John Derrick John Fitzgerald
Stefania Gnesi Sarfraz Khurshid
Michael Leuschel Steve Reeves
Elvinia Riccobene (Eds.)

LNCS 7316

Abstract State Machines, Alloy, B, VDM, and Z

Third International Conference, ABZ 2012
Pisa, Italy, June 2012
Proceedings

Springer

Lecture Notes in Computer Science 7316

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

John Derrick John Fitzgerald
Stefania Gnesi Sarfraz Khurshid
Michael Leuschel Steve Reeves
Elvinia Riccobene (Eds.)

Abstract State Machines, Alloy, B, VDM, and Z

Third International Conference, ABZ 2012
Pisa, Italy, June 18-21, 2012
Proceedings

Springer

Volume Editors

John Derrick
University of Sheffield, UK, E-mail: j.derrick@dcs.shef.ac.uk

John Fitzgerald
Newcastle University, UK, E-mail: john.fitzgerald@ncl.ac.uk

Stefania Gnesi
ISTI-CNR, Pisa, Italy, E-mail: stefania.gnesi@isti.cnr.it

Sarfraz Khurshid
The University of Texas at Austin, USA, E-mail: khurshid@ece.utexas.edu

Michael Leuschel
Universität Düsseldorf, Germany, E-mail: leuschel@cs.uni-duesseldorf.de

Steve Reeves
The University of Waikato, Hamilton, New Zealand, E-mail: stever@waikato.ac.nz

Elvinia Riccobene
Università degli Studi di Milano, Crema, Italy, E-mail: elvinia.riccobene@unimi.it

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-30884-0 e-ISBN 978-3-642-30885-7
DOI 10.1007/978-3-642-30885-7
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012939228

CR Subject Classification (1998): F.4, G.2, I.2.3, D.3.2, F.3, I.2.4, F.4.1

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface to iFM & ABZ 2012

iFM 2012, the 9th International Conference on Integrated Formal Methods, and ABZ 2012, the Third International Conference on Abstract State Machines, Alloy, B, VDM, and Z, joined together in a single event, iFM&ABZ 2012, to celebrate Egon Börger's 65th birthday and his contribution to state-based formal methods.

This co-location of iFM&ABZ 2012 was hosted by the Institute of Scienza e Tecnologie dell'Informazione A. Faedo of the National Research Council (ISTI-CNR) of Italy and took place at the Area della Ricerca del CNR in Pisa during June 18–21, 2012.

We would like to thank everyone in Pisa for making us feel very welcome during our time there. It was a pleasure to run an event to honor Egon.

Professor Egon Börger was born in Bad Laer, Lower Saxony, Germany. Between 1965 and 1971 he studied at the Sorbonne, Paris (France), Université Catholique de Louvain and Institut Supérieur de Philosophie de Louvain (in Louvain-la-Neuve, Belgium), and the University of Münster (Germany). Since 1985 he has held a Chair in Computer Science at the University of Pisa, Italy. In September 2010 he was elected a member of the Academia Europaea.

Throughout his work he has been a pioneer of applying logical methods in computer science. Particularly notable is his contribution as one of the founders of the Abstract State Machine (ASM) method. Egon Börger has been cofounder and Managing Director of the Abstract State Machines Research Center (see `www.asmcenter.org`).

Building on his work on ASM, he was a cofounder of the series of international ASM workshops, which was part of this year's conference held under the ABZ banner. He contributed to the theoretical foundations of the method and initiated its industrial applications in a variety of fields, in particular programming languages, system architecture, requirements and software (re-)engineering, control systems, protocols, and Web services. In 2007, he received the Humboldt Research Award.

He has been coauthor of several books and over 150 research papers, and organizer of over 30 international conferences, workshops, and schools in logic and computer science.

As one can see, his influence has been broad as well as deep. It is an influence that one sees in all of the notations covered in the ABZ conference, as well as in the iFM event and the various integrations and combinations of formal methods seen there. Neither iFM nor ABZ have been here before, and it is thus especially fitting that we hold such an event in Pisa, where Egon has held a chair for many years.

In addition to contributed papers, the conference program included two tutorials and three keynote speakers. The tutorials were offered by: Eric C.R. Hehner

on Practical Predicative Programming Primer; Joost-Pieter Katoen, Thomas Noll, and Alessandro Cimatti on Safety, Dependability, and Performance Analysis of Extended AADL Models. We are grateful to Egon Böerger, Muffy Calder, and Ian J. Hayes, for accepting our invitations to address the conference.

Each conference, ABZ and iFM, had its own Program Committee Chairs and Program Committees, and we leave it to them to describe their particular conference. We shared invited speakers, so all conference attendees had the opportunity to hear Egon, Muffy, and Ian. We also shared some technical sessions so that all participants could see some of the best technical work from each conference.

We would like to thank the Program Committee Chairs, Diego Latella, CNR/-ISTI, Italy, Helen Treharne, University of Surrey, UK, for IFM 2012; Steve Reeves, University of Waikato, New Zealand, and Elvinia Riccobene, University of Milan, Italy, for ABZ 2012 for their efforts in setting up two high-quality conferences.

We also would like to thank the members of the Organizing Committee as well as several other people whose efforts contributed to making the conference a success and particular thanks go to the Organizing Committee Chair Maurice ter Beek.

April 2012

John Derrick
Stefania Gnesi

Preface to the Volume

The Third International ABZ 2012 Conference was held in Pisa (Italy), during June 18–21, 2012, in conjunction with iFM 2012, the 9th International Conference on Integrated Formal Methods, as a joint event in honor of Egon Börger's 65th birthday. The iFM proceedings appear as a separate LNCS volume, number 7321.

The ABZ conference series is dedicated to the cross-fertilization of five related state-based and machine-based formal methods: Abstract State Machines (ASM), Alloy, B, VDM and Z. They share a common conceptual foundation and are widely used in both academia and industry for the design and analysis of hardware and software systems. The main goal of this conference series is to contribute to the integration of these formal methods, clarifying their commonalities and differences to better understand how to combine different approaches for accomplishing the various tasks in modeling, experimental validation, and mathematical verification of reliable high-quality hardware/software systems.

The edition of ABZ to which this volume is dedicated follows the success of the first ABZ conference held in London (UK) in 2008, where the ASM, B, and Z conference series merged into a single event, and the success of the second ABZ 2010 conference held in Orford (Canada) where the Alloy community joined the event. The novelty of this third international event is the inclusion of the VDM community in the ABZ conference series.

ABZ 2012 received 59 submissions from all five research communities. Although organized as a single event, editorial control of the conference was vested in five separate Program Committees, one for each group: ASM, Alloy, B, VDM, and Z. Each submission was reviewed by at least three Program Committee members, and 33 papers were accepted for publication in this volume and presentation at the conference: 20 long papers covering a broad spectrum of research, from fundamental to applied work, and 13 short papers of work in progress, industrial experience reports, and tool demonstrations.

The ABZ program included two invited talks: one was given by Egon Börger, to whom this event is dedicated and whose paper also appears in the iFM proceedings, and one by Ian J. Hayes from the University of Queensland, Australia.

Organizing and running this event required a lot of effort from several people. We wish to thank all the Program Chairs, all members of the Program Committee, and all the external reviewers for their precise, careful evaluation of the papers and for their availability during the discussion period which considered each paper's acceptance. We wish to express our deepest gratitude to the CNR Institute in Pisa, which supported the event and provided all the necessary organizational support, and we also thank all the sponsors for their financial support.

The conference was managed with EasyChair, which was a valuable support for the submission and review process, and for the preparation of this volume.

A particular special thanks to Egon Börger, master of science and life.

April 2012

Steve Reeves
Elvinia Riccobene

INTECS S.p.A.

Formal Methods Europe

Banca Nazionale del Lavoro S.p.A.

European Association for Theoretical Computer Science Italian Chapter

EATCS Italian Chapter

Conference Organization

General Chairs

John Derrick	University of Sheffield, UK
Stefania Gnesi	ISTI-CNR, Italy

Conference Chairs

Steve Reeves	University of Waikato, New Zealand
Elvinia Riccobene	University of Milan, Italy

Program Chairs

John Fitzgerald (VDM)	Newcastle University, UK
Michael Leuschel (B)	University of Düsseldorf, Germany
Sarfraz Khurshid (Alloy)	University of Texas at Austin, USA
Steve Reeves (Z)	University of Waikato, New Zealand
Elvinia Riccobene (ASM)	University of Milan, Italy

ASM Program Committee

Roozbeh Farahbod	SAP Research, Karlsruhe, Germany
Vincenzo Gervasi	University of Pisa, Italy
Uwe Glässer	Simon Fraser University, Canada
Andreas Prinz	Agder University College, Norway
Alexander Raschke	University of ULM, Germany
Elvinia Riccobene (Chair)	University of Milan, Italy
Patrizia Scandurra	University of Bergamo, Italy
Gerhard Schellhorn	University of Augsburg, Germany
Klaus-Dieter Schewe	SCCH, Austria
Bernard Thalheim	Christian Albrechts University Kiel, Germany
Margus Veanes	Microsoft Research, USA
Kirsten Winter	University of Queensland, Australia

Alloy Program Committee

Juergen Dingel	Queen's University, Canada
Andriy Dunets	Codronic GmbH, Augsburg, Germany
Kathi Fisler	Worcester Polytechnic Institute, USA

Jeremy Jacob	University of York, UK
Sarfraz Khurshid (Chair)	University of Texas at Austin, USA
Daniel Le Berre	Université d'Artois, France
Darko Marinov	University of Illinois, USA
José Oliveira	Minho University, Portugal
Burkhardt Renz	THM, Gießen, Germany
Kevin Sullivan	University of Virginia, USA
Mana Taghdiri	Karlsruhe Institute of Technology, Germany

B Program Committee

Jean-Raymond Abrial	Marseille, France
Yamine Ait Ameur	IRIT-ENSEEIHT, Toulouse, France
David Deharbe	University of Rio Grande do Norte, Brazil
Steve Dunne	University of Teesside, UK
Kerstin Eder	University of Bristol, UK
Marc Frappier	University of Sherbrooke, Canada
Stefan Hallerstede	University of Aarhus, Denmark
Thai Son Hoang	ETH Zürich, Switzerland
Regine Laleau	Univesity of Paris-Est, France
Thierry Lecomte	ClearSy, France
Michael Leuschel (Chair)	University of Düsseldorf, Germany
Christophe Métayer	Systerel, France
Marie-Laure Potet	IMAG Grenoble, France
Ken Robinson	University of New South Wales, Australia
Steve Schneider	University of Surrey, UK
Colin Snook	University of Southampton, UK

VDM Program Committee

Nick Battle	Fujitsu Services, UK
Juan Bicarregui	STFC Rutherford Appleton Laboratory, UK
Dines Bjørner	DTU Informatics, Denmark
John Fitzgerald (Chair)	Newcastle University, UK
Klaus Havelund	Jet Propulsion Laboratory/NASA, USA
Cliff Jones	Newcastle University, UK
Peter Gorm Larsen	Aarhus School of Engineering, Denmark
José Oliveira	Minho University, Portugal
Shin Sahara	SCSK Corporation and Hosei University, Japan
Marcel Verhoef	CHESS BV, The Netherlands

Z Program Committee

Rob Arthan	Lemma 1 Ltd., UK
Eerke Boiten	University of Kent, UK
Jonathan P. Bowen	Museophile Limited, UK
Ana Cavalcanti	University of York, UK
John Derrick	University of Sheffield, UK
Anthony Hall	Independent Consultant
Ian J. Hayes	University of Queensland, Australia
Rob Hierons	Brunel University, UK
Steve Reeves (Chair)	University of Waikato, New Zealand
Thomas Santen	Microsoft Innovation Center, Germany

Tutorial Chair

Jonathan P. Bowen	Museophile Limited, UK

Posters and Tool Demos Chairs

Franco Mazzanti	ISTI-CNR, Italy
Gianluca Trentanni	ISTI-CNR, Italy

Financial Chair

Alessandro Fantechi	University of Florence and ISTI-CNR, Italy

Organizing Chair

Maurice ter Beek	ISTI-CNR, Italy

Additional Reviewers

Paolo Arcaini	Stefan Hallerstede
Vladimir Avram	Dominik Haneberg
Jens Bendisposto	Piper Jackson
Karoly Bosa	Theodorich Kopetzky
Sylvain Boulmé	Felix Kossak
Alcino Cunha	Lukas Ladenberger
Gidon Ernst	Rudolf Ramler
Maria Frade	Ken Robinson
Andreas Fürst	Ove Sörensen
Frédéric Gervais	Bogdan Tofan
Axel Habermaier	Hamed Yaghoubi Shahir

Table of Contents

Invited Talks

ASM Papers

Alloy Papers

B Papers

VDM Papers

Z Papers

ASM Short Papers

B Short Papers

Contribution to a Rigorous Analysis of Web Application Frameworks

Egon Börger, Antonio Cisternino, and Vincenzo Gervasi

Università di Pisa, Dipartimento di Informatica, I-56125 Pisa, Italy
{gervasi,cisterni,boerger}@di.unipi.it

Abstract. We suggest an approach for accurate modeling and analysis of web application frameworks.

1 Introduction

In software engineering the term 'application' traditionally refers to a specific program or process users can invoke on a computer. The emergence of distributed systems and in particular of web applications has significantly changed this meaning of the term. Here functionality is provided by a set of indipendent cooperating modules with a distributed state, in web applications all offering a unified interface to their user—to the point that the user may have no way to distinguish whether a single application or a set of distributed web applications is used. Also recent non-web systems, like mobile apps, follow the same paradigm allowing the state of an application to be persistent and distributed, no longer tied to the traditional notion of operating system process and memory.

There is still no precise general definition or model of what a web application is. What is there is a variety of (often vague and partly incompatible) standards, web service description languages at different levels of abstraction (like BPEL, BPMN, workflow patterns, see [9] for a critical evaluation of the latter two) and difficult to compare techniques, architectures and frameworks offered for implementations of web applications, ranging from CGI (Common Gateway Interface [23]) scripts to PHP (Personal Home Page) and ASP (Application Server Page) applications and to frameworks such as ASP.NET [19] and Java Server Faces (JSF [1]). All of them seem to share that a web application consists of a dynamically changing network of systems that send and receive through the HTTP protocol data to and from other components and provide services of all kinds which are subject to continuous change (as services may become temporarily or permanently unavailable), to dynamic interference with other services (competing for resources, suffering from overload, etc.) and to all sorts of failures and attacks.

The challenge we see is to discover and formulate the pattern underlying such client-server architectures for (programming and executing concurrent distributed) web applications. We want to make their common structural aspects

J. Derrick et al. (Eds.): ABZ 2012, LNCS 7316, pp. 1–20, 2012.

explicit by defining precise high-level (read: code, platform and framework independent) models for the main components of current web application systems such that the major currently existing implementations can be described as refinements of the abstract models. The goal of such a rational reconstruction is to make a rigorous mathematical analysis of web applications possible, including to precisely state and analyze the similarities and differences among existing frameworks, e.g. the similarities between PHP and ASP and the differences between PHP/ASP and JSP/ASP.NET. This has three beneficial consequences: a) it helps web application analysts to better understand different technologies before integrating them to make them cooperate; b) it builds a foundation for content-based certifiability of properties one would like to guarantee for web applications; c) it supports teachers and book authors to provide an accurate organic birds' perspective of a significant area of current computer technology.

For the present state of the art, given the lack of rigorous abstract models of (at least the core components of) web application frameworks, it is still a theoretical challenge to analyze, evaluate and classify web application systems along the lines of fundamental behavioral model properties which can be accurately stated and verified and be instantiated and checked for implementations.

The modeling concepts one needs to work on the challenge become clear if we consider the above mentioned feature all web applications have in common, namely to be an application whose interface is presented to the user via a web browser, whose state is split between a client and a server and where the only interaction between client and server is through the HTTP protocol. This implies that an attempt to abstractly model web application frameworks must define at least the following two major client-server architecture components with their subcomponents and the communication network supporting their interaction:

- the browser with all its subcomponents: launcher, netreader, (html, script, image) parsers, script interpreter, renderer, etc.
- the server with its modules providing runtimes of various programming languages (e.g. PHP, Python [2], ASP, ASP.NET, JSF),
- the asynchronous network which supports the interaction (in particular the communication) between the components.

This calls for a modeling framework with the following features:

- A notion of *agents* which execute each their (possibly dynamically changing) program concurrently, possibly at different sites.
- A notion of *abstract state* covering design and analysis at different levels of abstraction (to cope with heterogeneous data structures of the involved components) and the distributed character of the state of a web application.
- A sufficiently general *refinement method* to controllably link (using validation and/or verification) the different levels of abstraction, specifically to formulate different existing systems as instances of one general model.
- A flexible mechanism to express forms of *non-determinism* which can be restricted by a variety of constraints, e.g. by different degrees of transmission

reliability ranging from completely unreliable (over the internet) to safe and secure (like for components running on one isolated single machine).
- A flexible *environment adaptation mechanism* to uniformly describe web application executions modulo their dependence on run-time contexts.
- A smooth *support for traceable model change* and refinement changes due to changing requirements in the underlying (often de facto) standards.

1.1 Concrete Goals and Results So Far

As a first step towards the goal outlined above we started to model the client-server architecture of a browser interacting with a web server. In [17] the transport and stream levels of an abstract web browser model are defined. To this we add here models for the main components of the context level layer (Sect. 2) which together with the web server model defined in Sect. 3 allow one to describe one complete round of the Request-Reply pattern [18,8] that characterizes browser/server interactions (see Fig. 1).[1] In Sect. 3.1 a high-level functional Request-Reply web server view is defined which is then detailed (by refinement steps) for the two main approaches to module execution:

- the CGI-approach where the server delegates the execution of an external process to another agent (Sect. 3.3),
- the script-approach where the server itself executes script code (Sect. 3.4).

We explain how one can view existing implementations as instantiations of these models.

We use the ASM (Abstract State Machines) method [12] as modeling framework because it offers all the features listed above which are needed for our endeavor[2] and because various ASM models in the literature contribute specifically to the work undertaken here. For example both the browser and the server model use a third group of basic components, namely ScriptInterpreters for various Script languages, which can be specified by an ASM model adopting the method used in [22] to define an interpreter for Java (and reused in [11,15,16] to rigorously define the semantics of C# and the CLR). These models provide a significant part of the infrastructure web applications typically use. For example applets which run inside a browser, or the Tomcat application server [3], are written in Java. Furthermore, the method developed for modeling Java/JVM can be reused to define a model for the JavaScript interpreter (see [14] for some details) corresponding to the ECMAScript standard ECMA-262 [4], a standard that serves as glue to link various technologies together.

In Sect. 4 we list some verification goals we suggest to pursue on the basis of (appropriately completed) precise abstract models of web application framework components, i.e. to rigorously formulate and check (verify or falsify) properties of interest for the models and/or their implementations.

[1] In the Request-Reply pattern of two-way conversations the requestor (one application) sends a request to the provider (another application) and the provider returns a reply to the requestor.

[2] See [10] for the recent definition of a simple flexible *ambient ASM* concept.

The models we define and their properties we discuss come without any completeness claim and are intended to suggest an approach we consider to be promising for future FM research in a core area of computer technology.

2 Modeling Browser Components

Our browser models focus on those parts of the browser behaviour that are most relevant for the deployement and execution of web applications. The models are developed at four layers. The main components of the *transport layer* (expressing the TCP/IP communication via HTTP) and the *stream layer* (describing how information coming from the network is received and interpreted) are defined in [17]. In this section we add models for characteristic components of the *context layer*, which deals with the user interaction with the document represented by the Document Object Model (DOM). Without loss of generality we omit in this paper the *browser layer* where the behaviour of a web browser seen as an application of the host operating system is described. In practice, most web applications are entirely contained in a single browsing context; in fact an important issue in the development of web standards is how to ensure for security reasons that multiple browsing contexts in the same browser are sufficiently isolated from each other (a security property that we leave to future work).

2.1 Browsing Context

A *browsing context* is an environment in which documents are shown to the user, and where interaction with the user occurs. In web browsers, browsing contexts are usually associated with windows or tabs, but certain deprecated HTML structures (namely, frames) also introduce separate browsing contexts.

In our model, a browsing context is characterized primarily by five elements:

- a *document* (i.e. a DOM as described in [17]), which is the currently active document presented to the user;
- a *session history*, which is a navigable stack of documents the user has visited in this browsing context;
- a *window*, which is a designated operating system-dependent area where the Document is presented and where any user interaction takes place;
- a *renderer*, which is a component that produces a user-visible graphical rendering of the current Document (Section 2.2);
- an *event loop*, which is a component that receives and processes in an ordered way the various operating system-supplied events (such as user interaction or timer expiration) that serve as local input to the browser (Section 2.3).

We keep the *window* abstract, as its behaviour can be conveniently hidden by keeping the actual rendering abstract and by assuming that user interaction with the window is handled by the operating system. Thus we deal with events that have been already pre-processed by a window manager. We also omit the rather straightforward modeling of the *session history*.

When STARTing a newly created *B*rowsing *C*ontext k, $DOM(k)$ is initialized by a pre-defined implementation-dependent initial document *initialDOM*; it is usually referred to through the URL `about:blank` and may represent an empty page or a "welcome page" of some sort. Two agents are equipped with programs to execute the RENDERER and the EVENTLOOP for k.

STARTBC(k) =
  let a =**new** *Agent*, b =**new** *Agent* **in**
    $program(a)$:= RENDERER(k)
    $program(b)$:= EVENTLOOP(k)
    $DOM(k)$:= *initialDOM*

The RENDERER and EVENTLOOP macros are specified below.

2.2 Renderer

The RENDERER produces the user interface of the current *DOM* in the (implicit) given window. It is kept abstract by specifying only that it works when it is (a) supposed to perform (at system dependent *RenderingTime*) and (b) allowed to perform because no other agent has a lock on the *DOM* (e.g., while adding new nodes to the DOM during the stream-level loading of an HTML page).

RENDERER(k) =
  if *renderingTime*(k) **and** $\neg$*locked*($DOM(k)$) **then**
    GENERATEUI($DOM(k), k$)

2.3 Event Loop

We assume that *events* are communicated by the host environment (i.e., the specific operating system and UI toolkit of the client machine where the browser is executed) to the browser by means of an *event queue*. These UI events are merged and put in sequential order with other events that are generated in the course of the computation, e.g. DOM manipulation events (fired whenever an operation on the DOM, caused by user actions or by Javascript operations, leads to the execution of a Javascript handler or similar processing) or History traversal events (fired whenever a user operates on the Back and Forward buttons offered by most browsers to navigate through the page stack).

Here we detail the basic mechanism used in (the simplest form of) web applications to prepare a Request to be sent to the server (with the understanding that when a Response is received, it will replace the current page in the same browsing context). HTML *forms* are used to collect related data items, usually entered by the user, and to package them in a single Request. Figure 1 shows when the macros defined below and in [17] are invoked; lifelines represent agents executing a rule. Remember that ASM agents can change their program dynamically (e.g., when RECEIVE becomes HTMLPROC) and that operations by an agent in the same activation, albeit shown in sequence, happen in parallel.

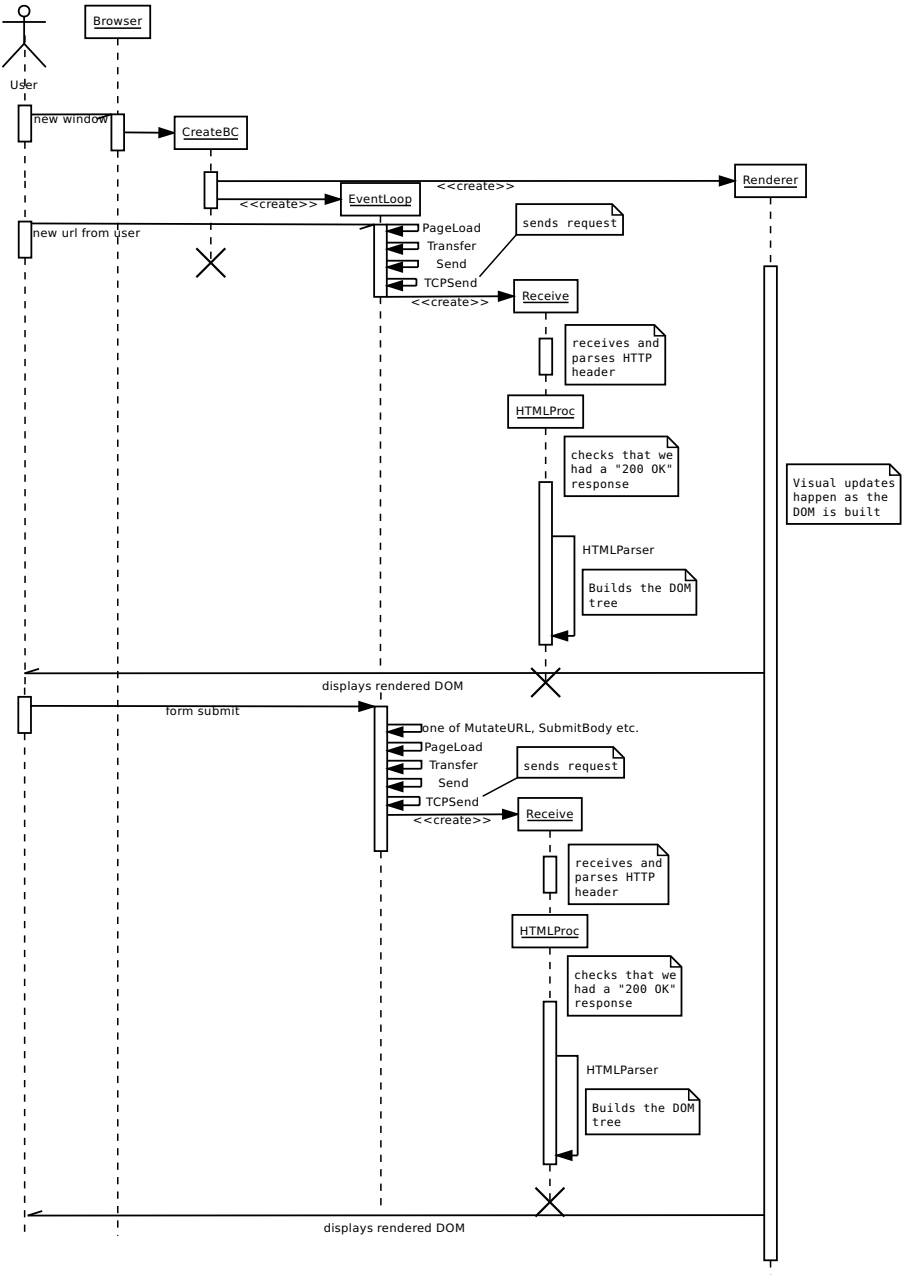


Fig. 1. A diagram depicting the behaviour of our browser model for a user who opens a new window in a browser, manually loads the first page of a web application, interacts locally with a form, and then sends the data back to the server, receiving a new or updated page in response

An HTML form is introduced by a `<FORM>` element in the page. All the input elements[3] that appear in the subtree of the DOM rooted at the `<FORM>` are said to belong to that form. Among the various input elements, there is normally a designated one (whose UI representation is often an appropriately labeled button) tasked with the function of *submitting* a form. This involves collecting all the data elements in the form, encoding them in an appropriate format, and sending them to a destination server through various means. This may include sending the data by email or initiating an FTP transfer, although these possibilities are seldom, if ever, used in contemporary web applications.

It is also of interest to note that submission of a form may be initiated from a script, by invoking the `submit()` method of the form object, and hence happen indipendently from user behaviour. In the following, we will not concern ourselves with the details of how a submit operation has been initiated, but only with the emergence of the submit event in the event queue, whatever its origin.

We model the existence of a separate event queue for each browsing context, which is processed by a dedicated agent created in the StartBC macro above. When an event is extracted from the event queue that indicates that the user has provided a new URL to load (e.g., by typing it in a browser's address bar, or by selecting an entry from a bookmarks list, etc.), the browsing context is navigated to the provided URL by starting an asynchronous transfer (in the normal case, the HTTP Request will be sent to the host mentioned in the URL, and later processing of the Response will replace the DOM displayed in the page).

When an event is extracted from the event queue that indicates a form submission, the form and related parameters are extracted from the event, appropriate encoding of the data is performed based on the action and method attributes as specified in the `<FORM>` node, and finally either the data is sent out (e.g., in the case of a `mailto:` action) or the browsing context is populated with the results returned from a web server identified by the form's action. In normal usage, that will be the same web server hosting the web application that originally sent out the page with the form, thus completing the loop between server and client and realizing the well-known page-navigation paradigm of web applications[4].

As for Renderer, the event loop receives a parameter, k, which identifies the particular instance. The macro PageLoad is defined below.

EventLoop(k) =
  if *eventAvailable*(*eventQueue*(k)) **then**
    let e = *headEvent*(*eventQueue*(k)) **in**
      dequeue e **from** *eventQueue*(k)
      if *isNewUrlFromUser*(e) **then**
        PageLoad(GET, *url*(e), $\langle\rangle$, k)
      elseif *isFormSubmit*(e) **then**

[3] These include elements such as `<INPUT>`, `<SELECT>`, `<OPTION>` etc.

[4] Notice that we are not considering here AJAX applications, where a Request is sent out directly from Javascript code, and the results are returned as raw data to the same script, instead of being used to replace the contents of the page. The general processing for this case is, however, similar to the one we describe here.

```
    let f = formElement(e), data = encodeFormData(f),
        a = action(f), m = method(f), u = resolveUrl(f, a) in
      match (schema(u), m) :
        case (http, GET) : MutateUrl(u, data, k)
        case (http, POST) : SubmitBody(u, data, k)
        case (ftp, GET) : GetAction(u, data, k)
        case (ftp, POST) : GetAction(u, data, k)
        case (javascript, GET) : GetAction(u, data, k)
        case (javascript, POST) : GetAction(u, data, k)
        case (data, GET) : GetAction(u, data, k)
        case (data, POST) : PostAction(u, data, k)
        case (mailto, GET) : MailHead(a, data)
        case (mailto, POST) : MailBody(a, data)
  else
    handle other events
```

We do not further specify here the mail-related variants MailHead and MailBody (although it is interesting to remark that they do not need further access to the browsing context, contrary to most other methods, since no reply is expected from them – and thus their applicability in web applications is close to nil). We also glide over the possibility of using a `https` schema, which however implies the same processing as `http`, with the only additional step of properly encrypting the communication. Given the purposes of this paper we omit a definition of GetAction and PostAction, since they involve URL schemas (namely: `ftp`, `javascript` and `data`) that have not been addressed in the transport layer model in [17]. Thus, below we only refine MutateUrl and SubmitBody together with PageLoad.

The macro MutateUrl consists in synthesizing a new URL from the action and the form data (which are encoded as query parameters in the URL) and in causing the browsing context to navigate to the new URL:

```
MutateUrl(u, data, k) =
  let u' = u · ? · data in PageLoad(GET, u', ⟨⟩, k)
```

The macro SubmitBody differs only in the way the data is encoded in the request, namely not as part of the URL, as above, but as body of the request:

```
SubmitBody(u, data, k) = PageLoad(POST, u, data, k)
```

The macro PageLoad starts an asynchronous Transfer—which is defined in [17]—and (re-)initializes the browsing context and the HTMLProcessor; the latter is also defined in [17] and will handle the Response:

```
PageLoad(m, u, data, k) =
  Transfer(m, u, data, HTMLProc, k)
  htmlParserMode(k) := Parsing
  let d =new Dom in
```

$DOM(k) := d$
$curNode(k) := root(d)$

Notice that while for the sake of brevity we have modeled navigation to the response provided by the server as a direct TRANSFER here, in reality it would require a few additional steps, including: storing the previous document and associated data in the session history, releasing resources used in the original page (e.g., freeing images or stopping plug-ins that were running), etc. While resource management can be conveniently abstracted, handling of history navigation (i.e., the Back, Forward and Reload commands available in most browsers) is a critical component in proving robustness, safety and correctness properties of web applications, and will be addressed in future work.

3 A High-Level WEBSERVER Model

We define here a companion model to the browser model: a high-level model WEBSERVER (Sect. 3.1) with typical refinements for the underlying handler modules, namely for file transfer (Sect. 3.2), CGI (Sect. 3.3) and scripting modules (Sect. 3.4).

To concentrate on the core issues we abstract in this section from the transmission protocol phase during which the connection between client and server is established and rely upon an abstract SEND mechanism; the missing elements to incorporate this phase can be defined as shown in detail for the browser component models in [17].

3.1 Functional Request-Reply Web Server View

In the high-level view the server appears as dispatcher which to handle a request finds and triggers the code (a 'module') the execution of which will provide a response to the request.[5] Thus a high-level web server model can be formulated as an ASM WEBSERVER which in a reactive manner, upon any *request* in its *requestQueue*, will delegate to a new agent (read: a thread we call *request handler*) to handle the EXECution of the request—if the *request* passes the *Security* check and the *requestedModule* is *Available* in and can be loaded by the server.

We succinctly describe checking various kinds of *Property* (here access security, module availability and loadability) by functions (here *checkSecurity*, *findModule loadModule*) whose values are

- either three-digit-values v in an interval $[n00, n99]$, for some $n \in [0, 9]$ as defined for each *Property* of interest in [5, Sect.4.1] to indicate that the *Property* holds or fails to hold (in the latter case of $PropertyFailure(v)$ the value v also indicates the reason for the failure), or

[5] The ASM model for the Virtual Provider (VP) defined in [7] has a similar structure: it receives requests, forwards them to appropriate providers and collects the replies from the providers to return them to the original requestor.

- some different value, like a found requested module, which implicitly also indicates that the checked *Property* holds, e.g. that the requested module is available or could be successfully loaded.

Since in case *PropertyFailure*(v) is true the function value v is assumed to indicate the reason for the failure, the value appears in the *failureReport* the WEBSERVER will SEND to the client. The function *failureReport* abstracts from the details of formatting the response message out of the parameters.

The *requestedModule* depends on the server *env*ironment, the *resourceName* that appears as part of the *request* and the *header*(*request*). For a loaded *module* STARTHANDLER creates a new thread and puts it into its *init*ial state from where the thread will start its program, namely to EXECute the *module*. A loaded *module* is of one of finitely many kinds. For the fundamental CGI and scripting module types we will detail in Sect. 3.3,3.4 what it means to EXECute such a module.

To reflect the functional client/server request/reply view STARTHANDLER appears as atomic action of the WEBSERVER which goes together with deleting the *request* from the *requestQueue*. At the transmission protocol level the latter action becomes closing the connection. The atomicity reflects the fact that once a request has been handled, the server is ready to handle the next request.[6]

```
WEBSERVER =
let request = head(requestQueue)
if request ≠ undef then  // react if there is some request
  let env = env(server, request)
  let s = checkSecurity(request, env)
  if SecurityFailure(s)
  then SEND(failureReport(request, s))
  else
    let requestedModule =
      findModule(env, resourceName(request), header(request))
    if ResourceAvailabilityFailure(requestedModule) then
      SEND(failureReport(request, requestedModule))
    else
      let module = loadModule(requestedModule, env)
      if ModuleLoadabilityFailure(module)
      then SEND(failureReport(request, module))
      else STARTHANDLER(module, request, env)
  CLOSE(request)
where
  SecurityFailure(s) iff s = 403
  ResourceAvailabilityFailure(m) iff m = 503
  ModuleLoadabilityFailure(module) iff module = 500
  STARTHANDLER(module, request, env) =
    let a = new (Agent) // launch a request handler thread
```

[6] The ASM model supports this view due to the reactive character of ASMs.

$program(a) :=$ Exec($module$)($request, env$)
$mode(a) := init$
Close($request$) = Delete($request, requestQueue$)

3.2 Refinement for File Transfer Execution

To start with a simple case we illustrate how the machine Exec($module$) can be detailed to a machine ExecFileTransfer($module$) which handles file transfer *module*s, the earliest form of server module. Such a *module* simply buffers the requested *file* in an output buffer if the *file* is present at the location determined by the path from the $root(env)$ to the $resourceName(request)$. We use a machine TransferDataFromTo which abstracts from the details of the (not at all atomic, but durative) transfer action of the requested file data to the output. The function $requestOutput(request)$ abstractly represents the appropriate socket through which the response data are sent from the server to the requesting browser.[7]

We leave it open what the scheduler does with the request handler when the latter is Deactivated once the file transfer *isFinished*, i.e. when it has been detected (here via TransferDataFromTo) that no more data are to be expected for the transfer.

ExecFileTransfer($module$)($request, env$) =
let $file = makePath(root(env), resourceName(request))$
if $mode($**self**$) = init$ **then**
  if $UndefinedFile(file)$ **then**
    Send($failureReport(request, ErrorCode(UndefinedFile))$)
    Deactivate(**self**) // request handler termination
  else
    Send($successReport(request, OkResponseCode)$)
    $mode($**self**$) := transferData$ // Start to transfer the *file*
  if $mode($**self**$) = transferData$ **then**
    TransferDataFromTo($file, requestOutput(request)$)
  if $isFinished(file)$ **then** Deactivate(**self**)
  where
    $ErrorCode(UndefinedFile) = 404$
    $OkResponseCode = 200$
    Deactivate(**self**) = ($mode($**self**$) := final$)

3.3 Refinement for Common Gateway Module Execution

A Common Gateway Interface (CGI) [23] *module* allows the request handler to pass requests from a client web browser to an (agent which executes an) external application and to return application output to the web browser. There are two main forms of CGI modules, the historically first one (called CGI) and

[7] Again this can be made precise as shown in detail for the browser model in [17].

an optimized one called FastCGI [13]. They differ in the way they introduce agents for external process execution: CGI creates one agent for each request, whereas FastCGI creates one agent and re-uses it for subsequent requests to the same application (though with different parameters).

CGI Module. A CGI *module* sends an error message if the *executable* for the requested process is not defined at the indicated location. Otherwise the requested process execution (by an independent newly created agent a, not by the request handler)[8] is triggered for the appropriate *requestVariables* (also called environment variables containing the request data), like Auth(entication)-Type, Query-String, Path-Info, RemoteAddr (of the requesting browser) and Remote-Host (of the browser's machine), etc.(see [23, Sect.5]) and a positive response is sent to the requesting client. Once the new agent a has been CONNECTed the request handler

- accepts any further *requestInput* stream (read: data stream coming from the browser) as input for the execution of the process by a, namely via the *stdin* stream of the *module*, and
- transmits any output which (via a's processing the *executable*) becomes available on the *module*'s *stdout* stream to the *requestOutput* stream (from where it will be sent to the requesting browser)—as long as there are data on the *requestInput* resp. on the *stdout* stream.

Thus to CONNECT a to (the agent **self** executing) the CGI *module* a channel is established between the *inputStream*(a) and the *module*'s *stdin* stream resp. between the *outputStream*(a) and the *module*'s *stdout* stream[9].

