the second international conference on applied human factors and ergonomics (AHFE 2008)

Buth B (2004) Analysing mode confusion: an approach using FDR2. In: Heisel M, Liggesmeyer P, Wittmann S (eds) Proceedings of the 23rd international conference on computer safety, reliability and security (SAFECOMP 2004). Lecture notes in computer science, vol 3219. Springer, pp 101–114

Campos JC, Harrison MD (2008) Systematic analysis of control panel interfaces using formal tools. In: Graham TCN, Palanque PA (eds) Proceedings of the 15th international workshop on design, specification and verification of interactive systems (DSV-IS 2008). Lecture notes in computer science, vol 5136. Springer, pp 72–85

Campos JC, Harrison MD, Loer K (2004) Verifying user interfaces behaviour with model checking. In: Augusto JC, Ultes-Nitsche U (eds) Proceedings of the 2nd international workshop on verification and validation of enterprise information systems (VVEIS 2004). INSTICC Press, pp 87–96

Carroll JM, Anderson NS, Olson JR et al (1987) Mental models in human-computer interaction: research issues about what the user of software knows, Number 12. National academies

Combéfis S (2013) A formal framework for the analysis of human-machine interactions. PhD thesis, Université catholique de Louvain

Combéfis S, Pecheur C (2009) A bisimulation-based approach to the analysis of human-computer interaction. In: Calvary G, Graham TCN, Gray P (eds) Proceedings of the ACM SIGCHI symposium on engineering interactive computing systems (EICS 2009). ACM, New York, NY, USA, pp 101–110

Combéfis S, Giannakopoulou D, Pecheur C, Feary M (2011a) Learning system abstractions for human operators. Proceedings of the 2011 international workshop on machine learning technologies in software engineering (MALETS 2011). New York, NY, USA. ACM, pp 3–10

Combéfis S, Giannakopoulou D, Pecheur C, Feary M (2011b) A formal framework for design and analysis of human-machine interaction. In: Proceedings of the 2011 IEEE international conference on systems, man, and cybernetics (SMC 2011). IEEE, pp 1801–1808

Delp C, Lam D, Fosse E, Lee C-Y (2013) Model based document and report generation for systems engineering. In: 2013 IEEE aerospace conference. IEEE, pp 1–11

Michael Heymann, Asaf Degani (2007) Formal analysis and automatic generation of user interfaces: approach, methodology, and an algorithm. Hum Factors: J Hum Factors Ergon Soc 49(2):311–330

Kieras David E, Polson Peter G (1985) An approach to the formal analysis of user complexity. Int J Man-Mach Stud 22(4):365–394

Palmer E (1995) "oops, it didn't arm."—a case study of two automation surprises. In: Jensen RS, Rakovan LA (eds) Proceedings of the 8th international symposium on aviation psychology (ISAP 1995), April 1995

Rushby J (2002) Using model checking to help discover mode confusions and other automation surprises. Reliab Eng Syst Saf 75(2):167–177

Sarter NB, Woods DD, Billings CE (1997) Automation surprises. In: Salvendy G (ed) Handbook of human factors and ergonomics, chapter 57. Wiley, pp 1926–1943

Spichkova M, Zhu X, Mou D (2014) Do we really need to write documentation for a system? CASE tool add-ons: generator+editor for a precise documentation. CoRR, 2014. http://arXiv.org/abs/1404.7265

Nancy Staggers, Norcio Anthony F (1993) Mental models: concepts for human-computer interaction research. Int J Man-Mach Stud 38(4):587–605

Thimbleby HW, Ladkin PB (1995) A proper explanation when you need one. In: Kirby MAR, Dix AJ, Finlay JM (eds) Proceedings of HCI '95 people and computers X, Huddersfield, August 1995. University Press, pp 107–118. ISBN 0-521-56729-7

Harold Thimbleby, Ladkin Peter B (1996) From logic to manuals. Softw Eng J 11(6):347–354

Tretmans J (2008) Model based testing with labelled transition systems. In: Hierons R, Bowen J, Harman M (eds) Formal methods and testing. Lecture notes in computer science, vol 4949. Springer, pp 1–38

Chapter 12
Reasoning About Interactive Systems in Dynamic Situations of Use

Judy Bowen and Annika Hinze

Abstract Interactive software, systems and devices are typically designed for a specific (set of) purpose(s) and the design process used ensures that they will perform satisfactorily when used as specified. In many cases, users will use these systems in unintended and unexpected ways where it seems appropriate, which can lead to problems as the differing usage situations have unintended effects on use. We have previously introduced a method of combining formal models of interactive systems with models of usage scenarios to allow reasoning about the effects that this unintended use may have. We now extend this approach to consider how such models might be used when considering deliberately extending the usage scenarios of existing interactive systems to support other activities, for example in emergency situations. This chapter explores a methodology to identify the effect of properties of emergency scenarios on the interactivity of interactive systems and devices. This then enables us to consider when, and how, we might utilise such devices in such emergencies.

12.1 Introduction

Interactive systems are typically designed around a specific set of use-cases (and in some cases particular users) to ensure that they will fulfil the needs of the *intended* users for the *intended* use. As part of this design process, issues of usability will also be addressed, and if the environment in which the system will be used is itself challenging (which is increasingly common), then this must also be included in the usability evaluations and user studies. While the established, and well-studied techniques of user-centred design, HCI or UX, will be successful in ensuring the systems are appropriate for their *intended* use, it is not always the case that systems will be used as intended. It is well known that users will often interact with systems in unexpected ways, but what we consider here is use in entirely different contexts than those

J. Bowen (✉) · A. Hinze
University of Waikato, Hamilton, New Zealand
e-mail: jbowen@waikato.ac.nz

A. Hinze
e-mail: hinze@waikato.ac.nz

B. Weyers et al. (eds.), *The Handbook of Formal Methods in Human-Computer Interaction*, Human-Computer Interaction Series,
DOI 10.1007/978-3-319-51838-1_12

for which the systems have been designed. There has, of course, been much research into context-aware and adaptive systems, which includes considerations of context-of-use, but here, we address the problem of non-context-aware systems being used in different contexts than those they were designed for.

If systems are used outside of their designed-for context, there may be unexpected or unintended consequences. These might be considered as merely annoying to a user. However, if the system in question is safety-critical and the user is unaware of limitations that may occur due to a different usage situation, then more serious consequences may result. Our original work in this area was motivated by exactly such a problem in healthcare environments. Medical devices, such as syringe and infusion pumps designed to be used in hospital wards or palliative care facilities, were being used in more challenging environments (emergency rescue helicopters, in situ accident response etc.) which could lead to problems with proper use.

Our focus on this problem led us to develop a modelling approach that allowed for reasoning about the effects of using interactive systems in different contexts of use. Specifically, when the system itself is not context-aware or adaptive and it is up to the user to understand how and why they may need to adapt their behaviour to successfully interact with the system. In this chapter, we extend this approach to consider the deliberate appropriation of interactive systems for use outside of their designed-for context.

We first give an overview of some of the motivations for this work and outline the types of scenarios that are of interest. We focus specifically on the example of using existing large-scale information displays out-of-context in situations of emergency. We first explore the information needs that arise in such situations. Emergencies such as earth quakes, floods or simple electricity cuts create unexpected information needs. During a conference stay, one of the authors found themselves stuck on the 6th floor of a darkened hotel without electricity. Established means of communication had ceased to function (phone, WiFi), and emergency lights were not working. A huge billboard opposite the hotel was clearly visible but did not provide any helpful information. This chapter is about using available means of communication, such as the large-screen billboard, in non-standard ways in case of emergency.

ICT is widely used in emergency situations. It has been observed, however, that social and organisational concerns often go beyond the available ICT support and even emergency-specific technological solutions and systems are abandoned (Bodeau et al. 2010). Emergency logistics and communication needs are unpredictable and may involve a number of government, private-sector or NGO agencies, local first responders, and ad hoc groups of citizens in addition to affected people in a disaster area. Bodeau et al. (2010) observed that the interdependent flow of information and control “cannot be easily disentangled” nor planned ahead precisely (Bodeau et al. 2010). Although information need and control flow between external agencies has been recognised as being complex, involving subgroups and dynamic cliques (Comfort and Haase 2006), we consider all of these parties as a transparent external body and abstract from the flow of control and information *within* this response agency.

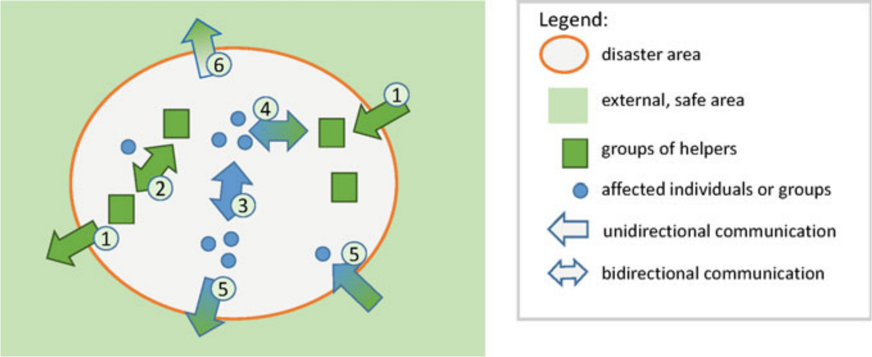

⑥
④
①
②
③
①
⑤
⑤
Legend:
disaster area
external, safe area
groups of helpers
affected individuals or groups
unidirectional communication
bidirectional communication

Fig. 12.1 Types of communication and information flow: between external management, responders, affected individuals, and from the disaster area via sensors

Management of complex situations such as natural disasters, flooding, forest fires, earth quakes and mass casualty accidents often leads to communication breakdown and information loss (Parush and Ma 2012). Good situational awareness is vital for decision making and quick response (Yang et al. 2013). A number of studies have analysed the information requirements of emergency response teams, e.g. (Diehl et al. 2005; Robillard and Sambrook 2008; Yang et al. 2013). Extreme importance is given to information about environmental conditions in an intervention as well as to response participants, status of casualties and available resources. These are to be obtained from local people, on-site personnel and sensors in the incidence area.

We can conclude that while categories of information needs may be predicable, detailed requirements will not be and suitability of locations will depend on the given context. Figure 12.1 gives an overview of parties involved and communication channels: Emergency management is typically located outside the disaster area, with groups of helpers entering the area in which groups or single individuals may be situated. Communication is needed ① between management and responders, ② between groups of responders, ③ between affected individuals, as well as ④ between individuals and responders and ⑤ directly between individuals and management if possible. Finally, emergency management may be able to obtain environmental information directly from sensors ⑥. The two main types of communication are information collection for emergency management and provision of information to the individuals in affected areas, using various channels. Some channels may allow two-way communication, others work only uni-directional. Here, we are concerned predominantly with communication channels ③, ④, ⑤ and ⑥, which are those that cannot rely on specialised or specifically provided communication technology (as ① and ② can).

In case of emergency, all available communication channels need to be used. Traditional emergency communications delivered via a central communication channel is to known recipients and relies on specific hardware with the individual (radio, phone etc.). If we want to exploit any existing technology in a vicinity, we need some way of managing the information about its capabilities and its limitations under certain circumstances. Because traditional mechanisms of communication may be interrupted by a disaster (e.g. phone networks) having the ability to provide large-scale communication in other ways may be important. In situ large-screen displays are potential communication media. It is conceivable that in situ displays can also be used to gather information and send it back to a central point. We explore this further in Sect. 12.4.

In this chapter, we describe a modelling approach to explore the use of large-screen displays for emergency communication. Section 12.2 gives an overview of how we have used this modelling approach to support interactive medical devices in different contexts of use. We then go on in Sect. 12.3 to show how this can be applied to understanding how public interactive displays (such as those described in case study 5) might be used in emergency situations. Section 12.4 explores a larger example from the area of emergency communication. In Sect. 12.5, we compare our approach with those from related work and conclude this chapter with a summary in Sect. 12.6.

12.2 Background

In earlier work, we investigated ways of modelling the interactions of a system or device to understand how it might be compromised when used in particular situations where environmental factors could interfere with normal methods of interaction (Bowen and Hinze 2012). The intention was to be able to inform users about these compromises so that they could adapt their use (so here, the user becomes context-aware and adaptive rather than the system) in order to successfully use the system differently. By creating models of interactive systems and their widgets, and characterising these by their interaction types (which in the broadest sense can be described as visual, audible, tactile etc.), we could then create relations to models of locations and limiting factors which subsequently enabled reasoning about the effects of these limiting factors.

This approach allows us to answer the following sorts of questions about interactive systems (*S* and *T*) in various situations of use (*L* and *M*):

- Can *S* be used in *L*?
- Is it better to use *S* or *T* in *L*?
- What does the user need to be aware of when using *S* in *L*?
- If *L* becomes *M* what effect does this have on *S*?

The answers can then be used to inform users and allow them to make decisions about which devices to use when, or to understand that how they use the device or system may need to change.

We start with an initial model of the interactive system which describes the interaction possibilities in terms of the widgets the system has (buttons, menus, displays etc.) and the categories, or types of these widgets which informs the *nature* of the interaction. For example, a button which requires touch for interaction can be labelled as haptic, whereas an alarm which sounds in a given condition would be categorised as audible. These are lightweight descriptions which aim to capture the essential elements only, so we do not describe the low-level descriptions of physical characteristics of widgets mentioned in Chap. 9 for example. The interaction information from this model is then combined with a model of a given situation of use, which are properties of different situations relating to environmental attributes (such as noise and lighting levels), tangible properties (which for our previous work in the medical domain were things like patient/practitioner ratios), as well as less tangible attributes such as the levels of stress experienced by users in a given location.

The models consist of descriptions and relations, and as such can be expressed, and reasoned about using a variety of formalisms or notations. From the interaction and location properties, we create a relationship between types of interaction and location attributes where the interaction would be adversely affected. For example, an audible widget (such as an alarm) would be adversely affected by a noisy environment (as the alarm might not be heard). Previously, we showed how ontologies could be used for the purpose of describing the interaction properties and location factors and for creating the relations between them (Bowen and Hinze 2012). We then used the semantic web reasoning language, SWRL (Horrocks et al. 2004a) within the ontology tool Protégé (Gennari et al. 2002) to generate classes of affected devices in given locations. Subsequently, we incorporated the reasoning into a tool which provides the relevant information (i.e. the answers to the questions listed above) in a palatable form for users (Bowen et al. 2014).

The four questions listed above were originally developed from the requirements of using medical devices outside of their intended usage scenario (for example, when a syringe pump designed to be used in a medical ward is taken into an emergency rescue helicopter). However, these questions are equally applicable in other domains (for example, when Department of Conservation workers need to use field equipment in new environments) and for other purposes (determining which equipment is most suitable to be placed in situ in an environment which has changing conditions). Here, we show how we can use the same approach to support dynamic reasoning about interactive systems in emergency situations in order to support emergency management scenarios. We are still focused on the problem of non-context-aware (and non adaptive) interactive systems. However, rather than using the models to determine how a user's interaction may have to change when using a particular device in a different location, we consider how we might take advantage of in situ devices (such as large-screen interactive public displays) in emergency situations by reasoning about the available interactions, given some environmental properties which result from that emergency. The information we are modelling remains the same, but the reasoning we perform and the intended results are different. As such, we have a fifth question which we wish to answer (and so add to the list above), given a collection of interactive systems $S^{1..n}$ in a location L, and a use-case U, in an emergency situation E:

- Which of $S^{1..n}$ in L can satisfy U in E?

One of the differences here is that we are now considering situations where the situational factors are not fixed, but rather occur due to some event and may change over time. However, the ability to reason about their effect remains the same, albeit more dynamic, and so we extend our approach to incorporate this. We will still have models of the interactive systems and their widget types, but in this case, the models will be for all interactive systems we know about in a given geographical location that we can utilise for some form of emergency management or communications. The reasoning rules will allow us to reason about what type of event attributes impact particular types of interaction—and therefore particular interactive systems in the domain. These will be used in conjunction with dynamic models of actual effects of an event as it occurs. We give details of this in the following.

12.3 Models and Reasoning

We create models for each of the information types involved in the reasoning. The first are the sets of properties of the interactive systems and devices which are known to exist in a given location. For our previous work with safety-critical medical devices, we typically already had initial models (based on the presentation models of Bowen and Reeves 2008) from which we could automatically extract widget and interaction information. As the interactive systems we are now dealing with are not safety critical, this is less likely to be the case, and so we manually create the sets of properties of interest. These are the *types* of interactions the widgets of the system provide. The second model describes the relationships between interaction types and *all* known possible effects that may result from an emergency event. In their simplest form, these can be characterised as a binary relation where a *factor* affects a *widget type*, such as *noise* affects *audible*, for example. However, in practice, the range of properties and effects are much more complex and include variable attributes (actual decibel levels of noise volume perhaps where the level affects different types of audible output). In this section, we primarily focus on binary relations as we describe the approach generally, and we will expand on these to show more complex relationships in Sect. 12.4 where we present a larger example.

We consider every possible effect that may hinder a particular type of interaction as the basis to build the relation used in the reasoning. The interaction types of the systems and the effect relation are considered to be "fixed" models, in that they can be created ahead of time and describe the things we already know and which do not change. Then, there is a dynamic model which is created when an event occurs. This contains only the event factors which are relevant in the given scenario, i.e. result from the event. It is these dynamic factors that are used to reason about the availability of both interactive systems and their interactive capabilities in the given emergency situation. The final description generated from the reasoning is a subset of devices containing a subset of their interactive capabilities, which can then be used to make decisions about which to use for any given use-case. That is, we choose from this subset of systems based on which still have the required interactions available for our needs.

There are many different ways of describing the models and reasoning about them (Bowen et al. 2014). Here, we describe the attributes and properties as sets to show the effects of the relations, but we can practically implement the reasoning described in several different ways, including following the ontology approach used previously. This is particularly useful for automatic reasoning on multi-faceted attributes with more complex relations, such as those we will present in our larger example.

The large interactive displays described in the case study are typical of the types of systems we may wish to exploit for information provision or data gathering in an emergency scenario. Suppose that both the Domain Mall Interactive Display and the Magic Carpet are located in an area we wish to incorporate into our emergency planning scenario (a large shopping mall for example). We can create initial fixed models for their interactivity (which we separate into input and output interactions) as follows:

Magic Carpet INPUT = {Touch}
Magic Carpet OUTPUT = {Visual, Audible}

Domain Mall Display INPUT = {Motion, Location}
Domain Mall Display OUTPUT = {Audible}

The full set of interactions for each of these systems is then the union of their input and output sets. The set of all possible interactions for any given collection of devices in a domain is the union of their respective interaction sets.

The second fixed model is the relation of effects to interaction types (e.g. Noise $\mapsto$ Audible). We have a set of all known situational factors and create a many-to-many relation between this and the set of interactions. Given the example systems above and an assumed set of factors, we can describe the following:

INTERACTIONS = {Touch, Visual, Audible, Motion, Location}
FACTORS = {Noise, Heat, Vibration }
EFFECTS = {Noise $\mapsto$ Audible, Heat $\mapsto$ Touch }

This is an example of the simplest type of relation, the binary relation where a named factor affects an interaction. However, in real-world scenarios, the effects are typically multi-faceted, as we will see later in our larger example. The dynamic model of scenario factors is generated in a given emergency situation. Suppose, we have a bomb warning in a public area which has led to the sounding of the evacuation alarms, from this we can determine that for this scenario:

FACTORS = {Noise}

Our emergency management requirements for this scenario are to provide additional information to assist with the safe evacuation of people by directing them to particular exits and keeping them away from specific areas of the building. We therefore only need to consider the available output interactions. We take the set of known factors and restrict it to create the *current effects* relation (a subset of the elements in effects) using the dynamic event factors. So here, we now have $\{\text{Noise} \mapsto \text{Audible}\}$. We then retrieve all interactions in the relation for these factors ($\{\text{Audible}\}$) and remove these affected interaction(s) from the interaction set of each device to gain an overview of remaining interactive availability, that is, the complement of affected interactions with respect to interactions:

$$\begin{aligned}&\text{AVAILABLE\_INTERACTIONS} = \\ &\quad \text{INTERACTIONS} \setminus \text{AFFECTED\_INTERACTIONS}\end{aligned}$$

If the result is the emptyset, $\emptyset$, there are no available interactions remaining for a system and we cannot use it in the given scenario. If the remaining interactions are a subset of total interactions, then we can use the system in a restricted capacity, otherwise we can use it fully.

For this example then, the remaining interactions for the Domain Mall Display and Magic Carpet would be:

$$\begin{aligned}&\text{Domain Mall Display} = \{\text{Visual, Audible}\} \setminus \{\text{Audible}\} = \{\text{Visual}\} \\ &\text{Magic Carpet} = \{\text{Visual}\} \setminus \{\text{Audible}\} = \{\text{Visual}\}\end{aligned}$$

So we can use both of these devices, but the Domain Mall Display only in a limited capacity to display information.

In order to support automated reasoning about more complex scenarios and factors, we have shown previously how we can model these relationships in an ontology using the Protégé tool (Gennari et al. 2002). Figure 12.2 shows a snippet of the class hierarchy for the medical device example. The members of "WidgetsRestrictedByEvent" are populated by the first level of reasoning using rules such as:

```
Widget(?w),
hasfactor(Event, ?f),
restrictsWidget(?f, ?w)
->WidgetsRestrictedByEvent
```

So, if an event has a factor that restricts a widget, then that widget is categorised by the reasoner as one of the widgets restricted by that event based on their interaction type. Similarly, we can create categories for systems affected by events (those which have widgets in the affected category) and so on, to expand the descriptions and generate the required information. When an event occurs, we use the effects it has as parameters to the reasoning rules.

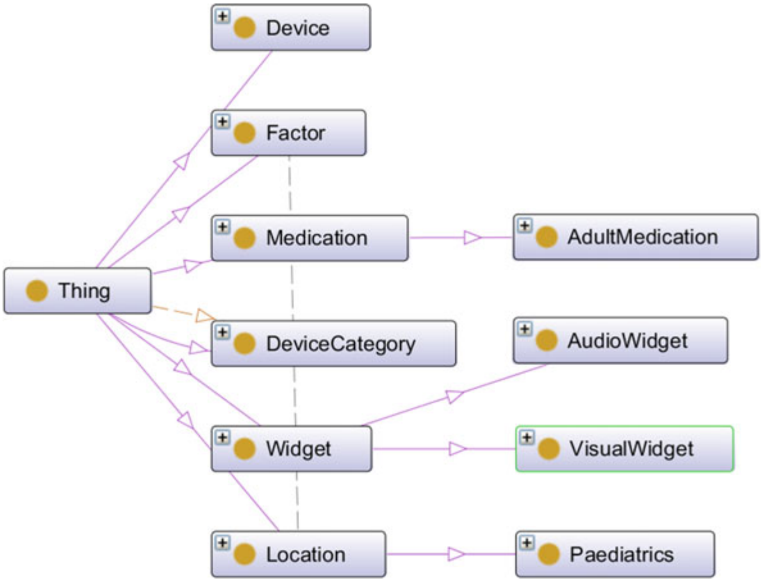


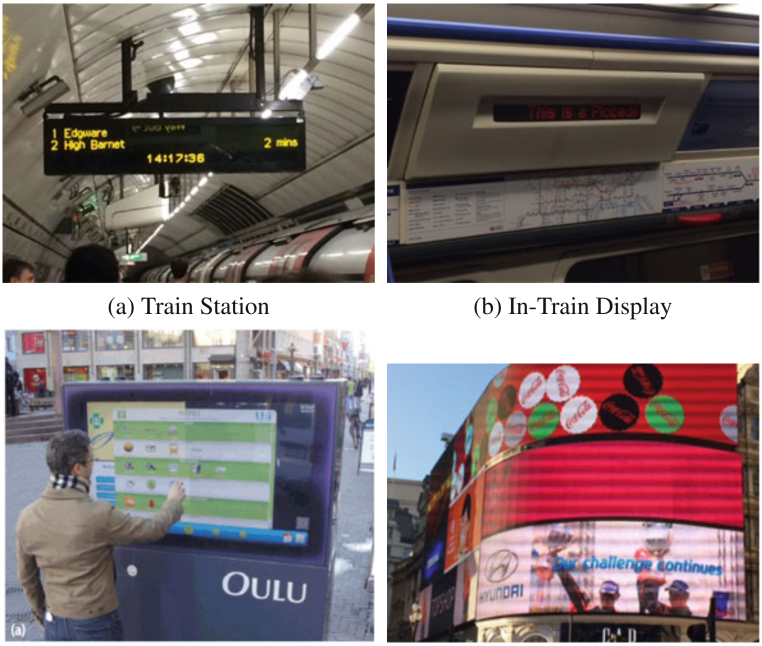
Fig. 12.2 Example of the ontology hierarchy

Typically, the sets of interactions and event factors will be larger and more complex than the simplistic values given above. Noise, for example, can be considered at a more granular level within certain ranges of volume. We want to incorporate this richness in order to be more precise about not only the details of the factors, but also the detail of the interactions. So rather than just considering how the system can be interacted with in terms of its categorisation, we also include details of what *types* of information are associated with these interactive categories.

The Domain Mall Display for example has different types of visual outputs it can display—images, text etc. Other systems, such as interactive maps, may be more limited, while others may have a larger range of capacities. It may be the case that only some of these are affected by different levels of event effects and so all of the models need to be expanded to incorporate this. These multi-faceted attributes can also be considered within an ontology. We can then use the same approach by modelling the event factors as objects with multiple facets of interest and extend the widget interactions to include detail around types of information. A more realistic set of factors, interactions and effects, therefore, is shown in the larger example in the next section.

12.4 Earthquake Emergency Management Example

We expand on our basic theory to describe how we can use this approach with more complex interactive systems in scenarios with a variety of more interesting properties. The example situation is that of an earthquake, with affected areas being those



(a) Train Station

(b) In-Train Display

(c) Public Display

(d) Large-screen Advertisement

Fig. 12.3 Large-screen public displays

found in a typical urban setting, e.g. shopping malls, living quarters, train stations, central business district (CBD) and indoor and outdoor amusement parks.

Communication in Earthquake Emergency

The types of large-scale displays one would encounter in such areas range from simple moving LED displays to announce the arrival of the next train (see Fig. 12.3a, b), interactive urban information screens (e.g. displaying maps and local information as in Fig. 12.3c, Kostakos et al. 2012), dynamic advertisement screens (Fig. 12.3d), media facades (Köster et al. 2015), multi-screen displays, sensor-equipped screens for environmental monitoring, to interactive art installations (e.g. using Sensacell technique[1] or using cameras, sensors and motion controllers Fortin et al. 2013). Interactive large-scale displays can capture input from motion (proximity sensor) and touch (touch screen elements) and provide output visually via the display or audibly via speakers.

In case of emergency, we may want to use screens and displays differently from their ordinary usage pattern. Some behaviours that are built-in (e.g. the response to inputs in normal usage) might be usefully employed in an adapted manner. Others might need to be used in a radically different way to their original design intention (e.g. for providing illumination or giving directions). The emergency usage scenario

[1] http://www.sensacell.com.

does not contain pre-defined event effects as our previous medical example did (see Sect. 12.2), rather the situation needs to be evaluated as it develops. For each scenario, the applicable effects need to be identified and it then needs to be reasoned which elements of the displays can or cannot be used due to restrictions.

In the aftermath of an earthquake, there will be increased risk of collapsing buildings, landslides during rain and liquefaction.[2] Communication tasks for such an example that could be supported via the large-scale displays would be:

- Inform: rescuers might wish to inform people in the emergency area about the situation and warn about potential aftershocks (using text, images, audio).
- Direct: directions may need to be given to both first-response rescuers as well as affected people. Examples are directions *towards* meeting points, safe areas, medical help or *away* from dangerous areas (using text information, maps or arrows, possibly via several screens).
- Warn: warnings about localised danger of collapsing buildings or areas of liquefaction (so-called red zone), or about dangerous volatile substances such as carbon monoxide risk or chemical hazards (e.g. via images, voice or alarm sound).
- Interact: to gain an overview of the situation, rescuers may wish to identify how many people are located in a restricted area and their particular needs (e.g. medical)
- Measure: any sensors on a display may be employed to gain further information about the local situation: stability of the ground, level of chemicals, heat, etc.

Not all displays can equally support all activities and interactions. For example, the communication via displays may further be hindered by hazards that affect the visibility, such as darkness, dust, steam, glare, (partially) broken display and accessible distance to display. Audio signals may have to be used in addition to other information or on their own. Audio signals may be impaired by environmental noise before or during the earthquake (so-called artillery-like "earthquake booms")[3] and after (e.g. from collapsing buildings or broken pipes) or obscured by broken speakers. The earthquake survivors might be injured, disoriented, alone or in groups, trapped inside or outside. They may be locals or tourists, with or without knowledge of the location or local language, mobile or impaired. We now describe a selection of specific situations and their communication needs. Based on the types of communication tasks described above and the situations that helpers and affected individuals might find themselves during an earthquake, the following communication tasks may arise.

Directing people towards medical help: Groups of affected individuals may be situated in, or move through, the disaster area, many of which may need medical help. One of the first responses is to set up medical emergency centres and to announce their locations to people in the affected areas. During the initial phase of medical response (moving from so-called solo treatment areas to disaster-medical-aid centres), traditional communication relied on runners (Schultz et al. 1996). The location

[2] Liquefaction is a process in which during an earthquake soil is rearranged such that it behaves more like a liquid than a solid.

[3] http://earthquake.usgs.gov/learn/topics/booms.php.

of treatment areas and (once these have been established) the locations of medical-aid centres need to be made known to affected individuals. The communication of dedicated locations may use one, or several, of a variety of location indicators, such as the name of a place or its address. These, however, may only be useful for people with local knowledge. Better location indicators may be widely visible landmarks, maps, signs and arrows, accompanied by simple indications of distance. LED-based displays (such as the ones shown in Fig. 12.3a, b) could be used to display text (such as name, address, distance) and simple signs (e.g. arrows) but could not serve complex maps. Using directional arrows relies on clear positioning information of the display and may be liable to repositioning due to damage. Information screens, advertisement or media facades (see Fig. 12.3c, d) can serve text, signs and complex maps. Directional arrows are safer to use on stationary facades as smaller displays may have been dislocated.

Warning about danger area: Some areas may be dangerous to enter after an earthquake, for example large or overhanging buildings that are in danger of collapsing, or areas in which the ground is unstable. Signage directly affixed to buildings or near these areas can be used to prevent people from entering. Communication via screens or facades (see Fig. 12.3c, d) may use warning signs, images, videos and text, with possible use of additional audio warnings. Displays in close proximity to the area would need to name the location, and use maps and signs. As with directing people, warnings using LED are restricted to text and simple signs with possible warning audio.

Interacting with people in a location: In order to gain intelligence about the emergency situation on the ground, and about the people affected, interaction with individuals that are located in the emergency area is crucial. Interaction may be directly via screens that have input options (such as urban information screens and art installations), or indirectly by alerting people to available communication channels. For example, LED screens, advertisements and media facades may show emergency telephone numbers or social media contacts. Screens with interaction capabilities may be used depending on the supported functionality, such as microphones, cameras and touch screens.

Other communication goals may be the interaction with people in enclosed spaces (inside collapsed building), coordination of numerous groups of first response helpers, and provision of general information about the situation (equivalent to using the billboard to provide information to people in a hotel without electricity).

Modelling of Earthquake Example

The nature of the data we model for both interactive systems and event effects is now extended to consider more specific attributes as well as valued attributes. Figure 12.4 shows how these can be used to build up more complex relations than the previous binary examples. It is no longer the case that an event effect simply restricts a type of interaction. The level of the effect (e.g. decibel level for noise) affects a particular type of interaction data of a system. So rather than just considering an output interaction to be “visual”, we use a more finely grained description which considers more detail of the interaction (not just how it occurs) which might include items such as

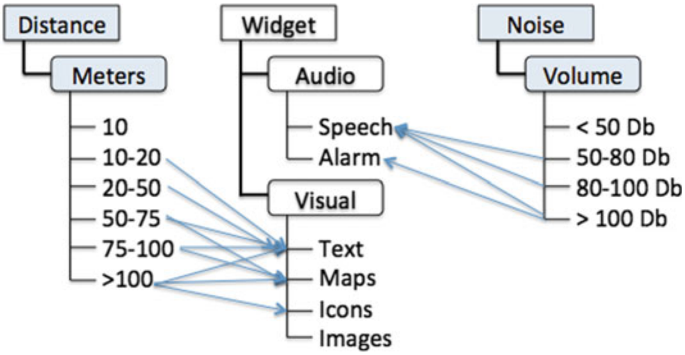


Fig. 12.4 More detailed relation of effects on interaction data types

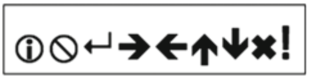


Fig. 12.5 Icons

text, images, voice audio and alarm audio. The relation then is more specific being between levels of effects and defined items of interaction data.

When we want to decide which of the public displays in our emergency communication area are available for us to use, we also need to identify the type of information we wish to communicate (so again, we aim to answer the question “which system is the best to use for this scenario?”) We also want to find out which (if any) of the available systems can provide this feature in the current event. For example, if the location of the Media Facade has obstructions on the ground which prevent people getting closer to it than 50 m, then it is not useful for displaying detailed text instructions as people cannot get close enough to read them. We can, however, use it to display images (which might include standard emergency representations such as a red cross to indicate medical facilities or a no entry sign) or simpler, easily recognised icons such as those shown in Fig. 12.5 for way finding or access control.

These can be viewed, recognised and identified from further away, and so will be affected by different values of the event attribute. Legibility of public displays is an important consideration, especially when being used in emergency scenarios. Work by Xie et al. (2007) has proposed methods to model and validate sign visibility during design and prototyping phases and we can imagine incorporating the data produced by such a modelling process into our effects relation.

The full data we must now encapsulate in our models includes:

- interaction sets for all interactive systems available in a given geographical area,
- data types of the interaction sets,
- event effects, and attribute values and
- requirements of data we wish to collect or output to manage the event

In the earthquake scenario described above, we have access to the following devices and interactions in the geographic location of interest:

LED Display INPUTS = { }
LED Display OUTPUTS = {Text, Icons, Numbers}
Info Display INPUTS = {Touch}
Info Display OUTPUTS =
{Text, Icons, Numbers, Maps, Images, VoiceAudio, Alarm}
Media Facade INPUTS = { }
Media Facade OUTPUTS = {Images, VoiceAudio}

As we are able to gather information about the event and its effects in the given location, we can begin to reason about the interactive systems and their available interactions in the manner described in Sect. 12.3. Now, however, we reason about the effects at a per system level, rather than for all systems, as the particular effect attributes and their values may differ from system to system depending on their location. We map the effect to the system based on precise geographical coordinates.

Our requirements are to provide information in the form of maps, text and icons to indicate location of medical services, while at the same time trying to gather information about the number of people in the geographical area. Our goal is to identify which (if any) of the public displays can be used to display any or all of these types of data (which have visual capacity for these items in this event scenario) and which (if any) can be used to gather data from people (can both display textual instructions and accept touch inputs which can be used as a simple counting system).

The general environmental factors of the earthquake in the specified location are noise, unstable ground, dust, obstructions and unsafe buildings. The particular effects for each of the interactive displays we are considering are expressed at the more detailed levels for the known parameters as we can identify them. For example, ground obstructions in front of a display are expressed in terms of estimated distance.

LED Display location effects = {Noise[>100 Db], Distance[>30 m]}
Info Display location effects = {Noise[30–50 Db], LightLevels[dark]}
Media Facade location effects = {Noise[50–75 Db], Vibration}

This allows us to generate the sets and types of available interactions for each of the displays and then coordinate our information provision on a best-fit basis. Using a valued-relation, such as that shown in Fig. 12.4, we can now reason about these effects. The resulting available interaction sets are generated by removing the affected interactions based on the current effects relation:

LED Display AFFECTED_INTERACTIONS =
{Text, Numbers, Maps, VoiceAudio, Alarm}
Info Display AFFECTED_INTERACTIONS = {VoiceAudio}
Media Facade AFFECTED_INTERACTIONS = {VoiceAudio}
LED Display AVAILABLE_INPUTS = { }
LED Display AVAILABLE_OUTPUTS = {Icons}
Info Display AVAILABLE_INPUTS = {Touch}
Info Display AVAILABLE_OUTPUTS =
{Text, Icons, Numbers, Maps, Images, Alarm}
Media Facade AVAILABLE_INPUTS = { }
Media Facade AVAILABLE_OUTPUTS = {Images}

Finally, we consider again the requirements we have for this emergency situation, which are to provide information in the form of maps, text and icons as well as gathering input using touch. From the restricted sets of interactions, we see that we can use the Info Display to gather information, and all three of the devices can be used to provide information but with limitations on which types of information they can display. Over time, the event effects may change, for example the noise level in the LED Display location may drop to below 50 Db; when we become aware of the changes, we can recalculate the effects and change our information provision as required.

We can use our previous approach of an ontology and reasoner to describe these properties and infer the information shown above. We have been using the Protégé (Gennari et al. 2002) tool to build ontologies using the OWL ontology language.[4] This type of ontology consists of classes, individuals (or instances) and properties.

We create two classes initially, *InteractiveDevices* which has three instances (*InfoDisplay, MediaFacade* and *LEDDisplay*) and *InteractionType* with eight instances (*Out_Alarm, Out_Images, Out_Maps, Out_Numbers, Out_Icons, Out_Text, In_Touch* and *Out_VoiceAudio*). Properties are binary relations on individuals, and we use two different types of property. First, an object property (which relates individuals) called *hasInteractionType* which relates instances of the *InteractionType* class to instances of *InteractiveDevices*, e.g. LED Display *hasInteractionType* Out_Text. We also create the inverse of this property *InteractionTypeOf* and the ontology reasoner automatically populates this relation. We then create datatype properties (which relates individuals to data literals) for the valued properties: *hasNoiseLevel, hasDistanceFrom, hasLightLevel* etc. and populate this with the individual instance values for this scenario, e.g. MediaFacade hasNoiseLevel ">= 50". Figure 12.6 shows the visualisation of the classes and relations.

Once we have added all of the known data to the ontology (which consists of both the pre-known device data as well as the current scenario data), we can create additional classes, *AffectedInteractions, AvailableLEDDisplayInteractions* etc. which will provide the dynamically generated results from the SWRL reasoning.

[4] http://www.w3.org/TR/owl-guide.

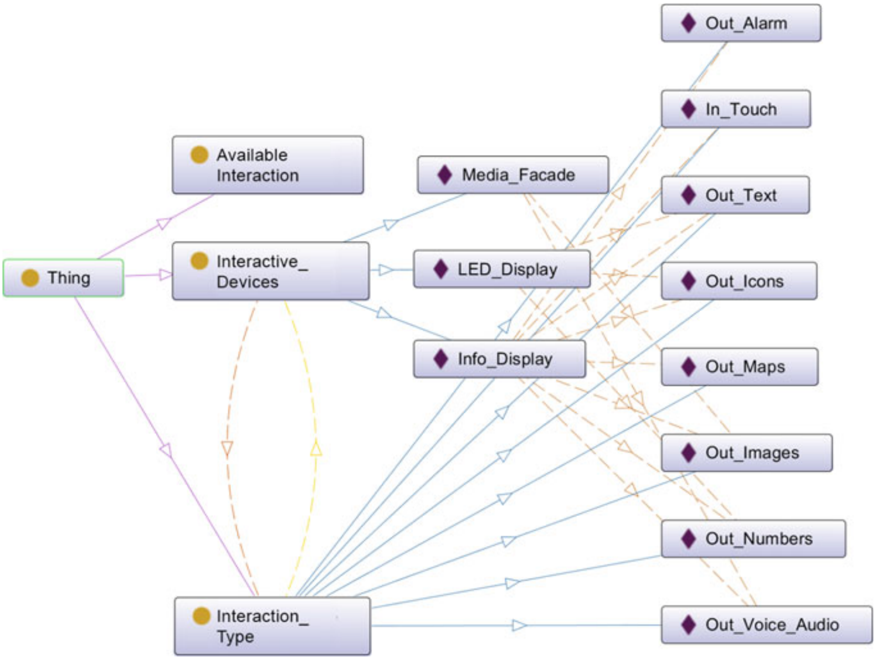


Fig. 12.6 Ontology structure for earthquake example

An example rule is shown below, describing the pattern to identify all devices with output affected by noise levels of more than 100 db.

```
Interactive_Devices(?device),
Interaction_Type(?output),
hasInteractionType(?device, ?output),
hasNoiseVolume(?device, ?noise),
greaterThan(?noise, 100),
SameAs(?device, LED_Display)
-> hasAffectedOutput(?device, ?output)
```

12.4.1 Use in Practice

As previously stated, while we can use Protégé in this way to support the modelling and reasoning via an ontology, this is not a practical approach for non-expert use. In order to solve this problem, we use the ontology data as the input to a custom-built tool which provides an easy-to-use front-end. This enables a user to pose specific questions such as “Can I use this device in this situation?” and provides relevant information to support their choice. Figure 12.7 gives an overview of the components

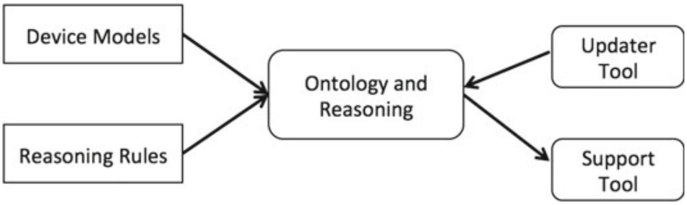
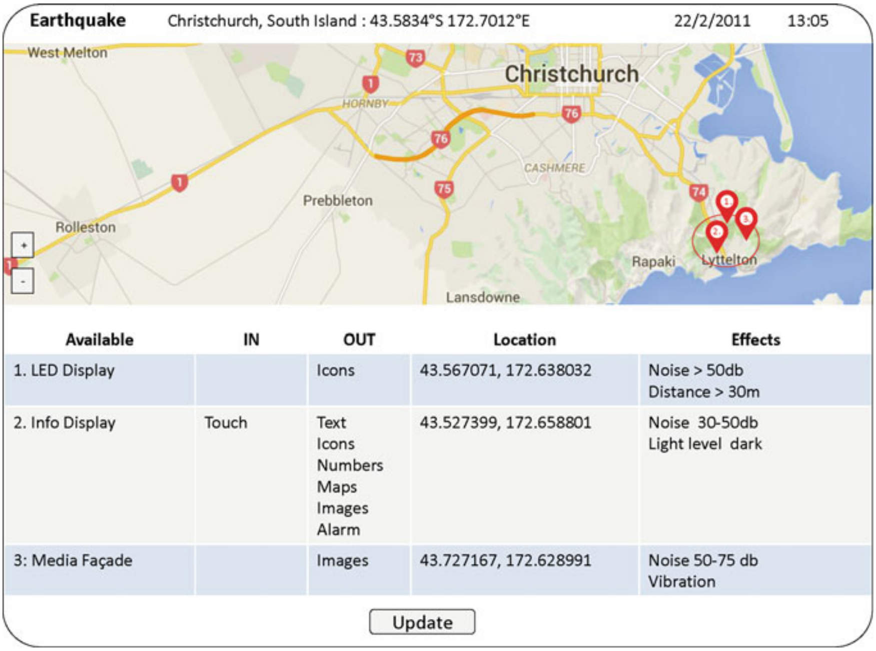


Fig. 12.7 Overview of tool structure


Earthquake Christchurch, South Island : 43.5834°S 172.7012°E 22/2/2011 13:05

Available	IN	OUT	Location	Effects
1. LED Display		Icons	43.567071, 172.638032	Noise > 50db Distance > 30m
2. Info Display	Touch	Text Icons Numbers Maps Images Alarm	43.527399, 172.658801	Noise 30-50db Light level dark
3: Media Façade		Images	43.727167, 172.628991	Noise 50-75 db Vibration

Fig. 12.8 Prototype of end-user tool

for such a tool. The ontology is built using the information from the device models and rules generated by the domain experts. The data from the ontology then provides the basis for the end-user tool. A second tool can be used to update the ontology if new data emerges during the emergency scenario allowing the reasoning to remain dynamic.

Such a tool must be tailored to the specific end-user and their requirements—for example a local government emergency management body. We have previously developed such a tool for use with our medical domain example (Bowen et al. 2014) and suggest using the same approach to develop a tool for the emergency management scenarios.

Figure 12.8 shows a proposed prototype interface for such a tool. An interactive map allows the user to navigate to selected areas within the disaster zone, and this shows the public displays that are available in that area. These are then listed with their available interactions to enable decisions to be made about which can be used for different communication requirements. The current effects of the situation and their locations are also shown. Clicking the "Update" button allows the user to add or edit location effects enabling dynamic reasoning as the situation changes over time.

So now, to find the answer to the question posed earlier: given a collection of interactive systems $S^{1..n}$ in a location *L*, and a use-case *U*, in an emergency situation *E*:

- Which of $S^{1..n}$ in *L* can satisfy *U* in *E*?

we can use such a tool. In the prototype of Fig. 12.8, each interactive system (*S*) is listed along with its location (*L*) and the available interactions in the emergency situation (*E*) to show which can be used to meet the use-case (*U*).

12.5 Related Work

We outline here related work on disaster management communications and information provision using large-scale public displays as well as on semantic modelling for interactive and context-aware systems.

12.5.1 Disaster Management and Communications

Much of the research focussing on the use of technology in disaster management and communication relates to one of two things. Firstly, there are approaches which consider how existing large-scale technological solutions can be used to gather, disseminate and manage information in emergency situations. This may include things like the use of social media and how shared information can be mined and utilised in uncertain conditions (Lu and Yang 2011) or the use of in situ technologies built on data gathering from RFID or sensor technologies communication via ad hoc wireless networks (Yang et al. 2009). Secondly are approaches which look to create better centralised views of disasters for control centres through approaches such as interactive mapping and two-way communications via, for example smartphones. An example of this can be seen in Schöning et al. (2008), where smartphones are used as augmented reality devices through combining cameras and paper maps to relay spatial information.

Large shared physical displays and pinboards have a long tradition, for example for fire-fighter support (Jiang et al. 2004). Increasingly, social media is used as auxiliary media type in disaster response (Yates and Paquette 2011; Horita et al. 2015) for communication between affected citizens, helpers and response agency, typically in an ad hoc fashion. Current use of displays in public places is typically targeted to serve for advertisements, traffic information or as elements of public art. The latest models of large-scale displays can now be used for delivering multimedia content and are context-aware (i.e. using sensor input to deliver content-specific content) (Davies et al. 2014). To remedy the lack of information, the use of large-scale displays has been explored in Olech et al. (2012). To avoid lost time when searching for the best route to a site, or for coordinating first responders, they propose an interactive pinboard. Public screens being used as digital pinboards had previously been used for artistic purposes or social communication (Cheok et al. 2007; Hosio et al. 2010; Thelen et al. 2010). Interactive public displays are used to broadcast information (i.e. as digital signage playing video, animation, photographs) or interactively. Ojala et al. (2012) explored the use of interactive features and found that beyond maps and consumer information, services were often unexpectedly popular or unpopular and not in keeping with previously stated information needs. They discovered that location is a major factor for the success of a service. These displays were placed purposefully and did not re-use in situ hardware.

The use of such pre-existing technology available in situ for signage and communication in emergency situations has been little explored. For example, Majumder and Sajadi (2013) mentions the option of using large displays for emergency response without giving further details. Both Rauschert et al. (2002) and Olech et al. (2012) propose the use of large-screen displays to support interaction and information gathering for emergency management. Rauschert et al. (2002) use a multi-modal interface using speech and gesture recognition integrated with GIS functionality. Their project was concerned with aspects of computer vision and speech processing. It assumed the availability of necessary hardware in appropriate places and was not concerned with possible restrictions due to the emergency or disaster that is being managed. Similarly, Olech et al. (2012) aims to provide first responders with so-called hotspot locations at which all important information is being displayed. Again, the focus was on software features, based on the assumed available hardware support. Both these approaches presuppose tidy access and clean communication channels. By contrast, our work aims to establish information flow using existing in situ (interactive) public displays of varying quality and capability and addresses situations that are potentially considerably less structured and orderly.

12.5.2 Semantic Modelling for Interactive and Context-Aware Systems

The system and techniques introduced in this chapter are often discussed in relation to context-aware systems.

Context-aware systems measure their context (e.g. location, user, environment etc.) and change their behaviour based on the measured context (Schilit et al. 1994). Context-aware systems are closely related to event-based systems, i.e. systems that react to the detection of events or patterns in their data input stream (Hinze et al.

2009). Examples of event-based systems are those that analyse social media patterns in order to detect events such as earthquakes (Sakaki et al. 2010), or the use of sensor networks in the detection of volcano eruptions (Werner-Allen et al. 2006). The latter system is context-aware as the locations of each of the sensors are taken into account. Other context-aware systems are those that deliver location-based data to mobile users (Cheverst et al. 2000), or support location-based reading of digital books (Hinze and Bainbridge 2015).

The systems described in this chapter share characteristics with context-aware systems (i.e. measuring of contextual data) but are themselves not context-aware. Instead they communicate to the user the implications about the use of a given interactive system in a given context.

Related work in modelling of, and reasoning for, interactive and context-aware systems spans a number of research areas. Traditional knowledge-based systems (KBS) in the field of artificial intelligence have the goal of replicating commonsense reasoning ability. They use a *knowledge base* of facts to infer further facts based on rules and conditions, typically encoded in the *inference engine* (Lenat and Guha 1989). Large ontologies, such as Cyc, contain hierarchies of generic to field-specific knowledge, and rules that give meaning to these facts.

Modelling and reasoning about interactive systems, on the other hand, are a means to ensure the systems' correctness and reliability (Back et al. 1999). The use of ontologies and semantic reasoning is only one way to explore a system's behaviour. Our approach uses the ontology to model different contexts of use.

Our approach is most similar to the ones used in a semantic web context (Berners-Lee et al. 2001). Semantic reasoning in the semantic web is used to infer facts and relationships between concepts (Horrocks et al. 2004b). Our work uses such semantic reasoning as a tool but does not focus on researching reasoning. Our approach is rather a combination of lightweight reasoning as done in the semantic web context, combined with concepts of event-based and context-aware systems.

12.6 Conclusions

12.6.1 Summary

In this chapter, we have discussed the use of models to capture both static and dynamic attributes of interactive systems and their environments of use, with the goal of allowing these to be used in non-standard ways to support communications in emergency situations. While there may be many more types of properties and effects than those we have envisaged here, the principles of our approach can be extended to cope with these in the manner described. Similarly, while our initial work relied on ontologies and reasoning using SWRL, there is no reliance on this approach and the models can be used with other tools or methods to the convenience of those adopting them.

So far, we have shown two main uses for this type of context/interaction modelling. The first being to support users of systems outside of their typical usage scenario and the second (described in this chapter) to enable a radical approach to emergency communications by providing information about available interactive systems and their remaining behaviours in situations where these are limited by environmental factors. This allows us to answer the sorts of questions posed in Sect. 12.2 regarding suitability of devices in given situations at a model/ontology level as well as practically as discussed in Sect. 12.4.1.

There are other possible uses of these models, for example to support dynamic configuration of multiple devices by exposing compatible interactive capabilities or for remote management of semi-autonomous systems (like the nuclear power plant from case study one for example) in hazard cases. However, we leave further discussion of these to future work.

12.6.2 Limitations and Future Work

Although we have presented a method for modelling different types of effects (simple binary effects as well as more fine-grained valued effects such as particular noise levels), there is no guarantee that we can capture and model every possible effect from an unexpected event. This is partly due to the unavailability of all information in an ongoing situation (so there may be things we do not know about which are therefore not included in the model); It may also be due to "new" effects that have not been experienced before which do not, therefore, fit into the modelling scenario; There are also some effects which may be specialised to an individual (distance of vision or acuity of hearing) or which are composed of multi-layered facets that interact with each other (rubble in front of a display may not only lead to people remaining at a certain distance from the interactive system, but may also mean they are at an angle which could also affect vision of anything displayed). Further investigations are required to further understand some of these more complex factors in order to find ways of including them in the modelling approach.

The starting point for the models and subsequent user tools shown here are an ontology. However, building and using ontologies, even with support tools, are both time-consuming and error prone. A thorough understanding of the domains and the reasoning required must be investigated prior to the ontology development. We have experimented with different approaches to the reasoning which do not rely on the ontology. We have also investigated the use of formal modelling (such as formal concept analysis) as a mechanism for supporting the development of the ontology. Both of these strands of research are ongoing.

The proof-of-concept tool suggested here in the prototype shown in Fig. 12.8 is based on a similar tool we have developed for our work in the medical domain. A fuller design project is required to analyse the requirements for this tool with relevant emergency personnel. This should be followed by usability evaluation and testing within the domain, in order to fine-tune the types of information provided and the mechanisms for requesting and updating information.

References

Back R, Mikhajlova A, von Wright J (1999) Reasoning about interactive systems. In: Formal methods 1999 (FM99). Springer, pp 1460–1476

Berners-Lee T, Hendler J, Lassila O et al (2001) The semantic web. Sci Am 284(5):28–37

Bodeau DJ, Graubart RD, Fabius-Greene J (2010) Improving cyber security and mission assurance via cyber preparedness (Cyber Prep) levels. In: IEEE international conference on social computing, pp 1147–1152

Bowen J, Hinze A (2012) Using ontologies to reason about the usability of interactive medical devices in multiple situations of use. In: ACM SIGCHI symposium on engineering interactive computing systems, EICS'12, pp 247–256

Bowen J, Reeves S (2008) Formal models for user interface design artefacts. Innov Syst Softw Eng 4(2):125–141

Bowen J, Hinze A, Reid S (2014) Model-driven tools for medical device selection. In: ACM SIGCHI symposium on engineering interactive computing systems, EICS'14, pp 129–138

Cheok AD, Rehman Mustafa AU, Fernando ONN, Barthoff AK, Wijesena IJP, Tosa N (2007) Blogwall: displaying artistic and poetic messages on public displays via sms. In: Mobile HCI. ACM, pp 483–486

Cheverst K, Davies N, Mitchell K, Friday A, Efstratiou C (2000) Developing a context-aware electronic tourist guide: some issues and experiences. In: SIGCHI conference on human factors in computing systems. ACM, pp 17–24

Comfort LK, Haase TW (2006) Communication, coherence, and collective action: the impact of Hurricane Katrina on communications infrastructure. Public Works Manage Policy 1–16

Davies N, Clinch S, Alt F (2014) Morgan & Claypool Publishers

Diehl S, Neuvel J, Zlatanova S, Scholten H (2005) Investigation of user requirements in the emergency response sector: the dutch case. In: Second Gi4DM. Springer, pp 3–19

Fortin C, DiPaola S, Hennessy K, Bizzocchi J, Neustaedter C (2013) Medium-specific properties of urban screens: towards an ontological framework for digital public displays. In: 9th ACM conference on creativity and cognition. ACM, pp 243–252

Gennari JH, Musen MA, Fergerson RW, Grosso WE, Crubzy M, Eriksson H, Noy NF, Tu SW (2002) The evolution of protg: an environment for knowledge-based systems development. Int J Hum-Comput Stud 58:89–123

Hinze A, Bainbridge D (2015) Location-triggered mobile access to a digital library of audio books using tipple. Int J Digital Libr 1–27: doi:10.1007/s00799-015-0165-z

Hinze A, Sachs K, Buchmann A (2009) Event-based applications and enabling technologies. In: Proceedings of the third ACM international conference on distributed event-based systems. ACM, pp 1–15

Horita FE, de Albuquerque JP, Degrossi LC, Mendiondo EM, Ueyama J (2015) Development of a spatial decision support system for flood risk management in Brazil that combines volunteered geographic information with wireless sensor networks. Comput Geosci 80:84–94

Horrocks I, Patel-Schneider PF, Boley H, Tabet S, Grosof B, Dean M (2004a) SWRL: a semantic web rule language combining owl and ruleml. W3c member submission

Horrocks I, Patel-Schneider PF, Boley H, Tabet S, Grosof B, Dean M, et al (2004b) SWRL: a semantic web rule language combining OWL and RuleML. W3C Member submission 21:79

Hosio S, Kukka H, Jurmu M, Ojala T, Riekki J (2010) Enhancing interactive public displays with social networking services. In: 9th international conference on mobile and ubiquitous multimedia. ACM, MUM'10, pp 23:1–23:9

Jiang X, Hong JI, Takayama L, Landay JA (2004) Ubiquitous computing for firefighters: field studies and prototypes of large displays for incident command. In: 2004 conference on human factors in computing systems. CHI, pp 679–686

Kostakos V, Jurmu M, Ojala T, Hosio S, Lindén T, Zanni D, Kruger F, Heikkinen T, Kukka H (2012) Multipurpose interactive public displays in the wild: three years later. Computer 45(5):42–49. doi:10.1109/MC.2012.115

Köster M, Schmitz M, Gehring S (2015) Gravity games: a framework for interactive space physics on media facades. In: 4th international symposium on pervasive displays. ACM, PerDis'15, pp 115–121

Lenat DB, Guha RV (1989) Building large knowledge-based systems; representation and inference in the Cyc project. Addison-Wesley Longman Publishing Co. Inc

Lu Y, Yang D (2011) Information exchange in virtual communities under extreme disaster conditions. Decis Supp Syst 50(2):529–538

Majumder A, Sajadi B (2013) Large area displays: the changing face of visualization. IEEE Comput 46(5):26–33

Ojala T, Kostakos V, Kukka H, Heikkinen T, Linden T, Jurmu M, Hosio S, Kruger F, Zanni D (2012) Multipurpose interactive public displays in the wild: three years later. Computer 45(5):42–49

Olech PS, Cernea D, Meyer H, Schoeffel S, Ebert A (2012) Digital interactive public pinboards for disaster and crisis management-concept and prototype design. In: International conference on information and knowledge engineering (IKE). WorldComp, p 1

Parush A, Ma C (2012) Team displays work, particularly with communication breakdown: performance and situation awareness in a simulated forest fire. In: Human factors and ergonomics society annual meeting, vol 56. Sage Publications

Rauschert I, Agrawal P, Sharma R, Fuhrmann S, Brewer I, MacEachren A (2002) Designing a human-centered, multimodal gis interface to support emergency management. In: Proceedings of the 10th ACM international symposium on advances in geographic information systems. ACM, pp 119–124

Robillard J, Sambrook RC (2008) Usaf emergency and incident management systems: a systematic analysis of functional requirements. Technical report, white paper for United States Air Force Space Command. http://www.uccs.edu/rsambroo/Research/EIM_REQS.pdf

Sakaki T, Okazaki M, Matsuo Y (2010) Earthquake shakes twitter users: real-time event detection by social sensors. In: 19th international conference on world wide web. ACM, pp 851–860

Schilit B, Adams N, Want R (1994) Context-aware computing applications. In: First workshop on mobile computing systems and application. IEEE, pp 85–90

Schöning J, Rohs M, Krüger A, Stasch C (2008) Improving the communication of spatial information in crisis response by combining paper maps and mobile devices. In: Mobile response. Springer, pp 57–65

Schultz CH, Koenig KL, Noji EK (1996) A medical disaster response to reduce immediate mortality after an earthquake. New Engl J Med 334(7):438–444

Thelen S, Cernea D, Olech PS, Kerren A, Ebert A (2010) DIP—a digital interactive pinboard with support for smart device interaction

Werner-Allen G, Lorincz K, Ruiz M, Marcillo O, Johnson J, Lees J, Welsh M (2006) Deploying a wireless sensor network on an active volcano. IEEE Internet Comput 10(2):18–25

Xie H, Filippidis L, Gwynne S, Galea ER, Blackshields D, Lawrence PJ (2007) Signage legibility distances as a function of observation angle. J Fire Protect Eng 17(1):41–64

Yang L, Prasanna R, King M (2009) On-site information systems design for emergency first responders. J Inf Technol Theory Appl 10(1):5–27

Yang L, Yang S, Plotnick L (2013) How the internet of things technology enhances emergency response operations. Technol Forecast Soc Change 80(9):1854–1867

Yates D, Paquette S (2011) Emergency knowledge management and social media technologies: a case study of the 2010 Haitian earthquake. Int J Inf Manag 31(1):6–13

Chapter 13
Enhanced Operator Function Model (EOFM): A Task Analytic Modeling Formalism for Including Human Behavior in the Verification of Complex Systems

Matthew L. Bolton and Ellen J. Bass

Abstract The enhanced operator function model (EOFM) is a task analytic modeling formalism that allows human behavior to be included in larger formal system models to support the formal verification of human interactive systems. EOFM is an expressive formalism that captures the behavior of individual humans or, with the EOFM with communications (EOFMC) extension, teams of humans as a collection of tasks, each composed representing a hierarchy of activities and actions. Further, EOFM has a formal semantics and associated translator that allow its represented behavior to be automatically translated into a model checking formalism for use in larger system verification. EOFM supports a number of features that enable analysts to use model checking to investigate human-automation and human-human interaction. Translator variants support the development of different task models with methods for accounting for erroneous human behaviors and miscommunications, the creation of specification properties, and the automated design of human-machine interfaces. This chapter provides an overview of EOFM, its language, its formal semantics and translation, and analysis features. It addresses the different ways that EOFM has been used to evaluate human behavior in human-interactive systems. We demonstrate some of the capabilities of EOFM by using it to evaluate the air traffic control case study. Finally, we discuss future directions of EOFM and its supported analyses.

M.L. Bolton (✉)
University at Buffalo, State University of New York, Buffalo, NY, USA
e-mail: mbolton@buffalo.edu

E.J. Bass
Drexel University, Philadelphia, PA, USA
e-mail: ejb96@drexel.edu

B. Weyers et al. (eds.), *The Handbook of Formal Methods in Human-Computer Interaction*, Human-Computer Interaction Series,
DOI 10.1007/978-3-319-51838-1_13

13.1 Introduction

In complex systems, failures often occur as a result of unexpected interactions between components including contributions from human behaviors (Hollnagel 1993; Perrow 1999; Reason 1990; Sheridan and Parasuraman 2005). To design and analyze complex systems, human factors engineers use task analysis (Kirwan and Ainsworth 1992) to describe required normative human behaviors. Erroneous human behavior models provide engineers with means for exploring the potential impact of human error (Hollnagel 1993; Reason 1990). To support design and analysis, Enhanced Operator Function Model (EOFM) (Bolton et al. 2011), based on the Operator Function Model (Mitchell and Miller 1986), was developed to allow engineers to represent task analytic human behavior formally and to include both normative and erroneous human behavior in formal verification analyses. EOFM with communication (EOFMC) further supports models with coordination and communication among a team of people (Bass et al. 2011).

The analyses supported by EOFM and EOFMC have evolved from older techniques that use formal verification to evaluate human-automation interaction (Bolton et al. 2013). In particular, EOFM is similar to techniques that attempt to include human behavior in formal verification analyses by using formal interpretations of task analytic models. As such, EOFM supports similar sorts of evaluations offered by other systems such as AMBOSS (Giese et al. 2008), ConcurTaskTrees (Aït-Ameur and Baron 2006; Paternò and Santoro 2001; Paternò et al. 1997), User Action Notation (Hartson et al. 1990; Palanque et al. 1996), HAMSTER (Martinie et al. 2011, 2014), and various approaches that require task concepts to be directly represented in other modeling formalisms (Basnyat et al. 2007; Degani et al. 1999; Fields 2001; Gunter et al. 2009). However, EOFM and EOFMC distinguish themselves by the rich feature set and the analyses they support. EOFM and EOFMC have formal semantics that give the task behaviors specified by the XML language unambiguous, mathematical meanings. The semantics serve as the basis for a series of translators that automatically convert EOFMs, written in XML, into formal models that can be used by a model checker. These translators support a number of different features that generate alternate task models with erroneous behavior, properties that can be model checked, and interface designs.[1] A deeper discussion about the place of EOFM in the larger formal HAI literature can be found in Bolton et al. (2011, 2013).

This chapter provides a general overview of EOFM and EOFMC. This includes a description of the EOFM-supported analysis process, EOFM and EOFMC syntax, formal semantics, translation, and a description of different analyses that leverage EOFM and EOFMC. As concepts are introduced, we illustrate some of EOFMC's capabilities with a variation of the air traffic control case study. Below, the case study is introduced. This is followed by a discussion of EOFM and its capabilities along

[1] The EOFM language specifications, translators, tools, documentation, and examples are freely available at http://fhsl.eng.buffalo.edu/EOFM/.

with applications to the case study. Finally, both the case study analysis results and EOFM are generally discussed with pointers to additional applications and descriptions of future research directions.

13.2 Case Study

To illustrate how EOFM can be used to evaluate a safety critical procedure involving human-human communication and coordination, we present a variation of Case Study 2. Specifically, we present an EOFMC model where an aircraft heading clearance is communicated by an air traffic controller (ATCo) to two pilots: the pilot flying (PF) and the pilot monitoring (PM).

In this scenario, both the pilots and the air traffic controller have push-to-talk switches which they press down when they want to verbally communicate information to each other over the radio. They can release this switch to end communication.

For the aircraft, the Autopilot Flight Director System consists of Flight Control Computers and the Mode Control Panel (MCP). The MCP provides control of the Autopilot (A/P), Flight Director, and the Autothrottle system. When the A/P is engaged, the MCP sends commands to the aircraft pitch and roll servos to operate the aircraft flight control surfaces. Herein the MCP is used to activate heading changes. The Heading (HDG)/Tracking (TRK) window of the MCP displays the selected heading or track (Fig. 13.1). The numeric display shows the current desired heading in compass degrees (from 0 and 359). Below the HDG window is the heading select knob. Changes in the heading are achieved by rotating and pulling the knob. Pulling the knob tells the autopilot to use the selected value and engages the HDG mode.

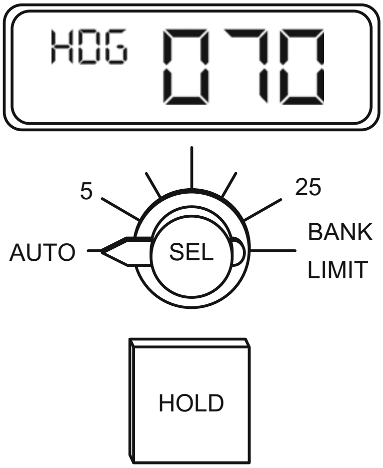


Fig. 13.1 Heading control and display

The following describes a communication protocol designed to ensure that this heading is correctly communicated from the ATCo to the two pilots:

1. The air traffic controller contacts the pilots and gives them a new heading clearance.
2. The PF starts the process of dialing the new heading into the heading window.
3. The pilots then attempt to confirm that the correct heading was entered. They do this by having the PM point at the heading window, contact the ATCo, and repeat back the heading the PM heard from the ATCo.
4. If the heading the PF hears read back to the air traffic controller does not match the heading he or she entered in the heading window, the PF enters the heading heard from the PM during the read back. The PM then points at the heading window again and repeats the heading he or she originally heard from the ATCo. This process (step 4) repeats while the heading window heading does not match what the PM heard from the ATCo. It completes if the heading the PF heard from the PM matches what is in the heading window.
5. If the heading the ATCo hears read back from the pilots (from step 3) does not match the heading the air traffic controller intended, then the entire process (starting at step 1) needs to be repeated until the correct heading is read back.
6. The PF engages the entered heading.

In what follows, we will show how EOFM concepts can be applied to this application and various EOFM features are introduced.

13.3 Enhanced Operator Function Model (EOFM) and EOFM with Communication (EOFMC)

EOFM and EOFMC (henceforth collectively referred to as EOFM except where additional clarification about EOFMC is required) are task analytic modeling languages for representing human task behavior. The EOFM languages support the formal verification approach shown in Fig. 13.2. An analyst examines target system information (i.e. design document, observational data, manuals) to manually

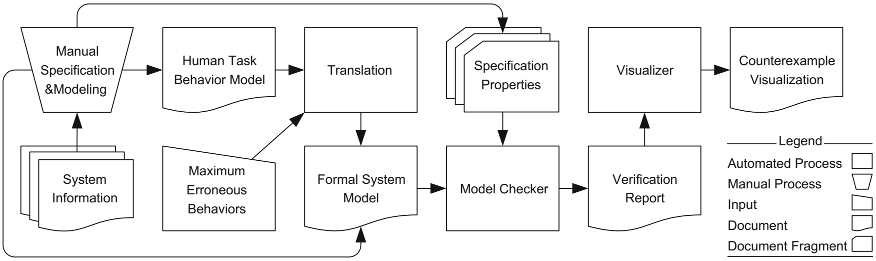


Fig. 13.2 Flow diagram showing how the verification method supported by EOFM works

model normative human operator behavior (which can be a single human operator or a team of human operators) using a task analytic representation, a formal system model (absent human operator behavior), and specification properties that he or she wants to check. The task analytic model is then automatically translated into the larger formal model by a translation process. As part of this translation process, the analyst can specify a maximum number of erroneous behaviors that will be modeled as optional paths through the formal representation of the human operator task behavior. The formal system model and the specifications are then run through the model checker that produces a verification report. If a violation of the specification is found, the report will contain a counterexample. Our visualizer uses the counterexample and the original human task behavior model to illustrate the sequence of human behaviors and related system states that led to the violation.

EOFM has an XML syntax that supports the hierarchical concepts compatible with task analytic models. This represents task behavior as a hierarchy of goal-directed activities (representing the behaviors and strategic knowledge required to accomplish a goal) that can decompose into sub-activities (for achieving sub-goals) and, at the bottom of the hierarchy, atomic actions (specific things the human operator(s) can do to the environment). The language allows for the modeling of human behavior, either individuals or groups, as an input/output system. Inputs may come from other elements of the system like human-device interfaces or the environment. Output variables are human actions. The operators' task models describe how human actions may be generated based on input and local variables (representing perceptual or cognitive processing as well as group coordination and communication). To support domain specific information, variables are defined in terms of constants, analyst-defined types, and basic types (reals, integers, and Boolean values).

To represent the strategic knowledge associated with procedural activities, activities can have preconditions, repeat conditions, and completion conditions (Boolean expressions written in terms of input, output, and local variables as well as constants) that specify what must be true before an activity can execute, when it can execute again, and what is true when it has completed execution respectively. Activities are decomposed into lower-level sub-activities and, finally, actions. Decomposition operators specify how many sub-activities or actions can execute and what the temporal relationships are between them. Actions are either an assignment to an output variable (indicating an action has been performed) or a local variable (representing a perceptual, cognitive, or group communication action). Optionally an action can include a value so that the human operator can set a value as part of the action.

All tasks in an EOFM descend from a top level activity, where there can be many tasks. Tasks can either belong to one human operator, or, in EOFMC, they can be shared among human operators. A shared task must be associated with two or more associates, and a subset of associates for the general task can be identified for each activity. This makes it explicit which human operators are participating in which activity. While the activities in these shared tasks can decompose in the same ways as their single-operator counterparts, they can explicitly include human-human communication (a non-shared activity cannot). For communication, a value (communicated information) from one human operator can be received by one or more other human operators (modeled as an update to a local variable).

13.3.1 Syntax

The EOFM language's XML syntax is defined using the REgular LAnguage for XML Next Generation (RELAX NG) standard (Clark and Murata 2001) (see Figs. 13.3, 13.4 and 13.5). These provide an explicit description of the EOFM language and can thus be employed by analysts when developing their own EOFM and EOFMC models.[2] Below we describe the EOFMC syntax. An example of the syntax used in an actual model is shown in Fig. 13.7, with highlights showing specific features.

XML documents contain a single root node whose attributes and sub-nodes define the document. For the EOFM, the root node is called eofms. The next level of the hierarchy has zero or more constant nodes, zero or more userdefinedtype nodes, and one or more humanoperator nodes. The userdefinedtype nodes define enumerated types useful for representing operational environment, human-device interface, and human mission concepts. A userdefinedtype node is composed of a unique name attribute (by which it can be referenced) and a string of data representing the type construction (the syntax of which is application dependent). A constant node is defined by a unique name attribute, either a userdefinedtype attribute (the name attribute of a userdefinedtype node) or basictype attribute. A constant node also allows for the definition of constant mappings and functions through the addition of optional (zero or more) parameter attributes for defining function arguments.

The humanoperator nodes represent the task behavior of the different human operators. Each humanoperator has zero or more input variables (inputvariable nodes and inputvariablelink nodes for variables shared with other human operators), zero or more local variables (localvariable nodes), one or more human action output variables (humanaction nodes) or human communication actions (humancomaction nodes), and one or more task models (eofm nodes). A human action (a humanaction node) describes a single, observable, atomic act that a human operator can perform. A humanaction node is defined by a unique name attribute and a behavior attribute which can have one of three values: autoreset (for modeling a single discrete action such as flipping a switch), toggle (for modeling an action that must be started and stopped as separate discrete events such as starting to hold down a button and then releasing it), or setvalue for committing a value in a single action. This type of human action has an additional attribute that specifies the type of the value being set

[2] Note that although EOFM does not have its own language creation tool, the RELAX NG language specification allows professional XML development environments (such as oXygen XML Editor; https://www.oxygenxml.com/) to facilitate rapid model development with code completion and syntax checking capabilities.

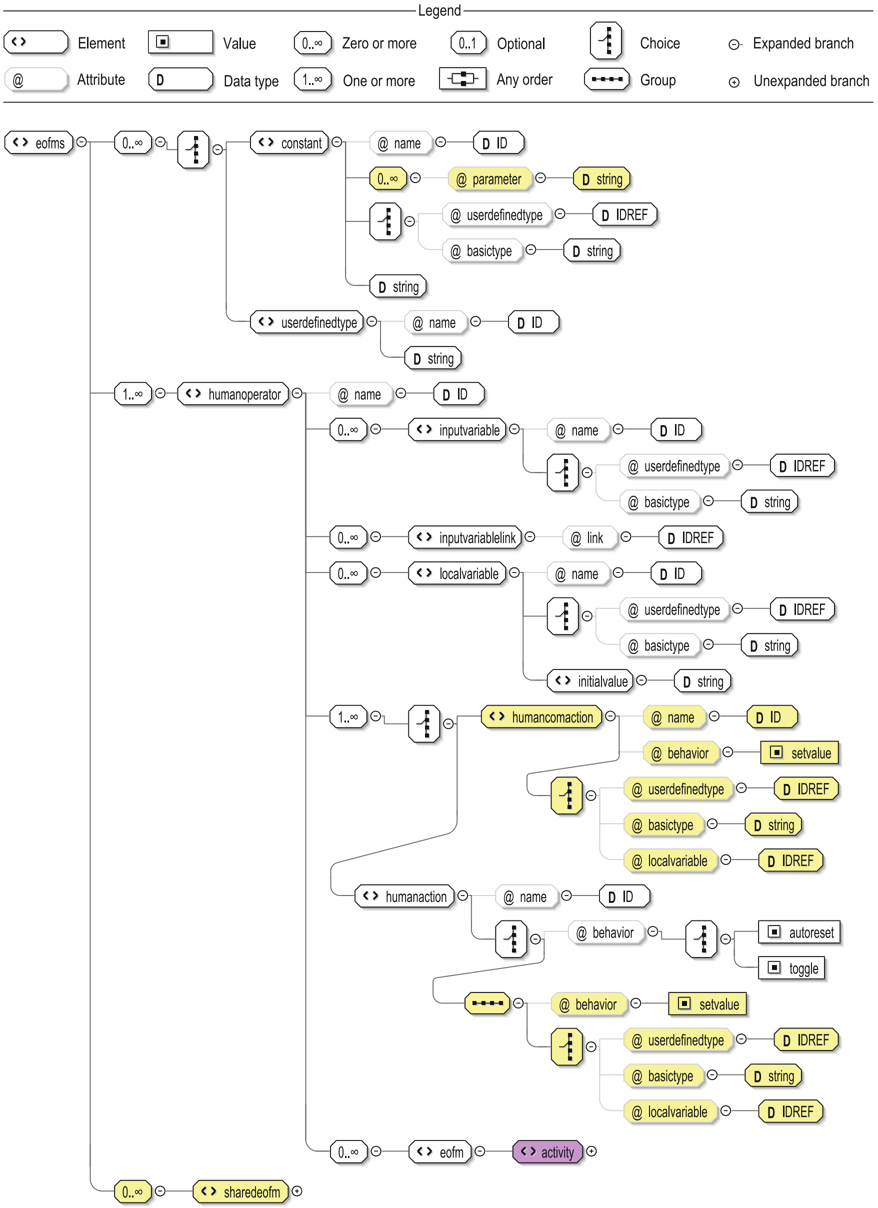


Fig. 13.3 A visualization of the EOFM Relax NG language specification that species EOFM syntax (see Syncro Soft 2016 for more details). *Yellow* indicates constructs added to EOFM for EOFMC. *Other colors* are used to indicate nodes representing identically defined syntax elements across Figs. 13.3, 13.4 and 13.5. The language specification for activity and sharedeofm nodes can be found in Figs. 13.4 and 13.5 respectively

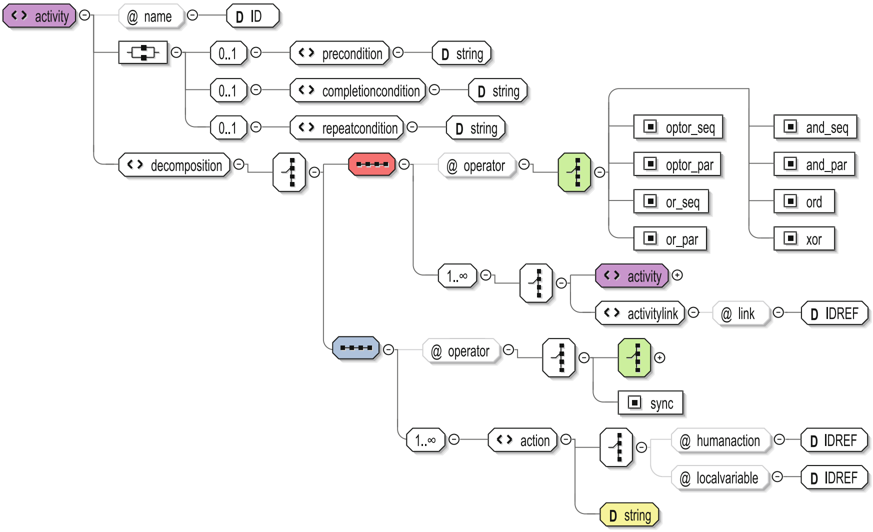


Fig. 13.4 Syntax visualization of the activity node from Fig. 13.3

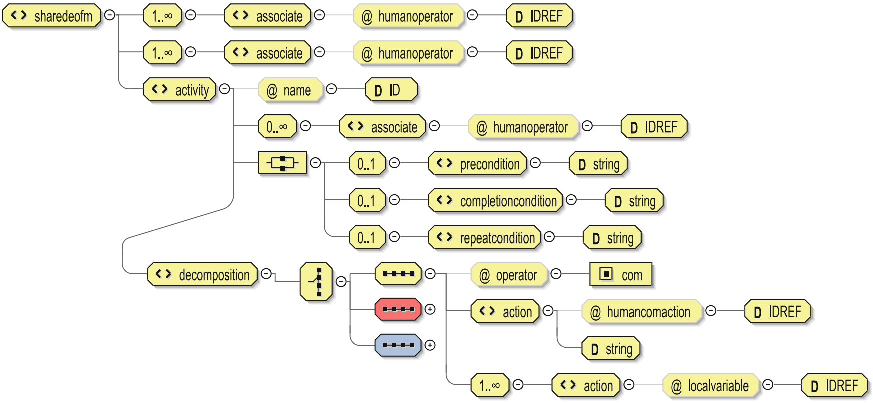


Fig. 13.5 Syntax visualization of the sharedeofm node from Fig. 13.3

(a reference to a userdefinedtype or a basictype) or a reference to a local variable which will contain the value to be set. A human communication action (a humancomaction node) describes a special type of setvalue action that a human operator can use to communicate something (a value or the value stored in an associated variable) to one or more other human operators.