It is usually assumed that the executable *program*(a) agent a gets equipped with eventually disconnects a (from the request handler **self**) so that the predicate *Connected*(a, **self**) becomes false. Then EXEC(*module*) terminates wherefor the request handler is DEACTIVATEd. Nevertheless the agent a even after having been disconnected may continue the execution of the associated *executable* and may not terminate at all, but such a further execution would be unrelated to the computation of the request handler and from the WEBSERVER's point of view yields a garbage process. Even more, no guarantee is given that *program*(a) does disconnect a. In these cases the operating system has to close the connection and/or to kill the process by descheduling its executing agent (e.g. via a time-out). The CGI standard [23] leaves this issue open, but is has to be investigated if one wants to provide some behavioral guarantees for the execution of CGI modules.

[8] Therefore each request triggers a fresh instance of the associated external application program to be executed. This is a possible source for exceeding the workload capacity of the machine where the server runs.

[9] In ASM terms *inputStream*(a) is a monitored and *outputStream*(a) an output location for the *executable*, whereas for the *module stdin* is an output location (whereby the request handler **self** passes input to a for the processing of the *executable*) and *stdout* a monitored location (whereby the request handler **self** receives from a output produced through processing the *executable*.)

```
EXEC(module)(request, env) =
let executable = makePath(root(env), resourceName(request), env)
if mode(self) = init then
  if UndefinedProcess(executable) then
    SEND(failureReport(request, ErrorCode(UndefinedProcess)))
    DEACTIVATE(self)
  else
    let a = new (Agent) // launch a new process instance
    program(a) := executable(processEnv(env, requestVariables(request)))
    CONNECT(a, self)
    SEND(request, OkResponseCode)
    mode(self) := transferData
if mode(self) = transferData then
  if DataAvailable(stdout)
    TRANSFERDATAFROMTO(stdout, requestOutput(request))
  if verb(request) = POST and DataAvailable(requestInput(request))
    then TRANSFERDATAFROMTO(requestInput(request), stdin)
if isDisconnected(a) then DEACTIVATE(self)
where
  ErrorCode(UndefinedProcess) = 404
  OkResponseCode = 200
  isDisconnected(a) = not Connected(a, self)
```

Remark. The server *env*ironment is needed as argument to compute the path information in *makePath*. This is particularly important for the optimized FastCGI version we describe now.

FastCGI Module. Concerning the execution of external processes a FastCGI module has the same function as a CGI module. There are two behavioral differences:

- A FastCGI module creates a new agent for the execution of a process only upon the first invocation of the latter by the request handler. An agent a which has been created to process an *executable* is kept alive once this processing *isFinished* so that the agent can become active again for the next invocation of that *executable*—with the new values for the *requestVariables*. To CONNECT$(a, \mathbf{self})$ now means to link its (local variables for) input resp. output locations, denoted below by $in(a)$, $out(a)$, to corresponding locations of the (request handler **self** executing the) *module* from where resp. to which the data transfer from *requestInput* resp. to *requestOutput* is operated. In particular $in(a)$ is used to pass the parameters *requestVariables*(*request*) of the process to initialize the *executable*.
- It is assumed that the program *program*(a) agent a gets equipped with eventually sets a location *EndOfRequest* for the current *request* to false, namely by updating this location during the TRANSFERDATAFROMCGI action. This makes the request handler terminate.

Thus the CGI structure is refined to the FastCGI module structure as follows:

```
EXEC(module)(request, env) =
let executable = makePath(root(env), resourceName(request), env)
if mode(self) = init then
  if UndefinedProcess(executable) then
    SEND(failureReport(request, ErrorCode(UndefinedProcess)))
    DEACTIVATE(self)
  else
    if thereisno a ∈ Agent with
        program(a) = executable(processEnv(env))
      then
        let a = new (Agent)
        program(a) := executable(processEnv(env))
    mode(self) := connect
if mode(self) = connect then
  let a = ιx(x ∈ Agent and
    program(a) = executable(processEnv(env)))
  CONNECT(a, self)
  INITIALIZE(program(a))
  mode(self) := transferData
if mode(self) = transferData then
  let reqin = requestInput(request), reqout = requestOutput(request)
  if DataAvailable(out(a))
    TRANSFERDATAFROMCGI(out(a), reqout, EndOfRequest(request))
  if verb(request) = POST and DataAvailable(reqin) then
    TRANSFERDATATOCGI(reqin, in(a))
if EndOfRequest(request) then DEACTIVATE(self)
where
  ErrorCode(UndefinedProcess) = 404
  INITIALIZE(program(a)) =
    PASSPARAMS(requestVariables(request), in(a))
    EndOfRequest(request) := false
```

TRANSFERDATATOCGI implies an encapsulation of the to be transmitted content into messages which carry either data or control information; inversely TRANSFERDATAFROMCGI implies a decoding of this encapsulation.

3.4 Refinement for Scripting Module EXECution

Scripting modules like ASP, PHP, JSP all provide dynamic web page facilities by allowing the server to run (directly through its request handler) dynamically provided code. We define here a scheme which makes the common structure of such scripting modules explicit.

As for CGI modules first the file for the to be executed code is searched at the place indicated by the *resourceName* of the *request*, starting at the *root* of the

server *env*ironment. If the file is defined, the code is executed not by an independent agent as for CGI modules, but directly by the request handler which uses as program the ScriptInterpreter. For the state management accross different server invocations by a series of requests from the same client the uniquely determined *sessionID* (associated to the *request* under the given *env*ironment) and the corresponding session and application (if any) have to be computed. The computation of session and application comprises that a new session resp. application is created in case none is defined yet in the server *env*ironment for the *sessionID* resp. *applicationName* of the *request*.[10] Furthermore the syntax conversion of the *script* file from quotation to full script code (denoted here by a machine QuoteToScript which is refined below for ASP, PHP and JSP) has to be performed and the corresponding host objects have to be created to be passed as parameters to the ScriptInterpreter call.

The functions involved to ComputeSession and to ComputeApplication, which allow the server to track state information between different requests of a same client, depend on the *module*, namely *sessionID*, *makeSession* (and therefore *session*), *applicationName*, *makeApplication* (and therefore *application*). Similarly for the functions involved to ComputeInterpreterObjects. We express this using the **amb** notation as defined in [10].

```
Exec(module)(request, env) =
let script = makePath(root(env), resourceName(request))
amb module in  // NB: use of module sensitive functions
  if mode(self) = init then
    if script = ErrorCode(UndefinedScript) then
      Send(failureReport(request, ErrorCode(UndefinedScript)))
      Deactivate(self)
    else
      let id =  sessionID(request, env)
        ComputeSession(id, request, env)
      let applName =  applicationName(resourceName(request))
        ComputeApplication(applName, request, env)
      scriptCode(request) ← QuoteToScript(script, env)[11]
      mode(self) := compInterprObjs
  if mode(self) = compInterprObjs then
    ComputeInterpreterObjects(request, id, applName)
    program(self) :=
      ScriptInterpreter(scriptCode(request), InterpreterObjects))
  where
    ErrorCode(UndefinedScript) = 404
    ComputeSession(id, request, env) =
      if session(id) = undef then
```

[10] Typical refinements of the *sessionID* function also contain specific security policies we necessarily have to abstract from in this high-level description.

[11] The definition of ASMs with return value supporting the notation $l \leftarrow M(x)$ is taken from [12, Def.4.1.7.].

```
        session(id) := makeSession(request, env, id)
    ComputeApplication(applName, request, env) =
      if application(applName) = undef then
        application(applName) := makeApplication(request, env, applName)
    ComputeInterpreterObjects(request, id, applName) =
      reqObj(request) := makeRequestHostObj(request)
      responseObj(request) := makeResponseHostObj(request)
      sessionObj(request) := makeSessionHostObj(session(id))
      applObj(request) := makeApplicationHostObj(application(applName))
      serverObj(request) := makeServerHostObj(request, env)
    InterpreterObjects =
      [reqObj(request), responseObj(request),
        sessionObj(request), applObj(request), serverObj(request)]
```

ASP/PHP/JSP Module. ASP, PHP and JSP modules are instances of the scripting module scheme described above. In fact their Exec(*module*) is defined as for the scripting scheme but each with a specific way to produce dynamic webpages, in particular with a specific computation of QuoteToScript, as we are going to describe below.

Also the following auxiliary functions and the called ScriptInterpreter are specific (as indicated by an index ASP, PHP, JSP) though not furthermore detailed here:

- The *make* ... *HostObj* functions are specialized to *make* ... $HostObj_{index}$ functions for each $index \in \{ASP, PHP, JSP\}$.
- ScriptInterpreter becomes $\text{ScriptInterpreter}_{index}$ for any *index* out of ASP, PHP, JSP.

See [14] for explanations how to construct an ASM model of the JavaScript interpreter as described in [4].

A PHP module acts as a filter: it takes input from a file or stream containing text or special PHP instructions and via their $\text{ScriptInterpreter}_{PHP}$ interpretation outputs another data stream for display.

ASP modules choose the appropriate interpreter for the computed *scriptCode* (so-called *active scripting*). Examples of the type of script code are JavaScript, Visual Basic and Perl.

Thus for ASP the definition of $\text{ScriptInterpreter}_{ASP}$ has the following form:

```
ScriptInterpreter_ASP(scriptCode, InterprObjs) =
  let scriptType = type(scriptCode)
  ScriptInterpreter_scriptType(scriptCode, InterprObjs)
```

The value of *scriptCode*(*request*) is defined as the **result** computed by a machine QuoteToScript for a *script* argument. For the original version of PHP, to mention one early example, this machine simply computed a syntax transformation *transform*(*script*). Later versions introduced some optimization. At the

first invocation of QuoteToScript(*script*)—i.e. when the syntactical transformation of (the code text recorded at) *script* has not yet been *compiled*—or upon later invocations for a *script* (with code text) changed since the last compilation of *transform*(*script*), due to some code text replacement stored at *script* that is out of the control of the web werver, the target bytecode is *compiled* and *timeStamp*ed, using a *compile*r which can be specified using the techniques explained for Java2JVM compilation in [22]. At later invocations of the same *script* the already available *compiled*(*transform*(*script*)) bytecode is taken as *scriptCode* instead of recompiling again. Since the value of the code text located at *script* is not controlled by the web server, the function *timeStamp*(*script*) appears in this model as a monitored function.

scriptCode(*request*) ← QuoteToScript(*script*, *env*)
where
  QuoteToScript(*script*) =
    let s = *transform*(*script*)
    if *compiled*(s) = **undef or**
      timeStamp(*lastCompiled*(*script*)) ≤ *timeStamp*(*script*)
    then
      compiled(s) := *compile*(s)
      result:= *compile*(s)
      timeStamp(*lastCompiled*(*script*)) := *now*
      type(*compile*(s)) := *typeOf*(*script*, *env*)
    else result:= *compiled*(s)

For ASP and PHP the QuoteToScript machine describes an optional optimization[12] that cannot be observed from outside. For ASP the machine has the additional update for the *type* of the computed **result** (namely the *scriptCode*) that uses a syntax function *typeOf* which typically yields a directive, e.g.

$$< \%@Language = \text{“}JScript\text{”} \% >$$

or a default value.

The type of the *scriptCode* depends on the *script* and on the *env*ironment; for example the *env*ironment typically defines a default type for the case that nothing else is specified.

For JSP no syntax translation is required (formally the *transform* function is the identity function) because *scriptCode* is a class file (Servlet which comes with a certain number of fixed interfaces like `doPost()`, `doGet()`, etc.) so that the operations are performed by a JVM. This permits to embed predefined actions (implemented by Java code which can also be included from some predefined file via appropriate JSP directives) into static content. Here the machine QuoteToScript is mandatory because different invocations of the same *scriptCode* can communicate with each other via the values of static class variables.

[12] It is an ASM refinement of the non-optimized original PHP version.

JSF/ASP.NET Modules. It seems that a detailed high-level description of EXEC(*module*) for the *module*s as offered by the Java Server Faces (JSF [1]) and Active Server Pages (ASP.NET [19]) frameworks can be obtained as a refinement of the ASM defined above for the execution of scripting modules. As mentioned above PHP, ASP and JSP use a character based approach in which the script outputs characters (either explicitly through the Response object or implicitly by using the special notation converted by QUOTETOSCRIPT). The JSF and ASP.NET frameworks use their virtual-machine based environment (JVM resp. CLR) to provide more flexible ways for the SCRIPTINTERPRETER to write on the response stream (e.g. in ASP.NET based on the Windows environment) and to define a server-side event and state management model that relieves the programmer from having to explicitly deal with the state of a web page made up by several components. The programming model offered by these environments provides a sort of DOM tree where each node upon being visited is asked for the data to be sent as part of the response so that the programmer has the impression of manipulating objects rather than generating text of a Web page. For example, a request handled by the ASP.NET module triggers a complex lifecycle[13] which allows the programmer to manipulate a tree of components each of which has its own state, in part stored inside the web page (in the form of a hidden field) and in part put by the application into the session state. We are currently working on modeling these features as refinements of the ASM model for scripting module execution.

4 The Challenge of Accurate Analysis

Once sufficiently rich rigorous abstract web application models have been defined they can be used to accurately define properties of interest one would like to prove or falsify for the models via proofs or counterexamples which are preserved by correct refinements for existing implementations. This is by no means an easy task. For an illustrative example we can refer to [22] where in terms of rigorous models for Java, the JVM and a compiler Java2JVM the mere mathematically precise formulation of the compiler correctness property stated in Theorem 14.1.1. (p.177-178) needs 10 pages, the entire section 14.1.[14] A formulation in terms of some logic language understood by a theorem prover (e.g. in the language of KIV which has been used for various mechanical verifications of properties of ASMs [20,21] or in Event-B [6]) is still harder and will be considerably longer, as characteristic for formalizations.

We list here some properties of web applications we suggest to precisely formulate and prove or disprove in terms of abstract web application models.

A first group consists of correctness properties for the crucial session and state management:

[13] See `http://msdn.microsoft.com/en-us/library/ms178472.aspx`

[14] In comparison the proof occupies 24 pages, the rest of chapter 14.

- Session management refers to the ability of an application to maintain the status of the interaction with a particular browser. A typical property is that session state is not corrupted by user actions like hitting the *Back/Forward* buttons or navigating away from the page and then coming back.
- State management is about the virtual state of the application, which is usually distributed among multiple components on both client and server side, with parts of the state 'embedded' into the local state of several programs, and often also replicated entirely or partially. Typical desirable properties are that at significant time instants replicated parts of the state
  - are consistent, that is they are allowed to be out-of-sync at times and consistence is considered up to appropriate abstraction functions,
  - are equivalent between the client-side and the server-side of the state,
  - can be reconstructed, e.g. when the client can change and its state must be persisted to another client (for example from desktop to mobile).

A second group concerns robustness e.g. upon loss of a session or client and server state going out-of-sync, security and liveness.

A third group consists of what we consider to be the most challenging properties which are also of greatest interest to the users, namely application correctness properties. These properties are about the dependence of the intended application-focussed behavior of web applications on the programming and execution infrastructure—on the used browser, web server, net infrastructure (e.g. firewall, router, DNS), connection, plug-ins, etc. Such components are based on their own (not necessarily compatible) standards and therefore may influence the desired application behavior in unexpected ways. This makes their rigorous high-level description mandatory for a precise analysis. An outstanding class of such application-group-specific properties is about application integration where common services are offered on an application-independent basis (e.g. authentication or electronic payment services). We see such investigations as a first step towards defining objective content-based criteria for the reliability of web application software and for building reliable web applications, read: web applications whose properties of interest can be certifiably guaranteed—by theorem proving or model checking or testing or combinations of these activities—to hold under precisely formulated boundary conditions.

Acknowledgement. This paper is published in the two Proceedings volumes of the joint iFM2012 and ABZ2012 Conference held in Pisa (Springer LNCS 7321 and 7316).

References

1. Java Server Faces, `http://www.jcp.org/en/jsr/detail?id=314`
2. Python, `http://www.python.org/`
3. Tomcat, `http://tomcat.apache.org/`
4. ECMAScript language specification. Standard ECMA-262, Edition 5.1 (June 2011), `http://www.ecma-international.org/publications/standards/Ecma-262.html`

5. HTTP1.1 part 2 message semantics, www.ietf.org (cosulted February 2012)
6. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
7. Altenhofen, M., Börger, E., Friesen, A., Lemcke, J.: A high-level specification for virtual providers. IJBPIM 1(4), 267–278 (2006)
8. Barros, A., Börger, E.: A Compositional Framework for Service Interaction Patterns and Interaction Flows. In: Lau, K.-K., Banach, R. (eds.) ICFEM 2005. LNCS, vol. 3785, pp. 5–35. Springer, Heidelberg (2005)
9. Börger, E.: Approaches to modeling business processes. A critical analysis of BPMN, workflow patterns and YAWL. JSSM, 1–14 (2011), doi:10.1007/s10270-011-0214-z
10. Börger, E., Cisternino, A., Gervasi, V.: Ambient Abstract State Machines with applications. JCSS 78(3), 939–959 (2012)
11. Börger, E., Fruja, G., Gervasi, V., Stärk, R.: A high-level modular definition of the semantics of C#. Theoretical Computer Science 336(2-3), 235–284 (2005)
12. Börger, E., Stärk, R.F.: Abstract State Machines. A Method for High-Level System Design and Analysis. Springer (2003)
13. Brown, M. R.: Fast CGI specification (April 1996), http://www.fastcgi.com/
14. Dittamo, C., Gervasi, V., Börger, E., Cisternino, A.: A formal specification of the semantics of ECMAScript. In: VSTTE 2010, Edinburgh (2010) Poster session
15. Fruja, N.G.: Towards proving type safety of.NET CIL. SCP 72(3), 176–219 (2008)
16. Fruja, N.G., Börger, E.: Modeling the.NET CLR Exception Handling Mechanism for a Mathematical Analysis. Journal of Object Technology 5(3), 5–34 (2006)
17. Gervasi, V.: An ASM Model of Concurrency in a Web Browser. In: Derrick, J., et al. (eds.) ABZ 2012. LNCS, pp. 79–93. Springer, Heidelberg (2012)
18. Hohpe, G., Woolf, B.: Enterprise Integration Patterns: Designing, Building and Deploying Messaging Solutions. Addison-Wesley Longman Publishing (2003)
19. Microsoft. ASP.NET, http://www.asp.net
20. Schellhorn, G., Ahrendt, W.: The WAM case study: Verifying compiler correctness for Prolog with KIV. In: Bibel, W., Schmitt, P. (eds.) Automated Deduction – A Basis for Applications, vol. III, pp. 165–194 (1998)
21. Schellhorn, G., Grandy, H., Haneberg, D., Reif, W.: The Mondex Challenge: Machine Checked Proofs for an Electronic Purse. In: Misra, J., Nipkow, T., Karakostas, G. (eds.) FM 2006. LNCS, vol. 4085, pp. 16–31. Springer, Heidelberg (2006)
22. Stärk, R.F., Schmid, J., Börger, E.: Java and the Java Virtual Machine: Definition, Verification, Validation. Springer (2001)
23. W3C. CGI: Common Gateway Interface, http://www.w3.org/CGI/

Integrated Operational Semantics: Small-Step, Big-Step and Multi-step

Ian J. Hayes[1] and Robert J. Colvin[2]

[1] School of Information Technology and Electrical Engineering, The University of Queensland
[2] Queensland Brain Institute, The University of Queensland

Abstract. Plotkin's structural operational semantics provides a tried and tested method for defining the semantics of a programming language via sets of rules that define valid transitions between program configurations. Mosses' modular structural operational semantics (MSOS) recasts the approach by making use of rules consisting of labelled transitions, allowing a more modular approach to defining language semantics. MSOS can be adapted by using "syntactic" labels that allow local variables and aliasing to be defined without augmenting the semantics with environments and locations. The syntactic labels allow both state-based constructs of imperative languages and event-based constructs of process algebras to the specified in an *integrated* manner.

To illustrate the integrated approach we compare its rules with Plotkin's original rules for both small-step and big-step operational semantics. One issue that arises is that defining concurrency requires the use of a small-step approach to handle interleaving, while defining a specification command requires a big-step approach. The integrated approach can be generalised to use a sequence of (small) steps as a label; we call this a *multi-step* operational semantics. This approach allows both concurrency and *non-atomic* specification commands to be defined.

1 Introduction

Operational semantics for programming languages is presented in a number of ways:

- Plotkin gives a structural operational semantics of imperative language constructs using rules defining relations between configurations [13], and
- process algebras, like CCS [8] and CSP [6], use labelled transition systems.

In order to specify a language, CSP_σ [3], with both process algebra constructs (concurrency, events, ...) and imperative programming constructs (state, assignment, ...), we made use of an *integrated* approach involving the use of labelled transitions to handle both aspects [5]. The process algebra constructs use standard event-labelled transitions (like those used for CCS or CSP), while the state-based constructs are handled using transitions labelled with simple state tests and updates (described in detail below). In his Modular Structural Operational Semantics (MSOS) Mosses [11] also uses labels on transitions to handle state, however the form of the labels differs from that used here.

J. Derrick et al. (Eds.): ABZ 2012, LNCS 7316, pp. 21–35, 2012.

Plotkin also gives both

- a big-step semantics in which, for example, the semantics of an expression is given in terms of a relation between a configuration consisting of an (expression, state)-pair representing the expression to be evaluated and the state in which it is to be evaluated, and a configuration consisting of a (value, state)-pair representing the final value of the expression and the final state after evaluation (to allow for side-effects), and
- a small-step semantics in which, for example, the semantics of an expression is given in terms of a relation between configurations consisting of (expression-state)-pairs, which define (atomic) steps in its evaluation.

In Plotkin's approach the small-step semantics is needed to define concurrency because a big-step semantics cannot handle the interleaving of the individual atomic steps of two concurrent processes.

When dealing with end-to-end specifications, like Back's nondeterministic assignment [2] or Morgan's specification statement [9], one only has an overall relation between the initial and final states. If the specification statement is *atomic*, it can be handled straightforwardly by a big-step semantics but not a small-step semantics. However, neither approach handles *non-atomic* specification statements. In order to allow both concurrency and *non-atomic* specifications in the one framework, the integrated style can be adapted to a multi-step operational semantics that supports both. The multi-step operational semantics is a generalisation of the small-step semantics that makes use of transitions labelled with a sequence of small-step labels.

In this paper we first overview the integrated approach to small-step operational semantics focusing on how state-based constructs can be handled using labeled transitions. Comparisons are made between the small-step, big-step and multi-step approaches; this shows how the latter is a generalisation of both small-step and big-step, in that it handles concurrency (where small steps are traditionally needed) and non-atomic specification commands (which neither the small-step or big-step approaches handle).

Sections 2 and 3 compare the operational semantics of expressions and commands, respectively, in the Plotkin and integrated styles. Section 4 compares local state (local variables) in the two styles. Sections 5 and 6 consider control structures and concurrency in the integrated style. The semantics used in Sections 2–6 is a small-step operational semantics. Section 7 compares the approaches for the big-step semantics and Section 8 introduces and compares multi-step semantics. The syntactic form of the labels greatly simplifies the semantics of aliasing (e.g., for call-by-reference parameters to procedures); aliasing is treated in Section 9.

2 Expressions

The following naming conventions are used: variables are denoted by $\mathsf{x}, \mathsf{y}, \mathsf{z}$; constants by κ; expressions by e; states by σ; and labels by ℓ. The abstract syntax of expressions follows.

$$\mathsf{e} ::= \kappa \mid \mathsf{x} \mid \mathsf{e}_0 \leq \mathsf{e}_1 \mid \mathsf{e}_0 + \mathsf{e}_1 \mid \ldots$$

The semantics of expressions is given in both the Plotkin style and in the integrated style in order to allow comparison of the styles. The initial focus is on small-step semantics. In the Plotkin style a state, σ, is represented by a total function mapping identifiers, *Ident*, to values, *Val*.

Plotkin-style operational semantics defines a relation over configurations consisting of pairs of expression and state (e, σ) in terms of transitions representing a (small) step of computation.

$$(\mathsf{e}, \sigma) \longrightarrow (\mathsf{e}', \sigma')$$

Integrated-style operational semantics defines a labelled transition system

$$\mathsf{e} \xrightarrow{\ell} \mathsf{e}'$$

in which the labels are either

- $\mathsf{x} = \kappa$ representing a state test, e.g., $\mathsf{x} = 2$ or $\mathsf{y} = 3$,
- $\mathsf{x} := \kappa$ representing a state update, e.g., $\mathsf{x} := 3$ or $\mathsf{y} := 2$, or
- τ representing a hidden (or internal) action.

Labels are restricted so that the left side is an identifier and the right side is a constant (not an expression). For expressions we do not make use of the state update label because the expressions considered here do not have side effects.

Figure 1 gives the expression evaluation rules in both Plotkin's original style and in the integrated style of operational semantics. The Plotkin-style Rule 2.1 (P-Variable) can be applied to variables x and y in state $\{\mathsf{x} \mapsto 2, \mathsf{y} \mapsto 3\}$ as follows.

$$(\mathsf{x}, \{\mathsf{x} \mapsto 2, \mathsf{y} \mapsto 3\}) \longrightarrow (2, \{\mathsf{x} \mapsto 2, \mathsf{y} \mapsto 3\})$$
$$(\mathsf{y}, \{\mathsf{x} \mapsto 2, \mathsf{y} \mapsto 3\}) \longrightarrow (3, \{\mathsf{x} \mapsto 2, \mathsf{y} \mapsto 3\})$$

In the integrated style the equivalent rule is Rule 2.2 (I-Variable) and the equivalent applications to x and y are the following two transitions.

$$\mathsf{x} \xrightarrow{\mathsf{x}=2} 2 \qquad \mathsf{y} \xrightarrow{\mathsf{y}=3} 3$$

The first of these two transitions can be read as expression x evaluates to 2 in any context in which x is 2. In the integrated style, the role of the (contextual) state is deferred to a separate set of rules covered in Section 4.

The rules for evaluating a binary expression cover the cases of evaluating its left and right operands, plus the case if both operands have been fully evaluated to values. The following is an instance of the Plotkin-style Rule 2.3 (P-Binary-Left) for the expression $\mathsf{x} \leq \mathsf{y}$ in state $\{\mathsf{x} \mapsto 2, \mathsf{y} \mapsto 3\}$,

$$\frac{(\mathsf{x}, \{\mathsf{x} \mapsto 2, \mathsf{y} \mapsto 3\}) \longrightarrow (2, \{\mathsf{x} \mapsto 2, \mathsf{y} \mapsto 3\})}{(\mathsf{x} \leq \mathsf{y}, \{\mathsf{x} \mapsto 2, \mathsf{y} \mapsto 3\}) \longrightarrow (2 \leq \mathsf{y}, \{\mathsf{x} \mapsto 2, \mathsf{y} \mapsto 3\})} \tag{1}$$

and the following is an instance of Rule 2.5 (P-Binary-Right).

Plotkin-style semantics	Integrated-style semantics
Rule 2.1 P-Variable $\dfrac{\sigma(\mathsf{x}) = \kappa}{(\mathsf{x}, \sigma) \longrightarrow (\kappa, \sigma)}$	Rule 2.2 I-Variable $\mathsf{x} \xrightarrow{\mathsf{x}=\kappa} \kappa$
Rule 2.3 P-Binary-Left $\dfrac{(\mathsf{e}_0, \sigma) \longrightarrow (\mathsf{e}_0', \sigma')}{(\mathsf{e}_0 \leq \mathsf{e}_1, \sigma) \longrightarrow (\mathsf{e}_0' \leq \mathsf{e}_1, \sigma')}$	Rule 2.4 I-Binary-Left $\dfrac{\mathsf{e}_0 \xrightarrow{\ell} \mathsf{e}_0'}{\mathsf{e}_0 \leq \mathsf{e}_1 \xrightarrow{\ell} \mathsf{e}_0' \leq \mathsf{e}_1}$
Rule 2.5 P-Binary-Right $\dfrac{(\mathsf{e}_1, \sigma) \longrightarrow (\mathsf{e}_1', \sigma')}{(\kappa_0 \leq \mathsf{e}_1, \sigma) \longrightarrow (\kappa_0 \leq \mathsf{e}_1', \sigma')}$	Rule 2.6 I-Binary-Right $\dfrac{\mathsf{e}_1 \xrightarrow{\ell} \mathsf{e}_1'}{\kappa_0 \leq \mathsf{e}_1 \xrightarrow{\ell} \kappa_0 \leq \mathsf{e}_1'}$
Rule 2.7 P-Binary-Final $\dfrac{\kappa = (\kappa_0 \leq \kappa_1)}{(\kappa_0 \leq \kappa_1, \sigma) \longrightarrow (\kappa, \sigma)}$	Rule 2.8 I-Binary-Final $\dfrac{\kappa = (\kappa_0 \leq \kappa_1)}{\kappa_0 \leq \kappa_1 \xrightarrow{\tau} \kappa}$

Fig. 1. Expression evaluation rules

$$\frac{(\mathsf{y}, \{\mathsf{x} \mapsto 2, \mathsf{y} \mapsto 3\}) \longrightarrow (3, \{\mathsf{x} \mapsto 2, \mathsf{y} \mapsto 3\})}{(2 \leq \mathsf{y}, \{\mathsf{x} \mapsto 2, \mathsf{y} \mapsto 3\}) \longrightarrow (2 \leq 3, \{\mathsf{x} \mapsto 2, \mathsf{y} \mapsto 3\})} \tag{2}$$

The corresponding instances of the rules in the integrated style, i.e., Rule 2.4 (I-Binary-Left) and Rule 2.6 (I-Binary-Right), follow; again state is treated separately using the rules from Section 4.

$$\frac{\mathsf{x} \xrightarrow{\mathsf{x}=2} 2}{\mathsf{x} \leq \mathsf{y} \xrightarrow{\mathsf{x}=2} 2 \leq \mathsf{y}} \qquad \frac{\mathsf{y} \xrightarrow{\mathsf{y}=3} 3}{2 \leq \mathsf{y} \xrightarrow{\mathsf{y}=3} 2 \leq 3} \tag{3}$$

When both operands of a binary operator have been evaluated to a value, the Plotkin-style rule Rule 2.7 (P-Binary-Final) can be applied, for example,

$$\frac{\text{true} = (2 \leq 3)}{(2 \leq 3, \{\mathsf{x} \mapsto 2, \mathsf{y} \mapsto 3\}) \longrightarrow (\text{true}, \{\mathsf{x} \mapsto 2, \mathsf{y} \mapsto 3\})} \tag{4}$$

and the equivalent integrated rule Rule 2.8 (I-Binary-Final) is applied as follows; in this case there is no dependence upon the state.

$$\frac{\text{true} = (2 \leq 3)}{2 \leq 3 \xrightarrow{\tau} \text{true}} \tag{5}$$

The combination of instances (1), (2) and (4) gives the evaluation of $\mathsf{x} \leq \mathsf{y}$ in Plotkin's style and the two steps in (3) plus step (5) give the corresponding evaluation in the integrated style.

Plotkin-style semantics	Integrated-style semantics
Rule 3.1 P-Assign-Step $\dfrac{(\mathsf{e},\sigma) \longrightarrow (\mathsf{e}',\sigma')}{(\mathsf{x} := \mathsf{e},\sigma) \longrightarrow (\mathsf{x} := \mathsf{e}',\sigma')}$	Rule 3.2 I-Assign-Step $\dfrac{\mathsf{e} \xrightarrow{\ell} \mathsf{e}'}{\mathsf{x} := \mathsf{e} \xrightarrow{\ell} \mathsf{x} := \mathsf{e}'}$
Rule 3.3 P-Assign-Final $(\mathsf{x} := \kappa,\sigma) \longrightarrow (\mathbf{nil},\sigma[\mathsf{x} \mapsto \kappa])$	Rule 3.4 I-Assign-Final $\mathsf{x} := \kappa \xrightarrow{x:=\kappa} \mathbf{nil}$
Rule 3.5 P-Sequential-Step $\dfrac{(\mathsf{c}_0,\sigma) \longrightarrow (\mathsf{c}_0',\sigma')}{(\mathsf{c}_0\ ;\ \mathsf{c}_1,\sigma) \longrightarrow (\mathsf{c}_0'\ ;\ \mathsf{c}_1,\sigma')}$	Rule 3.6 I-Sequential-Step $\dfrac{\mathsf{c}_0 \xrightarrow{\ell} \mathsf{c}_0'}{\mathsf{c}_0\ ;\ \mathsf{c}_1 \xrightarrow{\ell} \mathsf{c}_0'\ ;\ \mathsf{c}_1}$
Rule 3.7 P-Sequential-Final $(\mathbf{nil}\ ;\ \mathsf{c}_1,\sigma) \longrightarrow (\mathsf{c}_1,\sigma)$	Rule 3.8 I-Sequential-Final $\mathbf{nil}\ ;\ \mathsf{c}_1 \xrightarrow{\tau} \mathsf{c}_1$

Fig. 2. Semantics of basic commands

Relating the Plotkin and Integrated Styles

To show how the two styles of operational semantics are related, we first show how the labels in the integrated style can be interpreted as total binary relations on states. We define semantics brackets $[\![\_]\!]$ which transform a label into a binary relation on states as follows.

$$(\sigma,\sigma') \in [\![\mathsf{x} = \kappa]\!] \quad \Leftrightarrow \quad \sigma(\mathsf{x}) = \kappa \wedge \sigma = \sigma' \tag{6}$$

$$(\sigma,\sigma') \in [\![\mathsf{x} := \kappa]\!] \quad \Leftrightarrow \quad \sigma' = \sigma[\mathsf{x} \mapsto \kappa] \tag{7}$$

$$(\sigma,\sigma') \in [\![\tau]\!] \quad \Leftrightarrow \quad \sigma = \sigma' \tag{8}$$

A transition in the integrated style of the form $\mathsf{e} \xrightarrow{\ell} \mathsf{e}'$ corresponds to the following rule in the Plotkin style.

$$\frac{(\sigma,\sigma') \in [\![\ell]\!]}{(\mathsf{e},\sigma) \longrightarrow (\mathsf{e}',\sigma')}$$

For example, the transition $\mathsf{x} \xrightarrow{\mathsf{x}=\kappa} \kappa$ corresponds to

$$\frac{\sigma(\mathsf{x}) = \kappa \wedge \sigma = \sigma'}{(\mathsf{x},\sigma) \longrightarrow (\kappa,\sigma')}$$

which is equivalent to Rule 2.1 (P-Variable).

3 Commands

The following additional naming conventions are used: c denotes a command (statement), and b denotes a boolean expression. The abstract syntax of commands follows.

$$
\begin{aligned}
&(\mathsf{t} := \mathsf{x} \,;\, \mathsf{x} := \mathsf{y} \,;\, \mathsf{y} := \mathsf{t}, \{\mathsf{x} \mapsto 2, \mathsf{y} \mapsto 3, \mathsf{t} \mapsto 0\}) \\
\longrightarrow\ &(\mathsf{t} := 2 \,;\, \mathsf{x} := \mathsf{y} \,;\, \mathsf{y} := \mathsf{t}, \{\mathsf{x} \mapsto 2, \mathsf{y} \mapsto 3, \mathsf{t} \mapsto 0\}) \\
\longrightarrow\ &(\mathbf{nil} \,;\, \mathsf{x} := \mathsf{y} \,;\, \mathsf{y} := \mathsf{t}, \{\mathsf{x} \mapsto 2, \mathsf{y} \mapsto 3, \mathsf{t} \mapsto 2\}) \\
\longrightarrow\ &(\mathsf{x} := \mathsf{y} \,;\, \mathsf{y} := \mathsf{t}, \{\mathsf{x} \mapsto 2, \mathsf{y} \mapsto 3, \mathsf{t} \mapsto 2\}) \\
\longrightarrow\ &(\mathsf{x} := 3 \,;\, \mathsf{y} := \mathsf{t}, \{\mathsf{x} \mapsto 2, \mathsf{y} \mapsto 3, \mathsf{t} \mapsto 2\}) \\
\longrightarrow\ &(\mathbf{nil} \,;\, \mathsf{y} := \mathsf{t}, \{\mathsf{x} \mapsto 3, \mathsf{y} \mapsto 3, \mathsf{t} \mapsto 2\}) \\
\longrightarrow\ &(\mathsf{y} := \mathsf{t}, \{\mathsf{x} \mapsto 3, \mathsf{y} \mapsto 3, \mathsf{t} \mapsto 2\}) \\
\longrightarrow\ &(\mathsf{y} := 2, \{\mathsf{x} \mapsto 3, \mathsf{y} \mapsto 3, \mathsf{t} \mapsto 2\}) \\
\longrightarrow\ &(\mathbf{nil}, \{\mathsf{x} \mapsto 3, \mathsf{y} \mapsto 2, \mathsf{t} \mapsto 2\})
\end{aligned}
$$

(a) Plotkin style

$$
\begin{aligned}
&\mathsf{t} := \mathsf{x} \,;\, \mathsf{x} := \mathsf{y} \,;\, \mathsf{y} := \mathsf{t} \\
\xrightarrow{\mathsf{x}=2}\ &\mathsf{t} := 2 \,;\, \mathsf{x} := \mathsf{y} \,;\, \mathsf{y} := \mathsf{t} \\
\xrightarrow{\mathsf{t}:=2}\ &\mathbf{nil} \,;\, \mathsf{x} := \mathsf{y} \,;\, \mathsf{y} := \mathsf{t} \\
\xrightarrow{\tau}\ &\mathsf{x} := \mathsf{y} \,;\, \mathsf{y} := \mathsf{t} \\
\xrightarrow{\mathsf{y}=3}\ &\mathsf{x} := 3 \,;\, \mathsf{y} := \mathsf{t} \\
\xrightarrow{\mathsf{x}:=3}\ &\mathbf{nil} \,;\, \mathsf{y} := \mathsf{t} \\
\xrightarrow{\tau}\ &\mathsf{y} := \mathsf{t} \\
\xrightarrow{\mathsf{t}=2}\ &\mathsf{y} := 2 \\
\xrightarrow{\mathsf{y}:=2}\ &\mathbf{nil}
\end{aligned}
$$

(b) Integrated style

Fig. 3. Swapping x and y via t

$$
\mathsf{c} ::= \mathbf{nil} \mid \mathsf{x} := \mathsf{e} \mid (\mathsf{c}_0 \,;\, \mathsf{c}_1) \mid (\mathbf{state}\ \sigma \bullet \mathsf{c}) \mid \mathbf{if}\ \mathsf{b}\ \mathbf{then}\ \mathsf{c}_0\ \mathbf{else}\ \mathsf{c}_1 \mid \mathbf{while}\ \mathsf{b}\ \mathbf{do}\ \mathsf{c} \mid (\mathsf{c}_0 \parallel \mathsf{c}_1) \mid (\mathsf{x} == \mathsf{y} \bullet \mathsf{c})
$$

Plotkin-style defines a relation over configurations consisting of pairs of command and state (c, σ), with transitions representing steps of computation.

$$(\mathsf{c}, \sigma) \longrightarrow (\mathsf{c}', \sigma')$$

Integrated-style defines a labelled transition system

$$\mathsf{c} \xrightarrow{\ell} \mathsf{c}'$$

where the labels are as defined earlier.

The semantics of basic commands is given in Figure 2. An assignment $\mathsf{x} := \mathsf{e}$ is defined by rules that evaluate its expression e (Rules 3.1 and 3.2) and final rules that update the variable x (Rules 3.3 and 3.4). In the Plotkin style, Rule 3.3 (P-Assign-Final) updates the state so that the variable x has the value κ, while in the integrated style Rule 3.4 (I-Assign-Final) has a label $x := \kappa$ to indicate that the variable x in the (implicit) context is to be updated to be the constant κ. The rules for sequential composition are similarly split into rules for executing steps of the first command and rules that handle the case when the first command has terminated (is **nil**).

Figure 3 gives an application of the rules when swapping x and y (via t) in both styles. In the Plotkin style, expressions are explicitly evaluated using the state and assignments explicitly update the state. The transitions for the integrated style do not explicitly refer to the state but are intended to be embedded in a state in which x is initially two and y is three; the initial value of t does not matter.

Rule 4.1 I-Test-Local

$$\frac{\mathsf{c} \xrightarrow{\mathsf{x}=\kappa} \mathsf{c}' \quad \mathsf{x} \in \mathrm{dom}(\sigma) \quad \sigma(\mathsf{x}) = \kappa}{(\mathbf{state}\ \sigma \bullet \mathsf{c}) \xrightarrow{\tau} (\mathbf{state}\ \sigma \bullet \mathsf{c}')}$$

Rule 4.2 I-Test-Global

$$\frac{\mathsf{c} \xrightarrow{\mathsf{x}=\kappa} \mathsf{c}' \quad \mathsf{x} \notin \mathrm{dom}(\sigma)}{(\mathbf{state}\ \sigma \bullet \mathsf{c}) \xrightarrow{\mathsf{x}=\kappa} (\mathbf{state}\ \sigma \bullet \mathsf{c}')}$$

Rule 4.3 I-State-Hidden

$$\frac{\mathsf{c} \xrightarrow{\tau} \mathsf{c}'}{(\mathbf{state}\ \sigma \bullet \mathsf{c}) \xrightarrow{\tau} (\mathbf{state}\ \sigma \bullet \mathsf{c}')}$$

Rule 4.4 I-Update-Local

$$\frac{\mathsf{c} \xrightarrow{\mathsf{x}:=\kappa} \mathsf{c}' \quad \mathsf{x} \in \mathrm{dom}(\sigma)}{(\mathbf{state}\ \sigma \bullet \mathsf{c}) \xrightarrow{\tau} (\mathbf{state}\ \sigma[\mathsf{x} \mapsto \kappa] \bullet \mathsf{c}')}$$

Rule 4.5 I-Update-Global

$$\frac{\mathsf{c} \xrightarrow{\mathsf{x}:=\kappa} \mathsf{c}' \quad \mathsf{x} \notin \mathrm{dom}(\sigma)}{(\mathbf{state}\ \sigma \bullet \mathsf{c}) \xrightarrow{\mathsf{x}:=\kappa} (\mathbf{state}\ \sigma \bullet \mathsf{c}')}$$

Rule 4.6 I-State-Final

$$(\mathbf{state}\ \sigma \bullet \mathbf{nil}) \xrightarrow{\tau} \mathbf{nil}$$

Fig. 4. Rules for local state

Relating the Plotkin and Integrated Styles

An integrated transition of the form $\mathsf{c} \xrightarrow{\ell} \mathsf{c}'$ corresponds to the Plotkin style rule

$$\frac{(\sigma, \sigma') \in [\![\ell]\!]}{(c, \sigma) \longrightarrow (c', \sigma')}$$

For example, the integrated transition $\mathsf{x} := \kappa \xrightarrow{x:=\kappa} \mathbf{nil}$ corresponds to the rule

$$\frac{\sigma' = \sigma[\mathsf{x} \mapsto \kappa]}{(\mathsf{x} := \kappa, \sigma) \longrightarrow (\mathbf{nil}, \sigma')}$$

which is equivalent to Rule 3.3 (P-Assign-Final).

4 Local State

In the integrated style, local state is treated by a separate set of rules given in Figure 4. In each rule, the transition above the line is encapsulated in a local state σ below the line. Here, the state σ is a *partial* function from identifiers to values, the domain of σ, i.e., $\mathrm{dom}(\sigma)$, gives the names of the local variables, and for each variable x within $\mathrm{dom}(\sigma)$, $\sigma(\mathsf{x})$ gives the value of x in σ.

There are two rules each for state tests and updates, a rule to handle hidden transitions, and a rule to handle a terminated command. The choice between the two rules for state tests depends on whether the variable x being tested is in the local state σ. If it is (Rule 4.1) then the transition can only be encapsulated in the local state provided that the value of x in σ is the constant κ; in this case the transition becomes a hidden (τ) transition. If x is not local (Rule 4.2) the test label above the line is maintained as the label of the transition below the line. Similarly, the choice between the state update rules depends on whether x is local to σ. If it is (Rule 4.4) the local state σ is updated so that x takes on the value κ and the transition is hidden (has τ as a label). If x is not local (Rule 4.5)

Rules	Transitions	Old label
	(**state** $\{\mathsf{t} \mapsto 0\} \bullet \mathsf{t} := \mathsf{x} \,;\, \mathsf{x} := \mathsf{y} \,;\, \mathsf{y} := \mathsf{t}$)	
4.2	$\xrightarrow{\mathsf{x}=2}$ (**state** $\{\mathsf{t} \mapsto 0\} \bullet \mathsf{t} := 2 \,;\, \mathsf{x} := \mathsf{y} \,;\, \mathsf{y} := \mathsf{t}$)	
4.4	$\xrightarrow{\tau}$ (**state** $\{\mathsf{t} \mapsto 2\} \bullet$ **nil** $\,;\, \mathsf{x} := \mathsf{y} \,;\, \mathsf{y} := \mathsf{t}$)	was $\mathsf{t} := 2$
3.8, 4.3	$\xrightarrow{\tau}$ (**state** $\{\mathsf{t} \mapsto 2\} \bullet \mathsf{x} := \mathsf{y} \,;\, \mathsf{y} := \mathsf{t}$)	
4.2	$\xrightarrow{\mathsf{y}=3}$ (**state** $\{\mathsf{t} \mapsto 2\} \bullet \mathsf{x} := 3 \,;\, \mathsf{y} := \mathsf{t}$)	
4.5	$\xrightarrow{\mathsf{x}:=3}$ (**state** $\{\mathsf{t} \mapsto 2\} \bullet$ **nil** $\,;\, \mathsf{y} := \mathsf{t}$)	
3.8, 4.3	$\xrightarrow{\tau}$ (**state** $\{\mathsf{t} \mapsto 2\} \bullet \mathsf{y} := \mathsf{t}$)	
4.1	$\xrightarrow{\tau}$ (**state** $\{\mathsf{t} \mapsto 2\} \bullet \mathsf{y} := 2$)	was $\mathsf{t} = 2$
4.5	$\xrightarrow{\mathsf{y}:=2}$ (**state** $\{\mathsf{t} \mapsto 2\} \bullet$ **nil**)	
4.6	$\xrightarrow{\tau}$ **nil**	

Fig. 5. Swapping x and y (via local t) in the integrated style

the update label above the line is maintained as the label of the transition below the line. A hidden (τ) transition (Rule 4.3) remains hidden. When its body has terminated, a local state command can terminate (Rule 4.6).

Figure 5 gives an example of swapping non-local variables x and y via a local variable t. The transitions in this sequence are based on those given in Figure 3(b) by applying one of the local state rules at each stage and adding a final step that removes the local state from a terminated command using Rule 4.6 (I-State-Final). Two of the transitions from Figure 3(b) have labels that refer to the (now) local variable t and hence those transitions become hidden; these are marked in the figure.

Rules 4.1–4.5 may be combined into a single rule by the use of auxiliary operators that define the effect of a state on a label ($\ell[\sigma]$), the effect of a label on a state ($\sigma[\ell]$), and whether a label is consistent with a state ($\mathit{consistent}(\ell, \sigma)$). Each of Rules 4.1–4.5 becomes a special case of the following [5].

Rule 4.7 I-State-Step

$$\frac{\mathsf{c} \xrightarrow{\ell} \mathsf{c}' \quad \mathit{consistent}(\ell, \sigma)}{(\textbf{state}\ \sigma \bullet \mathsf{c}) \xrightarrow{\ell[\sigma]} (\textbf{state}\ \sigma[\ell] \bullet \mathsf{c}')}$$

where

$$\ell[\sigma] = \begin{cases} \tau & \text{if } \ell \text{ is } \mathsf{x} = \kappa \text{ and } \mathsf{x} \in \mathrm{dom}(\sigma) \\ \tau & \text{if } \ell \text{ is } \mathsf{x} := \kappa \text{ and } \mathsf{x} \in \mathrm{dom}(\sigma) \\ \ell & \text{otherwise} \end{cases}$$

$$\sigma[\ell] = \begin{cases} \sigma[\mathsf{x} \mapsto \kappa] & \text{if } \ell \text{ is } \mathsf{x} := \kappa \text{ and } \mathsf{x} \in \mathrm{dom}(\sigma) \\ \sigma & \text{otherwise} \end{cases}$$

$$\mathit{consistent}(\ell, \sigma) = \begin{cases} \sigma(\mathsf{x}) = \kappa & \text{if } \ell \text{ is } \mathsf{x} = \kappa \text{ and } \mathsf{x} \in \mathrm{dom}(\sigma) \\ \text{true} & \text{otherwise} \end{cases}$$

The effect of $\ell[\sigma]$ is to hide any label ℓ that either tests or updates a variable local to σ. If ℓ is a state update $\mathsf{x} := \kappa$ and x is local to σ, the state $\sigma[\ell]$ is the state σ with x updated to κ, otherwise $\sigma[\ell]$ is σ. A test label $\mathsf{x} = \kappa$ where x is local to σ is consistent with σ if $\sigma(\mathsf{x}) = \kappa$; all other labels are consistent with σ.

Plotkin-style rules explicitly merge state information into all the other rules and hence only one additional rule is required to handle the local state construct.

Rule 4.8 P-State-Step

$$\frac{(\mathsf{c}, \sigma[\mathsf{x} \mapsto \kappa]) \longrightarrow (\mathsf{c}', \sigma')}{((\textbf{state } \{\mathsf{x} \mapsto \kappa\} \bullet \mathsf{c}), \sigma) \longrightarrow ((\textbf{state } \{\mathsf{x} \mapsto \sigma'(\mathsf{x})\} \bullet \mathsf{c}'), \sigma'[\mathsf{x} \mapsto \sigma(\mathsf{x})])}$$

The main problem with the local-state rule in the Plotkin style is that the variable x is in the domains[1] of both σ and $\{\mathsf{x} \mapsto \kappa\}$. Because c is executed in the context of local state, above the line the state σ is updated so that x has the value κ. Below the line, the value of x in the local state after the transition is its value in σ', while the value of x in the global state σ is unchanged from its initial value, although other state variables within σ may have been updated within σ' by the execution step.

5 Control Structures

The rules for control structures in the integrated style are given below. Rule 5.1 (I-If-Step) handles evaluating the boolean condition in an "if" command, Rule 5.2 (I-If-True) handles the case when the expression evaluates to true and Rule 5.3 (I-If-False) when it evaluates to false. The rule for a "while" command simply unrolls the loop once to an "if" command so that the rules for the "if" command can then be used.

Rule 5.1 I-If-Step

$$\frac{\mathsf{b} \xrightarrow{\ell} \mathsf{b}'}{\textbf{if } \mathsf{b} \textbf{ then } \mathsf{c}_0 \textbf{ else } \mathsf{c}_1 \xrightarrow{\ell} \textbf{if } \mathsf{b}' \textbf{ then } \mathsf{c}_0 \textbf{ else } \mathsf{c}_1}$$

Rule 5.2 I-If-True
$\textbf{if } \text{true} \textbf{ then } \mathsf{c}_0 \textbf{ else } \mathsf{c}_1 \xrightarrow{\tau} \mathsf{c}_0$

Rule 5.3 I-If-False
$\textbf{if } \text{false} \textbf{ then } \mathsf{c}_0 \textbf{ else } \mathsf{c}_1 \xrightarrow{\tau} \mathsf{c}_1$

Rule 5.4 I-While-Unroll
$\textbf{while } \mathsf{b} \textbf{ do } \mathsf{c} \xrightarrow{\tau} \textbf{if } \mathsf{b} \textbf{ then } (\mathsf{c} \ ; \textbf{ while } \mathsf{b} \textbf{ do } \mathsf{c}) \textbf{ else nil}$

6 Concurrency

The rules for interleaving concurrency are similar to those for process algebras.

Rule 6.1 I-Parallel-Step-Left

$$\frac{\mathsf{c}_0 \xrightarrow{\ell} \mathsf{c}_0'}{\mathsf{c}_0 \parallel \mathsf{c}_1 \xrightarrow{\ell} \mathsf{c}_0' \parallel \mathsf{c}_1}$$

Rule 6.2 I-Parallel-Step-Right

$$\frac{\mathsf{c}_1 \xrightarrow{\ell} \mathsf{c}_1'}{\mathsf{c}_0 \parallel \mathsf{c}_1 \xrightarrow{\ell} \mathsf{c}_0 \parallel \mathsf{c}_1'}$$

Rule 6.3 I-Parallel-Final-Left
$\textbf{nil} \parallel \mathsf{c} \xrightarrow{\tau} \mathsf{c}$

Rule 6.4 I-Parallel-Final-Right
$\mathsf{c} \parallel \textbf{nil} \xrightarrow{\tau} \mathsf{c}$

[1] Recall that states used within configurations in the Plotkin rules are total functions.

In the standard operational semantics for process algebras, like CSP [6], transitions are labelled by events and there is a rule to handle the case where two processes synchronise on an event. Such rules can be incorporated directly into the integrated style of operational semantics by extending labels to include events. An integrated operational semantics for a language extending CSP with state-based constructs is given in [3].

7 Big-Step Operational Semantics

As well as small-step semantics, Plotkin also makes use of big-step operational semantics. The big-step rules for expressions consider their complete evaluation to a value, and those for commands their complete execution until termination. Rule 2.1 (P-Variable) is used unchanged. Rule 7.1 (P-Binary-Big-Step) below gives the big-step rule for a binary expression. Above the line e_0 evaluates to κ_0 and then e_1 evaluates to κ_1, and hence below the line $\mathsf{e}_0 \leq \mathsf{e}_1$ evaluates to the value of $\kappa_0 \leq \kappa_1$. For commands the big-step transitions are of the form $(\mathsf{c}, \sigma) \rightarrow (\mathbf{nil}, \sigma')$, i.e., from a command c and initial state σ to command **nil**, indicating a terminated command, and final state σ'. Rule 7.2 (P-Sequential-Big-Step) gives the big-step rule for a sequential composition. Rule 7.3 (P-Specification) gives the rule for an atomic specification command, $[R]$, consisting of a relation R between states.

Rule 7.1 P-Binary-Big-Step

$$\frac{(\mathsf{e}_0, \sigma) \longrightarrow (\kappa_0, \sigma') \qquad (\mathsf{e}_1, \sigma') \longrightarrow (\kappa_1, \sigma'') \qquad \kappa = (\kappa_0 \leq \kappa_1)}{(\mathsf{e}_0 \leq \mathsf{e}_1, \sigma) \longrightarrow (\kappa, \sigma'')}$$

Rule 7.2 P-Sequential-Big-Step

$$\frac{(\mathsf{c}_0, \sigma) \longrightarrow (\mathbf{nil}, \sigma') \qquad (\mathsf{c}_1, \sigma') \longrightarrow (\mathbf{nil}, \sigma'')}{(\mathsf{c}_0 \ ; \mathsf{c}_1, \sigma) \longrightarrow (\mathbf{nil}, \sigma'')}$$

Rule 7.3 P-Specification

$$\frac{(\sigma, \sigma') \in R}{([R], \sigma) \longrightarrow (\mathbf{nil}, \sigma')}$$

Big-step semantics can be expressed in the integrated style by allowing the labels of transitions to be relations. Figure 6 gives the equivalents of the above rules. Rule 7.4 (I-Variable-Big-Step) labels the transition with a relation $[\![\mathsf{x} = \kappa]\!]$, which denotes the identity relation on states restricted to those states in which x is κ (see (6)). For a binary expression, the relations R_0 and R_1 corresponding to the evaluations of e_0 and e_1 are composed to give the overall evaluation relation $R_0 \fatsemi R_1$, provided that this relation is non-empty. The rules for sequential composition are similar. The rule for a specification with relation R applies only if R is nonempty and simply labels the transition with R. The big-step rules only handle terminating constructs and cannot express interleaving.

Rule 7.4 I-Variable-Big-Step

$$\mathsf{x} \xrightarrow{[\![\mathsf{x}=\kappa]\!]} \kappa$$

Rule 7.5 I-Binary-Big-Step

$$\frac{\mathsf{e}_0 \xrightarrow{R_0} \kappa_0 \qquad \mathsf{e}_1 \xrightarrow{R_1} \kappa_1 \qquad R_0 \mathbin{⨾} R_1 \neq \varnothing \quad \kappa = (\kappa_0 \leq \kappa_1)}{\mathsf{e}_0 \leq \mathsf{e}_1 \xrightarrow{R_0 \mathbin{⨾} R_1} \kappa}$$

Rule 7.6 I-Sequential-Big-Step

$$\frac{\mathsf{c}_0 \xrightarrow{R_0} \mathbf{nil} \qquad \mathsf{c}_1 \xrightarrow{R_1} \mathbf{nil} \qquad R_0 \mathbin{⨾} R_1 \neq \varnothing}{\mathsf{c}_0 \,;\, \mathsf{c}_1 \xrightarrow{R_0 \mathbin{⨾} R_1} \mathbf{nil}}$$

Rule 7.7 I-Specification

$$\frac{R \neq \varnothing}{[R] \xrightarrow{R} \mathbf{nil}}$$

Fig. 6. Big-step operational semantics in the integrated style

Relation between the Big-Step Plotkin and Integrated Styles

The integrated big-step transitions $\mathsf{e} \xrightarrow{R} \kappa$ and $\mathsf{c} \xrightarrow{R} \mathbf{nil}$ correspond to the following Plotkin rules, respectively.

$$\frac{(\sigma, \sigma') \in R}{(\mathsf{e}, \sigma) \longrightarrow (\kappa, \sigma')} \qquad \frac{(\sigma, \sigma') \in R}{(\mathsf{c}, \sigma) \longrightarrow (\mathbf{nil}, \sigma')}$$

8 Multi-step Operational Semantics

Fewer rules are required for the big-step semantics than the corresponding small-step rules (in both styles). However, the small-step rules allow concurrency to be specified by interleaving (small) steps from the two parallel commands. This is not possible in the big-step rules because the small steps required for interleaving are not available.

On the other hand, in the big-step semantics one can express the behaviour of an atomic specification command $[R]$ as a single big step, however, defining such a command in a small-step style becomes problematic if one needs to ensure the specification terminates. To support both concurrency and non-atomic specification commands, the integrated style allows one to define a multi-step semantics in which transitions are labelled with a *sequence* of small-step labels. Because the small steps are retained, one can express concurrency as an interleaving of the small steps, and one can also express the behaviour of a non-atomic specification command, $[R]$, as allowing any sequence of small steps which when composed together satisfy R.

Rule 8.1 M-Sequential-Steps

$$\frac{\mathsf{c}_0 \xrightarrow{\ell s} \mathsf{c}_0'}{\mathsf{c}_0 \,;\, \mathsf{c}_1 \xrightarrow{\ell s} \mathsf{c}_0' \,;\, \mathsf{c}_1}$$

Rule 8.2 M-Sequential-Join

$$\frac{\mathsf{c} \xrightarrow{\ell s_0} \mathsf{c}' \quad \mathsf{c}' \xrightarrow{\ell s_1} \mathsf{c}''}{\mathsf{c} \xrightarrow{\ell s_0 \frown \ell s_1} \mathsf{c}''}$$

Rule 8.3 M-Parallel-Steps

$$\frac{\mathsf{c}_0 \xrightarrow{\ell s_0} \mathsf{c}_0' \quad \mathsf{c}_1 \xrightarrow{\ell s_1} \mathsf{c}_1' \quad \ell s \in \ell s_0 ||| \ell s_1}{\mathsf{c}_0 \parallel \mathsf{c}_1 \xrightarrow{\ell s} \mathsf{c}_0' \parallel \mathsf{c}_1'}$$

Rule 8.4 M-Null

$$\mathsf{c} \xrightarrow{\langle\rangle} \mathsf{c}$$

Rule 8.5 M-Specification

$$\frac{{}^{\circ}_{9}/[\![\ell s]\!] \subseteq R \qquad {}^{\circ}_{9}/[\![\ell s]\!] \neq \varnothing}{[R] \xrightarrow{\ell s} \mathbf{nil}}$$

Fig. 7. Multi-step operational semantics

A collection of interesting multi-step rules is given in Figure 7. The operator "$|||$" forms the set of all possible interleavings of the two sequences of labels given as its operands. The semantic brackets around a sequence of labels, ℓs, convert it into a sequence of the corresponding relations, and the operator "${}^{\circ}_{9}/[\![\ell s]\!]$" composes the sequence of relations to form a relation.

Rule 8.1 generalises Rule 3.6 (I-Sequential-Step) to allow c_0 to take any sequence of steps rather than just a single step, and Rule 8.2 allows two consecutive sequences of steps to be combined into a single sequence of steps. Rule 8.3 allows the parallel combination of two commands to evolve via a sequence of steps that is some interleaving of sequences of steps taken by the two commands. Rule 6.3 (I-Parallel-Final-Left) and Rule 6.4 (I-Parallel-Final-Right) can be reused to handle termination of either process. Rule 8.4 ensures the empty sequence of steps makes no progress. To handle a specification command, Rule 8.5 allows any finite sequence of steps whose composition is both non-empty and satisfies R. In the context of rely-guarantee reasoning about concurrent programs [7] this rule can be adapted to express a guarantee constraint, a relation g, on each step of the execution by requiring each label in the sequence ℓs to satisfy the relation g.[2]

Note that the small step rules can be viewed as special (single step) cases of the multi-step rules. For example, instantiating Rule 8.1 (M-Sequential-Steps) with ℓs as the singleton sequence $\langle \ell \rangle$ gives the following rule which is effectively the same as Rule 3.6 (I-Sequential-Step).

$$\frac{\mathsf{c}_0 \xrightarrow{\langle \ell \rangle} \mathsf{c}_0'}{\mathsf{c}_0 \,;\, \mathsf{c}_1 \xrightarrow{\langle \ell \rangle} \mathsf{c}_0' \,;\, \mathsf{c}_1}$$

[2] Handling the semantics of rely-guarantee was our initial motivation for exploring the multi-step approach for specifications.

Similarly, instantiating Rule 8.3 (M-Parallel-Steps) with ℓs_0 as $\langle \ell \rangle$ and ℓs_1 as the empty sequence $\langle\rangle$ gives a rule equivalent to Rule 6.1 (I-Parallel-Step-Left).

$$\frac{\mathsf{c}_0 \xrightarrow{\langle \ell \rangle} \mathsf{c}_0' \quad \mathsf{c}_1 \xrightarrow{\langle\rangle} \mathsf{c}_1 \quad \langle \ell \rangle \in \langle \ell \rangle ||| \langle\rangle}{\mathsf{c}_0 \parallel \mathsf{c}_1 \xrightarrow{\langle \ell \rangle} \mathsf{c}_0' \parallel \mathsf{c}_1}$$

9 Aliasing

The syntax $(\mathsf{x} == \mathsf{y} \bullet \mathsf{c})$ introduces a new name x as an alias for y for the execution of c. In the integrated style the semantics of aliasing is easily handled by replacing any occurrences of x in a label with y, thus any tests or updates of x become tests or updates of y. Note that occurrences of y in a label are not renamed, so any occurrence of either x or y becomes an occurrence of y.

Rule 9.1 I-Alias-Step

$$\frac{\mathsf{c} \xrightarrow{\ell} \mathsf{c}'}{(\mathsf{x} == \mathsf{y} \bullet \mathsf{c}) \xrightarrow{\ell[x \mapsto y]} (\mathsf{x} == \mathsf{y} \bullet \mathsf{c}')}$$

The simplicity of this rule relies on the fact that labels are syntactic rather than semantic and hence renaming can be applied to the labels. This is a small but significant difference from the labels used in MSOS [11], which contain environments and states, and hence do not allow this simple form of renaming.

To handle aliasing in the Plotkin style, locations are introduced with

- a store, $\sigma \in \mathit{Loc} \twoheadrightarrow \mathit{Val}$, that maps locations to values, and
- an environment, $\rho \in \mathit{Id} \twoheadrightarrow \mathit{Loc}$, that maps variable names to locations.

The rules always ensure that $\mathrm{ran}(\rho) \subseteq \mathrm{dom}(\sigma)$.

Unfortunately, all the previous rules need to be rewritten, e.g.,

$$\frac{\sigma(\mathsf{x}) = \kappa}{(\mathsf{x}, \sigma) \longrightarrow (\kappa, \sigma)} \quad \text{becomes} \quad \frac{\sigma(\rho(\mathsf{x})) = \kappa}{\rho \vdash (\mathsf{x}, \sigma) \longrightarrow (\kappa, \sigma)}$$

$$(\mathsf{x} := \kappa, \sigma) \longrightarrow (\mathbf{nil}, \sigma[\mathsf{x} \mapsto \kappa]) \text{ becomes } \rho \vdash (\mathsf{x} := \kappa, \sigma) \longrightarrow (\mathbf{nil}, \sigma[\rho(\mathsf{x}) \mapsto \kappa])$$

The Plotkin-style rule makes use of locations by mapping x to the same location as y.

Rule 9.2 P-Alias-Step

$$\frac{\rho[\mathsf{x} \mapsto \rho(\mathsf{y})] \vdash (\mathsf{c}, \sigma) \longrightarrow (\mathsf{c}', \sigma')}{\rho \vdash ((\mathsf{x} == \mathsf{y} \bullet \mathsf{c}), \sigma) \longrightarrow ((\mathsf{x} == \mathsf{y} \bullet \mathsf{c}'), \sigma')}$$

As an example of aliasing in the integrated style, we extend the swap example given in Figure 5. That example generated a sequence of labels

$$\langle \mathsf{x} = 2, \tau, \tau, \mathsf{y} = 3, \mathsf{x} := 3, \tau, \tau, \mathsf{y} := 2, \tau \rangle.$$

Because finite sequences of τ steps have no effect,[3] we can abbreviate the effect of that execution trace via the following multi-step transition, which is labelled with the above sequence minus the τ steps.

$$(\mathbf{state}\ \{\mathsf{t} \mapsto 0\} \bullet \mathsf{t} := \mathsf{x}\ ;\ \mathsf{x} := \mathsf{y}\ ;\ \mathsf{y} := \mathsf{t}) \xrightarrow{\langle \mathsf{x}=2,\mathsf{y}=3,\mathsf{x}:=3,\mathsf{y}:=2\rangle} \mathbf{nil}$$

The execution of the same command but aliasing x to v and y to w gives the following multi-step transition, in which x and y have been renamed to v and w, respectively.

$$(\mathsf{x}, \mathsf{y} == \mathsf{v}, \mathsf{w} \bullet (\mathbf{state}\ \{\mathsf{t} \mapsto 0\} \bullet \mathsf{t} := \mathsf{x}\ ;\ \mathsf{x} := \mathsf{y}\ ;\ \mathsf{y} := \mathsf{t})) \xrightarrow{\langle \mathsf{v}=2,\mathsf{w}=3,\mathsf{v}:=3,\mathsf{w}:=2\rangle} \mathbf{nil}$$

The aliasing command can be used to define call-by-reference parameters for procedures [5].

10 Conclusions and Related Work

Mosses' Modular Stuctural Operational Semantics (MSOS) [10,11] defines a general framework supporting label-based transition rules. Mosses shows that, in general, relocating information from the configurations (typically the state, or store and environment) results in more concise rules, and in particular supports *modularity*, which in this context means that as the complexity of a language increases (reflected in an increasing configuration size), the number of rules that need to be rewritten is minimal. This is of special benefit for incrementally developing a formal operational description of complex languages. The labels are tuples of relevant information, and can be used to seamlessly manage both state information as well as events such as exceptions or CSP event synchronisation.

In a sense, the semantics we present here is an instance of MSOS, except that instead of using environments and states as labels, syntactic labels representing (mini-)relations between states are used. Because we use labels, we obtain the benefit of modularity, and also have single-place configurations, which serves to keep the rules relatively concise.

The idea of using syntactic labels in operational semantics has appeared in at least two other independent pieces of work by Owens [12] and Abadi & Harris [1]. In both cases the labels were used to reduce the size of configurations and separate concerns. Neither work considers the local state command, nor extensions to simplify the semantics of aliasing and hence call-by-reference parameters to procedures.

Although not explored in detail here, the integrated style of operational semantics can support both event-based and state-based constructs. An integrated semantics has been given for a language CSP_σ, which extends CSP with state-based constructs [3]. In addition, an integrated semantics has been given for the more complex language of Behavior Trees, which as well as event-based and state-based constructs also includes a message passing facility similar to that used in publish/subscribe protocols [4].

[3] An infinite sequence of τ steps corresponds to (internal) divergence of the program.

Mosses recognises the possibility of using a sequence as a label but to quote [11, page 216] "The possibility of specifying interleaving in a big-step MSOS is a technical curiosity, but of little practical relevance for applications of MSOS, which generally stick to the small-step style." Our desire to combine both concurrency (which is not accommodated in the big-step style) and non-atomic specification commands (which are not handled by either approach) led us to more fully explore a multi-step semantics.

Overall the integrated style of operational semantics using "syntactic" labels, combined with a multi-step semantics using sequences of steps as labels, allows one to express a wider range of constructs more simply.

Acknowledgements. This research was supported by Australian Research Council Discovery Grant DP0987452. We would like to thank Brijesh Dongol for feedback on earlier drafts of this paper.

References

1. Abadi, M., Harris, T.: Perspectives on Transactional Memory. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 1–14. Springer, Heidelberg (2009)
2. Back, R.-J.R., von Wright, J.: Refinement Calculus: A Systematic Introduction. Springer (1998)
3. Colvin, R., Hayes, I.J.: CSP with Hierarchical State. In: Leuschel, M., Wehrheim, H. (eds.) IFM 2009. LNCS, vol. 5423, pp. 118–135. Springer, Heidelberg (2009)
4. Colvin, R.J., Hayes, I.J.: A semantics for Behavior Trees using CSP with specification commands. Science of Computer Programming 76(10), 891–914 (2011)
5. Colvin, R.J., Hayes, I.J.: Structural operational semantics through context-dependent behaviour. Journal of Logic and Algebraic Programming 80(7), 392–426 (2011)
6. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
7. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Transactions on Programming Languages and Systems 5(4), 596–619 (1983)
8. Milner, A.J.R.G.: Communication and Concurrency. Prentice-Hall (1989)
9. Morgan, C.C.: Programming from Specifications, 2nd edn. Prentice Hall (1994)
10. Mosses, P.D.: Exploiting labels in structural operational semantics. Fundam. Inform. 60(1-4), 17–31 (2004)
11. Mosses, P.D.: Modular structural operational semantics. J. Log. Algebr. Program. 60-61, 195–228 (2004)
12. Owens, S.: A Sound Semantics for OCaml$_{light}$. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 1–15. Springer, Heidelberg (2008)
13. Plotkin, G.D.: A structural approach to operational semantics. J. Log. Algebr. Program. 60-61, 17–139 (2004)

Test Generation for Sequential Nets of Abstract State Machines*

Paolo Arcaini[1], Francesco Bolis[2], and Angelo Gargantini[2]

[1] Dip. di Tecnologie dell'Informazione, Università degli Studi di Milano, Italy
paolo.arcaini@unimi.it
[2] Dip. di Ing. dell'Informazione e Metodi Matematici, Università di Bergamo, Italy
{francesco.bolis,angelo.gargantini}@unibg.it

Abstract. Test generation techniques based on model checking suffer from the state space explosion problem. However, for a family of systems that can be easily decomposed in sub-systems, we devise a technique to cope with this problem. To model such systems, we introduce the notion of *sequential net* of Abstract State Machines (ASMs), which represents a system constituted by a set of ASMs such that only one ASM is active at every time. Given a net of ASMs, we first generate a test suite for every ASM in the net, then we combine the tests in order to obtain a test suite for the entire system. We prove that, under some assumptions, the technique preserves coverage of the entire system. We test our approach on a benchmark and we report a web application example for which we are able to generate complete test suites.

1 Introduction

Model-based testing (MBT) aims to (re)use models and specifications for software testing. One of the main applications of MBT consists in test generation where tests are automatically generated from possibly partial and abstract models of the system under test. We here assume that MBT is performed in a typical black-box way: test suites are derived from models and not from source code.

Although MBT and test generation from models are rather mature topics in software testing and several approaches and tools exist [15], MBT for complex software systems is still an evolving field and its scalability is still questionable.

In a recent and still ongoing MBT project, we have tried to model web applications with Abstract State Machines (ASMs) and use a tool for test generation. Since the used technique is based on model checking [9], one of the main obstacles has been the scalability of the approach and soon we encountered the well known *state space explosion problem*. Indeed, the problem of the model checking method is that the computational complexity increases in an exponential mode together with the size of the model. Several techniques exist to overcome this limitation, like symbolic representation of states, compact storing of states,

* The second author has been supported by the project Ricerca Applicata per il Territorio - Berg. II - Regione Lombardia and Alcatel-Lucent Spa.

J. Derrick et al. (Eds.): ABZ 2012, LNCS 7316, pp. 36–50, 2012.

and efficient state space exploration. However, these techniques may still fail or weaken the coverage of the state space.

On the other hand, the system under test may have some peculiarities that can be exploited to limit the state explosion. We focus on systems that are composed of independent sub-systems that pass the control to each other such that only one sub-system is active at any time. In a web application, for instance, only one page is active at any time.

Such systems can be modeled as *sequential nets* of ASMs, defined in Sect. 3, that are sets of ASMs having some features including that only one ASM is active at every time.

In Sect. 4 we present a technique that is able to generate tests for a net of ASMs, reducing the state explosion. A test suite that covers every single machine is generated. These test suites are combined in order to obtain a test suite for the whole system. Under some assumptions, this technique preserves coverage of the entire system and reduces considerably the effort required to generate the whole test suite, as reported in the experiments using a benchmark example (in Sect. 5) and a simple web application (in Sect. 6).

2 Background

Software testing is a costly and time-consuming activity; specification-based (or model-based) testing [10] permits to considerably reduce the testing costs. In specification based testing, a specification describes the expected behavior of the system, and can be used as a test oracle to assess the correctness of the implementation. Moreover, specifications are also usually used to define test adequacy criteria, that determine if a test suite is adequate to test a software; various techniques exist to generate test sequences from formal specifications.

We assume that the reader is familiar with the ASMs [3]. In the following we give some basic definitions about test generation from ASMs.

Definition 1. *A* test sequence *(or* test*) is a finite sequence of states $s_1, \ldots, s_n$ whose first element s_1 is an initial state, and each state s_i (with $i \neq 1$) follows the previous one s_{i-1} by applying the transition rules. The final state s_n is the state where the test goal is achieved.*

Definition 2. *A* test suite *(or* test set*) is a finite set of test sequences.*

Definition 3. *A* test predicate *is a formula over the state and determines if a particular testing goal is reached. A coverage criterion C is a function that, given a formal specification, produces a set of test predicates. A test suite TS satisfies a coverage criterion C if each test predicate generated with C is satisfied in at least one state of a test sequence.*

Several coverage criteria have been defined in [9] for ASMs. One of the basic criteria for ASMs is the *rule coverage*. A test suite satisfies the *rule coverage* criterion if, for every rule r_i, there exists at least one state in a test sequence in which r_i fires and there exists at least a state in a test sequence in which r_i does not fire.

2.1 Test Generation for ASMs by Model Checking

In order to build test suites satisfying some coverage criteria, several approaches have been defined. In this paper we use a technique based on the capability of the model checkers to produce counterexamples [7]. The method consists of steps:

1. The test predicates set $\{tp_i\}$ is derived from the specification according to the desired coverage criteria;
2. The specification is translated into the language of the model checker;
3. For each test predicate tp_i the *trap property* $\Box\neg tp_i$ is proved, where $\Box$ means *always*. If the model checker finds a state s where tp_i is true, it stops and returns as counterexample a state sequence leading to s: such sequence is the test covering tp_i. If the model checker explores the whole state space without finding any state where the trap property is false, then the test predicate is said *infeasible* and it is ignored. In the worst case, the model checker terminates without exploring the whole state space and without finding a violation of the trap property (i.e., without producing any counterexample), usually because of the state explosion problem. In this case, the user does not know if either the trap property is true (i.e., the test is infeasible), or it is false (i.e., there exists a sequence that reaches the goal).

In this paper we use the Asmeta framework[1] and its ATGT tool [8], based on the model checker SPIN [11].

3 Sequential Nets of Abstract State Machines

We focus our attention on those systems that are composed of independent sub-systems that pass the control to each other, so that only one sub-system is active at any time. Usually, in order to describe such kind of systems, a model of each sub-system is developed. A model of coordination is needed for representing the execution of the entire system, i.e., the activation/deactivation of sub-system models according to their local decisions.

A typical example is that of web applications. In a web application just one web page is active at any time, and the active page *decides* which is the next page to be displayed. The coordination is performed by the web browser and the web server that are responsible of closing the current page and visualizing the next one (passing the control among pages).

3.1 Description of the Web Application Case Study

We describe a web application case study taken from [12] we used in our experiments. There are six php pages in the web application under test and each of them, as well as their corresponding ASM, is described below.

– `index.php` – It serves as the login interface for the website. A user is required to enter a username and a password in order to access the other three pages of the site. The *Reset* button clears all text entries, while the *Submit*

[1] `http://asmeta.sourceforge.net/`

button opens up `main.php`, as long as the identification credentials are correct. If any information is missing, an error message page is displayed.

– `error_b.php` – It is activated from `index.php` if any information is missing, or username or password are wrong.

– `main.php` – It permits users to execute different actions. Specifically, users can click on a link (at top left corner of page), upload a file by clicking on the *Browse* button, enter text into a textbox, select a checkbox, and click on a *Submit* button which loads `random.php`.

– `error_a.php` – It is displayed if any information is missing in `main.php`.

– `random.php` – It permits users to execute actions not available in `main.php`. Two links bring the user back to `index.php` and `main.php`. There are also drop-down lists, radio buttons, and a *Submit* button which loads `end.php`.

– `end.php` – It serves as the *end* of the web application. The user has the option of closing the web browser, or clicking on a link to return to `index.php`.

3.2 Definition of Sequential Net of ASMs

We assume that each component of the system is modeled with an ASM and we introduce the notion of sequential net of ASMs as follows.

Definition 4. *A* sequential net of machines *is a set of Abstract State Machines* $M1, \ldots, Mn$ *such that:*

1. *each machine has only one initial state,*
2. *the machine* $M1$ *is the initial machine,*
3. *only one machine is active at any time,*
4. *the active machine decides when and to which machine the control is passed,*
5. *the net is connected, i.e., each machine is reachable from the initial machine.*

A sequential net of ASMs allows one to model a set of machines that do not run in parallel, pass the control to each other, and do not share information, although they share the same environment. We call the net *sequential* because only one machine is running at any time, so the machines are not concurrent; however, there may not be an unique sequence among the machines, since every machine can decide the next machine depending on local decisions. A sequential net is a graph, where each node is a machine and an arc is a transfer of control between two machines.