Input variables (inputvariable nodes) are composed of a unique name attribute and either a userdefinedtype or basictype attribute (defined as in the constant node). To support the definition of inputs that can be perceived concurrently by multiple human operators (for example two human operators hearing the same alarm issued by an automated system) the inputvariablelink node allows a humanoperator node to access input variables defined in a different humanoperator node using the same input variable name. Local variables are represented by localvariable nodes, themselves defined with the same attributes as an inputvariable or constant node, with an additional sub-node, initialvalue, a data string with the variable's default initial value.

The task behaviors of a human operator are defined using eofm nodes. One eofm node is defined for each goal directed task behavior. The tasks are defined in terms of activity and action nodes. An activity node is represented by a unique name attribute, a set of optional conditions, and a decomposition node. Condition nodes contain a Boolean expression (in terms of variables and human actions) with a string that constrains the activity's execution. The following conditions are supported:

- *Precondition* (precondition in the XML): criterion to start executing;
- *RepeatCondition* (repeatcondition in the XML): criterion to repeat execution; and
- *CompletionCondition* (completioncondition in the XML): criterion to complete execution.

An activity's decomposition node is defined by a decomposition operator (an operator attribute) and a set of activities (activity or activitylink nodes) or actions (action nodes). The decomposition operator (Table 13.1) controls the cardinal and

Table 13.1 Decomposition operators

Operator	Description
optor_seq	Zero or more of the activities or actions in the decomposition must execute in any order one at a time
optor_par	Zero or more of the activities or actions in the decomposition must execute in any order and can execute in parallel
or_seq	One or more of the activities or actions in the decomposition must execute in any order one at a time
or_par	One or more of the activities or actions in the decomposition must execute in any order and can execute in parallel
and_seq	All of the activities or actions in the decomposition must execute in any order one at a time
and_par	All of the activities or actions in the decomposition must execute in any order and can execute in parallel
xor	Exactly one activities or actions in the decomposition must execute
ord	All activities or actions in the decomposition must execute in the order they appear
sync	All activities or actions in the decomposition must execute synchronously
com	All activities or actions in the decomposition must execute synchronously, where information is transferred between human operators

temporal execution relationships between the sub-activity and action nodes in the decomposition (its children). The EOFM language implements the following decomposition operators: and, or, optor, xor, ord, and sync. The com decomposition operator is exclusive to EOFMC. Some operators have two modalities: sequential (suffixed_seq) and parallel (suffixed_par). For the sequential mode, the activities or actions must be executed one at a time. In parallel mode, the execution of activities and action in the decomposition may overlap in any manner. For the xor, ord, sync, and com decomposition operators, only one modality can be defined: xor and ord are always sequential and sync and com are always parallel.

The activity nodes represent lower-level sub-activities and are defined identically to those higher in the hierarchy. Activity links (activitylink nodes) allow for reuse of model structures by linking to existing activities via a link attribute which names the linked activity node.

The lowest level of the task model hierarchy is represented by either observable, atomic human actions or internal (cognitive or perceptual) ones, all using the action node. For an observable human action, the name of a humanaction node is listed in the humanaction attribute. For an internal human action, the valuation of a local variable is specified by providing the name of the local variable in the localvariable attribute and the assigned value within the node itself.

Shared tasks are defined in sharedeofm nodes. Each sharedeofm contains two or more associate nodes explicitly defining which human operators collaborate to perform the shared tasks defined within these nodes. Tasks themselves are represented with the same hierarchy of activities and actions as in individual human operator tasks. The associates of each activity must be a subset of the associates of its ancestors. If no associates are defined in an activity, the associates are inherited from its parent node. Each action node is associated with a humanaction, localvariable, or communicationaction defined under the different human operators. Thus an action in a shared task is already affiliated with a humanoperator and does not require an explicitly defined associate.

While the activities in these shared tasks can decompose in the same ways as their single-operator counterparts, they have an additional decomposition option. The com decomposition operator explicitly models human-human communication. Assuming the associates are participating in the communication, com is a special form of the sync decomposition operator and assumes information is being transferred between human operators. The decomposition must start with the performance of a humancomaction, which commits a value (communicated information either explicitly defined in the XML markup or from a variable originally associated with the humancomaction when it was defined). The decomposition ends with one or more actions explicitly pointing to local variables to allow other human operators to register the information communicated via the humancomaction.

13.3.2 Visual Notation

The structure of an instantiated[3] EOFMC's task behaviors can be represented visually as a tree-like graph structure, where actions are depicted by rectangular nodes and activities by rounded rectangle nodes (see Fig. 13.6). In these visualizations, conditions are connected to the activity they modify: a *Precondition* is represented by a yellow, downward pointing triangle connected to the left side of the activity; a *CompletionCondition* is presented as a magenta, upward pointing triangle connected to the right of the activity; and a *RepeatCondition* is conveyed as a recursive arrow attached to the top of the activity. These standard colors are used for condition shapes to help distinguish them from each other and the other task structures. A decomposition is presented as an arrow, labeled with the decomposition operator, extending below an activity that points to a large rounded rectangle containing the decomposed activities or actions. Activities are labeled with the associated activity name. Actions are labeled with the name of the associated humanaction, localvariable, or humancomaction name. Values committed by a humanaction with set value behavior, assigned to a localvariable assignment, or communicated by a humancomaction are shown in bold below this label. Communication actions are presented in the same decomposition as the local variables used for receiving the communication. Arrows are used to show how the communicated value is sent to the other local variables in the decomposition. These visualizations can be automatically generated from the XML source, as is currently done with a MS Visio plugin we have created (Bolton and Bass 2010c). The visualizations are useful because they help communicate encapsulated task concepts in publications and presentations. They also facilitate counterexample visualization, something discussed in more depth in Sect. 13.3.8.

13.3.3 Case Study Model

We implemented the task from our communication protocol application in EOFMC. Figure 13.6 depicts its visualization while Fig. 13.7 shows the corresponding XML. The entire process starts when the ATCo has a new, correct clearance to communicate to the pilots (lATCoClearance = CorrectHeading; the precondition to aChangeHeading). This allows for the performance of aChangeHeading and its first subactivity aCommandAndConfirm. The ATCo performs the aToggleATCoTalk activity by pressing his push-to-talk switch (hATCoToggleTalkSwitch). Then, the ATCo communicates the heading (aToggleATCoTalk[4]) to the pilots (lATCoClearance via the sATCoTalk human communication action), such as "AC1 Heading 070 for spacing." Both pilots remember this heading (stored in the local variables lPFHead-

[3] An EOFM model created using the EOFM XML-based language.

[4] In this example, all headings are modeled abstractly as either being CorrectHeading, if it matches the heading clearance the ATCo intended to communicate, or IncorrectHeading, if it does not.

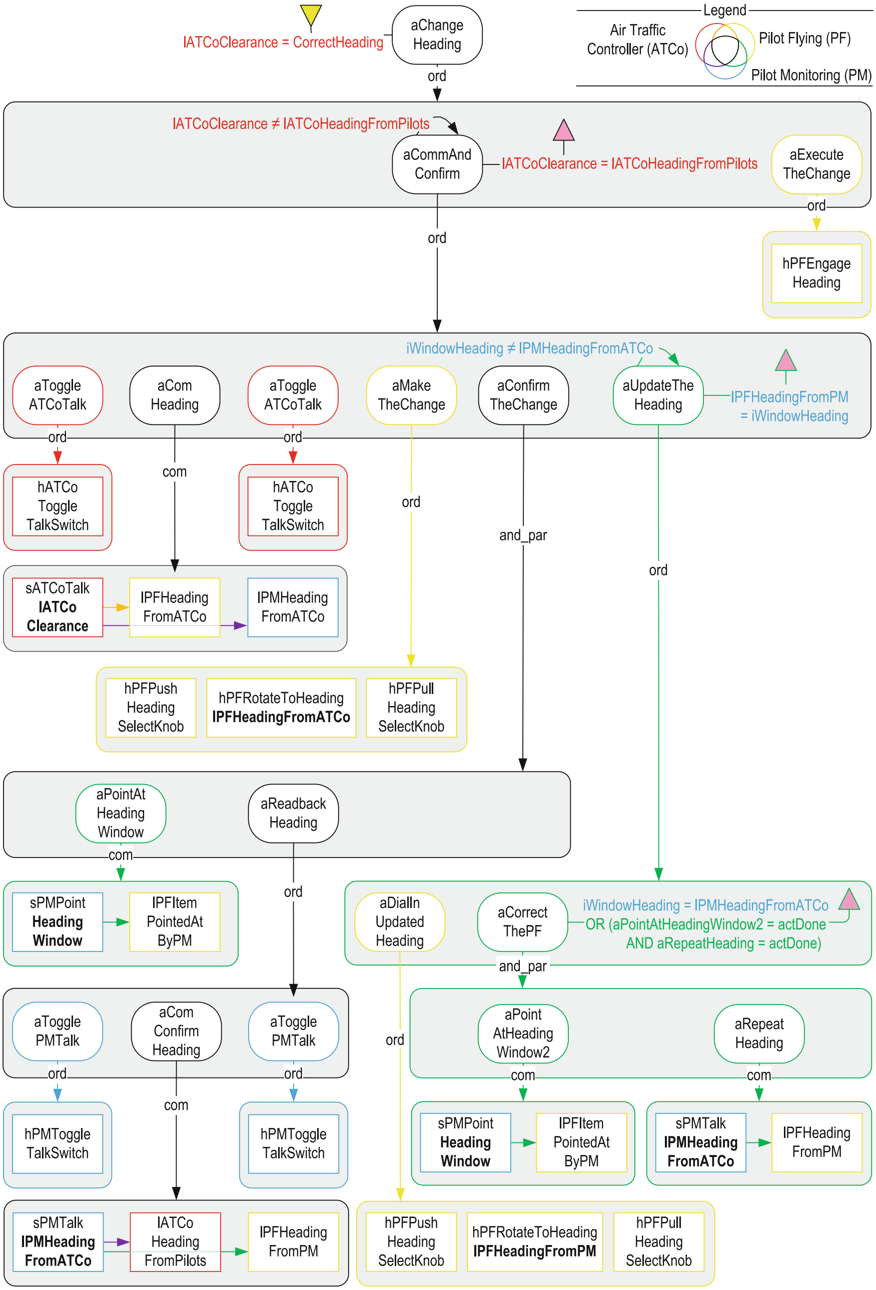
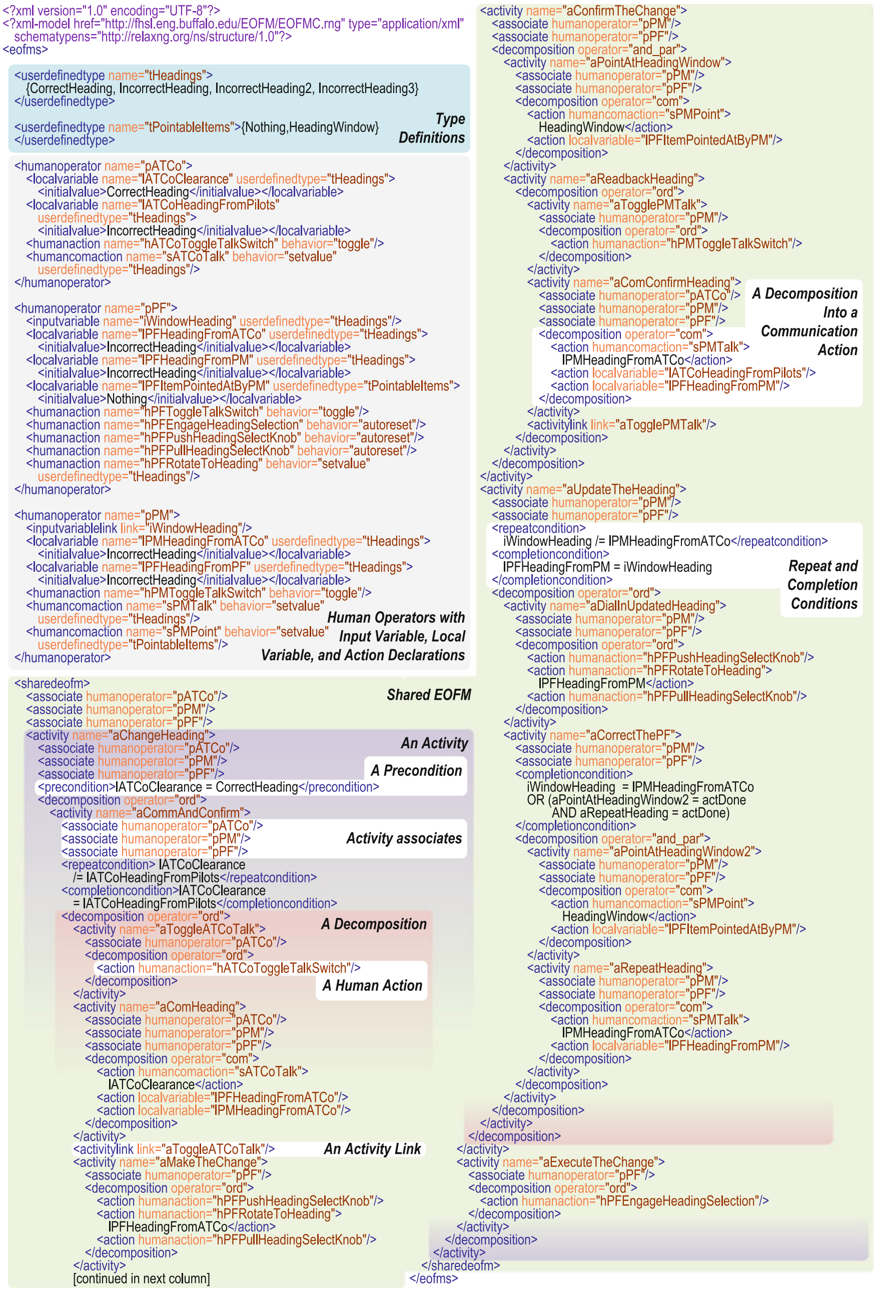


Fig. 13.6 Visualization of the instantiated EOFMC communication protocol from Fig. 13.7. The task is *colored* based on the associates (human operators) performing the task (see the legend)



Type Definitions
Human Operators with Input Variable, Local Variable, and Action Declarations
Shared EOFM
An Activity
A Precondition
Activity associates
A Decomposition
A Human Action
An Activity Link
A Decomposition Into a Communication Action
Repeat and Completion Conditions

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-model href="http://fhsl.eng.buffalo.edu/EOFM/EOFMC.rng" type="application/xml"
  schematypens="http://relaxng.org/ns/structure/1.0"?>
<eofms>
  <userdefinedtype name="tHeadings">
    {CorrectHeading, IncorrectHeading, IncorrectHeading2, IncorrectHeading3}
  </userdefinedtype>

  <userdefinedtype name="tPointableItems">{Nothing,HeadingWindow}
  </userdefinedtype>
  <humanoperator name="pATCo">
    <localvariable name="lATCoClearance" userdefinedtype="tHeadings">
      <initialvalue>CorrectHeading</initialvalue></localvariable>
    <localvariable name="lATCoHeadingFromPilots"
      userdefinedtype="tHeadings">
      <initialvalue>IncorrectHeading</initialvalue></localvariable>
    <humanaction name="hATCoToggleTalkSwitch" behavior="toggle"/>
    <humancomaction name="sATCoTalk" behavior="setvalue"
      userdefinedtype="tHeadings"/>
  </humanoperator>

  <humanoperator name="pPF">
    <inputvariable name="iWindowHeading" userdefinedtype="tHeadings"/>
    <localvariable name="lPFHeadingFromATCo" userdefinedtype="tHeadings">
      <initialvalue>IncorrectHeading</initialvalue></localvariable>
    <localvariable name="lPFHeadingFromPM" userdefinedtype="tHeadings">
      <initialvalue>IncorrectHeading</initialvalue></localvariable>
    <localvariable name="lPFItemPointedAtByPM" userdefinedtype="tPointableItems">
      <initialvalue>Nothing</initialvalue></localvariable>
    <humanaction name="hPFToggleTalkSwitch" behavior="toggle"/>
    <humanaction name="hPFEngageHeadingSelection" behavior="autoreset"/>
    <humanaction name="hPFPushHeadingSelectKnob" behavior="autoreset"/>
    <humanaction name="hPFPullHeadingSelectKnob" behavior="autoreset"/>
    <humanaction name="hPFRotateToHeading" behavior="setvalue"
      userdefinedtype="tHeadings"/>
  </humanoperator>

  <humanoperator name="pPM">
    <inputvariablelink link="iWindowHeading"/>
    <localvariable name="lPMHeadingFromATCo" userdefinedtype="tHeadings">
      <initialvalue>IncorrectHeading</initialvalue></localvariable>
    <localvariable name="lPFHeadingFromPF" userdefinedtype="tHeadings">
      <initialvalue>IncorrectHeading</initialvalue></localvariable>
    <humanaction name="hPMToggleTalkSwitch" behavior="toggle"/>
    <humancomaction name="sPMTalk" behavior="setvalue"
      userdefinedtype="tHeadings"/>
    <humancomaction name="sPMPoint" behavior="setvalue"
      userdefinedtype="tPointableItems"/>
  </humanoperator>
  <sharedeofm>
    <associate humanoperator="pATCo"/>
    <associate humanoperator="pPM"/>
    <associate humanoperator="pPF"/>
    <activity name="aChangeHeading">
      <associate humanoperator="pATCo"/>
      <associate humanoperator="pPM"/>
      <associate humanoperator="pPF"/>
      <precondition>lATCoClearance = CorrectHeading</precondition>
      <decomposition operator="ord">
        <activity name="aCommAndConfirm">
          <associate humanoperator="pATCo"/>
          <associate humanoperator="pPM"/>
          <associate humanoperator="pPF"/>
          <repeatcondition> lATCoClearance
            /= lATCoHeadingFromPilots</repeatcondition>
          <completioncondition>lATCoClearance
            = lATCoHeadingFromPilots</completioncondition>
          <decomposition operator="ord">
            <activity name="aToggleATCoTalk">
              <associate humanoperator="pATCo"/>
              <decomposition operator="ord">
                <action humanaction="hATCoToggleTalkSwitch"/>
              </decomposition>
            </activity>
            <activity name="aComHeading">
              <associate humanoperator="pATCo"/>
              <associate humanoperator="pPM"/>
              <associate humanoperator="pPF"/>
              <decomposition operator="com">
                <action humancomaction="sATCoTalk">
                  lATCoClearance</action>
                <action localvariable="lPFHeadingFromATCo"/>
                <action localvariable="lPMHeadingFromATCo"/>
              </decomposition>
            </activity>
            <activitylink link="aToggleATCoTalk"/>
            <activity name="aMakeTheChange">
              <associate humanoperator="pPF"/>
              <decomposition operator="ord">
                <action humanaction="hPFPushHeadingSelectKnob"/>
                <action humanaction="hPFRotateToHeading">
                  lPFHeadingFromATCo</action>
                <action humanaction="hPFPullHeadingSelectKnob"/>
              </decomposition>
            </activity>
            [continued in next column]
            <activity name="aConfirmTheChange">
              <associate humanoperator="pPM"/>
              <associate humanoperator="pPF"/>
              <decomposition operator="and_par">
                <activity name="aPointAtHeadingWindow">
                  <associate humanoperator="pPM"/>
                  <associate humanoperator="pPF"/>
                  <decomposition operator="com">
                    <action humancomaction="sPMPoint">
                      HeadingWindow</action>
                    <action localvariable="lPFItemPointedAtByPM"/>
                  </decomposition>
                </activity>
                <activity name="aReadbackHeading">
                  <decomposition operator="ord">
                    <activity name="aTogglePMTalk">
                      <associate humanoperator="pPM"/>
                      <decomposition operator="ord">
                        <action humanaction="hPMToggleTalkSwitch"/>
                      </decomposition>
                    </activity>
                    <activity name="aComConfirmHeading">
                      <associate humanoperator="pATCo"/>
                      <associate humanoperator="pPM"/>
                      <associate humanoperator="pPF"/>
                      <decomposition operator="com">
                        <action humancomaction="sPMTalk">
                          lPMHeadingFromATCo</action>
                        <action localvariable="lATCoHeadingFromPilots"/>
                        <action localvariable="lPFHeadingFromPM"/>
                      </decomposition>
                    </activity>
                    <activitylink link="aTogglePMTalk"/>
                  </decomposition>
                </activity>
              </decomposition>
            </activity>
            <activity name="aUpdateTheHeading">
              <associate humanoperator="pPM"/>
              <associate humanoperator="pPF"/>
              <repeatcondition>
                iWindowHeading /= lPMHeadingFromATCo</repeatcondition>
              <completioncondition>
                lPFHeadingFromPM = iWindowHeading
              </completioncondition>
              <decomposition operator="ord">
                <activity name="aDialInUpdatedHeading">
                  <associate humanoperator="pPM"/>
                  <associate humanoperator="pPF"/>
                  <decomposition operator="ord">
                    <action humanaction="hPFPushHeadingSelectKnob"/>
                    <action humanaction="hPFRotateToHeading">
                      lPFHeadingFromPM</action>
                    <action humanaction="hPFPullHeadingSelectKnob"/>
                  </decomposition>
                </activity>
                <activity name="aCorrectThePF">
                  <associate humanoperator="pPM"/>
                  <associate humanoperator="pPF"/>
                  <completioncondition>
                    iWindowHeading  = lPMHeadingFromATCo
                    OR (aPointAtHeadingWindow2 = actDone
                        AND aRepeatHeading = actDone)
                  </completioncondition>
                  <decomposition operator="and_par">
                    <activity name="aPointAtHeadingWindow2">
                      <associate humanoperator="pPM"/>
                      <associate humanoperator="pPF"/>
                      <decomposition operator="com">
                        <action humancomaction="sPMPoint">
                          HeadingWindow</action>
                        <action localvariable="lPFItemPointedAtByPM"/>
                      </decomposition>
                    </activity>
                    <activity name="aRepeatHeading">
                      <associate humanoperator="pPM"/>
                      <associate humanoperator="pPF"/>
                      <decomposition operator="com">
                        <action humancomaction="sPMTalk">
                          lPMHeadingFromATCo</action>
                        <action localvariable="lPFHeadingFromPM"/>
                      </decomposition>
                    </activity>
                  </decomposition>
                </activity>
              </decomposition>
            </activity>
          </decomposition>
        </activity>
        <activity name="aExecuteTheChange">
          <associate humanoperator="pPF"/>
          <decomposition operator="ord">
            <action humanaction="hPFEngageHeadingSelection"/>
          </decomposition>
        </activity>
      </decomposition>
    </activity>
  </sharedeofm>
</eofms>
```

Fig. 13.7 EOFM XML for the communication protocol application. Specific EOFM language features are highlighted and labeled with *bolded italic* text

ingFromATCo and lPMHeadingFromATCo for the PF and PM respectively). The ATCo releases the switch (under the second instance of aToggleATCoTalk; an activitylink in Fig. 13.7). Then, the PF performs the procedure for changing the heading in the heading window (aMakeTheChange): pushing the heading select knob (hPFPushHeadingSelectKnob), rotating it (hPFRotateToHeading) to the heading that the PF heard from the ATCo (lPFHeadingFromATCo), and pulling the heading select knob (hPFPullHeadingSelectKnob).

Next, the pilots perform a read back procedure (aConfirmTheChange). They do this by having the PM point at the heading window value (which can be observed by the PF) and the PM performs the procedure for repeating back the heading that the PM heard from the ATCo to the PF and the ATCo.

If the heading in the window does not match the heading the PF just heard read back to the ATCo, the pilots must collaborate to update the heading (aUpdateTheHeading). To do this, the PF enters the heading he or she heard read back to the ATCo by the PM (aCorrectThePF). If this heading in the window still does not match the heading the PM heard from the ATCo, the PM again points at the heading window and repeats the heading he or she heard from the ATCo back to the PF (all under aCorrectThePF). This entire process repeats as long as the heading window does not match the heading the PM heard from the ATCo.

If the clearance the ATCo heard read back from the pilots does not match what the ATCo intended, aCommAndConfirm must repeat until this is true. Once this is true, the PF engages the entered heading.

13.3.4 Formal Semantics

We now formally describe the semantics of the EOFM language's task models: explicitly defining how and when each activity and action in a task structure is *Executing*.

An activity's or action's execution is controlled by how it transitions between three discrete states:

- *Ready*: the initial (inactive) state which indicates that the activity or action is waiting to execute;
- *Executing*: the active state which indicates that the activity or action is executing; and
- *Done*: the secondary (inactive) state which indicates that the activity has finished executing.

While *Precondition*s, *RepeatCondition*s, and *CompletionCondition*s can be used to describe when activities and actions transition between these execution states, three additional conditions are required. These conditions support transitions based on the activity's or action's position in the task structure, the execution state of its parent, children (activities or actions into which the activity decomposes), and siblings (activities or actions contained within the same decomposition).

- *StartCondition*: implicit condition that triggers the start of an activity or action defined in terms of the execution states of its parent and siblings.
- *EndCondition*: implicit condition to end the execution of an activity or action defined in terms of the execution state of its children.
- *Reset*: implicit condition to reset an activity (have it return to the *Ready* execution state).

For any given activity or action in a decomposition, a *StartCondition* is comprised of two conjuncts with one stipulating conditions on the execution state of its parent and the other on the execution state of its siblings based on the parent's decomposition operator, generally formulated as:

$$(parent.state = Executing) \wedge \bigwedge_{\forall siblings\ s} (s.state \neq Executing)$$

This is formulated differently depending on the circumstances. If the parent's decomposition operator has a parallel modality, the second conjunct is eliminated. If the parent's decomposition operator is ord, the second conjunct is reformulated to impose restrictions only on the previous sibling in the decomposition order: $(prev\_sibling.state = Done)$. If it is the xor decomposition operator, the second conjunct is modified to enforce the condition that no other sibling can execute after one has finished:

$$\bigwedge_{\forall siblings\ s} (s.state = Ready)$$

An activity without a parent (a top level activity) will eliminate the first conjunct. Top level activities that are defined for a given humanoperator treat each other as siblings in the formulation of the second conjunct with an assumed and_seq relationship. All other activities are treated as if they are in an and_par relationship and are thus not considered in the formulation of the *StartCondition*. Top level activities that are defined for sharedeofms treat all other activities as if they are in an and_par relationship, thus they have *StartCondition*s that are always *true*.

An *EndCondition* is also comprised of two conjuncts both related to an activity's children. Since an action has no children, an action's *EndCondition* defaults to *true*. The first conjunct asserts that the execution states of the activity's children satisfy the requirements stipulated by the activity's decomposition operator. The second asserts that none of the children are *Executing*. This is generically expressed as follows:

$$\left(\bigoplus_{\forall subacts\ c} (c.state = Done) \right) \wedge \bigwedge_{\forall subacts\ c} (c.state \neq Executing)$$

In the first conjunct, $\bigoplus$ (a generic operator) is to be substituted with $\wedge$ if the activity has the and_seq, and_par, sync, or com decomposition operator; and $\vee$ if the activity has the or_seq, or_par, or xor decomposition operator. Since optor_seq and optor_par enforce no restrictions, the first conjunct is eliminated when the activity has either of these decomposition operators. When the activity has the ord decompo-

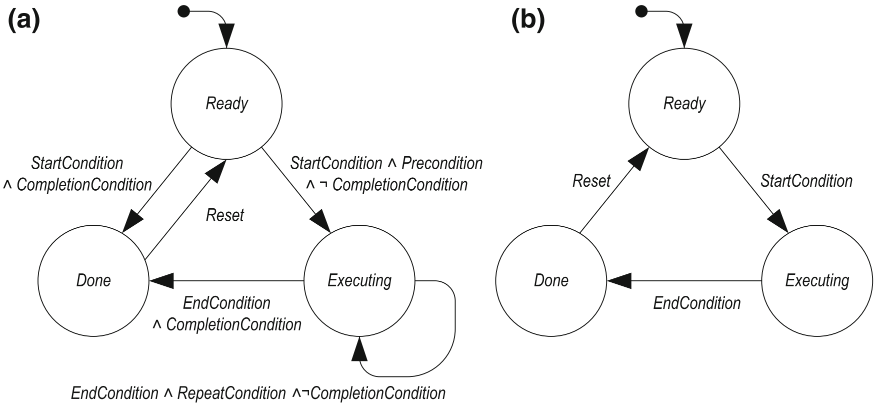


Fig. 13.8 **a** Execution state transition diagram for a generic activity. **b** Execution state transition diagram for a generic action

sition operator, the first conjunct asserts that the last activity or action in the decomposition has executed.

The *Reset* condition is *true* when an activity's or action's parent transitions from *Done* to *Ready* or from *Executing* to *Executing* when it repeats execution. If the activity has no parent (i.e., it is at the top of the decomposition hierarchy), *Reset* is *true* if that activity is in the *Done* execution state.

The *StartCondition*, *EndCondition*, and *Reset* conditions are used with the *Precondition*, *RepeatCondition*, and *CompletionCondition* to define how an activity or action transitions between execution states. This is presented in Fig. 13.8 where states are represented as nodes (rounded rectangles) and transitions as arcs. Guards are attached to each arc.

The transitions for an activity (Fig. 13.8a) are described in more detail below:

- An activity is initially in the inactive state, *Ready*. If the *StartCondition* and *Precondition* are satisfied and the *CompletionCondition* is not, then the activity can transition to the *Executing* state. However, if the *StartCondition* and *CompletionCondition* are satisfied, the activity moves directly to *Done*.
- When in the *Executing* state, an activity will repeat execution when its *EndCondition* is satisfied as long as its *RepeatCondition* is *true* and its *CompletionCondition* is not. An activity transitions from *Executing* to *Done* when both the *EndCondition* and *CompletionCondition* are satisfied.
- An activity will remain in the *Done* state until its *Reset* condition is satisfied, where it returns to the *Ready* state.

Note that if an activity does not have a *Precondition*, the *Precondition* condition is considered to be *true*. If the activity does not have a *CompletionCondition*, the *CompletionCondition* clause is removed from all possible transitions and the *Ready* to *Done* transition is completely removed (Fig. 13.8a). If the activity does not have a *RepeatCondition*, the *Executing* to *Executing* transition is removed (Fig. 13.8a).

The transition criteria for an action is simpler (Fig. 13.8b) since an action cannot have a *Precondition*, *CompletionCondition*, or *RepeatCondition*. Because actions do not have any children, their *EndCondition*s are always *true*. Actions in a sync or com decomposition must transition through their execution states at the same time.

The behavior of human action outputs and local variable assignments are dependent on the action formal semantics. For a humanaction with autoreset behavior, the human action output occurs when a corresponding action node in the EOFM task structure is *Executing* and does not otherwise. For a humanaction with toggle behavior, the human action switches between occurring or not occurring when a corresponding action node is *Executing*. A humanaction with a setvalue behavior will set the corresponding human action's value when the action is *Executing*. Similarly, a communication action or a local variable assignment associated with an action node will occur when the action is *Executing*.

13.3.5 EOFM to SAL Translation

To be utilized in a model checking verification, models written using EOFM's XML notation must be translated into a model checking language. We use the formal semantics to translate XML into the language of the Symbolic Analysis Laboratory (SAL). SAL was selected for use with EOFM because of the expressiveness of its notation, its support for a suite of checking and auxiliary tools, and its cutting edge performance at the time of EOFM's development. SAL is a framework for combining different tools to calculate properties of concurrent systems (De Moura et al. 2003; Shankar 2000). The SAL language (see De Moura et al. 2003) is designed for specifying concurrent systems in a compositional way. Constants and types are defined globally. Discrete system components are represented as modules. Each module is defined in terms of input, output, and local variables. Modules are linked by their input and output variables. Within a module, local and output variables are given initial values. All subsequent value changes occur as a result of transitions. A transition is composed of a guard and a transition assignment. The guard is a Boolean expression composed of input, output, and local variables as well as SAL's mathematical operators. The transition assignment defines how the value of local and output variables change when the guard is satisfied. The SAL language is supported by a tool suite which includes state of the art symbolic (SAL-SMC), bounded (SAL-BMC), and "infinite" bounded (SAL-INF-BMC) model checkers. Auxiliary tools include a simulator, deadlock checker, and an automated test case generator.

The EOFM to SAL translation is automated by custom Java software that uses the Document Object Model (Le Hégaret 2002) to parse EOFM's XML code and convert it into SAL code. Currently, several different varieties of EOFM to SAL translators exist; each supports different subsets of EOFM functionality. For example, there is a different translator for EOFMC that allows for the modeling and analyses of human-human communication and coordination. Despite the difference, the translators generally function on the same principles.

For a given instantiated EOFM, the translator defines SAL constants and types using the constant and userdefinedtype nodes. The translator creates a single SAL module for the group of human operators represented in humanoperator and sharedeofm nodes. Input, local, and output variables are defined in each module corresponding to the humanoperator nodes' inputvariable, localvariable, and humanaction nodes respectively. Input and local variables are defined in SAL using the name and type (basictype or userdefinedtype) attributes from the XML. Local variables are initialized to their values from the markup. All output variables represent humanactions. They are either treated as Boolean (for actions with autoreset and toggle behavior; *true* indicates this action is being performed) or are given the type associated with the value they carry (for setvalue behavior).

The translator defines two Boolean variables in the group module to handle a coordination handshake with the human-device interface module (see Bolton and Bass 2010a):

1. An input variable *InterfaceReady* that is *true* when the interface is ready to receive input; and
2. An output variable *ActionsSubmitted* that is *true* when one or more human actions are performed.

The *ActionsSubmitted* output variable is initialized to *false*.

The translator defines a SAL type, *ActivityState*, to represent the execution states of activities and actions: *Ready*, *Executing*, or *Done* (Fig. 13.8). As described previously, the activity and action state transactions define the individual and coordinated tasks of the group of human operators (Fig. 13.8). Each activity and action in a task structure has an associated local variable of type *ActivityState*. The transitions between the execution state for each activity and action are explicitly defined as nondeterministic transitions in the module representing human task behavior. Figure 13.9 presents patterns that show how the transitions from Fig. 13.8a for each activity are implemented in SAL by the translator. Note that for a given activity, the *StartCondition* and *EndCondition* are defined in accordance with the formal semantics (see Sect. 13.3.4). It is also important to note that, in these transitions, if an activity does not have a particular strategic knowledge condition, the clause in the transition guard is eliminated. Further, if an activity does not have a repeat condition, the *Executing* to *Executing* condition is eliminated completely. If an activity does not have a *CompletionCondition*, the *Ready* to *Done* transition is eliminated. Because the *Reset* occurs when an activity's parent resets, the *Reset* transition is handled differently than the others. When a parent activity transitions from *Executing* to *Executing*, its children's execution state variables (and all activities and actions lower in the hierarchy) are assigned the *Ready* state (see the *Executing* to *Executing* transition in Fig. 13.9a). Additionally, for each activity at the top of a task hierarchy, a transition (Fig. 13.9b) is created that checks if its execution state variable is *Done*. Then, in the transition assignment, this variable is assigned a value of *Ready* along with the lower level activities and actions in order to achieve the desired *Reset* behavior.

(a)

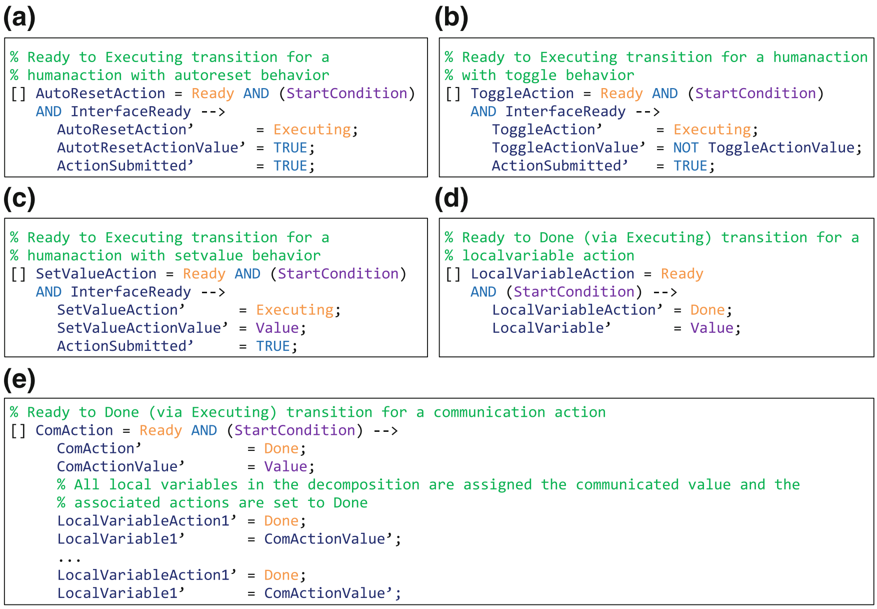
```
% Ready to Executing transition for an activity
[] Activity = Ready AND (StartCondition) AND (Precondition) AND NOT (CompletionCondition) -->
     Activity’ = Executing;
% Ready to Done transition for an activity
[] Activity = Ready AND (StartCondition) AND (CompletionCondition) -->
     Activity’ = Done;
% Executing to Executing (a repeat) transition for an activity
[] Activity = Executing AND (EndCondition) AND (RepeatCondition) AND NOT (CompletionCondition) -->
     Activity’ = Executing;
     % All sub-activities and actions are set to Ready (they are Reset)
     SubAct’   = Ready;
     ...
% Executing to Done transition for an activity
[] Activity = Executing AND (EndCondition) AND (CompletionCondition) -->
     Activity’ = Done;
```

(b)

```
% Done to Ready transition (Reset) for an activity with no parent
[] Activity = Done -->
     Activity’ = Ready;
     % All sub-activities and actions are set to Ready (the are Reset)
     SubAct’   = Ready;
     ...
```

Fig. 13.9 Patterns of SAL transitions used for an activity to transition between its execution state. **a** Represents all but the Reset transition. **b** Represents the Reset transition. Note that this Reset transition only occurs for activities at the top of a task model hierarchy (ones with no parent). Other resets occur based on the assignments that occur in these types of transitions or in repeat transitions (see a). In SAL notation (see De Moura et al. 2003), `[]` indicates the beginning of a nondeterministic transition. This is followed by a Boolean expression representing the transition’s guard. The guard ends with a `->`. This is followed underneath by a series of variable assignments where a `′` on the end of a variable indicates that the variable is being used in the next state. Color is used to improve code readability. Comments are *green*, variables are *dark blue*, reserved SAL words are *light blue*, and values are *orange*. *Purple* is used to represent expressions, such activity conditions, or values that would be explicitly inserted into the code by the translator

Transitions between execution states for variables associated with action nodes that represent humanactions are handled differently due to the variable action types and behaviors, simpler formal semantics, and the coordination handshake. Because resets are handled by activity transitions, explicit transitions are only required for activity *Ready* to *Executing* and *Executing* to *Done* transitions. Figure 13.10 shows how the *Ready* to *Executing* transitions are represented for each of the different action types. Note that only actions that are humanactions must wait for *InterfaceReady* to be *true* to execute. This is because these are the only actions that produce behavior that needs to be observed by other elements in the formal model (Fig. 13.10a–c). In the variable assignment for each of the actions, *ActionsSubmitted* is set to true. Similarly, to improve model scalability, other types of human actions (local variable assignments and communication actions; Fig. 13.10d, e) do not make use of the coordination protocol. Their *Executing* to *Done* transitions also occur automatically following the *Ready* to *Executing* because an action’s *EndCondition* is always *true*. Thus, the transitions shown in Fig. 13.10 effectively show a *Ready* to *Done* transition with all of the work associated with the action *Executing* occurring. Actions that fall in a sync decomposition are handled in accordance with the transitions in Fig. 13.10. Specifically, the *StartCondition* of the first action in the decomposition



(a)

```
% Ready to Executing transition for a
% humanaction with autoreset behavior
[] AutoResetAction = Ready AND (StartCondition)
   AND InterfaceReady -->
      AutoResetAction'       = Executing;
      AutotResetActionValue' = TRUE;
      ActionSubmitted'       = TRUE;
```

(b)

```
% Ready to Executing transition for a humanaction
% with toggle behavior
[] ToggleAction = Ready AND (StartCondition)
   AND InterfaceReady -->
      ToggleAction'      = Executing;
      ToggleActionValue' = NOT ToggleActionValue;
      ActionSubmitted'   = TRUE;
```

(c)

```
% Ready to Executing transition for a
% humanaction with setvalue behavior
[] SetValueAction = Ready AND (StartCondition)
   AND InterfaceReady -->
      SetValueAction'      = Executing;
      SetValueActionValue' = Value;
      ActionSubmitted'     = TRUE;
```

(d)

```
% Ready to Done (via Executing) transition for a
% localvariable action
[] LocalVariableAction = Ready
   AND (StartCondition) -->
      LocalVariableAction' = Done;
      LocalVariable'       = Value;
```

(e)

```
% Ready to Done (via Executing) transition for a communication action
[] ComAction = Ready AND (StartCondition) -->
      ComAction'            = Done;
      ComActionValue'       = Value;
      % All local variables in the decomposition are assigned the communicated value and the
      % associated actions are set to Done
      LocalVariableAction1' = Done;
      LocalVariable1'       = ComActionValue';
      ...
      LocalVariableAction1' = Done;
      LocalVariable1'       = ComActionValue';
```

Fig. 13.10 Patterns of SAL transitions used for an action to transition from *Ready* to *Executing* and/or *Ready* to *Done* with an implied execution in between. **a** The pattern used for a humanaction with autoreset behavior. **b** The pattern used for a humanaction with toggle behavior. **c** The pattern used for a humanaction with setvalue behavior. **d** The pattern used for a localvariable action. **e** The pattern used for a communication action. Note that `LocalVariable1`–`LocalVariableN` represent local variables in a com decomposition that received the value communicated. Further note that in **c**, **d**, and **e**, `Value` is used to represent a variable or specific value from the XML markup

must be satisfied in the transition guard and the variable assignment next state values for all actions in the decomposition are done in accordance with their type.

Because the *EndCondition* for all actions is always true, the *Executing* to *Done* transition for humanactions is handled by a single guard and transition assignment (Fig. 13.11). In this, the guard accounts for the handshake protocol. Thus the guard specifies that *ActionsSubmitted* is *true* and that *InterfaceReady* is *false*: verifying that the interface has received submitted humanaction outputs. In the assignment, *ActionsSubmitted* is set to *true*; any execution state variable associated with an *Executing* humanaction is set to *Done* (it is unchanged otherwise); and any humanaction output variables that supports the autoreset behavior are set to *false*.

Because all of the transitions are non-deterministic, multiple activities can be executed independently of each other when _par decomposition operators are used, when they are in non-shared task structures of different human operators, and when they are in *sharedeofms*. Multiple human actions resulting from such relationships are treated as if they occur at the same time if the associated humanaction output variables change during the same interval (a sequential set of model transitions) when *InterfaceReady* is *true*. However, human actions need not wait for all

(a)

```
% Ready to Executing transition for a
% humanaction with autoreset behavior
[] AutoResetAction = Ready AND (StartCondition)
   AND InterfaceReady -->
      AutoResetAction'        = Executing;
      AutotResetActionValue'  = TRUE;
      ActionSubmitted'        = TRUE;
```

(b)

```
% Ready to Executing transition for a humanaction
% with toggle behavior
[] ToggleAction = Ready AND (StartCondition)
   AND InterfaceReady -->
      ToggleAction'       = Executing;
      ToggleActionValue'  = NOT ToggleActionValue;
      ActionSubmitted'    = TRUE;
```

(c)

```
% Ready to Executing transition for a
% humanaction with setvalue behavior
[] SetValueAction = Ready AND (StartCondition)
   AND InterfaceReady -->
      SetValueAction'      = Executing;
      SetValueActionValue' = Value;
      ActionSubmitted'     = TRUE;
```

(d)

```
% Ready to Done (via Executing) transition for a
% localvariable action
[] LocalVariableAction = Ready
   AND (StartCondition) -->
      LocalVariableAction' = Done;
      LocalVariable'       = Value;
```

(e)

```
% Ready to Done (via Executing) transition for a communication action
[] ComAction = Ready AND (StartCondition) -->
      ComAction'             = Done;
      ComActionValue'        = Value;
      % All local variables in the decomposition are assigned the communicated value and the
      % associated actions are set to Done
      LocalVariableAction1' = Done;
      LocalVariable1'       = ComActionValue';
      ...
      LocalVariableAction1' = Done;
      LocalVariable1'       = ComActionValue';
```

Fig. 13.11 SAL transition pattern used to handle action *Executing* to *Done* transitions

possible actions to occur once *InterfaceReady* has become *true*. In this way, all possible overlapping and interleaving of parallel actions can be considered.

For our communication protocol case study, the EOFMC XML (Fig. 13.7) was translated into SAL using the automated translator.[5] The original model contained 164 lines of XML code. The translated model contained 490 lines of SAL code. A full listing of the model code can be found at http://tinyurl.com/EOFMCBook.[6] This model was asynchronously composed with a simple model representing the heading change window, where the heading can be changed when the pilot rotates the heading knob. These two models were composed together to create the full system model used in the verification analyses.

13.3.6 Erroneous Behavior Generation

EOFM supports three different types of erroneous behavior generation, each based on different theory and supported by different translators. For each of these generation techniques, an analyst can specify a maximum number of erroneous behaviors

[5] This translation also included miscommunication generation. This is discussed subsequently in Sect. 13.3.6.3.

[6] Note that variables presented in the text are slightly different than in the model to improve readability.

to consider. The associated translator then creates a formal model that will generate a formal representation of the EOFM represented behavior with the ability to perform erroneous behaviors based on the theory being used. When paired with a larger formal model and evaluated with model checking, the model checker will verify that the specification properties are either true or not with up to the maximum number of erroneous behaviors occurring, in all of the possible ways the erroneous behavior can occur up to the maximum. Thus, this feature allows consideration of the impact that potentially unanticipated erroneous behavior can have on system safety.

13.3.6.1 Phenomenological Erroneous Behavior Generation

The first erroneous behavior generation technique is only supported by non-EOFMC versions of EOFM. In this generation technique (introduced in Bolton and Bass 2010b and further developed in Bolton et al. 2012), each action in an instantiated EOFM task is replaced with a generative task structure that allows for the performance of Hollnagel's (1993) zero-order phenotypes of erroneous action (Fig. 13.12). Specifically, when it is time to execute an action, the new model allows for the

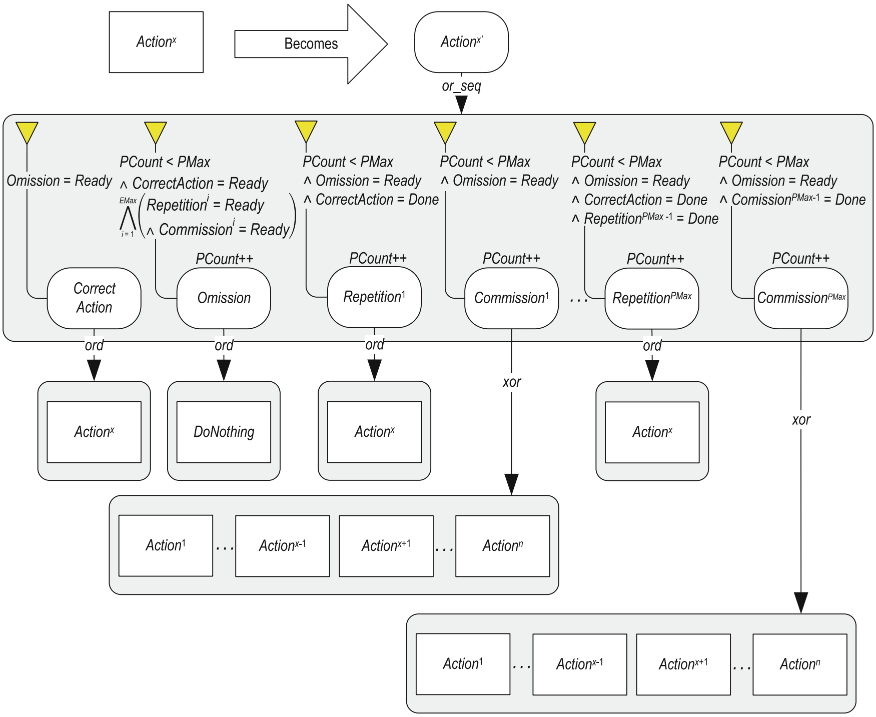


Fig. 13.12 EOFM task pattern that replaces actions at the bottom of EOFM task hierarchies to generate zero-order phenotypes of erroneous action

performance of the original action, the omission of that action, the repetition of the original action, and the commission of another action. Multiple erroneous repetitions and commissions can occur after either the performance of the original action and/or other repetitions or commissions. It is through multiple performances of erroneous actions that more complicated erroneous behaviors (more complex phenotypes) are possible.

Whenever an erroneous activity executes, a counter (*PCount*) increments. Preconditions on each activity keep the number of erroneous acts from exceeding the analyst specified maximum (*PMax*). The preconditions also ensure that there is only one way to produce a given sequence of actions for each instance of the structure. For example, an omission can only execute if no other activity in the structure has executed (all other activities must be *Ready*) and every other activity can only execute if there has not been an omission. See Fig. 13.12 and Bolton et al. (2012) for more details.

Note that analysts wishing to use this or the following erroneous behavior generation features will likely want to do so iteratively. That is, start with a maximum number of erroneous behaviors of 0 and incrementally increase it while checking desired properties at each iteration. This approach helps analysts keep model complexity under control while allowing them to evaluate the robustness of a system until they find a failure or are satisfied with the performance of their model.

13.3.6.2 Attentional Failure Generation

The second erroneous behavior generation technique (Bolton and Bass 2011, 2013) is supported by translators for both the original EOFM and EOFMC. Rather than generate erroneous behavior from the action level up, this second approach attempts to replicate the physical manifestation of Reason's slips (1990). Specifically, humans may have failures of attention that can result in them erroneously omitting, repeating, or committing an activity. We can replicate this behavior by changing the way the translator interprets the EOFM formal semantics. Specifically, all of the original transitions from the original formal semantics are retained (Fig. 13.8). However, additional, erroneous transitions are also included (Fig. 13.13) to replicate a human operator not properly attending to the environmental conditions described in EOFM strategic knowledge. A human operator can fail to properly attend to when an activity should be completed and perform an omission (an erroneous *Ready* to *Done* or *Executing* to *Done* transition), can fail to properly attend to an activity's *Precondition* and perform an erroneous *Ready* to *Executing* transition (a commission), or not properly attend to when an activity can repeat and perform a repetition (an erroneous *Executing* to *Executing* transition). Whenever an erroneous transition occurs, a counter (*ACount*) is incremented. An erroneous transition is only allowed to occur if the counter has not reached a maximum ($ACount < AMax$). More information on this erroneous behavior generation technique can be found in Bolton and Bass (2013).

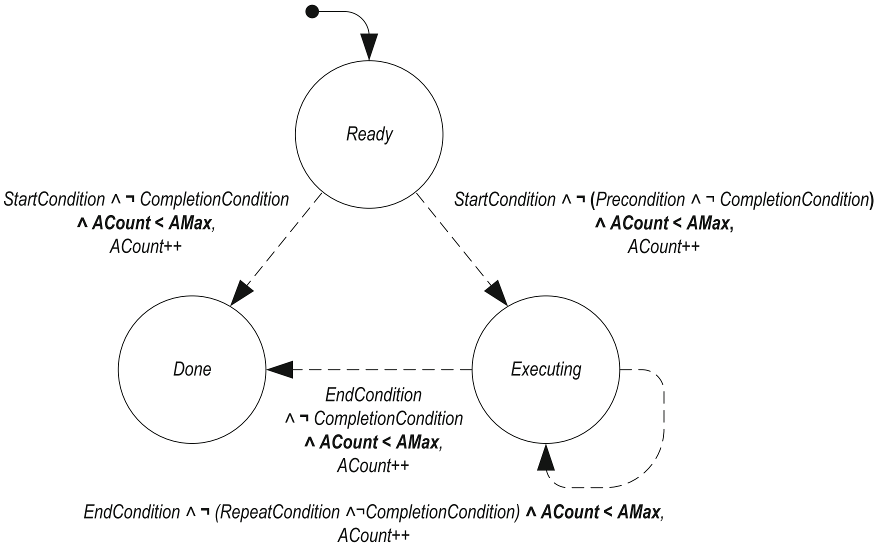

Ready
StartCondition ∧ ¬ CompletionCondition
∧ ACount < AMax,
ACount++
StartCondition ∧ ¬ (Precondition ∧ ¬ CompletionCondition)
∧ ACount < AMax,
ACount++
Done
Executing
EndCondition
∧ ¬ CompletionCondition
∧ ACount < AMax,
ACount++
EndCondition ∧ ¬ (RepeatCondition ∧¬CompletionCondition) ∧ ACount < AMax,
ACount++

Fig. 13.13 Additional transitions, beyond those shown in Fig. 13.8a, used to generate erroneous behaviors caused by a human failing to attend to information in EOFM strategic knowledge conditions

13.3.6.3 Miscommunication Generation

The final erroneous behavior generation is only supported by EOFMC translation and relates to the generation of miscommunications in communication events (Bolton 2015). This generation approach works similarly to attentional failure generation in that it adds additional formal semantics representing erroneous behaviors. Thus, miscommunication generation keeps all of the formal semantics described in Sect. 13.3.4. However, for each transition representing a human-human communication action, an additional transition is created (Fig. 13.14). In this, the value communicated can either assume the correct value meant for the original, non-erroneous, transition or any other possible communicable value as determined by the type of the information being communicated. Similarly, the local variables that are assigned the communicated value received by the other human operators can be assigned the value of what was actually communicated or some other possible communicable value. In this way, our method is able to model miscommunications as an incorrect message being sent, an incorrect message being received, or both. As with the other erroneous behavior generation methods, this approach is regulated by a counter (*CCounter*) and an analyst specified maximum (*CMax*). Every time a miscommunication occurs, a counter is incremented; and an erroneous transition can only ever occur if the counter is less than the maximum. More information on this process can be found in Bolton (2015).

```
% Ready to Done (via Executing) transition for a communication action with
% miscommunication generation
[] ComAction = Ready AND (StartCondition) AND CCount < CMax -->
      ComAction’              =  Done;
      ComActionValue’        IN {x: ComType | TRUE};
      LocalVariableAction1’  =  Done;
      LocalVariable1’        IN {x: ComType | TRUE};
      ...
      LocalVariableAction1’  =  Done;
      LocalVariable1’        IN {x: ComType | TRUE};
      LocalVariableAction1’  =  Done;
      CCount’                =  IF     ComActionValue’ /= Value
                                   OR LocalVariable1’ /= Value
                                   OR ...
                                   OR LocalVariableN’ /= Value
                                 THEN Count + 1
                                 ELSE Count ENDIF;
```

Fig. 13.14 SAL transition pattern used for generating miscommunications. In this, `ComType` is meant to represent the data type of the particular communication being performed. The statement `IN x: ComType | TRUE` is thus saying that the variable to the left of the expression can be assigned any value from the type `ComType` nondeterministically. Note that `Value` here represents the value that should have been communicated. Since `Value` is in `ComType`, the nondeterministic assignments can produce a correct communication. For this reason, the `IF...THEN...ELSE...ENDIF` expression checks to ensure that a miscommunication actually occurred. If it did, the `CCount` is incremented. Otherwise, it remains the same

In our communication protocol application, miscommunication generation was used when the original EOFMC XML code (Fig. 13.7) was converted into SAL's input language with our translator. Doing this, three versions of the model were created, starting with *CMax* values of 0, 1, and 2.

13.3.7 Specification and Verification

EOFM-supported verification analyses are designed to be conducted with SAL's symbolic model checker. Because of this, specification properties are formulated in linear temporal logic.

In the air traffic application, the purpose of the communication protocol is to ensure that the pilots set the aircraft to the heading intended by the air traffic controller. Thus, we formulate this in linear temporal logic as follows:

$$\mathsf{G}\left(\begin{array}{l}(aChangeHeading = Done)\\ \rightarrow (iHeadingWindowHeading = lATCSelectedClearance)\end{array}\right) \tag{13.1}$$

This specification was checked against the formal model with varying levels of *CMax* using SAL's symbolic model checker (sal-smc) on a windows laptop with an Intel Core i7-3667U running SAL on cygwin.

Verification analysis results are show in Table 13.2.[7]

[7] Note that an additional verification was conducted using the specification $\mathsf{F}(aChangeHeading = Done)$ for every version of the model to ensure that (13.1) was not true due to vacuity.

Table 13.2 Communication protocol formal verification results

Model with *CMax*	Verification data		
	Outcome	Time (s)	Visited states
0	✓	3.588	6498
1	✓	4.461	77177
2	×	4.602	415537

Note: A ✓ indicates verification of (13.1) was successful. A × indicates verification failed and produced a counterexample. Because verification times occur via symbolic model checking, we do not expect the verification times to vary linearly with respect the number of visited states

These results show that for maximums of zero and one miscommunications, the presented protocol will always be capable of allowing air traffic control to communicate headings to the pilot successfully and have the pilots execute that heading as long as the protocol is adhered to. However, a failure occurs when there are up to two miscommunications ($CMax = 2$).

13.3.8 Counterexample Visualization

Any produced counterexamples can be visualized and evaluated using EOFM's visual notation (Bolton and Bass 2010c). In a visualized counterexample, each counterexample step is drawn on a separate page of a document. Any task with *Executing* activities or actions is drawn next to a listing of other model variables and their values at the given step. Any drawn task hierarchy will have the color of its activities and actions coded to represent their execution state at that step. Other listed model variables can be categorized based on the model concept they represent: human mission, human-automation interface, automation, environment, and other. Any changes in an activity or action execution state or variable values from previous steps are highlighted. More information on the visualizer can be found in Bolton and Bass (2010c).

When counterexample visualization was applied to the counterexample produced for the communication protocol application, it revealed the following failure sequence:

1. When the ATCo first communicates the heading to the pilots (under aComHeading), a miscommunication occurs and the pilots both hear different incorrect headings.
2. Then when the PM reads the heading back the ATCo and the PF (under aReadbackHeading), a second miscommunication occurs and both the ATCo and the PF think they heard the original intended heading and the heading originally heard from the ATCo respectively.
3. As a result of this, the procedure proceeds without any human noticing an incorrect heading, and the incorrect heading is engaged.

A full listing of the counterexample visualization can be found at http://tinyurl.com/EOFMCBook.

13.4 Discussion

The presented application demonstrates the power of EOFM to find complex problems in systems that rely on human behavior for their safe execution, both with normative and generated erroneous human behavior. From the perspective of the application domain, the presented results are encouraging: as long as the protocol in Fig. 13.6 is adhered to, heading clearances will be successfully communicated and engaged. Ultimately it must be up to designers to determine how robust a protocol must be to miscommunication to support system safety. If further reliability is required, analysts might want to use additional pilot and ATCo read backs. Additional analyses that explore how further read backs can increase reliability can be found in Bolton (2015). Further, analysts may wish to conduct additional analyses to see how robust to other types of erroneous human behaviors the protocol is both with and without miscommunication. An example of how this can be done can be found in Pan and Bolton (2015).

Beyond this case study, EOFM has demonstrated its use in the analysis of a number of applications in a variety of domains. It is also being continually developed to improve its analyses and expand the ways it can be used in the design and evaluation of human-machine systems. These are discussed below.

13.4.1 Applications

EOFM and EOFMC have been used to evaluate an expanding set of applications. These are described below. While these have predominantly been undertaken by the authors and their collaborators, the EOFM toolset is freely available (see http://fhsl.eng.buffalo.edu/EOFM/). We encourage others to make use of these tools as EOFM development continues.

13.4.1.1 Aviation Checklist Design

Pilot noncompliance with checklists has been associated with aviation accidents. For example on May 31, 2014, procedural noncompliance of the flight crew of a Gulfstream Aerospace Corporation G-IV contributed to the death of the two pilots, a flight attendant, and four passengers (NTSB 2015). This noncompliance can be influenced by complex interactions among the checklist, pilot behavior, aircraft automation, device interfaces, and policy, all within the dynamic flight environment. We used EOFM to model a checklist procedure and employed model checking to evaluate checklist-guided pilot behavior while considering such interactions (Bolton and Bass 2012). Although pilots generally follow checklist items in order, they can complete them out of sequence. To model both of these conditions, two task models were created: one where the pilot will always perform the task in order (enforced by an ord decomposition) and one where he or she can perform them in any order (using the and_par decomposition operator).

Spoilers are retractable plates on the wings that, when deployed, slow the aircraft and decrease lift. If spoilers are not used, the aircraft can overrun the runway (NTSB 2001). A pilot can arm the spoilers for automatic deployment using a lever. Alternatively, a pilot can manually deploy the spoilers after touchdown. Arming the spoilers before the landing gear has been lowered or the landing gear doors have fully opened can result in automatic premature deployment which can cause the aircraft to lose lift and have a hard landing. Spoiler deployment is part of the *Before Landing* checklist. In our analyses, for the system model using the human task behavior (that is, the *Before Landing* checklist) with the and_par decomposition operator, we identified that a pilot could arm the spoilers early and then open the landing gear doors, a situation that could cause premature spoiler deployment. We explored how different design interventions could impact the safe arming and deployment of spoilers.

13.4.1.2 Medical Device Design

EOFM has been used to evaluate the human-automation interaction (HAI) of two different safety-critical medical devices. The first was a radiation therapy machine (Bogdanich 2010; Leveson and Turner 1993) based on the Therac-25. This device was a room-sized, computer-controlled, medical linear accelerator. It had two treatment modes: electron beam mode is used for shallow tissue treatment, and x-ray mode is used for deeper treatments—requiring electron beam current approximately 100 times greater than that used for the other mode. The x-ray mode used a beam spreader (not used in electron beam mode) to produce a uniform treatment area and attenuate the radiation of the beam. An x-ray beam treatment application without the spreader in place could deliver a lethal dose of radiation. We used EOFM with its phenomenological erroneous behavior generation to evaluate how normative and/or unanticipated erroneous behavior could result in the administration of an unshielded x-ray treatment (Bolton et al. 2012). We discovered that this could occur if a human operated accidentally selected x-ray mode (a generated erroneous act) and corrected it and administered treatment too quickly.

In the second medical device application, we used verifications with both normative behavior (Bolton and Bass 2010a) and attentional failure generation (Bolton and Bass 2013) to evaluate the safety of a patient controlled analgesia (PCA) pump. A PCA pump is a medical device that allows a patient to exert some control over intravenously delivered pain medication, where medication is delivered in accordance with a prescription programmed into the device by a medical practitioner. Using EOFM with its attentional failure generation, we were able to both discover when an omission caused by an attentional slip could result in an incorrect prescription being administered and how a simple modification to the device's interface could prevent this failure from occurring.

13.4.1.3 Medical Device User Manual Design

The EOFM language, the ability to create new models in EOFM, and the ease of composing formal task analytic models in SAL into larger system models helped to provide insights relevant to patient user manual evaluation. User manual designers generally use written procedures, figures and illustrations to convey procedural and device configuration information. However ensuring that the instructions are accurate and unambiguous is difficult. To analyze a user manual, our approach integrated EOFM's formal task analytic models and device models with safety specifications via a computational framework similar to those used for checklists and procedures. We demonstrated the value of this approach using alarm troubleshooting instructions from the patient user manual of a left ventricular assist device (LVAD) (Abbate et al. 2016).

During the process of encoding the written instructions into the formal EOFM task model, we discovered problems with task descriptions in the manual as statements were open to interpretation. During the process of developing the XML description of the procedure, for example, it became clear that the description in the user manual did not define which end of a battery cable to disconnect. We also identified issues with the order of troubleshooting steps in that the procedure included steps that did not fix the problem before ones that did. The ability to change a single decomposition operator in the model (from ord to to _seq) and subsequent model translation allowed model checking analyses visualized with our counter example visualizer to show that a better ordering was possible. In addition the ability to include a formal device model with the formal task model highlighted that the instructions did not consider all possible initial device conditions.

13.4.1.4 Human-Human Communication and Coordination Protocols

EOFMC has been used to evaluate the robustness of protocols humans use to communicate information and coordinate their collaborative efforts. In one set of such analyses, we evaluated the protocols air traffic controllers use to communicate clearances to pilots. These analyses modeled the protocols in EOFMC and were formally verified to determine if incorrect clearances could be engaged. Analyses without any miscommunication generation (Bass et al. 2011) did not discover any problems. Miscommunication generation was used to identify that a maximum of one miscommunication could cause the protocol to fail and show how this protocol could be modified to make it robust to higher maximum numbers of miscommunications (Bolton 2015). A variant of this protocol is used for the case study analyses presented in the next section.

A variant of this analysis was used to evaluate the protocol that a team of engineers use to diagnose alarms in a nuclear power plant (Pan and Bolton 2015, 2016). Rather than simply look for binary failure conditions in the performance of the team, this work compared different protocols based on guaranteed performance levels determined by the degree of correspondence between team member conclusions and system knowledge at the end of the protocol. This was used to evaluate two different versions of the protocol: one where confirmation or contradiction statements were used to show agreement or disagreement respectively and one where read backs were used. In both cases, miscommunication and attentional failure generation were included in formal verifications. These analyses revealed that the read back procedure outperformed the other, producing correct operator conclusions for any number of miscommunications and up to one attentional slip.

13.4.2 EOFM Extensions

Three EOFM extensions beyond its standard methods (Fig. 13.2) are described next.

13.4.2.1 Scalability Improvements

As the size of the EOFM used in formal verification analyses increases, the associated formal model size and verification times increase exponentially (Bolton and Bass 2010a; Bolton et al. 2012). This can limit what system EOFM can be used to evaluate. Because of this, efforts have attempted to improve its scalability. To accomplish this, the EOFM formal semantics (see Sect. 13.3.4) are interpreted slightly differently. In this new interpretation, the formal representation of the execution state are "flattened" so that they are defined as Boolean expressions purely in terms of the execution state of actions at the bottom of the EOFM hierarchy. This allows the transitions associated with activities to be represented in the transition logic of the actions. As such, there are fewer intermediary transitions in the formal EOFM representation. This has resulted in significant reductions in model size and verification time without the loss of expressive power for both artificial benchmarks and realistic applications pulled from the current EOFM literature (Bolton et al. 2016). For example, a PCA pump model that contained seven different tasks and the associated interface and mission components of the PCA pump in the larger formal model originally had 4,072,083 states and took 90.4 s to verify a true property. With the new translator, the model statespace size was reduced to 15,388 states and took only 5.6 s to verify the same property (Bolton et al. 2016).

13.4.2.2 Specification Property Generation

While EOFM can be used to evaluate HAI with model checking, most work requires analysts to manually formulate the properties to check, a process that may be error-prone. Further, analysts may not know what properties to check and thus fail to specify properties that could find unanticipated HAI issues. As such, unexpected dangerous interaction may not be discovered even when formal verification is used.

To address this, an extension of EOFM's method (Fig. 13.2) includes automatically generating specification properties from task models (Bolton 2011, 2013; Bolton et al. 2014). In general, these approaches work by using modified EOFM to SAL translators to automatically generate specification properties from the EOFM task models.

Two types of specification generation have been developed. The first types of generated specifications are task-based usability properties (Bolton 2011, 2013). These allow analysts to automatically check that a human-automation interface will support the fulfillment of the goals from EOFM tasks. While initial versions of this process only allowed the analyst to find problems (Bolton 2011), later extensions provided a diagnostic algorithm that enabled the reason for the usability failures to be systematically identified (Bolton 2013). This method was ultimately used to evaluate the usability of the PCA pump application discussed previously.

The second class of generated properties asserted qualities about the execution state of the task models based on their formal semantics (Bolton et al. 2014). This enabled analysts to look for problems in the HAI of a system by finding places where the task models would not perform as expected and thus elucidate potential unanticipated interaction problems. For example, properties would assert that every execution state of each activity and action were reachable, that every transition between execution states was achievable, that all activities and actions would eventually finish, and that any human task would always eventually be performable. This method was used to re-evaluate the previously discussed aircraft before landing checklist procedure, where it discovered an unanticipated issue (Bolton et al. 2014). It also was used in the evaluation of an unmanned aerial vehicle control system (van Paassen et al. 2014). These last two applications are particularly illustrative of the specification property generation feature's power because both discovered system problems previously unanticipated by the analysts.

13.4.2.3 Interface Generation

User-centered design is an approach for creating human-machine interfaces so that they support human operator tasks. While useful, user-centered design can be challenging because designers can fail to account for human-machine interactions that occur due to the inherent concurrence between the human and the other elements of the system. This extension of EOFM has attempted to better support user-centered designed by automatically generating formal designs of human-machine interface functional behavior from EOFM task models guaranteed to always support the task. To accomplish this, the method uses a variant of the L* algorithm (Angluin 1987) to learn a minimal finite state automation description of an interface's behavior; it uses a model checker and the formal representation of EOFM task behavior to answer questions about the interface's behavior (Li et al. 2015). This method has been used to successfully generate a number of interfaces including light switches, a vending machine, and a pod-based coffee machine (Li et al. 2015). All generated interfaces have been verified to support the various task-based properties discussed in the previous section and Bolton et al. (2014). Future work is investigating how to include usability properties in the generation process.

13.4.2.4 Improve Tool Interoperability and Usability

The use of XML allows EOFM-supported analyses to be platform independent. Despite this, to date, EOFM has only been used with the model checkers found in SAL. Future work should investigate how to make EOFM compatible with other analysis environments.

Additionally, using the XML to create task models is relatively easy for someone who is familiar with the language and had access to sophisticated XML development environments that support syntax checking and code completion. However, the usability of EOFM could be significantly improved with the addition of visual modeling tools that would allow model creation with a simple point and click interface. This will be investigated in future work.

13.5 Conclusions

As the above narrative demonstrates, EOFM offers analysts a generic, flexible task modeling system that supports a number of different formal analyses. This is evidenced by the different applications that have been evaluated using EOFM and the different analyses both supported by and extending the method (Fig. 13.2). It is important to note that although all of the analyses discussed here use the model checking tools in SAL, this need not be the case. Given that EOFM is XML-based, it should be adaptable to many other formal verification analysis environments. This is further supported by the fact that SAL's input notation is very similar to those offered by other environments. Future work will investigate how EOFM could be adapted to other model checkers to increase the scope of its analyses.

References

Abbate AJ, Throckmorton AL, Bass EJ (2016) A formal task analytic approach to medical device alarm troubleshooting instructions. IEEE Trans Hum-Mach Syst 46(1):53–65

Aït-Ameur Y, Baron M (2006) Formal and experimental validation approaches in HCI systems design based on a shared event B model. Int J Softw Tools Technol Transf 8(6):547–563

Angluin D (1987) Learning regular sets from queries and counterexamples. Inf Comput 75(2):87–106

Basnyat S, Palanque P, Schupp B, Wright P (2007) Formal socio-technical barrier modelling for safety-critical interactive systems design. Saf Sci 45(5):545–565

Bass EJ, Bolton ML, Feigh K, Griffith D, Gunter E, Mansky W, Rushby J (2011) Toward a multi-method approach to formalizing human-automation interaction and human-human communications. In: Proceedings of the IEEE international conference on systems, man, and cybernetics. IEEE, Piscataway, pp 1817–1824

Bogdanich W (2010) The radiation boom: radiation offers new cures, and ways to do harm. New York Times 23:23–27

Bolton ML, Bass EJ (2011) Using task analytic behavior models, strategic knowledge-based erroneous human behavior generation, and model checking to evaluate human-automation interaction. In: Proceedings of the IEEE international conference on systems man and cybernetics. IEEE, Piscataway, pp 1788–1794

Bolton ML (2011) Validating human-device interfaces with model checking and temporal logic properties automatically generated from task analytic models. In: Proceedings of the 20th behavior representation in modeling and simulation conference. The BRIMS Society, Sundance, pp 130–137

Bolton ML (2013) Automatic validation and failure diagnosis of human-device interfaces using task analytic models and model checking. Comput Math Organ Theory 19(3):288–312

Bolton ML (2015) Model checking human-human communication protocols using task models and miscommunication generation. J Aerosp Inf Syst 12:476–489

Bolton ML, Bass EJ (2010a) Formally verifying human-automation interaction as part of a system model: limitations and tradeoffs. Innov Syst Softw Eng: A NASA J 6(3):219–231

Bolton ML, Bass EJ (2010b) Using task analytic models and phenotypes of erroneous human behavior to discover system failures using model checking. In: Proceedings of the human factors and ergonomics society annual meeting. HFES, Santa Monica, pp 992–996

Bolton ML, Bass EJ (2010c) Using task analytic models to visualize model checker counterexamples. In: Proceedings of the 2010 IEEE international conference on systems, man, and cybernetics. IEEE, Piscataway, pp 2069–2074

Bolton ML, Bass EJ (2012) Using model checking to explore checklist-guided pilot behavior. Int J Aviat Psychol 22:343–366

Bolton ML, Bass EJ (2013) Generating erroneous human behavior from strategic knowledge in task models and evaluating its impact on system safety with model checking. IEEE Trans Syst Man Cybern: Syst 43(6):1314–1327

Bolton ML, Siminiceanu RI, Bass EJ (2011) A systematic approach to model checking human-automation interaction using task-analytic models. IEEE Trans Syst Man Cybern Part A 41(5):961–976

Bolton ML, Bass EJ, Siminiceanu RI (2012) Using phenotypical erroneous human behavior generation to evaluate human-automation interaction using model checking. Int J Hum-Comput Stud 70:888–906

Bolton ML, Bass EJ, Siminiceanu RI (2013) Using formal verification to evaluate human-automation interaction: a review. IEEE Trans Syst Man Cybern: Syst 43(3):488–503

Bolton ML, Jimenez N, van Paassen MM, Trujillo M (2014) Automatically generating specification properties from task models for the formal verification of human-automation interaction. IEEE Trans Hum-Mach Syst 44(5):561–575

Bolton ML, Zheng X, Molinaro K, Houser A, Li M (2016) Improving the scalability of formal human-automation interaction verification analyses that use task analytic models. Innov Syst Softw Eng: A NASA J (in press). doi:10.1007/s11334-016-0272-z

Clark J, Murata M (2001) Relax NG specification. Committee Specification. http://relaxng.org/spec-20011203.html

De Moura L, Owre S, Shankar N (2003) The SAL language manual. Technical Report CSL-01-01, Computer Science Laboratory, SRI International, Menlo Park. http://staffwww.dcs.shef.ac.uk/people/A.Simons/z2sal/saldocs/SALlanguage.pdf

Degani A, Heymann M, Shafto M (1999) Formal aspects of procedures: the problem of sequential correctness. Proceedings of the 43rd annual meeting of the human factors and ergonomics society. HFES, Santa Monica, pp 1113–1117

Fields RE (2001) Analysis of erroneous actions in the design of critical systems. PhD thesis, University of York, York

Giese M, Mistrzyk T, Pfau A, Szwillus G, von Detten M (2008) AMBOSS: a task modeling approach for safety-critical systems. In: Proceedings of the second international conference on human-centered software engineering. Springer, Berlin, pp 98–109

Gunter EL, Yasmeen A, Gunter CA, Nguyen A (2009) Specifying and analyzing workflows for automated identification and data capture. In: Proceedings of the 42nd Hawaii international conference on system sciences. IEEE Computer Society, Los Alamitos, pp 1–11

Hartson HR, Siochi AC, Hix D (1990) The UAN: a user-oriented representation for direct manipulation interface designs. ACM Trans Inf Syst 8(3):181–203

Hollnagel E (1993) The phenotype of erroneous actions. Int J Man-Mach Stud 39(1):1–32

Kirwan B, Ainsworth LK (1992) A guide to task analysis. Taylor and Francis, London

Le Hégaret P (2002) The w3c document object model (DOM). http://www.w3.org/2002/07/26-dom-article.html

Leveson NG, Turner CS (1993) An investigation of the therac-25 accidents. Computer 26(7):18–41

Li M, Molinaro K, Bolton ML (2015) Learning formal human-machine interface designs from task analytic models. In: Proceedings of the HFES annual meeting. HFES, Santa Monica, pp 652–656

Martinie C, Palanque P, Barboni E, Ragosta M (2011) Task-model based assessment of automation levels: application to space ground segments. In: Proceedings of the 2011 IEEE international conference on systems, man, and cybernetics. Piscataway, IEEE, pp 3267–3273

Martinie C, Navarre D, Palanque P (2014) A multi-formalism approach for model-based dynamic distribution of user interfaces of critical interactive systems. Int J Hum-Comput Stud 72(1):77–99

Mitchell CM, Miller RA (1986) A discrete control model of operator function: a methodology for information display design. IEEE Trans Syst Man Cybern Part A: Syst Hum 16(3):343–357

NTSB (2001) Runway Overrun During Landing, American Airlines Flight 1420, McDonnell Douglas MD-82, N215AA, Little Rock, Arkansas, June 1, 1999 Technical Report NTSB/AAR-01/02. National Transportation Safety Board, Washington, DC

NTSB (2015) Runway Overrun During Rejected Takeoff Gulfstream Aerospace Corporation G-IV, N121JM, Bedford, Massachusetts, May 31, 2014 Technical Report NTSB/AAR-15/03. National Transportation Safety Board, Washington, DC

Palanque PA, Bastide R, Senges V (1996) Validating interactive system design through the verification of formal task and system models. In: Proceedings of the IFIP TC2/WG2.7 working conference on engineering for human-computer interaction. Chapman and Hall Ltd., London, pp 189–212

Pan D, Bolton ML (2015) A formal method for evaluating the performance level of human-human collaborative procedures. In: Proceedings of HCI international. Springer, Berlin, pp 186–197

Pan D, Bolton ML (2016) Properties for formally assessing the performance level of human-human collaborative procedures with miscommunications and erroneous human behavior. Int J Ind Ergon (in press). doi:10.1016/j.ergon.2016.04.001

Paternò F, Santoro C (2001) Integrating model checking and HCI tools to help designers verify user interface properties. In: Proceedings of the 7th international workshop on the design, specification, and verification of interactive systems. Springer, Berlin, pp 135–150

Paternò F, Mancini C, Meniconi S (1997) Concurtasktrees: a diagrammatic notation for specifying task models. In: Proceedings of the IFIP TC13 international conference on human-computer interaction. Chapman and Hall Ltd, London, pp 362–369

Perrow C (1999) Normal accidents: living with high-risk technologies. Princeton University Press, Princeton

Reason J (1990) Human error. Cambridge University Press, New York

Shankar N (2000) Symbolic analysis of transition systems. Proceedings of the international workshop on abstract state machines, theory and applications. Springer, London, pp 287–302

Sheridan TB, Parasuraman R (2005) Human-automation interaction. Rev Hum Factors Ergon 1(1):89–129

Syncro Soft (2016) Relax NG schema diagram. In: User Manual of Oxygen XML Editor 17.1. https://www.oxygenxml.com/doc/versions/17.1/ug-editor/index.html#topics/relax-ng-schema-diagram.html

van Paassen MM, Bolton ML, Jimenez N (2014) Checking formal verification models for human-automation interaction. 2014 IEEE international conference on systems, man and cybernetics (SMC). IEEE, Piscataway, pp 3709–3714

Chapter 14
The Specification and Analysis of Use Properties of a Nuclear Control System

Michael D. Harrison, Paolo M. Masci, José Creissac Campos and Paul Curzon

Abstract This chapter explores a layered approach to the analysis of the Nuclear Power Plant Control System described in Chap. 4. A model is specified to allow the analysis of use-centred properties based on generic templates. User interface properties include the visibility of state attributes, the clarity of the mode structure and the ease with which an action can be recovered from. Property templates are used as heuristics to ease the construction of requirements for the control system interface.

14.1 Introduction

Formal modelling can offer substantial benefits when developing an interactive system. It enables systematic clarification of assumptions made about a design and supports verification that specified requirements have been met. This paper considers the nuclear power plant control system described in Chap. 4. The use cases introduced in the book offer slightly different perspectives that might suggest different approaches to analysis. Broadly, analysis approaches may be classified as task-orientated or task-based on the characteristics of the interface. In the first category, there are informal approaches, for example Cognitive Walkthrough (Polson et al. 1992), and formal approaches such as those of Bolton et al. (2012). These approaches are concerned with the representation of the intended task and then to analyse the system

M.D. Harrison (✉)
School of Computing Science, Newcastle University, Newcastle upon Tyne, UK
e-mail: michael.harrison@ncl.ac.uk

P.M. Masci · J.C. Campos
Dep. Informática/Universidade do Minho & HASLab/INESC TEC, Braga, Portugal
e-mail: paolo.masci@inesctec.pt

J.C. Campos
e-mail: jose.campos@di.uminho.pt

P. Curzon
EECS, Queen Mary University of London, Mile End Road, London, UK
e-mail: p.curzon@qmul.ac.uk

B. Weyers et al. (eds.), *The Handbook of Formal Methods in Human-Computer Interaction*, Human-Computer Interaction Series,
DOI 10.1007/978-3-319-51838-1_14

that is intended to perform the task. In the second category, analysis may be based on the characteristics of the interface (for example, Heuristic Evaluation Nielsen and Molich (1990) and formal approaches such as those of Campos and Harrison (2009). These approaches focus on, for example, the visibility or perceivability of key attributes of the device and analyse the properties of the supported actions (e.g. their predictability or undoability). The approach taken in this paper supports both styles of analysis. In the case of the task approach, the focus is on the constraints that determine the activities that the user performs, rather than focusing on prescribed normative behaviours. Constraints include the visibility of information (e.g. function key displays) that help the user to decide what action to be taken next.

A modelling approach based on the layers of specification is designed to unify these two approaches to analysis, with the aim of maintaining the integrity of the specification. Analysis of the interactive system is facilitated by the use of property templates.

This chapter is organised as follows: Sect. 14.2 describes the features of the example that are relevant to illustrating the analysis. Section 14.3 discusses the structure of the model that describes the interactive behaviour of the system. Section 14.4 describes the tools, including the set of property templates that are used to drive the analysis. Section 14.5 details the model of the example and describes the process of instantiating the property templates to be theorems over the model. Finally, we describe related work (Sect. 14.6) and conclusions (Sect. 14.7).

14.2 The Use Case

In the present example, two analytic perspectives are taken.

- How well does the interface support operating procedures[1] developed to help the controller start up or close down the system?
- Is the operator able to monitor and make appropriate adjustments to the process? Is there sufficient information for operators to understand what is happening and are suitable actions visible and available?

These two perspectives require different styles of analysis. The first is concerned with how effectively the display, and the actions it supports, can be invoked as required by an operator who is following the start-up and close-down operating procedures. The second is concerned with the display, the graphics, the status display, the sliders, the enabled actions and how these change the display and reflect the state of the underlying process. Whatever the level of analysis of the user interface, it is important to understand the interface to the underlying system. The interface of the system should aid understanding by making parts of the underlying process visible to the user; producing visible feedback to enable the operators to assess what has been done. Interactive systems of any complexity have a common characteristic that some elements of the state of the system are perceivable (e.g. visible or audible) and that user actions transform the state (Duke and Harrison 1993). Furthermore, not

[1] http://www.hci-modeling.org/nppsimulator/BWRSimulationDescription.pdf.

all actions are permitted all of the time, and the behaviour of actions can depend on distinguished state attributes called *modes*, see Gow et al. (2006) for further discussion. The modes in this case determine, for example, whether the control rods are being controlled automatically or manually. Modes also determine specific interactions related to the behaviour of the mouse: its position and whether the mouse button is pressed or not. For example, when the mouse button is pressed and the cursor position coincides with a slider on the screen, and the slider is not in automatic mode, then dragging the cursor moves the position of the slider thus changing the relevant behaviour of the component of the process that it represents.

Users have difficulty in understanding the progress of a system when the elements of the state of the system that are relevant to that understanding are not visible in a form that makes sense to them. At the same time, confusion can arise when actions relevant to the current activity are, apparently or actually, disabled by the system, or when the actions have an unexpected or inconsistent effect with respect to the users' knowledge and experiences of the system. Actions and states are therefore elemental in understanding interactive behaviour. Modes are also important. It is unusual that an interactive system is so simple that actions always have the same effect.

To achieve the goals and activities required of the users, most interactive systems are designed more or less effectively to ensure that the information required (we call them *information resources* (Campos et al. 2014)) is made explicitly available, and in a form that can be easily understood by the users. A role of a model of the interactive system is therefore to make these information resources explicit, so that assumptions about the constraints they impose may be analysed.

14.3 Structure of the Models

It is important to distinguish between the interactive systems and the components of interactive systems. Interactive systems are socio-technical systems involving people, devices and artefacts (desks, pieces of paper, pens, tablets and so on). The primary focus of the modelling approach illustrated here is on the interactive devices that are the components of the interactive system. The presented property templates capture aspects of the system that can facilitate device–user interaction.

14.3.1 The Interface Specification

The specification of an interactive system includes a definition of the set of actions, including user actions, that are possible within them. These actions affect and are affected by the state of the system. The behaviour of actions is often determined by the mode of the device. The proposed model of the interactive system also makes explicit the information resources that are assumed to aid the use of the system. Assumptions about the activities for which the system is designed are also made explicit. An action is a transformation supported directly by the interface. An activity is a means to achieve some work goal, for example achieving a steady state of the system with maximum voltage.

The interface specification describes what the display shows and captures the effects of user-level actions. The display will show some features of the state of the reactor, and these features may be encoded as part of the interface. It will also show the user actions that are translated into actions within the reactor. The specification includes display widgets, showing simple status information. These include widgets labelled RKS, RKT, KNT, TBN, $WP1$, $WP2$, CP, AU. These displays are associated with a range of colours indicating status. The display also shows actions associated with the valves: $SV1$, $SV2$, $WV1$, $WV2$ and sliders that change the position of the control rods and the status of the valves.

Analysis of an interactive device is then concerned with proving that relevant feedback is given on completing an action, that relevant information is available before an action is carried out, that it is possible to recover from an action in specified circumstances, and that it is always possible simply to step to some home mode whatever the state of the device and that actions can be completed consistently.

14.3.2 Structuring Specifications

The model of the interactive system is structured into four layers. The first layer simply specifies the constants and types used throughout the specification. It includes types related to the devices involved and the entities that are in the broader system. For example, in the case of the reactor these types would include notions such as $pressure$, $volume$ and $temperature$. There would also be types associated with pumps and valves. Constants would include maximum and minimum values required to trigger error events in certain situations.

The second layer describes assumptions about the underlying process, managed or controlled by the devices that are required to enable the analysis of the characteristics of the interactive device. This layer is often reused across families of device models when exploring the effects of differing user interfaces. For example, in Harrison et al. (2015b), different brands of IV infusion devices share the same pump layer. The process layer, in the case of the nuclear process, is the simplest model of the nuclear reactor that will allow a proper consideration of interactive behaviour of the control system. A specification of the underlying reactor, describing the details of the relation between reactor core and turbine, would include attributes defining water level and pressure for each. The specification at this level would also define the characteristics of the pumps and valves. The pumps would be associated with rates per minute, and the valves would be on or off. A number of actions will be specified at this level. An action $tick$ is used to represent the interval of one minute and update the attributes to describe the evolving process. There will be further actions switching pumps on and off, opening and closing valves and changing the value of flow in the pump, for example.

The third layer describes the interface to the interactive device or system. This model uses the process description described in the second layer. It makes those aspects of the state that are visible explicit through the interface. It describes the user actions, including how the sliders or buttons or other display widgets work. The third layer of the specification of the nuclear power plant control user interface specifies how the user sets, controls and views the operation of the reactor. It is specific to this particular interface, whereas the reactor specification (given in the second layer) may be more generic and therefore used with several user interfaces. It provides an opportunity to explore the variety of user interfaces that may be appropriate for supporting human–machine interactions necessary to control the reactor.

The fourth, and final, layer makes explicit the information resources that are required for different actions in different circumstances. It captures constraints on action based on the goals and activities that the user achieves (Campos et al. 2014). This layer contains an interactive system view. The activities and actions are "resourced" by user interfaces for devices that are used in the interactive system or, indeed, any other source of relevant information that is present within the interactive system. It adds attributes that are not captured by the devices and includes (meta-)actions that describe activities that may involve the actions of the interactive devices. An example of this fourth layer used in a different context can be found in Masci et al. (2012).

The models to be considered have some or all of the following characteristics depending on layer.

- A set of *actions* $a : A = S \longmapsto S$, where S is a set of states. Actions are partial functions. They are made total by including a value "undefined" ($\perp$). A permission function *per* takes an action and determines whether it is defined for a value in its domain $per : A \to (S \to T)$ such that $per(a)(s) = true$ if $a(s) \neq \perp$.
- A *state* is a set of attributes. Functions of the form $filter : S \to C$, where C is an attribute will often be used to extract an attribute of the state. The attribute is itself a domain, for example temperature or pressure. Similarly, some elements of the state are part of the interface and are perceivable. p_filter will often be used to describe the filter that extracts the corresponding visual attribute to the value extracted by $filter$. Alternatively, a predicate $vis\_filter : C \to T$ may be used to assert that the value of the filtered attribute is visible.
- The function *mode* is a particular form of $filter$, namely $mode : S \to MS$. It extracts the modes of the model, where MS is an attribute that ranges over a set of modes. In the example, one set of modes relates to the types of variable being entered through number entry.

14.4 Tool Support

Two approaches to specification and proof are feasible for the kinds of model described here: model checking and theorem proving. The theorem-proving approach is appropriate here because a potentially important feature of the analysis, not discussed further in this short chapter, concerns the mechanisms for number entry. Since the domain of numbers is relatively large, proof using model checking can result in analyses of very large models that can be intractable. Interested readers are redirected to Harrison et al. (2015a) for an application where the layered approach described in this paper is used in an example involving number entry.

14.4.1 Representing and Proving the Model

The analysis is performed using the *Prototype Verification System (PVS)* (Shankar et al. 1999), an automated theorem prover developed at SRI. The system combines a specification language based on higher-order logic with an interactive prover. PVS has been used extensively in several application domains. The higher-order logic supports the usual basic types such as `boolean`, `integer` and `real`. New types can be introduced either in a declarative form (these types are called *uninterpreted*) or through *type constructors*. Examples of type constructors used in the present specification are function and record types. Function types are denoted `[D -> R]`, where `D` is the domain type and `R` is the range type. Predicates are Boolean-valued functions. Record types are defined by listing the field names and their types between square brackets and hash symbols. Predicate subtyping is a language mechanism used for restricting the domain of a type by using a predicate. An example of a subtype is `{x:A | P(x)}`, which introduces a new type as the subset of those elements of type `A` that satisfy the predicate `P`. The notation `(P)` is an abbreviation of the subtype expression above. Predicate subtyping is useful for specifying partial functions. This notion is used to restrict actions to those that are permitted explicitly by the permission predicates mentioned when describing the models in general terms. Specifications in PVS are expressed as a collection of *theories*, which consist of declarations of names for types and constants, and expressions associated with those names. Theories can be parametrised with types and constants, and can use declarations of other theories by importing them. The prelude is a standard library automatically imported by PVS. It contains a large number of useful definitions and proved facts for types, including common base types such as Booleans (`boolean`) and numbers (e.g. `nat`, `integer` and `real`), functions, sets and lists.

The specification of the models takes a standard form as described in Sect. 14.3.2. A model consists of a set of actions and a set of permissions that capture when the actions can occur.