A possible way to model every single machine M_i of the net, so that it can signal the transfer of control, is the following:

1. add a domain $AsmDomain = \{M_1, \ldots, M_n\}$ to its signature;
2. add a 0-ary function *currAsm* of type *AsmDomain* to its signature; *currAsm*, in the initial state, must assume the value M_i;
3. write the main rule as follows: **if** $currAsm = M_i$ **then** r_mi[] **endif** where r_mi[] is a macro rule that contains the actions of the machine.

Every machine M_i can be independently executed. It executes some useful actions until it changes the value of *currAsm*; after that any other step of execution does not produce any change in the controlled part of the machine.

```
asm M1
signature:
 enum domain
  AsmDomain = {M1, M2, M3}
 monitored a: Integer
 controlled currAsm: AsmDomain
definitions:
  rule r_m1 =
    if a = 2 then
      currAsm := M2
    else if a = 5 then
      currAsm := M3
   else // do machine M1 actions
   endif endif

  main rule r_main1 =
    if currAsm = M1 then r_m1[]
    endif
default init s0:
    function currAsm = M1
```

Code 1. Machine M1.

```
asm M2
signature:
  enum domain
    AsmDomain = {M1, M2, M3}
  monitored b: Integer
  controlled currAsm: AsmDomain
definitions:
  rule r_m2 =
    if b = 2 or b = 30 then
      currAsm := M1
    else if b = 5 or b = 100 then
      currAsm := M3
    else // do machine M2 actions
    endif endif

  main rule r_main2 =
    if currAsm = M2 then r_m2[]
    endif
default init s0:
    function currAsm = M2
```

Code 2. Machine M2.

```
asm M3
signature:
  enum domain
    AsmDomain = {M1, M2, M3}
  monitored c: Integer
  controlled currAsm: AsmDomain
definitions:
  rule r_m3 =
    if c = 2 then
      currAsm := M2
    else if c = 5 then
      currAsm := M1
    else // do machine M3 actions
    endif endif

  main rule r_main3 =
    if currAsm = M3 then r_m3[]
    endif
default init s0:
    function currAsm = M3
```

Code 3. Machine M3.

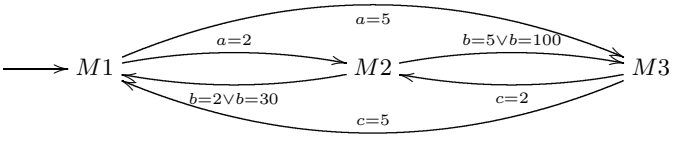


Fig. 1. Three ASMs constituting a sequential net

Example 1. Consider, for instance, the three ASMs shown in Codes 1, 2 and 3. They constitute a sequential net of ASMs (see Fig. 1). For the sake of brevity, we do not specify the internal actions of the machines.

3.3 Product Machine

Several validation and verification activities can be performed directly on the single machines. However, if we want to do a more general evaluation of the system (e.g., simulation of the transitions among machines, or test generation for the whole system), we must also provide a model of the coordination.

One possible simple way is to merge all the machines in an unique *product* ASM as follows:

- the signatures of the machines are merged in a single signature; there is just one copy of the *AsmDomain* domain and of the *currAsm* function in the product machine;
- all macro rules (except the main rules) of the single machines are included;
- in the main rule *r_main*, rules *r_mi*[] are individually called according to the value of the function *currAsm*;
- the initial states are merged; the function *currAsm* is initialized to the value *M1* (the first sub-system is active in the initial state).

Given the sequential net shown in Fig. 1, the product machine is the one shown in Code 4.

```
asm ProductM
signature:
    enum domain AsmDomain = {M1, M2, M3}
    monitored a: Integer
    monitored b: Integer
    monitored c: Integer
    controlled currAsm: AsmDomain
definitions:
    rule r_m1 = if a = 2 then currAsm := M2
                else if a = 5 then currAsm := M3 else // do machine M1 actions
                endif endif
    rule r_m2 = ...
    rule r_m3 = ...
    main rule r_main = if currAsm = M1 then r_m1[]
                       else if currAsm = M2 then r_m2[] else r_m3[] endif endif
default init s0:
    function currAsm = M1
```

Code 4. Product machine of the sequential net in Fig. 1.

4 Test Generation for Sequential Nets of ASMs

In order to efficiently test a system modeled as a sequential net of ASMs, it is not enough to test the single sub-systems, since also the interaction among them must be tested. So, we must generate test sequences that cover the whole application and not just the single sub-systems.

The first idea is to derive the test sequences directly from the product machine that already contains all the interactions among sub-systems. However, since test generation algorithms based on model checking may need to visit the whole state space of the model, the generation of test sequences from the product machine may suffer from the state explosion problem. It would be desirable to have a method in which the model checking must be executed just on the single machines and not on the product machine; indeed, it is computationally easier to execute the model checker several times over small models, rather than executing it one time over a big model. The method should also provide a mechanism for combining the test suites produced for the single machines in an unique test suite to use for testing the whole system: the time taken by the combination of the test suites should be negligible.

4.1 Generating the Test Suites for Every Machine

We use model checking as in [9] to generate a test suite for every ASM. Given the test sequences of a machine M_i, we define *inner* those sequences that terminate in a state in which *currAsm* is M_i, and *exiting* those sequences that terminate in a state in which *currAsm* is M_j (with $j \neq i$). Inner test sequences keep the control of the net in the current machine, whereas exiting sequences pass the control to another machine.

4.2 Building the Test Sequence Graph

The generated test sequences constitute a graph, called *test sequence graph*, where every node is a machine and every arc is a test sequence. Test sequences that do not change the current machine are self loops of a node; test sequences that change the current machine, instead, are arcs between different nodes.

4.3 Combining the Tests by Visiting the Test Sequence Graph

The algorithm used to visit the graph and build the *combined* test sequences is shown in Alg. 1.

Algorithm 1. Visiting the test sequence graph. Procedure *visitGraph*.

Require: the node n to visit
Require: a test sequence *prefix* that permits to reach the node
1: $visitedNodes \leftarrow visitedNodes \cup n$
2: $testSet \leftarrow testSet \cup prefix$
3: **for** $arc \in outArcs(n)$ **do**
4: $prefixToFn \leftarrow prefix + testSeq(arc)$
5: **if** $finalNode(arc) \notin visitedNodes$ **then**
6: $visitGraph(finalNode(arc), prefixToFn)$
7: **else**
8: $testSet \leftarrow testSet \cup prefixToFn$
9: **end if**
10: **end for**

The procedure executes a depth-first search of the graph. It takes as argument a node n to visit and a test sequence *prefix* that permits to reach n; n is marked as *visited* (line 1) in order to not be visited again and *prefix* is added to the test suite *testSet* we are building (line 2). Then, for each exiting arc of n

- the new prefix *prefixToFn* is built concatenating the current *prefix* with the test sequence that brings to the final node *fn* of the arc (line 4);
- if *fn* has not already been visited, *fn* is visited using as prefix *prefixToFn* (line 6); otherwise, *prefixToFn* is added to the test suite (line 8).

The procedure *visitGraph* is invoked using as argument the initial machine $M1$ of the net and the empty test sequence ϵ.

Note that the visit of the test sequence graph has linear complexity with the number of arcs and nodes and it requires a negligible amount of time with respect to the generation of the test suites.

It is straightforward to prove that the test sequences obtained with the presented algorithm are valid sequences for the product machine.

4.4 Coverage

We are interested in investigating the relationship between the coverage provided by a test suite obtained from the single machines and the coverage provided by using the product machine instead.

Definition 5. *A coverage criterion C is* preservable *if any test suite TS, obtained by the combination of tests suites $TS_1, \ldots, TS_n$ that satisfy C over the single machines $M_1, \ldots, M_n$, satisfies C over the product machine.*

If a criterion is preservable, we can satisfy it on the product machine deriving the test sequences from the single machines and combining them later. The *rule coverage* criterion, for example, is *preservable* because of the following reasons:

1. by definition of *sequential net*, every machine is reachable starting from the initial machine; in each single machine M_i, every transition from M_i to another machine is specified with the update of the *currAsm* function;
2. if the *rule coverage* criterion is satisfied in every machine, it means that every rule is executed, including all the updates of the function *currAsm*. So, for each transition, there is a test sequence that contains it;
3. by construction, the *visitGraph* algorithm assures that, if a node of the test sequence graph is reachable, a test sequence that reaches that node is built;
4. in the main rule, the product machine describes the sequential net without adding or removing any transition: at each step it simply executes the rule of the machine specified by *currAsm*.

4.5 Limits of the Approach

The major limit of the proposed approach is that not all criteria are preservable. A criterion, in order to be preservable, must satisfy a necessary (but not sufficient) condition: it must require that, for each machine M_i (with $i \neq 1$), there exists a test sequence of another machine that reaches M_i. The rule coverage criterion satisfies such condition, since it covers all the transitions to other machines. Let's see a criterion that, since it does not satisfy such condition, is not preservable:

> C_{np}: A test suite satisfies the criterion C_{np} if every macro rule r_i is fired in at least one test sequence.

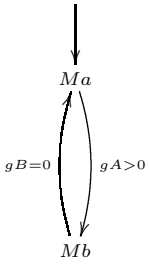
Let's see the test generation process using C_{np}. Let Ma and Mb be two ASMs, shown, respectively, in Code 5 and 6, that constitute a net. The product machine is shown in Code 7.

```
asm Ma
signature:
    enum domain AsmDomain = {Ma, Mb}
    monitored gA: Integer
    controlled currAsm: AsmDomain
definitions:
    rule r_mA = if gA > 0 then
                    currAsm := Mb endif
    main rule r_mainA =
        if currAsm = Ma then r_mA[] endif
default init s0:
    function currAsm = Ma
```

Code 5. Machine Ma.

```
asm Mb
signature:
    enum domain AsmDomain = {Ma, Mb}
    monitored gB: Integer
    controlled currAsm: AsmDomain
definitions:
    rule r_mB = if gB = 0 then
                    currAsm := Ma endif
    main rule r_mainB =
        if currAsm = Mb then r_mB[] endif
default init s0:
    function currAsm = Mb
```

Code 6. Machine Mb.




```
asm ProductMaMb
signature:
    enum domain AsmDomain = {Ma, Mb}
    monitored gA: Integer
    monitored gB: Integer
    controlled currAsm: AsmDomain
definitions:
    rule r_mA = if gA > 0 then currAsm := Mb endif
    rule r_mB = if gB = 0 then currAsm := Ma endif
    main rule r_main = if currAsm = Ma then r_mA[] else r_mB[] endif
default init s0:
    function currAsm = Ma
```

Code 7. Product machine of the machines Ma and Mb.

In the machine Ma, the criterion C_{np} is satisfied if there exists a test sequence in which the macro rule r_mA fires; C_{np} is satisfied, for example, by the test suite $TS_A = \{ts_A\} = \{[(gA = 0, currAsm = Ma), (gA = 0, currAsm = Ma)]\}$. In the machine Mb, C_{np} can be satisfied if there exists a test sequence in which the macro rule r_mB fires; it is satisfied, for example, by the test suite $TS_B = \{ts_B\} = \{[(gB = 0, currAsm = Mb), (gB = 1, currAsm = Ma)]\}$[2]. The test sequence graph obtained from test suites TS_A and TS_B is shown in Fig. 2.

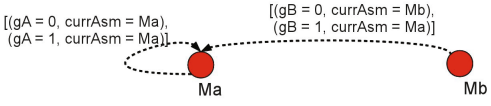


Fig. 2. Test sequence graph obtained with the criterion C_{np} over Ma and Mb

The test suite obtained from the visit of the test sequence graph is $TS_{AB} = \{ts_A\} = \{[(gA = 0, gB = 345, currAsm = Ma), (gA = 0, gB = 7, currAsm = Ma)]\}$, where the values of gB are randomly chosen. In the product machine *ProductMaMb*, shown in Code 7, C_{np} is not satisfied using the test suite TS_{AB}, since macro rule r_mB never fires.

Nevertheless, it is possible to build a test suite that satisfies the criterion C_{np} in *ProductMaMb*, such as $TS_P = \{[(gA = 1, gB = 235, currAsm = Ma), (gA = 456, gB = 1, currAsm = Mb), (gA = 73, gB = 3, currAsm = Mb)]\}$.

Another limit of our approach is that the model checker may fail to find any test sequence that reaches one machine, although such sequence would be required by the (preservable) criterion. This may happen, for instance, because of the state explosion problem in a single machine. Of course, if this case occurs, it would be even more likely that the model checker would fail on the product machine as well.

The assumption that the machines do not share information limits the applicability of our technique. It can be applied only if the different sub-systems modeled by different ASMs either do not share any information or share information that does not influence the behavior of the machines. For instance, in the case study application of Sect. 3.1, all the pages share the username (which is shown in the web pages) and the session information, which, however, do not appear in the ASMs since they do not influence the behavior. If the web pages shared behavioral information, then our approach would not be applicable. We plan in the future to introduce in sequential nets of ASMs also a way for the machines to share information.

[2] Any not empty test suite (with any value for monitored functions gA and gB) satisfies the criterion over machines Ma and Mb because the execution of macro rules r_mA and r_mB does not depend on the evaluation of any guard.

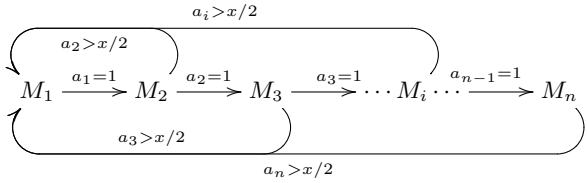


Fig. 3. The sequential net of ASMs for the combination lock problem

5 Initial Experiment

In order to evaluate our approach, we have experimented it with a small system. It resembles the combination lock finite state machine [13], for which generating a transition covering test suite becomes exponentially expensive. The problem is that of discovering the key of an electronic combination lock made of n digits having values from 1 to x. We have modeled the system as a sequential net of ASMs (see Fig. 3). The net is composed of n machines; every machine M_i has a monitored function a_i in the range $[1, x]$. If a_i (with $i = 1, \ldots, n-1$) takes the specific value 1 then the next machine M_{i+1} becomes active; if a_j (with $j = 2, \ldots, n$) becomes greater than $x/2$ then the system goes back to machine M_1, otherwise the machine M_j remains active.

We have evaluated our method depending on the number of digits (machines) n and/or the base x (the cardinality of the codomain of functions a_i).

For each combination of n and x we have built n single machines, where each machine has nx states since the signature of each machine M_i is composed of two 0-ary functions, a_i and $currAsm$, whose codomain sizes are, respectively, x and n. Then we have built the unique product machine that has nx^n states, since there are n 0-ary functions whose codomain size is x, and a 0-ary function whose codomain size is n.

Then we have generated the test sequences both for the product machine and for the sequential net of machines by the method introduced in this paper. As expected, we discovered that it is easier to execute n times the model checker over the single machines rather than executing the model checker one time over the product machine. The results of the experiment are shown in Fig. 4; the dependence between the execution time and the number of single machines n is reported. If the single machines are used, the execution time grows linearly with the number of machines; if the product machine is used, instead, the execution time grows exponentially with the number of machines. We made several experiments with different values for x (the cardinality of the codomain of functions a_i); as expected, in the product machine the value of x influences the execution time (even for small changes of x), whereas in the single machines it is irrelevant. We report the experiments made with the product machine with x equal to 10, 20 and 50, and the experiment made with the single machines with x equal to 50. We set a time limit of 1 hour for each experiment setting. All the experiments

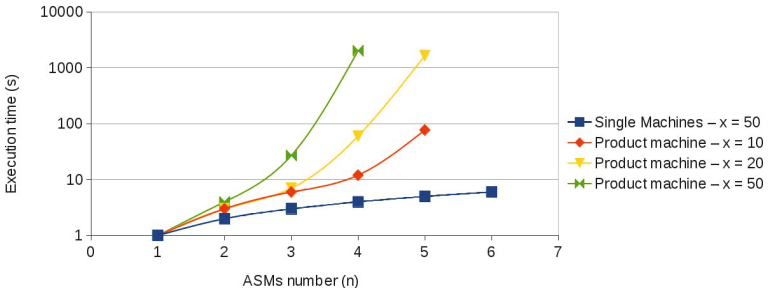


Fig. 4. Model checker executions times (sec.)

were executed on a Linux PC with 8 Intel(R) Xeon(R) CPUs E5430 @ 2.66GHz and 8 GB of RAM.

Test Suite Sizes. In Table 1 we report the sizes of the test suites obtained using the sequential net method and the product machine method. We report the sizes obtained with different number of machines; we do not report the value of x because it does not influence the test suite size.

Table 1. Test suite size

# ASMs	1	2	3	4	5	6
Sequential net	3	5	7	9	11	13
Product machine	3	7	11	15	19	n/a

From our experiments it seems that the test suites derived from the test sequence graph are smaller than those obtained directly from the product machine. However, we must notice that this can not be taken as a general law; we plan to do additional experiments to define more clearly the relationship between the sizes of the test suites obtained with the two methods.

Code Coverage. As sanity check, we measured also the code coverage obtained by using the two methods. We implemented the system, previously specified in ASM, into Java and translated the test suites in JUnit. We obtained the same code (statement and branch) coverage by using both the test sequences generated from the product ASM and from the sequential net.

6 Model-Based Testing of Web-Based Applications

We have studied the test generation for sequential nets of ASMs in the context of MBT of web-based applications [6]. In this context, every machine represents a single page of the application. The main purpose is to automatically generate test cases for web applications using a model-based approach. This is accomplished by first creating an ASM for each web page of the web application; in this scenario the *AsmDomain* can be interpreted as the set of web pages and the *currAsm*

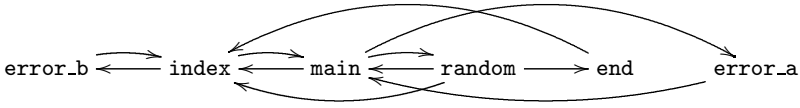


Fig. 5. Web-based application case study - Sequential net

function as the current active page. The methodology introduced in Sect. 4 is applied to obtain a test suite for the whole web application; finally, each test sequence can be mapped to a SAHI script [1] to exercise the tests directly on the web application. In the following we report the experiment made with the case study described in Sect. 3.1.

Modeling Every Page with an ASM. The first idea was modeling the complete web application with a single ASM. The model construction was feasible but the model checking was not able to complete the test generation. So, we modeled the web application using a sequential net of ASMs where every page is represented by an ASM and the domain *AsmDomain* is composed by the web pages. The obtained sequential net is shown in Fig. 5.

For translating a web page behavior into an ASM, we have put on a table the inputs of the web page (e.g., the values of the text fields) and identified, for every combination of inputs, a transition to another page or a set of state updates. In this way we have built an ASM for each web page.

Test Generation. For the test generation we have used, as described in Sect. 4.1, the ATGT tool over each ASM, using as coverage criteria all those described in [9].

Test Sequence Graph Construction. Then we have built the test sequence graph (see Fig. 6) as described in Sect. 4.2. Each transition of the sequential net has been covered in the test sequence graph.

Table 2 reports, for each ASM, the number of test sequences, divided between *inner* and *exiting*.

Table 2. Test sequences number

	index	error_b	main	error_a	random	end
# tests	24	3	36	3	45	2
# inner - # exiting	18 - 6	1 - 2	26 - 10	1 - 2	32 -13	1 - 1

Test Sequence Combination. Then, we have applied the technique presented in Sect. 4.3 in order to obtain a single test suite for the whole web application. The obtained test suite contains 212 test sequences and it satisfies all the coverage criteria used to generate the test suites over the single machines.

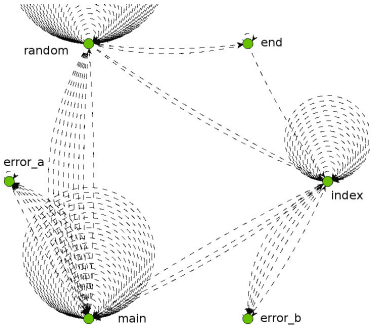


Fig. 6. Web-based application case study - Test sequence graph

Test of the Web Application. Finally, each test sequence of the test suite has been automatically mapped to a SAHI script; the execution of all the scripts has permitted us to test all the aspects of the web application. Code 8 shows one of the produced SAHI scripts.

```
_navigateTo("index.php");
_setValue(_textbox("username"),"admin");
_setValue(_textbox("password"),"pwd");
_click(_submit("submit"));
_click(_checkbox("agree"));
_setValue(_textarea("text"),"someText");
```

Code 8. SAHI script example.

7 Related Work

Our approach tries to mitigate the state space explosion problem during model checking for test generation. Traditionally several techniques attempt to solve the same problem for the verification of properties. They share the concept of building an abstract version of the original system that preserves properties.

The *cone of influence* (coi) technique [5] reduces the size of the transition graph by removing from the model the variables that do not influence the variables in the property one wants to check. In [14] the cone of influence technique is used to reduce the state space of *fFSM* models, a variant of Harel's Statecharts; models that could not be verified before, have been verified successfully after its application. The *data abstraction* technique [5], instead, consists of creating a mapping between the data values and a small set of abstract data values; the mapping, extended to states and transitions, usually reduces the state space, but it may not preserve properties. In [4] a technique to iteratively refine an abstract model is presented. The technique assures that, if a property is true in the abstract model, so it is in the initial model; if it is false in the abstract model, instead, the *spurious* counterexample may be the result of some behavior in the abstract model not present in the original model. The counterexample itself is used to refine the abstraction so that the *wrong* behavior is eliminated.

For test generation, these techniques may need to be modified, since they do not have to preserve properties but counterexamples to be used as tests. The coi

technique can be used as it is also for test generation, but it may not simplify our models, since the *currentAsm* function, which is used in the test goals, may be influenced by all the functions.

In [2] a web application is modeled by means of FSMs. They also face the state explosion problem; they try to overcome it by partitioning a web application into clusters that can contain web pages and other clusters. For each cluster an FSM is built; an *Application FSM* represents the entire application. Test sequences are derived from single FSMs. They share with us the need of decomposing the model into smaller models in order to keep the state space size tractable. As we do, they provide a technique for combining test sequences obtained from the single FSMs into a test suite to be used for testing the whole web application. The technique they propose also permits to propagate inputs among FSMs, while, in our approach, we currently do not allow the ASMs to exchange any information.

8 Future Work and Conclusion

We have tried to address the state explosion problem in test generation by model checking. For sequential nets of ASMs, our approach makes the test generation more scalable, without reducing the coverage obtained by the tests. Initial experiments show that our approach provides excellent benefits. We plan to extend the model of ASM nets by considering cases in which a single machine has several initial states and the machines share some locations. In such case, we believe that the test generation can not be done in advance for all the machines, but the construction and the visit of the graph must be done together.

We assume that the designer keeps the models separated from the beginning; as future work, we plan to study a methodology able, if possible, to split an existing complex ASM in a sequential net of ASMs.

Although our method shows its great usefulness when used in combination with (explicit state) model checking for test generation, we believe that any test generation technique can benefit from dividing the model in sub-models, even those techniques which do not suffer so much from the size of the model under test.

References

1. Sahi website, `http://sahi.co.in/`
2. Andrews, A.A., Offutt, J., Alexander, R.T.: Testing Web applications by modeling with FSMs. Software and Systems Modeling 4, 326–345 (2005)
3. Börger, E., Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer (2003)
4. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM 50, 752–794 (2003)
5. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (1999)
6. Di Lucca, G.A., Fasolino, A.R.: Testing Web-based applications: The state of the art and future trends. Inf. Softw. Technol. 48, 1172–1186 (2006)
7. Fraser, G., Gargantini, A.: An evaluation of model checkers for specification based test case generation. In: ICST 2009, Denver, Colorado, USA, April 1-4, pp. 41–50. IEEE Computer Society (2009)

8. Gargantini, A., Riccobene, E.: ASM-Based Testing: Coverage Criteria and Automatic Test Sequence Generation. J.UCS 7, 262–265 (2001)
9. Gargantini, A., Riccobene, E., Rinzivillo, S.: Using Spin to Generate Testsfrom ASM Specifications. In: Börger, E., Gargantini, A., Riccobene, E. (eds.) ASM 2003. LNCS, vol. 2589, pp. 263–277. Springer, Heidelberg (2003)
10. Hierons, R., Derrick, J.: Editorial: special issue on specification-based testing. Software Testing, Verification and Reliability 10(4), 201–202 (2000)
11. Holzmann, G.: Spin model checker, the: primer and reference manual, 1st edn. Addison-Wesley (2003)
12. Memon, A.M., Akinmade, O.: Automated Model-Based Testing of Web Applications. In: Google Test Automation Conference 2008 (2008)
13. Moore, E.F.: Gedanken experiments on sequential machines. In: Automata Studies, Princeton, pp. 129–153 (1956)
14. Park, S., Kwon, G.: Avoidance of State Explosion Using Dependency Analysis in Model Checking Control Flow Model. In: Gavrilova, M.L., Gervasi, O., Kumar, V., Tan, C.J.K., Taniar, D., Laganá, A., Mun, Y., Choo, H. (eds.) ICCSA 2006. LNCS, vol. 3984, pp. 905–911. Springer, Heidelberg (2006)
15. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan-Kaufmann (2006)

ASM and Controller Synthesis

Richard Banach[1,⋆], Huibiao Zhu[2,⋆⋆], Wen Su[2], and Xiaofeng Wu[2]

[1] School of Computer Science, University of Manchester,
Oxford Road, Manchester, M13 9PL, U.K.
banach@cs.man.ac.uk
[2] Software Engineering Institute, East China Normal University,
3663 Zhongshan Road North, Shanghai 200062, P.R. China
{hbzhu,wensu,xfwu}@sei.ecnu.edu.cn

Abstract. While many systems are naturally viewed as the interaction between a controller subsystem and a controlled, or plant subsystem, they are often most easily understood and designed monolithically. A practical implementation needs to separate controller from plant. We study the problem of when a monolithic ASM system can be split into controller and plant subsystems along syntactic lines derived from variables' natural affiliations. We give restrictions that enable the split to be carried out cleanly, and we give conditions that ensure that the resulting pair of controller and plant subsystems have the same behaviours as the original design. We illustrate the theory with a case study concerning eating with chopsticks. This leads to an extension of controller synthesis for continuous ASM systems, which are briefly covered. The case study is then extended into the continuous sphere.

1 Introduction

Today, when one considers the ubiquity of embedded controllers, which take on the digital role in the interaction of a digital and an external system, it becomes clear that many systems are naturally viewed as the interaction between a controller subsystem and a controlled, or plant subsystem. Such systems are often most easily and conveniently understood and designed monolithically — this allows the bulk of the design activity to focus on the overall system goals rather than lower level detail. However, a practical implementation needs to separate the controller from the plant, since it is the controller which behaves according to a human-created digital design, and the plant behaves according to patterns determined by the laws of nature. In this paper we study the problem of when a monolithic ASM system design, embodying this dual controller/plant nature, can be split into separate controller and plant subsystems along generic syntactic lines derived from the most natural associations of the system variables to one or other subsystem. This requires that the monolithic design satisfies some simple criteria *ab initio*.

⋆ The majority of the work reported in this paper was done while the first author was a visiting researcher at the Software Engineering Institute at East China Normal University. The support of ECNU is gratefully acknowledged.

⋆⋆ Huibiao Zhu is supported by National Basic Research Program of China (No. 2011CB302904), National High Technology Research and Development Program of China (No. 2011AA010101 and No. 2012AA011205), National Natural Science Foundation of China (No. 61061130541 and No. 61021004).

J. Derrick et al. (Eds.): ABZ 2012, LNCS 7316, pp. 51–64, 2012.

The rest of the paper is as follows. Section 2 describes the controller synthesis problem in abstract terms, focusing on the specific way that controller and plant are to be separated. A sufficient condition for the desired controller/plant separation is formulated and proved. The undecidability of controller synthesis is also briefly discussed by reduction to the Halting Problem in Section 2.1. In Section 3 we consider a computable subset of the controller synthesis problem and argue that it is adequate for practical purposes. Section 4 discusses an example based on the idea of picking up food with chopsticks, viewed as a control problem. Section 5 extrapolates the preceding ideas to the case of continuous ASM, in which smoothly changing (as well as discretely changing) behaviours are admitted. Section 6 extends the discussion of the chopsticks case study by taking on board the continuous notions. Section 7 concludes.

2 The Controller Synthesis Problem

We consider a generic ASM system consisting of basic ASM rules using straightforward single variable locations and a simple element of nondeterminism. Following [2], for our purposes, such a rule can be written as:

$$\begin{array}{l} \textsc{Op}(pars) \ = \\ \textbf{if} \ \ guard(xs, pars) \ \ \textbf{then choose} \ \ xs' \ \ \textbf{with} \ \ rel(xs', xs, params) \\ \quad \textbf{do} \ \ xs \ := \ xs' \end{array} \tag{1}$$

In (1), $pars$ are the input parameters (as needed) and xs are the variables modified by the rule. The rule's guard is $guard$, and rel represents the relationship that is to hold between the parameters, the before-values of the variables xs, and their after-values referred to as xs', when the rule fires. As usual, in a single step of a run of the system, all rules which are enabled (i.e. their guards are true) fire simultaneously, provided that the totality of updates defined thereby is consistent, else the run aborts.

In this paper we are interested in control applications, and we envisage the design done in a monolithic way at the outset, addressing system-wide design goals before plunging into the details of subsystem design. Thus the design may start by being expressed using system-wide variables. However, by a process of gradual refinement, the collection of variables will eventually end up such that each variable can be identified as belonging to either the controller-subsystem-to-be, or the plant-subsystem-to-be. Nevertheless, a legacy of the top-down design process is that many, or even all of the rules will still involve variables of both kinds.

The controller synthesis problem is the problem of taking such a collection of rules (call it Sys), and separating it into one set of rules for the controller (call it Con) and another set for the plant (call it Pla), each reading only the variables accessible to it, and each modifying only its own variables, such that the combination of the rules in Con and Pla generates the same behaviour (i.e. the same set of runs) as the original ruleset Sys.[1]

[1] In [2], the importance of distinguishing *controlled* functions from *monitored* ones is stressed, in a sense solving the controller synthesis problem right at the outset since the distinction already separates the controller from the plant. Our perspective is slightly different however, since it permits this aspect to be ignored for a portion of the development, and asks under what conditions the separation can be done later in a systematic way.

We perform the separation in a systematic manner. We assume that the variables Var of Sys can be partitioned into $xs_C \subseteq Var_C$, the variables for which the controller has write access, and $xs_P \subseteq Var_P$, the variables for which the plant has write access, with $Var_C \cap Var_P = \emptyset$. We assume that for each rule $\text{OP}(params) \in Sys$, the guard can be written in the form $guard(xs, pars) \equiv guard_C(xs_C, xs_P^c, pars_C) \wedge guard_P(xs_P, xs_C^p, pars_P)$, where xs_P^c are the plant variables to which the controller has read access, and xs_C^p are the controller variables to which the plant has read access. We also assume that for each rule, $rel(xs', xs, pars)$ can be written in the form $rel(xs', xs, pars) \equiv rel_C(xs_C, xs_P^c, pars_C) \wedge rel_P(xs_P, xs_C^p, pars_P)$. We say that a system is **admissible** iff the above hold.

Under the above assumptions, the desired construction is relatively clear. For each rule like (1) in Sys, we generate two fresh rules:

$$\begin{array}{l} \text{OP}_C(pars) \ = \\ \textbf{if } guard_C(xs_C, xs_P^c, pars_C) \textbf{ then choose } xs'_C \\ \quad \textbf{with } rel_C(xs'_C, xs_C, xs_P^c, pars_C) \textbf{ do } xs_C := xs'_C \end{array} \tag{2}$$

$$\begin{array}{l} \text{OP}_P(pars) \ = \\ \textbf{if } guard_P(xs_P, xs_C^p, pars_P) \textbf{ then choose } xs'_P \\ \quad \textbf{with } rel_P(xs'_P, xs_P, xs_C^p, pars_P) \textbf{ do } xs_P := xs'_P \end{array} \tag{3}$$

Of these, (2) goes into Con and (3) goes into Pla.

With Con and Pla thus constructed, and with initial states correspondingly constructed by restricting the initial states of Sys to the variables in Var_C and Var_P respectively (by existentially quantifying out $Var - Var_P$ in Con, and $Var - Var_C$ in Pla, provided there are no non-trivial joint initial properties), it is evident that whenever a rule OP of Sys is enabled, the corresponding rules OP_C and OP_P of Sys_C and Sys_P will also be enabled (since their guards are just weakenings of OP's guard). If we thus consider the system Sys_{C+P}, which consists of the variables and initial states of Sys,[2] and whose rules are the union of the OP_C and OP_P rules, then whenever a rule OP of Sys is enabled, it follows that in Sys_{C+P}, OP_C and OP_P will be enabled and both will be scheduled simultaneously by the ASM scheduling policy, replicating the update performed by OP in Sys. So the runs of Sys are a subset of the runs of Sys_{C+P}.

On the other hand, they may be a *proper* subset since the guards of the individual OP_C and OP_P rules are weaker than the guard of OP, and so may enable one or other of OP_C and OP_P without the other being enabled. This is highly undesirable from a requirements point of view since the overall objective was to achieve the behaviour of Sys, and not to introduce some spurious additional behaviours.

Definition 1. *A system Sys, with $Var = Var_C \uplus Var_P$ which is admissible, has a resolvable controller synthesis problem iff, after the construction above, the runs of Sys_{C+P} are exactly the runs of Sys.*

[2] The initial states are recovered by conjoining initial states of Sys_C and Sys_P.

Theorem 1. *Suppose a system Sys is admissible. Then Sys has a resolvable controller synthesis problem if:*

$$\begin{array}{l}\textit{For all rules } \textsc{Op}, \textit{ their derived rules } \textsc{Op}_C \textit{ and } \textsc{Op}_P, \textit{ and reachable states } xs \bullet \\ \quad [\ Domain(xs) \wedge guard_C(xs_C, xs_P^c, pars_C) \Rightarrow guard(xs, pars)\ ] \wedge \\ \quad [\ Domain(xs) \wedge guard_P(xs_P, xs_C^p, pars_P) \Rightarrow guard(xs, pars)\ ] \end{array} \tag{4}$$

where $Domain(xs)$ is the domain theory for the development of Sys.

Proof: To get the result, it is sufficient to show that when (4) holds, every run of Sys_{C+P} is a run of Sys, since we argued above that all Sys runs are Sys_{C+P} runs anyway. We proceed by induction on the length of the run. The base case is trivial since the initial states of Sys and of Sys_{C+P} are identical. Suppose then that we have the result for all Sys_{C+P} runs of length n or less. Choose a run rr of length n which is extendable. This means that there is some rule, $\textsc{Op}_C$ say, that is enabled in the final state xs reached by rr (the argument is symmetrical if it is $\textsc{Op}_P$ that is enabled). Since $\textsc{Op}_C$ is enabled in xs, $guard_C$ holds, whence $guard$ holds by (4). Since $guard_P$ weakens $guard$, $guard_P$ holds, whence $\textsc{Op}_P$ is enabled. Since both $\textsc{Op}_C$ and $\textsc{Op}_P$ are enabled, the update of Sys is emulated by Sys_{C+P} in the next step of the run. The same argument applies for all rules of Sys_{C+P} enabled in xs, so that the next Sys_{C+P} step from rr exactly mirrors a corresponding step of Sys. Doing the same for all possible ways of extending all extendable runs of length n completes the inductive step. □

2.1 Undecidability of Controller Synthesis

The presence of reachability in (4) makes the undecidability of the controller synthesis problem relatively unsurprising, so we just briefly sketch a reduction of the Halting Problem. Let TM be an arbitrary Turing Machine. Let TM_C^0 be an emulation of TM by an ASM constructed in a rather obvious way: i.e. there is an alphabet of states, another of tape symbols, a variable for the current state, a data structure for the tape, and a separate rule for each transition in the transition relation of TM. Let TM_P^0 be another such ASM emulation, isomorphic to TM_C^0, but with all alphabets and variables completely disjoint from those of TM_C^0. Consider the ASM TM_{C+P}^0 constructed as in the previous section. It has twice as many rules as TM has transitions, but they are enabled pairwise at exactly the same moments, so TM_{C+P}^0 just emulates two disjoint copies of TM running in lockstep. Consider the ASM $TM_{C\wedge P}^0$ constructed by fusing each corresponding pair of rules of TM_{C+P}^0 into a single rule by conjoining the guards, and combining the updates. It has exactly as many rules as TM has transitions. $TM_{C\wedge P}^0$ and TM_{C+P}^0 are bisimilar to each other and to TM. Now we modify TM_C^0, and modify TM_P^0, as follows.

Since TM is arbitrary, it may contain halting before-configs —i.e. pairs (t, s) where t is a tape symbol and s is a state— from which no transition issues. If TM has a halting before-config (t, s), we do the following. Let (t_C, s_C) be the counterpart of (t, s) in TM_C^0. To TM_C^0 we add a rule that implements a self-loop guarded on (t_C, s_C) (without moving the tape head), getting TM_C. Let (t_P, s_P) be the counterpart of (t, s) in TM_P^0. To TM_P^0 we add a rule that implements a self-loop guarded on s_P alone (i.e. ignoring the tape symbol, and without moving the tape head), getting TM_P.

Now consider the two ASM systems $TM_{C \wedge P}$ and TM_{C+P}. In $TM_{C \wedge P}$ (which plays the role of Sys above), the stronger guard of the TM_C rule in effect subsumes the weaker one of the TM_P rule, and the fused rule is only enabled exactly when the TM_C rule is enabled. However in TM_{C+P} (which plays the role of Sys_{C+P} above), this is not the case. There, the TM_P rule exists independently, and if the computation of TM reaches a machine configuration in which the tape symbol and state are (t, s), then the TM_P rule is also enabled when the tape symbol and state are $(\tilde{t}, s)$, for some $\tilde{t} \neq t$, giving rise to behaviours not reflected in $TM_{C \wedge P}$.

3 Computable Controller Synthesis

Restricting to a safe approximation to reachability, we get a computable version of (4), which we argue will be adequate for all practical purposes.

Theorem 2. *Suppose a system Sys is admissible and XS is a set of states that includes all reachable states. Then Sys has a resolvable controller synthesis problem if:*

$$\begin{aligned} &\textit{For all rules } \textsc{Op}, \textit{ their derived rules } \textsc{Op}_C \textit{ and } \textsc{Op}_P, \textit{ and all } xs \in XS \bullet \\ &\quad [\, Domain(xs) \wedge guard_C(xs_C, xs_P^c, pars_C) \vdash guard(xs, pars) \,] \wedge \\ &\quad [\, Domain(xs) \wedge guard_P(xs_P, xs_C^p, pars_P) \vdash guard(xs, pars) \,] \end{aligned} \tag{5}$$

where $Domain(xs)$ is the domain theory for Sys and $\vdash$ is provability in a suitable system.

4 An Example: Eating with Chopsticks

We now look at a simple example of the preceding theory: eating food with chopsticks. Fig. 1 shows the forces involved in grasping a morsel of food with chopsticks.

4.1 Food and Chopsticks

In a statically stable situation, the chopsticks extert forces on the food, and the food exerts equal and opposite forces on the chopsticks. The forces exerted by the food are $\boldsymbol{f}_{\text{FU}}$ on the upper chopstick and $\boldsymbol{f}_{\text{FL}}$ on the lower chopstick. For simplicity we assume that these forces sum to zero (else the food would accelerate) and colinear.[3] Reacting to $\boldsymbol{f}_{\text{FU}}$ and $\boldsymbol{f}_{\text{FL}}$, the chopsticks exert their forces $\boldsymbol{f}_{\text{HCU}}$ and $\boldsymbol{f}_{\text{HCL}}$, equal and opposite to $\boldsymbol{f}_{\text{FU}}$ and $\boldsymbol{f}_{\text{FL}}$. So we have:

$$\boldsymbol{f}_{\text{FU}} + \boldsymbol{f}_{\text{FL}} = \mathbf{0} \tag{6}$$

$$\boldsymbol{f}_{\text{HCU}} + \boldsymbol{f}_{\text{HCL}} = \mathbf{0} \tag{7}$$

$$\boldsymbol{f}_{\text{FU}} + \boldsymbol{f}_{\text{HCU}} = \mathbf{0} \tag{8}$$

[3] In reality, slight deviations from colinearity are compensated for by forces of friction and deformation arising from the food, aided where appropriate, by surface tension forces coming from any sauce that the food might be prepared in.

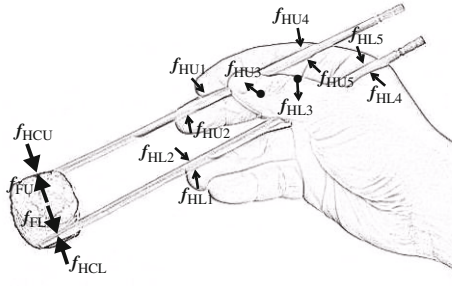


Fig. 1. Forces involved in grasping a piece of food with chopsticks

$$\boldsymbol{f}_{\mathrm{FL}} + \boldsymbol{f}_{\mathrm{HCL}} = \mathbf{0} \tag{9}$$

$$|\boldsymbol{f}_{\mathrm{FU}}| = |\boldsymbol{f}_{\mathrm{FL}}| = |\boldsymbol{f}_{\mathrm{HCU}}| = |\boldsymbol{f}_{\mathrm{HCL}}| \geq D \tag{10}$$

The last of these (10), expresses a constraint that the forces mentioned have to be large enough (D) that they generate additional frictional forces (which can be taken to be proportional to them), sufficient to counteract gravity (which we have not taken into account), thereby to stop the food from dislodging from the chopsticks when lifted.

We can write this as an ASM model, with a rule:

$$\begin{array}{l}\textsc{GraspFood} = \\ \textbf{choose } \boldsymbol{f}'_{\mathrm{FU}}, \boldsymbol{f}'_{\mathrm{FL}}, \boldsymbol{f}'_{\mathrm{HCU}}, \boldsymbol{f}'_{\mathrm{HCL}} \\ \quad \textbf{with } \boldsymbol{f}'_{\mathrm{FU}} + \boldsymbol{f}'_{\mathrm{FL}} = \boldsymbol{f}'_{\mathrm{HCU}} + \boldsymbol{f}'_{\mathrm{HCL}} = \boldsymbol{f}'_{\mathrm{FU}} + \boldsymbol{f}'_{\mathrm{HCU}} = \boldsymbol{f}'_{\mathrm{FL}} + \boldsymbol{f}'_{\mathrm{HCL}} = \mathbf{0} \wedge \\ \qquad |\boldsymbol{f}'_{\mathrm{FU}}| = |\boldsymbol{f}'_{\mathrm{FL}}| = |\boldsymbol{f}'_{\mathrm{HCU}}| = |\boldsymbol{f}'_{\mathrm{HCL}}| \geq D \\ \quad \textbf{do } \boldsymbol{f}_{\mathrm{FU}} := \boldsymbol{f}'_{\mathrm{FU}}, \boldsymbol{f}_{\mathrm{FL}} := \boldsymbol{f}'_{\mathrm{FL}}, \boldsymbol{f}_{\mathrm{HCU}} := \boldsymbol{f}'_{\mathrm{HCU}}, \boldsymbol{f}_{\mathrm{HCL}} := \boldsymbol{f}'_{\mathrm{HCL}}, \\ \qquad grasped := \textsc{true}\end{array} \tag{11}$$

There will be another rule DISLODGEFOOD, differing from (11) in the replacement of '$\geq D$' by '$< D$' and of TRUE by FALSE, regarding dislodgement of food as being due to inadequate force, and disregarding any other maladroitness on the part of the user. Given the similarity of the two rules, we will not mention DISLODGEFOOD further, unless it is unavoidable.

We can regard GRASPFOOD (and DISLODGEFOOD) as a simple design for a control system — the chopsticks are intended to control the food by grasping it. Thus we can pursue our earlier strategy by separating the system into plant (food) and controller (chopsticks) subsystems. The GRASPFOOD rule separates into $\textsc{GraspFood}_C$ and $\textsc{GraspFood}_P$:

$$\begin{array}{l}\textsc{GraspFood}_C = \\ \textbf{choose } \boldsymbol{f}'_{\mathrm{HCU}}, \boldsymbol{f}'_{\mathrm{HCL}} \\ \quad \textbf{with } \boldsymbol{f}'_{\mathrm{HCU}} + \boldsymbol{f}'_{\mathrm{HCL}} = \mathbf{0} \wedge |\boldsymbol{f}'_{\mathrm{HCU}}| = |\boldsymbol{f}'_{\mathrm{HCL}}| \geq D \\ \quad \textbf{do } \boldsymbol{f}_{\mathrm{HCU}} := \boldsymbol{f}'_{\mathrm{HCU}}, \boldsymbol{f}_{\mathrm{HCL}} := \boldsymbol{f}'_{\mathrm{HCL}}, \\ \qquad grasped := \textsc{true}\end{array} \tag{12}$$

$$\begin{array}{l}\textsc{GraspFood}_P \;= \\ \textbf{choose}\ \boldsymbol{f}'_{\text{FU}}, \boldsymbol{f}'_{\text{FL}} \\ \quad \textbf{with}\ \boldsymbol{f}'_{\text{FU}} + \boldsymbol{f}'_{\text{FL}} = \mathbf{0} \\ \quad \textbf{do}\ \boldsymbol{f}_{\text{FU}} := \boldsymbol{f}'_{\text{FU}}, \boldsymbol{f}_{\text{FL}} := \boldsymbol{f}'_{\text{FL}}\end{array} \tag{13}$$

In (12) and (13) we see that $\textsc{GraspFood}_C$ only 'owns' $\boldsymbol{f}_{\text{HCU}}$ and $\boldsymbol{f}_{\text{HCL}}$, so only assigns to those variables, and $\textsc{GraspFood}_P$ only 'owns' $\boldsymbol{f}_{\text{FU}}$ and $\boldsymbol{f}_{\text{FL}}$, so only assigns to them. We also observe that some pieces of GraspFood are not present in either $\textsc{GraspFood}_C$ or $\textsc{GraspFood}_P$, namely the terms that relate the food forces to the chopstick forces. This is explained by the observation that the relevant equations are part of the domain theory of statics: action and reaction are *always* equal statically, by Newton's Law. Additionally, that successful grasping needs adequate force is also part of the domain, so we can write:

$$\begin{aligned} Domain_{FHC} \;\equiv\; & \boldsymbol{f}_{\text{FU}} + \boldsymbol{f}_{\text{HCU}} = \mathbf{0} \wedge \boldsymbol{f}_{\text{FL}} + \boldsymbol{f}_{\text{HCL}} = \mathbf{0} \wedge \\ & (grasped = \text{TRUE} \Leftrightarrow |\boldsymbol{f}_{\text{HCL}}| \geq D) \end{aligned} \tag{14}$$

Now, in the context of (14), it is easy to see that:

$$Domain_{FHC} \wedge guard_{\textsc{GraspFood}_C} \vdash guard_{\textsc{GraspFood}} \tag{15}$$

$$Domain_{FHC} \wedge guard_{\textsc{GraspFood}_P} \vdash guard_{\textsc{GraspFood}} \tag{16}$$

4.2 Chopsticks and Hand

The preceding was rather elementary. In particular, it presumed that chopsticks somehow grasp food by themselves, which is silly. In reality, chopsticks are held in the right hand, which causes them to exert the forces spoken of previously. We now enrich our model by considering the hand-chopstick system as a further control system, and decomposing it further into a plant subsystem (the chopsticks themselves) and a controller subsystem (the hand).

We refer to Fig. 1 again. For a solid object to remain stable in 3D space, it needs to have four non-colinear forces summing to zero acting on it. If gravity is acting (as it normally is) then it supplies one force, and we derive the well-known fact that an object needs to be supported from underneath by three or more forces for stability.

This applies to the hand-chopstick system, where for simplicity, we can ignore gravity. Given how chopstick are disposed with respect to the hand, it is in fact convenient to view the hand as exerting five forces per chopstick. Fig. 1 shows the forces involved.

The middle of the lower chopstick is held steady on the ring finger. Typically it is gently wedged in the angle between the edge of the fingernail and the side of the fleshy pad of the fingertip, which we model by the forces $\boldsymbol{f}_{\text{HL1}}$ and $\boldsymbol{f}_{\text{HL2}}$ in Fig. 1. These are predominantly directed in the plane of the diagram, with a small component at right angles, out of the plane of the diagram, towards the reader. The back end of the lower chopstick is held on the fleshy part between the thumb and palm, and the forces are modeled by $\boldsymbol{f}_{\text{HL4}}$ and $\boldsymbol{f}_{\text{HL5}}$. Again these are mostly in the plane of the diagram, with a small component outwards, towards the reader. Opposing all the outwards components is $\boldsymbol{f}_{\text{HL3}}$ (the force drawn with the blob at its tail in Fig. 1), which is exerted by the lower end of the thumb, predominantly inwards into the diagram.

If the chopstick is merely being held steady, then these forces sum to zero. However, if food is being held, then the user adjusts the individual forces so that they sum to $\boldsymbol{f}_{\mathrm{HCL}}$:

$$\boldsymbol{f}_{\mathrm{HL1}} + \boldsymbol{f}_{\mathrm{HL2}} + \boldsymbol{f}_{\mathrm{HL3}} + \boldsymbol{f}_{\mathrm{HL4}} + \boldsymbol{f}_{\mathrm{HL5}} \;=\; \boldsymbol{f}_{\mathrm{HCL}} \tag{17}$$

The story for the upper chopstick is similar. The forces $\boldsymbol{f}_{\mathrm{HU1}}$ and $\boldsymbol{f}_{\mathrm{HU2}}$, formed by the more pronounced wedge between first and second fingers, serves to firmly hold and direct the middle of the chopstick in order to open and close the chopsticks for grasping food. Forces $\boldsymbol{f}_{\mathrm{HU4}}$ and $\boldsymbol{f}_{\mathrm{HU5}}$, exerted by the dip between the palm knuckle and first knuckle of the index finger, support the back of the chopstick. And vertical movement is restrained by $\boldsymbol{f}_{\mathrm{HU3}}$, once more indicated with a blob at its tail in Fig. 1, exerted by the upper part of the thumb. Again, if the chopstick is just being held steady, then these forces sum to zero. However, if food is being grasped, then they sum to $\boldsymbol{f}_{\mathrm{HCU}}$:

$$\boldsymbol{f}_{\mathrm{HU1}} + \boldsymbol{f}_{\mathrm{HU2}} + \boldsymbol{f}_{\mathrm{HU3}} + \boldsymbol{f}_{\mathrm{HU4}} + \boldsymbol{f}_{\mathrm{HU5}} \;=\; \boldsymbol{f}_{\mathrm{HCU}} \tag{18}$$

(N.B. In reality, many guides to eating with chopsticks recommend all sorts of alternative configurations for holding chopsticks (see eg. [3]), but the configuration described here is the only one that the first author has found to permit both adequate chopstick maneuvrability and sufficient deployable resultant force, especially when it comes to bigger pieces of food.)

With these observation, we can decompose the $\textsc{GraspFood}_C$ function into its plant and controller subsystems, rules $\textsc{Chopstick}_P$ and $\textsc{Hand}_C$.

In those rules, we have singled out $\boldsymbol{f}_{\mathrm{CU}}$ and $\boldsymbol{f}_{\mathrm{CL}}$ as output parameters in the signature of $\textsc{Hand}_C$ for emphasis. They are quantities derived from the underlying hand forces, which the chopsticks react to by setting their forces appropriately. The equalities $\boldsymbol{f}_{\mathrm{HCU}} = \boldsymbol{f}_{\mathrm{CU}}$ and $\boldsymbol{f}_{\mathrm{HCL}} = \boldsymbol{f}_{\mathrm{CL}}$ again become part of the domain theory of statics.

$$\begin{array}{l}
\textsc{Chopstick}_P \;= \\
\textbf{choose}\;\; \boldsymbol{f}'_{\mathrm{HCU}}, \boldsymbol{f}'_{\mathrm{HCL}} \\
\quad \textbf{with}\;\; \boldsymbol{f}'_{\mathrm{HCU}} + \boldsymbol{f}'_{\mathrm{HCL}} = \mathbf{0} \\
\quad \textbf{do}\;\; \boldsymbol{f}_{\mathrm{HCU}} := \boldsymbol{f}'_{\mathrm{HCU}}, \boldsymbol{f}_{\mathrm{HCL}} := \boldsymbol{f}'_{\mathrm{HCL}}
\end{array} \tag{19}$$

$$\begin{array}{l}
\textsc{Hand}_C(\textbf{out}\; \boldsymbol{f}_{\mathrm{CU}}, \boldsymbol{f}_{\mathrm{CL}}) \;= \\
\textbf{choose}\;\; \boldsymbol{f}'_{\mathrm{HU1}}, \boldsymbol{f}'_{\mathrm{HU2}}, \boldsymbol{f}'_{\mathrm{HU3}}, \boldsymbol{f}'_{\mathrm{HU4}}, \boldsymbol{f}'_{\mathrm{HU5}}, \boldsymbol{f}'_{\mathrm{HL1}}, \boldsymbol{f}'_{\mathrm{HL2}}, \boldsymbol{f}'_{\mathrm{HL3}}, \boldsymbol{f}'_{\mathrm{HL4}}, \boldsymbol{f}'_{\mathrm{HL5}} \\
\quad \textbf{with}\;\; \boldsymbol{f}'_{\mathrm{HU1}} + \boldsymbol{f}'_{\mathrm{HU2}} + \boldsymbol{f}'_{\mathrm{HU3}} + \boldsymbol{f}'_{\mathrm{HU4}} + \boldsymbol{f}'_{\mathrm{HU5}} + \\
\qquad \boldsymbol{f}'_{\mathrm{HL1}} + \boldsymbol{f}'_{\mathrm{HL2}} + \boldsymbol{f}'_{\mathrm{HL3}} + \boldsymbol{f}'_{\mathrm{HL4}} + \boldsymbol{f}'_{\mathrm{HL5}} = \mathbf{0} \\
\qquad |\boldsymbol{f}'_{\mathrm{HU1}} + \boldsymbol{f}'_{\mathrm{HU2}} + \boldsymbol{f}'_{\mathrm{HU3}} + \boldsymbol{f}'_{\mathrm{HU4}} + \boldsymbol{f}'_{\mathrm{HU5}}| \;= \\
\qquad |\boldsymbol{f}'_{\mathrm{HL1}} + \boldsymbol{f}'_{\mathrm{HL2}} + \boldsymbol{f}'_{\mathrm{HL3}} + \boldsymbol{f}'_{\mathrm{HL4}} + \boldsymbol{f}'_{\mathrm{HL5}}| \geq D \\
\quad \textbf{do}\;\; \boldsymbol{f}_{\mathrm{HU1}} := \boldsymbol{f}'_{\mathrm{HU1}} \ldots \boldsymbol{f}_{\mathrm{HU5}} := \boldsymbol{f}'_{\mathrm{HU5}}, \boldsymbol{f}_{\mathrm{HL1}} := \boldsymbol{f}'_{\mathrm{HL1}} \ldots \boldsymbol{f}_{\mathrm{HL5}} := \boldsymbol{f}'_{\mathrm{HL5}}, \\
\qquad \boldsymbol{f}_{\mathrm{CU}} := \boldsymbol{f}'_{\mathrm{HU1}} + \boldsymbol{f}'_{\mathrm{HU2}} + \boldsymbol{f}'_{\mathrm{HU3}} + \boldsymbol{f}'_{\mathrm{HU4}} + \boldsymbol{f}'_{\mathrm{HU5}}, \\
\qquad \boldsymbol{f}_{\mathrm{CL}} := \boldsymbol{f}'_{\mathrm{HL1}} + \boldsymbol{f}'_{\mathrm{HL2}} + \boldsymbol{f}'_{\mathrm{HL3}} + \boldsymbol{f}'_{\mathrm{HL4}} + \boldsymbol{f}'_{\mathrm{HL5}}, \\
\qquad grasped := \mathrm{TRUE}
\end{array} \tag{20}$$

5 Continuous Controller Synthesis

The reader may well have noticed that there are some slightly unnatural aspects of the account of chopstick use that we gave. The ASM rules in the preceding section were

the usual kind of discrete ASM rules. However, grasping via chopsticks is not the usual kind of discrete event control system. In particular, both the chopsticks and the food react instantaneously to the force exerted by the other, and not to the previous value maintained by the other, as one would expect in a normal discrete event control system. We handled this via the domain theory, which demanded that the opposed forces exactly matched, without giving any inkling as to how this might be accomplished.

In a more realistic account, the force applied by the chopsticks to the food moves smoothly from zero to a value sufficient to ensure grasping, and the food senses this and smoothly reacts by offering a matching resistive force. The sudden assignment to equal and opposite values in the discrete picture is replaced by a pair of differential equations which state that the derivatives of the chopstick and food forces are equal and opposite over time, which together with initial conditions stating that both are zero, guarantees that the forces themselves remain equal and opposite.

Incorporating these insights into the ASM framework requires an extension of ASM to include continuously varying behaviours as well as discrete changes. In [1] the authors give such an extension which we briefly recapitulate now.

5.1 Continuous ASM

We partition the variables into two subsets: the **mode variables**, whose types are discrete sets, and the **pliant variables**, whose types include topologically dense sets, and which are permitted to evolve both continuously and via discrete changes. By restricting to mode variables alone, we recover the conventional discrete ASM framework.

Time is modelled as an interval $\mathcal{T}$ of the real numbers $\mathbb{R}$, with a finite left endpoint for the initial state, and with a right endpoint which is finite or infinite, as needed. $\mathcal{T}$ partitions into a sequence of left-closed right-open intervals, $\langle [t_0 \ldots t_1), [t_1 \ldots t_2), \ldots \rangle$, the coarsest partition such that all discontinuous changes take place at some boundary point t_i. Mode variables are constant on each of these intervals, while pliant variables evolve continuously. Otherwise arbitrary continuous evolution is constrained within reasonable bounds by three main restrictions:

I Zeno: there is a constant δ_{Zeno}, such that for all i needed, $t_{i+1} - t_i \geq \delta_{\mathsf{Zeno}}$.

II Limits: for every variable x, for every time $t \in \mathcal{T}$, and with $\delta > 0$, the left limit $\lim_{\delta \to 0} x(t - \delta)$ written $\overrightarrow{x(t)}$ and right limit $\lim_{\delta \to 0} x(t + \delta)$, written $\overleftarrow{x(t)}$ exist, and for every t, $x(t) = \overleftarrow{x(t)}$.

III Differentiability: The behaviour of every pliant variable x in the interval $[t_i \ldots t_{i+1})$ is given by the solution of a well posed initial value problem $\mathcal{D}\, xs = \phi(xs, t)$ (where $\mathcal{D}$ is the time derivative).

The two kinds of variable (mode and pliant) are reflected in two kinds of transitions: mode and pliant. Mode transitions, given by rules of the form (21), just record discrete transitions from before-values to after-values of variables, with the use of the left limit for before-values and right limit for after-values making the semantics of these transitions instantaneous. Both kinds of variable can be subject to a mode transition, and in

(21), where we decorate the variables with this limit information, we single out inputs is and outputs os in the signature of OP.

$$\begin{array}{l}\textsc{Op}(\textbf{in } \overrightarrow{is}, \textbf{out } \overleftarrow{os}) \ = \\ \textbf{if } guard(\overrightarrow{xs}, \overrightarrow{is}) \textbf{ then choose } \overleftarrow{xs}, \overleftarrow{os} \textbf{ with } rel(\overleftarrow{xs}, \overrightarrow{xs}, \overrightarrow{is}, \overleftarrow{os}) \\ \quad \textbf{do } xs, os := \overleftarrow{xs}, \overleftarrow{os}\end{array} \tag{21}$$

Pliant transitions describe continuous changes for pliant variables. While a mode transition captures a single before-/after-value pair, a pliant transition is a family of before-/after-value pairs parameterized by the relevant time interval $[t_i \ldots t_{i+1})$. The before-value is, in each case, the value at t_i, while the after-value refers to an arbitrary time in the interval, so the two values are separated in time. A rule for a pliant transition can be written as in (22), where the symbol $\stackrel{\text{c}}{=}$ syntactically distinguishes a pliant transition from a mode transition.

$$\begin{array}{l}\textsc{PliOp}(\textbf{in } is(t \in (t_{\mathrm{L}(t)} \ldots t_{\mathrm{R}(t)})), \textbf{out } os(t \in (t_{\mathrm{L}(t)} \ldots t_{\mathrm{R}(t)}))) \ \stackrel{\text{c}}{=} \\ \textbf{if } IV(xs(t_{\mathrm{L}(t)})) \textbf{ and } guard(xs(t_{\mathrm{L}(t)})) \textbf{ then with } rel(xs, is, os, t) \\ \quad \textbf{do } xs(t), os(t) := \textbf{solve } DE(xs(t), is(t), os(t), t)\end{array} \tag{22}$$

In (22), $\mathrm{L}(t) = \max\{i \,|\, t_i \leq t\}$ and $\mathrm{R}(t) = \min\{i \,|\, t_i > t\}$ so that we do not have to statically know the index i for the interval $[t_i \ldots t_{i+1})$, thus making the notation generic. Furthermore, IV and $guard$ refer to the initial value and any additional guard restriction that apply for the initial value problem in $[t_i \ldots t_{i+1})$. DE is the differential equation of the initial value problem, while rel expresses any additional constraints that must hold beyond DE. Inputs is and outputs os (shown as depending on the whole interval $(t_{\mathrm{L}(t)} \ldots t_{\mathrm{R}(t)})$) again appear in the signature. If, as can often happen, we know the form of the continuous behaviour that we want (in contrast to merely knowing a differential equation for it), then we can replace the **solve** clause with a straightforward assignment using a **do**.

A continuous ASM ruleset, consisting of rules as we have described, is **well formed** iff the initial transition is a mode transition, every mode transition enables a pliant transition (but no mode transition), and every pliant transition (except perhaps for a final one) enables a mode transition (which, during runtime, preempts it).

Given a conventional discrete ASM system, we can rather trivially turn it into a continuous ASM system, as follows:

- consider the original discrete ASM rules as mode rules,
- decide on a fixed duration δ_t,
- determine that each state of the discrete event ASM system will persist for δ_t,
- add continuous ASM rules setting time derivatives of all ASM state variables to 0,
- add a time variable, and enable all mode transitions after integral multiples of δ_t.

5.2 Continuous Controller Synthesis

We can ask how the process of separating a set of rules into controller and plant rules goes, when we have pliant as well as mode transitions. In fact, the process is very similar to what went before. Since mode rules are identical to the rules we considered earlier, there is nothing new for them. For pliant rules, they also have a $guard$ and a rel, and for

these we demand the same conditions as previously. But there is also the **solve** clause. We need to stipulate that it separates cleanly into controller and plant in the same way that $guard$ and rel do so that the rule as a whole splits neatly.

The tuple of differential equations $\mathcal{D}\,xs = \phi(xs,t)$ contained in the **solve** clause naturally splits into two: $\mathcal{D}\,xs_C = \phi_C(xs,t)$ and $\mathcal{D}\,xs_P = \phi_P(xs,t)$. But there is no *a priori* guarantee that $\phi_C(xs,t)$ contains only the variables xs_C, xs_P^c, and $\phi_P(xs,t)$ contains only the variables xs_P, xs_C^p. So this is what we must additionally demand for admissibility.

It is clear that the embedding of discrete ASMs into continuous ASMs at the end of the last section is admissible in the extended sense just discussed, provided the original discrete ASM system is admissible, so that the properties derived for controller synthesis in Sections 2 and 3 carry through essentially unchanged.

6 Continuous Grasping

Let us revisit the chopsticks case study in the continuous ASM framework to see how the latter can lend it a more persuasive air.

As before, we restrict the modeling to that of forces only (albeit now allowing them to vary continuously). This avoids complications arising from having to consider movement of either the food or the chopsticks, or distortions of the shape of either the food or chopsticks consequent on them experiencing the forces that we model, and keeps the model within a relatively limited space.

We concentrate on elaborating the simpler model in Section 4.1. Time $t = 0$ triggers the intial mode rule:

$$\begin{array}{l}\textsc{Start} = \\ \textbf{if } t = 0 \textbf{ then} \\ \quad \textbf{do } mode := grasping, grasped := undef, \\ \qquad \boldsymbol{f}_{\text{FU}} := \mathbf{0}, \boldsymbol{f}_{\text{FL}} := \mathbf{0}, \boldsymbol{f}_{\text{HCU}} := \mathbf{0}, \boldsymbol{f}_{\text{HCL}} := \mathbf{0}\end{array} \tag{23}$$

The $grasping$ mode enables the following pliant rule:

$$\begin{array}{l}\textsc{Grasping} \stackrel{\text{c}}{=} \\ \textbf{if } mode = grasping \textbf{ then} \\ \quad \textbf{do } \boldsymbol{f}_{\text{FU}}, \boldsymbol{f}_{\text{FL}}, \boldsymbol{f}_{\text{HCU}}, \boldsymbol{f}_{\text{HCL}} := \\ \qquad \textbf{solve } [\, \mathcal{D}\,\boldsymbol{f}_{\text{FU}}, \mathcal{D}\,\boldsymbol{f}_{\text{FL}}, \mathcal{D}\,\boldsymbol{f}_{\text{HCU}}, \mathcal{D}\,\boldsymbol{f}_{\text{HCL}} \,] = [\, \mathbf{e}_{\text{z}}, -\mathbf{e}_{\text{z}}, -\mathbf{e}_{\text{z}}, \mathbf{e}_{\text{z}} \,]\end{array} \tag{24}$$

This rule causes the forces $\boldsymbol{f}_{\text{FU}}, \boldsymbol{f}_{\text{FL}}, \boldsymbol{f}_{\text{HCU}}, \boldsymbol{f}_{\text{HCL}}$ to acquire suitable pairwise equal and opposite rates of change, of magnitude 1, oriented along the unit vector of the z axis. This causes these forces to change continuously (although in fact non-smoothly[4]) away from zero at a uniform rate. The continuous grasping persists until a time t_{STOP}, when it is determined whether enough force has been applied to hold the food:

$$\begin{array}{l}\textsc{StopGrasped} = \textbf{if } t = t_{\text{STOP}} \wedge \boldsymbol{f}_{\text{HCU}} \geq D \textbf{ then} \\ \quad \textbf{do } mode := stop, grasped := \text{TRUE}\end{array} \tag{25}$$

[4] Since the derivatives of the forces jump discontinuously at $t = 0$, the forces themselves, though continuous, experience a kink at $t = 0$.

$$\text{STOPDISLODGED} \;=\; \textbf{if}\ t = t_{\text{STOP}} \wedge \boldsymbol{f}_{\text{HCU}} < D\ \textbf{then} \quad (26)$$
$$\quad \textbf{do}\ mode := stop, grasped := \text{FALSE}$$

The stopped mode just enters a pliant final state:

$$\text{F-IDLE} \stackrel{\text{c}}{=} \textbf{if}\ mode = stop\ \textbf{then do}\ \mathsf{skip} \quad (27)$$

The above is all consistent with the domain theory (14), although the theory would have to be augmented by various facts concerning time and the additional variables introduced above, in order that the natural continuous counterparts of the statements in (5) could hold.[5]

6.1 Decomposing Continuous Grasping

We now look at applying the decomposition strategy discussed earlier to the above integrated model. We asssume that the chopsticks, as controller, are in charge, and own variables like $mode$ and $grasped$. We decompose the rules above one by one, starting with START:

$$\text{START}_C \;= \quad (28)$$
$$\textbf{if}\ t = 0\ \textbf{then}$$
$$\quad \textbf{do}\ mode := grasping, grasped := undef, \boldsymbol{f}_{\text{HCU}} := 0, \boldsymbol{f}_{\text{HCL}} := 0$$

$$\text{START}_P \;= \quad (29)$$
$$\textbf{if}\ t = 0\ \textbf{then do}\ \boldsymbol{f}_{\text{FU}} := 0, \boldsymbol{f}_{\text{FL}} := 0$$

Next, the decomposition of the GRASPING rule. This yields:

$$\text{GRASPING}_C(\textbf{out}\ \boldsymbol{of}_{\text{HCU}}, \boldsymbol{of}_{\text{HCL}}) \stackrel{\text{c}}{=} \quad (30)$$
$$\textbf{if}\ mode = grasping\ \textbf{then}$$
$$\quad \textbf{do}\ \boldsymbol{f}_{\text{HCU}}, \boldsymbol{f}_{\text{HCL}} := \textbf{solve}\ [\ \mathcal{D}\,\boldsymbol{f}_{\text{HCU}}, \mathcal{D}\,\boldsymbol{f}_{\text{HCL}}\ ] = [\ -\mathbf{e}_z, \mathbf{e}_z\ ],$$
$$\qquad \boldsymbol{of}_{\text{HCU}} := \boldsymbol{f}_{\text{HCU}}, \boldsymbol{of}_{\text{HCL}} := \boldsymbol{f}_{\text{HCL}}$$

$$\text{GRASPING}_P(\textbf{in}\ \boldsymbol{if}_{\text{HCU}}, \boldsymbol{if}_{\text{HCL}}) \stackrel{\text{c}}{=} \quad (31)$$
$$\textbf{if}\ mode = grasping\ \textbf{then do}\ \boldsymbol{f}_{\text{FU}} := -\boldsymbol{if}_{\text{HCU}}, \boldsymbol{f}_{\text{FL}} := -\boldsymbol{if}_{\text{HCL}}$$

The above rules display a slightly more complex manner of decomposition than we have considered hitherto. Instead of merely partitioning the variables and determining that subsystem B has read access to some of the variables owned by subsystem A, we have introduced input and output variables that do this job explicitly. So the chopsticks have output variables $\boldsymbol{of}_{\text{HCU}}$ and $\boldsymbol{of}_{\text{HCL}}$, which are just copies of variables $\boldsymbol{f}_{\text{HCU}}$ and $\boldsymbol{f}_{\text{HCL}}$, and the food has input variables $\boldsymbol{if}_{\text{HCU}}$ and $\boldsymbol{if}_{\text{HCL}}$, which are used to read the relevant values in. Thus, the modeling is a now little different in that the food explicitly reacts to the forces it senses (by generating equal and opposite forces of its own) — we have substituted equals for equals, but have gone beyond the simple syntactic transformation described earlier in the paper. It is a natural temptation to do this at the more realistic and

[5] The domain theory would also have to be supplemented with a background theory of facts about calculus, continuous mathematics etc., as needed.

practical level of modeling that we have reached. Since the new variables are just copies of existing ones, only trivial modifications are needed to the earlier formal results.

Next are the STOP rules:

$$\text{STOPGRASPED}_C \;=\; \textbf{if}\ t = t_{\text{STOP}} \wedge \boldsymbol{f}_{\text{HCU}} \geq D\ \textbf{then} \quad \textbf{do}\ \ mode := stop, grasped := \text{TRUE} \tag{32}$$

$$\text{STOPDISLODGED}_C \;=\; \textbf{if}\ t = t_{\text{STOP}} \wedge \boldsymbol{f}_{\text{HCU}} < D\ \textbf{then} \quad \textbf{do}\ \ mode := stop, grasped := \text{FALSE} \tag{33}$$

$$\text{STOPGRASPED}_P \;=\; \textbf{if}\ t = t_{\text{STOP}}\ \textbf{then do}\ \ \mathsf{skip} \tag{34}$$

$$\text{STOPDISLODGED}_P \;=\; \textbf{if}\ t = t_{\text{STOP}}\ \textbf{then do}\ \ \mathsf{skip} \tag{35}$$

And lastly the final idle rules:

$$\text{F-IDLE}_C \stackrel{c}{=} \textbf{if}\ mode = stop\ \textbf{then do}\ \ \mathsf{skip} \tag{36}$$

$$\text{F-IDLE}_P \stackrel{c}{=} \textbf{if}\ mode = stop\ \textbf{then do}\ \ \mathsf{skip} \tag{37}$$

The preceding shows that the controller synthesis procedure that we have described is as applicable to the continuous extension of ASM as it is to the discrete version. We could now go on to apply the same approach to create a continuous version of the decomposed hand+chopsticks model, but lack of space prevents us from doing this.

7 Conclusion

In this paper we have introduced the controller synthesis problem for ASM systems. The motivation was that from a goal oriented point of view, it is often more convenient to focus on overall system objectives at the outset, and to postpone detailed implementation issues, such as the specific assignment of functionality to controller or to plant, till later.

We showed briefly that controller synthesis, as we have defined it, is undecidable, and we gave a safe approximation. We then illustrated the problem with a case study based on holding food with chopsticks.

We note that the conditions demanded of the controller and of the plant in our conditions for safe controller synthesis in (4), each relate the subsystem in question to the originating system (and only to the originating system). Thus they are completely symmetrical between the controller and plant and do not depend either on there being exactly two subsystems in play. Therefore, the result generalizes to a partition of the originating system into an arbitrary number of subsystems, each built in the same fashion, with some variables to which it has exclusive write access, and a larger set of variables to which it has read access.

The preceding remark is well illustrated by the chopstick case study, since after the initial decomposition into food (plant) and hand plus chopsticks (controller), we were able to repeat the decomposition of the hand plus chopsticks subsystem yielding a further separation into chopsticks (plant) and hand (controller), resulting in a three way partition of the original system.

In practice, the successful satisfaction of the conditions in (4) often demands that a nontrivial domain theory plays a significant role. In effect, this captures the fact that

control of a system can be achieved by applying certain signals to it, only because natural laws connect these signals to the behaviour of other system attributes in a predictable way. Our simple chopstick case study illustrated this admirably.

We then considered continuous ASMs, and briefly discussed how the controller synthesis problem could be extended to that formalism, illustrating it with a further elaboration of the chopsticks case study.

Although we have focused on a very simple scenario, the ideas that we have explored have an applicability that is much wider than we have mentioned hitherto, especially in the context of today's hybrid and cyber-physical systems [6,4,5,7]. In these, there is nowadays a strong tendency towards distributed solutions to problems decribable in a global manner. So the initial global conception of the problem needs to be decomposed into a number of subsystems that co-operate to form the global solution. Not only are many of these problems intrinsically control problems anyway, making our approach directly applicable, but the abstract version of the decomposition technique that we have explored, tailored as it is to the details of ASM rule scheduling, acts as a surrogate for a much wider gamut of problems and their solutions.

References

1. Banach, R., Zhu, H., Su, W., Wu, X.: Continuous ASM, and a Pacemaker Sensing Fragment. In: Derrick, J., et al. (eds.) ABZ 2012. LNCS, vol. 7316, pp. 65–78. Springer, Heidelberg (2012)
2. Börger, E., Stärk, R.: Abstract State Machines. A Method for High Level System Design and Analysis. Springer (2003)
3. Google search: Eating with chopsticks
4. Platzer, A.: Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics. Springer (2010)
5. Sztipanovits, J.: Model Integration and Cyber Physical Systems: A Semantics Perspective. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, p. 1. Springer, Heidelberg (2011), http://sites.lero.ie/download.aspx?f=Sztipanovits-Keynote.pdf
6. Tabuada, P.: Verification and Control of Hybrid Systems: A Symbolic Approach. Springer (2009)
7. Willems, J.: Open Dynamical Systems: Their Aims and their Origins. Ruberti Lecture, Rome (2007), http://homes.esat.kuleuven.be/~rjwillems/Lectures/2007/Rubertilecture.pdf

Continuous ASM, and a Pacemaker Sensing Fragment

Richard Banach[1,⋆], Huibiao Zhu[2,⋆⋆], Wen Su[2], and Xiaofeng Wu[2]

[1] School of Computer Science, University of Manchester,
Oxford Road, Manchester, M13 9PL, U.K.
banach@cs.man.ac.uk
[2] Software Engineering Institute, East China Normal University,
3663 Zhongshan Road North, Shanghai 200062, P.R. China
{hbzhu,wensu,xfwu}@sei.ecnu.edu.cn

Abstract. The ASM framework is extended to include continuously varying quantities as well as conventional discretely changing ones. This opens the door to the more faithful modeling of many scenarios where digital systems have to interact with the continuously varying physical world. Transitions in the extended framework are thus either *moded* (for discontinuous changing quantities), or *pliant* (for smoothly changing quantities). Refinement and retrenchment are defined in the extended context. The framework is used to develop a fragment of a simple system for the sensing problem for cardiac pacemakers, in the context of the pacemaker verification challenge.

1 Introduction

Conventional model based formal refinement technologies (see for example [1,26,2,9]) are based on discrete mathematical and logical concepts. These are typically ill suited to modeling and developing applications whose models are expressed in continuous mathematics. Nevertheless, many such applications are these days implemented using digital techniques. So there is a mismatch between the ideal of continuous modeling at the abstract level, and the discrete techniques used close to implementation.

In this paper we present an extension of the ASM formalism that enables us to treat continuously changing quantities fluently, and we develop the accompanying extension of ASM refinement and retrenchment to cope with it. The ASM extension is based on restricting the continuous behaviours to solutions of well posed initial value problems. The resulting framework includes all the behaviours needed for the kind of engineering problems that arise in practice.

We apply this framework to a simple version of the sensing problem for heart pacemakers. Pacemakers have been proposed as a study problem for the Verification Grand

⋆ Work partly done while the first author was a visiting researcher at the Software Engineering Institute at East China Normal University. The support of ECNU is gratefully acknowledged.

⋆⋆ Huibiao Zhu is supported by National Basic Research Program of China (No. 2011CB302904), National High Technology Research and Development Program of China (No. 2011AA010101 and No. 2012AA011205), National Natural Science Foundation of China (No. 61061130541 and No. 61021004).

J. Derrick et al. (Eds.): ABZ 2012, LNCS 7316, pp. 65–78, 2012.

Challenge [13,24,25]. Pacemakers are interesting from this viewpoint since the historical approach to their evolution and development is so deeply ingrained in clinical practice and experimentation [7,11]. From the formal point of view, the most pressing question that this prompts is: "what exactly are the requirements?"