```
action: TYPE = [state -> state]
```

For each action, there is a predicate:

```
  per_action: TYPE = [state -> boolean]
```

that indicates whether the action is permitted.

14.4.2 Property Templates

Property templates are generic mathematical formulae designed to help developers to construct theorems appropriate to the analysis of user interface features. The aim is to make these programmable devices more predictable and easy to use. The particular set of templates considered here is derived from Campos and Harrison (2008). A formulation of these properties based on *actions*, *states* and *modes* is presented, along with a brief summary of the use-related concerns captured by the template. There are two types of property: properties that relate states where a specific action has taken place, and properties that relate a state to *any* state that can be reached by *any action* from that state. The relation $transit : S \times S$ relates states that can be reached by any action. For a particular model, $transit$ will be instantiated to connect states by the actions provided by the system.

Completeness. This template checks that the interactive system allows the user to reach significant states in one (or a few steps). For example, being able to reach "home" from any device screen in one step is a completeness property. The completeness template asserts that a user action will transform any state that satisfies a predicate $guard : S \rightarrow T$ into another state that satisfies a predicate $goal : S \rightarrow T$. The guard is introduced to make it possible to exclude states that may not be relevant.

Completeness

$$\begin{aligned} &\forall s \in S : guard(s) \wedge \sim goal(s) \\ &\qquad \Longrightarrow \exists a \in A \wedge per(a)(s) \wedge goal(a(s)) \end{aligned} \tag{1}$$

Feedback. When certain important actions are taken, a user needs to be aware of whether the resulting device status is appropriate or problematic (AAMI 2010). Feedback breaks down into *state feedback*, requiring that a change in the state (usually specific attributes of the state rather than the whole state) is visible to the user, and *action feedback*, requiring that an action always has an effect that is visible to the user.

State feedback

$$\begin{aligned} &\forall s1, s2 \in S,\ guard(s1) \wedge guard(s2) \wedge transit(s1, s2) \wedge \\ &\qquad filter(s1) \neq filter(s2) \\ &\qquad\qquad \Longrightarrow p\_filter(s1) \neq p\_filter(s2) \end{aligned} \tag{2}$$

Action feedback

$$\begin{aligned}&\forall a \in S \rightarrow S,\ \forall s \in S : per(a)(s) \wedge\\ &\quad guard(s) \wedge (filter(s) \neq filter(a(s)))\\ &\qquad \implies p\_filter(s) \neq p\_filter(a(s)) \qquad (3)\end{aligned}$$

In the case of state feedback, the guard may be used, for example, to restrict the analysis to ensure that the device or system is considered to be in the same mode as a result of the state transition. Variants of the *feedback* properties will also be used that assume separate *visible* attributes are not specified in the model. Instead, a relevant predicate $vis\_filter : S \rightarrow T$ is linked to $filter : S \rightarrow A$. $vis\_filter(s)$ is true for $s \in S$ if $filter(s)$ is visible. Both these variants will be used in Sect. 14.5. The choice is based on how the model is constructed.

Consistency. Users quickly develop a mental model that embodies their expectations of how to interact with a user interface. Because of this, the overall structure of a user interface should be consistent in its layout, screen structure, navigation, terminology and control elements (AAMI 2010). The consistency template is formulated as a property of a group of actions $A_c \subseteq \wp(S \rightarrow S)$, or it may be the same action under different modes, requiring that all actions in the group have similar effects on specific state attributes selected using a filter. The relation *consistent* connects a filtered state, before an action occurs, with a filtered state after the action. The description of the filters and the *consistent* relation capture the consistency across states and across actions.

Consistency

$$\begin{aligned}&\forall a \in A_c \subseteq \wp(S \rightarrow S), s \in S, m \in MS :\\ &\quad guard : S \times MS \rightarrow T\\ &\quad consistent : C \times C \rightarrow T\\ &\quad filter\_pre : S \times MS \rightarrow C\\ &\quad filter\_post : S \times MS \rightarrow C\\ &\quad guard(s, m) \wedge\\ &\quad consistent(filter\_pre(s, m),\\ &\qquad filter\_post(a(s), m) \qquad (4)\end{aligned}$$

Consistency is a property of, for example, the cursor move actions when the mouse button is pressed. It may be used to prove that while the button is down, the move actions will continue to be interpreted in the relevant mode.

Reversibility. Users may perform incorrect actions, and the device needs to provide them with functions that allow them to recover by reversing the effect of the incorrect action. The reversibility template is formulated using a $guard : S \rightarrow T$ and a $filter : S \rightarrow FS$, which extracts a set of focus attributes of the state:

Reversibility

$$\forall s \in S : guard(s) \implies \exists b : S \to S : filter(a(b(s)) = filter(s) \quad (5)$$

Some properties simply maintain invariants for any state. Examples of such properties are *visibility* and *universality*. There are alternative formulations of these two properties. The first asserts that a predicate applied to one filtered value is true if and only if an appropriate predicate is true of the other filtered value. The second asserts that a filtering of the first value is equal to the determined filter of the second value. These two formulations are appropriate in different circumstances as will be briefly explored in Sect. 14.5. The style of interface described in this case study lends itself particularly to the second option.

Visibility. This property describes an invariant relation between a state variable that is not necessarily visible to the user and a user interface value that is visible to the user. Examples of these properties are as follows: the current operational mode is always unambiguously displayed; a slider that shows the position of the control rods always shows the actual position of the control rods in the underlying process; the colour of the status attribute describes general characteristics of the value of the attribute. $filter(s)$ and $p\_filter(s)$ are the filters for the attribute and its perceivable counterpart.

Visibility

$$\forall s1, s2 \in S : transit(s1, s2) \land visible(s1) \implies visible(s2) \quad (6)$$

where $visible(s) = pred\_filter(s) \Leftrightarrow pred\_p\_filter(s)$ or $visible(s) = filter(s) = p\_filter(s)$

Universality. Universality generalises the visibility property requiring that given two filters of the state: $filter1$ and $filter2$, there are predicates on the filters that are equivalently true.

Universality

$$\forall s1, s2 \in S : transit(s1, s2) \land universal(s1) \implies universal(s2) \quad (7)$$

where $universal(s) = pred\_filter1(s) \Leftrightarrow pred\_filter2(s)$ or $universal(s) = filter1(s) = filter2(s)$

14.5 Modelling the Nuclear Power Plant Control User Interface

The fragments of specification described in this section were taken from the description to be found in Chap. 4 and a simulator (see the simulator description and code[2]) that includes a version of an interface to the nuclear controller. A more thorough description of the user interface was required than was available in the use case

[2] http://www.hci-modeling.org/nppsimulator/BWRSimulationDescription.pdf.

material to do a thorough analysis using the property templates. The analysis of IV infusion pumps described in Harrison et al. (2015a) illustrates the more thorough approach. If such a detailed description had been available, then it could be further explored using PVSio-web (Masci et al. 2015) to ensure that assumptions made seem realistic and to demonstrate where properties of the system fail to be true. This would provide confidence that the behaviour of the interface as specified conforms with the expected user experience. The issue of validation of the model of the system is explored in more detail in Harrison et al. (2014). The introduction to the use case contains the following paragraph.

> The operation of a nuclear power plant includes the full manual or partially manual starting and shutdown of the reactor, adjusting the produced amount of electrical energy, changing the degree of automation by activating or deactivating the automated steering of certain elements of the plant, and the handling of exceptional circumstances. In case of the latter, the reactor operator primarily observes the process because the safety system of today's reactors suspends the operator step by step from the control of the reactor to return the system back to a safe state.

The interface involves schematics of the process, the availability of actions as buttons and graphical indications of key parameters, for example temperature and levels. The specification of the model can be layered using the levels described in Sect. 14.3 as follows.

14.5.1 Types and Constants

This contains generic definitions that will be used throughout other layers of the theory. It defines types such as:

```
pump_type: TYPE = [# speed: speed_type,
                     on: boolean
                     #]
```

The pump attribute is defined to have a speed of flow and to be either on or off. There are several pumps with the same characteristics as defined by the following function type:

```
 pumps_type: TYPE = [vp_number -> pump_type]
```

This type definition allows the definition of multiple pumps indexed by an integer (`vp_number`). This type relates to the process layer. The following types are used in the interface layer.

```
cursor_type = TYPE [* x: x_type,
                      y: y_type *]
```

The type `cursor_type` specifies the type of the cursor on the controller display. It is tied to the physical position of the mouse. The details of how this is done will not be described here. It will be assumed that there is a function `mouse` that extracts the current cursor position of the mouse. The slider which is also found in the interface layer is specified as follows:

```
slider_type : TYPE =
                  [#  ypos: y_type,
                        lx: x_type,
                        rx: x_type,
                        xpos: x_type #]
```

The slider type specifies the current x-position of the cursor when the slider has been selected (`xpos`). It specifies the left and right limits of the slider (`lx` and `rx`) and the y-position of the slider (`ypos`). As a simplification for the illustration, the slider is assumed to have no depth. In the real system, sliders also have a depth and therefore the y-coordinates will also have boundaries.

14.5.2 The Process Layer

The process layer describes sufficient details of the underlying process of the nuclear reactor to provide an adequate underpinning for the interface. The interface captures, for example, those situations where the process automates and therefore removes the ability of the operator to change settings manually. The model describes the ongoing process in terms of a single action *tick* that updates the attributes of the pump state as time progresses.

```
tick(st: npp): npp =
st WITH
 [time := time(st) +1,
  sv :=
    LAMBDA (n: vp_number):
      COND
        n=1 -> (#
        flow :=
         COND
            sv(st)(1)‘on ->
            (reactor(st)‘pressure -
                 condensor(st)‘pressure)/10,
            ELSE -> 0
         ENDCOND,
        on := sv(st)(1)‘on
                   #),
        n=2 -> (#
        flow :=
           COND
              sv(st)(2)'on ->
                 (reactor(st)'pressure -
                   condensor(st)'pressure)/2.5,
              ELSE -> 0
           ENDCOND,
        on := sv(st)(2)'on
                      #)
       ENDCOND,
   poi_reactor :=
      LET num_reactor =
            (old_pos_rods_reactor(st) -
                 pos_rods_reactor(st))
        IN (
         COND
             num_reactor >= 0 ->
                   num_reactor / (time(st) -
                                time_delta_pos(st)),
             ELSE ->
                    - num_reactor /
                         (time(st) - time_delta_pos(st))
         ENDCOND ),
   bw := (2*(100 - pos_rods_reactor(st))*
                     (900-reactor(st)'pressure))/620,
      ...
      ]
```

This fragment of specification illustrates the form of the process layer. The action that is described is `tick`. Its function is to update the process state (defined by type `npp`) attributes. Some of these attributes can be changed by actions that can be invoked by the operator (e.g. the two valves `wv(1)` and `wv(2)`), while others are internal to the process (e.g. `poi_reactor` and `bw`).

Further actions determine transformations of the process that can be invoked directly through the user interface. For example `control_rods` is an action that updates the process state to its new position. This position is determined by where the cursor is, as represented in the control rods slider, when the rod's position is not under automatic control.

14.5.3 The Interface Layer

The interface layer describes those attributes of the state of the process that are visible to the user and the actions that can be performed by the operator. The interface presents the state of the process to provide the operator with situation awareness.

There are also displayed attributes that indicate sliders and buttons that can be used by the operator to control aspects of the process.

An illustration of the layer considers the mouse actions: *move*, *click* and *release*. The actions *move* and *release* have effects that depend on the modes that the interface is in. The action *click* changes the mode. The change in mode depends on the position of the cursor. Two sets of modes are specified. One set relates to the sliders on the display (slider_mode), while the other relates to the actions that are offered by the display (action). When the mouse has not been clicked or is over a space in the display that does not correspond to a slider or an action, then slider_mode = nulslimo and action = null_action. When the mouse is clicked, then a boolean attribute clicked is true. The permission that allows click to be an available action is as follows:

```
per_click(st: npp_int): boolean = NOT clicked(st)
```

click is an action that:

1. assigns a value to slider mode if the cursor is at the appropriate y-coordinate (ypos) and in the relevant range of x-coordinates for the slider (in reality, there may also be a range of y-coordinates) otherwise it sets the slider mode to the relevant null value.
2. assigns a value to action, the action mode, if the x- and y-coordinates are within the relevant range of an action button and the action is enabled. If the cursor is outside the defined button ranges, then the action is set to null.

Fragments of the action specification are given below. They show the effect when the mouse is within the slider for the pump *wp*1 and the effect when the mouse is in the area of the action button that sets control rods to automatic mode.

```
click(st : npp_int): npp_int =
  LET x = cursor(st)`x AND y = cursor(st)`y IN
  st WITH
  [ slidermode :=
    COND
      y = wp1_slider(st)`ypos AND
      (x >= wp1_slider(st)`lx) AND
      (x <= wp1_slider(st)`rx) AND
                              NOT auto_wp1s(st) -> wp1s,
      ...
      ELSE -> nulslimo
    ENDCOND,
    action :=
    COND
      ((x <= acrarea(st)`lx) AND
      (x >= acrarea(st)`rx) AND
```

```
         (y <= acrarea(st)‘dy) AND
         (y >= acrarea(st)‘uy)) ->
               COND
                   auto_cr -> acroff,
                   ELSE -> acron
                ENDCOND,
         ...
         ELSE -> null_action
       ENDCOND,
       ...
       clicked := true
      ]
```

Different actions and therefore modes are specified depending on whether the pumps or the control rods are in automatic mode. Moving the cursor has different significance, depending on whether the mouse is clicked or not. When the mouse is clicked outside the sensitive areas or when the mouse button is not depressed, then the cursor coordinates only are changed. If the mouse is clicked within the space, then the appropriate mode is taken. If the mode is related to a slider, then the cursor on the slider is moved.

```
  move(st: npp_int): npp_int =
      LET new_cursor = mouse(st) IN
      st WITH [
        cursor := new_cursor,
        new_wp1speed :=
         COND
          (slidermode = wp1s) AND
          (new_cursor‘x > wp1_slider(st)‘lx) AND
          (new_cursor‘x <= wp1_slider(st)‘rx)
             -> (wp1_slider(st)‘lx - new_cursor‘x) * max_flow /
                   (wp1_slider(st)‘lx - wp1_slider(st)‘rx),
          ELSE -> new_wp1speed(st)
         ENDCOND,
       new_wp2speed := ...,
       new_cpspeed := ...,
       new_crposition := ...,
      ]
```

release has the effect of invoking actions in the process layer if the slider mode is non-null and also the action is non-null. Hence, for example, if `slidermode = wp1s`, that is the flow rate of *wp*1 is being changed, then a function is invoked that changes the flow rate of the pump. This function is defined as part of the interface layer, but it invokes the relevant function in the process layer.

```
release(st: npp_int): npp_int =
    COND
      slider_mode(st) = wp1s ->
         modify_wp1flow(st),
      slider_mode(st) = wp2s ->
         modify_wp2flow(st),
      slider_mode(st) = cps ->
         modify_cpflow(st),
      slider_mode(st) = crs ->
        modify_crpos(st),
      ELSE -> perform_action(st)
    ENDCOND
```

Aspects of the status of the process are captured in indicators (e.g. RKS, RKT). The colours of the indicators are linked to the states of the underlying reactor (modelled in the second layer), for example if the value of a process attribute is outside specified bounds then the indicator shows the colour red. The model also specifies that the user can perform open/close actions on valves by highlighting the available option in the display.

14.5.4 Proving Properties of the Interface Layer

Two examples will be used to illustrate how the template properties are instantiated in the interface layer. The first example is concerned with the *visibility* of different aspects of the underlying process, while the second is concerned with the *consistency* when releasing the mouse button. The concern in the first example is with the CP pump as it transports water through the cooling pipes in the condenser. A display, specified by the attribute that is part of the interaction mode, *status_cp* simply indicates whether the pump flow is normal (green) or out of bounds (red). The property to prove is that the display shows these colours correctly. This can be proved by instantiating the visibility property template (Eq. 6 in Sect. 14.4.2).

```
cp_status_visible: THEOREM
   FORALL (pre, post: npp_int):
     init_state(pre) => cp_visible(pre) AND
     transit(pre, post) AND cp_visible(pre) =>
             cp_visible(post)
```

This theorem contains an induction based on the accessible states. The initial state is specified by the predicate `init_state`. The visibility property to be proved is:

```
cp_visible(st: npp_int):
  (pred_cp_filter(st) <=> pred_p1_cp_filter(st)) AND
  (NOT pred_cp_filter(st) <=> pred_p2_cp_filter)
```

The filter predicates are specified as follows:

```
pred_cp_filter(st: npp_int): boolean =
       process(st)`cp`speed > max_flow
pred_p1_cp_filter(st: npp_int): boolean =
       status_cp(st) = red
pred_ps_cp_filter(st: npp_int): boolean =
       status_cp(st) = green
```

Similar properties can be proved of a range of display features relating to, for example, the status of the process attributes; whether the pumps and control rods are in automatic mode; the values of pump flows and the position of control rods; whether it is possible to switch pumps on or off.

To illustrate the *consistency* property (Eq. 4 in Sect. 14.4.2), the *release* action is considered in relation to the sliders. The theorem instantiates the template properties indicated in the formulation of the property. The guard and consistency predicates are specified over all the slider modes. We therefore slightly modify the formulation. The aim is to prove the property:

```
consistency_sliders(st: npp_int): boolean =
  con_guard(st) IMPLIES con_release(st)
```

There are four modes to be considered in MS, namely `wp1s`, `wp2s`, `cps` and `crs`. The guard checks that the mouse cursor is in the relevant region of the display depending on mode that release is permitted and that the particular function is not currently automated.

```
con_guard(st: npp_int): boolean =
x_in_area(cursor(st)`x, slidermode(st), st) AND
per_release(st) AND NOT auto(slidermode(st), st)
```

The predicate `x_in_area` checks that the cursor is in a relevant position in relation to the slider.

```
x_in_area(x: x_type, sl: slimo_type, st: npp_int): boolean =
((sl=wp1s) AND
   (x>=wp1_slider(st)`lx) AND (x<=wp1_slider(st)`rx)) OR
((sl=wp2s) AND
   (x>=wp2_slider(st)`lx) AND (x<=wp2_slider(st)`rx)) OR
((sl=wp1s) AND
   (x>=cp_slider(st)`lx) AND (x<=cp_slider(st)`rx)) OR
```

```
((sl=wp2s) AND
   (x>=rods_slider(st)`lx) AND (x<=rods_slider(st)`rx))
```

The consistency relation is distributed across these modes as follows:

```
con_release(sl: slimo_type, st: npp_int): boolean =
release(st) =
st WITH
[ pump :=
  COND
    sl = wp1s -> pump(st)`wp1_flow(
                  (cursor(st)`x - wp1_slider(st)`lx)*
                   (flow_range/sliderrange)),
    sl = wp2s -> pump(st)`wp2_flow(
                  (cursor(st)`x - wp2_slider(st)`lx)*
                   (flow_range/sliderrange)),
    sl = cps -> pump(st)`cp_flow(
                  (cursor(st)`x - wp2_slider(st)`lx)*
                   (flow_range/sliderrange)),
    sl = crs -> pump(st)`control_rods(
                  (cursor(st)`x - crs_slider(st)`lx)*
                   (control_range/sliderrange)),
    ELSE -> pump(st)
   ENDCOND,
   slider_mode := nulslimo,
   action := nullaction ]
```

The relation *consistent* is equally distributed over the modes; the `filter_pre` indicates what the new state of the process should be, that is for each mode a function in the underling process should be invoked that updates the relevant state attribute.

```
filter_pre(st) =
st WITH
[ pump :=
  COND
    sl = wp1s -> pump(st)`wp1_flow(
                  (cursor(st)`x - wp1_slider(st)`lx)*
                   (flow_range/sliderrange)),
    sl = wp2s -> pump(st)`wp2_flow(
                  (cursor(st)`x - wp2_slider(st)`lx)*
                   (flow_range/sliderrange)),
    sl = cps -> pump(st)`cp_flow(
                  (cursor(st)`x - wp2_slider(st)`lx)*
                   (flow_range/sliderrange)),
    sl = crs -> pump(st)`control_rods(
                  (cursor(st)`x - crs_slider(st)`lx)*
                   (control_range/sliderrange)),
    ELSE -> pump(st)
   ENDCOND,
   slider_mode := nulslimo,
   action := nullaction ]
```

and `filter_post(st) = release(st)`. *Consistency* relates the change in state `filter_post(st)` to the state before the action in which the mode determined action takes place in the process layer. It determines that *release* always invokes the mode-relevant process action (changing pump flow or control rod position). The instantiated consistency theorem is an induction on the actions of the interaction model.

```
consistency_sliders_thm: THEOREM
  FORALL (pre, post: npp_int):
    (init_state(pre) IMPLIES consistency_sliders(pre))
 AND
    (consistency_sliders(pre) AND transit(pre, post)) =>
      consistency_sliders(post)
```

Attempting to prove this theorem identifies an issue with the simulator display. The four sliders occupy the same x-space. The sliders are implemented so that the slider will continue to be dragged across even when the y-coordinate is not in the slider area relating to the mode. It would be imagined that this characteristic would not be a feature of the real control room display.

14.5.5 The Activity Layer

The purpose of the activity layer is to specify assumptions about how the attributes specified in the interface layer, as well as other specified attributes that may be external, are used to carry out the intended activities of the system. It is clearly necessary to know what the activities are that will be performed by the controllers. Typically, this information would be gathered by observing existing processes, by interviewing controllers, or by developing scenarios with domain experts that relate to anticipated constraints in terms of new design concepts. The approach is described in more detail in Campos et al. (2014).

Given the limited information provided by the use case, it is difficult to develop and assess plausible assumptions. However, we do have operating procedures associated with starting up and closing down the reactor. This will be the information that provides the basis for sketches of the activity layer given here.

The aim of start-up is to bring output power to 700 MW (100% of possible output power) and to hold the water level in the reactor tank stable at 2100 mm. The operating procedure is as follows:

1. Open SV2
2. Set CP to 1600 u/min
3. Open WV1
4. Set WP1 to 200 u/min
5. Stabilise water level in the reactor tank at 2100 mm by pulling out the control rods
6. Open SV1
7. …

The process of developing the activity model involves, for each step in the operating procedure, considering the information resources that are conjectured to enable the user to take the appropriate action in the interface. The action is resourced if information relevant to the use of the action is clear in the interface. The activity model also considers how the user is notified what the next action is to be performed. In the case of this fragment, it will be assumed that the written operating procedure will be used to decide the sequence. However, in other circumstances it should be considered whether the user will be allowed by the control system to change the order of the operating procedure and what the effect of such a change would be. The action `OpenSV2` is expressed in the model as

```
open_valve(st WITH [action := opensv2])
```

The open valve action is generic to the valves supported by the interface and is made specific by the attribute `action`. It may be assumed that this action would be triggered if

- the openSV2 button area is enabled, that is it is highlighted: `highlightosv2 = true`. This should only be true if `sv2_open = false`, a property checked of the interface model.
- the cursor is within the osv2 area:

```
    (cursor(st)`x <= osv2area(st)`lx) AND
    (cursor(st)`x >= osv2area(st)`rx) AND
    (cursor(st)`y <= osv2area(st)`dy) AND
    (cursor(st)`y >= osv2area(st)`uy)
```

The resource layer specifies all the constraints based on assumptions about what triggers the actions supported by the interface as well as activities that are to be performed. When these assumptions have been specified, they can be used as additional constraints when proving theorems based on the templates. The resource layer makes it possible to prove whether the properties are true in circumstances that afford some measures of plausibility in relation to what users do.

Additional actions may be specified that characterise the activities performed by users. For example, consider the user activity *recover*, in contrast to the autonomous action that causes recovery. This activity would involve several actions before the goal of the activity is achieved. Information resources would help the operator to begin the activity. This means that the activity also has information resource constraints. For example, it would specify that "increasing pressure", using the relevant action in the interface layer, would occur only if other actions had already been completed and the displayed tank, valve and pump parameters specified in the second layer were displayed (in the interface layer), indicating particular values.

Further activities include, for example, "monitor recovery". This would be expressed as an action that describes the constraints on the operator when monitoring an autonomous recovery. The specification of the action would include the information resources that would be required in the monitoring process at different stages of the recovery and would specify the conditions in which any user actions would take place.

The value of expressing constraints in this way is that theorems that are instantiations of property patterns of Sect. 14.4.2 can be proved subject to the resource constraints. Properties can be considered that only relate to plausible interactions. This would be relevant if it was considered inappropriate to analyse properties across sequences of actions that would not plausibly occur. The implications of such an analysis are that an understanding of whether an action is plausible becomes more relevant, and this requires an understanding of the human factors of a situation. This topic is considered in more detail in Harrison et al. (2016). It can also be proved that, for the steps of the operating procedures, the constraints are satisfied.

14.6 Related Work

Models, of the type outlined, have been developed for other interactive systems using both model checking approach and theorem-proving approach (Masci et al. 2012; Harrison et al. 2014, 2015b; Campos et al. 2016). The advantage of model checking is that it is possible to explore, more readily, reachability properties as well as potential non-determinisms. The disadvantage is that the size of model is seriously limited. It is possible to explore the essential details of the control of the nuclear power plant using a model checking approach, but as soon as a realistic process model is used this becomes impossible. Making the model abstract enough, to make analysis feasible, would restrict what could be asked of the model. It would be more difficult to prove relevant properties.

Theorem proving allows analysis of larger models but properties may be more difficult to formulate and prove. In particular, while model checking allows simple formulations of reachability properties, these are difficult to specify using a theorem-proving approach. There is a trade-off to be made between the effort needed to develop a model amenable to verification and the effort needed to carry out the proofs. Typically, a theorem proving-based approach will gain advantage in the former, because of more expressive languages, and model checking in the latter, because of more automated analysis. In all the cases, how to identify and express the properties of interest is also an issue.

Design patterns and property templates have been extensively studied in engineering practices. Most of the effort, however, has been devoted to creating patterns and templates for the control part of a system, rather than for the human–machine interface. Vlissides et al. (1995) established a comprehensive set of standard design patterns for software components of a system. An example pattern is the *abstract factory*, which facilitates the creation of families of software objects (e.g. windows of a user interface). Another example is the *adapter* pattern, which converts the interface of software components to enable the integration of otherwise incompatible software components. These patterns are a de facto standard in the software engineering community, and they are widely adopted in engineering practices to solve common problems related to the realisation of software components. Konrad and Cheng (2002) discuss design patterns for the elements of embedded systems. An example pattern is the *actuator–sensor* pattern, providing a standard interface for sensors and actuators connected to the control unit of an embedded system. Similarly, Sorouri et al. (2012) present a design pattern for representing the control logic of an embedded system. Lavagno et al. (1999) introduced *Models of computation (MoC)* as design patterns for representing interactions between distributed system components. Recently, Steiner and Rushby (2011) have demonstrated how these MoC can be used in model-based development of systems, to represent in a uniform way different time synchronisation services executed within the system. These and similar activities are concerned with the design patterns for the control part of a system, as opposed to the human–machine interface—e.g. problems like how to correctly design the behaviour of data entry software in human–machine interfaces are out of scope.

Various researchers have introduced design patterns for the analysis of complex systems. For example, in Li et al. (2014), verification patterns are introduced that can be used for the analysis of safety interlock mechanisms in interoperable medical devices. Although they use the patterns to analyse use-related properties such as "*When the laser scalpel emits laser, the patient's trachea oxygen level must not exceed a threshold* Θ_{O_2}", the aim of their patterns is to facilitate the introduction of a model checker in the actual implementation of the safety interlock, rather than defining property templates for the analysis of use-related aspects of the safety interlock. Other similar work, e.g. Tan et al. (2015), King et al. (2009), Larson et al. (2012), also introduce design patterns for the verification of safety interlocks, but the focus of the patterns is again on translating verified design models into a concrete implementation—in Tan et al. (2015), for example—the design patterns are developed for the automatic translation of hybrid automata models of a safety interlock into a concrete implementation.

Proving requirements similar to the properties produced from the templates of this paper has been the focus of previous work. For example, a mature set of tools has been developed using SCR (Heitmeyer et al. 1998). Their approach uses a tabular notation to describe requirements which makes the technique relatively acceptable to developers. Combining simulation with model checking has also been a focus, in other work, for example Gelman et al. (2013). Recent work concerned with simulations of PVS specifications provides valuable support to this complementarity (Masci et al. 2013). Had the specification been developed as part of a design process, then a tool such as Event B (Abrial 2010) might have been used. In such an approach, an initial model is first developed that specifies the device characteristics and incorporates the safety requirements. This model is gradually refined using details about how specific functionalities are implemented.

In our previous work, we have introduced modelling patterns for decomposing interactive (human–machine) system models into a set of layers to facilitate models reuse (Harrison et al. 2015b). Bowen and Reeves (2015), who are concerned with design patterns for user interfaces, complements our work on modelling patterns. They have introduced four modelling patterns: the *callback* pattern, representing the behaviour of confirmation dialogues used to confirm user operations; the *binary choice* pattern, representing the behaviour of input dialogues used to acquire data from the user; the *iterator* pattern, representing the behaviour of parametric user interface widgets that share the same behaviour but have a different value parameter, such as the numeric entry keys 0–9; and the *update* pattern, for representing the behaviour of a numeric display.

14.7 Discussion and Conclusions

Two approaches to specification and proof are possible with the considered examples: model checking and theorem proving. Model checking is the more intuitive of the two approaches. Languages such as Modal Action Logic with interactors (MAL) (Campos 2008) express state transition behaviour in a way that is more acceptable to non-experts. The problem with model checking is that state explosion can compromise the tractability of the model so that properties to be proved are not feasible. Model checking, hence, is more convenient for analysing high-level behaviour, for example when checking the modal behaviour of the user interface. Theorem proving, while being more complex to apply, provides more expressive power. This makes it more suitable when verifying properties requires a high level of detail, such as those related to a number entry system, because the domain of numbers is relatively large.

To employ the strength of the two approaches, simple rules can be used to translate from the MAL model to the PVS model that is used for theorem proving. Actions are modelled as state transformations, and permissions that are used in MAL to specify when an action is permitted are described as predicates. The details of the specification carefully reflects its MAL equivalent. This enables us to move between the notations and verification tools, choosing the more appropriate tool for the verification goals at hand.

One aspect that has not been discussed in this chapter is the analysis and interpretation of verification results. Interpretation may be facilitated through the animation of the formal models to create prototypes of the modelled interfaces. These prototypes make it easier to discuss the results of verification with stakeholders. Such prototypes can be used either to *replay* traces produced by a model checker or interactively to both discuss the findings of the verification or help identify relevant features of the system that should be addressed by formal analysis. This approach is described in Masci et al. (2014).

Acknowledgements José Creissac Campos and Michael Harrison were funded by project ref. NORTE-07-0124-FEDER-000062, co-financed by the North Portugal Regional Operational Programme (ON.2 O Novo Norte), under the National Strategic Reference Framework (NSRF), through the European Regional Development Fund (ERDF), and by national funds, through the Portuguese foundation for science and technology (FCT). Paul Curzon, Michael Harrison and Paolo Masci were funded by the CHI+MED project: Multidisciplinary Computer Human Interaction Research for the design and safe use of interactive medical devices project, UK EPSRC Grant Number EP/G059063/1.

References

AAMI (2010) Medical devices—application of usability engineering to medical devices. Technical Report ANSI AMI IEC 62366:2007, Association for the advancement of medical instrumentation, 4301 N Fairfax Drive, Suite 301, Arlington VA 22203-1633

Abrial JR (2010) Modeling in event-B: system and software engineering. Cambridge University Press

Bolton ML, Bass EJ, Siminiceanu RI (2012) Generating phenotypical erroneous human behavior to evaluate human-automation interaction using model checking. Int J Human-Comput Stud 70:888–906

Bowen J, Reeves S (2015) Design patterns for models of interactive systems. In: 2015 24th Australasian software engineering conference (ASWEC). IEEE, pp 223–232

Campos JC, Harrison MD (2008) Systematic analysis of control panel interfaces using formal tools. In: Graham N, Palanque P (eds) Interactive systems: design, specification and verification, DSVIS ’08. Springer, no. 5136 in Springer lecture notes in computer science, pp 72–85

Campos JC, Harrison MD (2009) Interaction engineering using the IVY tool. In: Graham T, Gray P, Calvary G (eds) Proceedings of the ACM SIGCHI symposium on engineering interactive computing systems. ACM Press, pp 35–44

Campos JC, Doherty G, Harrison MD (2014) Analysing interactive devices based on information resource constraints. Int J Human-Comput Stud 72:284–297

Campos JC, Sousa M, Alves MCB, Harrison MD (2016) Formal verification of a space system’s user interface with the IVY workbench. IEEE Trans Human Mach Syst 46(2):303–316

Duke DJ, Harrison MD (1993) Abstract interaction objects. Comput Graph. Forum 12(3):25–36

Gelman G, Feigh K, Rushby J (2013) Example of a complementary use of model checking and agent-based simulation. In: 2013 IEEE international conference on, systems, man, and cybernetics (SMC), pp 900–905. doi:10.1109/SMC.2013.158

Gow J, Thimbleby H, Cairns P (2006) Automatic critiques of interface modes. In: Gilroy S, Harrison M (eds) Proceedings 12th international workshop on the design, specification and verification of interactive systems. Springer, no. 3941 in Springer lecture notes in computer science, pp 201–212

Harrison M, Campos J, Masci P (2015a) Patterns and templates for automated verification of user interface software design in pvs. Technical report TR-1485, School of computing science, Newcastle university

Harrison M, Campos J, Masci P (2015b) Reusing models and properties in the analysis of similar interactive devices. Innovations Syst Soft Eng 11(2):95–111

Harrison M, Campos J, Ruksenas R, Curzon P (2016) Modelling information resources and their salience in medical device design. In: EICS '16 proceedings of the 8th ACM SIGCHI symposium on engineering interactive computing systems. ACM Press, pp 194–203

Harrison MD, Masci P, Campos JC, Curzon P (2014) Demonstrating that medical devices satisfy user related safety requirements. In: Proceedings of fourth symposium on foundations of health information engineering and systems (FHIES) and sixth software engineering in healthcare (SEHC) workshop. Springer, in press

Heitmeyer C, Kirby J, Labaw B (1998) Applying the SRC requirements method to a weapons control panel: an experience report. In: Proceedings of the second workshop on formal methods in software practice (FMSP '98), pp 92–102

King AL, Procter S, Andresen D, Hatcliff J, Warren S, Spees W, Jetley R, Raoul P, Jones P, Weininger S (2009) An open test bed for medical device integration and coordination. In: ICSE companion, pp 141–151

Konrad S, Cheng BHC (2002) Requirements patterns for embedded systems. In: Proceedings of IEEE joint international conference on requirements engineering. IEEE, pp 127–136

Larson B, Hatcliff J, Procter S, Chalin P (2012) Requirements specification for apps in medical application platforms. In: Proceedings of the 4th international workshop on software engineering in health care. IEEE Press, pp 26–32

Lavagno L, Sangiovanni-Vincentelli A, Sentovich E (1999) Models of computation for embedded system design. In: System-level synthesis. Springer, pp 45–102

Li T, Tan F, Wang Q, Bu L, Cao J, Liu X (2014) From offline toward real time: a hybrid systems model checking and CPS codesign approach for medical device plug-and-play collaborations. IEEE Trans Parallel Distrib Syst 25(3):642–652

Masci P, Huang H, Curzon P, Harrison MD (2012) Using PVS to investigate incidents through the lens of distributed cognition. In: Goodloe AE, Person S (eds) NASA formal methods, Lecture notes in computer science, vol 7226. Springer, Berlin, Heidelberg, pp 273–278. doi:10.1007/978-3-642-28891-3_27

Masci P, Ayoub A, Curzon P, Lee I, Sokolsky O, Thimbleby H (2013) Model-based development of the generic PCA infusion pump user interface prototype in PVS. In: Bitsch F, Guiochet J, Kaâniche M (eds) Computer safety, reliability, and security, Springer lecture notes in computer science, vol 8153. Springer, pp 228–240

Masci P, Zhang Y, Jones P, Curzon P, Thimbleby HW (2014) Formal verification of medical device user interfaces using PVS. In: 17th international conference on fundamental approaches to software engineering, ETAPS/FASE2014. Springer, Berlin, Heidelberg

Masci P, Oladimeji P, Curzon P, Thimbleby H (2015) PVSio-web 2.0: joining PVS to human-computer interaction. In: 27th international conference on computer aided verification (CAV2015). Springer, Tool and application examples available at http://www.pvsioweb.org

Nielsen J, Molich R (1990) Heuristic evaluation of user interfaces. In: Chew J, Whiteside J (eds) ACM CHI proceedings CHI '90: empowering people, pp 249–256

Polson PG, Lewis C, Rieman J, Wharton C (1992) Cognitive walkthroughs: a method for theory-based evaluation of user interfaces. Int J Man-Mach Stud 36(5):741–773

Shankar N, Owre S, Rushby JM, Stringer-Calvert D (1999) PVS system guide, PVS language reference, PVS prover guide, PVS prelude library, abstract datatypes in PVS, and theory interpretations in PVS. Computer science laboratory, SRI international, Menlo Park, CA. http://pvs.csl.sri.com/documentation.shtml

Sorouri M, Patil S, Vyatkin V (2012) Distributed control patterns for intelligent mechatronic systems. In: 2012 10th IEEE international conference on industrial informatics (INDIN). IEEE, pp 259–264

Steiner W, Rushby J (2011) TTA and PALS: formally verified design patterns for distributed cyber-physical systems. In: 2011 IEEE/AIAA 30th digital avionics systems conference (DASC). IEEE

Tan F, Wang Y, Wang Q, Bu L, Suri N (2015) A lease based hybrid design pattern for proper-temporal-embedding of wireless CPS interlocking. IEEE Trans Parallel Distrib Syst 26(10):2630–2642

Vlissides J, Helm R, Johnson R, Gamma E (1995) Design patterns: elements of reusable object-oriented software, vol 49, no 120. Addison-Wesley, Reading, p 11

Chapter 15
Formal Analysis of Multiple Coordinated HMI Systems

Guillaume Brat, Sébastien Combéfis, Dimitra Giannakopoulou, Charles Pecheur, Franco Raimondi and Neha Rungta

Abstract Several modern safety-critical environments involve multiple humans interacting not only with automation, but also between themselves in complex ways. For example, in handling the National Airspace, we have moved from the traditional single controller sitting in front of a display to multiple controllers interacting with their individual display, possibly each other's displays, and other more advanced automated systems. To evaluate safety in such contexts, it is imperative to include the role of one or multiple human operators in our analysis, as well as focus on properties of human automation interactions. This chapter discusses two novel frameworks developed at NASA for the design and analysis of human–machine interaction problems. The first framework supports modeling and analysis of automated systems from the point of view of their human operators and supports the specification and verification of HMI-specific properties such as mode confusion, controllability, or whether operator tasks are compatible with a particular system. The second framework captures the complexity of modern HMI systems by taking a multi-agent approach to modeling and analyzing multiple human agents interacting with each other as well as with automation.

G. Brat · D. Giannakopoulou · N. Rungta
NASA Ames Research Center, Moffett Field, CA, USA
e-mail: Guillaume.P.Brat@nasa.gov

D. Giannakopoulou
e-mail: Dimitra.Giannakopoulou@nasa.gov

N. Rungta
e-mail: Neha.S.Rungta@nasa.gov

S. Combéfis (✉)
École Centrale des Arts et Métiers (ECAM), Woluwé-Saint-Lambert, Belgium
e-mail: sebastien.combefis@uclouvain.be; s.combefis@ecam.be

C. Pecheur
Université catholique de Louvain, Louvain-la-Neuve, Belgium
e-mail: charles.pecheur@uclouvain.be

F. Raimondi
Middlesex University, London, UK
e-mail: f.raimondi@mdx.ac.uk

B. Weyers et al. (eds.), *The Handbook of Formal Methods in Human-Computer Interaction*, Human-Computer Interaction Series,
DOI 10.1007/978-3-319-51838-1_15

15.1 Introduction

In a number of complex and safety-critical environments such as health care systems, autonomous automobiles, airplanes, and many others, the role of the human operator has shifted from manual control to "monitor and react". Human operators spend a large portion of their time monitoring automated systems and a smaller portion of their time performing actions based on the information provided by the system. There are several examples of such automated systems in use today including autopilots and collision avoidance systems in airplanes, systems that dispense prescribed dosage of medicine to patients, and systems used to perform trading in the financial markets. We expect this trend to continue with the introduction of autonomous cars, unmanned aerial vehicles, remote surgeries, and smart automation in our homes, among others.

To evaluate safety, it is imperative to consider the Human–Machine Interaction (HMI) system as a whole, by including the role of the human operator. Our research at NASA in this domain is targeted primarily on the National Airspace (NAS) in the context of civil aviation, but the techniques presented in this paper can be and have been applied to other domains as well.

The NAS is a complex environment in which humans and automated systems interact to enable safe transportation across the USA. The NAS has been designed to guarantee safety through human decision making with some support from automated systems. In other words, humans are at the core of the current air traffic management system. However, in order to accommodate increasing density of air traffic in the USA, there is a need to increase the use of automated systems that can assist humans such as air traffic controllers. Increased automation is also supported by FAA's Next Generation Air Transportation System (NextGen) program, where the FAA is the regulatory authority for civil aviation in the USA.

The NAS is turning into a complex, interconnected system, in which humans interact with automation extensively: we move from the traditional single controller sitting in front of a display to multiple controllers interacting with their individual display, possibly each other's displays, and other more advanced automated systems. Interactions can be as complex as information flowing from an automated system (possibly merging information coming from several automated systems) to another automated system with human controllers debating options to ensure safe and efficient traffic flow.

Despite the fact that increased automation is essential in assisting humans to problem solve in such a complex environment, infusion of a new technology is a very conservative process driven by the need to maintain current levels of safety. The current infusion process (from creating to certifying and fielding new automated systems) is extremely slow and limits our ability to cope with increasing traffic demands. This puts more pressure on the controllers and might result in pockets of reduced safety, which is not acceptable.

The main reason for the slow pace of infusion of new technology is our reliance on testing and human studies to assess the quality and consequences of fielding new automation. Human studies are very useful and provide many insights as to how controllers can react to new automation. However, they are always limited in scale and therefore fall short of accounting for off-nominal conditions and emergent behavior in this large, connected system that is the NAS. Clearly, we need better methods for analyzing the implications of fielding new automation.

This chapter discusses two novel frameworks developed at NASA for the design and analysis of human–machine interaction problems. The first framework supports modeling and analysis of automated systems from the point of view of their human operators in terms of observations made and commands issued. It supports the specification and verification of HMI-specific properties such as mode confusion, controllability, or whether operator tasks are compatible with a particular system.

The second framework aims at directly capturing the complexity of HMI systems involving multiple users interacting with each other as well as multiple automated systems. It provides a multi-agent approach to modeling and analyzing multiple human agents, e.g., pilots and controllers, interacting with each other as well as with automation. These models are created at a higher-level of abstraction that allows us to express actions, tasks, goals, and beliefs of the human such that experts in the domain of air traffic management and airspace system are able to understand the models. The multi-agent system framework provides support for modeling and scalable analysis to detect safety issues such as loss of separation, aircraft collision, or other user-specified properties in complex HMI systems. We describe the application of these two frameworks to realistic HMI systems.

15.2 From HMI Frameworks to Full-Blown Multi-agent Systems

Automated systems are increasingly complex, making it hard to design interfaces for human operators. Human–machine interaction (HMI) errors like automation surprises are more likely to appear and lead to system failures or accidents. Our work studies the problem of generating system abstractions, called mental models, that facilitate system understanding while allowing proper control of the system by operators as defined by the full-control property (Combéfis et al. 2011). Both the domain and its mental model have labeled transition systems (LTS) semantics. In this context, we developed algorithms for automatically generating minimal mental models as well as checking full-control. We also proposed a methodology and an associated framework for using the above and other formal method-based algorithms to support the design of HMI systems. The framework, implemented in the JavaPathfinder model checker (JavaPathfinder 2016), can be used for modeling HMI systems and analyzing models against HMI vulnerabilities. The analysis can be used for validation purposes or for generating artifacts such as mental models, manuals, and recovery procedures.

Despite the flexibility of our framework in capturing and analyzing properties that are specific to HMI systems, such as mode confusion, controllability by human operators, or whether an operator task is supported by an automated system, it has been clear from early stages of our research that expressing HMI systems directly in terms of LTSs is unmanageable for realistic systems. For this reason, we bridged our analysis framework with environments that domain experts use to model and prototype their systems. In particular, our efforts have focused on the NASA Ames HMI prototyping tool ADEPT (Combéfis et al. 2016). We extended the models that our HMI analysis framework handles to allow adequate representation of ADEPT models. We also provided a property-preserving reduction from these extended models to LTSs, to enable application of our LTS-based formal analysis algorithms. We demonstrated our work on several examples, most notably on an ADEPT autopilot (Combéfis 2013).

However, as discussed, the scale of modern HMI systems goes beyond the modeling and analysis capabilities of such tools. To handle the full scale of the NAS, where multiple humans interact with multiple automated systems, one needs to support appropriate high-level modeling mechanisms as well as a variety of analysis capabilities that range from simulation to exhaustive verification techniques, when possible. To address this challenge, we have moved toward modeling HMI systems as multi-agent systems.

Multi-agent systems (MAS) offer a design abstraction and modeling approach for systems involving humans and automation. They provide the ability for predictive reasoning about various safety conditions such as expected behavior of the autonomy, situational awareness of humans, workload of the human operators, and the amount of time taken from the start to the end of a complete or partial operation or procedure.

Rational agents are commonly encoded using belief–desire–intention (BDI) architectures. BDI architectures, originally developed by Michael Bratman (1987), represent an agent's mental model of information, motivation, and deliberation. Beliefs represent what the agent believes to be true, desires are what the agent aims to achieve, and intentions are how the agent aims to achieve its desires based on its current beliefs. The BDI model is a popular model for representing rational agents, i.e., agents that decide their own actions in a rational and explainable way. This success is probably due to its intuitive representation of human practical reasoning processes.

Our multi-agent framework is an extension of Brahms—a BDI-based MAS framework. Brahms is a simulation and development environment originally designed to model the contextual situated activity behavior of groups of people in a real world context (Clancey et al. 1998; Sierhuis 2001). Brahms has now evolved to model humans, robots, automated systems, agents, and interactions between humans and automated systems. Our extension connects Brahms to a variety of simulation and formal verification tools to enable the safety analysis of NAS models.

15.3 Formal Design and Analysis Techniques for HMI Properties

In this section, we describe our work on developing an HMI-specific formal analysis framework. In our framework, HMI models are represented using an extension of labeled transition systems (LTS) and are analyzed to assert system controllability given the commands that can be issued and the observations that can be made by the human operator. We illustrate the capabilities of the framework with a simple example from the medical community. Even though LTSs are an ideal abstraction for the specification and analysis of HMI properties, it is hard to model large, complex HMI systems directly as LTSs. We therefore also describe our work on connecting our analysis algorithms to the ADEPT prototyping tool, which provides a more intuitive modeling environment for domain experts.

15.3.1 Extended LTS

Since we are interested in controllability properties and detection of automation surprises, the distinction between commands and observations matters (Heymann and Degani 2007; Javaux 2002). Our approach is based on two models: the *system model*, which describes the detailed behavior of the system, and the *mental model*, which represents an abstraction of the system for the human operator.[1]

Our formal models are expressed with enriched *labeled transition systems* (LTS) called HMI LTS. HMI LTSs are essentially graphs whose edges are labeled with actions. The difference with classical LTSs is that three kinds of actions are defined:

1. *Commands* are actions triggered by the user on the system; they are also referred to as inputs to the system;
2. *Observations* are actions autonomously triggered by the system but that the user can observe; they are also referred to as outputs from the system;
3. *Internal actions* are neither controlled nor observed by the user; they correspond to internal behaviors of the system that is completely hidden to the user.

The detailed formalization is available in Combéfis and Pecheur (2009), Combéfis et al. (2011).

Formally, HMI LTS are tuples $\langle S, \mathscr{L}^c, \mathscr{L}^o, s_0, \delta \rangle$ where S is the set of states; $\mathscr{L}^c$ and $\mathscr{L}^o$ are the sets of commands and observations, respectively; s_0 is the initial state; and $\delta : S \times (\mathscr{L}^c \cup \mathscr{L}^o \cup \{\tau\}) \rightarrow 2^S$ is the transition function. Internal actions cannot be distinguished by the user and are thus denoted with the same symbol τ, called the internal action. The set of observable actions comprises commands and

[1] The mental model is commonly referred to as *conceptual model* (Johnson and Henderson 2002) in the literature, that is, an abstraction of the system which outlines what the operator can do with the system and what she needs in order to interact with it.

observations and is denoted $\mathscr{L}^{co} = \mathscr{L}^{c} \cup \mathscr{L}^{o}$. In this chapter, HMI LTS will simply be referred to as LTS.

When a transition exists between states s and s' with action a, that is $\delta(s, a) = s'$, we say that the action a is enabled in state s and we write $s \xrightarrow{a} s'$. A *trace* $\sigma = \langle \sigma_1, \sigma_2, \cdots, \sigma_n \rangle$ is a sequence of observable actions in $\mathscr{L}^{co}$ that can be executed on the system, that is $s_0 \xrightarrow{\sigma_1} s_1 \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_n} s_n$. The set of traces of an LTS $\mathscr{M}$ is denoted $\mathbf{Tr}(\mathscr{M})$. Internal actions can also occur between actions of a trace, which is written $s \overset{\sigma}{\Longrightarrow} s'$ and corresponds to $s \xrightarrow{\tau^* \sigma_1 \tau^* \cdots \tau^* \sigma_n \tau^*} s'$, where τ^* means zero, one or more occurrences of τ. The set of commands that are enabled in a state s, denoted $A^c(s)$, corresponds to actions a such that there exists a state s' with $s \overset{a}{\Longrightarrow} s'$. The set of enabled observations, denoted $A^o(s)$, is defined similarly.

15.3.2 Properties

Given a system model and a mental model, we are interested in the notion of *full controllability* that captures the fact that an operator has enough knowledge about the system in order to control it properly, i.e., at each instant of time, the operator must know exactly the set of commands that the system can receive, as well as the observations that can be made.

Formally, a mental model $\mathscr{M}_U = \langle S_U, \mathscr{L}^c, \mathscr{L}^o, s_{0_U}, \delta_U \rangle$ allows the full-control of a given system $\mathscr{M}_M = \langle S_M, \mathscr{L}^c, \mathscr{L}^o, s_{0_M}, \delta_M \rangle$[2] if and only if:

$$\begin{aligned} &\forall \sigma \in \mathscr{L}^{co*} \text{ such that } s_{0_M} \overset{\sigma}{\Longrightarrow} s_M \text{ and } s_{0_U} \xrightarrow{\sigma} s_U \; : \\ &\quad A^c(s_M) = A^c(s_U) \text{ and } A^o(s_M) \subseteq A^o(s_U) \end{aligned} \tag{15.1}$$

In other words, it means that for every trace σ that can be executed on both the system and the mental model ($s_{0_M} \overset{\sigma}{\Longrightarrow} s_M$ and $s_{0_U} \xrightarrow{\sigma} s_U$), the set of commands ($A^c$) that are enabled on the system model and on the mental model are exactly the same, and the set of observations (A^o) enabled according to the mental model contains at least the observations enabled on the system model.

With this definition, the user must always know all the possible commands, which is a strong requirement. In the work presented here, we use a weaker variant, where a user may not always know all the possible commands but only those which are relevant for interacting with the system.

The full-control property characterizes a conceptual model for a given system, which we refer to as full-control mental model. All the behaviors of the system must be covered by the full-control mental model, and it should allow the user to interact

[2] The subscript M for the system refers to *Machine* and the subscript U for the mental model refers to *User*.

correctly with the system. This means that the user always knows what can be done on the system and what can be observed from it.

The full-control property captures the behavior of a given system that a user should know in order to operate it without errors. Given a system model $\mathscr{M}_M = \langle S_M, \mathscr{L}^c, \mathscr{C}^o, s_{0_M}, \delta_M \rangle$, a trace $\sigma \in \mathscr{L}^{co*}$ can be put into one of three different categories. In other words, the set of traces of $\mathscr{M}_M$ can be partitioned into three sets: *Acc* (Accepted), *Rej* (Rejected), and *Dont* (Don't care).

Let σ be a trace and a an action (command or observation):

1. $\sigma a \in Rej$ if (i) $\sigma \in Rej$ or (ii) $\sigma \in Acc$, a is a command and there exists an execution of σ where a is not enabled in the reached state. This first category highlights the fact that the user must always know exactly the available commands.
2. $\sigma a \in Acc$ if (i) $\sigma \in Acc$ and either a is a command which is enabled for all the states that are reached after the execution of σ or (ii) a is an observation that is enabled in at least one state reached after the execution of σ. This second category contains the behavior of the system of which the user must be aware.
3. In the other cases, $\sigma a \in Dont$. This corresponds to two cases: (i) either $\sigma \in Dont$ or (ii) $\sigma \in Acc$, a is an observation and a is not enabled in any state reachable by σ. That last category reflects the fact that the user may expect an observation that will not necessarily occur in the system.

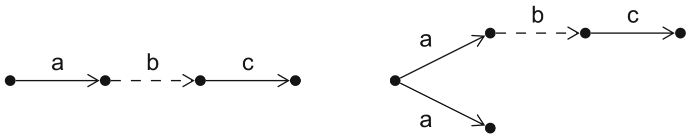
Traces from *Rej are forbidden* which means that they cannot be part of a full-control mental model. Traces from *Acc must be accepted* which means that any full-control mental model must contain those traces. Traces from *Dont may be accepted* which means that they can belong to a full-control mental model for the system, or not.

15.3.3 Analysis

When interacting with a system, a user does not always need to know all the behavior of the system. Most of the time, users are interested in performing some *tasks* that only partially exercise the capabilities of the system. Given the model of a system and a set of user tasks, the system allows the operator to perform all the user tasks if all the behavior covered by the tasks can be executed on the system. Such a system can of course have more behavior, as long as all the tasks are supported.

A user task can be expressed as an LTS $\mathscr{M}_T$. Trace inclusion between the system and the tasks ($\mathbf{Tr}(\mathscr{M}_T) \subseteq \mathbf{Tr}(\mathscr{M}_M)$) can be used to ensure that all traces of the task are supported by the system. However, trace inclusion is not a satisfactory criterion for this type of problem as illustrated by the following example.

In Fig. 15.1, where solid lines correspond to commands and dashed lines to observations, the set of traces of the user task is a subset of the set of traces of the system. But there is a situation where the user can be surprised. After performing an a command, the system can transition to a state where the b observation will never occur, resulting in the user not being able to complete the task.



(a) User task model. (b) System model.

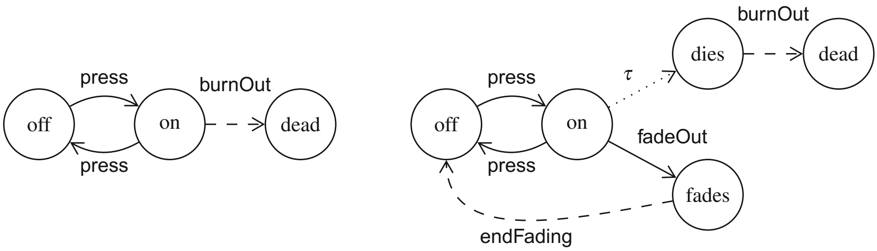
Fig. 15.1 A system (on the *right*) which can make the operator confused in some situations, when he wants to perform his tasks (on the *left*)

The full-control property can be used to achieve a more relevant check between a system model and a user task. Formally, a system model $\mathscr{M}_M = \langle S_M, \mathscr{L}^c, \mathscr{L}^o, s_{0_M}, \delta_M\rangle$ allows the operator to perform the tasks of the user task model $\mathscr{M}_T = \langle S_T, \mathscr{L}^c, \mathscr{L}^o, s_{0_T}, \delta_T\rangle$ if and only if:

$$\forall \sigma \in \mathscr{L}^{co*} \text{ such that } s_{0_T} \xrightarrow{\sigma} s_T \text{ and } s_{0_M} \overset{\sigma}{\Longrightarrow} s_M \;: \\ A^o(s_T) = A^o(s_M) \text{ and } A^c(s_T) \subseteq A^c(s_M) \qquad (15.2)$$

For a system to support a user task, the following must hold. At any point during the execution of the task, if the user needs to issue a command, then that command should be included in the commands available in the system at that point. Moreover, at any point during the execution of the task, the task should be prepared to receive exactly those observations available in the system at that point. Therefore, the full-control check can be applied between the system model $\mathscr{M}_M$ and the user task model $\mathscr{M}_T$, interchanging commands and observations.

Figure 15.2 shows an example of a lamp illustrating that relation. The task model indicates that the user should be able to switch a lamp on and off with a press command. If the user observes a burnOut signal, then the lamp is dead and nothing else can be done. The proposed system model allows full-control of the task model (inverting the roles of commands and observations). There is an additional behavior which



(a) Task model. (b) System model.

Fig. 15.2 An example of a set of user task $\mathscr{M}_T$ for a simple lamp (on the *left*) and a system model $\mathscr{M}_M$ which allows full-control of it (on the *right*)

allows the lamp to be dimmed down all the way to off if the user presses the fadeOut command while in the on state. That additional behavior is not a problem since, according to the task model, it will never be executed.

15.3.4 Example

The Therac-25 (Leveson and Turner 1993) is a medical system which was subject to an accident due to an operator manipulation error which took place during the interaction with the machine. The machine has the ability to treat patients by administering X-ray or electron beams. For the first treatment, a spreader has to be put in place so that the patient does not receive too much radiation. The accident occurred when patients were administered X-rays while the spreader was not in place.

The formal model described in Bolton et al. (2008) was represented in the JPF framework and had 110 states and 312 transitions, among which there are 194 commands, 66 observations, and 52 internal actions. The set of commands is {selectX, selectE, enter, fire, up}, and there is one observation corresponding to a timeout {8 second}. The model is illustrated as a statechart on Fig. 15.3.

The result of the analysis of the Therac-25 system is that it is well-behaved (i.e., it is full-control deterministic) and that it cannot be reduced. The minimal full-control model is thus exactly the same as the system model, without the τ transitions. The potential error with that system cannot be captured with the model as it has been described.

In fact, the error is actually due to *mode confusion*: the operator believes that the system was in the electron beam mode while it is in fact in the X-ray mode. That kind of error can be found with our framework, by enriching the system model with mode information. Loops are added on all the states where a mode is active with either X-

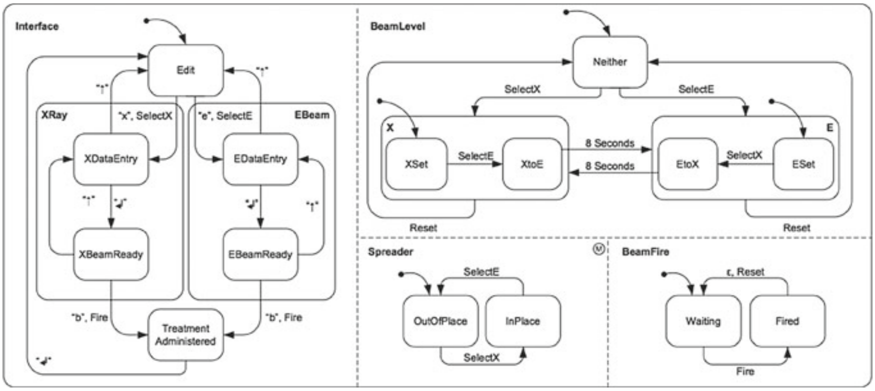


Fig. 15.3 Statechart model of the Therac-25 medical system (Leveson and Turner 1993). The enter command is represented with ↲ and the up command is represented with ↑

ray or E-beam, according to the mode the machine is in. These new labels are treated as commands, reflecting the fact that the operator must know exactly which mode the machine is in.

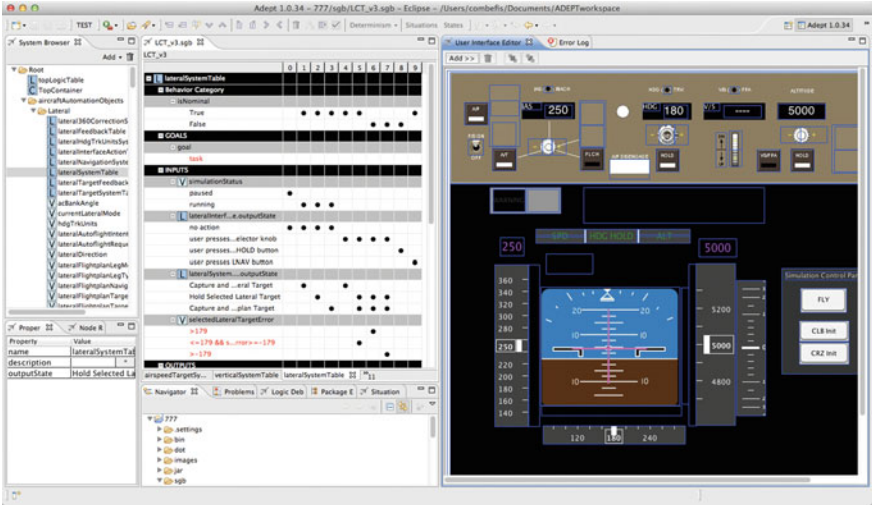
Analyzing that modified system leads to an error, because the system is no longer full-control deterministic. The counterexample produced by the framework is as follows: ⟨selectX, enter, fire, X-ray⟩. That trace corresponds to a trace that must be accepted and must be forbidden at the same time. Indeed, after selecting X-ray beam (selectX), validating it (enter) and administering the treatment (fire), the X-ray command may or may not be available depending on the execution followed in the system. This means that the system may end up either in the X-ray or in the E-beam mode, non-deterministically and with no observable difference. That behavior is due to an internal transition that occurs when the treatment has been administered, which represents the fact that the system is reset to its initial state. The controllability issue indicates that there should be an observation informing the user when the system is reset. Adding a reset observation makes the system full-controllable and a minimal full-control mental model for it can be generated, with 24 states.

There is another issue with that system. If the operator is not aware of the 8-second timer (or does not track the countdown), the issue described in Leveson and Turner (1993), Bolton et al. (2008) can also be found. It suffices to turn the observation 8 second into an internal transition τ and to make the system being reset when the operator presses enter after the treatment has been administered. The last actions of the returned counterexample (the most relevant part) is as follows: ⟨…, selectE, up, E-beam, selectX, E-beam⟩. That corresponds to the user selecting the electron beam mode, then changing his mind by pushing on the up button and selecting the X-ray mode. After that, the system may either be in E-beam or X-ray mode.

15.3.5 Analysis of ADEPT Models

Even though HMI LTSs are an ideal abstraction for the specification and analysis of the HMI properties described above, it is hard to model large, complex HMI systems directly as LTSs. For this reason, we have worked on connecting our analysis algorithms to prototyping tools that are more familiar and intuitive for domain experts. In particular, we have worked closely with the developers of the ADEPT tool.

ADEPT (*Automatic Design and Evaluation Prototyping Toolset*) (Feary 2010) is a Java-based tool developed at NASA Ames, which supports designers in the early prototyping phases of the design of automation interfaces. The tool also offers a set of basic analyses that can be performed on the model under development. An ADEPT model is composed of two elements: a set of logic tables, coupled with an interactive user interface (UI). The logic tables describe the dynamics of the system as state changes in reaction to user actions or to environmental events. For example, Fig. 15.4 shows a screenshot of the autopilot model opened in ADEPT. The left part of the window shows one of the logic tables and the right part shows the user interface.



Adept 1.0.34 – 777/sgb/LCT_v3.sgb – Eclipse – /Users/combefis/Documents/ADEPTworkspace

LCT_v3

	0	1	2	3	4	5	6	7	8	9
lateralSystemTable										
Behavior Category										
isNominal										
True		●	●	●	●	●				●
False							●	●	●	
GOALS										
goal										
task										
INPUTS										
simulationStatus										
paused	●									
running		●	●	●						
lateralInterf...e.outputState										
no action	●	●	●	●						
user presses...elector knob					●	●	●	●		
user presses...HOLD button									●	
user presses LNAV button										●
lateralSystem....outputState										
Capture and ...eral Target		●			●					
Hold Selected Lateral Target			●			●	●	●		
Capture and ...plan Target				●		●	●	●		
selectedLateralTargetError										
>179							●			
<=179 && s...rror>=-179						●				
>-179								●		
OUTPUTS										

Fig. 15.4 The autopilot model opened in ADEPT, with one logic table in the *left* part of the window and the user interface on the *right* part

The UI is composed of a set of components that are encoded as Java objects representing graphical widgets. The logic tables can refer to the elements of the UI and to the other components through their Java instance variables, and interact with them through their methods, using Java syntax. In particular, UI events are seen as Boolean variables that are set to true when the event occurs.

Behind the scene, an ADEPT model is compiled into a Java program that can be executed in order to directly try the encoded behavior with the user interface. That tool is meant to be used as a rapid prototyping tool. The models not only can then be tested and simulated by the designers, but can also be analyzed by systematic and rigorous techniques. Possible analyses include validity checks on the structure of logic tables, for example. We have extended the analysis capabilities of ADEPT with our framework, which requires the translation of ADEPT tables into the models that our framework can handle. We describe our work with ADEPT through an autopilot example.

Figure 15.5 shows one of the logic tables of the autopilot model. The table example illustrates the way it can interact with elements of the UI. Each light gray line of the table corresponds to a variable of the system. The variables can be related to a component of the UI (such as pfdAirspeedTargetTape.currentValue), or they can be state variables of the model (such as indicatedAirSpeed) or they can relate to the internal logic of the system (airspeedSystemTable.outputState). The latter kind of variables can be seen as a description of the mode of a particular component of the system (the airspeed part in this example). For example, the two first lines of the output part of the logic table example mean that the value of the currentValue field of the pfdAirspeedTargetTape component of the UI is updated with the value of the indicatedAirspeed

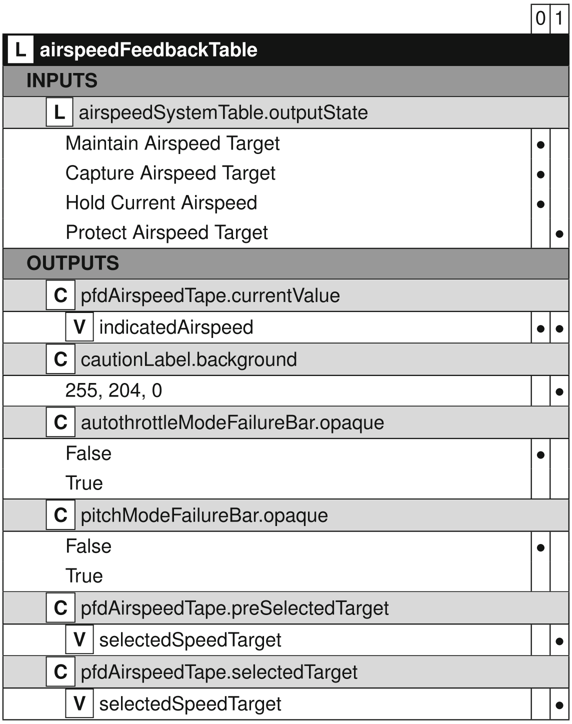


Fig. 15.5 An example of a logic table: the airspeed feedback table of the autopilot model contains the logic related to the update of the UI for the airspeed part

	0	1
L airspeedFeedbackTable		
INPUTS		
L airspeedSystemTable.outputState		
Maintain Airspeed Target	●	
Capture Airspeed Target	●	
Hold Current Airspeed	●	
Protect Airspeed Target		●
OUTPUTS		
C pfdAirspeedTape.currentValue		
V indicatedAirspeed	●	●
C cautionLabel.background		
255, 204, 0		●
C autothrottleModeFailureBar.opaque		
False	●	
True		
C pitchModeFailureBar.opaque		
False	●	
True		
C pfdAirspeedTape.preSelectedTarget		
V selectedSpeedTarget		●
C pfdAirspeedTape.selectedTarget		
V selectedSpeedTarget		●

state variable. Moreover, each column of an ADEPT table corresponds to a transition scenario. From any state of the system that satisfies the condition described by the input part of the table, the system can move to the state of the system that results in applying the update instructions described by the output part of the table.

The ADEPT autopilot partially models the behavior of the autopilot of a Boeing 777 aircraft. The full autopilot ADEPT model has a total of 38 logic tables. Three major groups of tables can be identified in the model, namely one for the lateral aspect, one for the vertical aspect, and finally, one for the airspeed aspect. For each of these aspects, the logic tables are further partitioned into three groups: the action tables, the system tables, and the feedback tables, successively executed in that order. Action tables determine actions from UI events, system tables update the state of the system according to the performed action, and feedback tables reflect the state of the system to UI elements.

For a full account of our autopilot case study, we refer the interested reader to (Combéfis 2013). Here, we report briefly on the extensions that we applied to our formalisms as well as our experiences and some observations with analyzing autopilot system tables. First of all, given the fact that ADEPT models are state-based and in order to be able to easily support the automatic translation of ADEPT models into HMI LTS for the application of our analysis algorithms, we extended our HMI models with state information, as described in Combéfis et al. (2016). These models are named HMI state-Valued System models, or HVS. Our techniques cannot scale to the full size of such a large and complex model. Therefore, our analyses were performed on parts of the autopilot model, each analysis considering subsets of the system tables. As is typical with formal techniques, we additionally had to abstract the infinite data domains involved in the models.

Using the outputState as a mode indicator, we performed mode confusion analysis and detected a potential mode confusion on the airspeedSystemTable. This was identified during the minimal model generation phase, where the generation algorithm produced an error trace witnessing the fact that the system model was not fc-deterministic. By analyzing the error trace manually, we localized the erroneous behavior in the involved ADEPT tables. One of the models that was analyzed was an HVS with 7680 states and 66242 transitions, among which 57545 are labeled with commands and 8697 are internal τ-transitions. The obtained minimal mental model has 25 states and 180 transitions.

Our experiences described in this section lead us to the conclusion that in developing modeling and analysis techniques that can capture multiple aspects of the NAS, we would need to move toward formalisms that enable the intuitive and scalable modeling of multiple interacting agents (human and automation), with support for a variety of analyses, starting from more scalable simulations and toward not only more sophisticated, but also more resource-consuming exhaustive techniques such as the ones we presented. These observations lead to the work that is described in the next section.

15.4 Coordinating HMIs as Multi-agent Systems

This section describes the modeling analysis of multiple interacting HMIs specified in the Brahms modeling language (Clancey et al. 1998; Sierhuis 2001). The input to our framework is a Brahms model along with a Java implementation of its semantics. Brahms is a multi-agent simulation system in which people, tools, facilities, vehicles, and geography are modeled explicitly. The air transportation system of the NAS is modeled as a collection of distributed, interactive subsystems such as airports, air traffic control towers and personnel, aircraft, automated flight systems and air traffic tools, instruments, and flight crew. Each subsystem, whether a person or a tool such as the radar, is modeled independently with properties and contextual behaviors. Brahms facilitates modeling various configurable realistic scenarios that allows the analysis of the airspace in various conditions and reassignment of roles and responsibilities among human and automation.

15.4.1 The Brahms Language

Brahms is a full-fledged multi-agent, rule-based, activity programming language. It is based on a theory of work practice and situated cognition (Clancey et al. 1998; Sierhuis 2001). The Brahms language allows for the representation of situated activities of agents in a geographical model of the world. Situated activities are actions performed by the agent in some physical and social context for a specified period of time. The execution of actions is constrained (a) locally: by the reasoning capabilities of an agent and (b) globally by the agents beliefs of the external world, such as where the agent is located, the state of the world at that location and elsewhere, located artifacts, activities of other agents, and communication with other agents or artifacts. The objective of Brahms is to represent the interaction between people, off-task behaviors, multitasking, interrupted and resumed activities, informal interactions, and knowledge, while being located in some environment representative of the real world.

At each clock tick, the Brahms simulation engine inspects the model to update the state of the world, which includes all of the agents and all of the objects in the simulated world. Agents and objects have states (factual properties) and may have capabilities to model the world (e.g., radar display is modeled as beliefs, which are representations of the state of the aircraft). Agents and objects communicate with each other; the communication can represent verbal speech, reading, writing, etc. and may involve devices such as telephones, radios, and displays. Agents and objects may act to change their own state, beliefs, or other facts about the world.

15.4.2 MAS Formal Analysis

We use model checking-based techniques to systematically explore the various behaviors in Brahms scenarios, i.e., in our case study collision scenarios of the Überlingen model configuration. In Hunter et al. (2013) and Rungta et al. (2013), we present an extensible verification framework that takes as input a multi-agent system model and its semantics as input to some state space search engine (or a model checker). The search engine generates all possible behaviors of the model with respect to its semantics. The generated behaviors of the model are then encoded as a reachability graph $G = (N, E)$, where N is a set of nodes and E is a set of edges. This graph is automatically generated by the search engine. Each node $n \in N$ is labeled with the belief/facts values of the agents and objects. In Hunter et al. (2013), we generate the reachability graph using the Java PathFinder byte-code analysis framework. An edge between the nodes represents the updates to beliefs/facts and is also labeled with probabilities. The reachable states generated by JPF are mapped to the nodes in a reachability graph. This reachability graph is essentially an LTS.

The verification of safety properties and other reachability properties is performed on the fly as new states and transitions are generated. JPF is an explicit-state analysis engine that stores the generated model in memory. Capturing the state of all the agents and objects in Brahms including their workframes and thoughtframes can lead to large memory requirements. Additionally, for large systems, it is often intractable to generate and capture even just the intermediate representation in memory.

To overcome these limitations, we adopt a stateless model checking approach. Stateless model checking explores all possible behaviors of the program or model without storing the explored states in a visited set. The program or model is executed by a scheduler that tracks all the points of non-determinism in the program. The scheduler systematically explores all possible execution paths of the program obtained by the non-deterministic choices. Stateless model checking is particularly suited for exploring the state space of large models. In this work, we instrument the Brahms simulator to perform stateless model checking. The instrumented code within the Brahms engine generates all possible paths (each with different combinations of activity durations) in depth-first ordering. Stateless model checkers like VeriSoft (Godefroid 1997) do not in general store paths; however, in order to perform further analysis of the behaviors space, the Brahms stateless model checker can store all the generated paths in a database.

15.4.2.1 Non-determinism in Brahms

There are two main points of non-determinism in Brahms models. The first point of non-determinism is due to durations of primitive activities. The different primitive activities in Brahms have a duration in seconds associated with them. The duration of the primitive activity can either be fixed or can vary based on certain attributes of the primitive activities. When the random attribute of a primitive activity is set to true, the simulator randomly selects the primitive activity duration between the min and max durations specified for the activity. The second point of non-determinism arises from probabilistic updates to facts and beliefs of agents and objects. Updates to facts and beliefs are made using conclude statements in Brahms. Here is an example of a conclude statement:

$$\mathit{conclude}((\mathit{Pilot}.\mathit{checkStall} = \mathit{false}), \mathit{bc} : 70, \mathit{fc} : 70)$$

This states that the belief and fact, *checkStall*, in the *Pilot* agent will be updated to false with a probability of 70%. Here, *bc* represents belief certainty while *fc* represents fact certainty.

In the Überlingen model, currently there are only deterministic updates to facts or beliefs. The updates to facts and beliefs are asserted with a 100% probability. Nevertheless, there is a large degree of non-determinism due to variations in activity durations. The difference in minimum and maximum duration ranges from 2 s to a few 100 s. This can potentially lead to a large number of timing differences between the various events.

15.4.2.2 Behavior Space

The scheduler within the stateless Brahms model checker generates all possible paths through the different points of non-determinism in the Brahms model. Note that in describing the output of the Brahms stateless model checker, we use the terms path and trace interchangeably. Intuitively, a path (or trace) generated by the Brahms stateless model checker is equivalent to a single simulation run. More formally, a path or trace is a sequence of events executed by the simulator $< e0, e1, e2, ..., ei >$. Each event in the trace is a tuple, $< a, t, (u, val) >$ where a is the actor id, t is the Brahms clock time, and u is the fact or belief updated to the value val. For each trace, we generate a sequence of nodes in the intermediate representation $n_{init}, n_0, n_1, n_2, ..., n_i$. The initial node in the sequence, n_{init} is labeled with the initial values of belief/facts values for the various agents and objects. The event $e_0 := < a_0, t_0, (u_0, val_0) >$ is applied to the initial node n_{init} where the value assigned to u_0 is updated to val_0. Each event is applied in sequence to a node in the intermediate representation to generate $n_{init}, n_0, n_1, n_2, ..., n_i$.

15.4.3 Case Study: The Überlingen Collision

The Überlingen accident, (Überlingen 2004), involving the (automated) Traffic Collision Avoidance System (TCAS), is a good example to illustrate problems arising from multiple human operators interacting with multiple automated systems. TCAS has the ability to reconfigure the pilot and air traffic control center (ATCC) relationship, taking authority from the air traffic control officer (ATCO) and instructing the pilot.

TCAS is an onboard aircraft system that uses radar transponder signals to operate independently of ground-based equipment to provide advice to the pilot about conflicting aircraft that are equipped with the same transponder/TCAS equipment. The history of TCAS dates at least to the late 1950s. Motivated by a number of midair collisions over three decades, the United States Federal Aviation Administration (FAA) initiated the TCAS program in 1981. The system in use over Überlingen in 2002 was TCAS II v.7, which had been installed by US carriers since 1994: TCAS II issues the following types of aural annunciations:

- Traffic advisory (TA)
- Resolution advisory (RA)
- Clear of conflict

When a TA is issued, pilots are instructed to initiate a visual search, if possible, for the traffic causing the TA. In the cases when the traffic can be visually acquired, pilots are instructed to maintain visual separation from the traffic. When an RA is issued, pilots are expected to respond immediately to the RA unless doing so would jeopardize the safe operation of the flight. The separation timing, called TAU, provides the TA alert at about 48 s and the RA at 35 s prior to a predicted collision.

On July 1 2002, a midair collision between a Tupolev Tu-154M passenger jet traveling from Moscow to Barcelona, and a Boeing 757-23APF DHL cargo jet manned by two pilots, traveling from Bergamo to Brussels, occurred at 23:35 UTC over the town of Überlingen in southern Germany. The two flights were on a collision course. TCAS issued first a Traffic Advisory (TA) and then a Resolution Advisory (RA) for each plane. Just before TCAS issued an RA requesting that the Tupolev climb, the air traffic controller in charge of the sector issued a command to descend; the crew obeyed this command. Since TCAS had issued a Resolution Advisory to the Boeing crew to descend that they immediately followed, both planes were descending when they collided.

The decision of the Tupolev crew to follow the ATC's instructions rather than TCAS was the immediate cause of the accident. The regulations for the use of TCAS state that in the case of conflicting instructions from TCAS and ATCO, the pilot should follow the TCAS instructions. In this case study, the conflict arose because the loss of separation between the two planes was not detected or corrected by the ATCO. The loss of separation between airplanes are frequent occurrences; it is part of the normal work of air traffic control to detect and correct them accordingly.

There were a set of complex systemic problems at the Zurich air traffic control station that caused the ATCO to miss detecting the loss of separation between the two planes. Although two controllers were supposed to be on duty, one of the two was resting in the lounge: a common and accepted practice during the lower workload portion of night shift. On this particular evening, a scheduled maintenance procedure was being carried out on the main radar system, which meant that the controller had to use a less capable air traffic tracking system. The maintenance work also disconnected the phone system, which made it impossible for other air traffic control centers in the area to alert the Zurich controller to the problem. Finally, the controllers workload was increased by a late arriving plane. An A320 that was landing in Friedrichshafen required the ATCO's attention, who then failed to notice the potential separation infringement of the two planes.

The Überlingen collision proves that methods used for certifying TCAS II v7.0 did not adequately consider human–automation interactions. In particular, the certification method treated TCAS as if it were flight system automation, that is, a system that automatically controls the flight of the aircraft. Instead, TCAS is a system that tells pilot how to maneuver the aircraft, an instruction that implicitly removes and/or overrides the ATCs authority. Worldwide deployment of TCAS II v7.1 was still in process in 2012, a decade after the Überlingen collision.

15.4.3.1 Brahms' Model

In a Brahms model, the entire work system is modeled, including agents, groups to which they belong, facilities (buildings, rooms, offices, spaces in vehicles), tools (e.g., radio, radar display/workstation, telephone, vehicles), representational objects (e.g., a phone book, a control strip), and automated subsystems (e.g., TCAS), all located in an abstracted geography represented as areas and paths. Thus, the notion of human–system interaction in Brahms terms is more precisely an interaction between an agent and a subsystem in the model; both are behaving within the work system. A workframe in Brahms can model the interaction between an agent's beliefs, perception, and action in a dynamic environment, for example, these characteristics are leveraged when modeling how a pilot deploys the aircraft landing gear. A pilot uses the on-board landing control and then confirms that the landing gears are deployed while monitoring the aircraft trajectory on the Primary Flight Display. This is modeled in Brahms as follows: a pilot (e.g., the DHL pilot) is a member of the PilotGroup, which has a composite activity for managing aircraft energy configuration. A specific instance of a conceptual class is called a conceptual object. A particular flight (e.g., DHX611, a conceptual object) is operated by a particular airline and consists of a particular crew (a group) of pilots (agents) who file a particular flight plan document (an object), and so on. Each instance of an agent and object have possible actions defined by workframes where each workframe contains a set of activities that are ordered and often prioritized. Certain workframes are inherited from their group (for agents) or class (for objects). The set of possible actions are modeled at a general level and all members of a group/class have similar capabilities (represented as activities, workframes, and thoughtframes); however, at any time during the simulation, agent and object behaviors, beliefs, and facts about them will vary depending on their initial beliefs/facts and the environment with which they are interacting. The model incorporates organizational and regulatory aspects implicitly, manifest by how work practices relate roles, tools, and facilities.

A Brahms simulation model configuration consists of the modeled geography, agents, and objects, as well as their initial facts and beliefs of agents and objects. The different configurations allow us to perform a what-if analysis on the model. The time of departure for a flight might be an initial fact in a Brahms model. One can modify the model to assign a different time of departure for a flight in each simulation run. Another example of configurable initial facts may include work schedules for air traffic controllers. In one configuration of the work schedules, an air traffic controller may be working alone in the ATCC, while in another configuration, two controllers would be present in the ATCC. Initial beliefs of an agent might be broad preferences affecting behavior (e.g., TCAS should overrule the ATC), thus initial beliefs can be used as switches to easily specify alternative configurations of interest. Alternative configurations are conventionally called scenarios. Thus for example, a scenario might be a variation of the Überlingen collision in which two aircraft have flight times that put them on an intersecting path over Überlingen; the only other flight is a late arriving flight for Friedrichshafen and maintenance degrades the radar, but the telephones are in working order.

In general, a model is designed with sufficient flexibility to allow investigating scenarios of interest. The set of causal factors of interest (e.g., use of control strips when approving aircraft altitude changes, availability of telephones) constitute states of the world and behaviors that can be configured through initial facts and beliefs. The initial settings define a space of scenarios. Using Brahms to evaluate designs within this space, while using formal methods to help modelers understand its boundaries so they can refine the model to explore alternative scenarios constitutes the main research objective of this work.

The simulation engine determines the state of a modeled object (e.g., aircraft). It determines the state of its facts and beliefs. Some objects are not physical things in the world, but rather conceptual entities, called conceptual classes in the Brahms language. These represent processes, a set of people, physical objects, and locations (e.g., flights), and institutional systems (e.g., airlines) that people know about and refer to when organizing their work activities.

15.4.3.2 High-Level Structure of Model

The systems that are mentioned in the accident report and play a role in accident have been modeled in the Brahms Überlingen model. These include the pilots in each aircraft, two ATCOs at Zurich, the relevant airspace and airports, all the aircraft relevant to the accident, on-board automation, and other flight systems. The following key subsystems and conditions are modeled in the Brahms Überlingen model:

1. Interactions among Pilot, Flight Systems, and Aircraft for climb and cruise with European geography for one plane, the DHL flight plan.
2. BTC flight, flight plan (two versions: on-time and delayed with collision) and geography: this is independent of ATCO actions to confirm that simulation reproduces collision with flight paths actually flown.
3. Radar Systems and Displays with ATCOs, located in Control Centers, monitoring when flights are entering and exiting each European flight sector in flight plans.
4. Handover interactions between Pilot and ATCOs for each flight phase.
5. Two ATCOs in Zurich, Radar Planner (RP), and ARFA Radar Executive (RE), assigned to two workstations (RE has nothing to do under these conditions).
6. Add TCAS with capability to detect separation violations, generate Traffic Advisory (TA) and Resolution Advisory (RA). DHL and BTC are delayed (on collision course, which tests TCAS)
7. Pilots follow TCAS instructions
8. ATCO may intervene prior to alert depending on when ATCO notices conflict in Radar Displays since ATCO is busy communicating with other flights, moving between workstations, and trying to contact Friedrichshafen control tower on the phone.
9. AEF flight and flight plan so Zurich ARFA RE performs landing handoff to Friedrichshafen controller.
10. Third plane, the AEF flight, arrives late, requiring ATCO communications and handoff to Friedrichshafen: (a) Handled by ATCO in Zurich at right workstation (ARFA sector) and not left East and South sector workstation. (b) Phone communications for handovers, (c) Methods used by ATCO when phone contact does not work:

   a. Ask Controller Assistant (CA) to get another number (pass-nr); requires about 3 min for CA to return
   b. After pass-nr fails, discuss with CA other options about 30 sec
   c. When not busy handling other flights, try pass-nr again.
   d. When plane is at Top-Of-Descent waypoint, as specified in STAR, for landing at airport, within N nm of airport, method of last resort is to call pilots on radio and ask them to contact the tower directly

11. STCA added to ATCO workstations (modeling normal and fallback mode without optical alert). The ATCO responds to alert by advising Pilot to change flight level based on next flight segment of flight plan.
12. Reduce to one Zurich ATCO which triggers the sequence of variations from the nominal situation; now Zurich ATCO must operate flights from two workstations.

15.4.3.3 Analysis Results

The question that the analysis tries to answer, using both simulation and verification, is why under certain conditions, a collision is averted, while in others it is not? In the analysis, we try to gauge how the temporal sensitivity and variability of the interactions among ATCO, TCAS, and the pilots impacts the potential loss of separation and collision of the planes. Concretely, the questions that we ask during the analysis are as follows:

- Given that the arrival of the AEF flight is disrupting the ATCOs monitoring of the larger airspace (e.g., if it arrives sufficiently late, no collision occurs), what is the period (relative to the BTC and DHL flights paths) when AEF's arrival can cause collision?
- During this period, does a collision always occur or are there variations of how the AEF handoff occurs, such that sometimes the separation infringement is averted?
- Is there evidence that high-priority activities such as monitoring the sector are repeatedly interrupted or deferred, implying the ATCO is unable to cope with the workload?

The Brahms Überlingen Model defines a space of work systems (e.g., is STCA optical functioning? are there two ATCOs?) and events (e.g., the aircraft and flights). Every model configuration, which involves configuring initial facts, beliefs, and agent/object relations, constitutes a scenario that can be simulated and will itself produce many different outcomes (chronology of events), because of non-deterministic timings of agent and object behaviors. The model was developed and tested with a variety of scenarios (e.g., varying additional flights in the sector; all subsystems are working properly). The Überlingen accident is of special interest, in which systems are configured as they were at the time of the accident and the DHL and BTC planes are on intersecting routes.

The key events that occur during simulation are logged chronologically in a file that constitutes a readable trace of the interactions among the ATCO, pilots, and automated systems. The log includes information about the following:

(a) ATCO–pilot interaction regarding a route change, including flight level and climb/descend instruction,
(b) Separation violation events detected by TCAS, including TAU value,
(c) Closest aircraft and separation detected by ATCO when monitoring radar,
(d) STCA optical or aural alerts, including separation detected,
(e) Agent movements (e.g., ATCO shifting between workstations),
(f) Aircraft movements, including departure, entering and exiting sectors, waypoint arrival, landing, collision, airspeeds, and vertical,
(g) Aircraft control changes (e.g., autopilot disengaged),
(h) Radio calls, including communicated beliefs, and
(i) Phone calls that fail to complete.

The outcome of ten simulation runs of Brahms Überlingen model configured for the collision scenario are shown in Table 15.1. In simulation runs 1, 2, and 3, the ATCO intervenes before TCAS TA, but planes have not separated sufficiently, TCAS will take BTCs descent into account, advising DHL to climb. In the simulation runs 4, 5, 7, 8, and 9, the ATCO intervenes between TA and RA. In these runs, whether the planes collide depends on timing. As shown in Table 15.1 two of the five runs results in a collision. Note that in our model, a collision is defined as occurring when the vertical separation between the planes is less than a 100 feet. Finally, in the simulation runs 6 and 10, the ATCO intervenes about 10 s after the TCAS RA, which the BTC pilots ignore (or might be imagined as discussing for a long time); therefore, BTC continues flying level while DHL descends and they miss each other, separated by more than 600 ft at the crossing point. In other runs, we have also observed that ATCO intervenes so late, he actually takes the pilots' report about TCAS RA instructions into account.

When ATCO intervenes in the period between the TA and RA in runs 4, 5, 7, 8, and 9, a collision is possible, like what happened at Überlingen: ATCO has to intervene before the TA advising BTC to descend so that BTC can respond before TCAS advises DHL to climb. In runs 4 and 7, collision is narrowly averted because BTC begins to descend 4 or 5 s after the TCAS RA, which is sufficient for a narrow miss (just over 100 feet). In run 9, the BTC descent begins 5 s before the RA, hence the aircraft miss by more than 200 feet). Runs 5 and 8 lead to a collision because the TCAS RA and BTC AP disengage occur at the same time, like what happened at Überlingen.

Because the model uses the Überlingen descent tables to control the BTC and DHL aircraft during the emergency descent, simulation matches the paths of the aircraft at Überlingen guaranteeing a collision (within defined range of error). In both cases, TCAS did not instruct DHL to climb because BTC was above DHL at that time and of course had not begun its descent.

When ATCO intervenes after the RA, the BTC pilots in the simulations ignore the RA advice and continue level flight, which itself averts the collision, even though

Table 15.1 Outcomes of ten simulation runs of Überlingen scenario. Bold indicates greatest potential for collision (ATCO intervenes between TA and RA; both aircraft descending)

Run	Collide?	Explanation	ATCO-BTC	TCAS RA-DHL	ATCO relative TA/RA
1	No	TCAS detects BTC plane descending due to ATCO; so advises DHL to Climb	Descend	Climb	Before
2	No	TCAS detects BTC plane descending due to ATCO; so advises DHL to Climb	Descend	Climb	Before
3	No	TCAS detects BTC plane descending due to ATCO; so advises DHL to Climb. AEF flight arrives very late after TCAS TA	Descend	Climb	Before
4	No	DHL TCAS Descend; BTC above. Planes crossed >100 ft vertical separation	Descend	Descent	During
5	YES	DHL TCAS Descend; BTC above. BTC AP turned off at DHL RA. Planes crossed <20 ft vertical separation	Descend	Descent	During
6	No	DHL TCAS Descend; BTC above. ATCO later than RA, so BTC level. Planes crossed >600 ft vertical separation	Descend	Descent	After
7	No	DHL TCAS Descend; BTC above. DHL turned off 2 s before BTC. Planes crossed >100 ft vertical separation	Descend	Descent	During
8	YES	DHL TCAS Descend; BTC above. Planes crossed <50 ft vertical separation	Descend	Descent	During
9	No	DHL TCAS Descend; BTC above. Planes crossed >200 ft vertical separation	Descend	Descent	During
10	No	DHL TCAS Descend; BTC above. ATCO later than RA, so BTC level. Planes crossed >600 ft vertical separation	Descend	Descent	After

ATCO advises BTC to descend (which implies ignoring that DHL is below them). We of course do not know what the BTC pilots would have done if ATCO had not intervened. With more than one pilot interpreting TCAS correctly, it appears possible the BTC would have climbed.

The final AEF handoff (directing the pilots to contact the tower) always occurs in the simulation after the TCAS RA; at Überlingen, it occurred prior to the TA. This discrepancy raises many questions about what variability is desirable. In the verification of the system, we were able to find certain cases where the final AEF handoff occurs before the TCAS TA and the planes collide.

15.5 Related Work

Several research groups have worked on modeling and analysis of HMI systems and properties and have taken a variety of approaches to the problem. There has been been work on fomalising user models such as CCT and PUMS since the 1980s, but most of these modeling approaches do not focus on the verification aspect Young et al. (1989), Butterworth and Blandford (1997), Bovair et al. (1990).

Campos and Harrison (2008, 2011) propose a framework for analyzing HMI using model checking. They define a set of generic usability properties (Campos and Harrison 2008), such as the possibility to undo an action. These properties can be expressed in a modal logic called MAL and checked with a model checker on the system. This approach targets specific and precise usability properties, whereas our approaches revolve around the higher-level "full-control" characteristic that we defined for an HMI system and as such is complementary to their analysis.

Thimbleby and Gow (2007), Thimbleby (2010) use graphs to represent models. They study usability properties of the system by analyzing structural properties of graphs such as the maximum degree and the value of centrality measures. In their approach, there is no distinction among actions and there is little focus on the dynamic aspects of the interaction.

Curzon et al. (2007) use a framework based on defining systems with modal logic. Properties of the model are checked using a theorem prover. Similarly to Campos et al., properties of interest are more targeted to a specific usability property while our approach is more generic.

Navarre et al. (2001), Bastide et al. (2003) also developed a framework to analyze interactive systems. Their focus is on the combination of user task models and system models, which we have also explored in the context of our work (Combéfis 2009).

Bolton et al. (2008, 2011), Bolton and Bass (2010) developed a framework used to help predicting human errors and system failures. Models of the system are analyzed against erroneous human behavior models. The analysis is based on task-analytic models and taxonomies of erroneous human behavior. All those models are merged into one model which is then analyzed by a model checker to prove that some safety properties are satisfied. Modeling human error is something that can be incorporated into our Brahms models and taken into account in our analysis.

Bredereke and Lankenau (2002, 2005) formalized mode confusions and developed a framework to reduce them. The formalization is based on a specification/implementation refinement relation. Their work is targeted on mode confusion while the work presented here is targeted to more general controllability issues.

Model-based testing has been used to analyze systems modeled as Input-Output Labeled Transition Systems (IOTS) (Tretmans 2008). The IO conformance relation (IOCO) is defined to describe the relationship between implementations and specifications. The IOCO relation states that the outputs produced by an implementation must, at any point, be a subset of the corresponding outputs in the specification. This is triggered by the fact that IOCO is used in the context of testing implementations. Outputs are similar to observations in our context. The full-control property defined in our work needs to consider commands (inputs) in addition to observations.

The works discussed above all focus on analysis of HMI properties but none of them have looked at the issue of facilitating the modeling and analysis of complex multi-agent systems. In a way, this is an orthogonal concern, and the techniques discussed above can be incorporated with a framework like Brahms.

There has been a large body of work in the verification safety-critical systems in the domain of civil aviation in the US as well as in Europe. The DO-178B titled *Software considerations in airborne systems and equipment certification* is the official guideline for certifying avionics software. Several model checking and formal verification techniques have been employed to verify avionic *software* in Miller et al. (2010), Ait Ameur et al. (2010) in accorrdance with the DO-178B. Recent work describes how changes in aircraft systems and in the air traffic system pose new challenges for certification, due to the increased interaction and integration (Rushby 2011). The certification is defined for the deployed software.

For the verification of model-based development, in Miller et al. (2010), the authors present a framework that supports multiple input formalisms to *model* avionic software: these include MATLAB Simulink/Stateflow and SCADE. These formalisms are then translated into an intermediate representation using Luster, a standard modeling language employed to model reactive systems with applications in avionics. Finally, Lustre models are translated to the input language of various model checkers, including NuSMV, PVS, and SAL. These models, however, do not account for the behavior of the human operators.

The work of Yasmeen and Gunter (2011) deals with the verification of the behavior of human operators to check the robustness of mixed systems. In this approach, the authors employ concurrent game structures as the modeling language and translate the verification problem to a model checking instance using SPIN. Our approach is different in that we do not perform syntactic translations, and we reason explicitly about probabilities and beliefs of the agents in the model.

15.6 Conclusions and Future Work

NASA has engaged in research to develop new analytical techniques that can model human–machine interactions and represent the real complexity of the NAS. These techniques represent and analyze scenarios with single users interacting with a single piece of automation as well as multiple humans interacting with multiple automated systems. In this chapter, we described our efforts in developing HMI-specific formal analysis algorithms, connecting them to models that domain experts are familiar with, and finally moving toward multi-agent modeling formalisms that are able to capture in a more scalable fashion the intricate interactions between agents in the NAS. We illustrated our work with a variety of relevant case studies.

In the future, we want to build a fast time simulation and analysis framework based on these technologies to evaluate design concepts for Air Traffic Management operations. Our ultimate goal is to provide algorithms and tool support for the quantitative analysis of safety, performance, and workload for human operators early in the design process. This will allow airspace designers to evaluate various design concepts before implementing them.

To this end, in recent ongoing work, we have developed Brahms models for arrivals and departures at the La Guardia airport based on the work on Departure Sensitive Arrival Scheduling (DSAS) concept developed by the Airspace Operations Lab (AOL) at NASA Ames (Lee et al. 2015). The DSAS concept provides the ability to maximize departure throughput at the LaGuardia Airport in the New York metroplex without impacting the flow of the arrival traffic; it was part of a research effort to explore solutions to delays incurred by departures in the New York metroplex. We are able to successfully model DSAS scenarios that consist of approximately 1.5 h real flight time. During this time, there are between 130 and 150 airplanes being managed by four enroute controllers, three TRACON controllers, and one tower controller at LGA who is responsible for departures and arrivals. The planes are landing at approximately 36–40 planes an hour (Rungta et al. 2016). On this model, airspace designers can evaluate different design candidates in terms of the safety, performance, and workload. Our goal is to then turn the modeled constructs into templates for a general framework that would allow airspace designers to extend components based on their needs.

References

Ameur YA, Boniol F, Wiels V (2010) Toward a wider use of formal methods for aerospace systems design and verification. Int J Softw Tools Technol Transf 12(1):1–7. ISSN 1433-2779. doi:10.1007/s10009-009-0131-4

Bastide R, Navarre D, Palanque P (2003) A tool-supported design framework for safety critical interactive systems. Interact Comput 15(3):309–328

Bolton M, Siminiceanu R, Bass E (2011) A systematic approach to model checking human-automation interaction using task analytic models. IEEE Trans Syst Man Cybern Part A: Syst Hum 41(5):961–976

Bolton M, Bass E (2010) Using task analytic models and phenotypes of erroneous human behavior to discover system failures using model checking. In: Proceedings of the 54th annual meeting of the human factors and ergonomics society, pp 992–996

Bolton M, Bass E, Siminiceanu R (2008) Using formal methods to predict human error and system failures. In: Proceedings of the second international conference on applied human factors and ergonomics (AHFE 2008)

Bovair Susan, Kieras DE, Polson PG (1990) The acquisition and performance of text-editing skill: a cognitive complexity analysis. Hum Comput Interact 5(1):1–48

Bratman M (1987) Intention, plans, and practical reason

Bredereke J, Lankenau A (2002) A rigorous view of mode confusion. In: Proceedings of the 21st international conference on computer safety, reliability and security (SAFECOMP 2002). Springer, pp 19–31, Sept 2002

Bredereke J, Lankenau A (2005) Safety-relevant mode confusions–modelling and reducing them. Reliab Eng Syst Saf 88(3):229–245

Butterworth R, Blandford A (1997) Programmable user models: the story so far. Middlesex University, London

Campos JC, Harrison MD (2008) Systematic analysis of control panel interfaces using formal tools. In: Nicholas Graham TC, Palanque PA (eds) Proceedings of the 15th international workshop on design, specification and verification of interactive systems (DSV-IS 2008), vol 5136. Springer, Lecture notes in computer science, pp 72–85

Campos JC, Harrison MD (2011) Model checking interactor specifications. Autom Softw Eng 8(3):275–310

Clancey WJ, Sachs P, Sierhuis M, Van Hoof R (1998) Brahms: simulating practice for work systems design. Int J Hum Comput Stud 49(6):831–865

Combéfis S (2009) Operational model: integrating user tasks and environment information with system model. In: Proceedings of the 3rd international workshop on formal methods for interactive systems, pp 83–86

Combéfis S (2013) A formal framework for the analysis of human-machine interactions. PhD thesis, Université catholique de Louvain

Combéfis S, Giannakopoulou D, Pecheur C (2016) Automatic detection of potential automation surprises for ADEPT models. IEEE Trans Hum Mach Syst Spec Issue Syst Approaches Hum Mach Interface Improv Resil Robust Stab 46(2):

Combéfis S, Giannakopoulou D, Pecheur C, Feary M (2011) A formal framework for design and analysis of human-machine interaction. In: Proceedings of the IEEE international conference on systems, man and cybernetics, Anchorage, Alaska, USA, 9–12 Oct 2011. IEEE, pp 1801–1808. ISBN 978-1-4577-0652-3. doi:10.1109/ICSMC.2011.6083933

Combéfis S, Pecheur C (2009) A bisimulation-based approach to the analysis of human-computer interaction. In: Calvary G, Nicholas Graham TC, Gray P (eds) Proceedings of the ACM SIGCHI symposium on engineering interactive computing systems (EICS'09)

Curzon P, Rukšėnas R, Blandford A (2007) An approach to formal verification of human-computer interaction. Formal Aspects Comput 19(4):513–550

Feary MS (2010) A toolset for supporting iterative human—automation interaction in design. Technical Report 20100012861, NASA Ames Research Center, March 2010

Godefroid P (1997) Model checking for programming languages using verisoft. In: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on principles of programming languages. ACM, pp 174–186

Heymann M, Degani A (2007) Formal analysis and automatic generation of user interfaces: approach, methodology, and an algorithm. Hum Factors: J Hum Factors Ergon Soc 49(2):311–330

Hunter J, Raimondi F, Rungta N, Stocker R (2013) A synergistic and extensible framework for multi-agent system verification. In: Proceedings of the 2013 international conference on autonomous agents and multi-agent systems. International Foundation for Autonomous Agents and Multiagent Systems, pp 869–876

JavaPathfinder (JPF) (2016). http://babelfish.arc.nasa.gov/trac/jpf/

Javaux D (2002) A method for predicting errors when interacting with finite state systems. How implicit learning shapes the user's knowledge of a system. Reliab Eng Syst Saf 75:147–165

Johnson J, Henderson A (2002) Conceptual models: begin by designing what to design. Interactions 9:25–32

Lee PU, Smith NM, Homola J, Brasil C, Buckley N, Cabrall C, Chevalley E, Parke B, Yoo HS (2015) Reducing departure delays at laguardia airport with departure-sensitive arrival spacing (dsas) operations. In: Eleventh USA/Europe air traffic management research and development seminar (ATM)

Leveson NG, Turner CS (1993) Investigation of the therac-25 accidents. IEEE Comput 26(7):18–41

Miller SP, Whalen MW, Cofer DD (2010) Software model checking takes off. Commun ACM 53(2):58–64. ISSN 0001-0782. doi:10.1145/1646353.1646372

Navarre D, Palanque P, Bastide R (2001) Engineering interactive systems through formal methods for both tasks and system models. In Proceedings of the RTO Human Factors and Medicine Panel (HFM) specialists' meeting, pp 20.1–20.17, June 2001

Rungta N, Brat G, Clancey WJ, Linde C, Raimondi F, Seah C, Shafto M (2013) Aviation safety: modeling and analyzing complex interactions between humans and automated systems. In: Proceedings of the 3rd international conference on application and theory of automation in command and control systems. ACM, pp 27–37

Rungta N, Mercer EG, Raimondi F, Krantz BC, Stocker R, Wallace A (2016) Modeling complex air traffic management systems. In: Proceedings of the 8th international workshop on modeling in software engineering. ACM, pp 41–47

Rushby JM (2011) New challenges in certification for aircraft software. In: Chakraborty S, Jerraya A, Baruah SK, Fischmeister S (eds) EMSOFT. ACM, pp 211–218. ISBN 978-1-4503-0714-7

Sierhuis M (2001) Modeling and simulating work practice. BRAHMS: a multiagent modeling and simulation language for work system analysis and design. PhD thesis, Social Science and Informatics (SWI), University of Amsterdam, SIKS Dissertation Series No. 2001-10, Amsterdam, The Netherlands

Thimbleby H (2010) Press on: principles of interaction programming. The MIT Press, Jan 2010. ISBN 0262514230

Thimbleby H, Gow J (2007) Applying graph theory to interaction design. In Gulliksen J, Harning MB, Palanque P, van der Veer G, Wesson J (eds) Proceedings of the engineering interactive systems joint working conferences EHCI, DSV-IS, HCSE (EIS 2007). Lecture notes in computer science, vol 4940, pp 501–519. Springer, Mar 2007

Tretmans J (2008) Model based testing with labelled transition systems. In: Hierons R, Bowen J, Harman M (eds) Formal methods and testing. Lecture notes in computer science, vol 4949. Springer, pp 1–38

uberlingeng (2004) Investigation Report AX001-1-2/02. Technical report, German Federal Bureau of Aircraft Accidents Investigation

Yasmeen A, Gunter EL (2011) Automated framework for formal operator task analysis. In Dwyer MB, Tip F (eds) ISSTA. ACM, pp 78–88

Young RM, Green TRG, Simon T (1989) Programmable user models for predictive evaluation of interface designs. In: ACM SIGCHI bulletin, vol 20. ACM, pp 15–19

Part IV
Future Opportunities and Developments

Chapter 16
Domain-Specific Modelling for Human–Computer Interaction

Simon Van Mierlo, Yentl Van Tendeloo, Bart Meyers and Hans Vangheluwe

Abstract Model-driven engineering (MDE) is an important enabler in the development of complex, reactive, often real-time, and software-intensive systems, as it shifts the level of specification from computing concepts (the "how") to conceptual models or abstractions in the problem domain (the "what"). Domain-specific modelling (DSM) in particular allows to specify these models in a domain-specific modelling language (DSML), using concepts and notations of a specific domain. It allows the use of a custom visual syntax which is closer to the problem domain and therefore more intuitive. Models created in DSMLs are used, among others, for simulation, (formal) analysis, documentation, and code synthesis for different platforms. The goal is to enable domain experts, such as a power plant engineer, to develop, to understand, and to verify models more easily, without having to use concepts outside of their own domain. The first step in the DSM approach when modelling in a new domain is, after a domain analysis, creating an appropriate DSML. In this chapter, we give an introduction to DSML engineering and show how it can be used to develop a human–computer interaction interface. A DSML is fully defined by its syntax and semantics. The syntax consists of (i) the abstract syntax, defining the DSML constructs and their allowed combinations, captured in a metamodel, and (ii) the concrete syntax, specifying the visual representation of the different constructs. The semantics defines the meaning of models created in the domain. In this chapter, we show how two types of semantics (operational and translational) can be

S. Van Mierlo (✉) · Y. Van Tendeloo · B. Meyers
University of Antwerp, Antwerp, Belgium
e-mail: simon.vanmierlo@uantwerpen.be

Y. Van Tendeloo
e-mail: yentl.vantendeloo@uantwerpen.be

B. Meyers
e-mail: bart.meyers@uantwerpen.be

H. Vangheluwe
University of Antwerp—Flanders Make, Antwerp, Belgium
e-mail: hv@cs.mcgill.ca

H. Vangheluwe
McGill University, Montreal, Canada

B. Weyers et al. (eds.), *The Handbook of Formal Methods in Human-Computer Interaction*, Human-Computer Interaction Series,
DOI 10.1007/978-3-319-51838-1_16

modelled using model transformations. Operational semantics gives meaning to the modelling language by continuously updating the model's state, effectively building a simulator. Translational semantics defines mappings of models in one language onto models in a language with known semantics. This enables the automatic construction of behavioural models, as well as models for verification. The former can be used for automated code synthesis (leading to a running application), whereas the latter leads to model checking. We choose to specify properties for model checking using the ProMoBox approach, which allows the modelling of properties in a syntax similar to the original DSML. A major advantage of this approach is that the modeller specifies both requirements (in the form of properties) and design models in a familiar notation. The properties modelled in this domain-specific syntax are verified by mapping them to lower-level languages, such as Promela, and results are mapped back to the domain-specific level. To illustrate the approach, we create a DSML for modelling the human–computer interaction interface of a nuclear power plant.

16.1 Introduction

Developing complex, reactive, often real-time, and software-intensive systems using a traditional, code-centric approach is not an easy feat: knowledge is required from both the problem domain (e.g. power plant engineering) and computer programming. Apart from being inefficient and costly, due to the need for an additional programmer on the project, this can also result in more fundamental problems. The programmer, who implements the software, has no knowledge of the problem domain, or basic knowledge at best. The domain expert, on the other hand, has deep knowledge of the problem domain, but only a limited understanding of computer programs. This can result in communication problems, such as the programmer making false assumptions about the domain or the domain expert to gloss over details when explaining the problem to the programmer. Furthermore, the domain experts will finally receive a software component that they do not fully understand, making it difficult for them to validate and modify if necessary. There is a conceptual gap in effect between the two domains, hindering productivity.

Model-driven engineering (MDE) (Vangheluwe 2008) tries to bridge this gap, by shifting the level of specification from computing concepts (the "how") to conceptual models or abstractions in the problem domain (the "what"). Domain-specific modelling (DSM) (Kelly and Tolvanen 2008) in particular makes it possible to specify these models in a domain-specific modelling language (DSML), using concepts and notations of a specific domain. The goal is to enable domain experts to develop, to understand, and to verify models more easily, without having to use concepts outside of their own domain. It allows the use of a custom visual syntax, which is closer to the problem domain and therefore more intuitive. Models created in such DSMLs are used, among others, for simulation, (formal) analysis, documentation, and code synthesis for different platforms. There is, however, still a need for a language engineer to create the DSML, which includes defining its syntax and providing the mapping between the problem domain and the solution domain.

16.1.1 Case Study

In this chapter, we explain the necessary steps for developing a system using the DSM approach, applied to the nuclear power plant control interface case study (see Chap. 4). We build this human–computer interaction interface incrementally throughout the chapter. The system consists of two parts:

1. The nuclear power plant, which takes actions as input (e.g. "lower the control rods"), and output events in case of warning and error situations. Each nuclear power plant component is built according to its specification, which lists a series of requirements (e.g. "the reactor can only hold 450 bar pressure for 1 min"). These are specified in the model of each component. Components send warning messages in case their limits are almost reached, such that the user can take control and alleviate the problem. When the user is unable to bring the reactor to a stable state, however, the component sends an error message, indicating that an emergency shutdown is required to prevent a nuclear meltdown. There is a distinction between two types of components:

   a. Monitoring components, which monitor the values of their sensors and send messages to the controller depending on the current state of the component. An example is the generator, which measures the amount of electricity generated. Their state indicates the status of the sensors: *normal*, *warning*, or *error*.
   b. Executing components, which receive messages from the controller and execute the desired operation. A valve, for example, is either open or closed. Their state indicates the physical state of the component, for example *open* or *closed*.

2. The controller, which acts as the interface between the plant and the user. Users can send messages to the controller by pressing buttons. It is, however, up to the controller to pass on this request to the actual component(s) or choose to ignore it (possibly sending other messages to components, depending on the state of the reactor core). We implement a controller which has three main modes:

   a. Normal operation, where the user is unrestricted and all messages are passed.
   b. Restricted operation, where the user can only perform actions which lower the pressure in the reactor. This mode is entered when any of the components sends out a warning message. When all components are back to normal, full control is returned to the user.
   c. Emergency operation, where control is taken away from the user, and the controller takes over. This mode is entered when any of the components sends out an error message. The controller will forcefully shut down the reactor and ignore further input from the user. As there is likely damage to the power plant's components, it is impossible to go back to normal operation without a full reboot.

We construct a DSML which makes it possible to model the configuration of a nuclear power plant and express the behaviour of each component, as well as the controller. We intend to automatically synthesize code, which behaves as specified in the model. Additionally, we use the ProMoBox approach (Meyers et al. 2014) to verify that all of the desired (safety) properties are fulfilled by the modelled human–computer interface.

We use the open-source metamodelling tool AToMPM (Syriani et al. 2013) ("A Tool for Multi-Paradigm Modelling") throughout this chapter, though the approach can be applied in other metamodelling tools with similar capabilities. All aspects of defining a new language are supported: creating an abstract syntax, defining custom concrete syntax, and defining semantics through the use of model transformations.

16.1.2 Terminology

The first step in the DSM approach when modelling in a new domain is, after a domain analysis, creating an appropriate DSML. A DSML is fully defined (Kleppe 2007) by:

1. Its **abstract syntax**, defining the DSML constructs and their allowed combinations. This information is typically captured in a metamodel.
2. Its **concrete syntax**, specifying the visual representation of the different constructs. This visual representation can be either graphical (using icons), or textual.
3. Its **semantics**, defining the meaning of models created in the domain (Harel and Rumpe 2004). This encompasses both the **semantic domain** (*what* is its meaning) and the **semantic mapping** (*how* to give it meaning).

For example, $1 + 2$ and (+ 1 2) can both be seen as textual concrete syntax (i.e. a visualization) for the abstract syntax concept "addition of 1 and 2" (i.e. what construct it is). The semantic domain of this operation is the set of natural numbers (i.e. what it evaluates to), with the semantic mapping being the execution of the operation (i.e. how it is evaluated). Therefore, the semantics, or "meaning", of $1 + 2$ is 3.

This definition of terminology is shown in Fig. 16.1. Each aspect of a formalism is modelled explicitly, as well as relations between different formalisms. Throughout the remainder of this paper, we present these four aspects in detail and present the model(s) related to the use case for each.

Aside from the language definition, properties are defined that should hold for models defined in the language and can be checked. Whereas such properties are normally expressed in property languages, such as LTL (Pnueli 1977), we will use the ProMoBox (Meyers et al. 2014) approach. With LTL, we would revert back to the original problem: the conceptual gap between the problem domain and the implementation level is reintroduced. With the ProMoBox approach, properties are modelled using a syntax similar to that of the original DSML. Thus, the modeller specifies both the requirements, or properties, and design models in a familiar notation, lifting both to the problem domain. In case a property does not hold, feedback is furthermore given by the system at the domain-specific level.

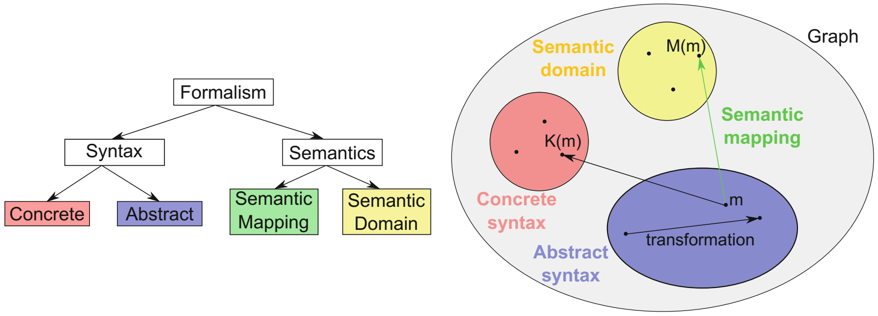


Fig. 16.1 Terminology

16.1.3 Outline

The remainder of this chapter is structured as follows. Section 16.2 presents the different aspects of syntax: both abstract and concrete. Section 16.3 presents an introduction to the definition of semantics, and how we apply this to our case study. Section 16.4 explains the need for properties at the same level as the domain-specific language and presents the ProMoBox approach. Section 16.5 concludes. Throughout the chapter, we use the nuclear power plant case study to illustrate how each concept is applied in practice.

16.2 Syntax

A syntax defines whether elements are valid in a specified language or not. It does not, however, concern itself with what the constructs mean. With syntax only, it would be possible to specify whether a construct is valid, but it might have invalid semantics. A simple, textual example is the expression $\frac{1}{0}$. It is perfectly valid to write this, as it follows all structural rules: a fraction symbol separates two recursively parsed expressions. However, its semantics is undefined, since it is a division by zero.

16.2.1 Abstract Syntax

The abstract syntax of a language specifies its constructs and their allowed combinations and can be compared to grammars specifying parsing rules. Such definitions are captured in a metamodel, which itself is again a model of the metametamodel (Kühne 2006). Most commonly, the metametamodel is similar to UML class diagrams, as is also the case in our case study. The metametamodel used in the examples makes it possible to define classes, associations between classes (with incoming and outgoing multiplicities), and attributes.

While the abstract syntax reasons about the allowable constructs, it does not state anything about how they are presented to the user. In this way, it is distinct from textual grammars, as they already offer the keywords to use (Kleppe 2007). It merely states the concepts that are usable in the domain.

A possible abstract syntax definition for the nuclear power plant use case is shown in Fig. 16.2. It defines the language constructs, such as a *reactor*, *pump*s, and *valve*s, which can have attributes defining its state (e.g. a valve can be open or closed). It also lists the allowed associations between abstract syntax elements (e.g. a generator cannot be directly connected to a steam valve).

The constructs for modelling the behaviour of the elements are also present. The abstract syntax requires all components to have exactly one behavioural definition. In this definition, the component is in a specific state which has transitions to other states. For each transition, it is possible to define cardinalities, to limit the number of outgoing links of each type. For example, the metamodel forbids a single state from having two outgoing transitions of certain types. While we do not yet give semantics to our language, we already limit the set of valid models (i.e. for which semantics needs to be defined). By preventing ambiguous situations in the abstract syntax, we do not need to take them into account in the semantic definition, as they represent invalid configurations.

This language definition, together with the concrete syntax definition, is used by AToMPM to generate a domain-specific syntax-directed modelling environment, which means that only valid instances can be created. For example, if the abstract syntax model specifies that there can only be a single SCRAM transition, then drawing a second one will give an error. This maximally constrains the modeller and ensures the models are (syntactically) correct by construction.

16.2.2 Concrete Syntax

The concrete syntax of a model specifies how elements from the abstract syntax are visually represented. The relation between abstract and concrete syntax elements is also modelled: each representable abstract syntax concept has exactly one concrete syntax construct, and vice versa. As such, the mapping between abstract and concrete syntax needs to be a bijective function. This does not, however, limit the number of distinct concrete syntax definitions, as long as each combination of concrete and abstract syntax has a bijective mapping. The definition of the concrete syntax is a determining factor in the usability of the DSML (Barišić et al. 2011).

Multiple types of concrete syntaxes exist, though the main distinction is between **textual** and **graphical** languages. Both have their advantages and disadvantages: textual languages are more similar to programming languages, making it easier for programmers to start using the DSML. On the other hand, visual languages can represent the problem domain better, due to the use of standardized symbols, despite them being generally less efficient (Petre 1995). An overview of different types of graphical languages is given in Costagliola et al. (2004). Different tools have differ-

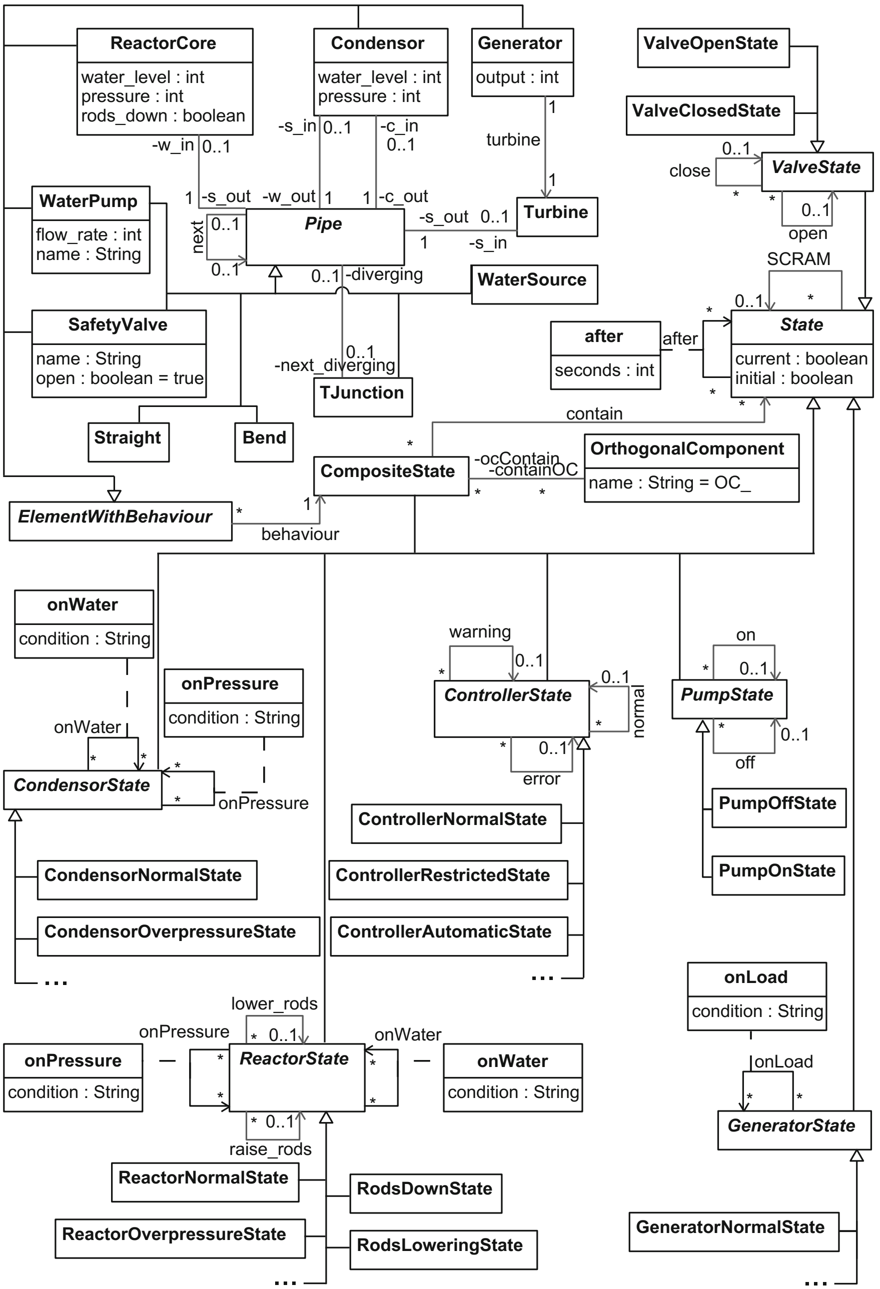


Fig. 16.2 Abstract syntax definition for the nuclear power plant domain (some subclasses omitted)

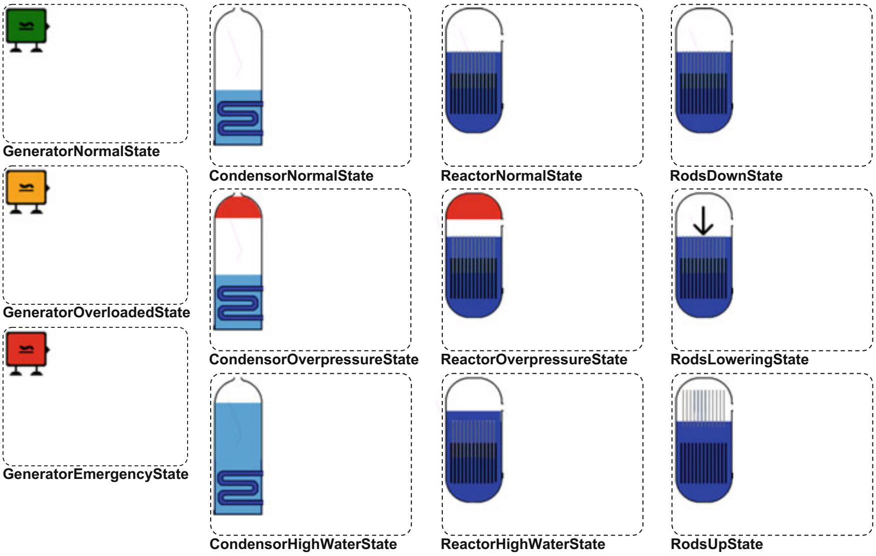


Fig. 16.3 Concrete syntax definition for the nuclear power plant domain (excerpt)

ent options for concrete syntax, depending on the expected target audience of the tool. For example, standard parsers always use a textual language, as their target audience consists of computer programmers who specify a system in (textual) code.

While the possibilities with textual languages are rather restricted, graphical languages have an almost infinite number of possibilities. In Moody (2009), several "rules" are identified for handling this large number of possibilities. As indicated beforehand, a single model can have different concrete syntax representations, so it is possible for one to be textual and another to be graphical.

An excerpt of a possible visual concrete syntax definition for the nuclear power plant use case is shown in Fig. 16.3. Each of the constructs presented in the concrete syntax model corresponds to the abstract syntax element with the same name. Every construct receives a visual representation that is similar to the one defined in the case study. In case of standardized icons or symbols, it would be trivial to define a new concrete syntax model. Furthermore, a specific concrete syntax was created for the definition of the states. Each state is a graphical representation of the state of the physical component, making it easier for users to determine what happens. We chose to attach the graphical representation given in the case study to represent the *normal* functioning of the reactor core; this results in an identical representation of the "normal reactor state" and the "rods down state".

Now that we have a fully defined syntax for our model, we create an instance of the use case in AToMPM, of which a screenshot is shown in Fig. 16.4. A domain-specific modelling environment, generated from the language definition, is loaded into AToMPM, as displayed by the icon toolbar at the top, below AToMPM's general purpose toolbars. This example instance shows a model very similar to the one in the case study. Each component additionally has a specification of its dynamic

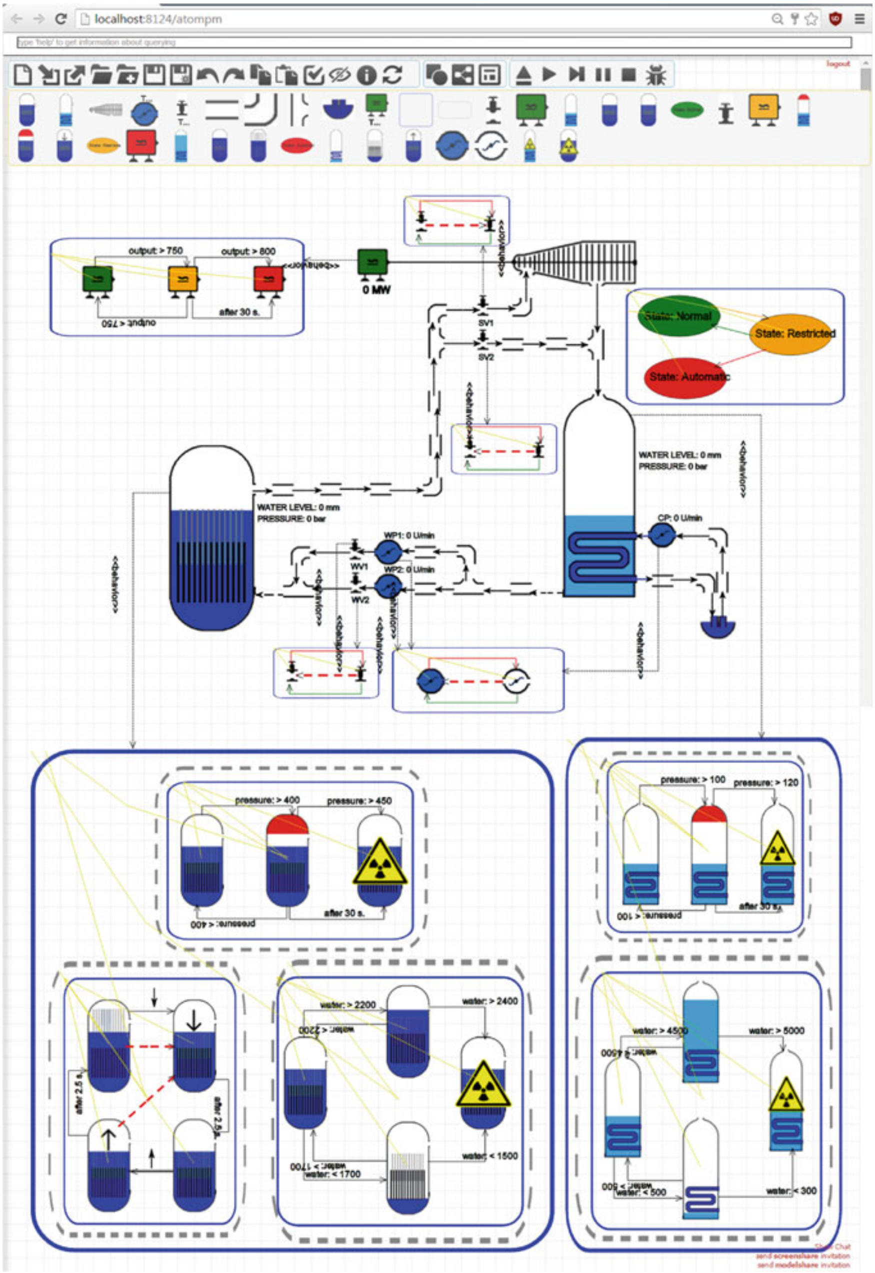


Fig. 16.4 Screenshot of AToMPM with an example nuclear power plant instance

behaviour. This behaviour definition specifies when to send out messages, using the state of the underlying system, as well as timeouts. The controller is also constructed as per our presented case study.

While the meaning of this model might be intuitively clear, this model does not yet have any meaning, as there is no semantics defined. Defining the semantics of this model is the topic of the next section.

16.3 Semantics

Since the syntax only defines what a valid model looks like, we need to give a meaning to the models. Even though models might be syntactically valid, their meaning might be useless or even invalid.

It is possible for humans to come up with intuitive semantics for the visual notations used (e.g. an arrow between two states means that the state changes from the source to the destination if a certain condition is satisfied). There is, however, a need to make the semantics explicit for two main reasons:

1. Computers cannot use intuition, and therefore, there needs to be some operation defined to convey the meaning to the machine level.
2. Intuition might only take us that far and can cause some subtle differences in border cases. Having semantics explicitly defined makes different interpretations impossible, as there will always be a “reference implementation”.

Semantics consists of two parts: the domain it maps to and the mapping itself. Both will be covered in the next subsections.

16.3.1 Semantic Domain

The semantic domain is the target of the semantic mapping. As such, the semantic mapping will map every valid language instance to a (not necessarily unique) instance of the semantic domain. Many semantic domains exist, as basically every language with semantics of its own can act as a semantic domain. The choice of semantic domain depends on which properties need to be conserved. For example, DEVS (Zeigler et al. 2000) can be used for simulation, Petri nets (Murata 1989) for verification, Statecharts (Harel 1987) for code synthesis, and Causal Block Diagrams (Cellier 1991) for continuous systems using differential equations. A single model might even have different semantic domains, each targeted at a specific goal.

For our case study, we use Statecharts as the semantic domain, as we are interested in the timed, reactive, and autonomous behaviour of the system, as well as code synthesis. Statecharts were introduced by Harel (1987) as an extension of state machines and state diagrams with hierarchy, orthogonality, and broadcast communication. It is used for the specification and design of complex discrete-event systems and is popular for the modelling of reactive systems, such as graphical user interfaces. The Statecharts language consists of the following elements:

- *states*, either basic, orthogonal, or hierarchical;
- *transitions* between states, either event-based or time-based;
- *actions*, executed when a state is entered and/or exited;
- *guards* on transitions, modelling conditions that need to be satisfied in order for the transition to "fire";
- *history* states, a memory element that allows the state of the Statechart to be restored.

Figure 16.5 presents a Statecharts model which is equivalent to the model in our DSML, at least with respect to the properties we are interested in. Parts of the structure can be recognized, though information was added to make the model compliant to the Statecharts formalism. Additions include the sending and receiving of events and the expansion of forwarding states such as in the controller. The semantic mapping also merged the different behavioural parts into a single Statechart. There are two types of events in the resulting Statecharts model: discrete events coming from the operator (e.g. *lower control rods*) and events coming from the environment, corresponding to sensor readings (e.g. *water level* in reactor). These discrete values are changed into boundary crossing checks, which cannot be easily modelled using Statecharts. Instead, to model these, a more suitable formalism should be chosen, such as Causal Block Diagrams (Cellier 1991) (CBDs). These CBD models then need to be embedded into the Statecharts model. It is necessary to connect both formalisms semantically, such that, for example, a signal value in the CBD model translates to an event in the Statecharts model (Boulanger et al. 2011). This is out of scope for this paper and does not influence the properties we are interested in. We assume the sensor readings are updated correctly and communicated to our Statecharts model.

As is usually the case, the DSML instance is more compact and intuitive, compared to the resulting Statechart instance. The Statecharts language itself also needs to have its semantics defined, as done in Harel and Naamad (1996).

In the following subsection, we define one semantic mapping that maps onto the Statecharts language (called "translational semantics") and one mapping that maps the language onto itself (called "operational semantics").

16.3.2 Semantic Mapping

While many categories of semantic mapping exist, as presented in Zhang and Xu (2004), we only focus on the two main categories relevant to our case study:

1. **Translational semantics**, where the semantic mapping translates the model from one formalism to another, while maintaining an equivalent model with respect to the properties under study. The target formalism has semantics (again, either translational or operational), meaning that the semantics is "transferred" to the original model.

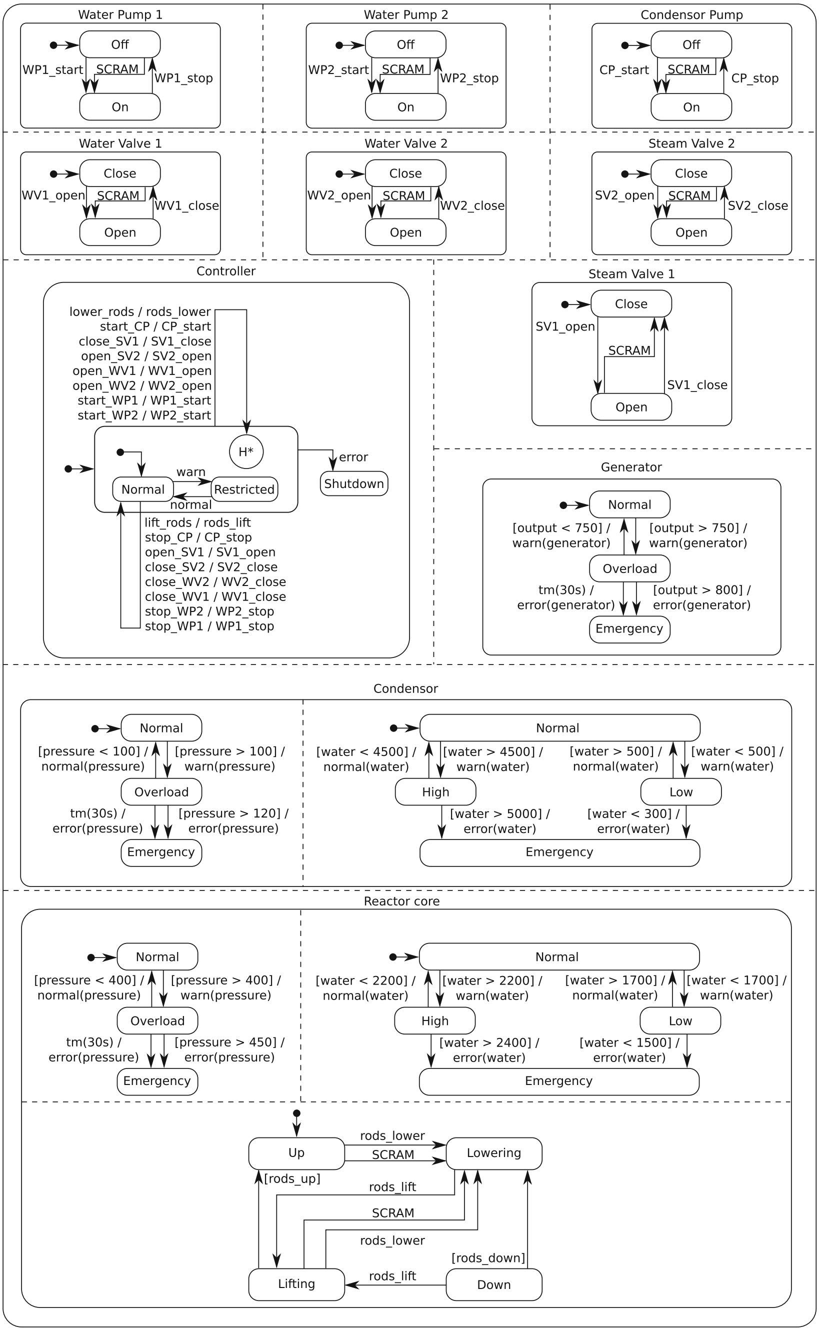


Fig. 16.5 Statechart equivalent to the behaviour shown in Fig. 16.4

2. **Operational semantics**, where the semantic mapping effectively executes, or simulates, the model being mapped. Operational semantics can be implemented with an external simulator, or through model transformations that simulate the model by modifying its state. The advantage of in-place model transformations is that semantics is also defined completely in the problem domain, making it suitable for use by domain experts. For our case study, this means implementing a simulator using model transformations.

Due to the possibly many semantic mappings, their chaining can be explicitly modelled in a Formalism Transformation Graph+Process Model (FTG+PM) (Lucio et al. 2013). An FTG+PM can encompass both automatic transformations (i.e. through model transformations) and manual transformations (i.e. through a user creating a related model).

The semantic mapping, which translates between a source and target model, is commonly expressed using model transformations, which are often called the heart and soul of model-driven engineering (Sendall and Kozaczynski 2003). A model transformation is defined using a set of transformation rules and a schedule.

A rule consists of a left-hand side (LHS) pattern (transformation precondition), right-hand side (RHS) pattern (transformation post-condition), and possible negative application condition (NAC) patterns (patterns which, if found, stop rule application). The rule is applicable on a graph (the host graph), if each element in the LHS can be matched in the model, without being able to match any of the NAC patterns. When applying a rule, the elements matched by the LHS are replaced by the elements of the RHS in the host graph. Elements of the LHS that do not appear in the RHS are removed, and elements of the RHS that do not appear in the LHS are created. Elements can be labelled in order to correctly link elements from the LHS and RHS.

A schedule determines the order in which transformation rules are applied. For our purpose, we use MoTiF (Syriani and Vangheluwe 2013), which defines a set of basic building blocks for the transformation rule scheduling. We limit ourself to three types of rules: the *ARule* (apply a rule once), the *FRule* (apply a rule for all matches simultaneously), and the *CRule* (for nesting transformation schedules).

In the following subsections, we define both the operational and translational semantics of our modelling language.

16.3.2.1 Translational Semantics

With translational semantics, the source model is translated to a target model, expressed in a different formalism, which has its own semantic definition. The (partial) semantics of the source model then corresponds to the semantics of the target model. As the rule uses both concepts of the problem domain and the target domain (Statecharts in our case), the modeller should be familiar with both domains. The Statecharts language, however, is still much more intuitive and, in the case of a real-time, reactive, and autonomous system, more appropriate to use than the average programming language.

The schedule of our transformation is shown in Fig. 16.6, where we see that each component is translated in turn. Each of these blocks is a composite rule, meaning that they invoke other schedules. One of these schedules is shown in Fig. 16.7, where the valves are translated. The blocks in the schedule are connected using arrows to denote the flow of control: green arrows originating from a check mark are followed when the rule executed successfully, while red arrows originating from a cross are followed when an error occurred. Three pseudo-states denote the start, the successful termination, and the unsuccessful termination of the transformation. Our schedule consists of a series of FRules, which translate all different valve states to the corresponding Statecharts states. After these are mapped, the transitions between them are also translated, as shown in the example rule in Fig. 16.8. In this rule, we look

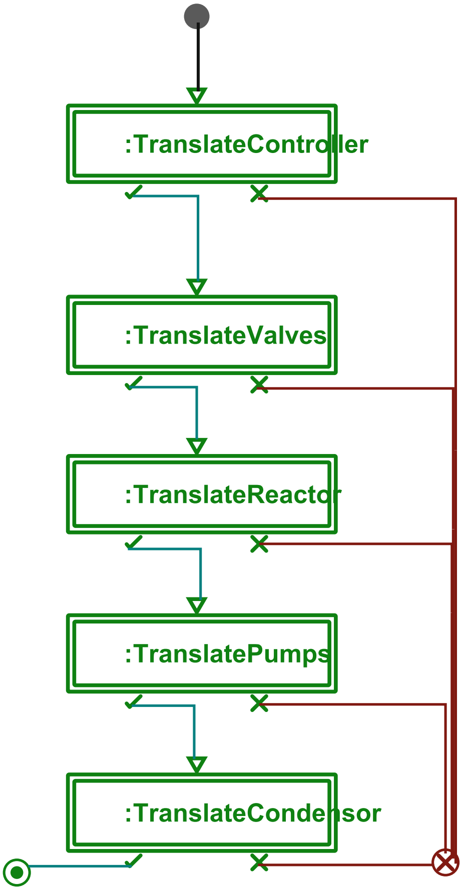


Fig. 16.6 Top-level transformation schedule

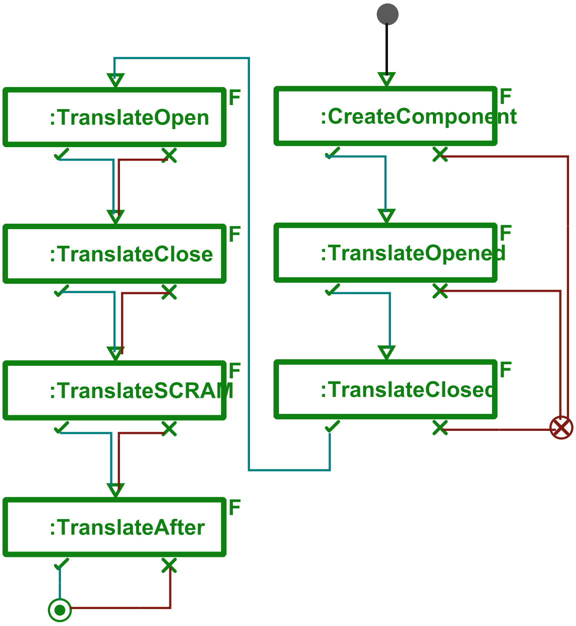


Fig. 16.7 Transformation schedule for valves

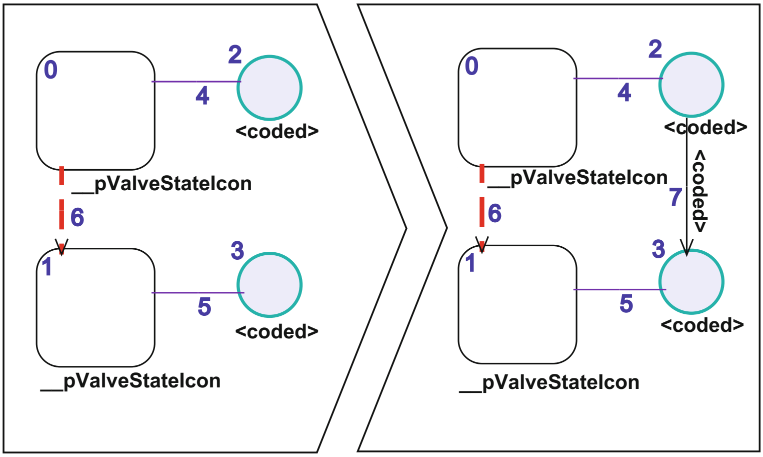
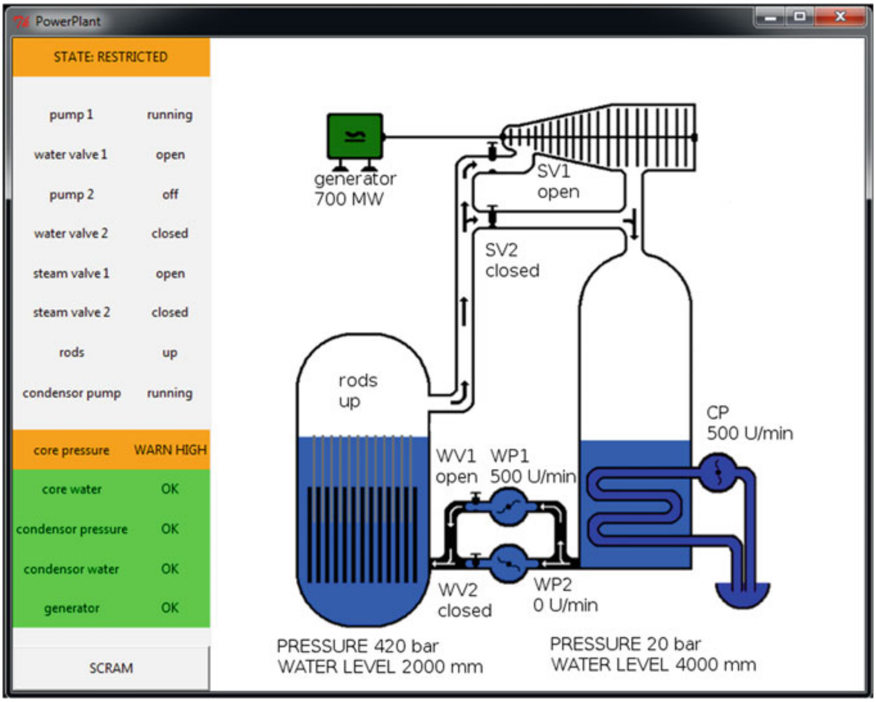


Fig. 16.8 Example transformation rule for translational semantics

up the Statecharts states generated from the domain-specific states and create a link between them if there was one in the domain-specific language. For each kind of link, there needs to be such a rule. In this case, we map the SCRAM message to a Statecharts transition which takes a specific kind of event. Note that we do not remove the original model, ensuring traceability.



STATE: RESTRICTED

pump 1	running
water valve 1	open
pump 2	off
water valve 2	closed
steam valve 1	open
steam valve 2	closed
rods	up
condensor pump	running
core pressure	WARN HIGH
core water	OK
condensor pressure	OK
condensor water	OK
generator	OK

SCRAM

Fig. 16.9 Example interface in Python TkInter

The generated Statechart can be coupled to a user interface, to allow user interaction. Figure 16.9 presents an example interface, created with Python TkInter. To the left, the user is presented with a colour-coded overview of the system. The states of operational elements (e.g. valves) are marked in grey, and elements that can emit a warning or error message are marked in either green, orange, or red. On top, an overview of the current state of the system is shown, indicating what level of responsiveness the user can expect. At the bottom is a SCRAM button, which, when pressed, will simply emit the "SCRAM" event to the Statecharts model. The visual representation at the right labels all parts and provides more in-depth information about each component. Elements can be clicked to interact with them. For example, a pump can be enabled or disabled by clicking on it in the figure. The icons used in the user interface are similar to those of the concrete syntax, to retain consistency with the system definition (modelled in the domain-specific language). This is not mandatory, however, and other icons could be chosen. As previously mentioned, this interface is only a minimal example to illustrate the applicability of our approach.

After creating this interface, it needs to be coupled to the Statecharts model. This is done through the use of events: the interface uses the interface of the model to raise and catch events to and from the Statecharts model. For example, pressing the "SCRAM" button raises the "SCRAM" event. Similarly, when the interface catches the event "warning high core pressure", it visualizes this change by changing the colour of the pressure reading. As such, the interface does not implement any autonomous behaviour, but relies fully on the Statecharts model. Different interfaces can be coupled, as long as they adhere to the interface of the model.

16.3.2.2 Operational Semantics

A formalism is operationalized by defining a simulator for that formalism. This simulator can be modelled as a model transformation that executes the model by continuously updating its state (effectively defining a "stepping" function). The next state of the model is computed from the current state, the information captured in the model (such as state transitions and conditions), and the current state of the environment. Contrary to translational semantics, the source model of operational semantics is often augmented with run-time information. This requires the creation of both a "design formalism" and a "run-time formalism". In our case study, for example, the run-time formalism is equivalent to the design formalism augmented with information on the current state and the simulated time, as well as a list of inputs from the environment.

An example rule is shown in Fig. 16.10. The rule changes the current state by following the "onPressure" transition. The left-hand side of the rule matches the current state, the value of the sensor, and the destination of the transition. It is only applicable if the condition on the transition (e.g. >450) is satisfied (by comparing it to the value of the sensor reading). We use abstract states for both source and target

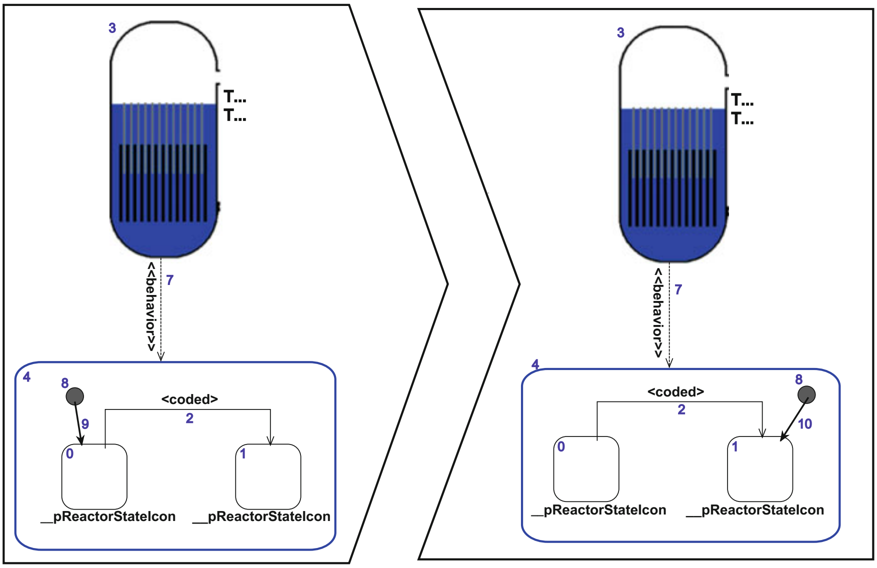


Fig. 16.10 Example transformation rule for operational semantics

of the transition, as we do not want to limit the application of the rule to a specific combination of states: the rule should be applicable for all pairs of reactor states that have an "onPressure" transition. The right-hand side then changes the current state to the target of the transition.

The schedule has the form of a "simulation loop", but is otherwise similar to the one for translational semantics and is therefore not shown here.

16.4 Verification of Properties

DSM mainly focuses on the design of software systems, but can also greatly help in formal verification of these systems. Requirements can be made explicit in the form of properties: questions one can ask of the system, for which the answer is either "yes" or "no". Depending on the nature of the verification problem, these properties can be checked using model checking, symbolic execution, constraint satisfaction, solving systems of equations, etc.

The general verification process is shown in Fig. 16.11, where a formal model and properties are fed into a verification engine. If no counterexample is found, the system satisfies the property. If a counterexample is found, it needs to be visualized to be able to correct the formal model, after which the verification process can be restarted.

In the context of DSM, model-to-model transformations can be used to transform the DSML instance to a formal model, on which model checking can be applied (Risoldi 2010). In this case, the modeller needs to specify and check the properties directly on the formal model, often in a notation as LTL (Pnueli 1977), and needs to transform the verification results back to the DSM level. Having to work with such intricate notations contradicts the philosophy of DSM, where models should be specified in terms of the problem domain.

One solution is to create a DSML for property specification in the same manner as the DSML for designing systems. An example is TimeLine (Smith et al. 2001): a

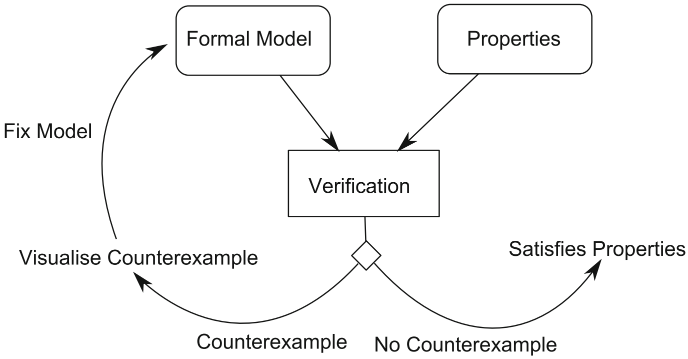


Fig. 16.11 Verification of formally specified systems

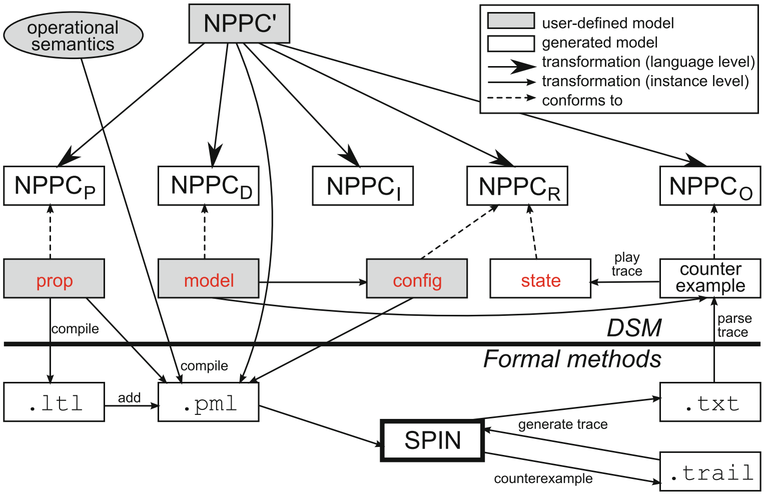


Fig. 16.12 The *ProMoBox* approach applied to the nuclear power plant control (*NPPC*) DSML

visual DSML for specifying temporal constraints for event-based systems. Developing and maintaining yet another DSML come at a great cost, again contradicting the DSM philosophy of fast language engineering. Possible counterexamples also need to be visualized somehow, possibly requiring yet another DSML.

In this section, we apply the *ProMoBox* approach (Meyers et al. 2013, 2014; Meyers and Vangheluwe 2014) which resolves the above issues to the nuclear power plant DSML. The *ProMoBox* approach for the nuclear power plant control (*NPPC*) is laid out in Fig. 16.12. Minimal abstraction and annotation of the *NPPC'* DSML is required to generate the *ProMoBox*—a family of five DSMLs that cover all tasks of the verification process:

- The *design language* $NPPC_D$ allows DSM engineers to design the static structure of the system, similarly to traditional DSM approaches as described earlier in this chapter.
- The *run-time language* $NPPC_R$ enables modellers to define a state of the system, such as an initial state as input of a simulation, or a particular "snapshot" or state during run-time.
- The *input language* $NPPC_I$ lets the DSM engineer model the behaviour of the system environment, for example by modelling an input scenario as an ordered sequence of events containing one or more input elements.
- The *output language* $NPPC_O$ is used to represent execution traces (expressed as ordered sequences of states and transitions) of a simulation or to show verification results in the form of a counterexample. Output models can also be created manually as part of an oracle for a test case.
- The *property language* $NPPC_P$ is used to express properties based on modal temporal logic, including structural logic and quantification.

The bottom side of Fig. 16.12 reflects one instance using the SPIN model checker of the verification process shown in Fig. 16.11. The modeller uses the *ProMoBox* to model not only the nuclear power plant system, but also temporal properties. The *ProMoBox* approach supports fully automatic compilation to LTL and Promela, verification using the *SPIN model checker* (Holzmann 1997), and subsequent visualization of counterexamples. In conclusion, by using the *ProMoBox* approach, the user can specify or inspect all relevant models at the domain-specific level, while the development overhead is kept to a minimum.

16.4.1 Abstraction and Annotation Phase

Since model checking techniques will be applied, a simplification step might be necessary to reduce the combinatorial search space of the model, to avoid long verification time or extensive memory usage. This scalability problem is inherent to model checking: typical culprits for state space explosion are the number of variables and the size of variable's domains. Attributes in the metamodel must therefore be simplified by abstracting the domain of some relevant language constructs (see Fig. 16.13).

- All attributes of type *integer* are reduced to *Boolean*s or *enumeration*s. We use system information to find the enumeration values, meaning that the DSML is reduced to an even smaller domain: the enumeration values represent ranges of integer values (e.g. *high*, *low*, or *normal* water levels abstract away the actual value of the water level, but can still be used for analysis, as we are only interested in what "state" the water level is in).

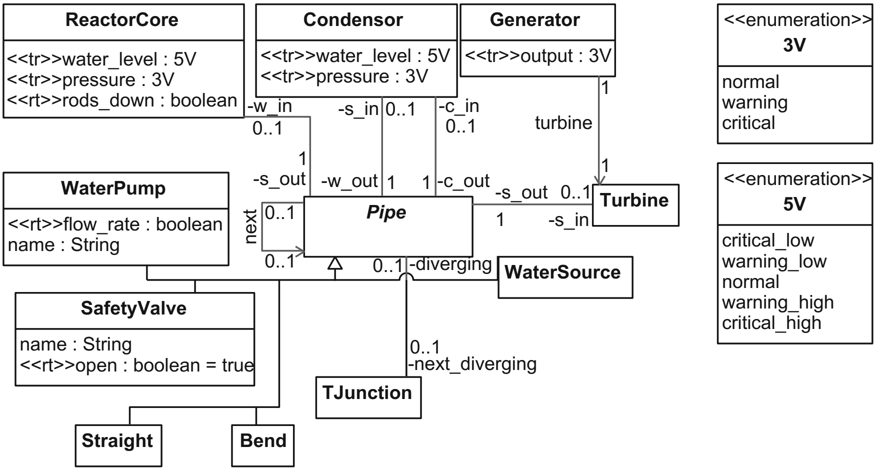


Fig. 16.13 The simplified and annotated metamodel (excerpt)

- Conditions on transitions are reduced to denote which of the enumeration values yield *true* (as a set of enumeration literals).
- The attribute denoting number of seconds in the *after* attribute of the association is removed, since no concept of real time will be used in the properties.

Other simplifications to reduce the state space, such as rearranging the transformation schedule and breaking down metamodel hierarchy, are beyond the scope of this chapter. The *ProMoBox* approach includes guidelines for simplification.

Once all necessary simplifications are performed, annotations need to be added to the DSML's metamodel. Annotations add semantics to elements in the metamodel (classes, associations, and attributes), denoting whether elements are *static* (do not change at run-time), are *dynamic* (change at run-time, thus reflecting the state of the system), or are part of the environment. To achieve this, annotations mark in which of the five languages these elements should be available. By default, elements are not annotated, meaning that they appear in the design, run-time, output, and property languages, but not in the input language.

Three annotations are provided:

- «*rt*»: run-time, annotates a dynamic concept that serves as output (e.g. a state variable);
- «*ev*»: an event, annotates a dynamic concept that serves as input only (e.g. a marking); and
- «*tr*»: a trigger, annotates a static or dynamic concept that also serves as input (e.g. a transition event).

Other annotations can be added to the annotations model, as long as certain constraints are met (e.g. availability in the design language implies availability in the run-time language).

In Fig. 16.13, we show some attribute annotations. The attributes *flow_rate*, *open*, *rods_down*, and *current* represent the observable state of the nuclear power plant and are therefore annotated with «*rt*». The attributes *water_level* and *pressure* can be changed by the environment and are therefore annotated with «*tr*». Classes, associations, and attributes such as *name* and *initial* are not annotated, meaning that they are only part of the static structure of the model.

The concrete syntax of the simplified DSML requires slight adaptation, such that the enumeration values are shown instead of the integers. The simplified and annotated metamodel, of which an excerpt is shown in Fig. 16.13, contains sufficient information to generate the *ProMoBox* consisting of five languages.

16.4.2 ProMoBox *Generation Phase*

A family of five languages is generated from the annotated metamodel using a template-based approach. For every language, the elements of the annotated metamodel according to the annotations are combined with a predefined template.

For example, Fig. 16.14 shows the metamodel of the generated *design language* $NPPC_D$. Generic template elements (shown as grey rectangles in Fig. 16.15) are combined with the DSL metamodel elements through the use of inheritance. The template consists of one simple class *DesignElement* with an attribute *id*, used for traceability. All DSML classes transitively inherit from this single class. Some attributes, such as *flow_rate*, *open*, *rods_down*, and *current*, are left out of the design language. They represent state information that is only part of the running system and not of the static structure of the system. Since the design language consists solely of static information, attribute values of the above attributes are not visible, nor available, in the design language's instance.

The metamodel of the *run-time language* $NPPC_R$ (not shown) is similar. It does, however, include the aforementioned attributes such that the state of the system can be represented with this language. An instance of the run-time language looks similar to an instance of the design DSML (shown in Fig. 16.4), where the current state is marked and attribute values are shown.

Figure 16.15 shows the metamodel of the generated *output language* $NPPC_O$. The template represents an output *Trace* with *Transition*s between system *State*s. These *State*s contain *OutputElement*s representing a run-time state of the modelled system, such that a *Trace* represents a step-by-step execution trace. As the annotations dictate, all elements of the annotated metamodel are present in $NPPC_O$. Instances of the input ($NPPC_I$) and output ($NPPC_O$) languages (not shown) look like a sequence of run-

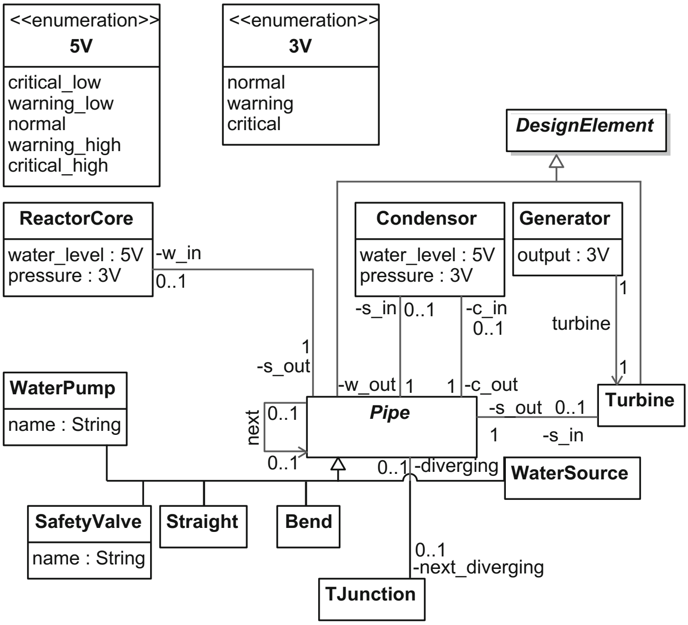


Fig. 16.14 The generated *NPPC* design language $NPPC_D$ (excerpt)

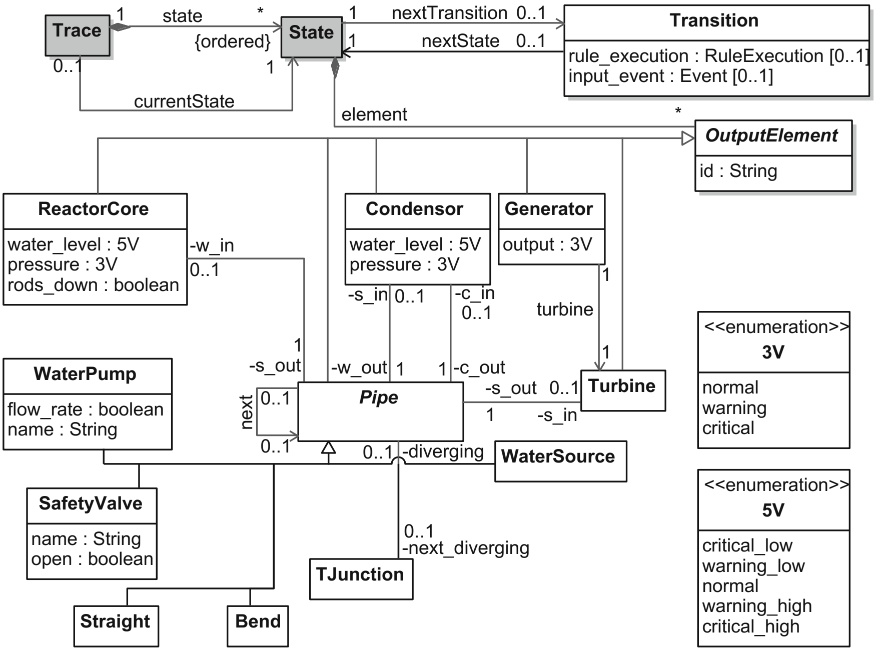


Fig. 16.15 The generated *NPPC* output language $NPPC_O$ (excerpt), shaded elements are part of the template

time language instances, but can also be left implicit due to spatial constraints. They can be "played out" step by step on the modelled system to replay the execution it represents.

The generated metamodel of *property language* $NPPC_P$, shown in Fig. 16.16, allows the definition of temporal properties over the system behaviour by means of four constructs:

- Quantification (∀ or ∃) allows the user to specify whether the following temporal property should apply to *all* or *at least one* match of a given pattern over the design model;
- Temporal patterns (based on Dwyer et al. 1999) allow the user to choose from a number of intuitive temporal operators over occurrences of a given pattern in a state of the modelled system, such as *after the reactor is in the* low water *state, it should eventually go back to the* normal *state*. These temporal patterns can be scoped (i.e. should only be checked) within certain occurrences of a given pattern;
- Structural patterns (based on Guerra et al. 2010, 2013) allow the user to specify patterns over a single state of the modelled system, such as *a component is in* warning *state*. Structural patterns can be composed, but are always evaluated on a single state of the modelled system;

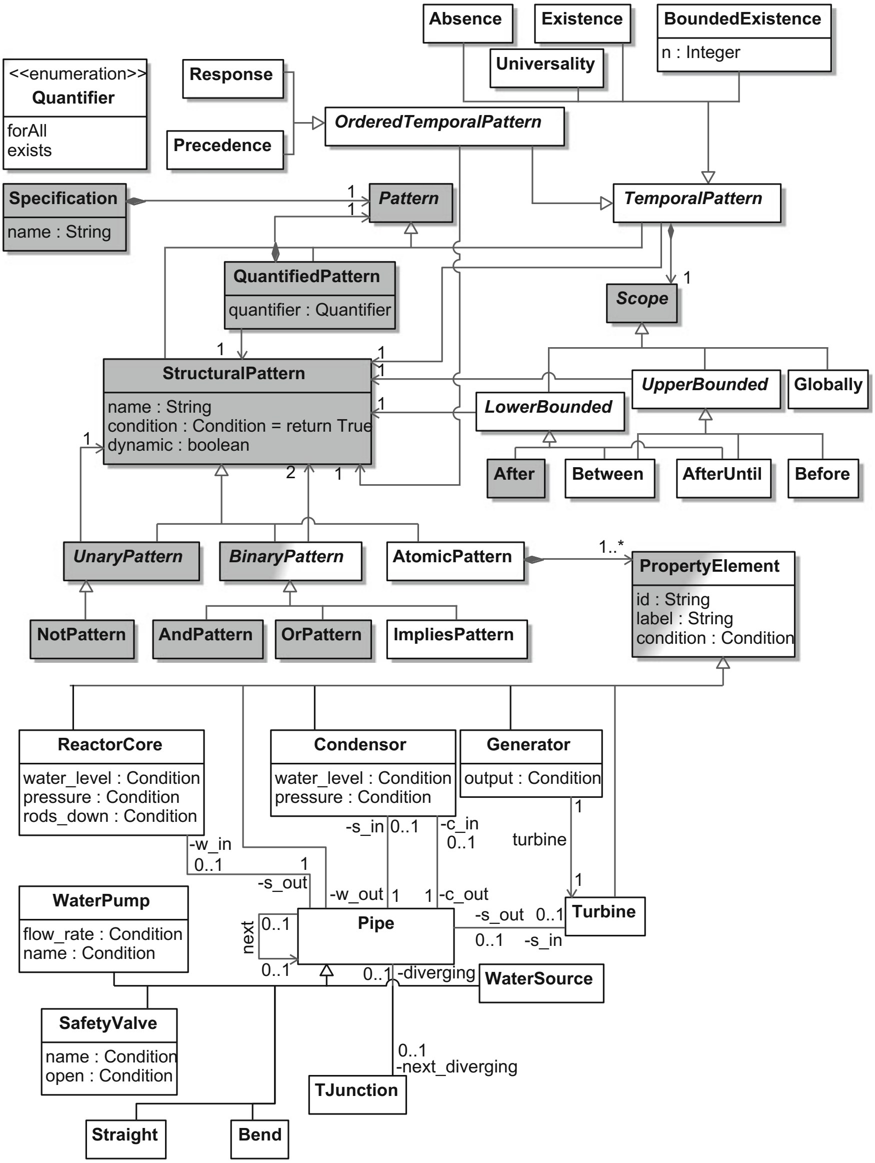


Fig. 16.16 The generated *NPPC* properties language *NPPC*$_P$ (excerpt), shaded elements are part of the template

- Instead of (a subset of) the DSML, a pattern language of the DSML (generated using the RAMification process (Kühne et al. 2009; Syriani 2011)) is used, such that domain-specific patterns rather than models can be specified. The effects of this are reflected in $NPPC_P$, where all attribute types are now *Condition*s (expressions that return a Boolean), all abstract classes are concrete, and the lower bounds of association cardinalities are relaxed.

The resulting language has a concrete syntax similar to the DSML, and the quantification and temporal patterns (instead of LTL operators) are raised to a more intuitive level through the use of natural language.

16.4.3 Specifying and Checking Properties Using ProMoBox

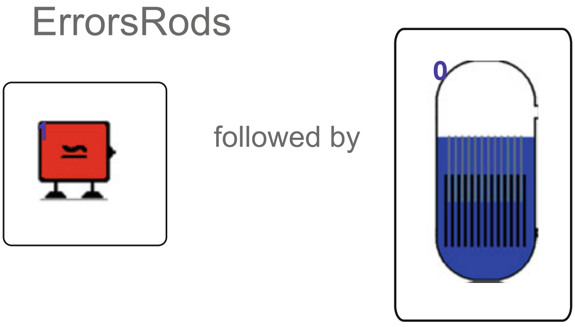
Figures 16.17 and 16.18 show properties specified in the property language $NPPC_P$. These patterns reuse domain-specific language elements to allow the specification of patterns and ultimately properties in a domain-specific syntax. Figure 16.17 shows a *Response* pattern, and Fig. 16.18 shows an *Existence* pattern, bounded by an *Upper* and *Lower* bound.

As shown in Fig. 16.12, properties specified in the property language are compiled to LTL, and the compiler produces a Promela model (Holzmann 1997) that includes a translation of the initialized system, the environment, and the rule-based operational semantics of the system. This translation is generic and thus independent of the DSML. The properties are checked by the SPIN model checker, which evaluates every possible execution path of the system. In case of Fig. 16.17, no counterexample is found, meaning that the system satisfies this property in every situation.

For the property modelled in Fig. 16.18, however, a counterexample is found which we will discuss in more detail. The property is translated to the following LTL formula: [](Q && !R -> (!R *W* (P && !R))) where:

- P = *generator.state* == *normal*
- Q = *generator.state* == *warning* && *reactor.state* == *rods_down*

Fig. 16.17 Property 1: *If the generator is in the error state, the rods will eventually be lowered*



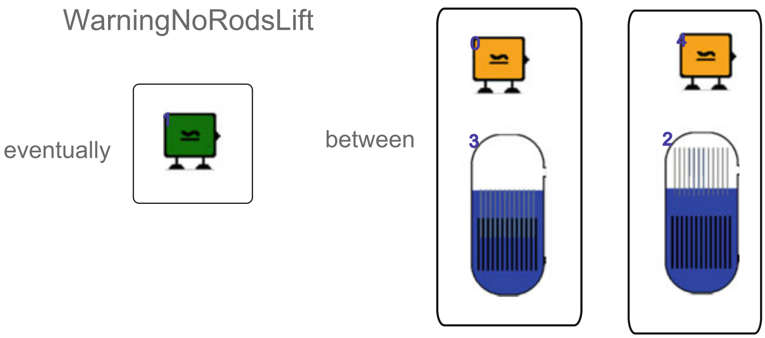


Fig. 16.18 Property 2: *If the generator is in the warning state, the rods can only become raised if the generator passed through the normal state*

- R = *generator.state == warning && reactor.state == rods_up*
- Operator *W* is "weak until" and defined as p *W* q = ([]p) || (p U q)

The counterexample found shows it is possible for two different components to go into the warning state, after which only one of them goes back to the normal state. If that happens, the controller will go back to the normal state, as there is no counter to store how many components are in the warning state. In this normal state, it is possible to raise the rods further, without having the generator go to the normal state again, causing the property to be violated. This counterexample can be played out with SPIN to produce a textual execution trace, which is translated back to the DSM level as an instance of the output language. This execution trace can be played out on the modelled system, by visualizing the states of the traces in the correct order.

The limitations of the framework are related to the mapping to Promela, as explained in Meyers et al. (2014). In its current state, *ProMoBox* does not allow dynamic structure models. Because of the nature of Promela and the compiler's design, boundedness of the state space is ensured in the generated Promela model.

16.5 Conclusion

In this chapter, we motivated the use of domain-specific modelling (DSM) for the development of complex, reactive, real-time, and software-intensive systems. Model-driven engineering, and in particular DSM, closes the conceptual gap between the problem domain and solution (implementation) domain.

We presented the different aspects of a domain-specific modelling language:

1. abstract syntax to define the allowable constructs;
2. concrete syntax to define the visual representation of abstract syntax constructs;
3. semantic domain to define the domain in which the semantics is expressed; and

4. semantic mapping to define the mapping to a model in the semantic domain that defines the (partial) semantics of the domain-specific model.

Each of these aspects was explained and applied to our case study: modelling the behaviour of a nuclear power plant controller and its subcomponents. The behaviour of the nuclear power plant interface is defined using both operational semantics ("simulation") and translational semantics ("mapping").

We extended our discussion to property checking, which was also lifted to the level of the problem domain. This enables domain experts to not only create models and reason about them, but also specify properties on them. Errors, or any other kind of feedback, are also mapped back onto the problem domain, meaning that the domain expert never has to leave his domain. All aspects of modelling were thus pulled up from the solution domain, up to the problem domain, closing any conceptual gaps. Moreover, a generative approach is used, in order to limit development overhead.

Acknowledgements This work was partly funded by a PhD fellowship from the Research Foundation - Flanders (FWO) and the Agency for Innovation by Science and Technology in Flanders (IWT). Partial support by the Flanders Make strategic research centre for the manufacturing industry is also gratefully acknowledged.

References

Barišić A, Amaral V, Goulão M, Barroca B (2011) Quality in use of domain-specific languages: a case study. In: Proceedings of the 3rd ACM SIGPLAN workshop on evaluation and usability of programming languages and tools, ACM, PLATEAU'11, pp 65–72

Boulanger F, Hardebolle C, Jacquet C, Marcadet D (2011) Semantic adaptation for models of computation. In: 2011 11th International conference on application of concurrency to system design (ACSD), pp 153–162. doi:10.1109/ACSD.2011.17

Cellier FE (1991) Continuous system modeling. Springer

Costagliola G, Deufemia V, Polese G (2004) A framework for modeling and implementing visual notations with applications to software engineering. ACM Trans Soft Eng Methodol 13(4):431–487

Dwyer MB, Avrunin GS, Corbett JC (1999) Patterns in property specifications for finite-state verification. In: International conference software engineering, pp 411–420

Guerra E, de Lara J, Kolovos DS, Paige RF (2010) A visual specification language for model-to-model transformations. In: VL/HCC

Guerra E, de Lara J, Wimmer M, Kappel G, Kusel A, Retschitzegger W, Schönböck J, Schwinger W (2013) Automated verification of model transformations based on visual contracts. Autom Soft Eng 20(1):5–46. doi:10.1007/s10515-012-0102-y

Harel D (1987) Statecharts: a visual formalism for complex systems. Sci Comput Program 8(3):231–274

Harel D, Naamad A (1996) The STATEMATE semantics of statecharts. ACM Trans Softw Eng Methodol 5(4):293–333

Harel D, Rumpe B (2004) Meaningful modeling: what's the semantics of "semantics"? Computer 37(10):64–72

Holzmann GJ (1997) The model checker SPIN. Trans Softw Eng 23(5):279–295

Kelly S, Tolvanen JP (2008) Domain-specific modeling: enabling full code generation. Wiley

Kleppe A (2007) A language description is more than a metamodel. In: Fourth international workshop on software language engineering

Kühne T (2006) Matters of (meta-) modeling. Softw Syst Model 5:369–385

Kühne T, Mezei G, Syriani E, Vangheluwe H, Wimmer M (2009) Explicit transformation modeling. In: MoDELS workshops. Lecture notes in computer science. Springer, 6002:240–255

Lucio L, Mustafiz S, Denil J, Vangheluwe H, Jukss M (2013) FTG+PM: an integrated framework for investigating model transformation chains. In: SDL 2013: model-driven dependability engineering. Lecture notes in computer science, vol 7916. Springer, pp 182–202

Meyers B, Vangheluwe H (2014) A multi-paradigm modelling approach for the engineering of modelling languages. In: Proceedings of the doctoral symposium of the ACM/IEEE 17th international conference on model driven engineering languages and systems, CEUR Workshop Proceedings, pp 1–8

Meyers B, Wimmer M, Vangheluwe H, Denil J (2013) Towards domain-specific property languages: the ProMoBox approach. In: Proceedings of the 2013 ACM workshop on domain-specific modeling. ACM, New York, NY, USA, DSM'13, pp 39–44. doi:10.1145/2541928.2541936

Meyers B, Deshayes R, Lucio L, Syriani E, Vangheluwe H, Wimmer M (2014) ProMoBox: a framework for generating domain-specific property languages. In: Software language engineering. Lecture notes in computer science, vol 8706. Springer International Publishing, pp 1–20

Moody D (2009) The "physics" of notations: toward a scientific basis for constructing visual notations in software engineering. IEEE Trans Softw Eng 35(6):756–779

Murata T (1989) Petri nets: properties, analysis and applications. Proc IEEE 77(4):541–580

Petre M (1995) Why looking isn't always seeing: readership skills and graphical programming. Commun ACM 38(6):33–44

Pnueli A (1977) The temporal logic of programs. In: Proceedings of the 18th annual symposium on foundations of computer science. IEEE computer society, Washington, DC, USA, SFCS'77, pp 46–57. doi:10.1109/SFCS.1977.32

Risoldi M (2010) A methodology for the development of complex domain-specific languages. PhD thesis, University of Geneva. http://archive-ouverte.unige.ch/unige:11842

Sendall S, Kozaczynski W (2003) Model transformation: the heart and soul of model-driven software development. IEEE Softw 20(5):42–45

Smith MH, Holzmann GJ, Etessami K (2001) Events and constraints: a graphical editor for capturing logic requirements of programs. In: Proceedings of the fifth IEEE international symposium on requirements engineering. IEEE computer society. Washington, DC, USA, RE'01, pp 14–22. http://dl.acm.org/citation.cfm?id=882477.883639

Syriani E (2011) A multi-paradigm foundation for model transformation language engineering. PhD thesis, McGill University, Canada

Syriani E, Vangheluwe H (2013) A modular timed graph transformation language for simulation-based design. Softw Syst Model 12(2):387–414

Syriani E, Vangheluwe H, Mannadiar R, Hansen C, Van Mierlo S, Ergin H (2013) AToMPM: a web-based modeling environment. In: Proceedings of MODELS'13 demonstration session, pp 21–25

Vangheluwe H (2008) Foundations of modelling and simulation of complex systems. ECEASST 10

Zeigler BP, Praehofer H, Kim TG (2000) Theory of modeling and simulation, 2nd edn. Academic Press

Zhang Y, Xu B (2004) A survey of semantic description frameworks for programming languages. SIGPLAN Not 39(3):14–30

Chapter 17
Exploiting Action Theory as a Framework for Analysis and Design of Formal Methods Approaches: Application to the CIRCUS Integrated Development Environment

Camille Fayollas, Célia Martinie, Philippe Palanque, Eric Barboni, Racim Fahssi and Arnaud Hamon

Abstract During early phases of the development of an interactive system, future system properties are identified (through interaction with end-users, e.g., in the brainstorming and prototyping phases of the development process, or by requirements provided by other stakeholders) imposing requirements on the final system. Some of these properties rely on informal aspects of the system (e.g. satisfaction of users) and can be checked by questionnaires, while others require the use of verification techniques over formal models (e.g. reinitializability) or validation techniques over the application (e.g. requirements fit). Such properties may be specific to the application under development or generic to a class of applications, but in all cases verification and validation tools are usually required for checking them. The usability of these tools has a significant impact on the validation and verification (V&V) phases which usually remain perceived as very resource consuming. This chapter proposes a user–centred view on the use of formal methods, especially on the use of formal modelling tools during the development process of an interactive system. We propose to apply Norman's action theory to the engineer tasks of formal modelling in

C. Fayollas (✉) · C. Martinie · P. Palanque · E. Barboni · R. Fahssi · A. Hamon
ICS-IRIT, University of Toulouse, 118 Route de Narbonne, 31062 Toulouse, France
e-mail: fayollas@irit.fr

C. Martinie
e-mail: martinie@irit.fr

P. Palanque
e-mail: palanque@irit.fr

E. Barboni
e-mail: barboni@irit.fr

R. Fahssi
e-mail: fahssi@irit.fr

A. Hamon
e-mail: hamon@irit.fr

B. Weyers et al. (eds.), *The Handbook of Formal Methods in Human-Computer Interaction*, Human-Computer Interaction Series,
DOI 10.1007/978-3-319-51838-1_17

order to analyse how modelling tools impact the following engineer activities: model editing, model verification and model validation. The CIRCUS Integrated Development Environment serves as an illustrative example to exemplify how engineer tasks may be impacted by modelling tools features.

17.1 Introduction

Nowadays, interactive applications are deployed in more and more complex command and control systems including safety critical ones. Dedicated formalisms, processes and tools are thus required to bring together various properties such as reliability, dependability and operability. In addition to standard properties of computer systems (such as safety or liveness Pnueli 1986), interaction-related properties have been identified. Properties related to the use of an interactive system are called external properties (Cockton and Gram 1996) and characterize the capacity of the system to provide support for its users to accomplish their tasks and goals, and prevent or help recover from errors. Although all types of properties are not always completely independent from another (some might also be conflicting), external properties are related to the user's point of view and the usability factor (International Standard Organization 1996), whereas internal properties are related to the design and construction of the system itself (e.g. modifiability, run-time efficiency). Interactive systems have to support both types of properties. Dedicated techniques and approaches have been studied for this purpose, among which are formal description techniques. Formal description techniques have proven their value in several domains and are required to understand, design and develop interactive systems and check their properties.

Formal description techniques have been studied within the field of human–computer interaction as a means to describe and analyse, in a complete and unambiguous way, interactions between a user and a system. Several types of approaches have been developed (Dix 1991), which encompass contributions about formal description of an interactive system's behaviour and/or formal verification of its properties.

The use of formal methods depends on the phase of the development process describing the activities necessary to develop the interactive system under consideration. In the case of critical interactive systems, Martinie et al. (2012) proposes a development process relying on formal methods and taking into account both interactive and critical aspects of such systems which necessitates several formal descriptions used by different user types (system designers, human factor specialists, etc.). Consequently, the usability of tools for validation and verification of these interactive systems target will depend on their users' activity in the development process.

Whether being used as a means for describing in a complete and unambiguous way the interactive system or as a means of verifying properties, formal description techniques and their associated tools need to be designed to be usable and not error prone.

This chapter, extending and refining previous work presented in (Barboni et al. 2003), proposes a user-centred view on the use of formal methods, especially on the use of formal modelling tools during the development process of an interactive system. We propose to apply Norman's action theory to the engineer's tasks of formal modelling in order to analyse how modelling tools impact the following engineer activities: model editing, model verification and model validation.

The chapter is structured as follows. The next section details engineer's tasks related to the production of models, as well as Norman's action theory (Norman 2013) and its seven stages. Section 17.3 presents the CIRCUS Integrated Development Environment that will be used as an illustrative example, throughout the chapter, to show how the modelling tasks are performed as well as the impact of the modelling tools on them. Sections 17.4, 17.5 and 17.6 present how the application of the Norman's action theory to the main engineer's tasks (editing, verification, validation) can be used to investigate effectiveness and efficiency issues. Still relying on Norman's action theory, Sect. 17.7 presents how the synergistic use of models can provide additional support to validation tasks. Section 17.8 discusses how modelling tools can enhance effectiveness and efficiency by helping the engineer in accomplishing several goals at once. Section 17.9 concludes the chapter and presents some perspectives for this work.

17.2 A Global View on Modelling Activities During the Development of Interactive Systems

This section presents the main goals that have to be achieved by the user of the development tools (thus the engineer) to produce models during interactive systems development phases. Norman's action theory is then used as a framework to analyse whether or not modelling tools provide support for these tasks.

17.2.1 Engineer's Tasks When Developing Interactive Systems

Several tasks have to be performed when producing models during the development of an interactive system. Editing of the models, verification of the edited models and validation of the models are the main ones. Figure 17.1 depicts how these main tasks are organized during the development process of the models that are used to design and develop interactive systems.

In Fig. 17.1, at the beginning of the process, the scope of the modelling activities has to be identified in order to determine which elements (related to the interactive system to be developed) will be modelled and which types of models will be used. Then, the user/engineer has to analyse the extant system, as well as the needs and

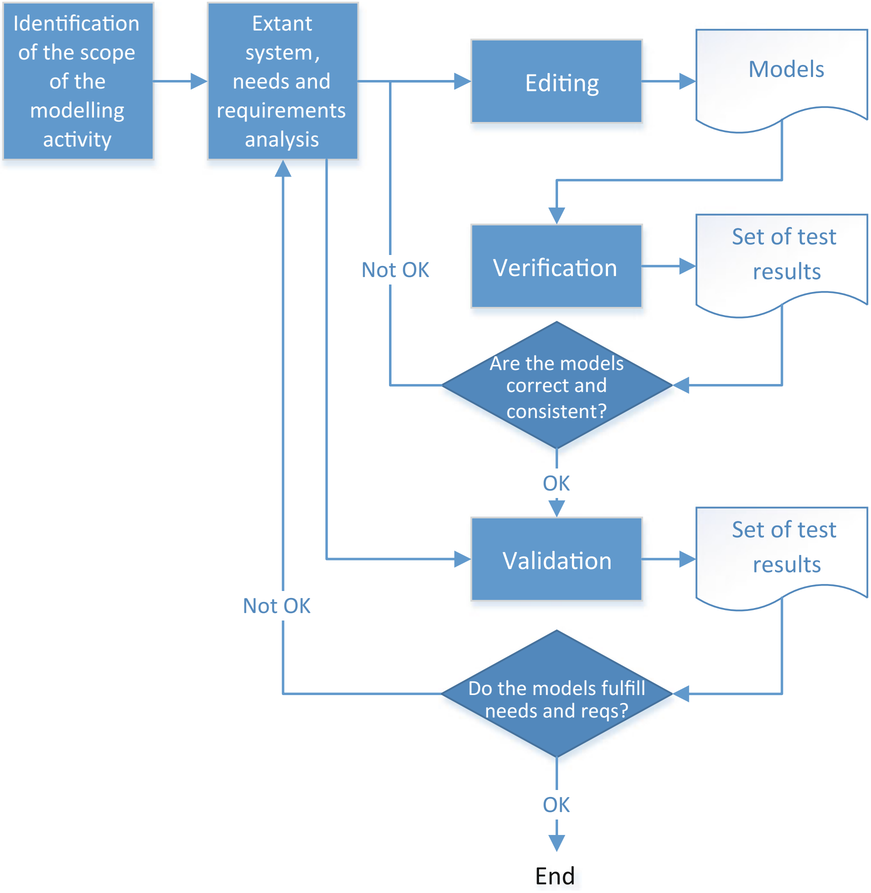

Identification of the scope of the modelling activity
Extant system, needs and requirements analysis
Editing
Models
Verification
Set of test results
Not OK
Are the models correct and consistent?
OK
Validation
Set of test results
Not OK
Do the models fulfill needs and reqs?
OK
End

Fig. 17.1 The process of producing models during the development of an interactive system

requirements for the interactive system to be developed. After this analysis step, the production of the models starts with the editing step. That step produces a first version of models. The verification step is then performed for checking the models in order to ensure that they are correct and consistent. If they are not the editing step has to be performed again to amend the models. This editing-verification loop will be performed again until the models are fully correct and consistent.

The validation step will then be initiated to check whether or not the model meets specified needs and requirements. If this is not the case, the user/engineer will have to review the needs and requirements and to correct the models by going back to the editing step, as well as editing-verification loop until the new version of the models is correct and consistent. This new version of the models will then be reassessed against needs and requirements during another validation step. This editing–verification–validation loop will be performed again until the models are complete, consistent and meet needs and requirements.

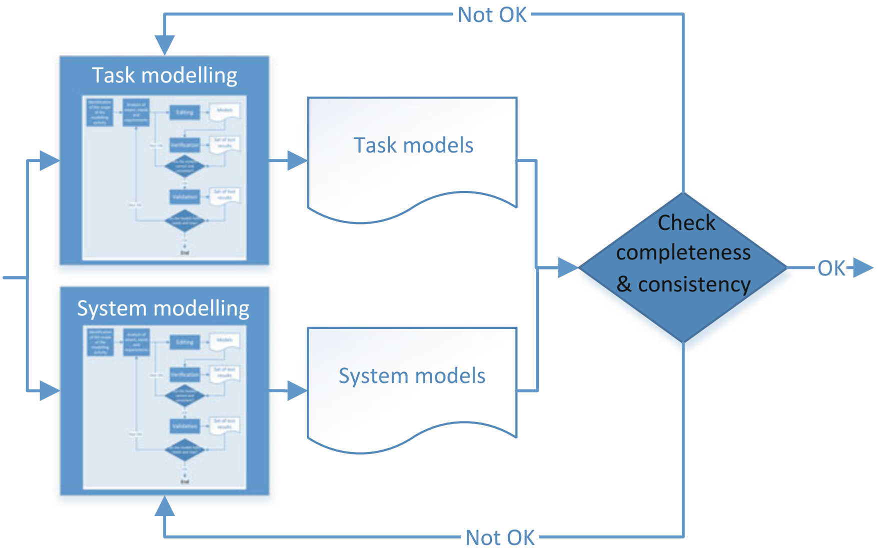


Fig. 17.2 Process for the combined production of task and system models

The editing, verification and validation steps will be detailed in Sects. 17.4, 17.5 and 17.6, respectively, and will be illustrated with the examples of modelling notations and tools. In particular, we will focus on the system and task modelling notations as they are the key types of notations that provide support for designing and developing interactive systems.

Figure 17.2 presents the process for the combined production of task and system models. This process is based on task modelling and system modelling. Task modelling may have been started before system modelling (during the preliminary task analysis phase). These two modelling activities are done following the process presented in Fig. 17.1 and lead to the production of task models and system models. The process then continues with the cross-validation of task and system models in order to check whether they are complete and consistent. When both task and system models are consistent and complete, the development process of the interactive system can go on (e.g. code generation, user testing). This particular cross-model validation step will be detailed in Sect. 17.8.

17.2.2 Norman's Action Theory and Its Application to Models Production Tasks

Figure 17.3 presents the seven stages of Norman's action theory (Norman 2013). The left-hand side represents the execution path which is composed of a set of activities that have to be carried out by the user in order to reach her/his goal. The right-hand

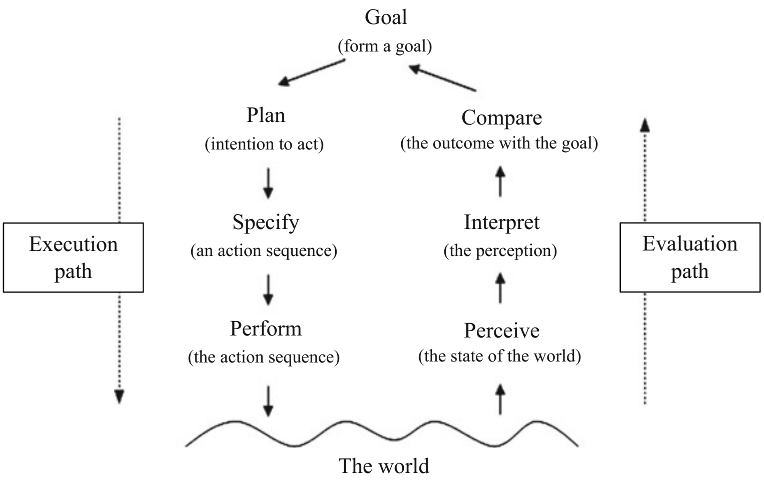


Fig. 17.3 The seven stages of Norman's action theory

side represents the evaluation path which is composed of a set of activities that has to execute by the user in order to interpret the real world state and how it was affected by her/his action.

This sequence of stages is iterative: the user will perform the depicted sequence of activities until the goal is reached. The user will act on the real world and interpret the result of the actions performed and the changes it implied in the real world. The user will then compare the perceived state and the one desired (which is part of his/her goal) and decide (if the goal is not reached) to perform another action. This will be done iteratively until the goal is reached or until the user gives up.

Figure 17.3 also depicts the main difficulties that the users may face when they interact with a system. The gulf of execution is an indicator about how the system support intended user actions. The gulf of evaluation is an indicator about how the information provided by the system helps the user in assessing the system's state. Therefore, reducing these gulfs (and thus reducing the gap between the different actions) helps in enhance the usability of the device.

Figure 17.4 presents the application of the seven stages of Norman's action theory to modelling work. That figure shows the refinement of specific activities to take into account modelling tasks. The two main stages we are focusing on are:

- On the execution side, the activity of going from an intention to its actual transcription into some model and
- On the evaluation side, the activity of perceiving model behaviour and interpreting this perception.

One of the main advantages of this model is that it provides a generic framework for investigating where the main difficulties can occur during system modelling

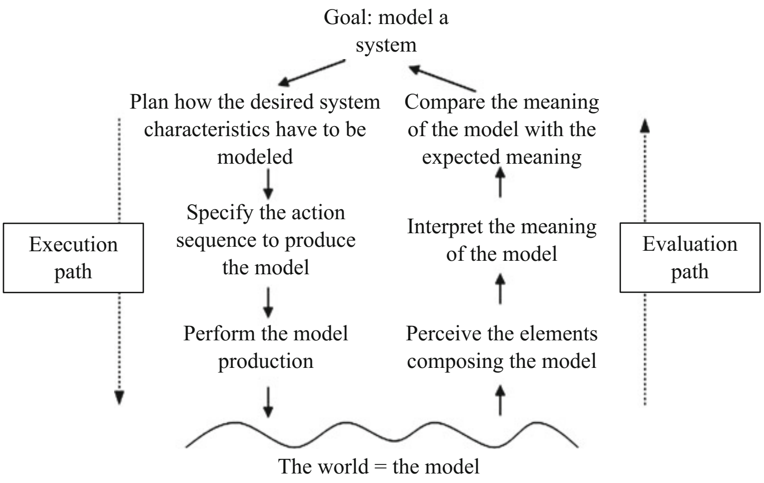

Goal: model a system
Plan how the desired system characteristics have to be modeled
Specify the action sequence to produce the model
Perform the model production
Compare the meaning of the model with the expected meaning
Interpret the meaning of the model
Perceive the elements composing the model
Execution path
Evaluation path
The world = the model

Fig. 17.4 Formal modelling within Norman's action theory

Table 17.1 The template used to describe the application of Norman's action theory to the different engineer task during model's development and cross-model validation

	Evaluation path				Execution path		
Engineer task	Perceive	Interpret	Compare	Goal	Plan	Specify	Perform

which is a highly demanding task on the engineer's side. This framework thus provides design rules for environments to support user's activities and reduce their difficulties.

Norman's action theory provides a generic framework for investigating where the main difficulties can occur during system modelling activities performed by an engineer. Each engineer task described in the previous section (editing, verification, validation and cross-type validation of models) is a goal that has to be accomplished during the development of interactive systems. And for each of these goals, the seven stages of Norman's action theory can be used to analyse the low-level activities that the engineer will execute. Sections 17.4, 17.5, 17.6 and 17.7, respectively, present these low-level activities for the editing, verification, validation and cross-type goals. Table 17.1 is a template that will be filled in for each goal (presented in each respective section).

The following questions (from Norman's action theory Norman 2013) will be used to fill in the table:

- Perceive: What happened?
- Interpret: What does it mean?
- Compare: Is this okay? Have I accomplished my goal?

- Goal: What do I want to accomplish?
- Plan: What are the alternative action sequences?
- Specify: What action can I do now?
- Perform: How do I do it?

17.3 Illustrative Example: The CIRCUS Integrated Development Environment

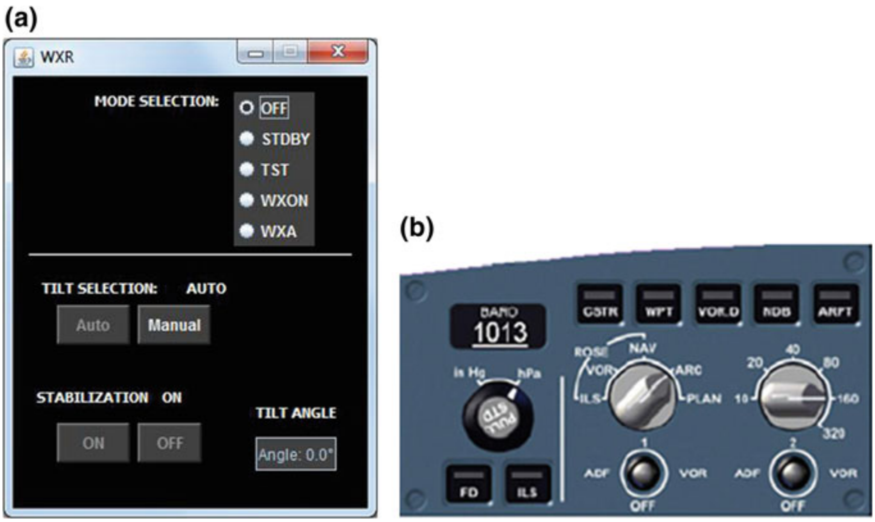
This section presents the CIRCUS integrated development environment that will be used as an illustrative example, throughout this chapter, to show how the modelling tasks are performed, as well as the impact of the modelling tools on them. In order to illustrate these tasks, we use a small system as an illustrative example: the weather radar application (called WXR application). This illustrative example will be used to perform the task modelling and system modelling activities, thus helping to highlight how the modelling tools have an impact on modelling tasks, according to the seven stages of Norman's action theory.

The first subsection presents the WXR application. The second subsection presents the formal modelling of this application (both task and system modelling). Finally, the last subsection presents the three tools composing the CIRCUS integrated development environment (for task modelling, system modelling and synergistic validation of both task and system models).

17.3.1 The Weather Radar Application Description

The weather radar (WXR) presented in this chapter is built upon an application currently deployed in many cockpits of commercial aircraft. WXR provides support to pilots' activities by increasing their awareness of meteorological phenomena during the flight journey. It allows them to determine the weather ahead of the aircraft which might end up in requesting a trajectory change, in order to avoid storms or precipitations, for example.

Figure 17.5 presents a screenshot of the weather radar control panel, used to operate the weather radar application. This panel provides the crew with two functionalities. The first one (see Fig. 17.5a) is dedicated to the mode selection of the weather radar and provides information about the radar status, to ensure that it can be set up correctly. The mode change can be performed in the upper part of the panel. The second functionality, available in the lower part of the window, is dedicated to the adjustment of the weather radar orientation (Tilt angle). This can be done in an automatic or a manual way (auto/manual buttons). Additionally, a stabilization function aims at keeping the radar beam stable even in case of turbulence. Figure 17.5b presents an image of the controls used to configure radar



WXR

MODE SELECTION: ○ OFF ● STDBY ● TST ● WXON ● WXA

TILT SELECTION: AUTO

Auto Manual

STABILIZATION ON

ON OFF

TILT ANGLE

Angle: 0.0°

Fig. 17.5 Image of **a** the weather radar control panel **b** of the radar display manipulation

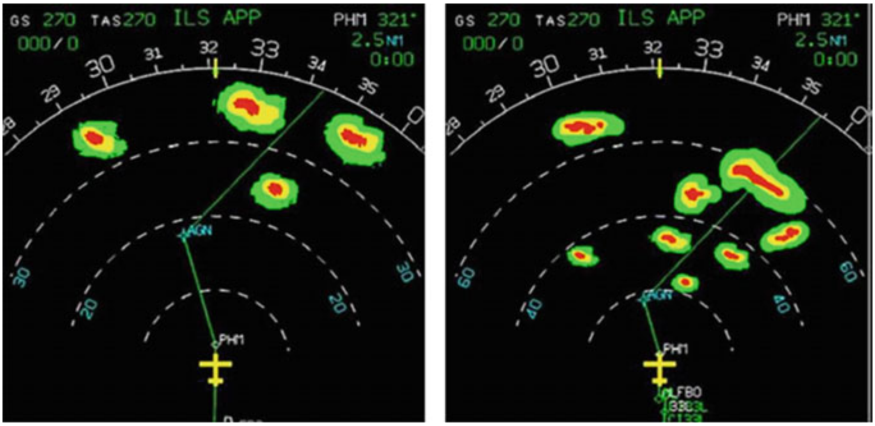


Fig. 17.6 Screenshot of weather radar displays

display, particularly to set up the range scale (right-hand side knob with ranges 20, 40, … nautical miles).

Figure 17.6 shows screenshots of weather radar displays according to two different range scales (40 NM for the left display and 80 NM for the right display). Spots in the middle of the images show the current position, importance and size of the clouds.

17.3.2 Formal Modelling of the WXR Application

This section presents the task modelling and system modelling of the WXR application. In order to simplify the reading, each subsection is divided in two parts: first the notation used is presented and then the model corresponding to the WXR application is presented.

17.3.2.1 Task Modelling

This section introduces the task modelling of the WXR application. For this purpose, we first quickly introduce and then present the HAMSTERS model of the WXR application.

Brief Presentation of HAMSTERS Notation

The HAMSTERS notation (Human-centred Assessment and Modelling to Support Task Engineering for Resilient Systems) has initially been designed to provide support for ensuring consistency, coherence and conformity between user tasks and interactive systems at the model level (Barboni et al. 2010). It has then been further enhanced and now encompasses notation elements such as a wide range of specialized tasks types, data and knowledge explicit representations, device descriptions, genotype and phenotypes of errors and collaborative tasks.

The HAMSTERS notation enables structuring users' goals and subgoals into a hierarchical tasks tree in which qualitative temporal relationship among tasks are described by operators (Martinie et al. 2011). The output of this decomposition/refinement is a graphical tree of nodes that can be tasks or temporal operators.

Table 17.2 Task types in HAMSTERS

	Abstract	Input	Output	I/O	Processing
Abstract	Abstract	Not Applicable	Not Applicable	Not Applicable	Not Applicable
User	User abstract	Perceptive	Motor	User	Cognitive
Interactive	Abstract interactive	Input	Output	Input/Output	Not Applicable
System	Abstract system	Output	Input	Input/Output	System

Tasks can be of several types and contain information such as a name, information details and criticality level. The different types of tasks in HAMSTERS are depicted in Table 17.2 (from top to bottom and left to right):

- *Abstract tasks*: tasks involving subtasks of any types;
- *System tasks*: tasks performed only by the system;
- *User tasks*: tasks describing a user activity that can be specialized as perceptive tasks (e.g. reading some information), cognitive task (e.g. comparing value, remembering information), or motor tasks (e.g. pressing a button);
- *Interactive tasks*: tasks describing an interaction between the user and the system that can be refined into input task when the users provide input to the system, output task when the system provides an output to the user and input/output task (both but in an atomic way);

It is important to note that only the single-user high-level task types are presented here, but they can be further refined. For instance, the cognitive tasks can be refined in analysis and decision tasks (Martinie et al. 2011a, b) and collaborative activities can be refined in several task types (Martinie et al. 2014).

Temporal operators are used to represent temporal relationships between subgoals and between activities. HAMSTERS temporal operators are depicted and defined in Table 17.3; they are similar to the ones proposed in (Wilson et al. 1993) and then reused and extended in CTT (Paternò 1999).

Tasks can also be tagged by properties to indicate whether or not they are iterative, optional or both (for instance, Fig. 17.7 presents two iterative tasks: the "manage modes" and "change mode" abstract tasks). HAMSTERS expressive power goes beyond most other tasks modelling notations particularly by providing detailed means for describing data that is required and manipulated in order to accomplish tasks (Martinie et al. 2013).

The HAMSTERS notation is supported by a CASE tool for editing and simulation of models. This tool has been introduced in order to provide support for task system integration at the tool level (Martinie et al. 2014). The HAMSTERS tool and notation also provide support for structuring a large number and complex set of tasks introducing the mechanism of subroutines, submodels and components (Forbrig et al. 2014). Such structuring mechanisms allow describing large and

Table 17.3 Illustration of the operator types within HAMSTERS

Operator type	Symbol	Description
Enable	T1 ≫ T2	T2 is executed after T1
Concurrent	T1 ||| T2	T1 and T2 are executed at the same time
Choice	T1 [] T2	T1 is executed OR T2 is executed
Disable	T1 [> T2	Execution of T2 interrupts the execution of T1
Suspend-resume	T1 | > T2	Execution of T2 interrupts the execution of T1, T1 execution is resumed after T2
Order independent	T1 | = | T2	T1 is executed then T2 OR T2 is executed then T1

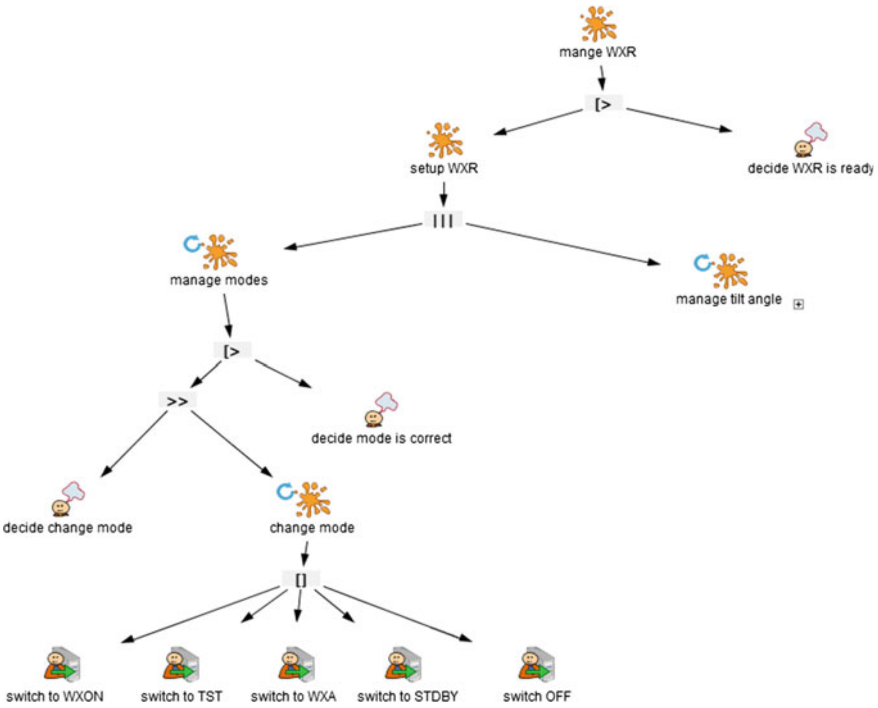


Fig. 17.7 Task model of the manage WXR application activity

complex activities by means of task models. These structuring mechanisms enable the breakdown of a task model in several ones that can be reused in the same or different task models.

HAMSTERS Task Model of the WXR Application

The HAMSTERS task model, corresponding to the WXR application, describing the crew activities when managing the WXR application, is depicted in Fig. 17.7 (due to space constraints, the managed tilt angle subparts are folded, as shown by the ⊞ symbol). When managing the modes of the WXR application, the crew members can decide to change the mode ("decide change mode" cognitive task). This activity is followed by choosing a mode ("change mode" iterative abstract task). This task is refined by the choice between the five WXR modes (five interactive input tasks corresponding to the choice of one of the five WXR modes). After choosing a new mode, the crew members can decide that they are satisfied by the activated mode ("decide mode is correct" cognitive task) or iterate on the same task and select another mode.

17.3.2.2 System Modelling

This section introduces the system modelling of the WXR application. For this purpose, we first quickly introduce the ICO notation and then present the ICO model of the WXR application.

Brief Presentation of ICO Notation

The ICO notation (Navarre et al. 2009) (Interactive Cooperative Objects) is a formal description technique devoted to specify interactive systems. Using high-level Petri nets for dynamic behaviour description, the notation also relies on an object-oriented approach (dynamic instantiation, classification, encapsulation, inheritance and client/server relationships) to describe the structural or static aspects of systems. This notation has been applied to formally specify interactive systems in the fields of air traffic management (Navarre et al. 2009), satellite ground systems (Palanque et al. 2006), and cockpits of military (Bastide et al. 2004) and civil aircraft (Barboni et al. 2006a; Hamon et al. 2013).

A complete definition of the ICO notation can be found in (Navarre et al. 2009). However, we describe here the notation elements that are necessary to understand the ICO model of the WXR application presented here after: the high-level Petri net-based behavioural description of the system. These elements are presented in Fig. 17.8.

In few words, a Petri net is an oriented graph composed of a set of places (usually represented by ellipses, e.g. places *p0*, *p1* and *p2* in Fig. 17.8) which symbolize states variables holding tokens which symbolize values and a set of transitions (usually represented by rectangles, e.g. transitions *t0* and *t1* in Fig. 17.8) which symbolize actions and state changes. A set of arcs connects places to transitions (called input arcs, e.g. the arc connecting place *p0* to transition *t0*) or transitions to places (called output arcs, e.g. the arc connecting transition *t0* to place *p2*) and represents the flow of tokens through the Petri net model. The state of the modelled system is represented by the distribution of tokens across the set of places (called marking). The dynamic behaviour of a marked Petri net is expressed by means of two rules, the enabling rule and the firing rule. A transition is enabled if each input place (places linked to the transition with input arcs) contains at least as many tokens as the weight of the input arcs it is linked to. When an enabled transition is fired, tokens are removed from the input places of the transition and new tokens are deposited into the output places (the number of tokens is defined by the weight of the output arcs).

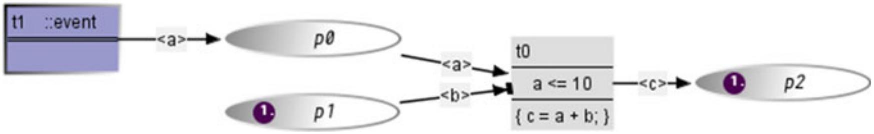


Fig. 17.8 Example of an ICO model

The ICO notation is based on high-level Petri nets and more particularly on Object Petri nets (Lakos and Keen 1991; Valk 1998); ICO models are Petri net in terms of structure and behaviour, but they hold a much more complex set of information. For instance, in the ICO notation, tokens in the places are comprised of a typed set of values that can be references to other objects (allowing, for instance, use in a transition of method calls) or preconditions about the actual value of some attributes of the tokens. For instance, in Fig. 17.8, the places *p0*, *p1* and *p2* comprises an integer, respectively, named *a*, *b* and *c*. The Object Petri nets formalism also allows the use of test arcs that are used for testing presence and values of tokens without removing them from the input place while firing the transition (e.g. the arc connecting place *p1* to transition *t0* in Fig. 17.8).

The ICO notation also encompasses multicast asynchronous communication principles that enable, upon the reception of an event, the firing of some transitions. For instance, the transition *t1* in Fig. 17.8 is fired upon the receipt of an event named *event*.

ICO System Model of the WXR Application

The ICO model of the WXR application is presented in Fig. 17.9. It describes how it is possible to handle both mode and tilt angle of the weather radar application.

Figure 17.9 shows the interactive means provided to the end-user of the application:

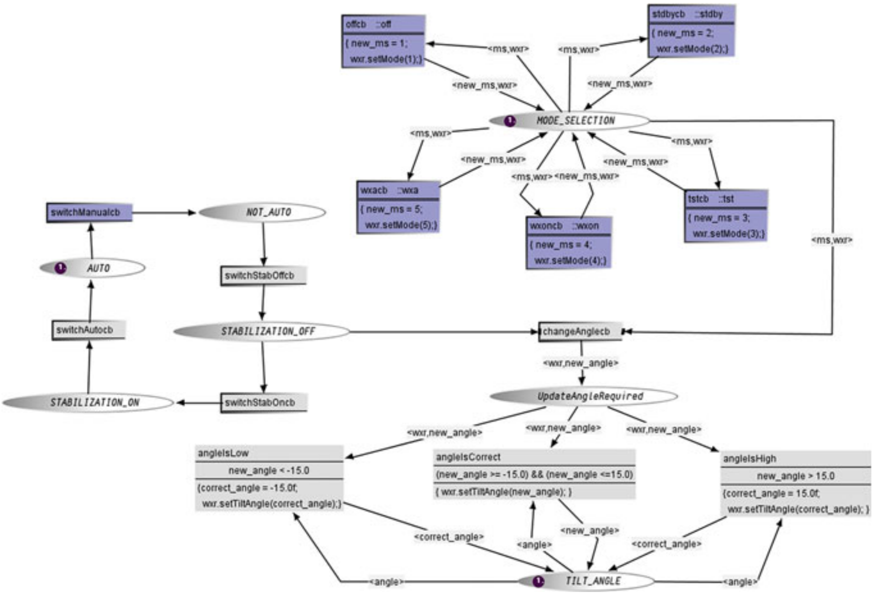


Fig. 17.9 ICO behaviour model of the WXR mode selection and tilt angle setting

- Switch between the five available modes (upper part of the figure) using radio buttons (the five modes being WXON to activate the weather radar detection, OFF to switch it off, TST to trigger a hardware check-up, STDBY to switch it on for test only and WXA to focus detection on alerts).
- Select the tilt angle control mode (lower part of the figure) among three modes (fully automatic, manual with automatic stabilization and manual selection of the tilt angle).

17.3.3 Tools Within the CIRCUS Integrated Development Environment

CIRCUS, which stands for Computer-aided design of Interactive, Resilient, Critical and Usable Systems, is an integrated development environment embedding both system and task modelling functionalities. The CIRCUS environment targets the following user types: engineers, system designers and human factors specialists. It aims at helping them to achieve their specific tasks during the design and development of interactive critical systems. CIRCUS embeds features for the formal verification of the system's behaviour as well as features for assessment of compatibility between the user's task and the system's behaviour.

The CIRCUS environment integrates three tools: the HAMSTERS tool (for task modelling), the PetShop tool (for systems modelling) and the SWAN tool (for synergistic validation of both task and system models). This section presents these three tools that will be used as subjects to investigate their usability issues with respect to each stage of the Norman's Action Theory.

17.3.3.1 HAMSTERS Tool

The HAMSTERS notation is supported by a CASE tool, named the same, for editing and simulation of models. This tool has been introduced in order to provide support for task system integration at the tool level (Martinie et al. 2014). The HAMSTERS tool and notation also provide support for structuring a large number and complex set of tasks introducing the mechanism of subroutines, submodels and components (Forbrig et al. 2014). Such structuring mechanisms allow describing large and complex activities by means of task models. These structuring mechanisms enable the breakdown of a task model in several ones that can be reused in the same or different task models.

As task models can be large, it is important to provide the analyst with computer-based tools for editing task models and for analysing them. To this end, the HAMSTERS task modelling tool provides support for creating, editing, and simulating the execution of task models.

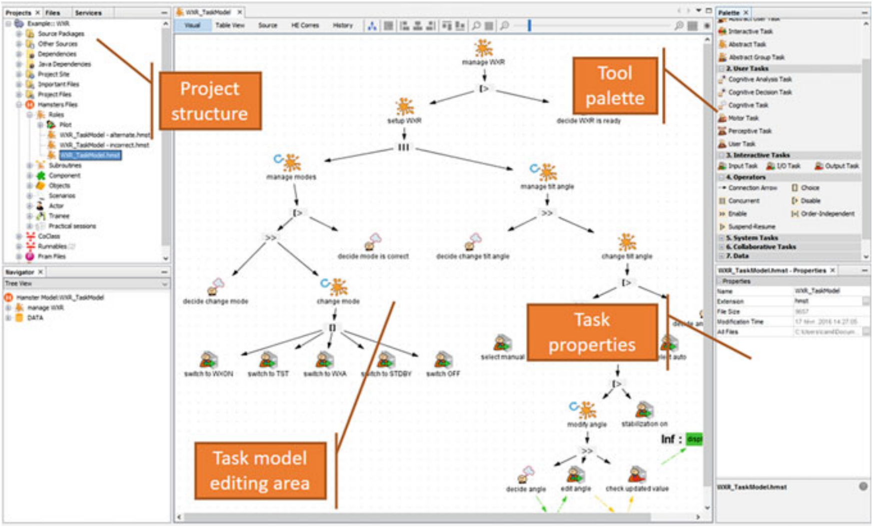


Fig. 17.10 Task model editing

Editing

As depicted in Fig. 17.10, the HAMSTERS software tool for editing task models is composed of three main areas:

- On the left-hand side, the project exploratory,
- In the centre, the task model editing area and
- On the right-hand side, the palette containing task types and temporal ordering operators.

Visual notation elements can be added to the task model by a drag and drop operation from the palette to the task model area (Fig. 17.10).

The task properties panel provides support, in the bottom left in Fig. 17.10, and provides support for viewing and editing task properties (such as estimated minimum and maximum time).

Simulation

The execution of task models can be launched from the project panel, by right-clicking on a task model and then by selecting the “Run simulator” menu option. Once this menu option has been selected, a pop-up window appears to enable the user to choose a name for the scenario. This scenario will contain the list of tasks that have been executed during the simulation. Once a name has been chosen for the scenario, the simulation panel appears on the right-hand side in the HAMSTERS software environment. At the same time, a new visual element appears in the project explorer panel, on the left-hand side of the HAMSTERS software environment. This visual element represents the new scenario file that has been created (see Fig. 17.11).

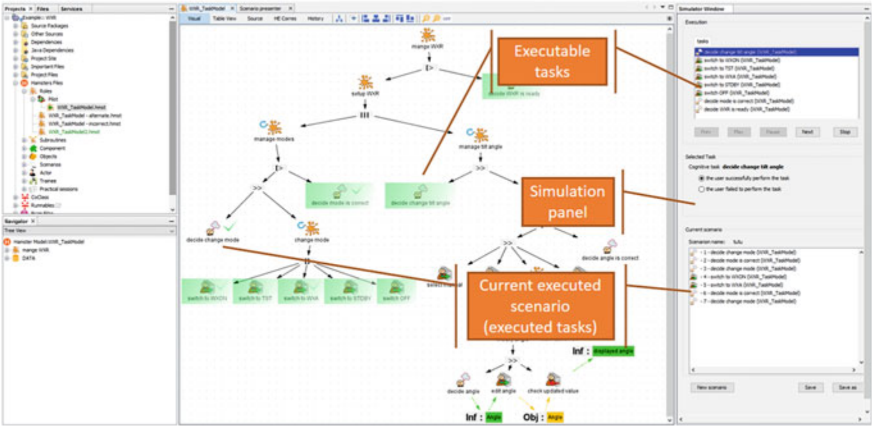


Fig. 17.11 Representation of executable and executed tasks during the simulation

The simulation panel provides information about:

- the current tasks that are available for execution (list in the upper part of the simulation panel in Fig. 17.11) and
- the scenario, i.e. the tasks that have been executed (list in the lower part of the simulation panel in Fig. 17.11).

The tasks which are available for execution are highlighted in green in the task model (in the central part in Fig. 17.11).

17.3.3.2 PetShop Tool

The ICO notation is supported by a CASE tool (named PetShop Palanque et al. 2009) for editing, simulating and analysing ICO models. As system models can be large, it is important to provide the analyst with computer-based tools for editing system models and for analysing them. To this end, the PetShop tool provides support for creating, editing, and simulating the execution of task models.

Editing

As depicted in Fig. 17.12, the PetShop environment is composed (as the HAMSTERS tool) of three main areas:

- On the left-hand side, the project exploratory,
- In the centre, the ICO model editing area,
- On the right-hand side, the palette containing the elements of the ICO notation (Petri net nodes: places and transitions and Petri net edges: arcs).

Visual notation elements can be added to the ICO model by a drag and drop operation from the palette to the ICO model area.

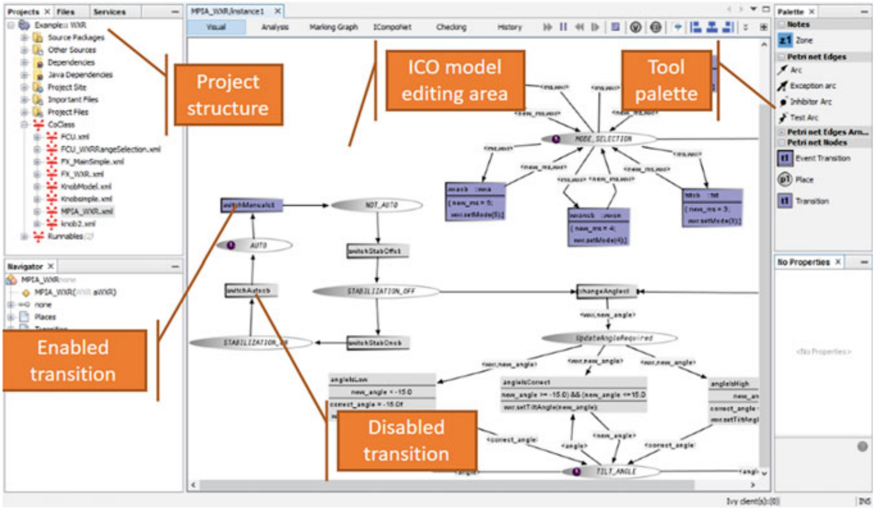


Fig. 17.12 PetShop tool screenshot

Simulation and Analysis

PetShop allows, at the same time, the editing (it is possible to add/remove places, transitions and arcs, change marking, etc.); simulation (while editing it is possible to see the enabled transitions (shown in a darker colour in Fig. 17.12) and analysis of the underlying Petri net structure (the set of place and transitions invariant are computed while editing and simulating (not visible in the figure but detailed in Sect. 6.2).

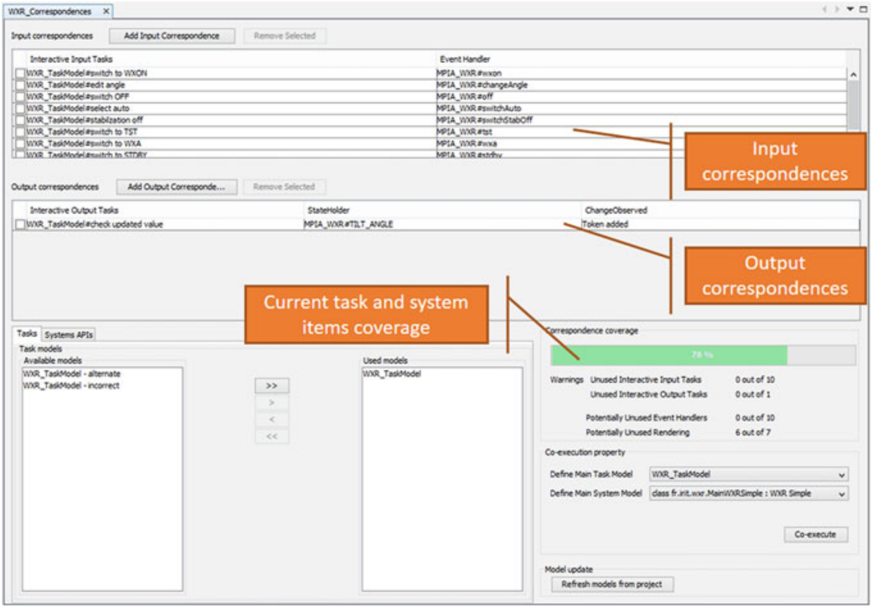
17.3.3.3 SWAN Tool

SWAN, which stands for Synergistic Workshop for Articulating Notations, is a tool enabling the co-execution of the ICO system models with the corresponding HAMSTERS user's task models. This tool enables the editing of correspondences between system and task models and their co-execution.

Editing

The editing of correspondences between the two models is done by a dedicated editor (depicted in Fig. 17.13) enabling the matching between interactive input tasks (from the task model) with system inputs (from the system model) and between system outputs (from the system model) with interactive output tasks (from the task model).

The table in the upper part of Fig. 17.13 represents the input correspondences, e.g. the link made between an input task and an activation adapter (a user event). In



Input correspondences

	Interactive Input Tasks	Event Handler
□	WXR_TaskModel#switch to WXON	MPIA_WXR#wxon
□	WXR_TaskModel#edit angle	MPIA_WXR#changeAngle
□	WXR_TaskModel#switch OFF	MPIA_WXR#off
□	WXR_TaskModel#select auto	MPIA_WXR#switchAuto
□	WXR_TaskModel#stabilization off	MPIA_WXR#switchStabOff
□	WXR_TaskModel#switch to TST	MPIA_WXR#tst
□	WXR_TaskModel#switch to WXA	MPIA_WXR#wxa
□	WXR_TaskModel#switch to STDBY	MPIA_WXR#stdby

Output correspondences

	Interactive Output Tasks	StateHolder	ChangeObserved
□	WXR_TaskModel#check updated value	MPIA_WXR#TILT_ANGLE	Token added

Task models

Available models

WXR_TaskModel - alternate
WXR_TaskModel - incorrect

Used models

WXR_TaskModel

Correspondence coverage

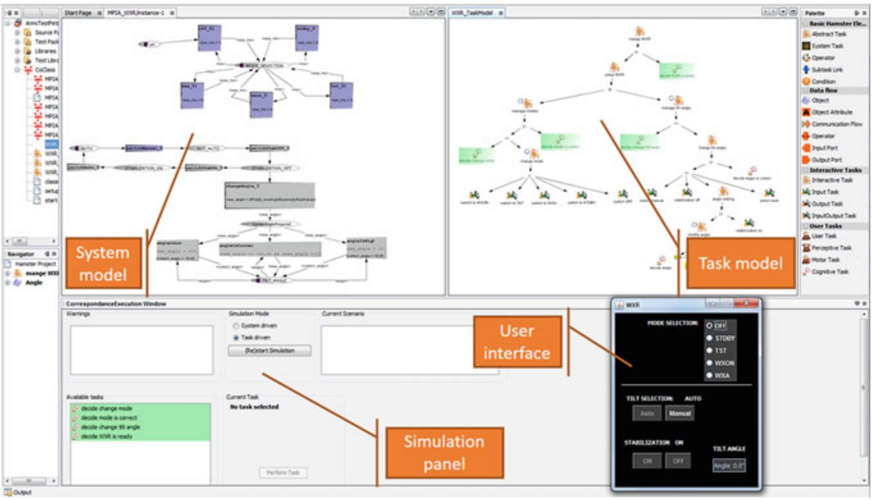
Warnings	Unused Interactive Input Tasks	0 out of 10
	Unused Interactive Output Tasks	0 out of 1
	Potentially Unused Event Handlers	0 out of 10
	Potentially Unused Rendering	6 out of 7

Co-execution property

Define Main Task Model: WXR_TaskModel
Define Main System Model: class fr.irit.wxr.MainWXRSimple : WXR Simple

Model update

Fig. 17.13 Editing of correspondences between tasks and system models



Warnings

Simulation Mode
○ System driven
● Task driven

Current Scenario

Available tasks
decide change mode
decide mode is correct
decide change tilt angle
decide WXR is ready

Current Task
No task selected

MODE SELECTION: ○ OFF ● STDBY ● TST ● WXON ● WXA
TILT SELECTION: AUTO
STABILIZATION: ON
TILT ANGLE: Angle: 0.0°

Fig. 17.14 WXR application and task models co-executing in the CIRCUS environment

the example, eight tasks are already connected to five user events (e.g. "switch_off" is connected to the user event "Off").

The table in the middle part of Fig. 17.13 represents the output correspondences, e.g. the link made between an output task and a rendering adapter. In the example

Fig. 17.13, one task is already connected to one rendering event: the interactive output task “check updated value” is connected to the rendering event “token added” in place “TILT_ANGLE”.

The lower right-hand part represents the current status of the editing of correspondences. It is made up of a progress bar showing the current coverage of task and system items by the edited correspondences.

Simulation

The co-execution of models may be launched (via the co-execution panel at the end of the lower right-hand part), even if correspondence editing is not completed. Figure 17.14 depicts the co-execution of the system models (left part), the task models (right part) and the visible part of the application (lower right part). The co-execution control panel stands in the lower part. The co-execution can be task driven (as depicted in Fig. 17.14) or system driven.

17.4 Editing Tools for the Development of Interactive Systems

In this section, we describe and illustrate how editing tools can feature user-centred functionalities that minimize the sequences of actions during the execution and evaluation paths according to Norman’s action theory presented above.

17.4.1 Norman’s Action Theory Applied to Editing Tools

In this task, the user has to edit the models. Table 17.4 details the application of the seven stages of Norman’s action theory to the editing of formal models. In this task, the user’s goal lies in writing the formal model using an editing tool:

Table 17.4 Norman’s action theory applied to the editing of formal models

	Evaluation path				Execution path		
	Perceive	Interpret	Compare	Goal	Plan	Specify	Perform
Editing	Perceive colours and shapes in the tool window	Interpret the notation elements	Compare the model with the desired one	Edit a model	Plan the list of notation elements that will be added	Specify the action sequence to edit the model with the identified notation elements	Perform actions to add the identified notation elements using the tool

- *On the execution path*, the user has to plan, specify and perform a sequence of actions corresponding to the description of the formal model. For instance, if the user has to construct a Petri net model, s/he will have to first plan the action sequence by formulating the intention to add places, transitions and arcs to construct the model. Then s/he will have to specify an action sequence by ordering the actions as, for instance, the following: add a place, add an arc, add a transition, etc. Finally, s/he will have to use the tool functionalities in order to add the elements to the model.
- *On the evaluation path*, the user has to perceive, interpret and compare the result of her/his actions with the expected model. For instance, if we stay with the previous example of Petri net modelling, the user will perceive the output presented by the tool such as rectangles, ellipses and arrows and will have to interpret them as transitions, places and arcs. S/he will then have to compare them with the model that corresponds to her/his goal.

The editing tool is used by the user during performing (at execution level) and perceiving (at interpretation level) activities. Each of the activities presented in Table 17.4 are separated by a gulf. Therefore, the editing tool can help reduce, at the execution level, the gulf between the specification of action sequence and its performance and, at the interpretation level, the gulf between the perception of model elements and their interpretation. A user-centred editing tool should enable, as much as possible, the reduction of these gulfs.

Graphical formal notations such as Petri nets give a good illustration of these gulfs. In that case, a graphical editor will enable the reduction of both execution and interpretation gulfs: at the execution level, it will simplify the transformation between a hand-drawn model on a sheet of paper to a digital one; at the interpretation level, it will simplify the interpretation as the user will not have to translate a textual representation into a graphical one, thus easing the comparison with the graphical one of the paper.

Another way to reduce execution and interpretation gulfs is to make available to the user generic functionalities such as cut/copy/paste or undo/redo. This way, if the user wants to undo a given action or set of actions, the task is directly provided by the tool, thus sparing the user from performing several iterations within Norman's action theory in order to reach the desired previous state.

Each notation has a specific lexicon, syntax and semantics. The editing tools should be as close as possible to these idiosyncratic aspects. For instance, a Petri net is only made up of two types of nodes connected by arcs (of various types). At the lexicon level, the tool should allow users to quickly select both of the node types. At the syntactic level the editing tool should prevent wrong models by making it impossible to connect a place to another place. Indeed, allowing the creation of syntactically incorrect models would require many further modifications to be performed by the user corresponding to iterations on Norman's action theory.

For a better understanding of how these gulfs can be reduced, next section describes how HAMSTERS and PetShop tools enable these gulf reductions.

17.4.2 Illustration with CIRCUS Environment

Both HAMSTERS and ICO are graphical notations. In order to reduce execution and interpretation gulfs at this level, both HAMSTERS and PetShop tools are graphical editors. They also enable the creation of all the elements of the associated notation by means of an editing palette, grouping all of them. Figure 17.15 presents the HAMSTERS and PetShop editing palettes. HAMSTERS proposes the different tasks, operators, data and structuring means while PetShop proposes the different nodes (transitions and places) and arcs to connect them. In both tools, the model is constructed by means of drag and drop actions. The tools also enable the editing of each element's properties. For instance, this is done in HAMSTERS tool through a panel directly accessible under the editing palette (see task properties panel in Fig. 17.10 enabling the editing of the properties of the corresponding highlighted task "decide WXR is ready"). Once again, the use of this panel allows to minimize both execution and interpretation gulf: a single click on the considered task is sufficient to edit and see its properties.

When using both HAMSTERS and ICO notations, the user will often have to perform typical action sequences (e.g. for ICO notation: add a place, add an arc, add a transition, etc.). These types of typical action sequences must be made available in order to reduce execution gulf when such action sequences have to be performed.

In order to reduce execution gulf, both HAMSTERS and PetShop tools enable the construction of the typical action sequences of each notation. PetShop even goes a little further by proposing a semi-automated creation of places or transitions.

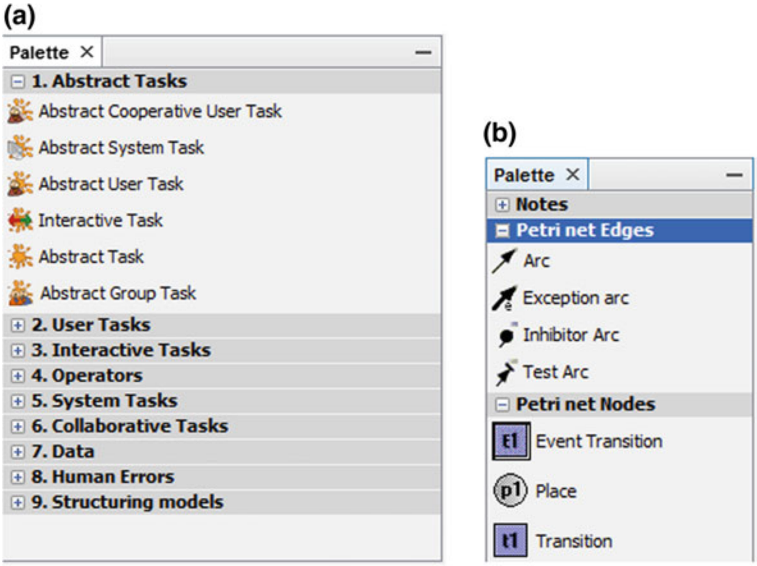


Fig. 17.15 **a** HAMSTERS and **b** PetShop palettes

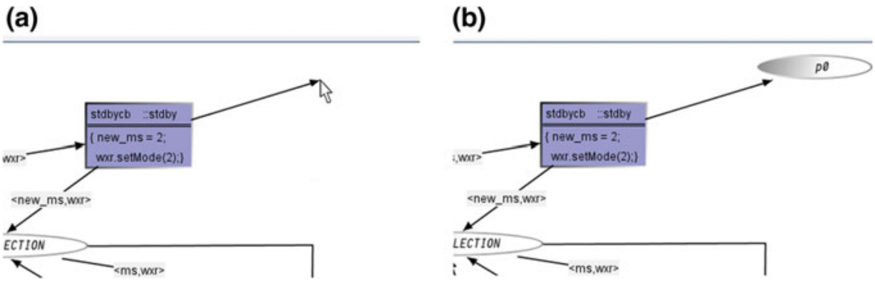


Fig. 17.16 Semi-automated place creation with PetShop

Fig. 17.17 HAMSTERS and PetShop zoom and align palette

Actually, in a Petri net formalism, places can only be connected to transitions and transitions can only be connected to places (a place cannot be connected to another place and a transition cannot be connected to another transition). Thus, when creating an arc going from a transition, this arc only can be connected to a place. To reduce the execution gulf, PetShop proposes the automated creation of places when creating an arc from a transition. This is illustrated in Fig. 17.16 where we can see that the creation of the arc (arc going from *stdbycb* transition in Fig. 17.16a) automatically creates a place (named *p0* in Fig. 17.16b) connected to this arc. The same mechanism is available for automatically creating transitions when creating arcs from places.

In order to reduce the interpretation gulf, graphical tools must also enable the management of real size models that can be very large ones. In the HAMSTERS and PetShop tools, this is achieved through some classical mechanisms such as zoom-in/zoom-out actions. The palette offering these mechanisms is depicted in Fig. 17.17. This palette also offers some aligning actions that reduce the execution gulf for aligning elements together; alignment of model elements support the interpret and perceive activities in Fig. 17.3.

HAMSTERS and PetShop tools and notations also propose the use of mechanisms that facilitate the interpretation of large models.

For this purpose, HAMSTERS notation and tool provide the mechanisms of subroutines, submodels and components (Forbrig et al. 2014) that enable the breakdown of a task model in several ones that can be reused in the same or different task models.

As for PetShop, a recurrent issue in large Petri net models is that a single place can be connected to a large number of transitions, thus creating a big arc density. To solve this issue, PetShop supports the creation of virtual places (i.e. virtual copies of a single place). They adopt the display properties (such as markings) of the place

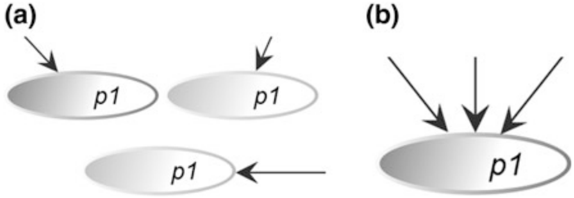


Fig. 17.18 Real view (*left*) versus semantic view (*right*)

Table 17.5 Norman's action theory applied to the verification of formal models

	Evaluation path				Execution path		
	Perceive	Interpret	Compare	Goal	Plan	Specify	Perform
Verification	Perceive colours and shapes in the tool window	Interpret the notation elements	Compare the notation elements with the language syntax	Verify a model	Plan the list of syntax errors that have to be corrected	Specify the action sequence to correct the model	Perform actions to correct the model

they copy but not the arc connections. The arcs can be connected to normal places or to virtual places. Therefore, the display is modified allowing an easier reorganization of models and increasing the model legibility through this graphical artefact. Figure 17.18 depicts an example of the use of virtual places. Figure 17.18a shows three places named *p1* (the "source" place with a darker border and two virtual places) each one connected to an input arc. Figure 17.18b shows the result of the reconstitution of the place *p1* (which is then connected to the three previous arcs). The semantic definition of virtual places along with concrete examples of how they can be used is presented in (Barboni et al. 2006b).

17.5 Verification Tools for the Development of Interactive Systems

In this section, we describe and illustrate how verification tools can feature user-centred functionalities that minimize the sequences of actions during the interpretation and evaluation paths according to Norman's action theory.

17.5.1 Norman's Action Theory Applied to Verification Tools

In this task, the user has to verify the models. Table 17.5 details the application of the seven stages of Norman's action theory to the verification of models. In this task, the user's goal lies in making sure that the notation elements have been correctly used in the produced model:

- *At the execution level*, the user has to plan, specify and perform a sequence of action corresponding to the verification of the formal model. For instance, if the user has to verify a Petri net model, s/he will have to first plan the action sequence by formulating the list of syntax errors that needs to be corrected. Then s/he will have to specify an action sequence by ordering the actions as, for instance, the following: delete an arc, delete a place, modify the action of a transition, etc.
- *At the interpretation level*, the user has to perceive, interpret and compare the result of her/his actions with the expected model. For instance, if we stay with the previous example of Petri net modelling, the user will perceive the output presented by the tool such as rectangles, ellipses and arrows and will have to interpret them as notation elements (places, arcs, transitions). S/he will then have to assess whether or not their syntax is correct.

The verification tool is used by the user during the performing (at execution level) and perceiving (at interpretation level) activities. Each of the activities presented in Table 17.5 is separated by a gulf. Therefore, the verification tool can help reduce both the execution gulf and the interpretation gulf. The execution gulf lies between the planning of tasks and their performance. At the evaluation level, the gulf lies between the perception of model elements and their interpretation. A user-centred verification tool should enable, as much as possible, the reduction of these gulfs.

Syntax highlighting functionalities in development environments are a good example of tools that can help to reduce both execution and interpretation gulfs. They highlight the elements that have been edited using an unknown syntax, helping the user to rapidly identify and correct the syntax errors.

The next section describes the mechanisms that are proposed in HAMSTERS and PetShop to enable these gulf reductions.

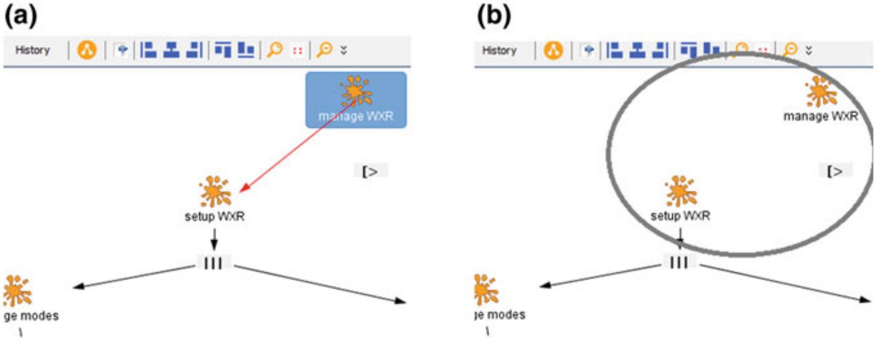


Fig. 17.19 HAMSTERS behaviour when trying to connect a task to another

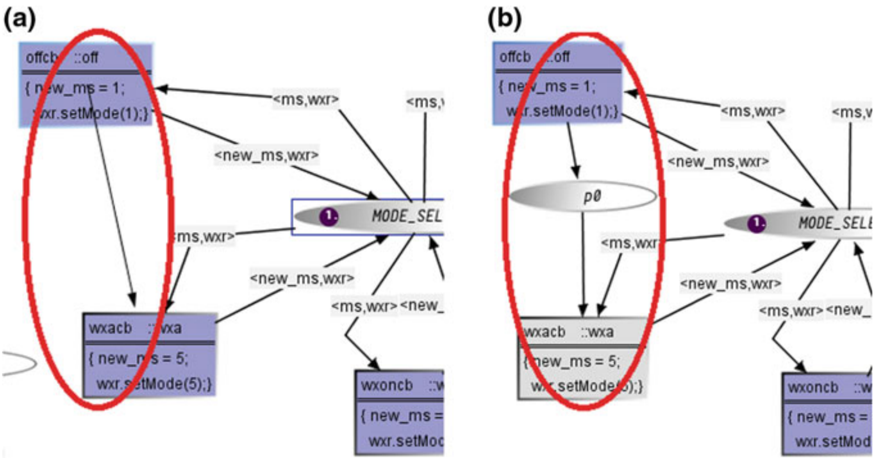


Fig. 17.20 Place creation when trying to connect a transition to another in PetShop

17.5.2 Illustration with CIRCUS Environment

HAMSTERS and PetShop provide mechanisms to prevent syntax errors. For example, in HAMSTERS a task cannot be connected to another task. It has to be connected to a temporal ordering operator. When the user tries to connect a task to another one by (a) clicking on the left mouse button on the source task and then (b) by dragging the cursor to the targeted task (the graphical rendering of these actions is depicted in Fig. 17.19a), the colour of the arc changes to red and even if the user releases the left mouse button, the arc never connects to the targeted task. When the user releases the left mouse button, the arc is not set (the graphical rendering of these actions is depicted in Fig. 17.19b).

Another example in PetShop is the automatic creation of notation elements to prevent wrong syntax in models. In Petri nets, a transition cannot be connected to another transition. The "place-transition-place-transition…" sequence is mandatory. When the user tries to connect a transition to another one by (a) clicking on the left mouse button on the source transition and then (b) by dragging the cursor to the targeted transition (the graphical rendering of these actions is depicted in Fig. 17.20a), the arc never connects to the targeted transition. Instead, a place is created in between the source and targeted transition. When the user releases the left mouse button, the arc is not set (the graphical rendering of these actions is depicted in Fig. 17.20b). In the same way, a transition is automatically created when a user tries to connect a place to another one.

The PetShop tool embeds many other functionalities, but as they are related to the Integrated Development Environment (NetBeans) on which PetShop is developed they are not presented here.

17.6 Validation Tools for the Development of Interactive Systems

In this section, we describe and illustrate how validation tools can feature user-centred functionalities that minimize the sequences of actions during the interpretation and evaluation paths according to Norman's action theory.

17.6.1 Norman's Action Theory Applied to Validation Tools

In this task, the user has to validate the models. Table 17.6 details the application of the seven stages of Norman's action theory to the validation of models. In this task, the user's goal lies in making sure that what is described in the produced models is compliant with the requirements and needs:

- *At the execution level*, the user has to plan, specify and perform a sequence of action corresponding to the validation of the formal model. For instance, if the user has to make sure that a task model described the activities that a user will have to perform with a system, s/he will have to first plan the action sequence by formulating the intention to review and correct asks and temporal operators that are in the model. Then, s/he will have to specify an action sequence by ordering the actions as, for instance, the following: add a task, delete a task, modify an operator, etc.
- *At the interpretation level*, the user has to perceive, interpret and compare the result of her/his actions with the expected model. For instance, if we stay with the previous example of HAMSTERS task models, the user will perceive the output presented by the tool such as task icons and temporal operators. S/he will have to interpret them as temporally ordered tasks. S/he will then have to assess whether or not the described temporally ordered tasks are compliant with users planned activities with the system.

The validation tool is used by the user during the performing (at execution level) and perceiving (at interpretation level) activities. Each of the activities presented in

Table 17.6 Norman's action theory applied to the validation of formal models

	Evaluation path				Execution path		
	Perceive	Interpret	Compare	Goal	Plan	Specify	Perform
Validation	Perceive colours and shapes in the tool window	Interpret the signification of the presented model	Compare the signification of the presented model against needs and requirements	Validate a model's signification	Plan the list of elements that have to be inspected and corrected	Specify the action sequence to correct the model and to continue to inspect	Perform actions to correct the models and to continue to inspect

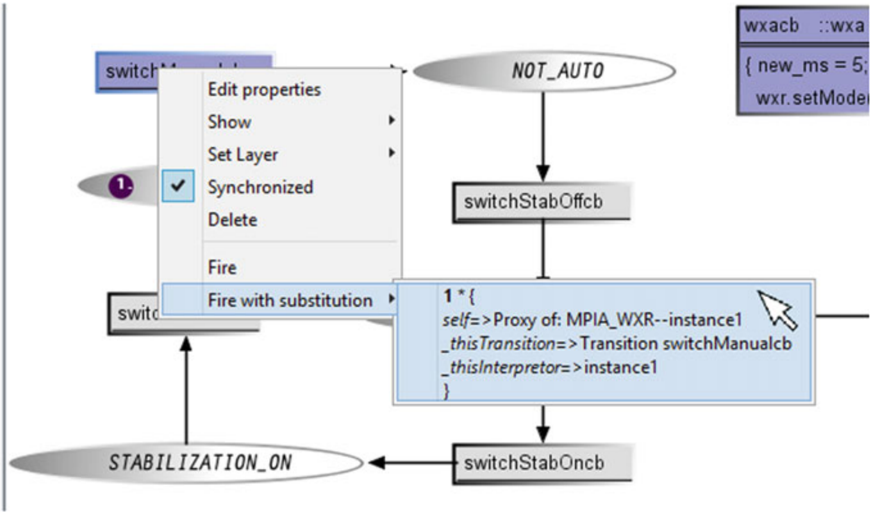
Table 17.6 is separated by a gulf. Therefore, the validation tool can help reduce both the execution gulf and the interpretation gulf: at the execution level, the execution gulf lies between the planning of tasks and their performance; at the interpretation level, the gulf lies between the perception of model elements and their interpretation. A user-centred validation tool should enable, as much as possible, the reduction of these gulfs.

Simulation is a good example of mechanisms that provide support for reducing the interpretation gulf. Simulation is the task where users (people building the model) have to check that the model built exhibits the expected behaviour. The interpretation task can be eased if the state changes in the model are shown in an animated way, thus reducing the users' activity of comparing the current state with the previous one.

Next section describes the mechanisms that are proposed in HAMSTERS and PetShop to enable these gulf reductions.

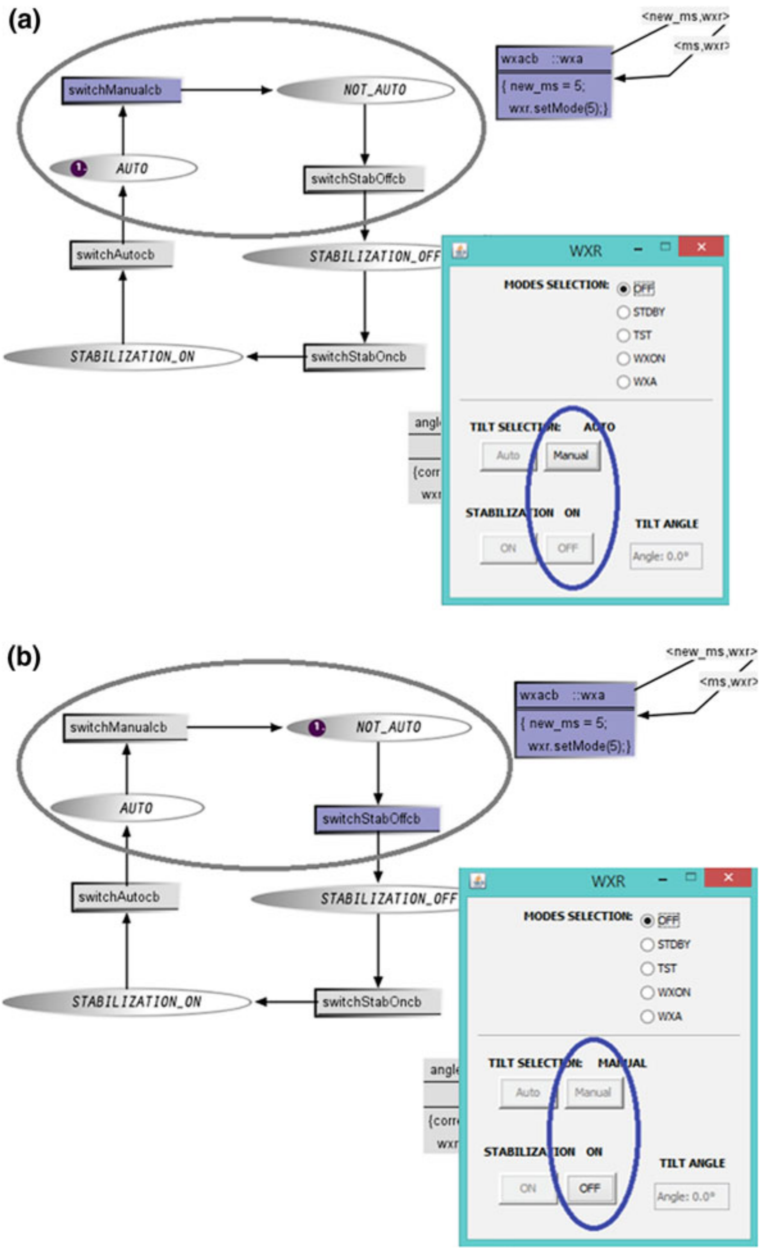
17.6.2 Illustration with CIRCUS Environment

HAMSTERS provide mechanisms to execute the task models in order to ensure that the tasks described in the model match the different possible sequences of activities that the user has to lead using the envisioned system. In particular, the simulation functionality enables execution of tasks in the temporal order that has been specified at editing time. For example, the simulation mode of HAMSTERS is depicted in Fig. 17.11. Available tasks are highlighted in green in the task model and listed in the upper part of the simulation panel (see Fig. 17.11). Executed tasks are ticked