The rest of this paper is as follows. In Section 2 we briefly introduce pacemakers, the pacemaker challenge and work done to date, and our own focus: pacemaker sensing. In Section 3 we review ASM modeling, we give the extension to continuous phenomena, and we develop the relevant refinement and retrenchment machinery. Section 4 explores the sensing problem in more detail. Section 5 presents some ASM models for the sensing problem, starting with a simple reference model. Section 6 concludes.

2 Heart Pacemakers, the Pacemaker Challenge, and Sensing

The heart has two atria and two ventricles. Blood collects in the atria, and is decanted into the ventricles by a wave of muscular contraction stimulated by the sinoatrial node. A short time later, another powerful wave of contraction in the ventricles pumps the blood round the body.

The heart beats when it is told to do so (via an electrical pulse) by its environment, in this case the sinoatrial node which initiates atrial depolarization. In normal working, the atrial pulse has to be of the right characteristics to cause depolarization. Similarly, the ventricular pulse causes the ventricular depolarization to happen. Inevitably, various things can go wrong with the nervous mechanisms that cause all this to happen. Problems of different kinds can arise with atrial depolarization, with ventricular depolarization, with both, and with the relationship between them. Such deficiencies are collectively referred to as heart block. Heart block can be addressed by the implantation and configuring of pacemakers, which supply electrical pulses that substitute for ones that the body is unable to generate properly itself.

Over the years, pacemakers have evolved into very sophisticated devices. To facilitate research into the computing dimension of pacemaker technology, a public domain specification of a pacemaker system has been produced by Boston Scientific [10] for use in the Verification Grand Challenge [13,24,25].

A perusal of this document reveals that the reader has to rely on a very large amount of additional knowledge. The most self-contained and relatively complete part of [10] deals with the different "modes" of pacemaker working. The wide range of possible electrical stimulation defects gives rise to a corresponding range of modes of pacemaker operation, each designed to address a specific defect. Each of these modes is assembled out of a number of available pacemaker features, where each relevant feature of a mode is tunable by the physician within a given range. Aided by a moderate amount of supplementary information, this aspect of pacemaker operation becomes tractable, and has attracted some interest from the verification community [16,12,17].

Normally, the pacemaker listens to the heart's activity. When it detects that the heart has generated a depolarization, it suppresses the artificial pulse — this is sensing. The heart's electrical activity can be detected on the patient's skin (via an electrocardiogram, (ECG)), and inside the heart itself (via a cardiac electrogram (EG)). Fig. 1 illustrates. Since the pacemaker is implanted inside the patient, it is the EG that it must sense.

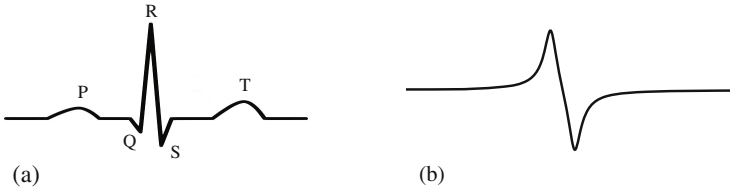


Fig. 1. (a) A surface electrocardiogram (ECG). (b) An internal cardiac electrogram (EG).

In the clinical view of sensing, e.g. in [21,3,15], it is assumed that sensing is done by a filter circuit. This works in the frequency domain, registering electrical activity in the relevant frequency range, suppressing other frequencies. The circuit outputs a discrete "Yes" or "No" according to the power spectrum of the filtered input.

The majority of the literatures in this area simply assume that the pacemaker knows whether or not a pulse has occured. Nevertheless, the clinical literature always cautions that there is a significant risk of sensing circuits confusing a true depolarization with other electrical activity. The source of this problem is the following.

The circuit senses only the power spectrum, so the phase information in the signal is discarded. Consequently, signals having the right power spectrum, but with a shape very different from Fig. 1.(b) can be erroneously identified with a depolarization.

In this paper, we illustrate our continuous extension of the ASM formalism by supplementing frequency domain sensing with a time domain tracking of the EG. This seeks to distinguish genuine depolarizations from spurious other signals sharing a similar power spectrum. Since pacemakers are capable of measuring the internal EG ([10] explicitly demands such a capability), this is an entirely credible strategy.

3 ASM, Discrete and Continuous

In this section we review the essentials of ASM [9,8], and extend the formalism to cope with continuously varying quantities, extending its reach to address many problems not treatable using the purely discrete theory alone.

3.1 Continuous ASM Models

To keep things simple, we assume that the states of an ASM model are given by valuations of the tuple of variables relevant to the model, i.e. functions from the tuple of variables to the tuple of the variables' types. To extend such models to include continuously varying phenomena, we partition the variables into two subsets: the **mode variables**, whose types are discrete sets, and which are therefore only permitted to change discontinuously and discretely, and the **pliant variables**, whose types include topologically dense sets, and which are permitted to evolve both continuously and via discrete changes. By restricting to mode variables alone, we recover the conventional discrete ASM framework.

We model time as an interval $\mathcal{T}$ of the real numbers $\mathbb{R}$, with a finite left endpoint representing the time at which the initial state of the model is created, and with a right

endpoint which is either finite or infinite, depending on whether the dynamics is finite or infinite. Now, the values of all variables become functions of $\mathcal{T}$. For the mode variables, this function is a piecewise constant function, constant on each element of a sequence of left-closed right-open intervals. Thus $\mathcal{T}$ itself also partitions into a sequence of left-closed right-open intervals, $\langle [t_0 \ldots t_1), [t_1 \ldots t_2), \ldots \rangle$, the coarsest partition of $\mathcal{T}$ such that all discontinuous changes take place at some boundary point t_i.

In a typical interval $[t_i \ldots t_{i+1})$, the mode variables will be constant, but the pliant variables will change continuously. However, merely insisting on continuity still allows for a wide range of mathematically pathological behaviours. To eliminate these, we make the following restrictions:

I Zeno: there is a constant δ_{Zeno}, such that for all i needed, $t_{i+1} - t_i \geq \delta_{\mathsf{Zeno}}$.[1]

II Limits: for every variable x, and for every time $t \in \mathcal{T}$, the left limit $\lim_{\delta \to 0} x(t-\delta)$ written $\overrightarrow{x(t)}$ and right limit $\lim_{\delta \to 0} x(t+\delta)$, written $\overleftarrow{x(t)}$ (with $\delta > 0$) exist, and for every t, $x(t) = \overleftarrow{x(t)}$. [N. B. At the endpoint(s) of $\mathcal{T}$, any missing limit is defined to equal its counterpart.]

III Differentiability: The behaviour of every pliant variable x in the interval $[t_i \ldots t_{i+1})$ is given by the solution of a well posed initial value problem $\mathcal{D}\, xs = \phi(xs, t)$ (where xs is a relevant tuple of pliant variables and $\mathcal{D}$ is the time derivative). "Well posed" means that $\phi(xs, t)$ has Lipschitz constants which are uniformly bounded over $[t_i \ldots t_{i+1})$ bounding its variation with respect to xs, and that $\phi(xs, t)$ is measurable in t.

It is recognised that ASM types can be mathematically complex entities. Therefore it is intended that I-III above apply to variables with as general a type as might be needed, provided that the concepts required in I-III (left/right limits, initial value problem, Lipschitz constants, uniform boundedness, measurability) make sense for them.

With I-III in place, the behaviour of every pliant variable is piecewise absolutely continuous, with the variation being described by a suitable differential equation (DE).

Accompanying the distinction between mode and pliant variables, is a distinction between mode and pliant transitions. Mode transitions are just like conventional ASM transitions in that they record a discrete transition from before-values to after-values of the mode variables, albeit that these are the values of piecewise constant functions of time. A rule for a mode transition OP can be written using familiar ASM notation:

$$
\begin{array}{l}
\textsc{Op}(\textbf{in}\ \overrightarrow{is}, \textbf{out}\ \overleftarrow{os}) \ = \\
\textbf{if}\ \ guard(\overrightarrow{xs}, \overrightarrow{is})\ \ \textbf{then choose}\ \ \overleftarrow{xs}, \overleftarrow{os} \\
\quad \textbf{with}\ \ rel(\overleftarrow{xs}, \overrightarrow{xs}, \overrightarrow{is}, \overleftarrow{os})\ \ \textbf{do}\ \ xs, os\ :=\ \overleftarrow{xs}, \overleftarrow{os}
\end{array}
\tag{1}
$$

In (1) we single out is and os, the inputs and outputs (read-only and write-only respectively), while xs are the state variables (accessed in read/write manner). Note that the

[1] Our approach to the Zeno problem contrasts with many others, which demand that any finite time interval contains only a finite number of transitions, or that the sequence of transition times contains no accumulation points. But this permits the sequence $t_{i+1} - t_i = 1/i$, which satisfies the mentioned restrictions, yet allows transitions to get arbitrarily close together.

choice of left limit for before-values and right limit for after-values makes (1) into the kind of instantaneous transition that we would expect. Also, if the after-values for xs and os are available explicitly, the relevant expression can be assigned in the **do** clause, and the **choose** and **with** clauses can be omitted.

Pliant transitions do the corresponding job for pliant variables. While a mode transition is a single before-/after-value pair, a pliant transition is a family of before-/after-value pairs parameterized by the relevant time interval $[t_i \ldots t_{i+1})$. Moreover, instead of the change from before-values to after-values taking place instantaneously, the before-value refers to the initial value at t_i while the after-value refers to an arbitrary time in the interval, so the before-value and after-value are separated in time. To reflect the constraints that apply to pliant transitions, we write rules for them thus:

$$\begin{array}{l}\textsc{PliOp}(\textbf{in}\ is(t \in (t_{\mathrm{L}(t)} \ldots t_{\mathrm{R}(t)})), \textbf{out}\ os(t \in (t_{\mathrm{L}(t)} \ldots t_{\mathrm{R}(t)}))) \stackrel{\mathrm{c}}{=} \\ \textbf{if}\ \ IV(xs(t_{\mathrm{L}(t)})) \wedge guard(xs(t_{\mathrm{L}(t)}))\ \ \textbf{then with}\ \ rel(xs, is, os, t) \\ \quad \textbf{do}\ \ xs(t), os(t)\ :=\ \textbf{solve}\ DE(xs(t), is(t), os(t), t)\end{array} \tag{2}$$

In (2), the symbol $\stackrel{\mathrm{c}}{=}$ signals the presence of a pliant transition, distinguishing it from the instantaneous kind. The inputs is and outputs os are continuously absorbed from and emitted to the environment, as indicated in the signature. For an arbitrary t, we let $\mathrm{L}(t) = \max\{i \mid t_i \leq t\}$ and $\mathrm{R}(t) = \min\{i \mid t_i > t\}$ so that: (a) we do not have to know the index i explicitly in $[t_i \ldots t_{i+1})$; (b) we can refer to the beginning and end of the interval during which the pliant event runs in a generic manner in the syntactic description of the event. Note that the initial values IV and guard $guard$ depend only on the before-value of the state, and not on the input, whereas rel, which expresses any additional constraints that must hold beyond the differential equation DE itself, can depend on all state and input values from the start of the interval $t_{\mathrm{L}(t)}$ up to the current time t. The assignment in (2) says that the after-state and output at t should satisfy the differential equation DE (as well as rel). As for the instantaneous case, if the continuous functions of t to be assigned to xs, os are known explicitly, we can omit the **with** and/or **solve** clauses as approporiate, and just assign xs, os to the relevant expression.

As mentioned earlier, pliant variables can undergo instantaneous discontinuous transitions as well as continuous ones. For such transitions, the structure in (1) is sufficient. We continue to call instantaneous transitions involving both kinds of variable **mode transitions**, introducing the term **pure mode transitions** for the former kind.

We say that a continuous ASM ruleset is **well formed** iff:

- Every enabled mode transition is feasible, i.e. has an after-state, and on its completion enables a pliant transition (but does not enable any mode transition). (3)
- Every enabled pliant transition is feasible, i.e. has a time-indexed family of after-states, and EITHER: (4)

  (i) During the run of the pliant transition a mode transition becomes enabled. It preempts the pliant transition, defining its end. ORELSE
  (ii) During the run of the pliant transition it becomes infeasible: finite termination. ORELSE
  (iii) The pliant transition continues indefinitely: nontermination.

A **run** of a continuous ASM system starts with a mode transition which creates the initial state, and then, pliant transitions alternate with mode transitions. The last transition (if there is one) is a pliant transition (whose duration may be finite or infinite).

3.2 Continuous ASM Refinement and Retrenchment

Now we develop our continuous ASM framework to encompass refinement and retrenchment of continuous ASM models. We start by describing the usual formulation, appropriate to pure mode transitions, and then show how to extend this to encompass the new kinds of transition.

In general, to prove a conventional ASM refinement or retrenchment, we verify so-called (m, n) diagrams, in which m abstract steps simulate n concrete ones in an appropriate way. This means that there is nothing that the n concrete steps can do that is not suitably reflected in m appropriately chosen abstract steps, where both m and n can be freely chosen to suit the application, and the meaning of "suitably reflected" depends on whether we are dealing with refinement or retrenchment. For this paper, it will be sufficient to focus on the refinement and retrenchment proof obligations (POs) which are the embodiment of this policy. The situation for refinement is illustrated in Fig. 2, in which we suppress input and output for clarity.

In Fig. 2 the retrieve relation $R_{\mathrm{A,C}}$, between abstract and concrete states, holds at the beginning and end of the (m, n) pair. This permits us to "glue together" such (m, n) diagrams to create relationships between abstract and concrete runs in which $R_{\mathrm{A,C}}$ is periodically re-established. [N. B. In much of the ASM literature, the main focus is on an *equivalence*, usually written $\equiv$, between abstract and concrete states. This is normally deemed to contain a "practically useful" subrelation $R_{\mathrm{A,C}}$, chosen to be easier to work with. The approach via $R_{\mathrm{A,C}}$ will be the focus of our treatment, and is also focus of the KIV [14] formalization in [19,20].]

The first PO is the initialization PO, common to both refinement and retrenchment:

$$\forall\, y' \bullet CInit(y') \Rightarrow (\exists\, x' \bullet AInit(x') \wedge R_{\mathrm{A,C}}(x', y')) \tag{5}$$

In (5), it is demanded that for each concrete initial state y', there is an abstract initial state x' such that the retrieve relation $R_{\mathrm{A,C}}(x', y')$ holds.

The second PO is correctness. The PO is concerned with the verification of (m, n) diagrams. For this, we have to have some way of deciding which (m, n) diagrams are sufficient for the application. Let us assume that we have done this. Let $CFrags$ be the set of fragments of concrete runs that we have previously determined will permit a covering of all the concrete runs of interest for the application. We write $y :: ys :: y' \in CFrags$ to denote an element of $CFrags$ starting with concrete state y, ending with concrete state y', and with intervening concrete state sequence ys. Likewise we write $x :: xs :: x' \in AFrags$ for abstract fragments. Let is, js, os, ps denote the sequences of abstract inputs, concrete inputs, abstract outputs, concrete outputs, respectively, belonging to $x :: xs :: x'$ and $y :: ys :: y'$ and let $In_{\mathrm{AOPS,COPS}}(is, js)$ and $Out_{\mathrm{AOPS,COPS}}(os, ps)$ denote suitable input and output relations. The specific form of the correctness PO now differs in form depending on whether we are dealing with refinement or retrenchment. We start with refinement. Then the correctness PO reads:

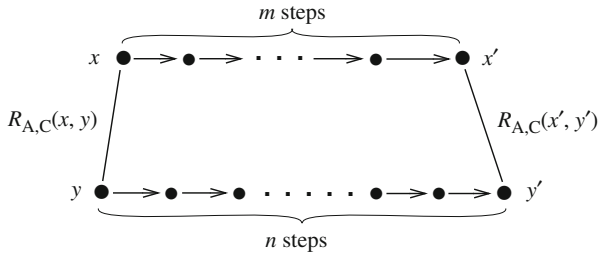


Fig. 2. An ASM (m, n) diagram, showing how m abstract steps, going from state x to state x' simulate n concrete steps, going from y to y'. The simulation is embodied in the retrieve relation $R_{\mathrm{A,C}}$, which holds for the before-states of the series of steps $R_{\mathrm{A,C}}(x, y)$, and is re-establshed for the after-states of the series $R_{\mathrm{A,C}}(x', y')$.

$$\begin{array}{l}\forall\, x, is, y, ys, y', js, ps \bullet y :: ys :: y' \in CFrags \wedge \\ \quad R_{\mathrm{A,C}}(x, y) \wedge In_{\mathrm{AOPS,COPS}}(is, js) \wedge \mathrm{COPS}(y :: ys :: y', js, ps) \;\Rightarrow \\ \quad\quad (\exists\, xs, x', os \bullet x :: xs :: x' \in AFrags \wedge \mathrm{AOPS}(x :: xs :: x', is, os) \wedge \\ \quad\quad\quad R_{\mathrm{A,C}}(x', y') \wedge Out_{\mathrm{AOPS,COPS}}(os, ps)) \end{array} \tag{6}$$

In (6), it is demanded that whenever there is a concrete run fragment of the form $\mathrm{COPS}(y :: ys :: y', js, ps)$, carried out by a sequence of concrete operations[2] COPS, with state sequence $y :: ys :: y'$, input sequence js and output sequence ps, such that the retrieve and input relations $R_{\mathrm{A,C}}(x, y) \wedge In_{\mathrm{AOPS,COPS}}(is, js)$ hold between concrete and abstract before-states and inputs, then an abstract run fragment $\mathrm{AOPS}(x :: xs :: x', is, os)$ can be found to re-establish the retrieve and output relations $R_{\mathrm{A,C}}(x', y') \wedge Out_{\mathrm{AOPS,COPS}}(os, ps)$.

The ASM refinement policy also demands that non-termination be preserved from concrete to abstract, but we will not need that in this paper.

Assuming that (5) holds, and that we can prove enough instances of (6) to cater for the application of interest, then the concrete model is a **correct refinement** of the abstract model. In a correct refinement, all the properties of the concrete model (that are visible through the retrieve and other relations), are suitably reflected in properties of the abstract model (because of the direction of the implication in (6)). If in addition, the abstract model is also a correct refinement of the concrete model (using the converses of the same relations), then the concrete model is a **complete refinement** of the abstract model. In a complete refinement, all relevant properties of the abstract model are also present in the concrete model (because of the direction of the implication in the modified version of (6)). Therefore, to ensure that the complete set of requirements of an intended system is faithfully preserved through a series of refinement steps, it is enough to express them all in a single abstract model, and then to ensure that each refinement step is a complete refinement. We now turn to the retrenchment version of the correctness PO.

[2] We define an operation as a maximal enabled set of rules — provided its updates are consistent. Enabled inconsistent updates cause abortion of the run.

For retrenchment, [6,5] give definitive accounts; latest developments are found in [18]. See also [4] for formulations of retrenchment adapted to several specific model based refinement formalisms including ASM. The retrenchment correctness PO weakens (6) by inserting *within*, *output* and *concedes* relations, $W_{\text{AOPS,COPS}}$, $O_{\text{AOPS,COPS}}$, $C_{\text{AOPS,COPS}}$ respectively into (6), to give extra flexibility and expressivity. In particular, the concession $C_{\text{AOPS,COPS}}$ weakens the conclusions of (6) disjunctively, giving room for many kinds of "exceptional" behaviour. The result is:

$$\begin{array}{l}\forall\, x, is, y, ys, y', js, ps \bullet y :: ys :: y' \in CFrags \wedge \\ \quad R_{\text{A,C}}(x, y) \wedge W_{\text{AOPS,COPS}}(is, js, x, y) \wedge \text{COPS}(y :: ys :: y', js, ps) \;\Rightarrow \\ \quad\quad (\exists\, xs, x', os \bullet x :: xs :: x' \in AFrags \wedge \text{AOPS}(x :: xs :: x', is, os) \wedge \\ \quad\quad\quad ((R_{\text{A,C}}(x', y') \wedge O_{\text{AOPS,COPS}}(x, x', is, os, y, y', js, ps)) \vee \\ \quad\quad\quad\quad C_{\text{AOPS,COPS}}(x, x', is, os, y, y', js, ps)))\end{array} \tag{7}$$

To ensure that retrenchment only deals with well defined transitions, and to ensure smooth retrenchment/refinement interworking, in retrenchment we also insist that $R_{\text{A,C}} \wedge W_{\text{AOPS,COPS}}$ always falls in the domain of the requisite operations, though this is another thing not needed here.

All of the preceding was still formulated for the exclusively discrete world. However, to extend it to the continuous world too, is simplicity itself. We just have to reinterpret the paths, $x :: xs :: x'$ and $y :: ys :: y'$ appearing in (6) and (7) appropriately, and we are done.

More precisely, a path like $x :: xs :: x'$ say, can consist of interleavings of pliant and mode transitions. If $x :: xs :: x'$ starts with a pliant transition, then the value x used in the POs (6) and (7) is the right limit at its initial point x. Similarly, if $x :: xs :: x'$ ends with a pliant transition, then the value x' used in the POs is the left limit at its endpoint x'. Similar remarks apply to the concrete path $y :: ys :: y'$. The fact that the POs are largely insensitive to what goes on in the interior of the paths, makes them equally applicable to paths that interleave pliant and mode transitions, as to paths that just have a sequence of discrete states in their interior.

4 Pacemaker Sensing

Our objective for the rest of this paper is to design an ASM system to track a signal that represents the continuous internal electrical activity of the heart. A theoretical treatment of the expected shape of the signal detected by an electrode inside the heart as a wave of depolarization passes over it has been carried out some time ago, based on the idea of a dipole of charge passing a detector [22,23]. This is the basic shape in Fig. 1.(b).

Thus, our abstract model focuses on Fig. 1.(b), allowing for an acceptable margin of error. This is shown in Fig. 3. There, the basic shape $pulse(t)$ (solid curve in Fig. 3) is given by (8), while the dashed error curves surrounding it are given by adding (or subtracting) $err(t)$, which is a small positive constant augmented by a strongly peaked Gaussian centred at the point of maximum variability of $pulse(t)$:

$$pulse(t) \;=\; -\,K\left(\frac{1}{\sqrt{(t+a)^2+b^2}} - \frac{1}{\sqrt{(t-a)^2+b^2}}\right) \tag{8}$$

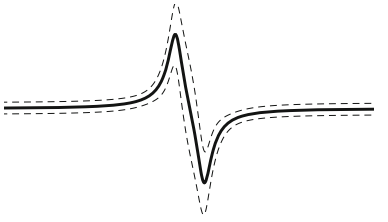


Fig. 3. Schematic of the EG of a single depolarization, with a margin of error to allow for noise

$$err(t) \;=\; c + r\exp(-p\,t^2) \tag{9}$$

In real pacemakers, the completion of the previous heartbeat initiates the start of a fresh cycle. Let us assume that this is at time $t = 0$. The new cycle begins with a so called refractory period, in which the heart signal is ignored, allowing the transients of the previous heartbeat to die off, and for myocardial polarization to be re-established. Say this refractory period lasts till T_0. Then, from T_0 to T_{MAX}, the heart signal is monitored for the presence of a shape similar to $pulse$. Assuming the heart is working normally, if the patient is being physically active, then the patient's heartrate will be higher and the pulse occurs sooner; whereas if the patient is being inactive then the heartrate will be lower and the pulse will occur later. Thus, we can specify a typical function $egm_A(t)$ that constitutes an acceptable electrogram shape as follows:

$$\begin{aligned}&(\exists\, R \bullet T_0 < R_0 \leq R \leq R_{MAX} < T_{MAX} \wedge \\ &\quad (\forall\, t \bullet T_0 < t < T_{MAX} \Rightarrow egm_A(t) \in egmWin_R(t)))\end{aligned} \tag{10}$$

where

$$egmWin_R(t) = [egmWin^-_R(t) \ldots egmWin^+_R(t)] \tag{11}$$

where

$$egmWin^-_R(t) = pulse(t - R) - err(t - R) \tag{12}$$

and

$$egmWin^+_R(t) = pulse(t - R) + err(t - R) \tag{13}$$

In (10), $egmWin_R(t)$ is set valued, and at each relevant time t, it specifies the allowable cardiac electrogram window within which a normally operating heart's electrogram behaviour is expected to fall. A minor complication is that the precise moment within the allowed timeframe at which the electrogram undergoes the pulse is not known exactly, but is merely constrained by $R_0 \leq R \leq R_{MAX}$, where, like T_0 and T_{MAX}, the constants R_0 and R_{MAX} are statically determined.

5 Sensing Models

5.1 The Reference Model

At the abstract level, we can view $egmWin_R(t)$ as as static function, depending on two parameters: time and the chosen value of R. With this, we can specify a reference

model for cardiac behaviour using a pliant rule that generates acceptable electrograms as follows, where any call of EGM_A produces an electrogram egm_A which stays within the acceptable window $egmWin_R$ during the period of interest, $T_0 < t < T_{MAX}$:

$$\begin{array}{l}
\text{EGM}_A(\textbf{out}\ ego(t)) \stackrel{\text{c}}{=} \\
\textbf{choose}\ eg'(t) \\
\textbf{with}\ (\exists\, R \bullet T_0 < R_0 \le R \le R_{MAX} < T_{MAX} \wedge T_0 < t < T_{MAX} \wedge \\
\qquad eg'(t) \in egmWin_R(t)) \\
\quad \textbf{do}\ egm(t) := eg'(t), ego(t) := eg'(t)
\end{array} \tag{14}$$

5.2 Heartrate-Aware ASM Sensing

We now present the continuous variable heartrate sensing model, the CVH model. Initialisation (assumed to be at time 0), resets the "verdict" variables $pulseGOOD$ and $pulseBAD$, and initialises the pliant variables dev, $egirise$ and $egifall$ which will measure the quality of the electrogram. Then it models the refractory period.

$$\begin{array}{l}
\text{INIT}_{CVH} = \\
\textbf{if}\ t = 0\ \textbf{then} \\
\quad \textbf{do}\ mode := refrac, pulseGOOD := \mathsf{false}, pulseBAD := \mathsf{false}, \\
\qquad winR := \emptyset, dev := 0, egirise := 0, egifall := 0
\end{array} \tag{15}$$

$$\text{REFRACTORY}_{CVH} \stackrel{\text{c}}{=} \textbf{if}\ mode = refrac\ \textbf{then do}\ \mathsf{skip} \tag{16}$$

A mode transition signals the start of sensing:

$$\begin{array}{l}
\text{STARTSENSING}_{CVH} = \\
\textbf{if}\ mode = refrac \wedge t = T_0\ \textbf{then} \\
\quad \textbf{do}\ mode := sensing, pulseGOOD := \mathsf{false}, pulseBAD := \mathsf{false}, \\
\qquad winR := \emptyset, dev := 0, egirise := 0, egifall := 0
\end{array} \tag{17}$$

The switch to sensing mode activates the sensing process.

$$\begin{array}{l}
\text{EGM}_{CVH}(\textbf{in}\ egi(t)) \stackrel{\text{c}}{=} \\
\textbf{let}\ wR(\bar{s}) = \textbf{\textit{if}}\ \bar{s} \notin \operatorname{dom} egi\ \textbf{\textit{then}}\ \emptyset\ \textbf{\textit{else}}\ \{R \mid R_0 \le R \le R_{MAX} \wedge \\
\qquad\qquad \forall \tilde{s} \bullet t_{\mathrm{L}(t)} < \tilde{s} \le \bar{s} \Rightarrow egi(\tilde{s}) \in egmWin_R(\tilde{s})\}\ \textbf{\textit{fi}}\ \textbf{in} \\
\textbf{let}\ s = \max\{\tilde{s} \mid t_{\mathrm{L}(t)} < \tilde{s} \le t \wedge wR(\tilde{s}) \neq \emptyset\}\ \textbf{in} \\
\textbf{if}\ mode = sensing\ \textbf{then} \\
\quad \textbf{do}\ winR(t) := wR(s), \\
\qquad dev(t) := \textbf{solve}\ \mathcal{D}\, dev(t) = \\
\qquad\qquad \min\{\mathrm{dist}(egi(t), egmWin_R(t)) \mid R \in winR(t)\}, \\
\qquad egirise(t) := \textbf{solve}\ \mathcal{D}\, egirise(t) = \mathcal{D}\, egi(t)\, \Theta(\mathcal{D}\, egi(t)), \\
\qquad egifall(t) := \textbf{solve}\ \mathcal{D}\, egifall(t) = \mathcal{D}\, egi(t)\, \Theta(-\mathcal{D}\, egi(t))
\end{array} \tag{18}$$

In (18), Θ returns 1 if its parameter is positive, else 0. Sensing continues until a signal conforming to the region of high variation in $pulse$ has been detected (indicating a spontaneous heartbeat), or the timeout expires (indicating abnormal cardiac function).

Consider a normal heartbeat. If $egi(t)$ is close to zero near $t = T_0$, then a large range of R values will satisfy $egi(t) \in egmWin_R(t)$. As time progresses, the normal

heartbeat reaches the point at which the double spike occurs. As the electrogram follows the shape of $pulse$, very soon the range of allowable R values is reduced to around $2t_0$ (where t_0 is the solution nearest to 0 to the equation $pulse(t_0) + err(t_0) = 0$), which is the width of the time window around the central point of $pulse$ in Fig. 3. Setting δ_{th} (the width of the window of allowable R values such that $egmWin_R$ contains the whole of the observed electrogram (which must be more than zero)) to slightly more than $2t_0$, gives us a feature to test for when confirming the presence of a normal heartbeat.

Also, since the whole of the observed electrogram is inside $egmWin_R$, $\mathcal{D}\, dev(t)$ remains at zero, and so dev stays at zero too, giving another feature to test for when confirming a normal heartbeat. And since $egirise$ and $egifall$ track the overall positive and negative change in the electrogram value, a normal heartbeat will also be characterised by $|egirise + egifall| < \delta_{rf}$ and $egirise > \Delta_{th}$, with δ_{rf} and Δ_{th} suitable constants (one small, one big). Observing also that the time for confirming a normal heartbeat will be less than T_{MAX}, a normal heartbeat enables the following mode transition:

$$\begin{array}{l}\textsc{PulseGood}_{CVH} = \\ \textbf{if } mode = sensing \wedge dev = 0 \wedge 0 < |\max winR - \min winR| < \delta_{th} \wedge \\ \quad |egirise + egifall| < \delta_{rf} \wedge egirise > \Delta_{th} \wedge T_0 < t < T_{MAX} \textbf{ then} \\ \quad \textbf{do } mode := refrac, pulseGOOD := \mathsf{true}\end{array} \tag{19}$$

Consider an abnormal heartbeat. By definition, an abnormal heartbeat does *not* conform to the envelope of permitted deviation around $pulse$. Thus $winR(t)$ will diminish over time, but instead of eventually remaining at a little less than δ_{th}, it will shrink to a single value, the last value $\bar{R}$ at the last time $\bar{t}$ for which the "electrogram-so-far remains in $egmWin_R$" property still holds.

After this moment, the distance between $egi(t)$ and $egmWin_{\bar{R}}(\bar{t})$ will become positive, leading to a positive $\mathcal{D}\, dev(t)$ and thus positive dev, which remains until the deadline T_{MAX} expires, giving a feature to test for when confirming an abnormal heartbeat. This enables the following mode transition:

$$\begin{array}{l}\textsc{PulseBad}_{CVH} = \\ \textbf{if } mode = sensing \wedge dev > 0 \wedge t = T_{MAX} \textbf{ then} \\ \quad \textbf{do } mode := refrac, pulseBAD := \mathsf{true}\end{array} \tag{20}$$

To deal with successive heartbeats, we need to slightly alter the way we deal with time. There are two relatively straightforward approaches. In the first, we continue with the perspective that t refers to real time (measured from some arbitrary starting point). In this view, we would need $\textsc{PulseGood}_{CVH}$ and $\textsc{PulseBad}_{CVH}$ to re-assign the values of T_0 and T_{MAX} to appropriate future values (by adding to them the duration of the heartbeat that has just elapsed), ready for the next heartbeat. In the second, we would regard t as a *clock* — which is reset at the beginning of every heartbeat (via assignments $t := 0$ in $\textsc{PulseGood}_{CVH}$ and $\textsc{PulseBad}_{CVH}$) instead of referring to real time. In this view, we would need to re-interpret the notations $t_{\mathrm{L}(t)}$ and $t_{\mathrm{R}(t)}$ so that they referred to the clock time at the beginning and end of the current invocation of the current pliant transition, rather than to real time. Both of these approaches are quite straightforward, allowing $\textsc{Refractory}_{CVH}$ to be re-enabled for a repetition of the sensing behaviour in the next heartbeat.

5.3 On EGM_A and EGM_{CVH}

In a full-blown formal development, the relationship between EGM_A and the more concrete EGM_{CVH} would be of interest in monitoring how the pacemaker sensing requirements were being met through the development. Here, for lack of space, we just sketch an aspect of it as an illustration of the flexibility of our techniques.

The main use of pacemakers is in situations where cardiac behaviour is expected to *not* conform to the normal for a significant proportion of the time. This makes the prospects for a conventional kind of refinement futile. On the other hand, if we take advantage of the additional flexibility of retrenchment, then the prospects for setting up a formal relationship between EGM_A and EGM_{CVH} are improved. We now give one possible such retrenchment from EGM_A to EGM_{CVH}.

The first concern in a retrenchment is the retrieve relation from the A model state space to the CVH model state space, $R_{A,CVH}$. Given that the two models operate on different timeframes, that the size of the allowed window of R values $egmWin_R$, varies over time, and recognising that the other relations of a retrenchment permit more finegrained control over states, it is acceptable to trivialise $R_{A,CVH}$:

$$R_{A,CVH}(egm, \langle winR, dev\rangle) \;\equiv\; \mathsf{true} \tag{21}$$

The within relation controls what is demanded of before-states and inputs in the hypotheses of the correctness PO (7):

$$\begin{aligned} &W_{\text{EGM}_A,\text{EGM}_{CVH}}(egi(t \in (T_0 \ldots T')), egm(T_0), \langle winR, dev\rangle(T_0)) \;\equiv \\ &\quad dev(T_0) = 0 \wedge egirise = 0 \wedge egifall = 0 \wedge |egi(T_0)| < \delta_{\text{small}} \end{aligned} \tag{22}$$

In (22), T' refers to the time at which the after-state of the heartbeat is reached. We see that $dev(T_0) = 0$ and $egirise = 0$ and $egifall = 0$ are all recorded in (22), consistent with (17), and with the fact that these initial values are not mentioned in (18). We also record that the right limit of the sensed electrogram, $egi(T_0)$, is small enough to substantiate our earlier assumptions that egi is near zero at the start of sensing.

Now, either EGM_{CVH} conforms to EGM_A and we have a normal heartbeat, or not. In the former case, it is appropriate to describe the properties of the two executions using the output relation:

$$\begin{aligned} &O_{\text{EGM}_A,\text{EGM}_{CVH}}(egm(T_0, T'), ego(t \in (T_0 \ldots T')), \\ &\qquad\qquad \langle winR, dev\rangle(T_0, T'), egi(t \in (T_0 \ldots T'))) \;\equiv \\ &\quad dev(T') = 0 \wedge 0 < |\max\, winR - \min\, winR| < \delta_{th} \wedge \\ &\quad |egirise + egifall| < \delta_{rf} \wedge egirise > \Delta_{th} \wedge \\ &\quad (\forall\, t \bullet T_0 < t < T' \Rightarrow \text{EGM}_A(ego(t)) \wedge ego(t) = egi(t)) \end{aligned} \tag{23}$$

The gist of (23) is that the variables dev, $egirise$ and $egifall$ behaved as expected, and also that, since we had a normal heartbeat, the input electrogram $egi(t)$ always remained within the permitted envelope, $egmWin_R(t)$ for some R, and therefore that there is a possible abstract electrogram $ego(t)$ that followed it exactly.

If though, we are dealing with the case of an abnormal heartbeat, we can describe the properties of the two executions using the concession:

$$\begin{aligned} &C_{\text{EGM}_A,\text{EGM}_{CVH}}(egm(T_0,T'), ego(t \in (T_0 \ldots T')),\\ &\qquad\qquad \langle winR, dev\rangle(T_0,T'), egi(t \in (T_0 \ldots T'))) \;\equiv\\ &\quad T' = T_{MAX} \wedge dev(T') > 0 \wedge (\forall\, R \bullet R_0 \leq R \leq R_{MAX} \Rightarrow\\ &\qquad (\exists\, t \bullet T_0 < t < T' \wedge egi(t) \notin egmWin_R(t))) \end{aligned} \tag{24}$$

In (24), the first clause states that the deadline has expired, while the second states that the accumulated deviation from (even the best possible) permitted electrogram windows is positive. The third clause makes the preceding more explicit by asserting the existence of a value t at which the observed electrogram falls outside the permitted window $egmWin_R(t)$ (for even the best possible R).

This completes the relationship bewtween EGM_A and EGM_{CVH}. As hinted above, the step from EGM_A to EGM_{CVH} constitutes but the first step of a formal development of the time domain sensing application. The remainder of the development throws up some fascinating technical challenges for our formal framework, which, unfortunately, we do not have the space to explore in the present paper.

6 Conclusion

In the preceding sections, we introduced the pacemaker sensing problem as a case study that was not only connected with the Verification Grand Challenge [13,24,25], but was also one that required continuous machinery to attain reasonably faithful modeling. We then gave an extension of the ASM framework to enable it to deal with continuous behaviours. These behaviours were not arbitrary, but were constrained by a Zeno condition, and the need to be describable using differential equations with right hand sides that are Lipschitz in the dependent variables and measurable in time. This class is flexible enough to include naturally occurring engineering discontinuities, without admitting unnecessarily pathological behaviours.

We then applied this framework to our case study. For lack of space, we were not able to pursue this beyond the first stage, giving merely a taste of both the richness of the case study, and of our framework's capabilities. A more extensive treatment, pursuing the development of the case study to near-implementation, will appear elsewhere.