```
1 * {
self=>Proxy of: MPIA_WXR--instance1
_thisTransition=>Transition switchManualcb
_thisInterpretor=>instance1
}
```

Fig. 17.21 Simulation of an ICO model with PetShop



(a)

```
{ new_ms = 5;
wxr.setMode(5);}
```

MODES SELECTION: ● OFF ○ STDBY ○ TST ○ WXON ○ WXA

TILT SELECTION: AUTO

STABILIZATION ON

TILT ANGLE Angle: 0.0°

(b)

```
{ new_ms = 5;
wxr.setMode(5);}
```

MODES SELECTION: ● OFF ○ STDBY ○ TST ○ WXON ○ WXA

TILT SELECTION: MANUAL

STABILIZATION ON

TILT ANGLE Angle: 0.0°

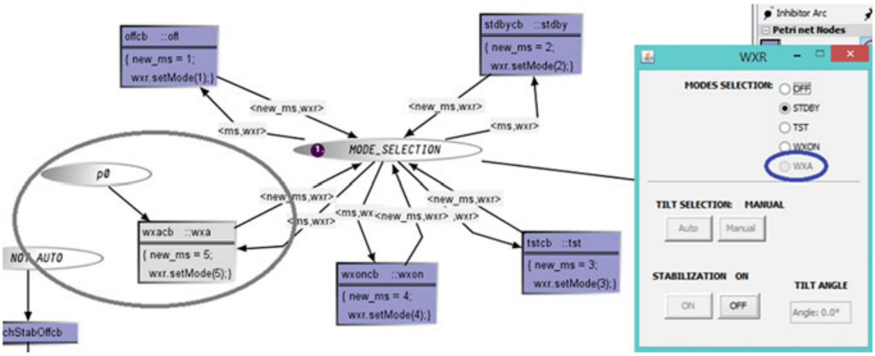
Fig. 17.22 Animation of the ICO model during simulation in PetShop

by a green mark in the task model and listed in the lower part of the simulation panel (see Fig. 17.11). This mode enables production of scenarios from the task model and to confront them with existing scenarios of usage of the interactive system under design and to ensure that the task model meets the user needs in terms of envisioned activities.

PetShop provides mechanisms to execute the ICO models in order to ensure that the described behaviour matches the targeted one. PetShop simulation mechanisms enable the simplification of the execution task by showing the user the set of enabled transitions and allowing her/him to act directly on the model. The set of enabled transitions are depicted in violet while the disabled ones are depicted in grey. The user can fire an enabled transition by right-clicking on it and choosing the "Fire" or "Fire with substitution" options (see Fig. 17.21). The interpretation task is also eased by the token shifting and the automatically update of the set of enabled transitions following the firing of a transition. Figure 17.22 presents the evolution of the WXR ICO model before (Fig. 17.22a) and after (Fig. 17.22b) the firing of "switchManualcb" transition; the part of the model that has been modified is highlighted in grey and the corresponding modification on the user interface is highlighted in blue.

Moreover, PetShop enables a modeless execution between editing and simulation. Therefore, the editing of the ICO model immediately modifies the set of enabled transitions and is immediately reported on the user interface. This is illustrated in Fig. 17.23 where we can see how the adding of place "p0" is impacting the set of enabled transitions (part of the model highlighted in grey in Fig. 17.23: the transition "wxacb" is not enabled anymore) and the corresponding user interface (part of the model highlighted in blue in Fig. 17.23: the "WXA" button is disabled).

Another good example of validation mechanisms relies on the analysis of the model. PetShop provides means to analyse ICO models through the analysis of the underlying Petri net model. For the purposes of this paper, we only use here classical Petri net analysis terms; for further details, the interested reader can refer to



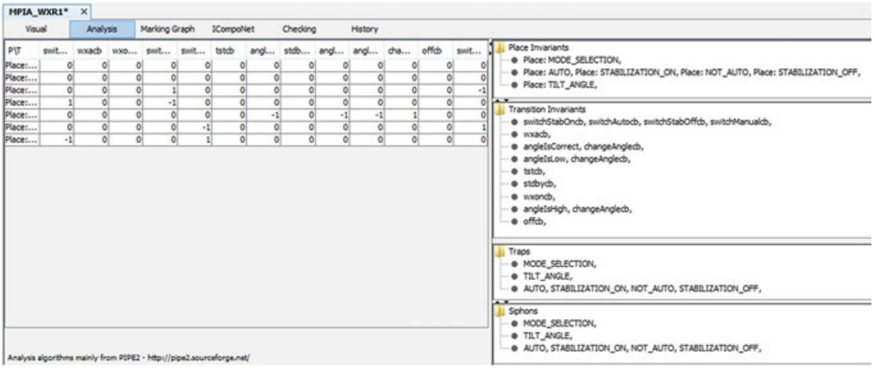
MODES SELECTION: ○ OFF ● STDBY ○ TST ○ WXON ○ WXA

TILT SELECTION: MANUAL

STABILIZATION ON

TILT ANGLE Angle: 0.0°

Fig. 17.23 Illustration of how the editing of the ICO model is impacting the simulation view



P\T	swit...	wxacb	wxo...	swit...	swit...	tstcb	angl...	stdb...	angl...	angl...	cha...	offcb	swit...
Place:...	0	0	0	0	0	0	0	0	0	0	0	0	0
Place:...	0	0	0	0	0	0	0	0	0	0	0	0	0
Place:...	0	0	0	1	0	0	0	0	0	0	0	0	-1
Place:...	1	0	0	-1	0	0	0	0	0	0	0	0	0
Place:...	0	0	0	0	0	0	-1	0	-1	-1	1	0	0
Place:...	0	0	0	0	-1	0	0	0	0	0	0	0	1
Place:...	-1	0	0	0	1	0	0	0	0	0	0	0	0

Analysis algorithms mainly from PIPE2 - http://pipe2.sourceforge.net/

Place Invariants
- Place: MODE_SELECTION,
- Place: AUTO, Place: STABILIZATION_ON, Place: NOT_AUTO, Place: STABILIZATION_OFF,
- Place: TILT_ANGLE,

Transition Invariants
- switchStabOncb, switchAutocb, switchStabOffcb, switchManualcb,
- wxacb,
- angleIsCorrect, changeAnglecb,
- angleIsLow, changeAnglecb,
- tstcb,
- stdbycb,
- wxoncb,
- angleIsHigh, changeAnglecb,
- offcb,

Traps
- MODE_SELECTION,
- TILT_ANGLE,
- AUTO, STABILIZATION_ON, NOT_AUTO, STABILIZATION_OFF,

Siphons
- MODE_SELECTION,
- TILT_ANGLE,
- AUTO, STABILIZATION_ON, NOT_AUTO, STABILIZATION_OFF,

Fig. 17.24 WXR model structural analysis—Tabular view

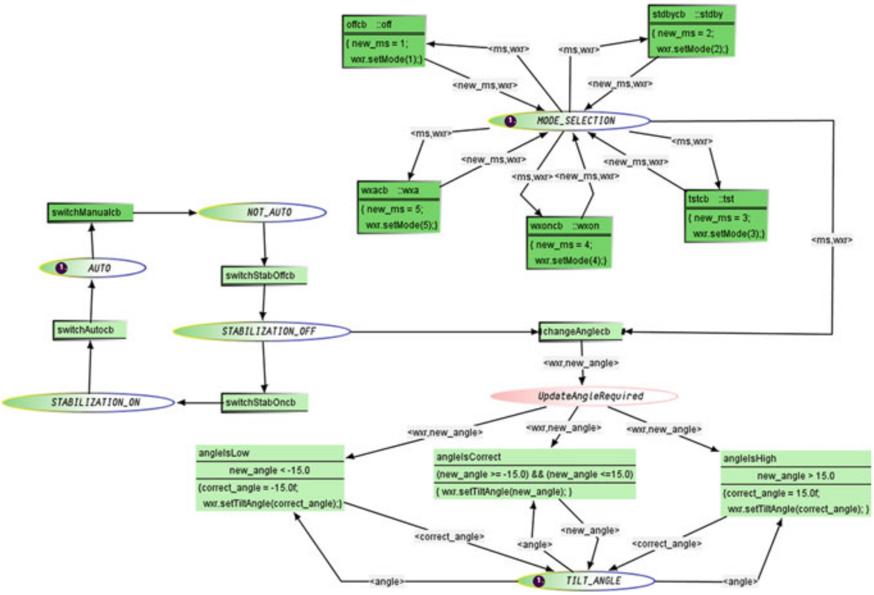


Fig. 17.25 Representation of invariants with the analysis feature of PetShop CASE tool

(Reisig 2013). For analysis-related tasks, users check the validity of some properties in the model. One of the issues is then to understand the analysis result (for instance, one place is not in any “P invariant”) in Petri net tools and then to map this analysis result with the goal (Was this place meant to be unbounded?). More precisely, PetShop provides support for analysis (Silva et al. 2014) through invariant analysis (place/transition invariants). It allows the user direct visualization of the structural analysis results while editing and simulating the Petri nets’ models. The analysis mode can be activated without stopping either the editing or the simulation.

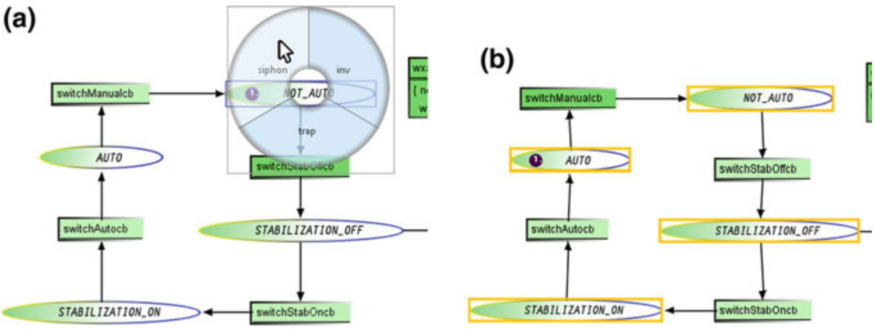


Fig. 17.26 Visualization of a siphon in PetShop—**a** pie menu enabling the display of siphons implying "NOT_AUTO" place and **b** display of the siphon implying "NOT_AUTO" place

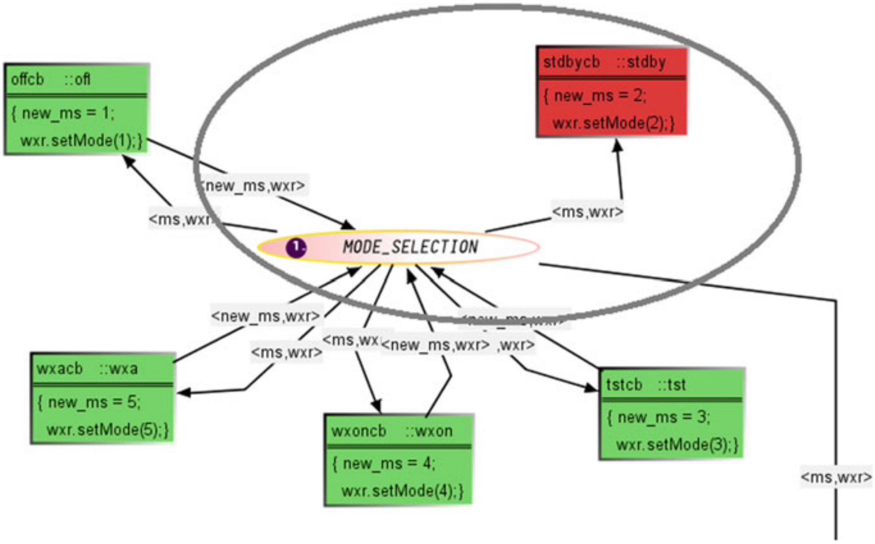


Fig. 17.27 Impact of the editing of the ICO model on the analysis visualization in PetShop

In this section, we emphasize the validation aspect of the system model presented in Fig. 17.9. When activated, the static analysis mode of PetShop displays a dedicated panel (see Fig. 17.24) presenting the incidence matrix and the P/T invariants, which allow the identification of potential deadlocks in the system models. The results of this analysis are also displayed in the main editing view as shown in Fig. 17.25. The green overlay on the places and transitions identifies the node part of the model's invariants otherwise the red overlay is used (as for the place "UpdateAngleRequired"). Places with yellow borders are siphons whereas traps use a blue stroke.

In order to determine which nodes belong to the same invariant as another node, a modal pop-up menu is provided, taking over the standard pop-up menu for editing. This modal pop-up menu can be switched from analysis mode to normal editing mode using a dedicated icon on the toolbar. This modal pop-up menu is a pie menu (as it is more usable than a classical menu) depicted in Fig. 17.26a for the "NOT_AUTO" place. This place is part of both a siphon and a trap (as shown by its yellow and blue border) and a p invariant (as shown by its green overlay). Therefore, the pie menu proposes to show which nodes belong to the invariant, the siphon or the trap the place is included in. Figure 17.26b is the result of the action of the user to display the nodes belonging to the siphon the place "NOT_AUTO" is included in: the places included in this siphon are highlighted in yellow (e.g. places "AUTO", "NOT_AUTO", "STABILIZATION_OFF" and "STABILIZATION_ON").

Furthermore, as for the simulation, the editing of the model has a direct impact on the analysis and its visualization. This is illustrated in Fig. 17.27 where a modification has been performed in the model. The arc between the place "MODE_SELECTION" and the transition has been removed. This modification is highlighted in grey in Fig. 17.27. This modification has the following impact: (i) the transition "stbcb" is not part of an invariant anymore (it is shown by its red overlay), (ii) the place "MODE_SELECTION" is not part of a siphon anymore (it is shown by the fact that the border is not blue anymore) and (iii) the place "MODE_SELECTION" is not an invariant anymore (it is shown by its red overlay).

17.7 Beyond Multiple Unrelated Views: Connecting Models to Leverage V&V Tasks

System and task models provide support to design and develop interactive systems, but there is still a need to ensure consistency between these two views of the system in use. In order to go beyond applying Norman's action theory to modelling activities for each type of models used, the activities of ensuring that all the models are consistent and compliant have to be taken into account also.

In this section, we describe and illustrate how V&V tools can feature user-centred functionalities that minimize the sequences of actions during the execution and evaluation paths according to Norman's action theory for cross-model validation.

17.7.1 Norman's Action Theory Applied to V & V Tools

In this task, the user has to validate the completeness and consistency of models of different types. Table 17.7 details the application of the seven stages of Norman's action theory to the cross-type validation of models. In this task, the user's goal lies

Table 17.7 Norman's action theory applied to the cross-type validation of models

	Evaluation path				Execution path		
	Perceive	Interpret	Compare	Goal	Plan	Specify	Perform
Cross-model validation	Perceive colours and shapes in the tool window	Interpret the relationships between elements of different model types	Compare the relationships between elements of different model types against needs and requirements	Check completeness and consistency of both model types	Plan the list of elements that have to be added, modified or removed	Specify the action sequence to correct the models	Perform actions to correct the models

in making sure that what is described in models from both types (task and system) is complete and consistent:

- *At the execution level*, the user has to plan, specify and perform a sequence of action corresponding to the cross-type validation of the formal models. For instance, if the user has to make sure that what is described in the task model can be performed by the system described by the system model. S/he will have to first plan the action sequence to correct the models (being task or system model) and/or the correspondences between the two types of models. Then, s/he will have to specify an action sequence by ordering the actions as, for instance, the following: choose the first part of the model (being task or system model) that has to be corrected, choose the notation element that has to be modified, added or deleted, etc. S/he finally performs the specified action sequence.
- *At the interpretation level*, the user has to perceive, interpret and compare the relationship between the elements of different model types both against each other and against need and requirements. For instance, using the example of task and system models, s/he will perceive the output presented by both tools (the one for task modelling and the one for system modelling) such as in the case of tasks modelling, task icons and temporal operators. S/he will have to interpret them as temporally ordered tasks (for task models) and system behaviour (for system models) and will then have to assess whether or not the described temporally ordered tasks and system behaviour are compliant together.

A V&V tool can be used by the user during the performing (at execution level) and perceiving (at interpretation level) of these activities. Each of the activities presented in Table 17.7 is separated by a gulf. Therefore, the V&V tool can help reduce both the execution gulf and the interpretation gulf. The execution gulf lies between the planning of tasks and their performance. At the interpretation level, the gulf lies between the perception of both model elements, their interpretation and their comparison. A user-centred V&V tool should enable, as much as possible, the reduction of these gulfs.

A good example of mechanisms that provide support for reducing the interpretation gulf is to provide the user a way of putting in correspondence both model types. Therefore, the use of such a tool will simplify the comparison of the output of both model types.

Co-simulation of both model types is also a good example of mechanisms that provide support for reducing the interpretation gulf. In this case, the interpretation task can be simplified if the state changes are linked between both model types.

17.7.2 Illustration with the CIRCUS Environment

In the CIRCUS environment, the cross-type validation of models is supported by a specific tool named SWAN (see Sect. 3.3.3). This tool supports the co-execution of HAMSTERS task models and ICO system models, based on a list of correspondences between the two model types.

The SWAN tool therefore enables the decrease of the interpretation gulf by allowing the user to establish the matching between interactive input tasks (from the task model) with system inputs (from the system model) and between system outputs (from the system model) with interactive output tasks (from the task model). The correspondence editing panel is completed by a representation of the current status of the editing of correspondences (see Fig. 17.13), reducing the interpretation gulf by enabling the user with information on the current coverage of task and system items by the edited correspondences. Therefore, the interpretation task of checking if all of the task items are corresponding to system items (or, respectively, if all system items are corresponding to task items) is simplified. These functionalities provide support for the verification of the compatibility of the models but also for the validation of the user activities with regard to the system functions.

The SWAN tool also enables the decrease of the interpretation gulf by allowing the user to co-execute both task and system models. This functionality provides support for validation as it allows finding inconsistencies such as sequences of user actions allowed by the system model and forbidden by the task model, or sequences of user actions that should be available but which are not because of the system's inadequate state.

On the execution side, the CIRCUS environment also enables the reduction of the gulf by integrating the HAMSTERS tool, the PetShop tool and the SWAN tool in an integrated environment. Therefore, the user is able to modify task models, system models and their corresponding correspondences in a unique environment without having to switch tools which would considerably increase the execution gulf.

Table 17.8 Norman's action theory applied to the different engineer's tasks when using formal models to develop interactive systems

	Evaluation path				Execution path		
	Perceive	Interpret	Compare	Goal	Plan	Specify	Perform
Editing	Perceive colours and shapes in the tool window	Interpret the notation elements	Compare the model with the desired one	Edit a model	Plan the list of notation elements that will be added	Specify the action sequence to edit the model with the identified notation elements	Perform actions to add the identified notation elements using the tool
Verification	Perceive colours and shapes in the tool window	Interpret the notation elements	Compare the notation elements with the language syntax	Verify a model	Plan the list of syntax errors that have to be corrected	Specify the action sequence to correct the model	Perform actions to correct the model
Validation	Perceive colours and shapes in the tool window	Interpret the signification of the presented model	Compare the signification of the presented model against needs and requirements	Validate a model's signification	Plan the list of elements that have to be inspected and corrected	Specify the action sequence to correct the model and to continue to inspect	Perform actions to correct the models and to continue to inspect
Cross-model validation	Perceive colours and shapes in the tool window	Interpret the relationships between elements of different model types	Compare the relationships between elements of different model types against needs and requirements	Check completeness and consistency of both model types	Plan the list of elements that have to be added, modified or removed	Specify the action sequence to correct the models	Perform actions to correct the models

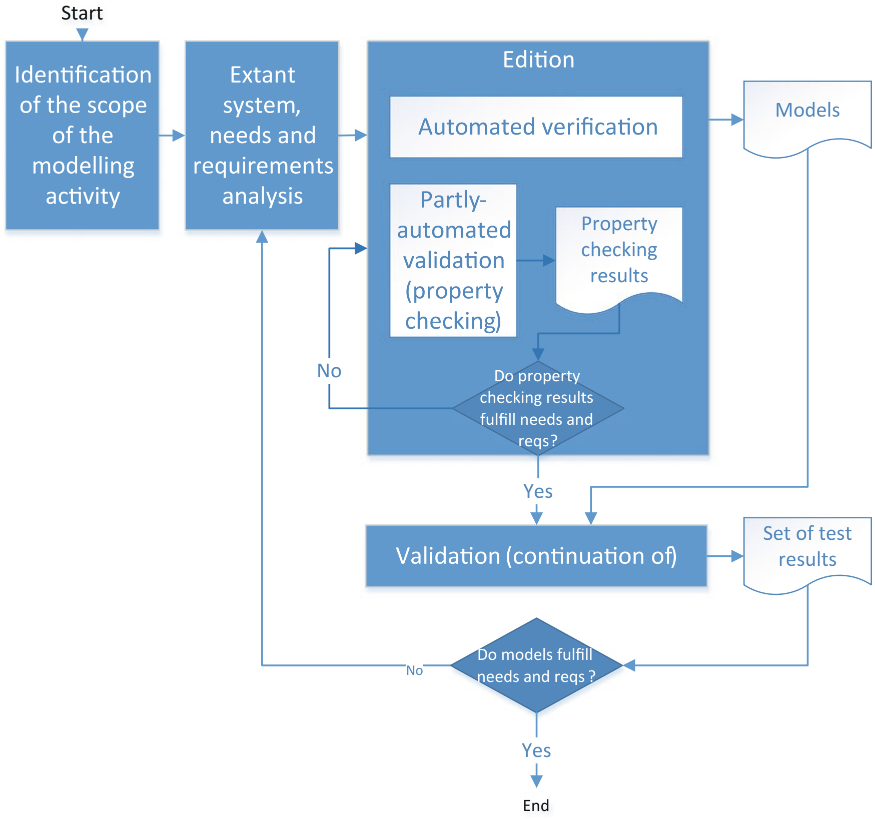

Start
Identification of the scope of the modelling activity
Extant system, needs and requirements analysis
Edition
Automated verification
Partly-automated validation (property checking)
Property checking results
Models
No
Do property checking results fulfill needs and reqs?
Yes
Validation (continuation of)
Set of test results
No
Do models fulfill needs and reqs ?
Yes
End

Fig. 17.28 ICO-based interactive application development process

17.8 Discussion

In previous sections, we have shown that the seven stages of Norman's action theory can be used to analyse to what extent an IDE tool provides support to engineer tasks. Table 17.8 recapitulates the engineer's tasks according to these seven stages. This table is intended to serve as a generic framework, detailing the activities that have to be led by the user/engineer. This table is a support for investigating how modelling activities can be supported at editing, verification and validation time.

For example, taking into account all the presented engineer's tasks, the above table helps in demonstrating that the PetShop modelling tool drastically decreases the number of actions that the engineer has to perform. The verification task has been completely automated within PetShop and the validation task is partly automated. Figure 17.28 presents the development process of ICO-based interactive application using PetShop.

The process depicted in Fig. 17.28 has been instantiated from the model's development process presented in Fig. 17.1 and has been refined according to the PetShop tool features. In particular, as depicted in Fig. 17.28, the verification step has been merged within the editing step as models edited within PetShop are automatically verified at editing time. And a part of the validation step has also been merged within the editing step as part of the properties of the models edited within PetShop can be automatically checked. This instantiation of the model's development process shows that it is possible to integrate user-centred functionalities in modelling tools that can help in reducing the gulfs of evaluation and the gulfs of execution. Validation and verification activities can be embedded and partly automated at editing time and there is no need to wait that a model is completely edited to start verification and validation activities. Some parts of the models can be checked automatically at editing time, reducing the number of actions that the user has to perform on the valuation path and on the execution path of Norman's action theory.

17.9 Conclusion and Perspectives

The use of formal modelling tools during the development process of an interactive system has an impact on the way the models will be edited, verified and validated. Norman's action theory has been applied to the modelling activity and provided a generic framework for investigating how modelling activities can be supported at editing, verification and validation time.

This chapter has identified the gulfs of the three steps of the construction of a formal model of an interactive system. This way it provides a structured research agenda for the engineering of tools aiming at supporting the design of interactive systems following a model-based approach.

We have exemplified those requirements on an integrated development environment: the CIRCUS environment. Even though the former is dedicated to task modelling and the latter to interactive systems modelling, we have demonstrated that:

- Norman model of the action theory provides a generic framework for identifying execution and interpretation gulf during building models;
- Solutions for gulf reductions can be identified both at generic level (e.g. undo/redo) and at specific level (e.g. preventing the editing of models not compliant with the syntax of the notation.

Beyond that, we have also applied those principles in the context of a multi-modelling approach where multiple views of the same world are used to describe and analyse interactive systems.

In this chapter, we focused on tools support as well as engineer's tasks about models' production with tools. However, notations also have an impact on the way the models are edited, verified and validated. To this end, Moody has proposed a set of principles to analyse the cognitive effectiveness of visual notation (Moody 2009). Future perspectives for this work would be to analyse the impact of the formal modelling notations (their semantics and their visual syntax) on the engineer's tasks as well as to analyse the combined impact of both notation and tool on the engineer's tasks.

References

Barboni E, Bastide R, Lacaze X, Navarre D, Palanque P (2003) Petri net centered versus user centered Petri nets tools. In: 10th workshop algorithms and tools for petri nets, AWPN

Barboni E, Conversy S, Navarre D, Palanque P (2006a) Model-based engineering of widgets, user applications and servers compliant with ARINC 661 specification. In: Interactive systems. design, specification, and verification. Springer, Berlin, pp 25–38

Barboni E, Navarre D, Palanque P, Basnyat S (2006b) Addressing issues raised by the exploitation of formal specification techniques for interactive cockpit applications. In: International conference on Human-Computer interaction in aeronautics (HCI-Aero)

Barboni E, Ladry JF, Navarre D, Palanque P, Winckler M (2010) Beyond modeling: an integrated environment supporting co-execution of tasks and systems models. In: Proceedings of the 2nd ACM SIGCHI symposium on engineering interactive computing systems. ACM, pp 165–174

Bastide R, Navarre D, Palanque P, Schyn A, Dragicevic P (2004) A model-based approach for real-time embedded multimodal systems in military aircrafts. In: Proceedings of the 6th international conference on Multimodal interfaces. ACM, pp 243–250

Cockton G, Gram C (1996) Design principles for interactive software. Springer Science & Business Media

Dix A (1991) Formal methods for interactive systems, vol 16. Academic Press, London, UK

Forbrig P, Martinie C, Palanque P, Winckler M, Fahssi R (2014) Rapid task-models development using sub-models, sub-routines and generic components. In: Human-Centered software engineering. Springer, Berlin, pp 144–163

Hamon A, Palanque P, Silva J L, Deleris Y, Barboni E (2013) Formal description of multi-touch interactions. In: Proceedings of the 5th ACM SIGCHI symposium on engineering interactive computing systems. ACM, pp 207–216

International Standard Organization (1996) DIS 9241-11: ergonomic requirements for office work with visual display terminals (VDT)—Part 11 Guidance on Usability

Lakos CA, Keen CD (1991) LOOPN-Language for object-oriented Petri nets. Department of computer science. University of Tasmania

Martinie C, Barboni E, Navarre D, Palanque P, Fahssi R, Poupart E, Cubero-Castan E (2014) Multi-models-based engineering of collaborative systems: application to collision avoidance operations for spacecraft. In: Proceedings of the 2014 ACM SIGCHI symposium on engineering interactive computing systems. ACM, pp 85–94

Martinie C, Palanque P, Barboni E, Ragosta M (2011a) Task-model based assessment of automation levels: application to space ground segments. In: 2011 IEEE international conference on systems, man, and cybernetics (SMC).IEEE, pp 3267–3273

Martinie C, Palanque P, Barboni E, Winckler M, Ragosta M, Pasquini A, Lanzi P (2011b) Formal tasks and systems models as a tool for specifying and assessing automation designs. In: Proceedings of the 1st international conference on application and theory of automation in command and control systems. IRIT Press, pp 50–59

Martinie C, Palanque P, Navarre D, Barboni E (2012) A development process for usable large scale interactive critical systems: application to satellite ground segments. In Human-Centered software engineering. Springer, Berlin, pp 72–93

Martinie C, Palanque P, Ragosta M, Fahssi R (2013) Extending procedural task models by systematic explicit integration of objects, knowledge and infor-mation. In: Proceedings of the 31st European conference on cognitive ergonomics. ACM, p 23

Martinie C, Palanque P, Winckler M (2011) Structuring and composition mechanisms to address scalability issues in task models. In: Human-Computer interaction–INTERACT 2011. Springer, Berlin, pp 589–609

Moody DL (2009) The "physics" of notations: toward a scientific basis for constructing visual notations in software engineering. IEEE Trans Softw Eng 35(6):756–779

Navarre D, Palanque P, Ladry JF, Barboni E (2009) ICOs: A model-based user interface description technique dedicated to interactive systems addressing usability, reliability and scalability. ACM Trans Comput-Hum Interact (TOCHI) 16(4):18

Norman DA (2013) The design of everyday things: revised and expanded edition. Basic books

Palanque P, Bernhaupt R, Navarre D, Ould M, Winckler M (2006) Supporting usability evaluation of multimodal man-machine interfaces for space ground segment applications using Petri net based formal specification. In: Ninth international conference on space operations, Rome, Italy

Palanque P, Ladry JF, Navarre D, Barboni E (2009) High-Fidelity prototyping of interactive systems can be formal too. In: Human-Computer interaction. New Trends. Springer, Berlin, pp. 667–676

Paternò F (1999) Model-Based design and evaluation of interactive application. Springer, Berlin

Pnueli A (1986) Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. Springer, Berlin, pp 510–584

Reisig W (2013) Understanding Petri Nets—Modeling techniques, analysis methods, case studies. Springer

Silva JL, Fayollas C, Hamon A, Martinie C, Barboni E (2014) Analysis of WIMP and post WIMP interactive systems based on formal specification. Electron Commun EASST 69

Valk R (1998) Petri nets as token objects. In: Application and theory of Petri nets 1998. Springer, Berlin, pp 1–24

Wilson S, Johnson P, Kelly C, Cunningham J, Markopoulos P (1993) Beyond hacking: a model based approach to user interface design. People and computers. University Press, BCS HCI, pp 217–217

Chapter 18
A Public Tool Suite for Modelling Interactive Applications

Marco Manca, Fabio Paternò and Carmen Santoro

Abstract Model-based approaches aim to support designers and developers through the use of logical representations able to highlight important aspects. In this chapter, we present a set of tools for task and user interface modelling useful for supporting the design and development of interactive applications. Such tools can be used separately or in an integrated manner within different types of development processes of various types of interactive applications. This tool suite is publicly available and, as such, can be exploited in real-world case studies and university teaching.

18.1 Introduction

Model-based approaches for interactive applications aim to exploit high-level descriptions that allow designers to focus on the main semantic aspects rather than starting immediately to address implementation details. They have been considered also because such features can be exploited in order to obtain better device interoperability through many possible implementation languages. Even the recent HTML5[1] has adopted some model-based concepts by providing some tags that explicitly characterise the semantics of the associated element. However, this language is limited to graphical user interfaces while we will show that model-based languages (such as the ones supported by the tools presented here) can be exploited to support multimodal user interfaces as well.

[1] https://www.w3.org/TR/html5/.

M. Manca (✉) · F. Paternò · C. Santoro
CNR-ISTI, HIIS Laboratory, Pisa, Italy
e-mail: marco.manca@isti.cnr.it

F. Paternò
e-mail: fabio.paterno@isti.cnr.it

C. Santoro
e-mail: carmen.santoro@isti.cnr.it

B. Weyers et al. (eds.), *The Handbook of Formal Methods in Human-Computer Interaction*, Human-Computer Interaction Series,
DOI 10.1007/978-3-319-51838-1_18

Over the years, research in model-based approaches for interactive applications has led to many approaches and related tools. However, while some approaches identified in the academic field remained of interest of just a limited community of scientists and researchers, other approaches have stimulated interest also at an industrial one, resulting in quite a large community. Such interest has also prompted initiatives at standardisation level. One of them has been the W3C Working Group on model-based UI,[2] which produced documents regarding various aspects, primarily task models and abstract user interfaces.

In this chapter, we present a tool suite that covers various types of model-based support to user interface design, particularly focusing on recent evolutions of such tools in order to meet novel requirements. In particular, Sect. 18.2 provides some background information useful to better understand the contribution described in this chapter, while in Sect. 18.3, we provide an overview of the set of tools belonging to the suite. Then, in the next sections, each considered tool will be presented separately. In particular, we highlight some recent contributions concerning how to support the development of task models also through touch-based mobile devices (see Sect. 18.4), how to obtain model-based generation of multimodal user interfaces (see Sect. 18.5) and how to reverse-engineer Web implementations at various abstraction levels (Sects. 18.6 and 18.7). Finally, at the end of the chapter, we summarise and discuss the main current trends identified within the model-based area, also sketching out future directions for research in this field.

18.2 Background

The community working on model-based approaches for human–computer interaction has mainly considered the abstraction levels that are indicated in the CAMELEON Reference Framework (Calvary et al. 2002). It identifies four abstraction levels: Task and Domain, Abstract UI, Concrete UI, Final UI. In this section, for each level, we describe the most relevant concepts, languages and tools of the proposed suite. This will be useful to introduce the tool suite presented in the next sections. In software engineering communities, the model-based approaches have been considered with slightly different concepts because they have a different scope and do not address the specific issues in user interface design and development. For instance, a different conceptual approach is the model-driven architecture (MDA) proposed by OMG (http://www.omg.org/mda/). The OMG's model-driven architecture (MDA) specifies a generic approach for model-driven software engineering and distinguishes four different levels of abstraction: the computation-independent model (CIM), the platform-independent model (PIM), the platform-specific model (PSM) and the implementation (or implementation-specific model—ISM).

[2] http://www.w3.org/2011/mbui/.

According to (Raneburger et al. 2013), the CAMELEON Reference Framework is compliant to the MDA and can be seen as a specialisation in the context of UI development.

The **Task and Domain models** level considers the tasks that need to be performed for achieving users' goals and the corresponding relationships and domain objects. As mentioned before, in our approach, this level is mainly covered by the ConcurTaskTrees (CTT) notation (Paternò 1999).

The **Abstract User Interface** (AUI) level describes a UI through the interaction semantics, without referring to a particular device capability, modality or implementation technology. In addition, at this level, the behaviour and the description of the data types manipulated by the user interface are specified. For this purpose, we use the MARIA language (Paternò et al. 2009), which also includes a data model, defined by using the standard XML Schema Definition constructs. More specifically, the AUI is composed of *presentations* that contain *interactors* and/or *composition operators*. The interactors are the elementary user interface elements, which can be either interactive or output-only, while the composition elements indicate how to put together elements that are logically associated. Examples of abstract interactive elements are *single choice*, *multiple choice*, *text edit*, *numerical edit*, *activator* and *navigator*, while examples of output-only elements are those supporting *descriptions* and *alerts*. *Navigator* elements allow users to move from one presentation to another, while *activators* are those elements that activate some functionalities. The composition operators are *grouping* (a group of elements logically connected to each other), *relation* (when two groups of elements have some type of relation, e.g. a set of input elements and a set of buttons to send their values to the server or clear them) and *repeat* (when a set of elements are grouped together since they share the same similar structure). The MARIAE tool (Paternò et al. 2011) provides automatic support for translating CTT task models into MARIA AUI specifications. This is a type of transformation that cannot be performed through simple mappings because task models and user interfaces involve different concepts and relationships. Since the user interface is structured into presentations, the first step is to identify them from the task model. For this purpose, we developed an algorithm that first identifies the so-called presentation task sets (PTSs): each PTS is a set of tasks enabled in the same period of time and thus it should be associated with a given abstract presentation. This is done by taking into account the formal semantics of the CTT temporal operators. After the identification of the abstract presentations, the interactors and the dialogue models associated with them are generated taking into account the following: (i) temporal relations among tasks (because the user actions should be enabled in such a way to follow the logical flow of the activities to perform); (ii) task hierarchy (because if one task is decomposed into subtasks, it is expected that the interactions associated with the subtasks are logically connected and this should be made perceivable to the user; thus, a corresponding grouping composition operator should be specified in the abstract specification); (iii) the type of task (which provides useful information to identify the most suitable interaction technique for the type of activity to perform, for instance, if it is a task supporting a selection of one element among several ones, a single choice interactor should be provided).

A **Concrete User Interface** (CUI) provides platform-dependent but implementation language-independent details of a UI. A platform is a set of software and hardware interaction resources that characterises a given set of devices, such as desktop, mobile, vocal and multimodal. Each CUI is a refinement of an AUI, specifying how the abstract concepts can be represented in the current platform taking into account the interaction modality available. MARIA currently supports various platforms: *graphical desktop and mobile*, *vocal*, *gestural*, and *multimodal* (which combines graphical and vocal modalities in various ways). For instance, in the graphical concrete description corresponding to Web implementations, the abstract element *single choice* is refined into either a *radio button*, or a drop-down *list*, or a *list box*, or an *image map*, while the multiple choice is refined into a *check box* or a *list box*, the *navigator* into a *link button* or an *image map*. The other elements are similarly refined to support the semantics of the corresponding abstract elements. In the case of a multimodal concrete language, we have to consider refinements for multiple modalities and indicate how to compose them. In particular, the MARIA concrete language for composing graphical and vocal modalities is based on the two previously defined concrete languages (one for the graphical and one for the vocal modality). It adds the possibility to specify how to compose them through the CARE (Complementarity, Assignment, Redundancy, Equivalence) properties (Coutaz et al. 1995), which can occur between the interaction techniques available in a multimodal user interface. Indeed, as we introduced before, the MARIA abstract language structures a user interface into a set of presentations. Each presentation has composition operators, which contain interactors that can be either interaction or only-output interface basic components. Such interactors can have event handlers associated with them indicating how they react to events. Each of these elements of the language, ranging from presentations to elementary interactors, has different refinements for the graphical and the vocal modality, and in the multimodal concrete language, we indicate how to compose them. Thus, a multimodal presentation has associated both graphical attributes (such as background colour or image or font settings) and vocal attributes (such as speech recogniser or synthesis attributes). For example, a grouping composition in the multimodal concrete language can exploit both visual aspects (using attributes such as position, dimension, border backgrounds) and vocal techniques (i.e. inserting keywords/sounds/pauses or changing synthesis properties). The interactors are enabled to exploit both graphical events (associated with mouse and keyboards) or vocal-specific ones (such as no input or no match input or help request).

The **Final UI** (FUI) corresponds to the UI in terms of some implementation language. From the CUI, different final user interfaces can be derived. A FUI can be represented in any UI programming (e.g. Java UI toolkit) or markup language (e.g. HTML).

18.3 The Proposed Tool Suite

In this section, we briefly introduce the main characteristics of the tools that will be presented in detail in this chapter. The set of tools discussed in this chapter covers the various abstraction levels indicated in the CAMELEON Reference Framework. In particular, the Task and Domain level is covered by the CTT language, which is supported by the desktop ConcurTaskTrees Environment (CTTE) and the ResponsiveCTT tool (see Sect. 18.4). The other abstraction levels can be specified by using the MARIA language and are supported by the MARIAE tool. The tool set supports the possibility of transforming representations at one abstraction level into another, which supports forward engineering transformations, the ones going from the most abstract levels down to the more concrete ones. In addition, it is worth noting that each described tool, apart from supporting such transformations, also enables the user to handle (i.e. create, edit, save) the concerned relevant models. There is another tool that supports transformations going from the most concrete levels to the higher ones (reverse engineering tool). Figure 18.1 provides an overview of the languages and the tools that have been developed. A different colour has been assigned to each of the three tools: light grey to ReverseCTT (see Sect. 18.7), black to MARIAE (see Sect. 18.5) and grey to ReverseMARIA (see Sect. 18.6).

The task model language has been recently adopted as the basis for the task model standardisation within W3C.[3] The language also supports the possibility of specifying task pre- and post-conditions (even combined in complex Boolean expressions), which can be exploited not only within an interactive simulation of the specification but also in the user interface generation process in order to derive meaningful and consistent user interface implementations.

CTT is supported by two main tools: Desktop CTTE and ResponsiveCTT. Desktop CTTE was also integrated with tools for model checking in order to formally reason about task properties (Paternò and Santoro 2000). Both tools allow the user to exploit the cloud to store and share task models remotely, which also facilitates potential sharing and collaboration among designers. ResponsiveCTT can be accessed through touch-based mobile devices such as smartphones and tablets. The tool is responsive as it provides adapted user interfaces to better support task modelling through various types of devices. For this purpose, it also exploits some information visualisation techniques, i.e. representations that make users comfortably analyse/modify task model specifications, which is especially useful when medium-to-large task models are rendered on the limited screen size of mobile devices.

In addition, in this chapter, we also focus on the relationships between task model specifications and models for user interface definition exploited in multi-device contexts. In particular, we focus on how task models can be exploited to derive user interface models at various abstraction levels and for various platforms,

[3] http://www.w3.org/TR/task-models/.

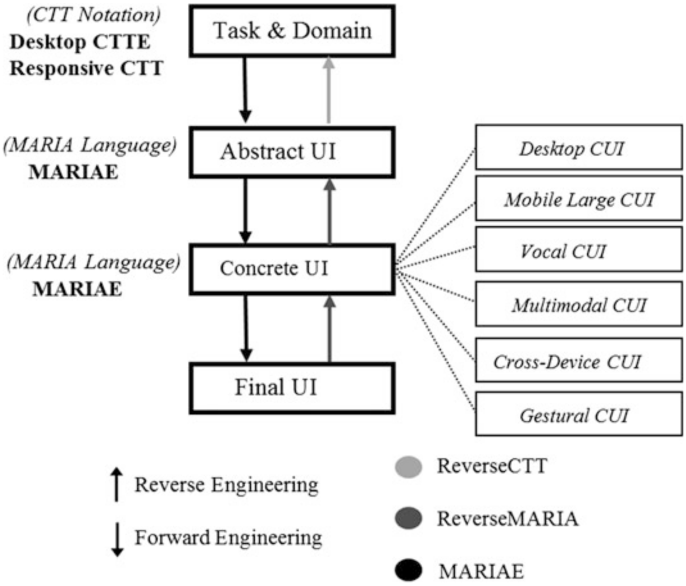


Fig. 18.1 Overview of the languages and the tools proposed

with particular attention to the UI models generated by MARIAE environment. Regarding the latter tool, an aspect addressed is how to provide support for the development of interactive applications able to access and exploit Web services, even from different types of interactive devices. The MARIAE tool is able to aid in the design of new interactive applications that access pre-existing Web services, which may also contain annotations supporting the user interface development. In addition, in MARIAE, various automatic generators are available for a number of platforms (e.g. desktop, mobile multimodal, vocal, distributed), even directly from a CTT task model. Indeed, one of the advantages of using a model-based language such as MARIA over modern languages such as HTML5 is the ability to describe and support even multimodal interfaces by still exploiting a set of core concepts. Another integrated contribution is a reverse engineering tool, which will be discussed to show how to derive interactive systems' descriptions at the various possible abstraction levels starting with Web application implementations.

The development of the tools presented in this chapter has been done according to a number of requirements that were identified and evolved over the years. Regarding the software tools covering Abstract UI, Concrete UI and Final UI, some requirements have been discussed in previous work. For instance, for a predecessor of MARIAE, requirements were identified in Mori et al. (2004): the tool should support mixed initiative, handle multiple logical levels described through XML-based languages and enable different entry points within the multilevel UI specification of CAMELEON conceptual framework. Further requirements for the MARIAE tool were identified in Paternò et al. (2009): the tool should provide designers with effective control of the user interfaces produced, the transformations for generating the corresponding implementations should not be hard-coded in the tool, the tool should provide support also for creating front ends for applications in which the functionalities are implemented in Web services.

Also for the Task and Domain level and in particular the CTTE Desktop tool, a number of requirements were identified in previous work (Mori et al. 2002): the tool is expected to provide support for modelling and analysis of single-user and cooperative CTT task models, more specifically, visualise/edit/simulate/check the validity of task models, save task models in various formats, support multiple interactive views of task model specifications, support the generation of task models from Web service descriptions (WSDL). ResponsiveCTT, as being more recently developed having in mind mobile devices, posed further requirements which will be discussed in detail in Sect. 18.4.1.

Regarding the tool suite on a more comprehensive level, a main requirement was that it should support all the levels of the CAMELEON framework and associated top-down/bottom-up transformations, which is fully satisfied in our case (see Fig. 18.1).

18.4 Task Modelling

This section introduces the CTT notation for task models. Such notation is supported by two tools: Desktop CTTE is a Java desktop application, and ResponsiveCTT is a responsive Web application, which can be used from various types of devices.

18.4.1 CTT Task Models

Task models indicate the tasks that need to be performed for achieving users' goals by interacting with the UI. Various task model notations are available, e.g. UsiXML (Limbourg et al. 2005), ConcurTaskTrees (Paternò 2000), Hamsters (Martinie et al. 2011), which differ on aspects such as the syntax, the level of formality and/or the operators. A key factor for their adoption is the availability of automatic environments that support model editing, analysis and transformation. However, not all task model notations are supported by (publicly available) tools, and the vast majority of such tools are limited to desktop-based environments. ConcurTaskTrees allows designers to concentrate on the activities that users aim to perform, which are the most relevant aspects when designing interactive applications, and encompasses both user- and system-related aspects. This approach allows designers to avoid dealing with low-level implementation details, which at the design stage would obscure the decisions to make. CTT has a hierarchical structure: this is generally considered as very intuitive since often when people have to solve a problem they tend to decompose it into smaller problems still maintaining the

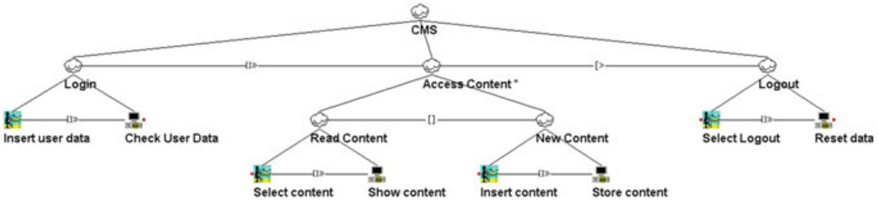


Fig. 18.2 Example of a CTT task model

relationships among the various parts of the solution. Figure 18.2 shows an example of task model related to interacting with a content management system.

The hierarchical structure of this specification has two advantages: it provides a wide range of granularity allowing large and small task structures to be reused, and it enables reusable task structures to be defined at both low and high semantic levels. A rich set of temporal relationships between tasks have also been identified. How the performance of the task is allocated is indicated by the related category and is explicitly represented by using icons. While the category of a task indicates the allocation of its performance, the type of a task allows designers to classify tasks depending on their semantics. Each category has its own types of tasks. In the *interaction* category, examples of task types are as follows: *selection*, when the task allows the user to select some information; *control*, when the task allows the user to trigger a control event that can activate a functionality; *editing*, when the task allows the user to enter a value; *zooming*, the possibility to zoom in/out; *filtering*, the possibility to filter out some irrelevant details. Depending on the type of task to be supported, a suitable interaction or presentation technique will be selected in the development process. Frequency of use is another useful type of information because the interaction techniques associated with more frequent tasks need to be better highlighted to obtain an efficient user interface. The platform attribute (desktop, cellular) allows the designer to indicate the types of devices the task is suitable for. This information is particularly useful in the design of applications that can be accessed through multiple types of platforms, because some tasks could not be available in some platforms. For each task, it is possible to indicate the objects that have to be manipulated to perform it. Since the performance of the same task in different platforms can require the manipulation of different sets of objects, it is possible to indicate for each platform which objects should be considered. It is also possible to exploit such objects to define pre- and post-conditions associated with the tasks. The presence of objects and conditions is indicated in the graphical representation of the model through some cues (see the small rounded icons beside some task icons in Fig. 18.2). Objects can be shared across multiple tasks, and each involved task can have different conditions associated with the object.

The language is also able to model multiuser applications through specific task models for each role involved and an additional model to indicate their relationships. The notation has long been supported by a Java-based desktop tool, the ConcurTaskTrees Environment (CTTE) (Mori et al. 2002), which provides a set of functionalities for editing, analysis and interactive simulations of the dynamic performance of sequence of tasks. It can be downloaded at http://giove.isti.cnr.it/tools/CTTE/home.

18.4.2 ResponsiveCTT

As mobile devices are now indisputably part of everyday life and widely applied in various domains, we judged it interesting to investigate the possibilities offered by them for task modelling. We focused on truly mobile devices, i.e. those that can be fully and comfortably used even when the user is on the go. As for task modelling tools, some approaches have been put forward, such as CTTE, HAMSTERS and K-MADe (Caffiau et al. 2010), although they all focused on desktop platforms. Attempts to consider modelling tasks on a different platform were carried out mainly in Eggers et al. (2013) and Spano and Fenu (2014). So, apart from a few attempts considering task modelling for mobile use, tools supporting task modelling have been mainly confined to considering desktop platforms. Thus, we have also investigated the possibilities of touch-based modelling on mobile devices through a new tool (ResponsiveCTT[4]). The development of the tool was driven by a number of requirements identified in our experience of several projects and work in which task modelling was exploited. First, in order to widen the impact and possible adoption of the tool, it was developed as a Web application exploiting HTML5, CSS3 (for the presentation) and JavaScript (for the dynamic aspects) accessible by any browser-enabled device. Also, the tool was conceived to support responsive design to effectively adapt the model representations to the screen area of the device, which is particularly important when mobile devices are used. In addition, since the screen size of mobile devices is a key factor for an effective analysis and editing of task models, we judged relevant to exploit information visualisation techniques for dynamically representing task models so as to harness the power of visualization anytime, anywhere, while requiring more limited cognitive effort than in stationary settings. Finally, to store and share task models remotely, the application is cloud-based, which also facilitates collaboration among users.

With the new tool, to edit the task model on a touch-based mobile device, users can touch an empty screen area to create a new task root and perform a tap gesture on a task to edit it, i.e. edit name, type, objects and precondition or add a task as a sibling or as a sub-task, copy/cut and paste a task and/or its children.

In the tool, a focus+context, fisheye-based visualisation technique (Cockburn et al. 2008) has been used as an interactive visual aid to support the visualization, exploration and navigation of task models on touch-based mobile devices, where precise task selections are difficult due to their small screen and limited accuracy of touch commands. In particular, the visualisation of a task model is arranged so that

[4] Available at http://ctt.isti.cnr.it.

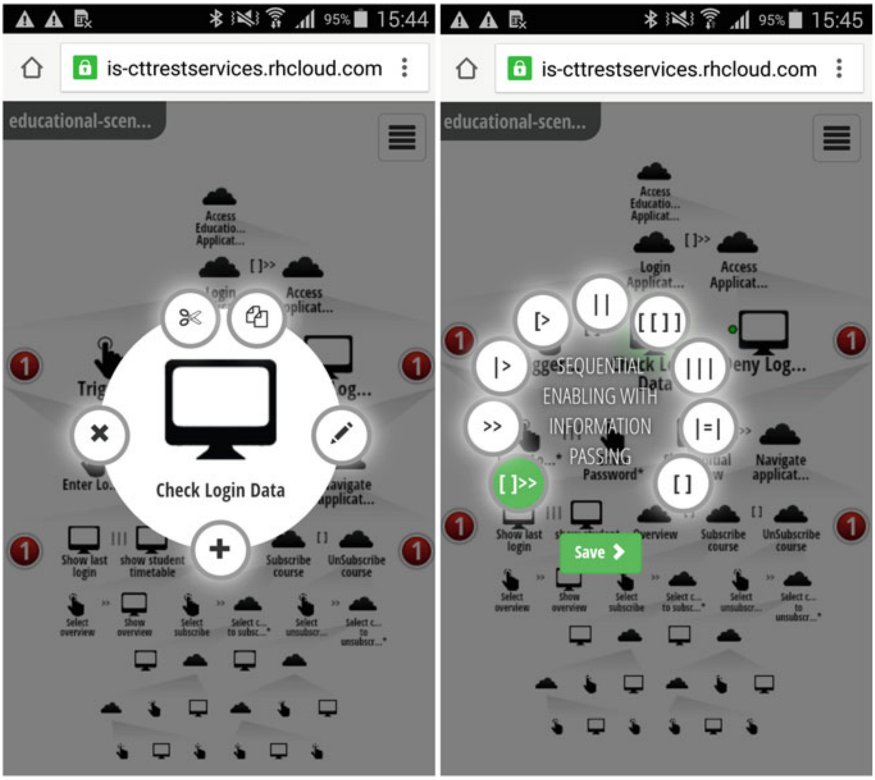


Fig. 18.3 UI for editing task (*left*) and operators (*right*)

the tasks closest to the one that currently has the focus (the task *Check Login Data* in Fig. 18.4) are more emphasised in terms of larger screen space of the associated icons, with a progressive fall-off in magnification towards the upper/bottom model edges. So, in our case, the focus area is determined by the selected task and includes its nearest siblings and children tasks, while the “context” is composed of the remaining tasks of the model. When selecting a task, the application sets the focus to that task and changes the fisheye visualisation of the model accordingly. When users tap on the task currently having the focus, a semi-transparent circular menu appears (see Fig. 18.3), showing the actions that can be executed on that task: change its properties, add new tasks, add objects and pre- and post-conditions associated with it. When users select a new task, the visualisation of the task model is dynamically recalculated to show the new task having the focus on a prominent position of the window and rearrange the model visualisation accordingly. By selecting the icon of a temporal operator, a contextual menu appears, visualising the possible operators (see Fig. 18.3, right part), presented according to their priority. Furthermore, a number of gestures are supported: “pinch to zoom” for zooming on the task model, “swipe down/up” to move up/down in the task model level. It is

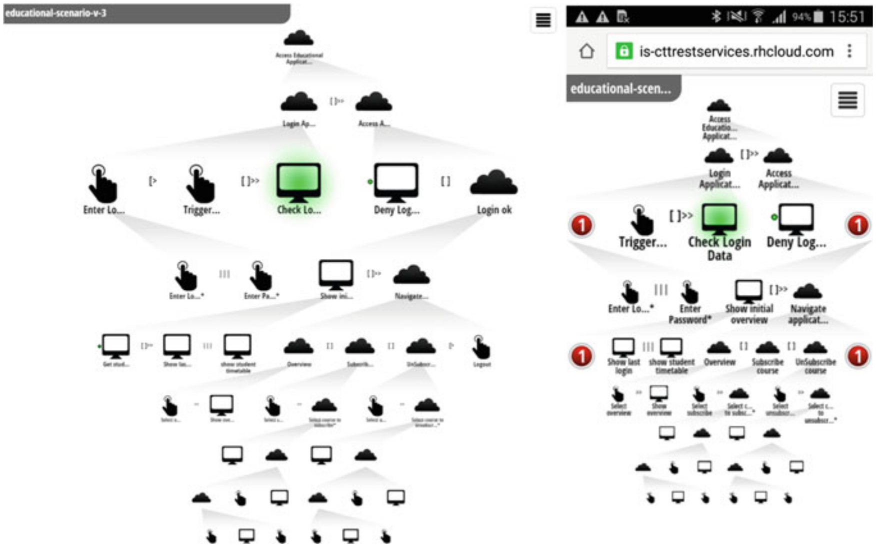


Fig. 18.4 (*left*) Complete task model (desktop); (*right*) Task model on a mobile platform

also possible to change various task attributes (e.g. task name, category) and add the specification of objects manipulated by the task and pre-/post-conditions associated with it. Finally, users can also save models in a dedicated cloud-based service.

As mentioned, the task that has the focus is supposed to be the currently most "important" one; thus, it is always placed in a central position within the working window and highlighted by a specific colour. More generally, every task has a *degree of interest* dynamically calculated, which is inversely proportional to its *distance* from the currently selected task. The dimension of the graphical representation of each task varies according to this distance factor: the further the focused task is, the smaller the icon of the considered task will be, where the "distance" between two tasks is represented by the number of levels that need to be traversed in the model to reach one task from the other. This algorithm is performed whenever a task model is loaded or any editing operation modifies its structure. When it becomes difficult to graphically represent some tasks in a way sufficiently perceivable by the user because of the limited space, they are replaced with a cue-based technique that shows numbers indicating how many tasks are currently hidden (see Fig. 18.4 right part). The numbers are visualised at the same task level and side as the hidden tasks, with the size of the numbered icon proportional to the number itself. By interactively selecting a numbered icon, the previously hidden tasks at the considered level are shown.

Tasks can have some preconditions visualised in the task model through a small coloured rounded icon close to the task icon, whose colour changes during the simulation phase according to the precondition state: if it is true the colour is green,

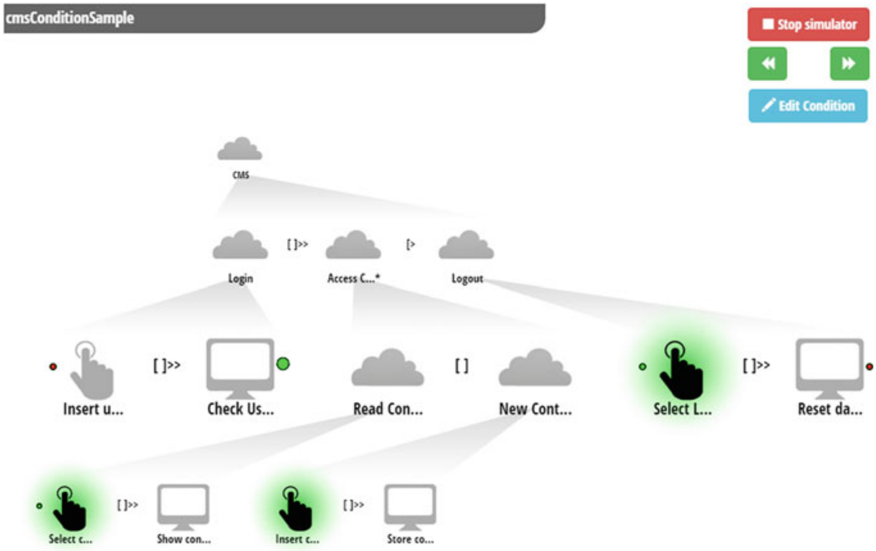


Fig. 18.5 Rendering tasks with preconditions within the ResponsiveCTT simulator

otherwise it is red. Figure 18.5 shows an example while the task model simulator is running.

First-user studies (Anzalone et al. 2015) delivered encouraging results in how the tool supported users in handling task models on mobile devices. They also indicate that tablets are more suitable for supporting task modelling than smartphones since modelling tasks are medium-/long-term, cognitively demanding activities which are better performed when the supporting devices allow for performing them in a comfortable manner.

18.5 Modelling and Generating Multimodal User Interfaces

While the previous section focused on the Task model level, in this section, we describe how to model user interfaces and obtain a corresponding implementation. MARIA has an abstract language and various concrete refinements that depend on the modalities considered. In order to illustrate how to model and generate user interfaces, it can be fruitful to consider the multimodal case, since little work has been dedicated to it and it is an important trend in the HCI area to obtain more natural interfaces. MARIA was developed to address various limitations in previous model-based languages such as TERESA (Berti et al. 2004). One important contribution of MARIA and the associated environment (MARIAE[5]) is the possibility of generating multimodal interactive applications, which can be executed in any browser-enabled device supporting the Web Speech APIs. This is relevant since most model-based approaches have focused only on graphical user interfaces. A model-based Framework for Adaptive Multimodal Environments (FAME) has been proposed in Duarte and Carrico (2006), but it does not provide support for automatic application development, as in our case. Octavia et al. (2010) describe an approach to design context-aware applications using a model-based design process, but they do not address multimodal adaptation. The model-based approach was also considered in Sottet et al. (2007) for its flexibility, although run-time adaptation is considered only by regenerating the whole UI and multimodal adaptation is not addressed. MyUI (Peissner et al. 2011) provides a framework for run-time adaptations while supporting accessibility. However, its design pattern-based approach can quickly become cumbersome.

Our solution is able to generate multimodal interactive Web applications and does not require using any particular API in addition to standard HTML, CSS and JavaScript. Such implementations are obtained from the concrete language in MARIA addressing multimodal user interfaces. It supports various combinations of graphical and vocal modalities, and it can be easily extended to address other modalities. At various granularities levels, it is possible to indicate the modalities that should be used to support the considered user interface part. There are four possibilities indicated by the CARE properties: *complementarity*, which means that part is associated with one modality and part with another one; *assignment* indicates that only one modality should be considered; *redundancy* is used to indicate that multiple modalities should be used for the same user interface part; *equivalence* is used when there is the possibility to choose one modality or another for the corresponding user interface elements. Depending on the modality indicated, the corresponding concrete attributes are specified.

The multimodal user interface generator produces HTML implementations structured in two parts: one for the graphical user interface and one for the vocal one. The implementation exploits the Web Speech APIs[6] for automatic speech recognition (ASR) and text-to-speech synthesis (TTS). The generator annotates the elements that need vocal support. Such annotations are detected through scripts that are activated when the page is loaded and call the vocal libraries for ASR and TTS. In particular, each UI element is annotated with a specific CSS class in the implementation generation, according to the indications of the CARE properties. If it contains a vocal part and the CARE value is redundancy or vocal assignment, the class *tts* for the *output* elements and the *prompt* part of *interaction element* are added, while the class *asr* is added for the input parts of interaction elements only if the CARE value of this part is *equivalent* or *vocal assignment*. The generated elements are marked with these classes because the multimodal scripts use them to

[5] Available at http://giove.isti.cnr.it/tools/MARIAE/home.

[6] https://dvcs.w3.org/hg/speech-api/raw-file/tip/speechapi.html.

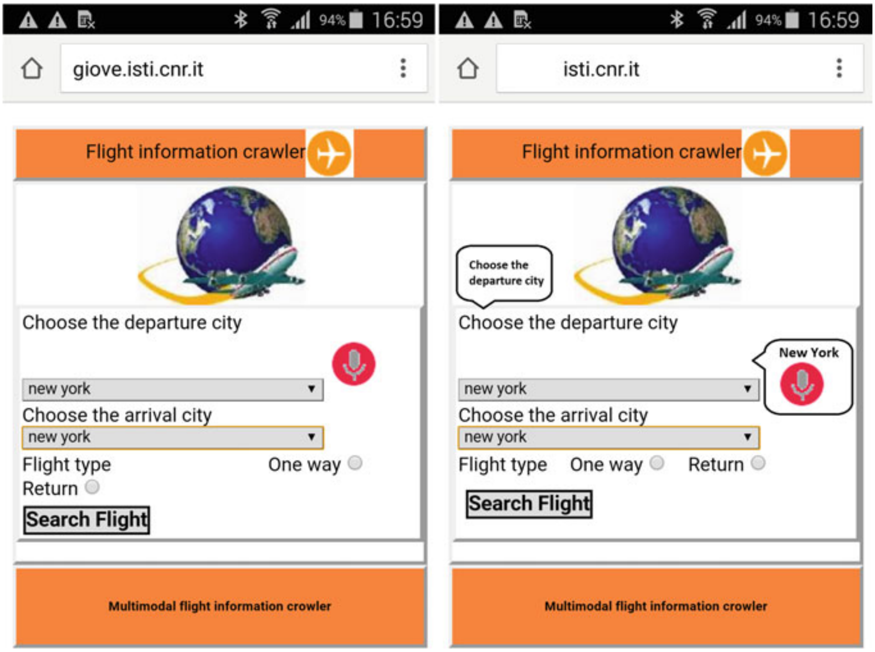

16:59
giove.isti.cnr.it
Flight information crawler
Choose the departure city
new york
Choose the arrival city
new york
Flight type
One way
Return
Search Flight
Multimodal flight information crowler
16:59
isti.cnr.it
Flight information crawler
Choose the departure city
Choose the departure city
New York
new york
Choose the arrival city
new york
Flight type
One way
Return
Search Flight
Multimodal flight information crowler

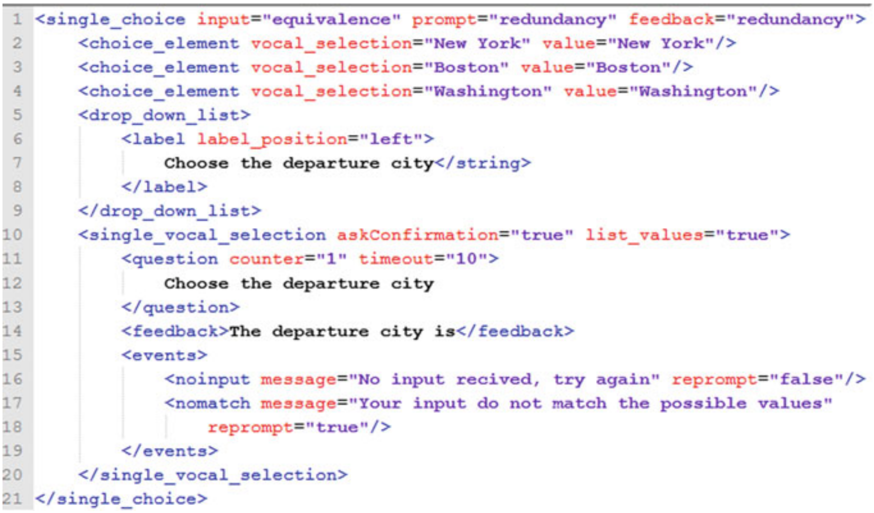
Fig. 18.6 Multimodal generated user interface

identify all the graphical elements having an associated vocal part. Figure 18.6 shows an implementation example for multimodal interaction generated from a concrete user interface (CUI).

In Fig. 18.7, we consider an excerpt of the MARIA multimodal concrete specification related to the definition of the single choice element corresponding to the selection of the departure city in a task aimed to search for a particular flight.

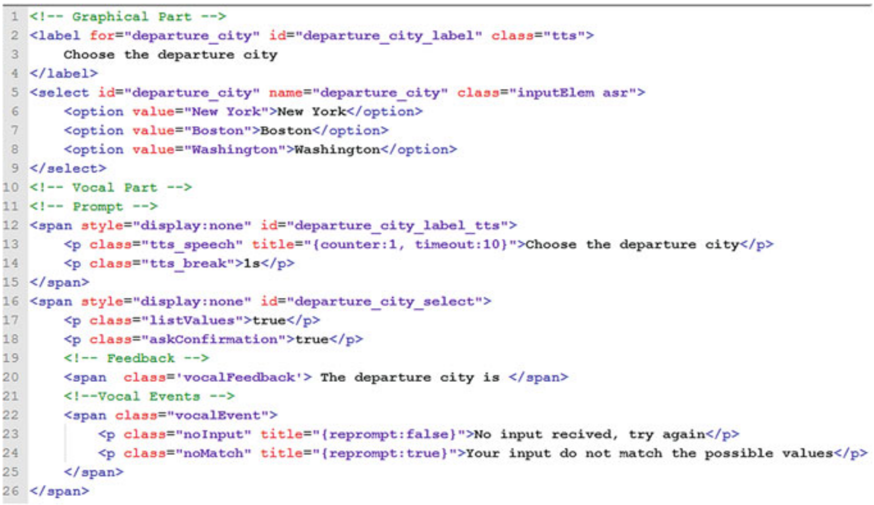
The considered UI element defines the CARE properties for each interaction part (see Fig. 18.7 line 1): for the input element, the CARE value is *equivalent*, which means that input can be entered through either the graphical modality or the vocal one; prompt and feedback phases are both *redundant*; thus, they are visible and vocally synthesised. The single choice element is refined as a drop-down list (line 5) from the graphical point of view and as a single vocal selection (line 10) for the vocal specification. The vocal part defines the vocal prompt, feedback and the events related to the vocal interaction.

Figure 18.8 shows an excerpt of the code generated from the MARIA multimodal specification in Fig. 18.7. Since the single choice element is specified in a multimodal way (see the CARE properties in Fig. 18.7 line 1), the corresponding generated code is composed of a graphical part (Fig. 18.8 from line 1 to 9) and a vocal part (Fig. 18.8 from line 10 to 26) that are not visible but still accessible from the multimodal scripts described before. The label element is annotated with the *tts* class to indicate that there is an associated vocal part with the same id plus the *tts*



```
<single_choice input="equivalence" prompt="redundancy" feedback="redundancy">
    <choice_element vocal_selection="New York" value="New York"/>
    <choice_element vocal_selection="Boston" value="Boston"/>
    <choice_element vocal_selection="Washington" value="Washington"/>
    <drop_down_list>
        <label label_position="left">
            Choose the departure city</string>
        </label>
    </drop_down_list>
    <single_vocal_selection askConfirmation="true" list_values="true">
        <question counter="1" timeout="10">
            Choose the departure city
        </question>
        <feedback>The departure city is</feedback>
        <events>
            <noinput message="No input recived, try again" reprompt="false"/>
            <nomatch message="Your input do not match the possible values"
                reprompt="true"/>
        </events>
    </single_vocal_selection>
</single_choice>
```

Fig. 18.7 Excerpt of the MARIA multimodal concrete specification

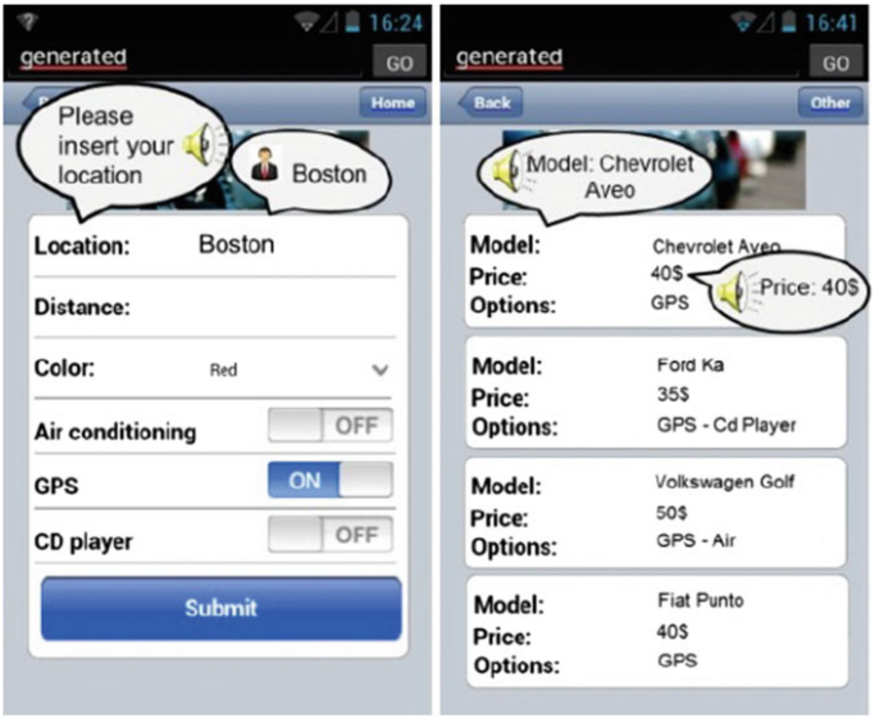


```
1  <!-- Graphical Part -->
2  <label for="departure_city" id="departure_city_label" class="tts">
3      Choose the departure city
4  </label>
5  <select id="departure_city" name="departure_city" class="inputElem asr">
6      <option value="New York">New York</option>
7      <option value="Boston">Boston</option>
8      <option value="Washington">Washington</option>
9  </select>
10 <!-- Vocal Part -->
11 <!-- Prompt -->
12 <span style="display:none" id="departure_city_label_tts">
13     <p class="tts_speech" title="{counter:1, timeout:10}">Choose the departure city</p>
14     <p class="tts_break">1s</p>
15 </span>
16 <span style="display:none" id="departure_city_select">
17     <p class="listValues">true</p>
18     <p class="askConfirmation">true</p>
19     <!-- Feedback -->
20     <span  class='vocalFeedback'> The departure city is </span>
21     <!--Vocal Events -->
22     <span class="vocalEvent">
23         <p class="noInput" title="{reprompt:false}">No input recived, try again</p>
24         <p class="noMatch" title="{reprompt:true}">Your input do not match the possible values</p>
25     </span>
26 </span>
```

Fig. 18.8 Excerpt of multimodal generated code

suffix (Fig. 18.8 line 2 and line 12). Since the select element is an input element, it is annotated with the *asr* class to indicate that there is an associated vocal part (Fig. 18.8 line 5 and 16).

Figure 18.9 shows another example of multimodal user interfaces for a car rental application. The first page supports a multimodal search through which users can provide the search parameters (also using the vocal modality): for all the interaction



Location: Boston

Distance:

Color: Red

Air conditioning OFF

GPS ON

CD player OFF

Model: Chevrolet Aveo
Price: 40$
Options: GPS

Model: Ford Ka
Price: 35$
Options: GPS - Cd Player

Model: Volkswagen Golf
Price: 50$
Options: GPS - Air

Model: Fiat Punto
Price: 40$
Options: GPS

Fig. 18.9 Multimodal car rental application

elements, the prompt and the feedback part are redundant (thus they are rendered both graphically and vocally), while the input part is equivalent (users can choose either modality). The second page is the search result, rendered in a redundant way as well: all the output elements are both graphical and vocal.

18.6 Reverse Engineering User Interface Logical Descriptions

While the previous two sections mainly analysed forward engineering transformations, in this section, we analyse ReverseMARIA, a tool able to reverse any web page (local or remote) to build the corresponding specification in the MARIA graphical Concrete UI language, and then it is also possible to obtain its Abstract UI specification. The tool is available at https://hiis.isti.cnr.it:8443/WebReverse/indexRev.html, and its development was aimed to facilitate building models associated with existing applications. Such models can then be used to obtain the specification of user interfaces more suitable for different contexts of use, for example for devices with different features (Paternò et al. 2008).

In general, there are two main approaches to reverse engineering user interfaces: pixel-based and code-based. Pixel-based approaches analyse the correlations between the display's pixels to find the set of interface components (buttons, forms) and the hierarchy amongst them. Although being implementation language—neutral—see Dixon et al. (2014) as an example—the main problem of pixel-based approaches is that they are able to retrieve only information about the visual presentation of UIs, and not about the behaviour of components, because they depend on the hidden source code implementation. Source-code reverse engineering in turn can be done in two ways: using static analysis through the application code, or analysing the running application in a dynamic way (dynamic analysis). It is also possible to perform hybrid analysis when static and dynamic analyses are used together. There are various tools that exploit the static analysis-based approach (see e.g. Bernardi et al. 2009a; Da Silva 2010). The benefit of static analysis is that all the information is stored in the code and ready to be processed, but it is not able to retrieve the part of the implementation that is not detectable unless the code is executed. Dynamic analysis solves this problem by analysing the behaviour of the user interface during execution. Examples of dynamic analysis are described in Maras et al. (2011) and (Morgado et al. 2010). For Web applications, a risk related to dynamic approaches is leaving many interface states unexplored and thus obtaining an incomplete interface model. For this reason, a hybrid analysis approach (see Silva and Campos 2013; Li and Wohlstadter 2008) can be more thorough with respect to either static or dynamic techniques.

We followed a novel hybrid approach for reverse engineering web pages to enable analysing them both when stored on a server and when loaded client side, also considering the current internal state of the UI and the user inputs. Our tool reverse-engineers web pages by taking as input the DOM representation which can be obtained in two ways: (i) getting the input page through an http request to the server, and generating the DOM from the HTML code using an external library[7]; (ii) serialising the HTML DOM on the client side through either the browser or a proxy. In addition to the extraction of DOM, the tool

- Associates the significant CSS properties with the related HTML elements;
- For each event associated with an HTML element, determines the type of event, the JavaScript functions invoked on its activation and their parameters;
- Transforms the result obtained into an equivalent MARIA element;
- Serialises the MARIA elements in an XML file using JAXB[8] technology.

[7] We used the Java Tidy Parser, available at http://tidy.sourceforge.net/, modified by us to handle HTML5 tags as well.

[8] Java Architecture for XML Binding.

18.6.1 The Reverse Algorithm

The tool has a first pre-processing phase in which it creates the DOM (see Fig. 18.10), moves all the JavaScript nodes to an external file and stores in cache memory all the CSS properties that are in external files or in style nodes. At the beginning, it creates the MARIA *interface* element, which contains information regarding the MARIA version and schema, and a *presentation* element, which contains the default page properties taken from the HEAD part of the page (e.g. page title). Then, it performs a depth-first search visit of the DOM starting from the *body* node. For each node, it first analyses the type and its attributes, then adds the id attribute if it is not set and introduces an attribute used to analyse the nodes' textual properties. Then, it analyses the neighbouring node to retrieve correlations between nodes and classifies single nodes according to their content and visual properties. The result obtained is the logical representation of the DOM structure. The following example (involving the Yahoo home page) shows how the numbered parts of the HTML structure and CSS properties of the page shown in Fig. 18.11 are transformed into the MARIA descriptions shown in Figs. 18.12 and 18.13.

Figure 18.12 shows the MARIA elements (a grouping and a description element) that correspond to the HTML elements indicated by (1) in Fig. 18.11, while MARIA attributes correspond to CSS properties.

In Fig. 18.13, the red part (2) represents the input text element in the considered web page (see Fig. 18.11), while the green part (3) represents the submit button.

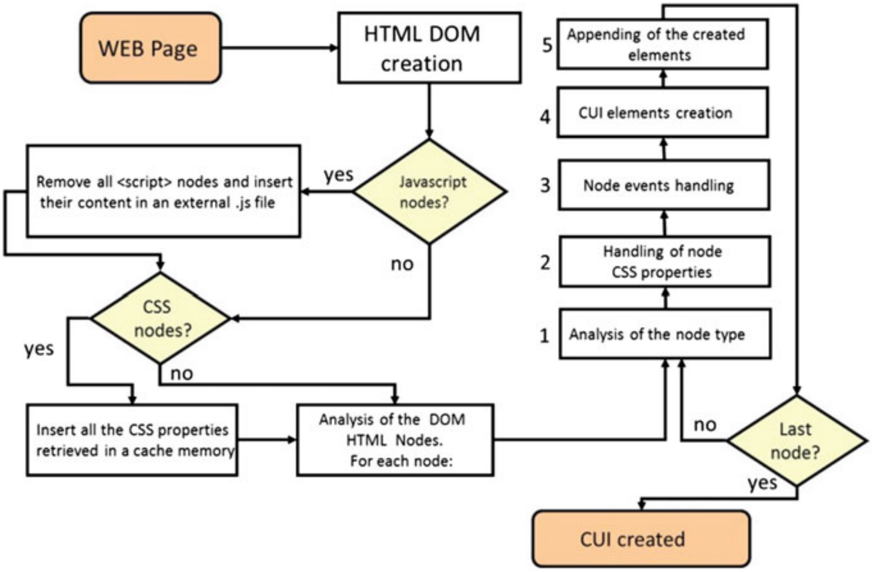

WEB Page
HTML DOM creation
Javascript nodes?
yes
no
Remove all <script> nodes and insert their content in an external .js file
CSS nodes?
yes
no
Insert all the CSS properties retrieved in a cache memory
Analysis of the DOM HTML Nodes. For each node:
1 Analysis of the node type
2 Handling of node CSS properties
3 Node events handling
4 CUI elements creation
5 Appending of the created elements
Last node?
no
yes
CUI created

Fig. 18.10 Reverse engineering algorithm

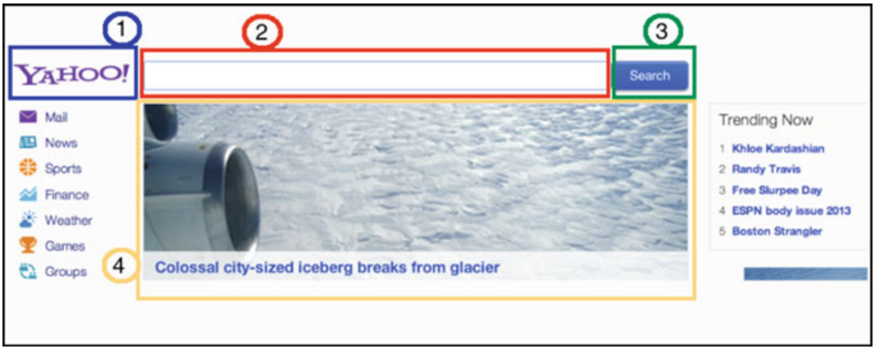
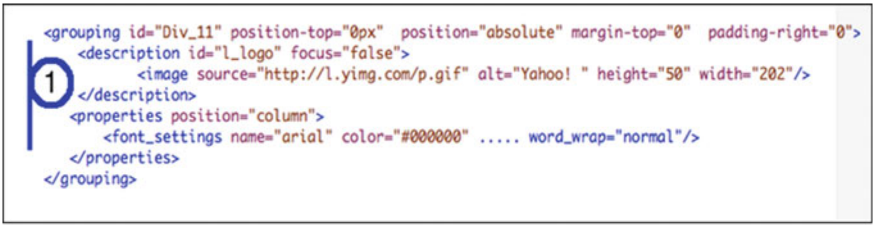
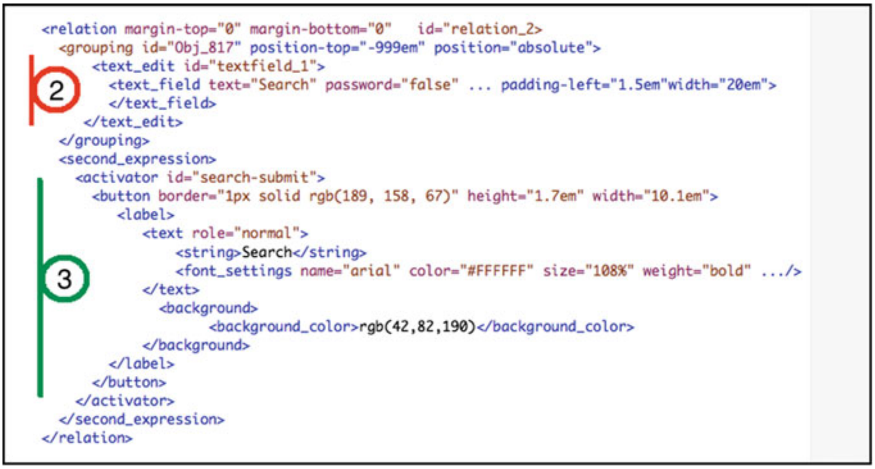


Fig. 18.11 Yahoo Home page



```
<grouping id="Div_11" position-top="0px"  position="absolute" margin-top="0"  padding-right="0">
    <description id="l_logo" focus="false">
          <image source="http://l.yimg.com/p.gif" alt="Yahoo! " height="50" width="202"/>
    </description>
   <properties position="column">
       <font_settings name="arial" color="#000000" ..... word_wrap="normal"/>
   </properties>
</grouping>
```

Fig. 18.12 Result corresponding to element 1



```
<relation margin-top="0" margin-bottom="0"   id="relation_2>
  <grouping id="Obj_817" position-top="-999em" position="absolute">
      <text_edit id="textfield_1">
        <text_field text="Search" password="false" ... padding-left="1.5em"width="20em">
        </text_field>
     </text_edit>
  </grouping>
  <second_expression>
    <activator id="search-submit">
      <button border="1px solid rgb(189, 158, 67)" height="1.7em" width="10.1em">
         <label>
            <text role="normal">
                <string>Search</string>
                <font_settings name="arial" color="#FFFFFF" size="108%" weight="bold" .../>
            </text>
              <background>
                    <background_color>rgb(42,82,190)</background_color>
            </background>
        </label>
      </button>
    </activator>
  </second_expression>
</relation>
```

Fig. 18.13 Results corresponding to elements 2 and 3

Parts 2 and 3 are contained in a relation MARIA element, which represents an HTML form.

Part 4 in Fig. 18.11 corresponds to a link, with its label composed of a picture and a text. This HTML part corresponds to a MARIA navigator element, along with a connection indicating the target presentation.

18.7 Reverse Engineering Task Models

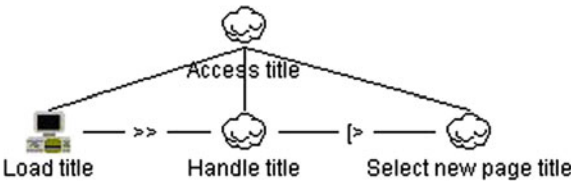
In this section, we present the ReverseCTT tool, which covers the Abstract UI-to-Task Model transformation of the CAMELEON Reference Framework. While ReverseMARIA is a tool able to reverse any web page (local or remote) and build the corresponding specification in the graphical desktop concrete language of MARIA and then obtain its abstract description, ReverseCTT reverses the MARIA AUI specification into a CTT task model description. Thus, it has a more modular approach than WebRevenge (Paganelli and Paternò 2003b), which aimed to create task models directly from the website code.

The ReverseCTT process is articulated into a number of rules. In order to understand them, it is important to briefly remind the characteristics of the input that this transformation receives, namely the AUI. Every AUI is structured into a set of *presentations*, and each presentation has a *name* attribute. The relationships between the AUI presentations are modelled through *connections,* which can be elementary or complex, and which mainly support navigation between the different presentations belonging to each AUI. In other terms, connections are the logical/abstract counterpart of navigational links in Web UIs. The structure of each presentation is articulated into a set of abstract *interactors* (which can have different *types* ranging from editing, navigation, choice, to description and activators), whose relationships are modelled through *operators* (e.g. grouping, relation) defined in the language. Our assumption is that, after a user selects a presentation, the same pattern repeats, regardless of the specific presentation selected. This pattern is the following (for each presentation): load the (selected) presentation, handle the presentation (for user's goals) and then select a new presentation.

Having said that, the idea is that for each AUI presentation, a new task model is created, whose root name is derived by the name of the AUI presentation (by concatenating "Access" + <AUI presentation name>). The root task just created has three children: one is an application task ("Load" + <AUI presentation name>. This application task is followed (through an enabling operator) by an abstract task ("Handle" + <AUI presentation name>), which in turn is disabled by a second abstract task ("Select new page" + <AUI presentation name>). The resulting CTT task model after this step is shown in Fig. 18.14.

In the second step, the process analyses the elements of type "*elementary connection*" in the AUI, which mainly represent HTML anchors/links. For instance,

Fig. 18.14 From AUI to CTT: the model after the first step of the transformation



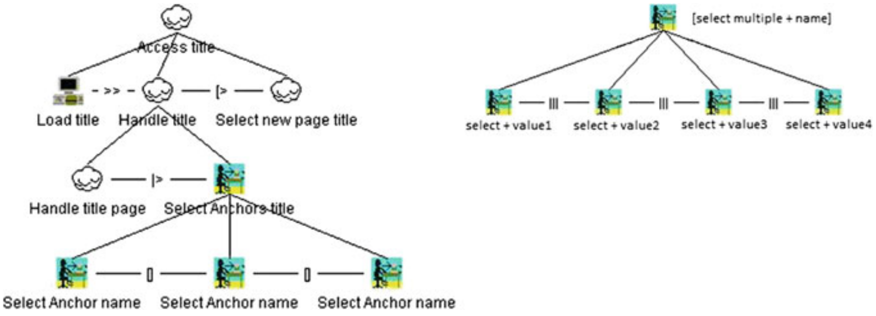
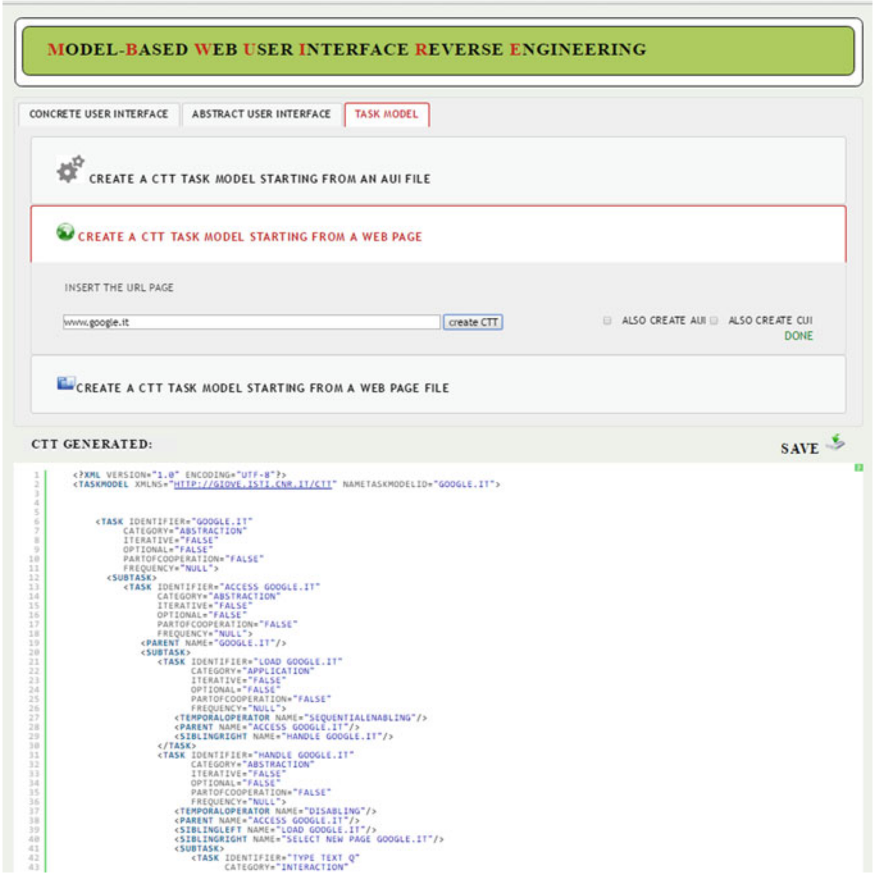


Fig. 18.15 From AUI to CTT: the resulting CTT model after translating elementary connections associated with HTML anchors (*left part*) and multiple choice elements (*right part*)

when they are anchors, this means that the navigation remains within the same page/presentation at hand; therefore, the "Handle title" node created in the first step will be expanded. This will be done by adding two new tasks (an abstract task and an interactive task) combined through a suspend/resume operator (see "Handle title page" |> "Select Anchors title" in Fig. 18.15, left part). Then, for each anchor found, an interactive task is created as child of the lastly created task. All such interactive tasks will be linked together through a choice operator (see Fig. 18.15, left part).

In the next step, the algorithm continues by analysing the type of the AUI interactors included in the AUI presentation and translating each of them into the corresponding CTT task models. For instance, if the AUI contains an element supporting a *multiple choice*, all the choice elements referring to the same multiple choice elements are translated into interactive tasks linked together through an interleaving operator (see Fig. 18.15, right part). In Fig. 18.16, a screenshot of the tool showing an example of translation of an AUI description into a CTT task model specification is shown.


MODEL-BASED WEB USER INTERFACE REVERSE ENGINEERING
CONCRETE USER INTERFACE
ABSTRACT USER INTERFACE
TASK MODEL
CREATE A CTT TASK MODEL STARTING FROM AN AUI FILE
CREATE A CTT TASK MODEL STARTING FROM A WEB PAGE
INSERT THE URL PAGE
create CTT
□ ALSO CREATE AUI □ ALSO CREATE CUI
DONE
CREATE A CTT TASK MODEL STARTING FROM A WEB PAGE FILE
CTT GENERATED:
SAVE

www.google.it

```
<?XML VERSION="1.0" ENCODING="UTF-8"?>
<TASKMODEL XMLNS="HTTP://GIOVE.ISTI.CNR.IT/CTT" NAMETASKMODELID="GOOGLE.IT">



    <TASK IDENTIFIER="GOOGLE.IT"
         CATEGORY="ABSTRACTION"
         ITERATIVE="FALSE"
         OPTIONAL="FALSE"
         PARTOFCOOPERATION="FALSE"
         FREQUENCY="NULL">
      <SUBTASK>
         <TASK IDENTIFIER="ACCESS GOOGLE.IT"
               CATEGORY="ABSTRACTION"
               ITERATIVE="FALSE"
               OPTIONAL="FALSE"
               PARTOFCOOPERATION="FALSE"
               FREQUENCY="NULL">
            <PARENT NAME="GOOGLE.IT"/>
            <SUBTASK>
               <TASK IDENTIFIER="LOAD GOOGLE.IT"
                     CATEGORY="APPLICATION"
                     ITERATIVE="FALSE"
                     OPTIONAL="FALSE"
                     PARTOFCOOPERATION="FALSE"
                     FREQUENCY="NULL">
                  <TEMPORALOPERATOR NAME="SEQUENTIALENABLING"/>
                  <PARENT NAME="ACCESS GOOGLE.IT"/>
                  <SIBLINGRIGHT NAME="HANDLE GOOGLE.IT"/>
               </TASK>
               <TASK IDENTIFIER="HANDLE GOOGLE.IT"
                     CATEGORY="ABSTRACTION"
                     ITERATIVE="FALSE"
                     OPTIONAL="FALSE"
                     PARTOFCOOPERATION="FALSE"
                     FREQUENCY="NULL">
                  <TEMPORALOPERATOR NAME="DISABLING"/>
                  <PARENT NAME="ACCESS GOOGLE.IT"/>
                  <SIBLINGLEFT NAME="LOAD GOOGLE.IT"/>
                  <SIBLINGRIGHT NAME="SELECT NEW PAGE GOOGLE.IT"/>
                  <SUBTASK>
                     <TASK IDENTIFIER="TYPE TEXT Q"
                           CATEGORY="INTERACTION"
```

Fig. 18.16 Tool for reverse engineering CTT task models

18.8 Conclusions and Future Work

We have presented an integrated set of model-based languages and associated tools for supporting design and development of interactive applications also using various modalities. They can be applied in various domains such as safety-critical systems (see for example at https://www.eurocontrol.int/ehp/?q=node/1617), ERP, workflow management systems and Ambient Assisted Living.

Some of such tools have been used by a broad community since they are publicly available (see http://giove.isti.cnr.it/tools.php), and over time, external use has provided suggestions for small improvements.

Future work will be dedicated to investigating how they can be exploited in combination with agile methodologies. This represents an interesting challenge since modelling requires some time and this may conflict with the fast pace adopted in such methodologies. However, the use of model-based tools able to support fast prototyping can be able to address this issue. Another important area is how to exploit task models in analysing user behaviours. Previous tools, such as Web-RemUsine (Paganelli and Paternò 2003a), have addressed how to compare client side Web logs representing actual behaviour with desired behaviour represented by the task model. It can be interesting to extend this approach to analyse broader human behaviour detected through various sensors and compare it with that described by task models in order to identify potential issues.

We also plan to continue to carry out studies in order to investigate improvements for the usability of the languages and the associated tools.

References

Anzalone D, Manca M, Paternò F, Santoro C (2015) Responsive task modelling. In: Proceedings of ACM SIGCHI symposium on engineering interactive computing systems, pp 126–131

Bernardi ML, Di Lucca GA, Distante D (2009a) The RE-UWA approach to recover user centered conceptual models from web applications. Int J Softw Tools Technol Transf 11(6):485–501

Berti S, Correani F, Paternò F, Santoro C (2004) The TERESA XML language for the description of interactive systems at multiple abstraction levels. In: Proceedings workshop on developing user interfaces with XML: advances on user interface description languages, pp 103–110

Caffiau S, Scapin D, Girard P, Baron M, Jambon F (2010) Increasing the expressive power of task analysis: systematic comparison and empirical assessment of tool-supported task models. Interact Comput 22(6):569–593 (2010)

Calvary G, Coutaz J, Thevenin D, Bouillon L, Florins M, Limbourg Q, Souchon N, Vanderdonckt J, Marucci L, Paternò F (2002) The CAMELEON reference framework. Deliverable D 1

Cockburn A, Karlson A, Bederson B (2008) A review of overview+detail, zooming, and focus +context interfaces. ACM Comput Surv 41(1):2

Coutaz J, Nigay L, Salber D, Blandford A, May J, Young R (1995) Four easy pieces for assessing the usability of multimodal interaction: the CARE properties. Proc Interact 1995:115–120

da Silva CBE (2010) Reverse engineering of rich internet applications. Master thesis, Minho University, 2010

Dixon M, Laput G, Fogarty J (2014) Pixel-based methods for widget state and style in a runtime implementation of sliding widgets. In: Proceedings of annual conference on human factors in computing systems, pp 2231–2240

Duarte C, Carriço L (2006) A conceptual framework for developing adaptive multimodal applications. In: Proceedings of IUI 2006, pp 132–139

Eggers J, Hülsmann A, Szwillus G (2013) Aufgabenmodellierung am Multi-Touch-Tisch. In: Boll S, Maaß S, Malaka R (eds) Mensch & Computer 2013: Interaktive Vielfalt, pp 325–328

Li P, Wohlstadter E (2008) View-based maintenance of graphical user interfaces. In: Proceedings of 7th international conference on aspect-oriented software development, p 156

Limbourg Q, Vanderdonckt J, Michotte B, Bouillon L, López-Jaquero V (2005) UsiXML: a language supporting multi-path development of user interfaces. In: Proceedings of engineering human computer interaction and interactive systems, pp 200–220

Maras J, Stula M, Carlson J (2011) Reusing web application user-interface. Lect Notes Comput Sci 6757:228–242

Martinie C, Palanque P, Winckler M (2011) Structuring and composition mechanisms to address scalability issues in task models. In: Proceedings of INTERACT, pp 589–609

Morgado IC, Paiva AC, Pascoal Faria J (2010) Dynamic reverse engineering of graphical user interfaces. Int J Adv Softw 5(3):224–236

Mori G, Paternò F, Santoro C (2002) CTTE: support for developing and analyzing task models for interactive system design. IEEE Trans Softw Eng 28(8):797–813

Mori G, Paternò F, Santoro C (2004) Design and development of multidevice user interfaces through multiple logical descriptions. IEEE Trans Softw Eng 30(8):507–520

Octavia JR, Vanacken L, Raymaekers C, Coninx K, Flerackers E (2010) Facilitating adaptation in virtual environments using a context-aware model-based design process. In: England D, Palanque P, Vanderdonckt J, Wild PJ (eds) Proceedings of TAMODIA 2009, pp 58–71

Paganelli L, Paternò F (2003a) Tools for remote usability evaluation of web applications through browser logs and task models. Behav Res Methods Instrum Comput Psychon Soc Publ 35 (3):369–378

Paganelli L, Paternò F (2003b) A tool for creating design models from web site code. Int J Softw Eng Knowl Eng World Sci Publ 13(2):169–189

Paternò F (1999) Model-based design and evaluation of interactive applications. Springer

Paternò F, Santoro C (2000) Integrating model checking and HCI tools to help designers verify user interfaces properties. In: Proceedings of DSV-IS’2000, pp 135–150

Paternò F, Santoro C, Scorcia A (2008) Automatically adapting web sites for mobile access through logical descriptions and dynamic analysis of interaction resources. Proc AVI 2008:260–267

Paternò F, Santoro C, Spano LDM (2009) A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. ACM Trans Comput Human Interact 16(4):19:1–19:30

Paternò F, Santoro C, Spano LD (2011) Engineering the authoring of usable service front ends. J Syst Softw 84:1806–1822

Raneburger D, Meixner G, Brambilla M (2013) Platform-independence in model-driven development of graphical user interfaces for multiple devices. In: Proceedings of ICSOFT, pp 180–195

Silva CE, Campos JC (2013) Combining static and dynamic analysis for the reverse engineering of web applications. In: Proceedings of 5th ACM SIGCHI symposium on engineering interactive computing systems, p 107

Sottet JS, Ganneau V, Calvary G, Demeure A, Favre JM, Demumieux R (2007) Model-driven adaptation for plastic user interfaces. In: Baranauskas C, Abascal J, Barbosa SDJ (eds) Proceedings of INTERACT 2007, pp 397–410

Spano LD, Fenu G (2014) IceTT: a responsive visualization for task models. In: Proceedings of the 2014 ACM SIGCHI symposium on engineering interactive computing systems. ACM, pp 197–200

Chapter 19
Formal Modelling of App-Ensembles

A Formal Method for Modelling Flexible Systems of User Interfaces Driven by Business Process Models

Johannes Pfeffer and Leon Urbas

Abstract This chapter shows how flexible systems of user interfaces with an underlying formal model can be created by learning from the success story of apps on mobile devices and leveraging the concept of business process modelling. It is shown how these multi-app user interfaces are modelled and how they can be transformed into Petri nets in order to apply existing formal analysis methods. The created user interfaces are called App-Ensembles and resemble interactive systems comprised of self-contained apps that are connected in a purposeful manner via navigation links and data channels. A formal language for modelling these App-Ensembles is introduced (AOF-L). The graphical modelling elements are taken exclusively from BPMN 2.0 (Business Process Model and Notation). The underlying notation is formalized using an OWL 2 ontology. It is shown how App-Ensembles can be easily integrated into existing classical business process models. Two use cases illustrate the utility of App-Ensembles and the practicality of the modelling approach. This chapter demonstrates that it is useful to combine a semi-formal graphical notation with a strictly formal mathematical model. Strengths of the approach include the ability to run reasoning algorithms on the model, to query the model using languages such as SPARQL, and to perform a formal verification regarding contradictions and BPMN compliance.

J. Pfeffer (✉) · L. Urbas
Technische Universität Dresden, Dresden, Germany
e-mail: johannes.pfeffer@tu-dresden.de

L. Urbas
e-mail: leon.urbas@tu-dresden.de

B. Weyers et al. (eds.), *The Handbook of Formal Methods in Human-Computer Interaction*, Human-Computer Interaction Series,
DOI 10.1007/978-3-319-51838-1_19

19.1 Introduction

Originally, the term *app* was just an abbreviation for the word application. Today, in times of ubiquitous mobile devices, the notion of *app* usually does not refer to a general software application, such as an operating system or a Web server. When speaking of apps, we usually mean a program that is installed on a mobile device by downloading it from an app-store. Such apps have become a huge success, in particular with the rise of the iOS and Android devices, such as smartphones and tablets. In order to understand App-Ensembles, it helps to bring to mind the drivers behind this success. Apps are specialized programs, mostly used on mobile devices that assist users in completing tasks that have a clearly defined scope. They stand in contrast to monolithic multi-purpose programs on desktop computers. Users often prefer apps over traditional software for a number of reasons: apps are easy to install and uninstall, easy to use due to lower complexity, mostly cheap or free, and serve as dedicated problem-solvers: "… there's an app for that!" In a more general sense, apps are specialized software tools. They do not exist on mobile devices only; some Web browsers, operating systems, automotive systems, etc., have also adopted the concept successfully.

Additionally, in many areas, professional workflows have become more dynamic. They change more regularly in structure, composition, and goals. Consequently, large multi-purpose applications have become increasingly tedious to keep up to date with changing requirements. Ultimately, monolithic tools may pose a burden of implementing lean and agile work processes.

This chapter presents the research towards establishing a formal model for App-Ensembles.

Definition App-Ensembles are interactive systems comprised of self-contained apps that are connected in a purposeful manner via navigation links and data channels.

App-Ensembles are especially useful in the following situations:

- when separate tools with separate UIs are used regularly to accomplish a single goal;
- when the order in which the tools are used varies frequently; and
- when business processes for the performed tasks are available (e.g. for maintenance or customer care).

In the field of HCI, formal methods are typically used to model a user interface at the level of UI elements. The language presented in this chapter models user interfaces at a higher level. The most granular element of the modelling language is a self-contained app. For modelling the UI of the app, another formal method may be used.

This chapter is organized as follows. In the following section, related work is presented. Afterwards, the background for the presented work is briefly described. Then, the formal modelling language for App-Ensembles is introduced and its graphical representation is explained. Next, the textual notation of the language using Semantic Web technologies is shown. Then, the graphical language is applied to two use case examples for maintenance tasks of wind turbines and a nuclear power plant. Afterwards, an approach for formal modelling of App-Ensembles is described. Next, it is shown that App-Ensemble models can be transformed into Petri nets and well-known algorithms can be used to identify modelling issues. The presented work is discussed, and the chapter closes with a conclusion and outlook.

19.2 Related Work

There are other modelling approaches that share some of AOF-L's goals. However, they differ in several key points. Most importantly, they are not focused on modelling highly encapsulated entities such as apps. While State Chart XML (Barnett et al. 2015), short SCXML, can be used to control high-level UI components such as dialogues (Lager 2013), it is mainly focused on Web-based UIs. The notation provides its own state-machine-based execution environment. In BPMN (Business Process Model and Notation), the execution environment is not part of the specification. Business processes are executed by external engines that do not necessarily behave like state machines (Fischer 2011). Alternative process modelling approaches such as the widely recognized YAWL (van der Aalst and ter Hofstede 2005) are in some respects more expressive than BPMN. For example, YAWL contains many additional workflow patterns. It has been used in combination with SCXML to create a user interface for workflow management systems. In Garcia et al. (2008), a to-do list and a workflow list are created and sequences of user dialogues are generated. The approach focuses on the generation of user interfaces with a set of basic GUI elements from task definitions, and there is no integration of external encapsulated tools such as apps.

The research field of SOA (e.g. Erl 2005), namely service composition, is also related to AOF-L. Pietschmann et al. (2009) present the idea of User Interface Services and consequently apply the ideas of service composition to arranging UIs. Daniel et al. (2010) go one step further and propose a model and language for service-based composition of UIs. However, the focus lies on UI distribution and developing mashup-like applications without the encapsulation found in mobile apps.

Sousa et al. (2008) present a model-driven approach that is used to link business processes with user interfaces. In their approach, BPMN elements, such as a split (a decision), are linked to user interface elements (e.g. a drop-down list). In AOF-L, the user interfaces are also linked to BPMN but at a much higher level. A complementary BPMN model for an App-Ensemble is introduced in order to associate an app-based user interface with a BPMN task or sequence of BPMN tasks.

19.3 Background

Pfeffer et al. (2013) describe a concept to dynamically create mobile interactive systems for complex tasks by orchestrating self-contained apps with defined interfaces. The approach addresses the challenge to generate flexible, adaptable, and easy-to-use app-based user interfaces with minimal effort. It takes into account that data sources, workflows, users, and the apps themselves may be subject to continuous change. The approach heavily relies on the Semantic Web stack (Bizer et al. 2009) and its adaptation for industrial environments (Graube et al. 2011). For easy integration into existing business processes and virtual manufacturing networks, it is based on established concepts from the field of business process modelling.

One method to create useful stand-alone App-Ensembles is the *App-Orchestration* process (Ziegler et al. 2012) that consists of three steps: *select*, *adapt*, and *manage* (as shown in Fig. 19.1). The process relies on an existing set of semantically described mobile apps, a well-defined information space, i.e. data sources, and a business process model that defines necessary tasks and their relations. In the *select* step, a subset of apps is selected from an app pool that are best to support the tasks given in the actual business process. To increase reusability, in addition to commercial off-the-shelf apps, the app pool may contain generic apps tailored for orchestration that are not yet adapted to a specific use case or information source. Instead, they are adaptable to various specific tasks. This is done in the *adapt* step. Adaptation may include the visual appearance, the interaction style, the actual data source, and others. In the *manage* step, the adapted apps are connected according to a navigation design model which is derived from the business process model. Therefore, only useful navigation paths are presented to the user.

All three steps are performed at design time. As a result, a deployable *App-Ensemble* consisting of a model and installable app artefacts is produced. The App-Ensemble is then deployed to a device (e.g. an Android tablet). At runtime, a workflow execution engine enforces the navigation design (switching from app to app, allowing the user to make decisions at workflow branches) and facilitates the information exchange between the apps. The same set of apps can be orchestrated in many ways, depending on the needs of the underlying business process.

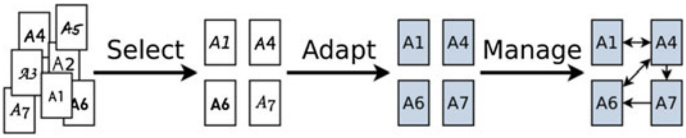


Fig. 19.1 App-Orchestration steps—select, adapt, manage

In order to realize this concept, formal models and tools are required. The set of tools and specifications for App-Orchestration is collected in the Application Orchestration Framework (AOF). The modelling language to describe apps, App-Ensembles and Orchestration, is called AOF-Language (AOF-L).

19.4 The AOF-Language

The AOF-Language consists of two parts that describe the following: (i) the capabilities of a single app and (ii) the setup and structure of an App-Ensemble. The first part defines language elements that allow semantic description of properties of an app. This includes name, textual description, version, and others. Also, the interface of the app is described (entry points, exit points, inputs, outputs), and other information needed to start the app, provide data to it, and receive results. This part of the language is not a major subject of this chapter. The current state of the specification can be found in Pfeffer (2015).

The second part which is the main focus of this chapter allows the formal description of an App-Ensemble. AOF-L uses BPMN 2.0 (Business Process Model and Notation, OMG 2011) as a basis to describe App-Ensembles. It has been chosen because the notation is easy to learn, has execution semantics, is widely used in business and industry, and is extensible with custom modelling elements. BPMN is standardized in the ISO/IEC standard 19510:2013. This research is only concerned with version 2.0 of BPMN. Therefore, from here on, the version number is omitted for brevity.

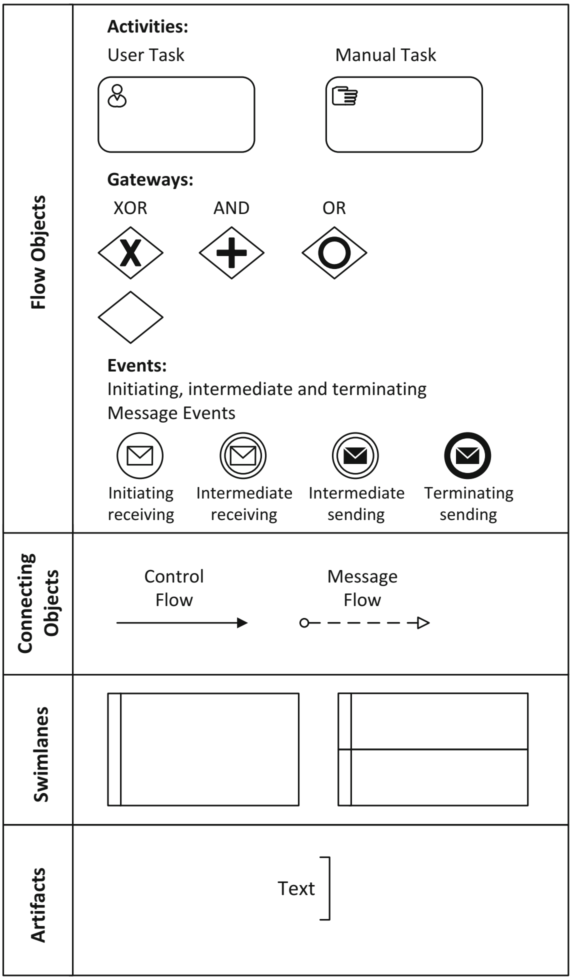
The language specification also includes a formal semantic vocabulary (AOF-Vocabulary) specified using RDFS, the Resource Description Framework Schema (Klyne and Carroll 2004), and concepts from the Web Ontology Language, specifically OWL 2 (W3C OWL Working Group 2012).

The combination of a well-known business process modelling language and Semantic Web technologies makes it possible to easily publish BPM and App-Ensembles in form of Linked Data (Bizer et al. 2009; Graube et al. 2013). This enables integration in collaborative Virtual Enterprises as described in (Münch et al. 2014).

For execution, the model is serialized as RDF (Resource Description Framework) using concepts defined in the AOF-Vocabulary, from a BPMN Ontology[1] (Rospocher et al. 2014), Friend of a friend (FOAF) (Brickley and Miller 2014), Dublin Core (DC) (ISO-15836 2009), RDF, and RDFS (Klyne and Carroll 2004). There is a graphical representation which is exclusively based on modelling elements of BPMN. This chapter is mainly concerned with the graphical notation because it is well suited to illustrate an App-Ensemble model for a use case.

[1] Accessible at https://dkm.fbk.eu/bpmn-ontology.

Fig. 19.2 Subset of BPMN modelling elements used in AOF-L



19.4.1 *Graphical Modelling Elements*

AOF-L uses a subset of BPMN modelling elements as shown in Fig. 19.2. BPMN modelling elements other than those shown are not allowed. All BPMN connection rules that apply to the allowed modelling elements are valid (for an overview see OMG 2011, p. 40ff). None of the semantics of BPMN are contradicted. But for simplification and analysis, additional restrictions apply that are described in this section at the individual element explications.

19.4.1.1 Activities

By definition, a *user task* is executed by a human being through a user interface. In AOF-L, this task type is used to represent an app in an App-Ensemble. The activity-type user task is extended in compliance with the BPMN specification (OMG 2011, p. 55) with a reference to a semantic app-description. The activity is completed when the app is closed by the user or an interaction with the app is performed that yields output values.

A *manual task* is executed by a human without the help of an app or other type of user interface. A typical example in the use case domain would be to open or close a valve. Using a manual task in an App-Ensemble tasks can be modelled that users have to perform without assistance of an app. They are treated as tasks and have to be confirmed as completed to the execution engine before the control flow continues past the activity.

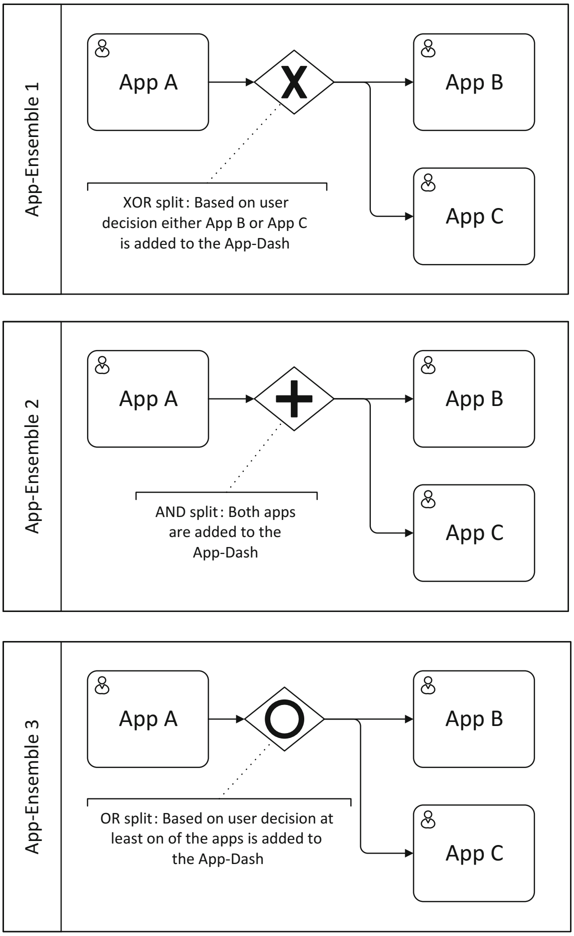
19.4.1.2 Gateways

Gateways are used to split or join a workflow. The flow of control is affected in the same manner as in standard BPMN. However, due to the fact that user tasks represent apps that run on a physical device and can be used by a user only one at a time, no real parallelism is possible. Thus, when two or more branches with user tasks are activated at the same time as a result of a split, they are placed in an *App-Dash*. The App-Dash is a list of available apps—comparable to a task list in classical business process execution engines. Where appropriate, the user is asked by the process execution engine to choose between possible next apps to continue the workflow.

The behaviour of the gateways in an App-Ensemble is described in the following paragraphs. This is a specialization of the behaviour specified by BPMN. In cases where a user decision is needed (XOR-Split, OR-Split), a dialogue is presented and the user must select the preferred action, e.g. decide between several branches.

An *XOR-Split* splits an App-Ensemble workflow into multiple branches (see Fig. 19.3). Exactly one of the branches is activated based on user decision. The other branches are not activated. An *XOR-Join* joins multiple workflow branches. The workflow continues as soon as one of the incoming branches is activated. In an App-Ensemble, this occurs when at least one app (user task) directly preceding the gateway finishes.

Fig. 19.3 Branching XOR, AND, and OR gateways


App-Ensemble 1
App A
App B
App C
XOR split: Based on user decision either App B or App C is added to the App-Dash
App-Ensemble 2
App A
App B
App C
AND split: Both apps are added to the App-Dash
App-Ensemble 3
App A
App B
App C
OR split: Based on user decision at least on of the apps is added to the App-Dash

The *AND-Split* splits an App-Ensemble workflow into multiple branches that are all activated at the same time. This results in multiple new apps placed in the users App-Dash (see Fig. 19.3). An *AND-Join* allows the workflow to continue when all of the incoming branches are activated, i.e. when all preceding apps have finished.

For an *OR-Split,* the App-Ensemble workflow is split based on user decision. The user may decide to activate one or more branches resulting in one or more apps being placed in the App-Dash (see Fig. 19.3). At an *OR-Join,* the workflow is continued based on user decision when at least one incoming branch has been activated.

19.4.1.3 Events

Data can be received from a superordinate process using *Receiving Message Events* (intermediate or initiating) represented by a white envelope. Data can be sent to a superordinate process using *Sending Message Events* (intermediate or terminating) represented by a black envelope. The use of these events in App-Ensembles is exemplified below in Example 2.

19.4.1.4 Connecting Objects

The *Control Flow* arrow is used just as in BPMN. *Message Flow* arrows are used to connect an App-Ensemble with superordinate business processes. They are used to synchronize the business process with the App-Ensemble.

19.4.1.5 Swimlanes

Optionally, App-Ensembles may be placed in their own *Pool*. They must be placed in their own exclusive Pool whenever the BPM describes a collaboration (more than one participant). *Lanes* may be used to organize the modelling elements in the Pool but have, as in BPMN, no relevance to the control flow.

Start and End Events are optional according to the BPMN specification (see OMG 2011, p. 237 and p. 246). In AOF-L, they are not allowed, except for initiating and terminating message events.

19.4.2 Textual Notation of the Model

The notation of an App-Ensemble model is facilitated using RDF. Since the focus of this chapter is the graphical notation for a specific use case, this shall be illustrated only briefly by the example of a single user task representing an app. Using the BPMN ontology (Rospocher et al. 2014) and the AOF-Vocabulary, an App-Ensemble can be described in Turtle RDF notation (Beckett and Berners-Lee 2008) as follows. The excerpt describes the right column of *Example 1* (see section Examples) with three user tasks and one XOR-split gateway.

```
@prefix aof: <http://eatld.et.tu-dresden.de/aof/>.
@prefix app: <http://example/app-pool/>.
@prefix bpmn2: <http://dkm.fbk.eu/index.php/BPMN2_Ontology#>.
@prefix ae: <http://example/wind-turbine-maintenance/1/>.

# Tasks, Gateways
ae:userTask_3 a aof:userTask;
  bpmn2:id "userTask_3";
  bpmn2:name "Visual inspection check list app";
  aof:assignedApp app:VisChecker.

ae:userTask_4 a aof:userTask;
  bpmn2:id "userTask_4";
  bpmn2:name "Camera app";
  aof:assignedApp app:IntegratedCameraApp.

ae:userTask_5 a aof:userTask;
  bpmn2:id "userTask_5";
  bpmn2:name "Diagnosis app";
  aof:assignedApp app:WTDiagnosisApp.

ae:ExclusiveGateway_1 a bpmn2:exclusiveGateway;
  bpmn2:id "ExclusiveGateway_1";
  bpmn2:incoming ae:SequenceFlow_4;
  bpmn2:outgoing ae:SequenceFlow_5;
  bpmn2:outgoing ae:SequenceFlow_6;

# Sequence Flows
ae:SequenceFlow_4 a bpmn2:sequenceFlow ;
  bpmn2:id "SequenceFlow_1" ;
  bpmn2:sourceRef ae:UserTask_3;
  bpmn2:targetRef ae:ExclusiveGateway_1.

ae:SequenceFlow_5 a bpmn2:sequenceFlow ;
  bpmn2:id "SequenceFlow_1" ;
  bpmn2:sourceRef ae: ExclusiveGateway_1;
  bpmn2:targetRef ae: UserTask_4.

ae:SequenceFlow_6 a bpmn2:sequenceFlow ;
  bpmn2:id "SequenceFlow_1" ;
  bpmn2:sourceRef ae: ExclusiveGateway_1;
  bpmn2:targetRef ae: UserTask_5.