References

1. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (1996)
2. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
3. Aubert, A., Goldreyer, B., Wyman, M., Jaquemlyn, E., Ector, H., de Geest, H.: Filter Characteristics of the Atrial Sensing Circuit of a Rate Responsive Pacemaker. To See or Not to See. PACE 12, 525–536 (1989)
4. Banach, R.: Model Based Refinement and the Design of Retrenchments. Available from [18]

5. Banach, R., Jeske, C., Poppleton, M.: Composition Mechanisms for Retrenchment. J. Log. Alg. Prog. 75, 209–229 (2008)
6. Banach, R., Poppleton, M., Jeske, C., Stepney, S.: Engineering and Theoretical Underpinnings of Retrenchment. Sci. Comp. Prog. 67, 301–329 (2007)
7. Barold, S., Stroobandt, R., Sinnaeve, A.: Cardiac Pacemakers and Resynchronization Step by Step: An Illustrated Guide. Wiley-Blackwell (2010)
8. Börger, E.: The ASM Refinement Method. FACJ 15, 237–257 (2003)
9. Börger, E., Stärk, R.: Abstract State Machines. A Method for High Level System Design and Analysis. Springer (2003)
10. Boston Scientific: PACEMAKER System Specification (2007), http://www.cas.mcmaster.ca/sqrl/_SQRLDocuments/PACEMAKER.pdf
11. Ellenbogen, K., Wood, M.: Cardiac Pacing and ICDs, 5th edn. Wiley-Blackwell (2008)
12. Gomes, A.O., Oliveira, M.V.M.: Formal Specification of a Cardiac Pacing System. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 692–707. Springer, Heidelberg (2009)
13. Jones, C., O'Hearne, P., Woodcock, J.: Verified Software: A Grand Challenge. IEEE Computer 39, 93–95 (2006)
14. Karlsruhe Interactive Verifier, http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/kiv/
15. Keinert, M., Elmqvist, H., Strandberg, H.: Spectral Properties of Atrial and Ventricular Endocardial Signals. PACE 2, 11–19 (1979)
16. Macedo, H.D., Larsen, P.G., Fitzgerald, J.: Incremental Development of a Distributed Real-Time Model of a Cardiac Pacing System Using VDM. In: Cuellar, J., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 181–197. Springer, Heidelberg (2008)
17. Méry, D., Singh, N.: Functional Behavior of a Cardiac Pacing System. Tech. rep., LORIA, Université Henri Poincaré - Nancy I (2011), http://www.loria.fr/~singhnne/Home_files/downloads/ijdecs2010.pdf, Int. J. Discrete Event Control Systems
18. Retrenchment Homepage, http://www.cs.man.ac.uk/retrenchment
19. Schellhorn, G.: Verification of ASM Refinements Using Generalized Forward Simulation. JUCS 7, 952–979 (2001)
20. Schellhorn, G.: ASM Refinement and Generalizations of Forward Simulation in Data Refinement: A Comparison. Theor. Comp. Sci. 336, 403–435 (2005)
21. Schuchert, A., Aydin, A., Israel, C., Gaby, G., Paul, V.: Arial Pacing and Sensing Characteristics in Heart Failure Patients Undergoing Cardiac Resynchronization Therapy. Europace 7, 165–169 (2005)
22. Wilson, F., Macleod, A., Barker, P.: The Distribution of the Action Currents Produced by Heart Muscle and Other Excitable Tissues Immersed in Extensive Conducting Media. J. Gen. Physiol. 16, 423–456 (1933)
23. Wilson, F., Macleod, A., Barker, P.: The Distribution of the Currents of Action and of Injury Displayed by Heart Muscle and Other Excitable Tissues. University of Michigan Studies. Scientific Series, vol. 10. University of Michigan Press, Ann Arbor (1933); Reprinted in: Lepeschkin, Johnston (eds.) Selected Papers of Wilson, F.N., Edwards, J.W.: Ann Arbor (1954)
24. Woodcock, J.: First Steps in the The Verified Software Grand Challenge. IEEE Computer 39, 57–64 (2006)
25. Woodcock, J., Banach, R.: The Verification Grand Challenge. JUCS 13, 661–668 (2007)
26. Woodcock, J., Davies, J.: Using Z, Specification, Refinement and Proof. Prentice Hall (1996)

An ASM Model of Concurrency in a Web Browser

Vincenzo Gervasi

Dipartimento di Informatica, Università di Pisa, Pisa, Italy and
Faculty of Engineering and Information Technology, UTS, Sydney, Australia
gervasi@di.unipi.it

Abstract. We define an abstract standards-compliant web browser model. The model focuses on those parts of the browser behaviour which are most relevant for the deployment and execution of web applications, such as interaction with a scripting language (here, ECMAScript), cookies, and asynchronous behaviour of the network layer, while hiding other aspects, such as page navigation and presentational issues.

We use a multi-agent Abstract State Machine as our formal model, showing how the browser behaviour can be partitioned into a number of distinct components, and specifying precisely their interactions. The specification can also be used as basis to prove consistency properties of common frameworks for web applications.

1 Introduction

A basic web browser *could* be modeled as a function mapping uniform resource locators (URLs, i.e. web addresses) into an on-screen presentation of contents. User interaction with the page results in another URL being requested, iterating the process. The semantics of a web application would then be seen as a fix-point computation of this function. However, such a purely functional specification would miss a number of important (in practice, fundamental) details.

One of the defining characteristics of web applications is, in fact, their reliance on unreliable, asynchronous networks to retrieve all the various components of an application: visual (i.e., text and graphics on a web page), logic (i.e., source code to be executed on the browser) and services (i.e., sending data to a server for processing). In this paper, we will mostly illustrate these points.

The purpose of our effort is not to provide a full formal specification of HTML 5 [5], nor of any specific web browser (in existence or conceptual), but rather to highlight certain aspects of how a browser, seen as a thin client for web applications, behaves. As such, we will often skip the more contrived details, ignore the issues with graphics and layout entirely, and at times describe a specific implementation where the specification would allow several possibilities[1].

[1] In particular, several aspects of HTML 5 are unneedingly complicated by the need to ensure that the quirks of several different implementations in widespread use are deemed conformant.

J. Derrick et al. (Eds.): ABZ 2012, LNCS 7316, pp. 79–93, 2012.

Our model consists of several layers. At the basis, we have a *transport-level* layer, where we describe (rather abstractly) the TCP/IP communication according to the HTTP protocol which relates a web server and a web browser.

On top of the transport layer, we have a *stream-level* layer, with individual agents in charge of receiving and interpreting information coming from the network. These agents are instantiated dynamically, and roughly correspond to the multiple threads that are often found in real browsers.

Above the stream level, a *context-level* layer defines the behaviour of a *browsing context*. A browsing context typically corresponds to a single *Document* (which in turn has a *DOM* or Document Object Model) and regulates the user interaction with the same. Most typically, each window, tab, or frame in a web browser is a different browsing context.

Finally, on top of all this we have a *browser-level* layer, where we specify (in part) the behaviour of a web browser, seen as an application of the host operating system. At this level we describe initialization of new browsing contexts and interaction with the host operating system.

Due to space considerations, we only describe here the first two layers, up to the construction of a Document in the context layer. These layers are where most of the asynchronicity and non-determinism lay, and are thus more interesting from our point of view. In contrast, in the top two layers, the security model of the browser and of the OS tend to isolate ("sandbox") various components, and the event manager of the UI tend to sequentialize user events, thus leading to a more traditional view oriented towards sequential programming. The interested reader can find more details about the context-level layer and event processing in [2].

2 Notation

Providing a *complete* specification of all the different technologies involved, from the basics of point-to-point networking to the various protocols, languages, and frameworks employed by contemporary realistic web applications would be an herculean task, and moreover would hide in an unnecessary amount of details the interesting points that we want to highlight. Hence, in the following we will make somewhat liberal use of descriptions in natural language for clerical operations, and of text in this style to indicate non-trivial operations that, however, have found no place in our effort, as being out of scope for the present work. One could think of such fragments as of undefined macros of which we even omit the name and signature, relying on the text to describe their purpose.

We will also use at times *meta-variables*, denoted in this style, to indicate a family of proper syntactic elements whose identifier is built by replacing the meta-variable with a value from a given set. So, for example, a family of predicates *has*Attrib : *Element* → *Boolean* would stand for the whole set of predicates *hasName*, *hasId*, *hasStyle*, *hasSrc* etc. Other notation is as popularized by standard ASM practice, e.g. as used in [1].

3 Transport Layer

3.1 Channels and Buffers

At the basis of the transport layer we have the concept of *Channel*s. A Channel consists of a pair of queues called *Buffer*s, each Buffer serving as a send-queue for the sending machine, and as a receive-queue for the receiving machine. There is an underlying expectation that what is written in the send-queue on one end of a channel, will appear, in order, in the receive-queue on the other end of the channel, and vice versa. However, this is not specified in our model, and in fact which data is read from a channel is, formally, totally non-deterministic. This allows us to reason on fringe cases, including communication errors, dropped connections, man-in-the-middle attacks, transparent and filtering proxies along the route, browser plug-ins to remove advertisement, antivirus and anti-scam software, parental filters, etc.

It is interesting to notice that the existence of all those potential intermediaries in practice makes TCP/IP's guarantee of "a data packet will arrive either intact and in order, or not arrive at all" [3] inapplicable in our case. In allowing for non-determinism in data transfer, we explicitly model our renounce to that guarantee.

Buffers transfer data as sequences of octets (bytes), but to simplify our models, we will assume that our background contains functions to turn these sequences of octets into the corresponding abstract types. So, for example, we will assume the ability to recognize a whole HTML element such as `<FONT size=1>` without going into the details of how the character sequence is transformed into an HTML element. Similarly for other data types (images, scripts, etc.).

On Buffers, we assume (as part of our background) the following operations:

- The macro

  TCPSend(*host*, *data*, *buffer*)

  will initiate a network transfer of the given *data* to the *host* (which includes address and port), preparing to receive a reply, if any, through the *buffer*. It models the act of creating a socket, binding it to an address and writing a data packet to the socket, and eventually reading the reply in a given memory buffer.
- x*Available* : *Buffer* → *Boolean* a family of predicates that return true if a full data element of type x is available for reading from the head of a buffer, or false otherwise.
- *head*X : *Buffer* → X a family of functions that return the data element of type X that is available for reading at the head of a buffer, or **undef** if no data is available.
- The command

  dequeue *e* **from** *buffer*

  is used to remove data element *e* from the head of the buffer (thus updating the buffer).

- *isFinished* : *Buffer* → *Boolean* a predicate that is true if the transfer associated to a buffer is finished and no more data have to be expected. This situation corresponds in actual implementation to a `close` operation on a TCP socket.

Notice that our transport-level background is fit to describe full TCP or UDP exchange, whereas the macros and functions defined above are sufficient to describe typical HTTP interactions (from the browser's perspective).

3.2 HTTP Request/Response

The HTTP protocol specifies that Requests should be sent to compatible servers in a specific format. First is a compulsory *request line* including a *method* (one of GET, POST, PUT, HEAD, and a handful of others), a *resource* (most typically, a pathname with optional query parameters), and protocol versioning information. The request line is followed by a (possibly empty) sequence of *headers* each of which is a pair $(key, value)$, and by an optional *body* (which is relevant only for the PUT method), containing arbitrary data to be processed by the server. Headers and body are separated by an empty line.

In the following we associate a unique identifier k to each request/response pair; we will see later how k can also serve to associate a request/response pair to higher-level operations and data. Moreover, since network transfers happen asynchronously, we use a *callback pattern*, where the response to a given request will be processed at some future time by a machine *proc* which is provided with the request.

The syntactical details of how a Request is structured need not concern us here; the actual sending of a request is modeled through the following macro, where *host* represent the (abstract) network identity of the machine that will receive the request, *head* includes the request line and headers, and *data* is as defined above:

$\textsc{Send}(host, head, data, proc, k) =$
  let *buffer* =**new** *Buffer*, *a* =**new** *Agent* **in**
    $ag(k) := a$
    $buf(k) := buffer$
    $\text{TCP}\textsc{Send}(host, head \cdot EMPTYLINE \cdot data, buffer)$
    $mode(k) := ExpectStatus$
    $program(a) := \textsc{Receive}(proc, k)$

The macro above creates a new buffer to hold the server response, constructs an HTTP Request by joining head and body, and more importantly creates a new agent whose task is to (eventually) process the response through the callback *proc* (once completed, the agent will terminate):

$\textsc{Receive}(proc, k) =$
  if $mode(k) = ExpectStatus$ **then**
    if $lineAvailable(buf(k))$ **then**

```
        let l = headLine(buf(k)) in
          dequeue l from buf(k)
          mode(k) := ExpectHeader
          status(k) := l
    if mode(k) = ExpectHeader then
      if lineAvailable(buf(k)) then
        let l = headLine(buf(k)) in
          dequeue l from buf(k)
          if isEmptyLine(l) then mode(k) := ExpectData
          if isSetCookie(l) then ∀cookie ∈ l, StoreCookie(cookie, rurl(k))
          else [manage other headers, e.g. for cache control]
    if mode(k) = ExpectData then
      proc(k)
```

The program for processing the data portion of the response is provided by the caller of the macro (and hence, eventually, by the initiator of the transfer). All elements are bound together through the unique key k, that serves as the unique identifier for this particular HTTP interaction. Notice how Receive stores any cookie sent by the server in a global (undescribed here) storage, whence they will be retrieved by the *cookiesFor*() function used in the next macro.

A full HTTP transfer is initiated by invoking the Transfer macro below:

```
Transfer(method, url, data, proc, k) =
  rmethod(k) := method
  rurl(k) := url
  rdata(k) := data
  if protocol(url) = http then
    let cookies = cookiesFor(url),
    hheader = makeHeader(method, url, cookies),
    hdata = makeData(data),
    host = addressFor(url) in
      Send(host, hheader, hdata, proc, k)
  else
    [other forms of transfer, e.g. file, ftp, etc.]
```

The macro first saves the parameters that characterize the request into state location indexed by the unique key k (for possible later reference, e.g. in error messages), then obtains the set of stored cookies that match the given URL (we will see later how these cookies are established); finally it builds the HTTP header by combining the method, the URL, and the cookies via the *makeHeader* function, and analogously builds the body of the request via the *makeData* function (which in real implementations performs, among other processing, the base-64 encoding of the binary data provided with the request). The destination host is identified by parsing the provided URL via the *addressFor* function, then the full HTTP request is sent to the network as described above.

Notice that our Transfer is a simplified version of the fetching algorithm described in full in [5, §2.7].

We hide here the details of how cookies are stored and retrieved by the browser (e.g., in a file on the user's home directory) into abstract functions and macros, yet it might be noticed that we are assuming a locking mechanism for the cookie storage, provided by the underlying file system or operating system, since multiple transfers can be occurring at the same time. In practice, most browsers in widespread use would rely on file system locks to ensure that multiple threads concurrently trying to write and retrieve cookies from a common storage would not interfere with each other in unexpected ways.

4 Stream Layer

The transport layer described how HTTP requests are sent out, and how responses are streamed into a Buffer. Here we describe how these streams are interpreted upon reception by showing a number of *stream processor* sub-machines. These machines receive incoming data in a buffer, in a streaming fashion (that is, the data is made available piecemeal, as soon as it is obtained from the network), and they incrementally process it in a variety of ways.

We split each stream processor in two layers: the first discriminates between the various *return codes* returned in the response, and – depending on whether the request was successful, or an error was returned, or other special actions need to be taken – executes the appropriate rule. In case of a successful request, the actual processing of the data returned is delegated to a specialized *parser*.

4.1 HTML Streams

The most important of these processors is the HTML one, whose main task is to parse an HTML document and build the corresponding DOM (Document Object Model).

HTML Processor. The HTML processor dispatches the handling of successful transfers to an HTML parser (described in the next section), whereas error codes are handled by an abstract macro that we will not further detail (typically, a synthetic "error page" is presented to the user; this would be simply modeled by replacing the current Document with a prepared one), and redirections are processed by restarting the transfer with a new URL (which is provided as part of the response itself).

```
HTMLProc(k) =
  if isSuccessCode(status(k)) then
    HTMLParser(k)
  elseif isErrorCode(status(k)) then
    HandleHTMLError(k)
  elseif isRedirectCode(status(k)) then
    RestartTransfer(k)
  else
    handling of other return codes
```

HTML Parser. In the following, we will assume the existence of a *DOM Tree*, whose elements are *Node*s. An exact specification of the contents of this tree, and of how the various nodes are build, is outside the scope of this document; the interested reader can however refer to [5, §1.8] and [5, §2.1.3] for a quick introduction. Here, we assume that navigation functions (dynamic functions such as *parent*(), *firstChild*(), *nextSibling*(); derived functions such as *root*(), *lastChild*(), etc.) are always available and describe the intended structure of the tree. We also assume that there is a *current tree* and a *current node* while the tree is being built; the abstract macros AddText, AddChild etc. modify the node data and navigation functions of the current tree as expected (these macros are detailed later).

Finally, we assume a range of functions over nodes to access their attributes and embedded content (e.g., the text contained in a CTEXT node); their usage in the following will be clear from context.

The machine below highlights three aspects of the HTML parsing process (which are among the most relevant ones for web applications): building the DOM tree, loading further resources, and executing scripts. These aspects will be illustrated in the following subsections. We will instead glide over other issues such as handling of malformed content, converting different character encodings, and applying style sheets, since these do not normally[2] affect the execution of well-behaved web applications.

The parser for HTML contents will thus be:

```
HTMLParser(k) =
  if ¬paused(k) then
    if textAvailable(buf(k)) then
      let t = headText(buf(k)) in
        dequeue t from buf(k)
        AddText(t, curNode(k))
    if tagAvailable(buf(k)) then
      let e = headTag(buf(k)) in
        dequeue e from buf(k)
        if isOpeningTag(e) then
          let n = newNodeFor(e) in
            AddChild(n, curNode(k))
            if ¬isClosingTag(e) then curNode(k) := n
            match e
              case <SCRIPT src=url> :
                Transfer(GET, url, ⟨⟩, ScriptProc, n)
              case <IMG src=url> :
                Transfer(GET, url, ⟨⟩, ImageProc, n)
```

[2] Notice that techniques such as using a style sheet to hide a certain UI component, thus preventing the normal user from issuing certain UI commands to the application, are not to be considered among the best practices. In fact, user agents (such as web browsers) can ignore or allow the user to override such style specifications, regaining control of the hidden elements, to unforeseen effects.

```
            case <LINK rel=rel src=url> :
              if "stylesheet" ∈ rel then
                TRANSFER(GET, url, ⟨⟩, STYLESHEETPROC, n)
            case [similar nodes that require a background transfer]
      if isClosingTag(e) then
        if ¬isOpeningTag(e) then curNode(k) := parent(curNode(k))
        match e
          case </SCRIPT> :
            if isAsync(curNode(k)) then STARTASYNC(curNode(k), k)
            elseif isDeferred(curNode(k)) then ADDDEFERRED(curNode(k), k)
            else RUNIMMEDIATE(curNode(k), k)
          case [similar nodes which require post-processing]
if isFinished(buf(k)) then
  if hasDeferred(k) then
    RUNDEFERRED(k)
  else
    FINALIZELOADING(k)
    program(self) :=undef
```

Building the DOM Tree. In building the DOM tree, the HTMLPARSER assumes that, at every instant, there is a current node $curNode(k)$ (where k is the unique instance token of the parser) which is the parent of the content that is currently being parsed. For example, in the following stream of HTML code, we have marked below each element the corresponding *curNode* at the time of parsing the element:

```
<DIV><FONT size=1>Sample <B>text</B> here</FONT></DIV>
     <DIV>        <FONT size=1>     <B>    <FONT size=1>  <DIV>
```

For a new transfer, $curNode(k)$ is initialized at the root element of an empty DOM (this is performed by the browser when, for example, opening a new tab).

The HTML parser uses a number of predicates and functions which are informally defined in the following.

- ADDTEXT($text, node$) appends a section of *text* to the contents of a *node*.

  ```
  ADDTEXT(text, node) =
    if ctext(node) =undef then
      ctext(node) := text
    else
      ctext(node) := ctext(node) · text
  ```

  This is most commonly the case for the content text of web pages. Notice that the text will be appended, in order, in chunks as it arrives from the network. Also, we assume that *headText* takes care of converting named entities (such as `&eacute;`) into the corresponding symbol (`é`) and of other encoding conventions, including tokenization as documented in [5, §8.2.4].

- *isOpeningTag* : *Element* → *Boolean* and *isClosingTag* : *Element* → *Boolean* are two predicates that indicate if a given tag is an opening tag (e.g., `<FONT>`) or a closing tag (e.g., `</FONT>`); notice that both could be true of the same tag (e.g., `<IMG ... />` or empty elements such as `<BR>`), whereas at least one of the two must be true for any given tag.
- *newNodeFor* : *Element* → *DOMNode* builds a fresh node, setting appropriate dynamic functions based on the supplied element, so that later it will be possible to retrieve the tag name, the value of its attributes, etc.
- AddChild(*child*, *parent*) add the *child* node to the DOM tree, as the last child of *parent*.

  AddChild(c, p) =
  if *firstChild*(p) =**undef then** *firstChild*(p) := c
  else let *last* = *lastChild*(p) **in**
  nextSibling(*last*) := c
  parent(c) := p

- The **match** construct we use is intended as a short-hand for compare & bind sequences. For example,

  match e
  case `<SCRIPT src=`*url*`>` : ...

  is a shorthand for

  if *tagName*(e) = `SCRIPT` ∧ *hasAttribute*(e, `src`) **then**
  let *url* = *valueOfAttribute*(e, `src`) **in** ...

The process we described in HTMLParser suffices for our purposes, but the reader should keep in mind that the full specification for building the DOM tree in [5, §8.2.5] includes a large number of other special cases (which, however, do not influence the execution of web applications, and hence are out of scope for the present work).

Loading of External Resources. While most HTML elements include references to external resources, only a few of the latter are automatically loaded at the same time as the page itself is. This is in particular the case of images, scripts, and style sheets, and also of less-used resources such as audio, video, embedded objects, etc.

These cases are handled by the **match** construct in HTMLParser. When a tag that requires background loading of further resources is encountered, the Transfer macro is invoked, retrieving the given URL and processing the retrieved contents through the appropriate parser.

It is important to stress that these transfers happen "in the background", without pausing or interrupting the transfer of the main HTML page that is being performed. In fact, after the HTTP request to retrieve them has been sent out, the processing of the results is delegated to a new agent, different from the one that is processing the page. These agents receive as their context (through the unique transfer key) the DOM node that caused the transfer, n. Based on our model, n is not yet added to the DOM tree when the Transfer macro is called

(since its execution happens in the same step as the execution of AddChild), but it will be properly installed by the time the response for the transfer is parsed, since that will happen in subsequent steps[3].

Script Execution. The last element in the HTML parser concerns the execution of scripts encountered in the page, either as embedded scripts, or referenced through an external URL.

Traditionally, script execution in HTML pages was strictly *serialized*, and neither the Javascript language, nor any interpreter in common use, supported any form of multi-threading. Moreover, execution was *blocking*: since a script could generate parts of the document on-the-fly, which were to be textually inserted immediately after the script itself, and before any other contexts, or even cause a redirection – thus halting the loading of the page entirely –, it was not possible to overlap execution and page loading.

The current HTML standards however, allows three different modes of script execution: *synchronous*, *asynchronous* and *deferred*. While the details of the different modes will be illustrated later, [5, §1.5.1] notes the following in a *non-normative* section:

> To avoid exposing Web authors to the complexities of multithreading, the HTML and DOM APIs are designed such that no script can ever detect the simultaneous execution of other scripts. Even with workers, the intent is that the behaviour of implementations can be thought of as completely serializing the execution of all scripts in all browsing contexts.
> The `navigator.yieldForStorageUpdates()` method, in this model, is equivalent to allowing other scripts to run while the calling script is blocked.

Immediate execution. The first mode of execution (and the default one, absent the `async` or `deferred` attributes of the `<SCRIPT>` tag) is the synchronous or *immediate* execution.

RunImmediate($node, k$) =
  $paused(k) := true$
  match $type(node)$
    case `text/javascript` :
      ECMAScriptInterpret($contents(node), node,$ RunCompleted$, k$)
    case specific versions and other languages handled similarly

There is a need to pause the HTML parser while executing a script in immediate mode. In fact, due mostly to historical legacy from the initial implementations of Javascript, it is possible to write from a script parts of the document to be parsed, i.e. generate dynamically the page itself (including, possibly, further `<SCRIPT>` tags that would then be executed in turn). While this technique offers significant flexibility, at the same time it clearly impedes continuing the parsing

[3] In particular, it cannot happen in the current step since the agent having the parser as its program will not exist yet.

of the page till the execution of the script is complete. This is obtained in our model by pausing the parser via the *paused*(k) function.

Along the same lines, there is a need to restart the parser once the execution is complete. This is obtained by passing to the interpreter a *callback* macro and a token parameter k (this is the same technique that we used in RECEIVE). In this case, the callback will be

RUNCOMPLETED(k) =
  buf(k) := *documentWriteBuffer*(k) · *buf*(k)
  paused(k) := *false*

The typical method for generating HTML contents to be injected into the page from a script is through the `document.write()` method (see [5, §3.5.3]). While the specification more accurately describes the processing to be performed in this case, we will be satisfied by postulating that all output generated by `document.write()` and `document.writeln()` during the executing of a given script is collected, in order, in *documentWriteBuffer*(k) and prepended to *buf*(k) at the end of the execution of the script, but prior to resuming parsing the HTML source.

It is worth remarking that the ECMASCRIPTINTERPRET might have to wait till the script has finished loading prior to actually starting the execution, in case its source text is obtained by a TRANSFER (this will be better illustrated in 4.2).

Deferred execution. Deferred execution consists in postponing the execution of a script until the page is fully loaded. This is accomplished by storing in a set of pending scripts the information needed for later execution:

ADDDEFERRED(*node*, k) =
  enqueue *node* **to** *deferred*(k)

The predicate *hasDeferred*() tells whether a certain document has pending scripts:

$$hasDeferred(k) \;=\; deferred(k) \neq \emptyset$$

In that case, the scripts are executed, sequentially and in-order, at the end of page loading and parsing:

RUNDEFERRED(k) =
  let *node* = *head*(*deferred*(k)) **in**
    dequeue *node* **from** *deferred*(k)
    RUNIMMEDIATE(*node*, k)

Given our previous definitions, this is sufficient to pause the parser (and, with it, the initiation of further executions from the deferred set), and properly serialize the execution of all pending scripts until the *deferred*(k) queue is empty.

We realize here what is prescribed in [5, §4.3.1], point 13 (with the minor simplification of not considering the parser-inserted flag):

> If the element has a `src` attribute, and the element has a `defer` attribute, and the element has been flagged as "parser-inserted", and the element does not have an `async` attribute: The element must be added to the end of the list of scripts that will execute when the document has finished parsing associated with the Document of the parser that created the element.

and then by [5, §8.2.6], point 3, (again, with the minor simplification of not considering here the event loop spinning, since we do not describe the dispatching of user input in this paper — but see [2] for the structure of EventLoop in our model):

> If the list of scripts that will execute when the document has finished parsing is not empty, run these substeps:
>
> 1. Spin the event loop until the first script in the list of scripts that will execute when the document has finished parsing has its "ready to be parser-executed" flag set and there is no style sheet that is blocking scripts.
> 2. Execute the first script in the list of scripts that will execute when the document has finished parsing.
> 3. Remove the first script element from the list of scripts that will execute when the document has finished parsing (i.e. shift out the first entry in the list).
> 4. If the list of scripts that will execute when the document has finished parsing is still not empty, repeat these substeps again from substep 1.

Asynchronous execution. For asynchronous scripts, we need not pause the HTML parser; script execution and further parsing of the HTML document can proceed concurrently, subject to proper mutual exclusion when accessing shared data (mostly, the DOM tree itself). In our model, the locking protocol is abstracted into the AddText and AddChild macros used by the parser.

As a result, starting an asynchronous script execution is remarkably similar to the immediate execution, with the provisio that the parser is not paused. Notice that, since ECMAScriptInterpret provides its own agent to execute the interpreter, no agent creation is needed here.

```
StartAsync(node, k) =
  match type(node)
    case text/javascript :
      ECMAScriptInterpret(contents(node), node, skip, k)
    case [ specific versions and other languages handled similarly ]
```

Final Processing. When the entire DOM tree has been created and all deferred scripts have finished execution, the parser *fires* a number of events, namely: `DOMContentLoaded`, `load`, `pageshow`[4].

This processing is abstracted in the FinalizeLoading macro that we do not describe here (again, these events are enqueued at the main event loop of the browser). Full details are in [5, §8.2.6].

[4] For malformed documents, that we do not consider here, further processing is performed to close all unclosed tags, firing corresponding `popstate` events.

4.2 Script Streams

Script streams are used whenever the browser needs to load the source code for a script from a remote URL, which typically happens when a `<SCRIPT src=`*url*`>` element is processed while parsing HTML data.

Script Processor. Processing scripts is similar to other forms of stream processing, in that the data is accumulated in case of a successful transfer, whereas errors (e.g., attempts to load a script from a non-existing URL) will simply result in an empty content. This is different from the behaviour of the HTML processor, which would notify the user in case of a failure in loading a page.

```
ScriptProc(k) =
  if isSuccessCode(status(k)) then
    ScriptParser(k)
  elseif isErrorCode(status(k)) then
    programText(k) := ""
  elseif isRedirectCode(status(k)) then
    RestartTransfer(k)
  else
    handling of other return codes
```

Script Parser. The script parser is invoked to process the source text of a script, while it is being received from the network following an occurrence of a `<SCRIPT src=`*url*`>` tag in the page.

Although the precise rules for the encoding of script source text are a little different than those for general text, as specified in [5, §4.3.1.2], we will simplify the matter here[5] and use the same functions we already use for general text, as follows:

```
ScriptParser(k) =
  if textAvailable(buf(k)) then
    let t = headText(buf(k)) in
      dequeue t from buf(k)
      programText(k) := programText(k) · t
  if isFinished(buf(k)) then
    complete(k) := true
    program(self) :=undef
```

Notice that in our model the execution of a script can be *started* while the script is still loading, but will not progress until *complete*() signals that loading has finished, and the full text of the program is available. The specification allows for varied behaviour in this respect (e.g., an implementation could start building the parse tree incrementally while the loading is still in progress, or postpone any processing to after the full program has been received).

[5] Notice that we already applied the same principle in loading in-line script source as text for `<SCRIPT>` nodes without the `src` attribute.

4.3 Image Streams

Image streams are used when receiving images from the server, as part of a web page, or via instantiation of the corresponding classes in the Javascript library.

Image Processor. The behaviour in case of a successful transfer is analogous to that of previous stream processors (i.e., the Image parser is run). Erroneous cases are handled differently, i.e. by substituting a pre-defined error image (e.g., a large red X or an icon depicting a broken link) for the missing image.

```
ImageProc(k) =
  if isSuccessCode(status(k)) then
    ImageParser(k)
  elseif isErrorCode(status(k)) then
    imgData(k) := errorImgData
  elseif isRedirectCode(status(k)) then
    RestartTransfer(k)
  else
    handling of other return codes
```

Image Parser. The parsing of image data is often done incrementally, in order to properly implement so-called *progressive* image formats (i.e., when data are arranged in such a way that it is possible to construct a low-quality version of an image early in the loading stage, and then refine that to better resolution or colour depth as more data arrives) or to show load progress (i.e., by updating the rendered image on a scan-line basis as soon as data is available). We abstract from all the details of various image formats, and from how the browser distinguishes them based on their MIME types. The only aspect that we want to highlight is the progressive nature of the loading, since it expresses the fact that the graphical user interface of the web application may be not fully loaded when the application code starts executing.

The general process of loading an image is thus described as follows:

```
ImageParser(k) =
  if dataAvailable(buf(k)) then
    let d = headData(buf(k)) in
      dequeue d from buf(k)
      imgData(k) := imgData(k) · data
  if isFinished(buf(k)) then
    program(self) :=undef
  UpdateImage(k)
```

Here, we assume that UpdateImage(k) will perform any needed update to the internal data structures holding the actual image, based on $imgData(k)$. Notice that the macro is invoked at each step even if no new data has been received; this allows the implementation to perform any timed decoding or animation (e.g., a hourglass or spinning circle or progress bar) to indicate loading progress.

5 Conclusions

We have presented in this paper two main contributions.

On one hand, we have leveraged the solid semantics foundation of ASMs to provide a precise model of certain aspects of concurrency in contemporary web browsers, namely how the construction of the DOM and the execution of scripts happen in a streaming, concurrent and asynchronous manner while a page is loading. In particular, we have shown how the various parts of the DOM are retrieved concurrently by multiple agents from multiple servers, and how the DOM itself is progressively constructed (including running asynchronous script code). This model has, in itself, an explicatory and teaching value, and could also be used to support the discussion of those facets of the HTML 5 specification which are left somewhat vague (i.e., as non-normative sections). Moreover, as illustrated in the companion paper [2], it can be used together with a corresponding model of a web server as a basis to prove properties of web application frameworks.

On the other hand, we have shown the usefulness of a little used technique for controlling the execution of multiple agents in distributed ASMs, namely that of spawning a new agent while passing to it, as argument, a callback macro to be executed on completion. We have used this technique thoroughly in SEND, RECEIVE, TRANSFER and, as a consequence, in all the parsers and processors for various streams. This proof of usefulness supports the importance of considering ASM rules (or submachines) as proper *values* in the ASM universe, especially in executable versions of the language. This, in turn, questions the expediency of the traditional view of rules-as-macros whereas the name of the macro is replaced by an expansion of its body. Our usage of rule names is instead closer to that of CoreASM [4], whereas a *Rules* universe, populated by each and every rule defined in an ASM, is assumed to exist in the background.

Acknowledgement. The author would like to thank Egon Börger for his encouragement in writing this paper, and for reviewing an early draft of the same.

References

1. Börger, E., Stärk, R.F.: Abstract state machines: a method for high-level system design and analysis. Springer (2003)
2. Börger, E., Cisternino, A., Gervasi, V.: Contribution to a Rigorous Analysis of Web Application Frameworks. In: Derrick, J., et al. (eds.) ABZ 2012. LNCS, pp. 1–20. Springer, Heidelberg (2012)
3. Cerf, V.G., Dalal, Y., Sunshine, C.: RFC675: Specification of Internet Transmission Control Program (December 1974)
4. Farahbod, R., Gervasi, V., Glaesser, U.: CoreASM: An extensible ASM execution engine. Fundamenta Informaticae 77, 71–103 (2007)
5. World Wide Web Consortium. HTML 5: A vocabulary and associated APIs for HTML and XHTML W3C Working Draft (October 19, 2010), `http://www.w3.org/TR/html5`

Modeling the Supervisory Control Theory with ALLOY*

Benoît Fraikin, Marc Frappier, and Richard St-Denis

Département d'informatique,
Université de Sherbrooke,
Sherbrooke (Québec), J1K 2R1, Canada
{Benoit.Fraikin,Marc.Frappier,Richard.St-Denis}@USherbrooke.ca

Abstract. Scientific literature reveals that symbolic representation techniques behind some formal methods are attractive to synthesize parts or verify properties of large discrete event systems. They involve, however, complex encoding schemata and fine tuning heuristic parameters in order to translate specific problems into efficient BDD or SAT-based representations. This approach may be too costly when the main goal is to explore a theory, understand by simulation its underlying concepts and computation procedures, and conduct experiments by applying them to small problems. Based on previous work with ALLOY on the synthesis of observers and nonblocking supervisors of a system organized hierarchically with a flat state space estimated to 10^{31} states, this paper investigates more deeply issues raised with its use in the modeling and prototyping of the supervisory control theory, including the application of models to practical problems. This study was conducted in a broader context than just hierarchical control since it embraces various variants of this theory.

Keywords: ALLOY, KODKOD, bounded model checking, SAT-solver, supervisory control theory, controllability, normality, *N*-inference observability, observational equivalence.

1 Introduction

Novice researchers face numerous challenges in learning a new theory for the first time, particularly if their mental representations of knowledge acquired in their previous learning do not correspond to those required to grasp astonishing concepts. Developing an abstract model of a theory in a declarative manner with ALLOY [8] provides an interactive simulation platform that can be used to explore various instances of the model. As far as people become familiar with concepts and acquire a deep intuition of the theory, they can explore the abstract model with their own instances, even adding new relationships between concepts, in order to get solutions to practical problems, and ultimately extend the theory. Indeed, prototyping and applying a theory in this way constitutes a source of motivation and creativity to identify new problems and solve them, especially when the ALLOY specification has roughly the same size as the mathematical formulation. This paradigm shift contrasts with the usual approach that consists in

* The research described in this paper was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC).

J. Derrick et al. (Eds.): ABZ 2012, LNCS 7316, pp. 94–107, 2012.

codification of synthesis or verification procedures in a conventional programming language or modification of open-source tools. In fact, these actions require manipulation of complex data structures and efforts to understand more than ten thousands of lines of code (versus hundreds written in the Alloy declarative language).

This paper investigates the aforementioned approach throughout a control theory for discrete event systems (DES), called the *supervisory control theory* (SCT), which was formulated by Ramadge and Wonham in the last decades of the 20th century [14]. In this theory, a control problem is stated w.r.t. a given system architecture. Its constituent units and underlying attributes are represented by mathematical objects (e.g., formal languages over an event alphabet), which are used to define properties (e.g., controllability, normality) to be satisfied by the desired behavior. Then, based on these properties, necessary and sufficient conditions are formulated for the existence of supervisors. When these conditions are not fulfilled, the infimal or supremal element of a family of languages satisfying these conditions (if such an element exists) is considered to solve the problem. Then, synthesis algorithms allow for the automatic derivation of supervisors from mathematical models of DES often expressed using automata for practical reasons [9]. Finally, translation procedures specific to code generation for software controllers from supervisors can be exploited to obtain executable solutions that are correct by construction. After many years of effort, several system architectures (e.g., distributed, horizontal and vertical, conditional) and control patterns (e.g., decentralized, hierarchical, multi-decision) along these lines have been explored with success (e.g., [15,7,3]).

Compared to BDD-based techniques, which have gained widespread use in the SCT community, little effort seems to have been invested in bounded model checking with SAT-solvers to verify properties formulated in the context of SCT or generate automatically nonblocking supervisors. This idea has been mentioned by Ma and Wonham [12], but Claessen et al. [4] were the first to show how to encode controllability property and deadlock freedom, together with the transition functions of automata modeling the plant and control specification, as propositional formulas with the aim of checking their satisfiability w.r.t. the plant and control specification. They did not, however, provide experimental data on systems with significant size. They also proposed a synthesis process via an iterative specification refinement without giving any concrete implementation. To avoid complex encoding schemata, Côté et al. [5] used Alloy to verify several properties defined in the hierarchical control architecture framework of SCT [17] in order to generate observers and supervisors associated with small reusable components. Indeed, among 44 components of a modular production system, 75 % of those components have been successfully checked by the Alloy analyzer, the other 25 % could not because their state space was too large. Without exploiting this hierarchical framework, it would have been impossible to achieve such a level of scalability with Alloy. In this paper, we provide, among other things, solutions for some open issues raised in [5].

Based on these two studies [4,5], we drew the following conclusions. Firstly, properties to be satisfied are often expressed on languages, more often, infinite sets of words over an event alphabet. Since this form is not convenient for the SAT-based solving approach, they must be expressed on finite mathematical objects (e.g., automata, bounded-length words). There are properties for which such equivalent formulations over finite

sets can be found in the literature. When this is not the case, a manual proof is necessary to show that the new formulation is correct w.r.t. the given property (e.g., the observer property for which such a proof, missing in [5], is given in Section 3.4). Secondly, attempts to automatically generate supervisors were unsatisfactory because such processing includes the computation of a supremal element and there is no guarantee that the SAT-solver will find the optimal solution unless it is called inside an iterative procedure that checks if optimality has been reached (such a procedure is only mentioned in [5], but now described in Section 4). Thirdly, sometimes an iterative process has to take place when human intervention is required in order to obtain, for instance, a useful interface for a software component while satisfying specific properties prescribed by the theory. Such interventions arise between successive calls to the SAT-solver. Finally, SAT-solvers and Alloy impose limitations that impact on the way to conceive an abstract model (e.g., the maximum number of atoms allocated when quaternary relations are used instead of ternary relations), size of instances of control problems that can be processed (due, for example, to bit-width for integers) and CPU resources consumed to produce a solution (due to state space explosion). How to bypass these restrictions represents, yet again, a genuine challenge w.r.t. the current state of the art in the domain.