...
```

Some information that a full App-Ensemble would in include was omitted in the listing above because it is not relevant to the example. Especially metadata about the creator, App-Descriptions for the apps that define data channels and state how the app can be started are not shown. The task `ae:userTask_4` is an instance of `aof:userTask` (in Turtle “a” is an abbreviation for `rdf:type` which describes an instance of relation). The RDF resource `aof:UserTask` is a class that is derived from the original BPMN user task class. The lines with `aof:assignedApp` relate to a resource describing the app semantically. This semantic description defines the interface of the app by stating entry points with input variables of a certain type and exit points with output variables (for an example of a semantic app-description refer to the AOF-Language specification (Pfeffer 2015).

All other modelling elements, such as gateways and control flow arrows, are written in the same fashion. Naturally, any other RDF serialization such as N3 or RDF-XML can be used in place of Turtle.

19.5 Use Case Examples

The use case examples both focus on asset maintenance. In the field of maintenance, it is often necessary to work with tools from various vendors, the order of usage varies with the necessary tasks, and new or changed work processes are frequently introduced.

19.5.1 Example 1: Wind Turbine Maintenance

The first example is set in the case study *Interactive Systems in Rural and Urban Areas—Maintenance of wind turbines*. It is intentionally straightforward and serves to illustrate the general principle of App-Ensembles (see Fig. 19.4). The App-Ensemble was created along the lines of the select–adapt–manage methodology mentioned above (see section *Background*) but without a previously defined business process. To create a model for the maintenance workflow of a wind energy technician, first, the necessary apps to accomplish the tasks are selected from a

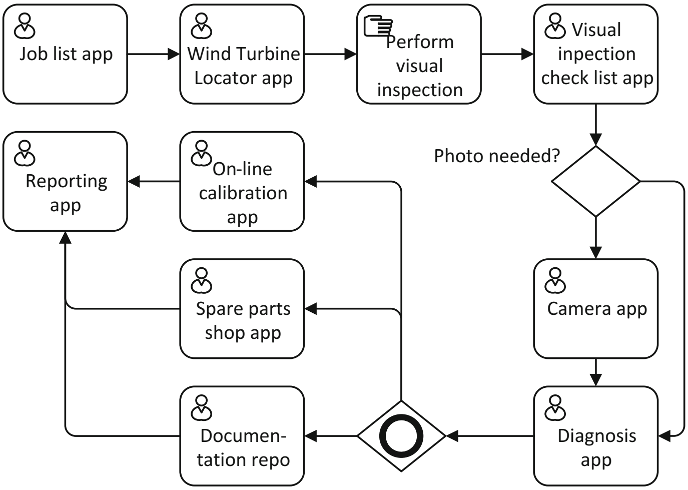


Fig. 19.4 Wind turbine maintenance use case (Example 1)

repository, e.g. an app-store. In the adaptation step, the data sources for wind turbine locations and sensor data are added to the app-descriptions. This makes it possible to always show contextual information regarding the current point of interest. Next, the model is jointly constructed by a modelling and a domain expert (in this case the technician) using a graphical modelling tool for App-Ensembles. In the manage step, a navigation design is derived from the model which is packaged together with all required apps as an App-Ensemble that can be directly deployed to a mobile device.

At the beginning of his or her maintenance work, the technician starts the App-Ensemble. First, a job list is presented from which the technician may select the maintenance job he or she would like to work on first (j*ob list app*). Then, the location of the wind turbine is shown on a map and optional turn-by-turn navigation is offered (*wind turbine locator app*). When the technician has arrived at the correct location, he or she is asked to perform a visual inspection. This inspection is currently done without assistance by an app; thus, it is represented by a manual task. Next, a checklist for the visual inspection is completed (*visual inspection check list app*). Afterwards, the technician may decide to take pictures of the observations made (*camera app*) or may advance directly to a diagnosis tool (*diagnosis app*). This is realized by an AND-split gateway and is presented to the technician as a decision over which app to put on the App-Stack. Depending on the result of the diagnosis, several further tools may or may not be used at the technician's discretion (*documentation repository app, spare parts shop app, online calibration app*). The used OR-split gateway is presented to the user as a list of apps from which he or she may select any number to be added to the App-Stack. Finally, a report is created and sent (*report app*). All join gateways in the model are implicit OR-join gateways.

In this scenario, the technician executes the App-Ensemble when needed and repeats the workflow until the workday is over or all assigned jobs have been completed. The following Example 2 adds to this by illustrating how App-Ensembles can be controlled by superordinate business processes.

19.5.2 Example 2: Nuclear Power Plant Maintenance

The second use case presented here is loosely based on the case study *Control of a Nuclear Power Plant.* It serves to show how existing business process models can be used as a higher level control process for App-Ensembles. It also illustrates some of the problems that may arise and presents a way how to mitigate these issues using the formal model of the App-Ensemble.

A nuclear power plant is a special type of process plant with very high security requirements. Like any other process plant, it consists of machines, devices, physical connections (pipes, wires, etc.), sensors, and other equipment. In the example, business process models have been created for all maintenance tasks in

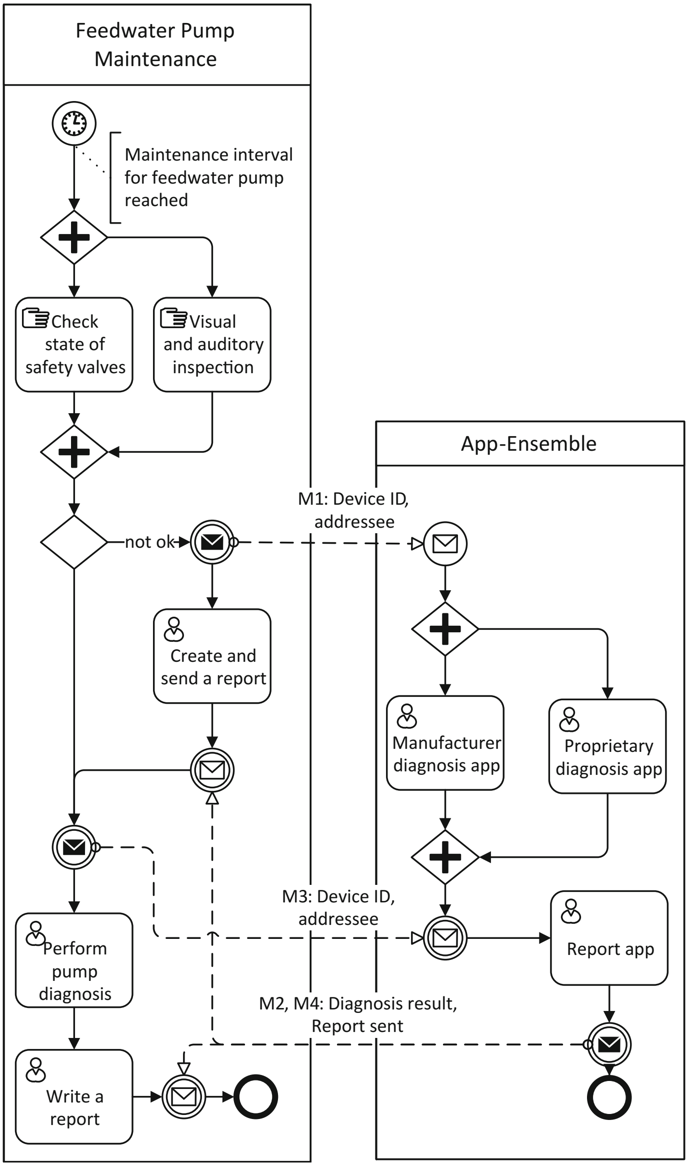

Feedwater Pump Maintenance
Maintenance interval for feedwater pump reached
Check state of safety valves
Visual and auditory inspection
not ok
M1: Device ID, addressee
App-Ensemble
Create and send a report
Manufacturer diagnosis app
Proprietary diagnosis app
M3: Device ID, addressee
Perform pump diagnosis
Report app
M2, M4: Diagnosis result, Report sent
Write a report

Fig. 19.5 Nuclear power plant maintenance use case. Workflow and corresponding App-Ensemble. The model contains an intentional deadlock for demonstration purposes (Example 2)

the power plant in order to improve controllability of task execution and reporting of deviations. The use case focuses on a fraction of a business process model (BPM) that is concerned with maintenance tasks for the feedwater pumps of the nuclear power plant (see Fig. 19.5, left-hand side).

Existing BPM can be adapted for execution with App-Ensembles by adding message events before and after tasks that are completed with the help of apps. These message events serve as synchronization points between the original BPM and the App-Ensemble. In the use case, the BPM is triggered when the maintenance interval for the feedwater pump is reached. The presented model has been simplified for reasons of brevity, but the basic workflow is realistic.

At the beginning, the maintenance personnel checks the state of the safety valves and performs a visual inspection of the pump (e.g. for leakage). The latter two tasks are manual and are not supported by any tool controlled by the process. If something notable is observed, a report on the findings is created. Finally, the pump diagnosis is performed and another report is written. In a BPM that is not supported by an App-Ensemble, the process looks just the same, with the exception of the additional send and receive events. Thus, existing BPM can be easily extended for use with App-Ensembles by adding a participant (in BPMN terminology, this refers to separate process) containing the App-Ensemble model to the collaboration.

In Fig. 19.5, right-hand side, a possible corresponding App-Ensemble is shown. The maintenance personnel uses two apps for diagnosing the pump. The control flow starts with a token being generated either on the receiving start message event or on the receiving intermediate message event. In the first case, the control flow splits into two parallel branches and the token is cloned. Now, the user must serialize this parallelism by choosing the order in which the apps shall be presented by selecting an app from the App-Dash. Only when both apps have been used (i.e. the activity has been completed) and expected data has been yielded, the control flow continues and a report can be created. There is a second entry point to the App-Ensemble that is triggered when a report has to be written after preliminary inspection of the pump. This shows that parts of App-Ensembles can be flexibly reused and may be instantiated multiple times.

After the result data set (which may consist of the diagnosis result and/or the report) is returned to the superordinate process, the token is consumed.

However, especially in complex business processes, problems with creating correct BPMN models and App-Ensembles may arise. In the next sections, it will be shown how the formal model of the process and its Petri net representation can be used to identify the modelling problems as deadlocks and other model characteristics.

19.6 Formal Modelling

On the one hand, the AOF-Language is RDF-based and represents an RDF graph. On the other hand, it makes heavy use of concepts from BPMN. Thus, it is self-evident to evaluate formalization via methodologies that have been used to formalize the business process models. In the following section, this is investigated in order to discuss the approach during the workshop.

Since App-Ensembles can be expressed via a subset of BPMN modelling elements, it is possible to formally model an App-Ensemble process along the lines described in Dijkman et al. (2007):

$$An\,App - Ensemble\,process\,is\,a\,tuple\,\mathcal{P} = \\ \left(\mathcal{O}, \mathcal{A}, \mathcal{E}, \mathcal{G},\ \left\{t^U\right\}, \left\{t^M\right\}, \left\{e^S\right\}, \left\{e^{I_R}\right\}, \left\{e^{I_S}\right\}, \left\{e^E\right\}, \mathcal{G}^S, \mathcal{G}^J, G^{O_S}, \\ \mathcal{G}^{O_J}, \mathcal{G}^X, \mathcal{G}^M, \mathcal{F}\right)$$

where

$\mathcal{O}$ is a set of objects which can be partitioned into disjoint sets of activities $\mathcal{A}$, events $\mathcal{E}$, and gateways $\mathcal{G}$;
$\mathcal{A}$ can be partitioned into disjoint sets of user tasks $\left\{t^U\right\}$ and manual tasks $\left\{t^M\right\}$;
$\mathcal{E}$ can be partitioned into disjoint sets of start message events $\left\{e^S\right\}$, receiving intermediate message events $\left\{e^{I_R}\right\}$, sending intermediate message events $\left\{e^{I_S}\right\}$, and end message events $\left\{e^E\right\}$; $\mathcal{G}$ can be partitioned into disjoint sets of splitting AND gateways $\mathcal{G}^S$, joining AND gateways $\mathcal{G}^J$, splitting OR gateways G^{O_S}, joining OR gateways $\mathcal{G}^{O_J}$, splitting XOR gateways $\mathcal{G}^X$, and joining XOR gateways $\mathcal{G}^M$;
$\mathcal{F} \subseteq \mathcal{O} \times \mathcal{O}$ is the control flow relation, i.e. a set of sequence flows connecting objects.

Within this definition, an App-Ensemble is a directed graph with nodes $\mathcal{O}$ and arcs $\mathcal{F}$. For any node $x \in \mathcal{O}$, input nodes of x are given by $in(x) = \{ \in \mathcal{O} \mid y\mathcal{F}x\}$ and output nodes of x are given by $out(x) = \{ \in \mathcal{O} \mid x\mathcal{F}y\}$.

19.7 Petri Net Representation

Based on this formal syntax, an App-Ensemble Model can be formulated and requirements for it being well-formed can be defined. As shown in Dijkman et al. (2007), the model can now be mapped to Petri nets.

Figure 19.6 shows the first part of the feedwater pump maintenance BPM. It models the BPM until the first AND-join gateway (G^{J1}). The illustration has been split for better understanding and in order to fit the format of the book. Figure 19.7 shows the remaining Petri net. All elements in the Petri net have been named according to the formal notation defined in the previous section.

Since the resulting Petri net for the example is not very complex, a practiced Petri net user will see immediately that it contains a deadlock. Similarly, an experienced BPMN modeller will recognize this issue in the App-Ensemble. The intentionally inserted deadlock is located in transition $t(e^{IR3})$ (see Fig. 19.7) which waits for firing of place $P(e^{IS2}, e^{IR3})$ that in turn can never occur due to the feedback in transition $t(e^{IR1})$. When the deadlock has been identified in the Petri net using

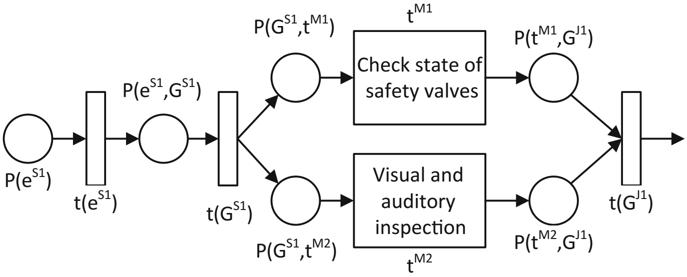


Fig. 19.6 Petri net representation of use case 2 (part 1)

well-known algorithms, it can be circumvented in the BPM and consequently in the Petri net, e.g. by introducing a distinction of cases around $t(e^{IR3})$.

While some types of analysis could also be performed on the original App-Ensemble using existing BPMN tools, the available algorithms for reachability, liveness, and boundedness of Petri nets provide a rich additional toolbox for finding problems in large, hierarchical, and complex models.

19.8 Discussion

It should be noted that due to ambiguities in the BPMN specification in certain cases, the mapping from BPMN to Petri net may be problematic. For instance, when multiple start events occur, it is not clearly defined whether a new process instance or a new token should be created in the currently running instance. However, aside from these ambiguities, transformation to Petri nets opens a rich and well-established set of tool for analysis and verification of App-Ensembles. While Petri nets could be used for modelling App-Ensembles in the first place, the presented language is much more concise and allows for easy visual understanding of business processes (compare Fig. 19.5 to Figs. 19.6 and 19.7).

The App-Ensemble model relies on the formal BPMN ontology (Rospocher et al. 2014). Consequently, it inherits many of the strengths as well as limitations of BPMN. Strengths include the ability to run reasoning algorithms on the model (e.g. for consistency checking or inferring implicit types), the ability to query the model using languages such as SPARQL, and the ability to perform a formal verification regarding contradictions and BPMN compliance. Due to OWL 2 limitations, some conditions and default values described in the BPMN specification could not be encoded in the ontology (see Rospocher et al. 2014, p. 5f).

App-Ensembles are interactive systems of user interfaces. The granularity of the App-Ensemble model ends at the level of the app—a self-contained user interface that is tailored for tending to a well-contained need. If the notion of *app* can be seen as a specialization of the term *user interface* as defined in Weyers (2012), instances

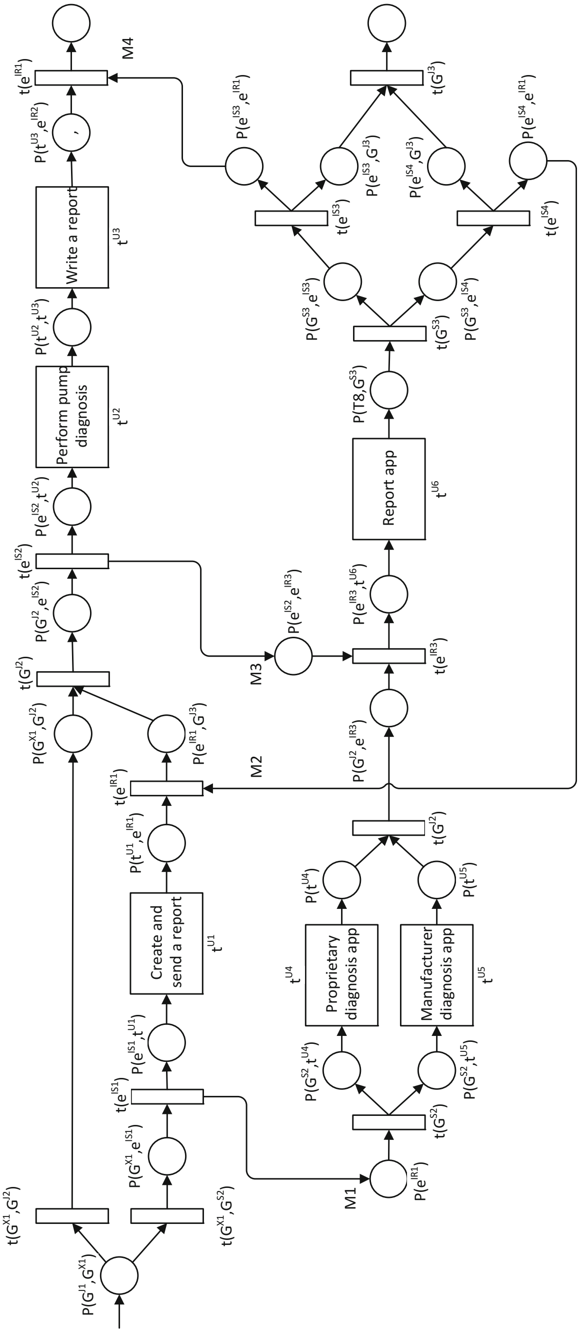


Fig. 19.7 Petri net representation of use case 2 (part 2)

of user interfaces in App-Ensembles can be modelled using the FILL and VFILL language. While AOF-L is restricted to modelling input and output of an app that is provided by a business process, FILL can model input and output that is generated or consumed by the user (e.g. changing diagnosis parameters and viewing the results). By linking a FILL model of an app user interface to a semantic app-description, a model of the whole interactive system can be created. If additionally, the cognitive model of the user interface is included (Urbas and Leuchter 2005), the model becomes even more holistic and useful, e.g. for controlling or for usability or performance evaluation.

19.9 Conclusion and Outlook

This research has presented advancements towards modelling language for App-Ensembles. The graphical representation of this language is based on BPMN. Its application has been shown in use cases to substantiate the ability of the language to model interactive systems of multi-app user interfaces. At the time of writing, the ontological formalization of the AOF-L was still in progress.

A mathematical formalization of the App-Ensembles has been presented, and it has been shown how the models created with AOF-L can be transformed into Petri nets in order to subject them to readily available formal analysis.

Since AOF-L is an RDF-based language, RDF-based approaches to formalization should also be considered. In Hayes and Gutierrez (2004), bipartite graphs are used as an intermediate model for RDF. The authors (Morales and Serodio 2007) model RDF as a directed hypergraph which is a very concise and expressive model for RDF graphs. These approaches will be investigated in coming research.

Acknowledgements The research leading to the presented work was partially funded by the German Ministry of Education and Research (BMBF Grant No. 01IS14006A) and by the European Community's Seventh Framework Programme under grant agreement no. FP7-284928 ComVantage.

References

Barnett J, Akolkar R, Auburn RJ et al (2015) State Chart XML (SCXML): state machine notation for control abstraction

Beckett D, Berners-Lee T (2008) Turtle-terse RDF triple language

Bizer C, Heath T, Berners-Lee T (2009) Linked data-the story so far. Int J Semant Web Inf Syst IJSWIS 5:1–22

Brickley D, Miller L (2014) FOAF vocabulary specification. http://xmlns.com/foaf/spec/

Daniel F, Soi S, Tranquillini S et al (2010) From people to services to UI: distributed orchestration of user interfaces. In: Hull R, Mendling J, Tai S (eds) Business process management. Springer, Berlin, Heidelberg, pp 310–326

Dijkman RM, Dumas M, Ouyang C (2007) Formal semantics and analysis of BPMN process models using Petri nets

Erl T (2005) Service-oriented architecture: concepts, technology, and design. Prentice Hall PTR, Upper Saddle River, NJ, USA

Fischer L (2011) BPMN 2.0 handbook, 2nd edn. Future Strategies Inc., Lighthouse Point, FL

Garcia JG, Vanderdonckt J, Calleros JMG (2008) FlowiXML: a step towards designing workflow management systems. Int J Web Eng Technol 4:163–182

Graube M, Pfeffer J, Ziegler J, Urbas L (2011) Linked data as integrating technology for industrial data. In: Proceedings 14th international conference network-based information systems (NBiS), pp 162–167

Graube M, Ziegler J, Urbas L, Hladik J (2013) Linked data as enabler for mobile applications for complex tasks in industrial settings. In: Proceedings 18th IEEE conference emerging technologies factory automation (ETFA), pp 1–8

Hayes J, Gutierrez C (2004) Bipartite graphs as intermediate model for RDF

ISO-15836 (2009) The Dublin Core metadata element set. https://www.iso.org/obp/ui/#iso:std:iso:15836:ed-2:v1:en

Klyne G, Carroll JJ (2004) Resource description framework (RDF): concepts and abstract syntax. http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/

Lager T (2013) Statecharts and SCXML for dialogue management. In: Habernal I, Matoušek V (eds) Text, speech, and dialogue. Springer, Berlin, Heidelberg, p 35

Morales AAM, Serodio MEV (2007) A directed hypergraph model for RDF. In: Proceedings of knowledge web PhD symposium

Münch T, Hladik J, Salmen A et al (2014) Collaboration and interoperability within a virtual enterprise applied in a mobile maintenance scenario. Revolut Enterp Interoper Sci Found 137–165

OMG (2011) Business process modelling notation, v.2.0—specification. http://www.omg.org/spec/BPMN/2.0/

Pfeffer J (2015) AOF language specification version 002. http://eatld.et.tu-dresden.de/aof/spec/

Pfeffer J, Graube M, Ziegler J, Urbas L (2013) Networking apps for complex industrial tasks—orchestrating apps efficiently. atp edition 55(3):34–41

Pietschmann S, Voigt M, Rümpel A, Meißner K (2009) CRUISe: composition of rich user interface services. In: Gaedke M, Grossniklaus M, Díaz O (eds) Web engineering. Springer, Berlin, Heidelberg, pp 473–476

Rospocher M, Ghidini C, Serafini L (2014) An ontology for the business process modelling notation. In: Garbacz P, Kutz O (eds) Formal ontology in information systems—proceedings of the eighth international conference, FOIS2014, 22–25 Sept 2014. IOS Press, Rio de Janeiro, Brazil, pp 133–146

Sousa K, Mendonça H, Vanderdonckt J (2008) A model-driven approach to align business processes with user interfaces. J Univers Comput Sci 14:3236–3249

Urbas L, Leuchter S (2005) Model based analysis and design of human-machine dialogues through displays. KI 19:45–51

van der Aalst WMP, ter Hofstede AHM (2005) YAWL: yet another workflow language. Inf Syst 30:245–275

W3C OWL Working Group (2012) OWL 2 Web ontology language document overview, 2nd edn. http://www.w3.org/TR/owl2-overview/

Weyers B (2012) Reconfiguration of user interface models for monitoring and control of human-computer systems. Dr. Hut Verlag, Berlin

Ziegler J, Pfeffer J, Graube M, Urbas L (2012) Beyond app-chaining: mobile app orchestration for efficient model driven software generation. In: Proceedings of the 17th international IEEE conference on emerging technologies and factory automation. Krakau, Poland

Chapter 20
Dealing with Faults During Operations: Beyond Classical Use of Formal Methods

Camille Fayollas, Philippe Palanque, Jean-Charles Fabre, Célia Martinie and Yannick Déléris

Abstract Formal methods provide support for validation and verification of interactive systems by means of complete and unambiguous description of the envisioned system. Used in the early stages of the development process, they allow detecting and correcting development faults **at design and development time**. However, events that are beyond the envelope of the formal description may occur and trigger unexpected behaviours in the system **at execution time** (inconsistent from the formally specified system) resulting in failures. Sources of such interactive system failures can be permanent or transient hardware failures, due to, e.g., natural faults triggered by alpha particles from radioactive contaminants in the chips or neutrons from cosmic radiation. This chapter first presents a systematic identification of the faults that can be introduced in the system during both development and operations time and how formal methods can be used in such context. To exemplify such usage of formal methods, we present their use to describe software architecture and self-checking components to address these faults in the context of interactive systems. As those faults are more likely to occur in the high atmosphere, the proposed solutions are illustrated using an interactive application within the field of interactive cockpits. This application allows us to demonstrate the use of the

C. Fayollas (✉) · P. Palanque · C. Martinie
ICS-IRIT, University of Toulouse, 118 Route de Narbonne, 31062 Toulouse, France
e-mail: Camille.Fayollas@irit.fr

P. Palanque
e-mail: Philippe.Palanque@irit.fr

C. Martinie
e-mail: Celia.Martinie@irit.fr

C. Fayollas · J.-C. Fabre
LAAS-CNRS, 7 avenue du colonel Roche, 31077 Toulouse, France
e-mail: Jean-Charles.Fabre@laas.fr

Y. Déléris
AIRBUS Operations, 316 Route de Bayonne, 31060 Toulouse, France
e-mail: yannick.deleris@airbus.com

B. Weyers et al. (eds.), *The Handbook of Formal Methods in Human-Computer Interaction*, Human-Computer Interaction Series,
DOI 10.1007/978-3-319-51838-1_20

concepts and their application for WIMP interactive systems (such as the ones of the nuclear case study of the book).

20.1 Introduction

A safety-critical system is a system in which failures or errors potentially lead to loss of life or injuries of human beings (Bowen and Stavridou 1993) while a system is considered as critical when the cost of a potential failure is much higher than the development cost. Whether or not they are classified as safety critical or "only" critical, interactive systems have made their way into most of the command and control workstations including satellite ground segments, military and civil cockpits, air traffic control. Furthermore, the complexity and quantity of data manipulated, the amount of systems to be controlled and the high number of commands to be triggered in a short period of time have pulled sophisticated interaction techniques into most of them.

The overall dependability of an interactive system is one of its weakest components, and there are many components in such systems ranging from the operator processing information and physically exploiting the hardware (input and output devices), interaction techniques, to the interactive application and possibly the underlying non-interactive system being controlled.

Building reliable interactive systems is a cumbersome task due to their very specific nature. The behaviour of these reactive systems is event-driven. As these events are triggered by human operators manipulating hardware input devices, these systems have to react to unexpected events. On the output side, information (such as the current state of the system) has to be presented to the operator in such a way that it can be perceived and interpreted correctly. Lastly, interactive systems require addressing together hardware and software aspects (e.g. input and output devices together with their device drivers).

In the domain of fault-tolerant systems, empirical studies have demonstrated (e.g. Nicolescu et al. 2003) that software crashes may occur even though the development of the system has been extremely rigorous. One of the many sources of such crashes is called natural faults (Avižienis et al. 2004) triggered by alpha particles from radioactive contaminants in the chips or neutron from cosmic radiation. A higher probability of occurrence of faults (Schroeder et al. 2009) concerns the systems deployed in the high atmosphere (e.g. aircraft) or in space (e.g. manned spacecraft (Hecht and Fiorentino 1987)). Such natural faults demonstrate the need to go beyond classical fault avoidance at development time (usually brought by formal description techniques and properties verification) and to identify all the threats that can impair interactive systems.

This chapter presents a systematic identification of the faults that can be introduced in the system during both development and operations time and how formal methods can be used in such context. To exemplify such usage of formal methods, we present their use to describe software architecture and self-checking components to address these faults in the context of interactive systems. As those faults are more likely to occur in the high atmosphere, the proposed solutions are illustrated using an interactive application within the field of interactive cockpits which is described in detail in the case study chapter of the book.

This chapter is structured as follows. The next section focuses on the identification of the issues to be addressed for improving the dependability of critical interactive systems. The third section focuses on the identification of potential solutions for addressing these different issues, leading to the identification of issues that have to be investigated in the future. The fourth section discusses how the use of formal methods can help for addressing these issues. The fifth section illustrates the discussion with the example of the use of formal methods when dealing with both development software faults and operational natural faults in interactive cockpits development. Section 20.6 presents the perspectives of this work and Sect. 20.7 concludes the chapter.

20.2 Identifying Issues for the Dependability of Interactive Systems

As mentioned in Introduction, building dependable interactive systems is a cumbersome task that raises the need to identify and treat the threats that can impair the dependability of interactive systems. This section focuses first on the identification of all the faults that can be introduced in the system during its development and all the faults that can affect it during operations (on system, environmental and human side). This section then presents the different ways of dealing with these faults.

20.2.1 Fault Taxonomy

To be able to ensure that the system will behave properly whatever happens, a system designer has to consider all the issues that can impair the functioning of that system. To this end, the domain of dependable computing has defined a taxonomy of faults, e.g. Avižienis et al. (2004). This taxonomy leads to the identification of 31 elementary classes of faults. Figure 20.1 presents a simplified view of this taxonomy and makes explicit the two main categories of faults (top level of the figure): (i) the ones made at development time (see left-hand side of the figure), including bad designs and programming errors and (ii) the one made at operation times (see right-hand side of the figure) including operator errors such as slips, lapses and mistakes as defined in Reason (1990).

The leaves of the taxonomy are grouped into five different categories as each of them brings a special problem (issue) to be addressed:

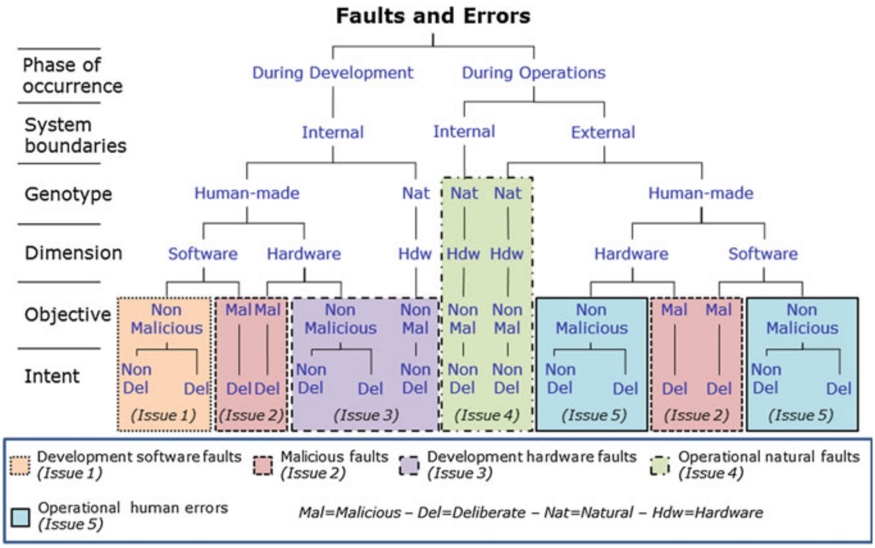


Fig. 20.1 Taxonomy of faults in computing systems (adapted from Avižienis et al. 2004) and associated issues for the dependability of these systems

- *Development software faults (issue 1)*: software faults introduced by humans during system development. They can be, for instance, bad design errors, bugs due to faulty coding, development mistakes.
- *Malicious faults (issue 2)*: faults introduced by humans with the deliberate objective of damaging the system. They can be, for instance, an external hack causing service denial or crash of the system.
- *Development hardware faults (issue 3)*: natural (e.g. caused by a natural phenomenon without human involvement) as well as human-made faults affecting the hardware during its development. They can be, for instance, a short circuit within a processor (due to bad construction).
- *Operational natural faults (issue 4)*: faults caused by a natural phenomenon without human participation, affecting hardware as well as information stored on hardware and occurring during the service of the system. As they affect hardware, faults are likely to propagate to software as well. They can be, for instance, a memory alteration due to a cosmic radiation.
- *Operational human errors (issue 5)*: faults resulting from human action during the use of the system. They include faults affecting the hardware and the software, being deliberate or non-deliberate but do not encompass malicious ones. Connection between this taxonomy and classical human error classification as the one defined in Reason (1990) can be easily made with deliberate faults corresponding to mistakes or violations (Polet et al. 2002) and non-deliberate ones being either slips or lapses.

20.2.2 Approaches for Addressing Faults

In the domain of dependable systems, fault-related issues have been studied, and the current state of the art identifies four different ways of increasing systems' dependability as presented in Avižienis et al. (2004) and Bowen and Stavridou (1993):

- *Fault prevention*: avoiding as much as possible the introduction of faults during the development of the system. It is usually performed by following rigorous development processes, the use of formal description techniques, proving properties over models, as well as introducing barriers as proposed in Hollnagel (2004) and Basnyat et al. (2007).
- *Fault removal*: reducing the number of faults that can occur. It can be performed (i) during system development, usually using verification of properties (Pnueli 1986), theorem proving, model checking, testing, fault injection or (ii) during the use of the system via corrective maintenance for instance.
- *Fault tolerance*: avoiding service failure in the presence of faults via fault detection and fault recovery. Fault detection corresponds to the identification of the presence of faults, their type, and possibly their source. Fault recovery aims at transforming the system state that contains one or more faults into a state without fault so that the service can still be delivered. Fault recovery is usually achieved by adding redundancy or diversity using multiple versions of the same software. Fault mitigation is another aspect of fault tolerance whose target is to reduce the impact of faults.
- *Fault forecasting*: estimating the number, future incidence and likely consequences of faults (usually by statistical evaluation of the occurrence and consequences of faults).

20.3 Addressing the Dependability of Interactive Systems

As mentioned previously, the dependability of an interactive system can be achieved if and only if each group of fault identified in the previous section is covered. In this section, we present a quick overview of the state of the art on addressing all these groups of faults. This state of the art does not aim at describing exhaustively all the existing approaches but targets at providing an overview of these approaches in order to identify the issues that have been studied in the state of the art and the issues that have to be investigated in the future for addressing all these faults in the area of interactive systems. How formal methods can be used to deal with these faults is presented in Sect. 20.4.

20.3.1 Addressing Development Software Faults (Issue 1)

Development software faults are usually addressed using fault-prevention, fault-removal and fault-tolerance approaches. Due to the intrinsic characteristics of interactive system, standard software engineering approaches cannot be reused for building reliable interactive systems. To address development software faults, a lot of work has been carried out in the engineering interactive systems community extending and refining approaches including software architecture (e.g. the ARCH model presented in Bass et al. 1991); testing (e.g. the generation of test cases for the graphical user interface as proposed in Memon et al. 2001); the use of dedicated process (e.g. the process proposed in Martinie et al. (2012) taking into account the usability and criticality of interactive systems); the use of standardization (e.g. the Common User Access standard (IBM 1989) created by IBM to standardize WIMP interfaces) and the use of formal methods (e.g. the temporal logics (Johnson and Harrison 1992)). However, most of this work has been focusing on avoiding the occurrence of faults by removing software defects prior to operation, i.e. during the development of the interactive system, thus proposing fault prevention (processes, formal methods) and fault removal (test analysis using formal methods) approaches.

In the area of interactive cockpits (more details about this type of application are provided in the case study section), Barboni et al. proposed in (2006) the use of a dedicated formal description technique for describing in a complete and unambiguous way widgets, windows managers and dialog controller (thus covering the software part of the interactive system). This is in line with DO 178B/C standards (RTCA and EUROCAE 2012) that, in that domain, promote the systematic use of formal methods (RTCA and EUROCAE 2011).

20.3.2 Addressing Malicious Faults (Issue 2)

This issue has received a lot of attention in the field of human–computer interaction, even leading to a specialized symposium on privacy and security called SOUPS (Symposium on Usable Privacy and Security) since 2005. However, the human–computer interaction community work on usability is particularly focused on usability of authentication (e.g. the work of Wright et al. that proposed in (2012) to apply recognition to textual passwords, thus limiting the number of errors while typing the password). The issues of data protection and integrity are thus left to the computing security field (e.g. the work of Reiter and Stubblebine that proposed in (1999) a metric to evaluate the perceived confidence of an authentication technique).

Furthermore, security issues have not been studied when it comes to critical interactive systems. This is due to the fact that these systems were until now closed and not accessible from the outside world, thus limiting the possibility of security threats. However, with the introduction of new technologies such as Wi-fi connectivity for maintenance purposes in aeronautics, the security issue is becoming relevant as shown, for example, in the work of Dessiatnikoff et al. that presented in (2013) how the functioning of avionics embedded systems can be affected by some attacks.

20.3.3 Addressing Development Hardware Faults (Issue 3)

Hardware components used for the development of interactive systems are quite similar to the one used in other computing systems. Development hardware faults can thus be addressed with classical hardware development processes as the one described in the DO 254 standard (RTCA and EUROCAE 2000), providing guidance for the development of airborne electronic hardware.

However, when considering interactive systems, the major difference with non-interactive computing systems is that input devices have to be taken into account in the design of the interactive system. For instance, the introduction of multitouch interaction into an avionic cockpit cannot be done unless the screen has been proven to have a weak probability of being faulty (e.g. the probability of failure for the screen should be less than 10^{-9} per flight hour). Furthermore, a system designer has to take into account the reliability of the hardware used for the development of the system and can then compensate a lack of reliability through software or interaction means. For instance, touch screen reliability can be improved by replacing horizontal linear gestures by gestures linking two consecutive directions so that it is not depending on a unique grid on the touch screen.

20.3.4 Addressing Operational Natural Faults (Issue 4)

Addressing operational natural faults needs the use of fault-tolerance mechanisms as their occurrence is unpredictable and unavoidable. As explained in Introduction, the issue of operational natural faults has hardly been studied in the field of human–machine interaction and just a few contributions are available about this topic. For instance, Navarre et al. proposed in (2008) the reconfiguration of the interaction techniques or possibly the organization of display when required by the occurrence of hardware faults in order to guarantee the controllability of the interactive system under input devices failures.

However, this solution only addresses hardware aspects leaving software aspects uncovered. The issue of operational natural faults must be addressed, more particularly when dealing with safety-critical interactive systems as the one in aircraft as a higher probability of occurrence of these faults (Schroeder et al. 2009) concerns systems deployed in the high atmosphere (e.g. aircraft) or in space (e.g. manned spacecraft (Hecht and Fiorentino 1987)). The occurrence of natural faults demonstrates the need to go beyond classical fault prevention techniques at development time and to deal with faults that occur at operation time. Tankeu-Choitat et al. (2011a, b) proposed to address this aspect by embedding fault-tolerance mechanisms in the widgets behaviour. However, this is not sufficient as the entire interactive system must be fault-tolerant.

20.3.5 Addressing Operational Human Errors (Issue 5)

Human error has received a lot of attention in the field of psychology and HCI. This type of faults can be addressed by preventing or tolerating human error (Dearden and Harrison 1995), and several means have been identified to do so. A first way to address human error can be the use of barriers (Hollnagel 2004) whether or not being of software (e.g. confirmation buttons) or hardware nature (e.g. poka-yoké also called mechanical coding). Of course knowledge acquired from the accident or barrier analysis can be used to inform the development side. It is vital to prevent accidents from reoccurring (Basnyat et al. 2006) by, for instance, including socio-technical barriers which can be, in turn, modelled using a formal description technique in order to prevent the appearance of development faults in the barriers themselves (Basnyat et al. 2007). Human error prevention can also be done by specialized training designed within the system development as presented in Martinie et al. (2011) and better design of contextual help (Palanque et al. 1993) and user manuals (Bowen and Reeves 2012). Another way to address human error is to study user behaviour to modify the system design (e.g. the work of Rajkomar and Blandford, using cognitive models to analyse the impact of interruptions (2012) or the work of Palanque and Basnyat, using task patterns for taking into account user behaviours within the system development (2004)) to propose better designs (e.g. the work of Thimbleby and Gimblett, proposing better designs for keyed data entry (2011)) or to generate erroneous human behaviours in order to evaluate their impacts on the system (as in the work of Bolton et al. where model checking is used to evaluate the behaviour of the system in the presence of erroneous human behaviours (Bolton 2015; Bolton and Bass 2013 and Bolton et al. 2012)).

20.3.6 Concluding Remarks on the Identified Issues

The taxonomy presented in previous sections identifies all the faults that can impair an interactive critical system. This taxonomy groups faults in five categories, each of them raising different issues for the dependability of interactive critical systems. For each category, we have presented existing solutions highlighting their benefits and their limitations. While some issues have received a lot of attention from researchers, some received little and need to be addressed in order to build resilient interactive systems:

- (Issue 2) the faults addressing malicious attacks,
- (Issue 3) the faults affecting hardware during system development, and
- (Issue 4) the natural faults occurring during the system operation.

In the area of safety-critical systems, issue 2 is mainly addressed by confinement, i.e. having the system completely independent from other systems open to the outside world. This might not be a sustainable solution in the near future as maintenance requires opening up the systems and communication media that become wireless removing by nature the confinement. For this reason, issue 2 belongs to the issues that have to be investigated in the future. Issue 3 has to be addressed by the hardware manufacturers, and little can be done by researchers beyond the solutions presented above and would remain in the issues that have to be investigated in the future. Issue 4, on the opposite, has a strong potential for impacting operations and should receive full attention even though researchers in the field of HCI only start touching it.

20.4 Connection with Formal Methods

While formal methods are usually used in the field of human–computer interaction to deal with development software faults (to ensure that the system will be working) and operational human errors (to reduce the errors done by operators during operations), their use can help for all the different types of faults that have been identified in Sect. 20.2. While Sect. 20.3 has presented, for each one of the type of faults, how they are currently covered, this section presents how formal methods can be used to deal with them.

20.4.1 Development Software Faults

Formal methods are classically used to prevent and remove development software faults. Even if, as detailed by Hall (1990), they cannot guarantee a perfect software due to the imperfection of the human developers using a formal method either at modelling time or when moving from models to the actual code of the application. However, they are the only candidates for this purpose and their use brings many advantages. Indeed, as detailed by Stavely (1998), the use of formal methods forces software engineers to program more simply and clearly, thus preventing many development software faults. These results can be improved with the performance of software verification and testing, these two additional steps aiming at detecting the remaining development software faults, thus allowing to remove them.

However, as presented in previous sections, considering development software faults is not enough to guarantee the interactive system functioning as some other events (that are beyond the envelope of the specification) may occur and trigger unexpected behaviours from the specified system resulting in failures.

20.4.2 Malicious Faults

Malicious faults are usually dealt with by the introduction of authentication mechanisms such as the use of a password along with the use of stars to display it when it is typed.

In this case, formal methods can be used to describe in a complete and unambiguous way these authentication mechanisms. They also enable the verification of properties on these mechanisms along with their systematic testing.

20.4.3 Development Hardware Faults

Development hardware faults have to be addressed by the manufacturer of hardware equipment. However, when considering highly interactive devices such as multitouch interactions, tangible interactions (e.g. the use of fidget widgets as presented in Karlesky and Isbister 2013), the hardware and software are closely interleaved and cannot be treated separately.

In this case, formal methods can be used to develop jointly hardware and software. They also can be used as means for a better verification and validation of the associated hardware and software components.

20.4.4 Operational Natural Faults

Operational natural faults are usually dealt with by the introduction of detection and recovery mechanisms such as the use of fault-tolerant architecture which is illustrated in the next section.

In this case, formal methods can be used to describe in a complete and unambiguous way these detection and recovery mechanisms. They also enable the verification of properties on these mechanisms along with their systematic testing.

20.4.5 Operational Human Errors

Operational human errors are usually dealt with by the introduction of detection and recovery mechanisms such as the use of the rule that no action should be performed after the goal is reached to avoid post-completion error (e.g. forgetting the credit card after withdrawing some cash).

In this case, formal methods can be used for human behavioural modelling. They also enable the verification of properties on the behavioural models along with the verification and validation of the consistence between the behavioural model and the system model (e.g. using co-execution mechanisms enabling the coupling of tasks models and interactive applications as presented in Martinie et al. 2015).

20.5 Illustrative Example: Dealing with Both Development Software Faults and Operational Natural Faults in Interactive Cockpits

This section illustrates how formal methods can be used to deal with both development software faults (their classical use) and operational natural faults. To this end, we first present the use of a formal notation to describe the system behaviour (in order to prevent development software faults) and then a solution for building fault-tolerant interactive applications (in order to address natural faults occurring at run-time). These two approaches are illustrated using the example of the Flight Control Unit Software (FCUS) interactive cockpits (introduced in the case study chapter of the book).

It is important to note that we only describe here the elements that are relevant for dealing with operational faults. More information can be found in Fayollas et al. (2013, 2014).

This section is divided into four subsections: the first subsection describes the hypotheses of this illustrative example and the functional failures that are taken into account in the approach. The second subsection describes the preventive approach to deal with development software faults and its application to the FCUS case study. The third subsection describes the fault-tolerance approach to deal with operational natural faults, its application to the FCUS case study and how this approach relates to formal methods. The last subsection presents how to apply these principles to the entire interactive system.

20.5.1 Main Hypotheses and Functional Failures Taken into Account

Considering the case study of interactive cockpit applications and more particularly the FCUS application (c.f., “Case Study 3—Interactive Aircraft Cockpits” section in Chap. 4), our focus is on development software faults and operational natural faults. We thus consider the system-side dependability of interactive system and consider human error as out of scope. This is indeed a very strong hypothesis, but human reliability aspects can be considered independent from the ones addressed here as neither development software faults nor natural faults at operation time are influenced by operator’s behaviour (and vice versa).

More precisely, the solution presented in this section focuses on the functional failures of the two generic software parts of the interactive cockpits architecture (presented in the “Case Study 3—Interactive Aircraft Cockpits” section in Chap. 4 of the book, reproduced in Fig. 20.2 for sake of direct access to information)—i.e. the widgets and the server. The User Application (which is controlling the behaviour of the application), the aircraft components and the input and output devices are considered out of the scope.

The solution we describe aims at ensuring that the interactive system processes correctly the input events from operators and renders correctly parameters received from the User Application. To be more concrete, we are targeting at managing four possible functional failures that could lead to catastrophic events according to the DO-178C terminology:

- ***Loss of control***: loss of function so that control (from crew member to User Application) is not performed. For instance, the user clicks on the AP1 (Auto Pilot 1) button, but the FCUS User Application does not receive the corresponding *A661_Event*.
- ***Erroneous control***: malfunction so that control is performed in an inappropriate manner (wrong control or correct control sent with wrong data). For instance, the user clicks on the AP1 button but the *A661_Event* corresponding to the click on FD (for flight director) button is sent to the FCUS User Application.

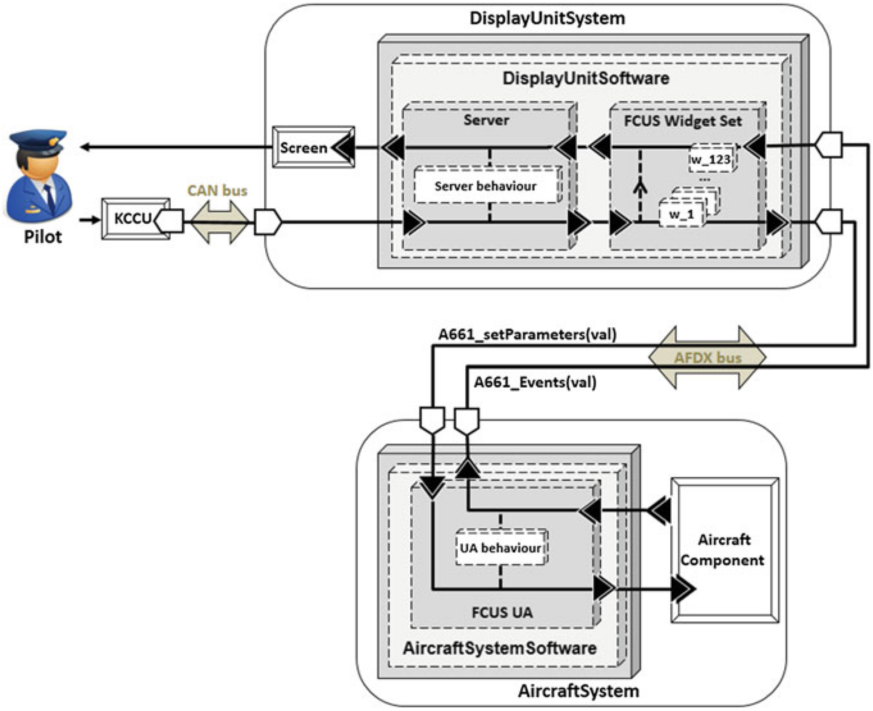


Fig. 20.2 Interactive cockpits architecture exemplified with the FCUS application

- ***Loss of data display***: loss of function so that the data display (from User Application to Control and Display System) is not performed. For instance, the FCUS User Application sends a setParameter to a widget for displaying the notification of AP1 activation but the corresponding graphical update (display of three green bars on the AP1 button) is not performed.
- ***Erroneous data display***: malfunction so that the data display is performed in an inappropriate manner and thus may be misleading to crew members. For instance, the FCUS User Application is asked to display the notification of AP1 activation but the graphical update corresponding to the activation of FD is performed instead of it.

20.5.2 Dealing with Development Software Faults in Interactive Cockpits

This section presents the preventive approach to deal with development software faults. The first subsection introduces the approach, while the second presents its application on the FCUS case study.

20.5.2.1 A Fault Prevention Approach Using Formal Description Techniques

Building software without any development software faults is also known as building a so-called zero-defect software which is very difficult to achieve (Hamilton 1986). Nevertheless, it is possible to build almost zero-defect software by the use of formal methods (Stavely 1998), thus corresponding to using **fault prevention** techniques. They can then be supplemented by **fault detection** techniques such as formal models testing or formal proofs that can then lead to **fault removal**.

Our goal here is to prevent the occurrence of software development faults as much as possible. In that intent, we propose the use of a model-based development for the interactive software. This is achieved through the use of a formal description technique dedicated to the specification and verification of interactive systems. To this purpose, we propose the use of the ICO (interactive cooperative objects) formalism (Navarre et al. 2009). This formalism uses high-level Petri nets (Genrich 1991) to describe the behavioural aspects of the interactive system and concepts borrowed from the object-oriented approach to describe the structural and static aspects of the interactive system. This formalism enables the handling of the specific aspects of interactive systems such as their event-driven nature. Its main interest is to provide a way for the interactive system developer to create non-ambiguous and concise models compliant with the system specifications.

Due to space constraints, we will not present here the definition of the ICO notation. However, the interested reader can find them in Navarre et al. (2009). Furthermore, we only present here the description of interactive systems behaviour using this notation, but the ICO formalism supports a process towards the reliability of interactive software due to its formal definition and its groundings in Petri nets theory providing means to formally verify the properties over ICO models. It thus enables the verification of an interactive software through the structural analysis of its ICO formal specification (as presented in Palanque and Bastide 1995) or through the analysis of its marking graph (as presented in Palanque and Bastide 1997). The ICO notation is supported by a tool named PetShop providing means to develop ICO models, formally verifying properties over them and also providing a run-time support for models' execution. The infrastructure of PetShop and its ability to support both prototyping and verification can be found here (Palanque et al. 2009).

In a previous work (Barboni et al. 2006), we propose the whole modelling of the interactive cockpit using the ICO formalism (except the server for which few limited parts are described by Java code) in order to address reliability aspects of such applications. Due to place and legibility issues, we will not describe in this chapter all the interactive cockpit modelling. However, we illustrate it with the use of the ICO formalism to describe the behaviour of a PicturePushButton (which is, among more than 60 widgets, defined by the ARINC 661 standard (Airlines Electronic Engineering Committee 2002), a widely used widget as demonstrated by the FCUS application, presented in "Case Study 3—Interactive Aircraft Cockpits" section in Chap. 4, using more than 30 instances of it). A PicturePushButton is a simple user interface button enabling the display of information through either a picture or a label allowing commands to be triggered. The associated command is triggered only if an event click is received by the button and if the PicturePushButton is both visible and enabled. Following the ARINC 661 specification, the aircraft application may only modify widgets' state by calling setParameter method on their parameters modifiable at run-time, and widget can only raise events to be caught by the aircraft application.

20.5.2.2 Illustration with the FCUS Case Study

Figure 20.5 presents the complete formal specification of a PicturePushButton using the ICO formalism. The point here is not to present in detail such a specification (that in any case should be manipulated by means of a dedicated tool such as PetShop) but to provide some insights on how the various elements of such an interactive component are described and how these elements are connected.

The parameters accessible by an avionics application for a PicturePushButton widget are *Visible*, *Enable*, *StyleSet*, *PictureReference* and *Labelstring*. Figure 20.3 details the ICO modelling of a PicturePushButton's method *SetVisible(X)* (see *Visible management* in Fig. 20.5). If the formal parameter *X* holds the value *True,* then the widget is set to the state *Visible*. If the value is *False,* the widget is set to

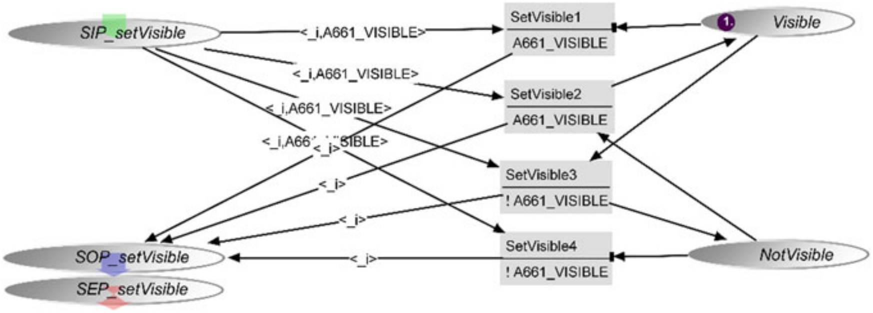


Fig. 20.3 ICO model of the management of the visible parameter for the PicturePushButton

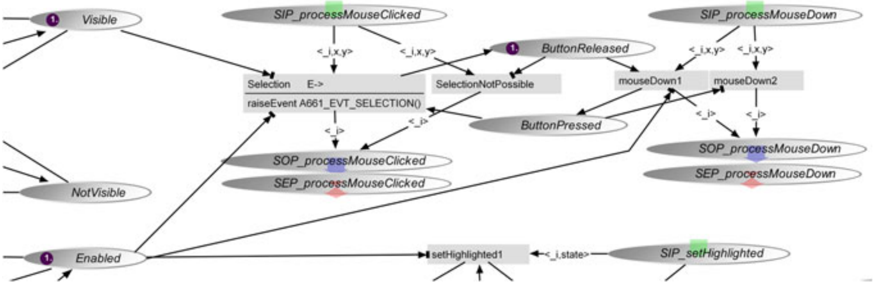


Fig. 20.4 ICO model of mouse click management for the PicturePushButton

the state *NotVisible*. Using the ICO formalism, these method calls are modelled by an input place (holding the name of the method) SIP_setVisible, and the return value is modelled by an output place SOP_setVisible. The Petri net in between the input and output places models the detailed behaviour of the widget in order to process the method call.

The PicturePushButton can only send information by raising the *A661_EVT_SELECTION* event. The connection of the widgets to the input devices (as shown on the architecture of interactive cockpits in Fig. 20.2) is done by methods call performed by the server. This is modelled in the same way as the method calls presented above. For the PicturePushButton, those methods are *processMouseClicked*, *processMouseDown*, and *processMouseReleased*. According to the inner state of the widget (*Visible* and/or *Enabled*), these method calls are processed and produce different output. For instance, as detailed in Fig. 20.4, if the widget is *Enabled*, *Visible* and already received a *processMouseDown* method call (token in *ButtonPressed* place instead of token in *ButtonReleased* place), a call of the method *processMouseClicked* will raise the event *A661_EVT_SELECTION* (*Selection* transition firing).

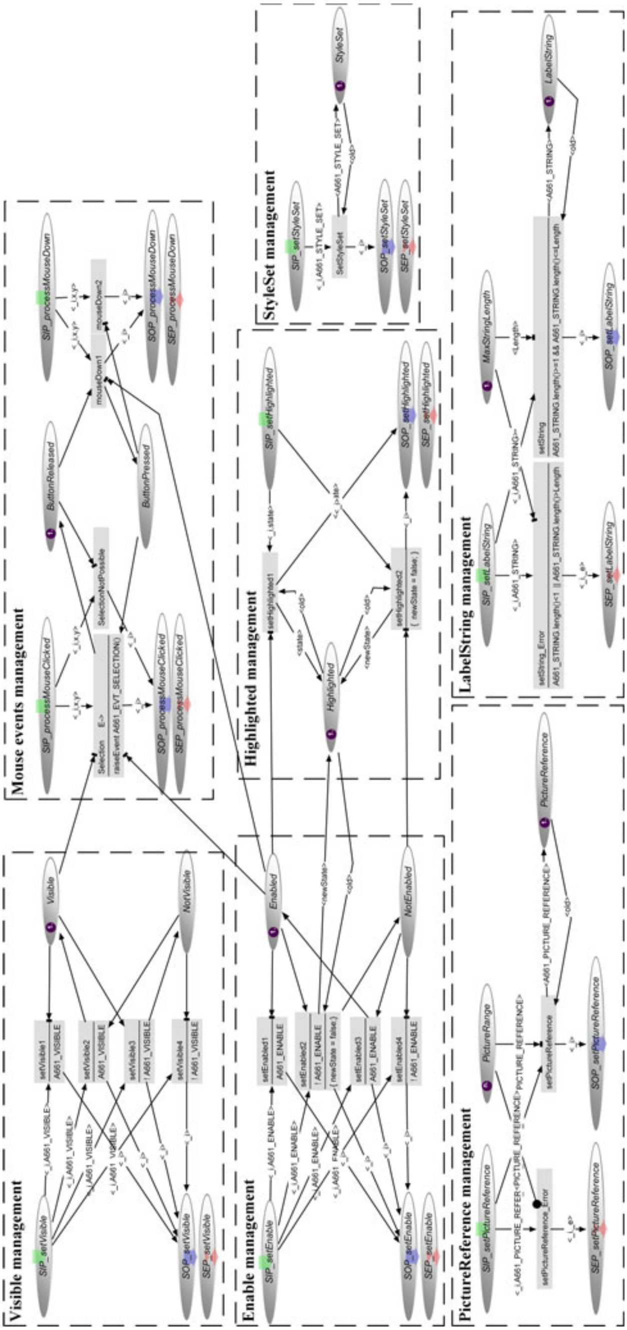


Fig. 20.5 ICO model of a PicturePushButton

20.5.3 Dealing with Operational Natural Faults in Interactive Cockpits

This section presents the fault-tolerance approach to deal with operational natural faults. It is divided into four subsections: the first subsection presents related works on fault-tolerant systems. The second subsection presents architecture for fault-tolerant interactive components and the third subsection presents the application of this architecture on the FCUS case study. Finally, the last subsection presents how this work relates to formal methods.

20.5.3.1 Related Work on Fault-Tolerant Systems

In the previous section, we have presented our solution for avoiding software faults prior to operation. However, as explained in the first section, a preventive approach, dealing with development software faults, is not sufficient as interactive system crashes may occur at run-time, due to one of the four other fault groups (malicious faults, development hardware faults, operational natural faults and human error). In this section, we focus on operational natural faults such as subtle transient faults due to single-event effects (Normand 1996). To deal with these faults, we propose to integrate, within the previous approach, standard dependability mechanisms.

In the area of dependable computing software, many approaches have been investigated in order to embed fault-tolerance mechanisms into software systems. The most commonly used techniques are self-checking components (Laprie et al. 1990 and (Yau 1975), N-Version Programming (Yeh 1996) and N-Self-Checking Programming (Laprie et al. 1990).

The approach presented in this paper builds on top of the self-checking components technique as many dependability strategies rely on it and as it has been proven to be effective for many safety-critical functions in avionic systems (Traverse et al. 2004).

According to Laprie et al. (1990), a self-checking approach consists in adding redundancy to a program so that it can check its own dynamic behaviour at execution time. As presented in Fig. 20.6, a self-checking component is also called COM-MON (Traverse et al. 2004); COM stands for COMmand (e.g. the program) and MON stands for MONitoring (e.g. the redundancy mechanism enabling the checking of the behaviour of the program).

Natural faults are detected by the redundancy mechanisms but only if the two software components (COM and MON) are executed in segregated context to avoid causality between failures (a fault in the COM generating a failure in the MON) and to avoid common point of failure (a natural fault impacting the COM and the MON together). The self-checking component has thus the capability to achieve fault detection, triggering error notifications when faults are detected (when there is a discrepancy between the behaviour of COM and MON).

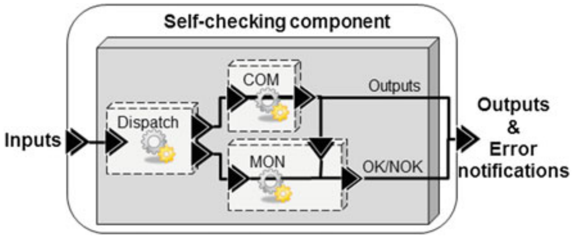


Fig. 20.6 Architecture of a self-checking component (detection only)

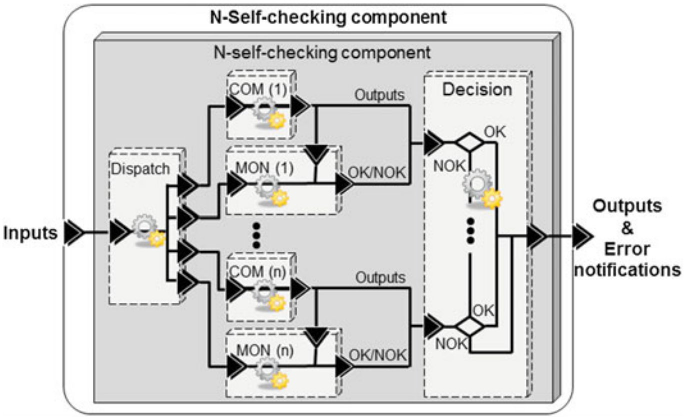


Fig. 20.7 Fault-tolerant (including detection and recovery) architecture for interactive component

A self-checking component can only perform fault detection. It has thus to be further extended for being able to achieve fault recovery. This can be achieved, for instance, by introducing a specific mechanism to handle the error notification or by using the N-Self-Checking Programming concepts. The N-Self-Checking architecture, as presented in Fig. 20.7, is composed of more than two self-checking components which are given a sequence number. The input events are dispatched to all self-checking components and are processed in parallel by them. Every output or error notification is delivered to a decision mechanism. If the first self-checking component provides a result without any error, then the output is transmitted as an output. Otherwise, the decision mechanism does not consider the first self-checking component anymore and uses the result of the next in a row. This is done until the last self-checking component fails, then an error is raised. If all self-checking widgets are faulty, then the N-self checking component is only detecting faults. If one of the self-checking components is executed without fault, then the previous faults (if any) have been detected and recovered.

20.5.3.2 A Self-checking Architecture for Fault-Tolerant Interactive Components

In order to embed both fault detection and recovery (thus embedding fault tolerance) in interactive systems, we propose to use the self-checking mechanism along with the introduction of specific recovery mechanisms in interactive systems. When considering the interactive cockpit architecture (see Fig. 20.2), we choose to address the Control and Display System for the first time as it is composed of generic components (the server and the widgets) that will be reused for each interactive application in the cockpit. The corresponding fault-tolerant architecture is presented in Fig. 20.8. In this architecture, the Control and Display System is implemented as a self-checking component: it is composed of two subcomponents (the COM and the MON). It is therefore able to send error notifications to the User Application which implements some recovery mechanisms.

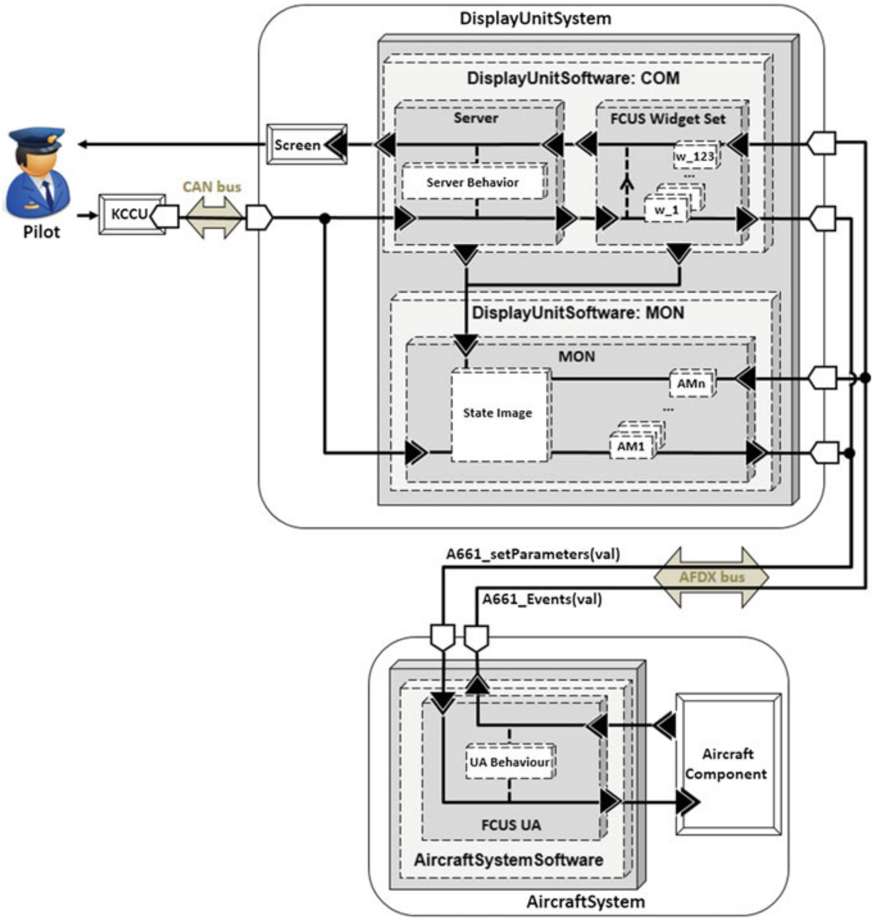


Fig. 20.8 Fault-tolerant interactive cockpit architecture with the FCUS application

Once this architecture is defined, a challenge remains in the definition of the monitoring (MON) component. Several ways of implementing this component were investigated in Fayollas et al. (2013). This investigation leads to the choice of an implementation following an assertion-based monitoring principle. In that case, the MON component is responsible for the verification of several properties associated with the interactive objects organization and their semantics in operational context. These properties are run-time assertions designed to detect errors that could lead to system malfunctions. Each property (or assertion A) is associated with a dedicated assertion monitor (AM). The monitoring component implements thus a state image (i.e. *State image* in Fig. 20.8) that is composed of all the data necessary to compute the monitoring of all the assertions (performed by all the assertion monitors, i.-e. *AM1* … *AMn* in Fig. 20.8). More details on this architecture and its implementation can be found in Fayollas et al. (2014).

The recovery here is achieved by the introduction of a specific mechanism into the User Application behaviour to handle the error notification. The User Application can thus implement specific mechanisms enabling the recovery from the detected faults depending on the application semantics. Another possibility would be that the User Application notifies the operator of detected faults. In this case, recovery is left with the operator. This approach has the advantage of keeping the operator in the loop and leaving with him/her the decision on how to handle the fault. However, such an approach cannot be used when multiple faults occur in a very short period of time.

As explained briefly in the related work section, a self-checking mechanism is not enough to ensure fault detection if a fault occurring on one component (e.g. COM) potentially interferes with another component (e.g. MON). This would be the case if all the components of the architecture are executed in the same partition. ARINC 653 (Airlines Electronic Engineering Committee 2003) defines such partitioning in the domain of civil aviation, and the architecture proposed in this paper makes explicit the segregation of COM and MON being here represented by two different virtual processors (light grey boxes in Fig. 20.8) that can be considered as separated ARINC 653 partitions.

The next section presents an example of an assertion and its monitors within the fault-tolerant FCUS application.

20.5.3.3 Illustration with the FCUS Case Study

To exemplify the assertion that has to be monitored and its associated monitors, we propose to use once again the PicturePushButton example and more precisely, its management of the reception of a *processMouseClicked* method call. We only present here a small example, but the whole process for assertions monitoring is available in Fayollas et al. (2014).

Figure 20.9 presents formal definition of the corresponding assertion. It is built on the safety analysis of the PicturePushButton behaviour described by the ICO model detailed in Fig. 20.4. Thus, this assertion is defined as the following: the

A1: Process mouse click in a PicturePushButton

Let w be a PicturePushButton,
let f = {source, target, functionName, parameters} be a function call,
let We = {source, eventName, parameters} be a widget event

f = {source, w, processMouseClicked, parameters}
$\quad \wedge$ *w.buttonPressed = true* $\wedge$ *w.visible = true* $\wedge$ *w.enabled = true*
$\Leftrightarrow$
We = {w, A661_EVT_SELECTION, ∅}

Fig. 20.9 Formal definition of the assertion A1: process mouse click in a PicturePushButton

sending of an *A661_EVT_SELECTION* by a PicturePushButton should be performed if and only if the corresponding widget is *Enabled*, *Visible*, has already received a *processMouseDown* method call (the *ButtonPressed* state being thus true) and received call of the method *processMouseClicked*.

This assertion leads to the definition of two assertion monitors that are depicted in Fig. 20.10. The first assertion monitor (*AM1* in Fig. 20.10) enables the detection of both a lack of execution and an erroneous execution of the *processMouseClicked* method call. The second assertion monitor (*AM2* in Fig. 20.10) enables the detection of the sending of an *A661_EVT_SELECTION* without any *processMouseClicked* method call.

20.5.3.4 Connection with Formal Methods

Work done in the area of fault-tolerant systems usually proposes generic solutions remaining at a high level of abstraction such as the architecture we presented in the two previous sections. Of course, when going to implementation, there is a need to refine such architecture describing in detail the inputs and outputs of each component of the architecture (COM and MON components) as well as the connections between those components. It is then needed to produce low-level description of the behaviour of the various components of the architecture. Such behaviour can be rather complex and their integration even more.

This can be achieved with the use of formal methods. For instance, the fault-tolerant architecture presented in the previous section can also be supported by formal methods. To this purpose, we presented, in Fayollas et al. (2014), the use of a safety process to obtain the formal definitions of the assertions that have to be monitored. This process builds on the interactive system architecture along with the ICO formal description of all its software components.

Therefore, the use of formal methods when dealing with operational natural faults offers two main benefits: on the one hand, it provides support for the definition of the assertions that have to be monitored; on the other hand, it enables taking advantages of all the benefits they provide (development faults prevention, support for verification, analysis and fault removal techniques) for the definition of the assertions and their monitors. Indeed, the assertion monitors are software components, and they are thus prone to development software faults.

```
A1.AM1: ppb.processMouseClicked.assert
//MON state
boolean w.visible, w.enabled;

// ppb.processMouseClicked.assert
int errorDetected =-1;

if (functionCall== {source, w, processMouseClicked, parameters}){
      if (w.visible == true && w.enabled == true){
            boolean timeOut = startTimer();
      }
}
while (!timeOut){
      if (! timeOut && widgetEvent.contains({w,A661_EVT_SELECTION,Ø})){
            errorDetected = 0;
            sendError(functionCall, errorDetected);
     }
}
if (timeOut && errorDetected ==-1){
      errorDetected = 1;
      sendError(functionCall, errorDetected);
}
```

```
A1.AM2: A661_EVT_SELECTION.assert
//MON state
boolean w.visible, w.enabled;


// ppb.processMouseClicked.assert
int errorDetected = 0;

if (widgetEvent == {w,A661_EVT_SELECTION,Ø}){
      if (functionCall.contains({source, w, processMouseClicked, parameters}) && w.visible == true
            && w.enabled == true){
            errorDetected = 0;
            sendError(functionCall, errorDetected);
     }else{
            errorDetected = 1;
            sendError(functionCall, errorDetected);
      }
}
```

Fig. 20.10 Implementation of the assertion monitors in C for the assertion A1: process mouse click in a PicturePushButton

20.5.4 Dealing with These Faults for the Entire Interactive System

When computing systems are considered in terms of dependability, the dependability level of the entire system is the one of its weakest components. We have proposed in previous sections architecture for interactive components able to detect and recover from faults in the Control and Display System component (composed of the server and the widgets). However, similar mechanisms have to be defined for the other components of the interactive cockpits architecture presented in Fig. 20.2 (i.e. input and output devices, device drivers and User Application).

Furthermore, as stated in Introduction, we focused in this section on the Control and Display System (widgets and server) and adding fault-tolerance mechanisms to these two components raises additional challenges due to the timing constraints related to short feedback loop of input device event handling. We are currently working on those aspects in the context of both WIMP interaction techniques, and direct manipulation and multitouch (Hamon et al. 2013).

20.6 Future Work

While formal methods have proven their usefulness for developing the system right (Boehm 1984) by allowing the verification of safety and liveness properties, the issue of building the right system is left to the area of requirement engineering (on the input side) and validation (on the output side). These aspects are beyond the content of the chapter but are of course of primary importance in terms of both functional requirements (ensuring that the system will offer the expected functionalities) and non-functional requirements (ensuring properties such as usability and safety). To this purpose, verification techniques must be complemented by other approaches such as requirements and need elicitation techniques (e.g. personas, brainstorming) and software or usability testing.

Similarly, formal methods can be very useful for describing the dependability mechanisms that have been presented above in order to ensure that they behave correctly, but they will not allow to assess that these mechanisms actually detect faults and that recovery mechanisms will be adequate and understandable by the operators. Such objectives can only be reached by performing test campaigns involving, for instance, fault injection techniques fed with operational data related to the underlying fault model (Arlat et al. 1990).

Lastly, human error is usually beyond the grasp of formal methods even though some previous work has proposed their use for workload analysis (Ruksenas et al. 2009) or human-to-human communication analysis (Bolton 2015). Human reliability analysis has thus to be complemented with more informal approaches such as the ones described in Boring et al. (2010) or based on standard tabular safety analysis methods (Department of the Army 2006) extended to encompass human aspects and especially human errors (Martinie et al. 2016).

20.7 Conclusion

This chapter argues that formal methods are good candidates for addressing software development faults. However, through the presentation of a complete and systematic taxonomy of the faults that can affect interactive systems, this chapter argues that considering development software faults is not enough to guarantee the interactive system dependability. Exploiting this taxonomy, this chapter has outlined issues that have to be investigated in the future to improve interactive systems' dependability, taking into account the faults made at development time and faults occurring at operation time. These issues demonstrate that the four other sources of failures (malicious faults, hardware development faults, operational natural faults and operational human errors) have to be considered. Besides, this chapter also argues that formal methods can be useful when dealing with these other groups of faults, and we have demonstrated how they can be useful for describing natural faults detection and natural faults recovery mechanisms. It is important to note that the proposed formal method was the interactive cooperative objects one (a Petri nets-based notation) but any other one with a similar expressive power would fit.

Acknowledgements This work is partly funded by Airbus under the contract CIFRE PBO D08028747-788/2008.

References

Airlines Electronic Engineering Committee (2002) ARINC Specification 661: cockpit display system interfaces to user systems

Airlines Electronic Engineering Committee (2003) ARINC Specification 653: avionics application software standard interface

Arlat J, Aguera M, Amat L et al (1990) Fault injection for dependability validation: a methodology and some applications. IEEE Trans Softw Eng 16(2):166–182

Avižienis A, Laprie JC, Randell B, Landwehr C (2004) Basic concepts and taxonomy of dependable and secure computing. IEEE Trans Dependable Secure Comput 1(1):11–33

Barboni E, Conversy S, Navarre D, Palanque P (2006) Model-based engineering of widgets, user applications and servers compliant with ARINC 661 specification. In: Interactive systems, design, specification, and verification. Springer, Berlin, Heidelberg, pp 25–38

Basnyat S, Chozos N, Palanque P (2006) Multidisciplinary perspective on accident investigation. Reliab Eng Syst Saf 91(12):1502–1520

Basnyat S, Palanque P, Schupp B, Wright P (2007) Formal socio-technical barrier modelling for safety-critical interactive systems design. Saf Sci 45(5):545–565

Bass L, Little R, Pellegrino R, Reed S, Seacord R, Sheppard S, Szezur MR (1991) The arch model: Seeheim revisited. UI Developpers' WorNshop, 1

Boehm BW (1984) Verifying and validating software requirements and design specifications. IEEE Softw 1(1):75

Bolton ML (2015) Model checking human-human communication protocols using task models and miscommunication generation. J Aerosp Inf Syst 12(7):476–489

Bolton ML, Bass EJ (2013) Generating erroneous human behavior from strategic knowledge in task models and evaluating its impact on system safety with model checking. IEEE Trans Syst Man Cybern: Syst 43(6):1314–1327

Bolton ML, Bass EJ, Siminiceanu RI (2012) Generating phenotypical erroneous human behavior to evaluate human–automation interaction using model checking. Int J Human Comput Stud 70 (11):888–906

Boring RL, Hendrickson SM, Forester JA, Tran TQ, Lois E (2010) Issues in benchmarking human reliability analysis methods: a literature review. Reliab Eng Syst Saf 95(6):591–605

Bowen J, Reeves S (2012) Modelling user manuals of modal medical devices and learning from the experience. In: Proceedings of ACM symposium on engineering interactive computing systems. ACM, pp 121–130

Bowen J, Stavridou V (1993) Formal methods, safety-critical systems and standards. Softw Eng J 8(4):189–209

Dearden AM, Harrison MD (1995) Formalising human error resistance and human error tolerance. In: Proceedings of the fifth international conference on human-machine interaction and artificial intelligence in aerospace, EURISCO

Department of the Army (2006) TM 5–698-4, Failure modes, effects and criticality analysis (FMECA) for command, control, communications, computer, intelligence, surveillance, and reconnaissance (C4ISR) facilities

Dessiatnikoff A, Nicomette V, Alata E, Deswarte Y, Leconte B, Combes A, Simache C (2013) Securing integrated modular avionics computers. In: Proceedings of the 32nd digital avionics system conference (DASC), Syracuse (NY, USA), 6–10 Oct

Fayollas C, Fabre JC, Palanque P, Barboni E, Navarre D, Deleris Y (2013) Interactive cockpits as critical applications: a model-based and a fault-tolerant approach. Int J Crit Comput-Based Syst 4(3):202–226

Fayollas C, Fabre JC, Palanque P, Cronel M, Navarre D, Deleris Y (2014) A software-implemented fault-tolerance approach for control and display systems in avionics. In: Proceedings of the IEEE 20th Pacific rim international symposium on dependable computing, pp 21–30

Genrich HJ (1991) Predicate/transitions nets. In: Jensen K, Rozenberg G (eds) High-levels petri nets: theory and application. Springer, Heidelberg, pp 3–43

Hall A (1990) Seven myths of formal methods. IEEE Softw 7(5):11–19

Hamilton MH (1986) Zero-defect software: the elusive goal: it is theoretically possible but difficult to achieve; logic and interface errors are most common, but errors in user intent may also occur. IEEE Spectr 23(3):47–53

Hamon A, Palanque P, Silva JL, Deleris Y, Barboni E (2013) Formal description of multi-touch interactions. In: Symposium on engineering interactive computing systems. ACM, pp 207–216

Hecht H, Fiorentino E (1987) Reliability assessment of spacecraft electronics. In: Annual reliability and maintainability symposium. IEEE, pp 341–346

Hollnagel E (2004) Barriers and accident prevention. Ashgage

IBM (1989) Common user access: advanced interface design guide. IBM, SC26-4582-0

Johnson C, Harrison M (1992) Using temporal logic to support the specification and prototyping of interactive control systems. Int J Man Mach Stud 37(3):357–385

Karlesky M, Isbister K (2013) Fidget widgets: secondary playful interactions in support of primary serious tasks. In: CHI ’13 extended abstracts on human factors in computing systems. ACM, pp 1149–1154

Laprie JC, Arlat J, Béounes C, Kanoun K (1990) Definition and analysis of hardware and software fault-tolerant architectures. IEEE Comput 23(7):39–51

Martinie C, Navarre D, Palanque P, Fayollas C (2015) A generic tool-supported framework for coupling task models and interactive applications. In: Proceedings of the 7th ACM SIGCHI symposium on engineering interactive computing systems. ACM, pp 244–253

Martinie C, Palanque P, Fahssi R, Blanquart JP, Fayollas C, Seguin C (2016) Task model-based systematic analysis of both system failures and human errors. IEEE Trans Human-Mach Syst 46(2):243–254

Martinie C, Palanque P, Navarre D, Barboni E (2012) A development process for usable large scale interactive critical systems: application to satellite ground segments. In: Proceedings of the 4th international conference on human-centered software engineering. Springer, Berlin, Heidelberg, pp 72–93

Martinie C, Palanque P, Navarre D, Winckler M, Poupart E (2011) Model-based training: an approach supporting operability of critical interactive systems. In: Proceedings of ACM symposium on engineering interactive computing systems. ACM, pp 53–62

Memon AM, Pollack EM, Soffa ML (2001) Hierarchical GUI test case generation using automated planning. IEEE Trans Softw Eng 27(2):144–155

Navarre D, Palanque P, Basnyat S (2008) Usability service continuation through reconfiguration of input and output devices in safety critical interactive systems. In: The 27th international conference on computer safety, reliability and security. LNCS 5219, pp 373–386

Navarre D, Palanque P, Ladry J, Barboni E (2009) ICOs: a model-based user interface description technique dedicated to interactive systems addressing usability, reliability and scalability. ACM TOCHI 16(4):1–56

Nicolescu B, Peronnard P, Velazco R, Savaria Y (2003) Efficiency of transient bit-flips detection by software means: a complete study. In: Proceedings of the 18th IEEE international symposium on defect and fault tolerance in VLSI systems. IEEE, pp 377–384

Normand E (1996) Single-event effects in avionics. IEEE Trans Nucl Sci 43(2):461–474

Palanque P, Basnyat S (2004) Task patterns for taking into account in an efficient and systematic way both standard and erroneous user behaviours. In: 6th international conference on human error, safety and system development. Springer, pp 123–139

Palanque P, Bastide R (1995) Verification of an interactive software by analysis of its formal specification. In: Proceedings of the IFIP TC 13 human-computer interaction conference, Lillehammer, Norway, 27–29 June 1995, pp 191–197

Palanque P, Bastide R (1997) Synergistic modelling of tasks, users and systems using formal specification techniques. Interact Comput 9:129–153

Palanque P, Bastide R, Dourte L (1993) Contextual help for free with formal dialogue design. In: Proceedings of the fifth international conference on human-computer interaction, Orlando, Florida, USA, 8–13 Aug, p 2

Palanque P, Ladry JF, Navarre D, Barboni E (2009) High-fidelity prototyping of interactive systems can be formal too. In: 13th international conference on human-computer interaction San Diego, CA, USA, LNCS, pp 667–676

Pnueli A (1986) Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. LNCS n° 224. Springer, pp 510–584

Polet P, Vanderhaegen F, Wieringa P (2002) Theory of safety related violation of system barriers. Cogn Technol Work 4(3):171–179

Rajkomar A, Blandford A (2012) A distributed cognition model for analysing interruption resumption during infusion administration. In: Proceedings of the 30th European conference on cognitive ergonomics. ACM, pp 108–111

Reason J (1990) Human error. Cambridge University Press

Reiter MK, Stubblebine SG (1999) Authentication metric analysis and design. ACM Trans Inf Syst Secur 2(2):138–158

RTCA and EUROCAE (2000) DO-254—design assurance guidance for airborne electronic hardware

RTCA and EUROCAE (2011) DO-333 formal methods supplement to DO-178C and DO-278A software tool qualification considerations

RTCA and EUROCAE (2012) DO-178C/ ED-12C, Software considerations in airborne systems and equipment certification

Ruksenas R, Back J, Curzon P, Blandford A (2009) Verification-guided modelling of salience and cognitive load. Formal Asp Comput 21(6):541–569

Schroeder B, Pinheiro E, Weber WD (2009) DRAM errors in the wild: a large-scale field study. In ACM SIGMETRICS, pp 193–204

Stavely AM (1998) Toward zero-defect programming, 1st edn. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA

Tankeu-Choitat A, Fabre JC, Palanque P, Navarre D, Deleris Y (2011a) Self-checking components for dependable interactive cockpits. In: Working conference on dependable computing, ACM

Tankeu-Choitat A, Navarre D, Palanque P, Deleris Y, Fabre JC, Fayollas C (2011b) Self-checking components for dependable interactive cockpits using formal description techniques. In: Proceedings of 17th IEEE Pacific rim international symposium on dependable computing

Thimbleby H, Gimblett A (2011) Dependable keyed data entry for interactive systems. In: Proceedings of the 4th international workshop on formal methods for interactive systems

Traverse P, Lacaze I, Souyris J (2004) Airbus fly-by-wire: a total approach to dependability. In: Proceedings of the 18th IFIP world computer congress, building the information society, pp 191–212

Wright N, Patrick AS, Biddle R (2012) Do you see your password?: applying recognition to textual passwords. In: Proceedings of symposium on usable privacy and security. ACM

Yau SS, Cheung RC (1975) Design of self-checking software. In: Proceedings of the international conference on reliable software. IEEE, pp 450–457

Yeh YC (1996) Triple-triple redundant 777 primary flight computer. In: IEEE aerospace applications conference, pp 293–307

Erratum to: The Handbook of Formal Methods in Human-Computer Interaction

Benjamin Weyers, Judy Bowen, Alan Dix and Philippe Palanque

Erratum to: B. Weyers et al. (eds.), *The Handbook of Formal Methods in Human-Computer Interaction*, Human-Computer Interaction Series, DOI 10.1007/978-3-319-51838-1

The original version of the book was inadvertently published without the following corrections:

In Chapter 2, S. Van Mierlo (e-mail: simon.vanmierlo@uantwerpen.be), Y. Van Tendeloo (e-mail: yentl.vantendeloo@uantwerpen.be), Y. Van Tendeloo, B. Meyers (e-mail: bart.meyers@uantwerpen.be), and H. Vangheluwe (e-mail: hv@cs.mcgill.ca) have to be changed to read as J. Bowen (e-mail: jbowen@waikato.ac.nz), A. Dix (e-mail: alanjohndix@gmail.com), A. Dix, P. Palanque (e-mail: palanque@irit.fr), and B. Weyers (e-mail: weyers@vr.rwth-aachen.de), respectively.

In Chapter 5, S. Van Mierlo (e-mail: simon.vanmierlo@uantwerpen.be) has to be changed to read as B. Weyers (e-mail: weyers@vr.rwth-aachen.de).

In Chapter 6, S. Van Mierlo and Y. Van Tendeloo (e-mail: simon.vanmierlo@uantwerpen.be) has to be changed to read as J. Bowen and S. Reeves (e-mail: jbowen@waikato.ac.nz); Y. Van Tendeloo (e-mail: yentl.vantendeloo@uantwerpen.be) has to be changed to read as S.Reeves (e-mail: stever@waikato.ac.nz).

The updated online version for this book can be found at
DOI 10.1007/978-3-319-51838-1_2,
DOI 10.1007/978-3-319-51838-1_5,
DOI 10.1007/978-3-319-51838-1_6,
DOI 10.1007/978-3-319-51838-1_8,
DOI 10.1007/978-3-319-51838-1_10,
DOI 10.1007/978-3-319-51838-1_11,
DOI 10.1007/978-3-319-51838-1_12,
DOI 10.1007/978-3-319-51838-1_13,
DOI 10.1007/978-3-319-51838-1_14,
DOI 10.1007/978-3-319-51838-1_15 and DOI 10.1007/978-3-319-51838-1_16

B. Weyers et al. (eds.), *The Handbook of Formal Methods in Human-Computer Interaction*, Human-Computer Interaction Series,
DOI 10.1007/978-3-319-51838-1_21

In Chapter 8, S. Van Mierlo and Y. Van Tendeloo (e-mail: simon.vanmierlo@uantwerpen.be) has to be changed to read as P. Curzon and R. Rukšėnas (e-mail: p.curzon@qmul.ac.uk); Y. Van Tendeloo (e-mail: yentl.vantendeloo@uantwerpen.be) has to be changed to read as R. Rukšėnas (e-mail: r.ruksenas@qmul.ac.uk).

In Chapter 10, S. Van Mierlo (e-mail: simon.vanmierlo@uantwerpen.be) has to be changed to read as B. Weyers (e-mail: weyers@vr.rwth-aachen.de).

In Chapter 11, S. Van Mierlo and Y. Van Tendeloo (e-mail: simon.vanmierlo@uantwerpen.be) has to be changed to read as G. Maudoux and C. Pecheur (e-mail: guillaume.maudoux@uclouvain.be); Y. Van Tendeloo (e-mail: yentl.vantendeloo@uantwerpen.be) and B. Meyers (e-mail: bart.meyers@uantwerpen.be) have to be changed to read as C. Pecheur (e-mail: charles.pecheur@uclouvain.be) and S. Combéfis (e-mail: s.combefis@ecam.be), respectively.

In Chapter 12, S. Van Mierlo and Y. Van Tendeloo (e-mail: simon.vanmierlo@uantwerpen.be) has to be changed to read as J. Bowen and A. Hinze (e-mail: jbowen@waikato.ac.nz); Y. Van Tendeloo (e-mail: yentl.vantendeloo@uantwerpen.be) has to be changed to read as A. Hinze (e-mail: hinze@waikato.ac.nz).

In Chapter 13, S. Van Mierlo (e-mail: simon.vanmierlo@uantwerpen.be) and Y. Van Tendeloo (e-mail: yentl.vantendeloo@uantwerpen.be) have to be changed to read as M.L. Bolton (e-mail: mbolton@buffalo.edu) and E.J. Bass (e-mail: ejb96@drexel.edu), respectively.

In Chapter 14, S. Van Mierlo (e-mail: simon.vanmierlo@uantwerpen.be) has to be changed to read as M.D. Harrison (e-mail: michael.harrison@ncl.ac.uk); Y. Van Tendeloo and B. Meyers (e-mail: yentl.vantendeloo@uantwerpen.be) has to be changed to read as P.M. Masci and J.C. Campos (e-mail: paolo.masci@inesctec.pt); B. Meyers (e-mail: bart.meyers@uantwerpen.be) and H. Vangheluwe (e-mail: hv@cs.mcgill.ca) have to be changed to read as J.C. Campos (e-mail: jose.campos@di.uminho.pt) and P. Curzon (e-mail: p.curzon@qmul.ac.uk), respectively.

In Chapter 15, S. Van Mierlo , B. Meyers, and N. Rungta (e-mail: simon.vanmierlo@uantwerpen.be) has to be changed to read as G. Brat , D. Giannakopoulou, and N. Rungta (e-mail: Guillaume.P.Brat@nasa.gov); B. Meyers (e-mail: bart.meyers@uantwerpen.be), N. Rungta (e-mail: Neha.S.Rungta@nasa.gov), Y. Van Tendeloo (e-mail: yentl.vantendeloo@uantwerpen.be), H. Vangheluwe (e-mail: hv@cs.mcgill.ca), and F. Raimondi (e-mail: f.raimondi@mdx.ac.uk) have to be changed to read as D. Giannakopoulou (e-mail: Dimitra.Giannakopoulou@nasa.gov), N. Rungta (e-mail: Neha.S.Rungta@nasa.gov), S. Combéfis (e-mail: sebastien.combefis@uclouvain.be; s.combefis@ecam.be), C. Pecheur (e-mail: charles.pecheur@uclouvain.be), and F. Raimondi (e-mail: f.raimondi@mdx.ac.uk), respectively.

In Chapter 16, G. Brat, S. Combéfis, and D. Giannakopoulou (e-mail: Guillaume.P.Brat@nasa.gov) has to be changed to read as S. Van Mierlo, Y. Van Tendeloo, and B. Meyers (e-mail: simon.vanmierlo@uantwerpen.be); S. Combéfis (e-mail: sebastien.combefis@uclouvain.be; s.combefis@ecam.be), D. Giannakopoulou (e-mail: Dimitra.Giannakopoulou@nasa.gov), C. Pecheur (e-mail:

charles.pecheur@uclouvain.be), and C. Pecheur have to be changed to read as Y. Van Tendeloo (e-mail: yentl.vantendeloo@uantwerpen.be), B. Meyers (e-mail: bart.meyers@uantwerpen.be), H. Vangheluwe (e-mail: hv@cs.mcgill.ca), and H. Vangheluwe, respectively.

The erratum book has been updated with the changes.