Besides being the first to model a fertile control theory with Alloy, this paper goes a step further. Based on a subset of SCT that is sufficiently representative to illustrate the aforementioned issues, it details solutions for them and suggests potential modifications to Alloy. It is organized as follows. Section 2 introduces typical control problems formulated within the framework of SCT with the aid of an example. It is useful for readers unfamiliar with this theory. Section 3 presents Alloy models of SCT, paying special attention on controllability, normality, N-inference observability and observational equivalence. Section 4 shows how to take advantage of Kodkod when Alloy is not powerful enough in itself to entirely solve a given control problem. Section 5 ends with a discussion about future work in regards to the advantages and limitations of Alloy as a tool for solving real problems in the context of SCT.

2 Typical Control Problems within the Framework of SCT

A typical problem in SCT consists in keeping the behavior of a given system within the limits imposed by operational constraints with the aid of a nonblocking supervisor, which is calculated from a model of system behavior and a control specification. Consider the injector of a modular production system from FESTO available in our laboratory. An injector consists of a jack strongly coupled with a cylinder barrel located in the basis of a gravity-feed magazine, which is monitored by a sensor to detect the arrival of a workpiece. The jack pushes the bottom workpiece out of the magazine and holds it on a plate until it is rearmed by the control logic.

Figure 1 depicts a Mealy machine that models the behavior of the injector under control. In this model, Xi ($i = 0, 1, 2$) and Y0 denote sensor inputs and an actuator command, respectively. The sensors, that detect the presence of a workpiece and end travel positions of the jack, are represented by the state variable X0 and two state variables X1 and X2, respectively, where $\overline{\text{X0}}$ is the state value of the first sensor when an object obstructs the beam of light (i.e., X0 = 0) and $\overline{\text{X1}} \wedge \overline{\text{X2}}$ holds when the jack is

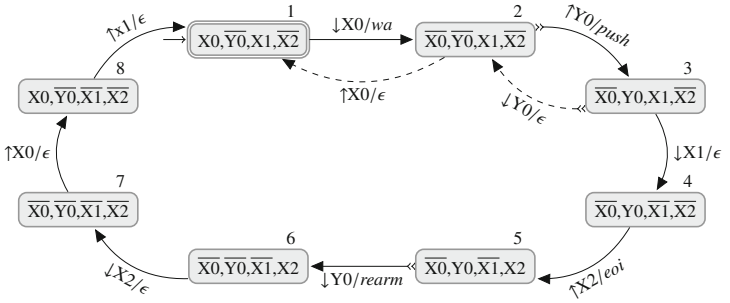


Fig. 1. The behavior of the injector under control

neither completely retracted nor fully extended. The state value $\overline{\text{Y0}}$ indicates that the jack is armed and in its unique stable position when X1 = 1 (i.e., completely retracted). Finally, the symbols ↑X*i* and ↓X*i* denote a signal rising edge and a signal falling edge, respectively. They represent uncontrollable events and belong to the input alphabet. Symbols ↑Y0 and ↓Y0 also belong to the input alphabet, but they stand for controllable events. The event ↑Y0 is enabled only when $\overline{\text{X0}} \wedge \overline{\text{Y0}} \wedge \text{X1}$ holds (i.e., presence of a workpiece in the barrel and the jack retracted in its stable position) and ↓Y0 is enabled when $\text{Y0} \wedge \text{X2}$ holds (i.e., the jack is fully extended at the end travel position outside the magazine). It should be noted that the transition from state 2 (labeled $\langle \overline{\text{X0}}, \overline{\text{Y0}}, \text{X1}, \overline{\text{X2}} \rangle$) to state 1 (labeled $\langle \text{X0}, \overline{\text{Y0}}, \text{X1}, \overline{\text{X2}} \rangle$) on event ↑X0 (indicated by a dashed arrow) cannot be disabled since the event ↑X0 is uncontrollable. Nonetheless, such transitions have been experimentally identified as highly unlikely to occur, otherwise the specification (*the injector is activated only when there is a workpiece in the barrel*) represented by a formal language would be uncontrollable w.r.t. the exhaustive behavior of the injector and set of controllable events. Also the supremal controllable sublanguage would be empty in this specific case. The transition from state 3 (labeled $\langle \overline{\text{X0}}, \text{Y0}, \text{X1}, \overline{\text{X2}} \rangle$ to state 2 on ↓Y0 (indicated by a dashed arrow) can be inhibited since this event is controllable. Apart from these two dashed transitions, all nonessential transitions have been omitted in the graph for clarity. Verifying the controllability property and synthesizing the supremal controllable sublanguage of a given language are essential tasks to achieve an optimal control.

Sometimes some events are unobservable by the supervisor because of lack of sensors. Partial observation is captured by defining a mask which associates each event with an observed event or ϵ when the event is unobservable. Additional properties must then be considered during the calculation of supervisors: observability and normality, which is stronger than observability to ensure the existence of a supremal element. Another type of mask, called a causal map, is used to abstract the controlled behavior with the aim of providing an interface when the system is considered as a component. Such a causal map must satisfy the observer property if components must be integrated into a nonblocking hierarchical system. The controlled behavior of a component (i.e., the agent under control) and the causal map are represented by a Mealy machine like the one of Figure 1, where the symbols *push*, *eoi*, *rearm* and *wa* belong to the output

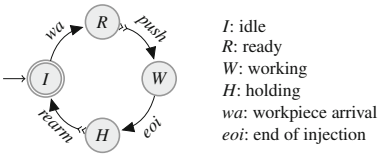


Fig. 2. The interface of the injector

alphabet. Applying this map yields the interface of Figure 2, which is the only visible part outside the component. In this interface, the events *push* and *rearm* are commands (or controllable events). Thus, the interface makes it possible to push a workpiece on a plate for an eventual transfer to the testing station with the aid of a crane and retract the injector in its stable position when the workpiece has been grabbed by the crane. The Mealy machine and the automaton modeling the interface are defined as follows in ALLOY:

```
1  open Theta [State, In, Out, agent, interface] as TT
2  enum State {S1, S2, S3, S4, S5, S6, S7, S8, I, R, W, H}
3  enum In {rX0, fX0, rX1, fX1, rX2, fX2, rY0, fY0}
4  enum Out {wa, push, eoi, rearm, epsilon}
5  one sig lrX0, lfX0, lrX1, lfX1, lrX2, lfX2, lrY0, lfY0 extends TT/Label {}
6  fact                                          -- Causal map
7  {
8   inL  = lrX0 -> rX0 + lfX0 -> fX0 + lrX1 -> rX1 + lfX1 -> fX1
9        + lrX2 -> rX2 + lfX2 -> fX2 + lrY0 -> rY0 + lfY0 -> fY0
10  outL = lrX0 -> epsilon + lfX0 -> wa  + lrX1 -> epsilon + lfX1 -> epsilon
11       + lfX2 -> epsilon + lrX2 -> eoi + lrY0 -> push    + lfY0 -> rearm
12 }
13 one sig agent extends TT/IO/CAutomaton {}    -- Agent under control
14 {
15  states = S1 + S2 + S3 + S4 + S5 + S6 + S7 + S8
16  labels = lrX0 + lfX0 + lrX1 + lfX1 + lrX2 + lfX2 + lrY0 + lfY0
17  initialState = S1
18  finalStates = S1
19  transition = S1 -> lfX0 -> S2 + S2 -> lrY0 -> S3 + S3 -> lfX1 -> S4
20             + S4 -> lrX2 -> S5 + S5 -> lfY0 -> S6 + S6 -> lfX2 -> S7
21             + S7 -> lrX0 -> S8 + S8 -> lrX1 -> S1
22  controllable = lrY0 + lfY0
23 }
24 one sig interface extends TT/O/CAutomaton {} -- Interface
25 {
26  states = I + R + W + H
27  labels = wa + push + eoi + rearm
28  initialState = I
29  finalStates = I
30  transition = I -> wa  -> R + R -> push  -> W
31             + W -> eoi -> H + H -> rearm -> I
32  controllable = push + rearm
33 }
```

This specification uses the module `Theta` (line 1) suitable for handling the constituent elements of a component (e.g., the state space, input/output alphabets, automata for the agent under control and interface). The input/output alphabets are defined at lines 3 and 4, respectively. Line 5 contains the transition labels of the Mealy machine, which is defined at lines 13–23. The causal map defined at lines 6–12 associates an input symbol and an output symbol with each label. Finally, the deterministic automaton of the interface is at lines 24–33.

3 State-Based Formulation of Properties

For purposes of generality, the behavior of a DES is represented by the set of all its execution traces. This set defines a language L over an event alphabet Σ and $L = pr(L)$, where $pr(L)$ is the prefix closure of L. The traces in L that symbolize complete tasks are represented by the language $L_m \subseteq L$. Thus (L, L_m) is a language model of the DES. Furthermore, Σ is partitioned into two subsets: Σ_c, the set of controllable events, and Σ_u, the set of uncontrollable events.

3.1 Controllability Property

The controllability property appears in almost all control problems [14]. Given (L_m, L) the language model of a plant, a language $K \subseteq \Sigma^*$ is controllable w.r.t. L and Σ_u *iff*

$$(\forall s, \sigma \mid s \in \Sigma^*, \sigma \in \Sigma_u : s \in pr(K) \wedge s\sigma \in L \Rightarrow s\sigma \in pr(K)). \tag{1}$$

Intuitively, a language K, which represents the control specification, is controllable if any subtask of K followed by an uncontrollable event that is physically possible in L is also a subtask of K. If all the languages are regular, then two automata $G = (Q, \Sigma, \delta, q_0, Q_m)$ and $H = (X, \Sigma, \xi, x_0, X_m)$ can be used to represent the system behavior and control specification, respectively. Generally, H refines G. Therefore, there exists a correspondence function f between the states of H and states of G such that $f(\xi(x_0, s)) = \delta(q_0, s)$ with $s \in pr(K)$ [19]. Using this representation, is is easy to check the controllability property with Alloy:

```
1  pred controllability
2  {
3   all x1:H.states, q1,q2:G.states, e1:G.labels | let delta = q1->e1->q2 |
4       (e1 not in G.controllable && delta in G.transition && x1->q1 in f)
5   implies
6       (some x2:H.states, e2:H.labels | e1 = e2 &&
7        e2 not in H.controllable && let xi = x1->e2->x2 | xi in H.transition)
8  }
```

The state-based formulation of Property (1) imposes that for any state $x_1 \in X$ (all states are accessible by hypothesis) and uncontrollable transition in G from q_1 such that $f(x_1) = q_1$ (lines 3–4), the corresponding transition must be in H (lines 6–7). Otherwise K in uncontrollable because the transition from q_1 on e_1 cannot be disabled by control to achieve K.

Even though a counterexample (a bad uncontrollable transition) is found by the Alloy analyzer, which can be easily shown to the user with an appropriate layout theme, it could be cumbersome to iterate on bad transitions and remove them to obtain the supremal controllable sublanguage of K. A better solution consists in adopting a dual approach, called state-based control, in which the language K is replaced by a set of forbidden or *bad* states, and taking advantage of the reflexive transitive closure operator to recognize the *good* states (those of the supervisor). This solution also ensures that the supervisor is nonblocking (i.e., it can always complete any subtask):

```
1  pred goodStates[G:Automaton, unc:set Label, bad,good:set State]
2  {
3   let gT = good <: (G.transitionsOn[Label]) :> good,
```

```
4       uT = (G.transitionsOn[unc]) |
5       {
6        good in (G.initialState).*gT
7        no (*uT.bad & good)
8        all q : good | some (q.*gT & G.finalStates)
9        good.uT in good
10      }
11  }
```

The term gT defined at line 3 by using the domain restriction (<:) and range restriction (:>) operators represents the set of transitions between good states. The term uT defined at line 4 denotes the set of uncontrollable transitions. The good states are reachable from the initial state (line 6), coreachable to the set of final states (line 8) and closed under uncontrollable transitions (line 9). Furthermore, no walk of uncontrollable transitions leads to a forbidden state from a good state (line 7). It should be noted that $Q - bad$ is generally larger than the set of good states.

An important question remains open with respect to this solution. Does the instance found by Alloy (i.e., the set of good states) always correspond to the supremal solution? In theory, one must determine the exact number of atoms for each signature involved in the solution. On the one hand, if one guesses a higher number, no instance is found by Alloy. On the other hand, if one guesses a lower number, the particular instance, which corresponds to the supremal solution, may be not considered by Alloy. Even if the exact number of atoms is known, this method does not really work in practice because of the combinatorial explosion. The following method avoids these obstacles. Let S be the supervisor found by Alloy following the generation of a set of good states. Then S corresponds to the supremal controllable sublanguage of K if the analyzer does not find a counterexample that violates the following predicate (i.e., there exists another supervisor s with a larger set of states):

```
1  pred IsOptimalSupervisor[S:Supervisor, G:Automaton,
2                           unc:set Label, bad:set State]
3  {
4   S.isSupervisor[G, unc, bad]
5   all s : Supervisor | s.isSupervisor[G, unc, bad]
6           implies s.states in S.states
7  }
```

3.2 Normality Property

The normality property is useful when a supervisor partially observes the events generated by a plant due to the presence of faulty sensors or lack of sensors [9]. Given (L_m, L) the language model of a plant and an observation mask $M : \Sigma \to \Lambda \cup \{\epsilon\}$, a language $K \subseteq \Sigma^*$ is normal w.r.t. L and M *iff*

$$(\forall s, t \mid s, t \in L : s \in pr(K) \wedge M(s) = M(t) \Rightarrow t \in pr(K)). \tag{2}$$

The language K is normal if $pr(K)$ is the union of some subsets of L, the largest sets of words that are indistinguishable from each other under M. This property is equivalent to $M^{-1}M(pr(K)) \cap L \subseteq pr(K)$. In addition to G, H and f that represent the system behavior, control specification and correspondence function (again H refines G), a deterministic automaton C, such that $L(C) = M(K)$[1], is necessary to identify the transitions that do

[1] Recall that $L(C)$ is the language generated by the automaton C. Formally, $L(C) = \{s \in \Lambda^* \mid \zeta(z_0, s) \text{ is defined}\}$.

not have the following *normal* property. A transition of C, from a state z_1 to a state z_2, labeled with an event $\lambda \in \Lambda \cup \{\epsilon\}$, is normal if there is no state $x \in z_1$ (each state of C is a set of states of H) and event $\sigma \in \Sigma$ such that $\delta(f(x), \sigma)$ is defined, $M(\sigma) = \lambda$ and $\xi(x, \sigma)$ is undefined [1]. The translation of this property in Alloy is straightforward, but it requires a relation r (used at line 4) that associates to every state of C its corresponding subset of states that belongs to 2^X:

```
1  pred normality
2  {
3   all x1:H.states, z1,z2:C.states, e1:C.labels, e2:G.labels |
4       let zeta = z1->e1->z2 | (zeta in C.transition && z1->x1 in r
5                                                     && e2->e1 in M) &&
6        (some q1,q2:G.states | x1->q1 in f &&
7                               let delta = q1->e2->q2 | delta in G.transition)
8       implies
9        (some x2:H.states | let xi = x1->e2->x2 | xi in H.transition)
10 }
```

Finding an Alloy model that allows for the efficient retrieval of a deterministic automaton that generates $M(K)$ and the association between its states and those of H in order to remove nondeterminism in $M(H)$ is the key of a complete automatic verification procedure for normality. This remains an open issue because Alloy provides no native mechanism to handle the powerset of a set. It must be encoded in some way, which further makes complex the modeling of this aspect. To circumvent this problem, it is assumed that the specifiers explicitly provide C and r.

The normality property can be considered in addition to the controllability property in the derivation of a nonblocking supervisor. As in the case described in the previous subsection, this is done in the context of the state-based control paradigm. Therefore, the language K is replaced by a predicate P on the state space of G and the supremal controllable and normal predicate stronger than P is obtained from an iterative computational procedure [11]. This requires an extension of the predicate `goodStates`, including the predicate that checks optimality.

3.3 N-Inference Observability Property

The N-inference observability property plays a central role in decentralized control, where a global decision results from independent, local control decisions. In this framework, each inference-based local supervisor S_i ($i \in I := \{1, 2, \ldots, n\}$) has its own sets of controllable events $\Sigma_{ic} \subseteq \Sigma$ and observable events $\Sigma_{io} \subseteq \Sigma$, and $S_i : P_i(L) \times \Sigma_{ic} \to C \times \mathbb{N}$, where $P_i : \Sigma^* \to \Sigma_{io}$ is the natural projection that hides the event of $\Sigma - \Sigma_{io}$ (a particular case of observation mask) and $S_i(P_i(s), \sigma) = (c_i(P_i(s), \sigma), n_i(P_i(s), \sigma))$. More precisely, $c_i(P_i(s), \sigma) \in \{0, 1, \phi\}$ is the control decision of S_i for a locally controllable event σ following an observation $P_i(s) \in P_i(L)$ and $n_i(P_i(s), \sigma) \in \mathbb{N}$ is the ambiguity level of the control decision of S_i [10]. Let $In(\sigma) := \{i \in I \mid \sigma \in \Sigma_{ic}\}$. The global decision of the decentralized supervisor $\{S_i\}_{i \in I}$ is

$$\{S_i\}_{i \in I}(s, \sigma) := \begin{cases} 1, \ (\forall i : i \in In(\sigma) \mid n_i(P_i(s), \sigma) = n(s, \sigma) \Rightarrow c_i(P_i(s), \sigma) = 1) \\ 0, \ (\forall i : i \in In(\sigma) \mid n_i(P_i(s), \sigma) = n(s, \sigma) \Rightarrow c_i(P_i(s), \sigma) = 0) \\ \phi, \text{ otherwise} \end{cases} \quad (3)$$

where $n(s, \sigma) := (\min i : i \in In(\sigma) \mid n_i(P_i(s), \sigma))$ is the minimum ambiguity level of local decisions. Intuitively, a global control decision is the same as the one taken by the

local supervisors, for which the ambiguity level of the decision is the minimum, as far as everyone agrees on the decision. A nonblocking N-inferring decentralized supervisor $\{S_i\}_{i\in I}$ that acts on the system to achieve the control specification K exists if K is at least N-inference observable (i.e., $D_{N+1}(\sigma) = \emptyset$ or $E_{N+1}(\sigma) = \emptyset$ for all $\sigma \in \Sigma_c$). For instance, $E_0(\sigma) := \{s \in pr(K) \mid s\sigma \in pr(K)\}$ and $E_{k+1}(\sigma) := E_k(\sigma) \cap (\cap_{i\in In(\sigma)} P_i^{-1} P_i(D_k(\sigma)))$. These expressions define the set of words where σ must be enabled and a sublanguage of $E_k(\sigma)$ having words for which there exists a P_i-indistinguishable word in $D_k(\sigma)$ for each $i \in In(\sigma)$, respectively. In a first attempt to model this decentralized decision-making process, an Alloy model has been developed for exactly two local supervisors (i.e., $n = 2$). So the model is not parameterized for an arbitrary number of local supervisors. Such a reduced model has been elaborated very quickly (w.r.t. a program written in a conventional programming language) and used to explore with success a complex part of the theory. Using a state-based formulation, $E_0(\sigma)$ and $E_1(\sigma)$ become in Alloy:

```
fun getE0[event:one Label] : set State
{
 { x1:G.states | some x2:G.states-Bad, e:event |
                 let delta = x1->e->x2 | delta in G.transition }
}
fun getE1[event:one Label] : set State
{
 getE0[event]
 & { x1:G.states | some x2:getD0[event], y:H1.states |
                   y->x2 in r1 && y->x1 in r1}
 & { x1:G.states | some x2:getD0[event], y:H2.states |
                   y->x2 in r2 && y->x1 in r2}
}
```

The main reason why the inductive definition has been unwound is related to the computation of functions c_i and n_i, which explicitly use $E_0, E_1, \ldots, E_{N+1}$ and $D_0, D_1, \ldots, D_{N+1}$. The details are omitted due to space limitation.

For the case in which $n = 2$, the Alloy specification includes 19 functions (i.e., `fun`) for a total about 200 LOC (without the definition of automata). For instance $\{S_i\}_{i\in I}$ has been written as follows:

```
fun getSi_States[event:one Label, d:one Decision] : G.states
{
 { x:G.states | some k:Int | x->k in getn[event] &&
  (let y1 = r1.x | y1->k in getn1[event] implies y1->d in getc1[event]) &&
  (let y2 = r2.x | y2->k in getn2[event] implies y2->d in getc2[event]) }
}
fun getSi[event:one Label] : G.states -> Decision
{
 let enabled_states = getSi_States[event, enabled] |
 let disabled_states = getSi_States[event, disabled] |
 let unsure_states = G.states - (enabled_states + disabled_states) |
     enabled_states -> enabled + disabled_states -> disabled
                               + unsure_states -> unsure
}
```

It should be noted that the functions (e.g., c_i, n_i) returned by some `fun` have not been reified in the Alloy specification due to excessive execution time.

3.4 Observational Equivalence Property (Observer)

A causal map θ, as the one defined in Figure 1, involves information hiding and relabeling. It must satisfy the observer property, which is crucial for an effective refinement and

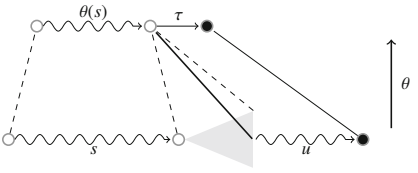


Fig. 3. The observer property

aggregation to achieve hierarchical consistency while preserving nonblockingness [17]. This property is closely related to the concept of observation equivalence defined by Milner [13]. Generally, θ is guessed based on a target abstraction of the system behavior or interface requirements. If it violates the observer property (i.e., it is not an observer), further vocalized transitions must be added, until the coarsest observer which is finer than the given causal map is obtained, thanks to the algorithm of Wong and Wonham [18] based on an efficient procedure for bisimulation equivalence [6]. It can then be used as is, but it usually serves as a heuristic concerning the modifications that should be made to the original causal map in order to make it into an observer. The observer property is illustrated in Figure 3 and formally defined as follows. The causal map θ is an observer *iff*

$$(\forall s, \tau \mid s \in L \wedge \tau \in T : \theta(s)\tau \in \theta(L) \implies (\exists u \mid u \in \Sigma^+ : su \in L \wedge \theta(su) = \theta(s)\tau)). \quad (4)$$

Intuitively, if a given behavior of the abstraction can be extended to an admissible word, no matter in which state the system is, as long as its image is $\theta(s)$, its behavior can be extended to a word that belongs to L with $\theta(su) = \theta(s)\tau$.

Again, this property needs to be reformulated in terms of automata to get it into a suitable form for bounded model checking. To the best of our knowledge such a formulation does not exist in the literature. To illustrate the versatility of Alloy, both languages L over Σ and $\theta(L)$ over T are represented by automata and walks in the transition graphs are considered. This contrasts with the encoding of the controllability and normality properties, in which the language K, that stands for the control specification, has been replaced by an equivalent set of forbidden states or a predicate. Thus, a word is defined as follows:

```
1  abstract sig Word
2  {
3   sequence : seq Label,         -- sequence of symbols
4   visitedStates : seq State,    -- sequence of states
5   wrtAutomaton : one Automaton -- automaton in which the walk takes place
6  }
```

Several functions and predicates have been defined to state the observer property as concisely as possible. Due to space limitation, only a few of them are given here:

```
1  fun wordsize[w:Word] : Int { #inds[w.sequence] }
2  fun walksize[w:Word] : Int { #inds[w.visitedStates] }
3  fun lastVisitedState[w:Word] : State { last[w.visitedStates] }
4  fun eClosure[s:State] : IOAutomaton.states
5      { IOAutomaton.IO/getReachableStatesFrom[s, outL.SilentLabels] }
6  fun outputFrom[s:State] : Out
```

```
7       { s.(IOAutomaton.transition).State.outL }
8   fun followedBy[w1:Word, w2:Word] : Word
9       { { w:Word | w.wrtAutomaton = w1.wrtAutomaton &&
10                   w.wrtAutomaton = w2.wrtAutomaton &&
11                   w.sequence = w1.sequence.append[w2.sequence] } }
12  pred isPrefix[w:Word] { walksize[w] = wordsize[w] + 1 }
13  pred isSilent[l:Label] { l in SilentLabels }
14  pred isSymbol[w:Word, l:one Label]
15       { w.sequence.first = l and wordsize[w] = 1 }
```

For instance, the functions `eClosure`, `outputFrom` and `followedBy` return the ϵ-closure of a state, the set of output symbols of all the transitions from a given state of a Mealy Machine and the concatenation of two words, respectively. Based on these definitions the observer property is written as follows:

```
1   pred thetaIsAnObserver
2   {
3    all s:IO/Word, t:O/Word, tau:Out |
4          ((s.isPrefix and wordsize[s] <= #IOAutomaton.transition) &&
5           not tau.isSilent && t.isSymbol[tau] &&
6           (theta[s]).followedBy[t].isPrefix)
7        implies
8          (some q1:IOAutomaton.states | let q2 = lastVisitedState[s] |
9                q1 in eClosure[q2] and tau in outputFrom[q1])
10  }
```

The terms `s.isPrefix` (line 4) and `not tau.isSilent` (line 5) mean that $s \in L$ and $\tau \in T$, respectively. The term `t.isSymbol[tau]` (line 5) is similar to a cast operation because of the distinction between a symbol of T and a word of length one over T in the Alloy model. The formula at lines 8–9 corresponds to the conclusion of Property (4), but in terms of the automaton representation of u. It should be noted that the term `wordsize[s] <= #IOAutomaton.transition` (line 4) restricts the number of words that must be examined by Alloy because the language L may be infinite. The following lemmas and proposition show that checking the satisfaction of the observer property for words of bounded length is sufficient for the satisfaction of Property (4) under the following assumption.

Assumption 1. *Cycles are closed trails with the same vertex for entry and exit.*

Lemma 3.1. *Let G be the transition graph of the automaton that generates L while satisfying Assumption 1. Let s be a walk in which the trail s' appears more than once. If (4) does not hold for s then it does not hold for s' either.*

Proof. Only the basic argument is given. A cycle is split into two parts s' and s''. Suppose that (4) holds for s' but not for $s = s's''s'$. This is impossible, since $\theta(s's''s'u) = \theta(s's'')\theta(s'u) = \theta(s's'')\theta(s')\tau = \theta(s's''s')\tau$.

Lemma 3.2. *Let G be the transition graph of the automaton that generates L while satisfying Assumption 1. If (4) holds for all trails, then it holds for all walks.*

Proof. Let s be a walk for which (4) does not hold for a given $\tau \in T$. Lemma 3.1 ensures that there is a trail for which (4) does not hold either. By hypothesis, this is impossible.

Proposition 3.3. *Let G be the transition graph of the automaton that generates L while satisfying Assumption 1. If (4) holds for all walks s, with $|s| \leq |E(G)|$, and $\tau \in T$, then it holds for all walks (i.e., all words in L).*

The transition graph of Figure 1 verifies Assumption 1. Otherwise, it would be unwound in order to fulfill it. Furthermore, the causal map included in the specification of the injector satisfies the predicate `thetaIsAnObserver`. By Proposition 3.3, it can be concluded that it is an observer.

4 Implementation of Iterative Specification Refinement Processes

The synthesis of an optimal supervisor is used as an example to detail an iterative specification refinement process and its implementation with Kodkod [16]. The initial step of the process consists in running `S.IsSupervisor[G, unc, bad]`, where G is the automaton of the plant, *unc* the set of uncontrollable events and *bad* the control specification (i.e., the set of bad states). Furthermore, S refers to the instance generated by the SAT-solver after running the predicate. At the end of this step, nothing guarantees that S is optimal (i.e., maximally permissive), but it is nonblocking and prevents the system to reach a bad state. For that reason, the predicate

```
S.IsOptimalSupervisor[G, unc, bad]
```

introduced at the end of Section 3.1 must be checked. This is the body of the iterative process. If this predicate does not hold for S, then there exists a nonblocking supervisor s more permissive than S that forces the system to be only in good states. In fact, the instance that witnesses the consistency of the predicate `isSupervisor` is not the supremal element and s is a counterexample that violates `IsOptimalSupervisor`. In that case both the solution S and counterexample s are merged to obtain a new supervisor that is more permissive than s and S. The latter can be considered as a *local* optimal supervisor. Its set of states is `S.states + s.states` and its set of transitions is `S.transition + s.transition`. Only the union of sets of states (denoted Q) can be considered, if the local optimal supervisor is derived from the restriction of G to Q in order to handle only one set in the Kodkod program. This step is repeated until no counterexample is found by the SAT-solver, which means that the global optimal solution has been reached. In summary, this process produces a sequence of local optimal supervisors (w.r.t. narrow spaces of supervisors) until the global optimal supervisor is found. Therefore, at each iteration, a new space of supervisors is then constructed and it can be kept small enough to avoid combinatorial explosion by setting appropriately the number of atoms in the `for` clause of the command `check`.

Implementing this process with Kodkod entails several difficulties that must be treated with caution. After recovering the Kodkod program recorded by Alloy, the sequential program is reorganized w.r.t. the aforementioned iterative specification refinement process, including the main loop. To manipulate objects according to the name used in the Alloy specification, a map (`nameMap`) is defined to associate atoms with these names. Also, a method (`findCE`) that iterates on the relations of the solution returned by the method `solver.solve` is useful in order to get the one that corresponds to the counterexample. Finally, few statements are added to make the union of two sets to states:

```
Solution sol = solver.solve(x10, bounds);
TupleSet ts = findCE(sol);
Iterator<Tuple> itts = ts.iterator();
```

```
while (itts.hasNext())
      {
       Tuple tp = itts.next();
       x29 = x29.union(nameMap.get(tp.atom(0).toString()));
      }
```

5 Conclusion

The starting point of this work was to translate typical control problems encountered in SCT into corresponding propositional satisfiability problems. This work, which is a complement to [5], shows that Alloy is attractive for the verification of properties, but not sufficiently expressive for solving synthesis problems without coding iterative procedures in Kodkod. The main idea proposed in this paper is to use counterexamples and progressively alter the initial model in order to converge to an acceptable solution. This suggests a possible extension to Alloy, which could include a metalanguage to describe procedural processing based on instances generated by the Alloy analyzer in order to avoid atoms recovery w.r.t. the underlying control problem. This *hybrid* Alloy would combine symbolic and algorithmic computation. It would be also interesting that the metalanguage supports parameterized models. This feature would be useful in the modeling of distributed control as described in Section 3.3. Crocopat [2], which manipulates relations of any arity, could be a potential candidate for this purpose, but to the detriment of a greater encoding effort, compared to the aforementioned metalanguage, because it does not provide automatic solving facilities.

Nevertheless, the application of this approach is limited by the inherent characteristics of the SAT-based paradigm. To get round these limitations we must be aware of pitfalls and handle them carefully. Firstly, dealing with large systems involves combining smaller systems by using the synchronous product. Even if it is relatively easy to conceive an Alloy model for a product of two automata, the generalization to n automata is beyond reason. Indeed, it takes 328 772 ms to generate the clauses for a product of two automata, each of them having five states (the cat and mouse problem in [14]), and 70 299 ms to find an instance (i.e., the solution). Likewise, dealing with partial observation implies nondeterminism, and so the construction of a deterministic automaton that simulates a nondeterministic automaton. Such objects must be precalculated by another tool and integrated into the model before its analysis by Alloy for efficiency reasons. Secondly, since the number of atoms is limited in Alloy, the transition relation of an automaton defined initially as a quaternary relation should be expressed by using only relations of smaller arity. This could make it possible to cope with larger state spaces, even though we considered equivalence classes of states to reduce the number of transitions in order to solve real control problems. Thirdly, unwinding the transition relation of an automaton a finite number of times is a common technique in bounded model checking. It has been demonstrated that for all the properties that we considered, it suffices to unwind the transition relation only once to be sure that they hold for all the words of an infinite language when Alloy does not find a counterexample. Finally, reification must be avoided in order to achieve better execution time.

References

1. Barbeau, M., Custeau, G., St-Denis, R.: An algorithm for computing the mask value of the supremal normal sublanguage of a legal language. IEEE Trans. Automat. Contr. 40, 699–703 (1995)
2. Beyer, D., Noack, A., Lewerentz, C.: Efficient relational calculation for software analysis. IEEE Trans. Soft. Eng. 31, 137–149 (2005)
3. Chakib, H., Khoumsi, A.: Multi-decision supervisory control: parallel decentralized architectures cooperating for controlling discrete event systems. IEEE Trans. Automat. Contr. 56, 2608–2622 (2011)
4. Claessen, K., Een, N., Sheeran, M., Sórensson, N., Voronov, A., Åkesson, K.: SAT-solving in practice, with a tutorial example from supervisory control. J. Discrete Event Dynamic Systems: Theory and Appl. 19, 495–524 (2009)
5. Côté, D., Fraikin, B., Frappier, M., St-Denis, R.: A SAT-Based Approach for the Construction of Reusable Control System Components. In: Salaün, G., Schätz, B. (eds.) FMICS 2011. LNCS, vol. 6959, pp. 52–67. Springer, Heidelberg (2011)
6. Fernandez, J.-C.: An implementation of an efficient algorithm for bisimulation equivalence. Sci. Comput. Program. 13, 219–236 (1990)
7. Hill, R.C., Cury, J.E.R., de Queiroz, M.H., Tilbury, D.M., Lafortune, S.: Multi-level hierarchical interface-based supervisory control. Automatica 46, 1152–1164 (2010)
8. Jackson, D.: Software Abstractions. MIT Press, Cambridge (2006)
9. Kumar, R., Garg, V.K.: Modeling and Control of Logical Discrete Event Systems. Kluwer Academic Publishers, Boston (1995)
10. Kumar, R., Takai, S.: Inference-based ambiguity management in decentralized decision-making: decentralized control of discrete event systems. IEEE Trans. Automat. Contr. 52, 1783–1794 (2007)
11. Li, Y.: Control of vector discrete-event systems. Ph.D. Thesis, Graduate Dept. of Electrical and Computer Engineering, University of Toronto, Toronto, Canada (1991)
12. Ma, C., Wonham, W.M.: Nonblocking Supervisory Control of State Tree Structures. LNCIS, vol. 317. Springer, Heidelberg (2005)
13. Milner, R.: Communication and Concurrency. Prentice Hall, New York (1989)
14. Ramadge, P.J., Wonham, W.M.: The control of discrete event systems. Proc. of the IEEE 77, 81–98 (1989)
15. Su, R., van Schuppen, J.H., Rooda, J.E.: Aggregative synthesis of distributed supervisors based on automaton abstraction. IEEE Trans. Automat. Contr. 55, 1627–1640 (2010)
16. Torlak, E., Jackson, D.: Kodkod: A Relational Model Finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer, Heidelberg (2007)
17. Wong, K.C., Wonham, W.M.: Hierarchical control of discrete-event systems. J. Discrete Event Dynamic Systems: Theory and Appl. 6, 241–273 (1996)
18. Wong, K.C., Wonham, W.M.: On the computation of observers in discrete-event systems. J. Discrete Event Dynamic Systems: Theory and Appl. 14, 55–107 (2004)
19. Wonham, W.M., Ramadge, P.J.: On the supremal controllable sublanguage of a given language. SIAM J. Control and Optimization 25, 637–659 (1987)

Preventing Arithmetic Overflows in Alloy

Aleksandar Milicevic and Daniel Jackson

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
{aleks,dnj}@csail.mit.edu

Abstract. In a bounded analysis, arithmetic operators become partial, and a different semantics becomes necessary. One approach, mimicking programming languages, is for overflow to result in wrap-around. Although easy to implement, wrap-around produces unexpected counterexamples that do not correspond to cases that would arise in the unbounded setting. This paper describes a new approach, implemented in the latest version of the Alloy Analyzer, in which instances that would involve overflow are suppressed, and consequently, spurious counterexamples are eliminated. The key idea is to interpret quantifiers so that bound variables range only over values that do not cause overflow.

1 Introduction

A popular approach to the analysis of undecidable logics artificially bounds the universe, making a finite search possible. In model checking, the bounds may be imposed by setting parameters at analysis time, or even hardcoded into the system description. The Alloy Analyzer [1] is a model finder for the Alloy language that follows this approach, with the user providing a 'scope' for an analysis command that sets the number of elements for each basic type.

Such an analysis is not sound with respect to proof; just because a counterexample is not found (in a given scope) does not mean that no counterexample exists (in a larger scope). But it is generally sound with respect to counterexamples: if a counterexample is found, the putative theorem does not hold.

The soundness of Alloy's counterexamples is a consequence of the fact that the interpretation of a formula in a particular scope is always a valid interpretation for the unbounded model. There is no special semantics for interpreting formulas in the bounded case. This is possible because the relational operators are closed, in the sense that if two relations draw their elements from a given universe of atoms, then any relation formed from them (for example, by union, intersection, composition, and so on) can be expressed with the same universe.

Arithmetic operators, in contrast, are not closed. For example, the sum of two integers drawn from a given range may fall outside that range. So the arithmetic operators, when interpreted in a bounded context, appear to be partial and not total functions, and call for special treatment. One might therefore consider applying the standard strategies that have been developed for handling logics of partial functions.

J. Derrick et al. (Eds.): ABZ 2012, LNCS 7316, pp. 108–121, 2012.

A common strategy is to make the operators total functions by selecting appropriate values when the function is applied out of domain. In some logics (e.g. [9]) the value is left undetermined, but this approach is not easily implemented in a search-based model finder. Alternatively, the value can be determined. In the previous version of the Alloy Analyzer, arithmetic operators were totalized in this way by giving them wrap-around semantics, so that the smallest negative integer is regarded as the successor of the largest positive integer. This matches the semantics in some programming languages (e.g., Java), and is relatively easy to implement. Unfortunately, however, it results in counterexamples that would not arise in the unbounded context, so the soundness of counterexamples is violated. This approach leads to considerable confusion amongst users[2], and imposes the burden of having to filter out the spurious cases.

Another common strategy is to introduce a notion of undefinedness — at the value, term or formula level — and extend the semantics of the operators accordingly. However this is done, its consequence will be that formulas expressing standard properties will not hold. The associativity of addition, for example, will be violated, because the definedness of the entire expression may depend on the order of summation. In logics that take this approach, the user is expected to insert explicit guards that ensure that desired properties do not rely on undefined values. In our setting, however, where the partiality arises not from any feature of the system being described, but from an artifact of the analysis, demanding that such guards be written would be unreasonable, and would violate Alloy's principle of separating description from analysis bounds.

This paper provides a different solution to the dilemma. Roughly speaking, counterexamples that would result in arithmetic overflow are excluded from the analysis, so that any counterexample that is presented to the user is guaranteed not to be spurious. This is achieved by redefining the semantics of quantifiers in the bounded setting so that the models of a formula are always models of the formula in the unbounded setting. This solution has been implemented in Alloy4.2 and can be activated via the "Forbid Overflows" option.

The rest of the paper is organized as follows. Section 2 illustrates some of the anomalies that arise from treating overflow as wraparound. Section 3 shows the problem in a more realistic context, by presenting an Alloy model of a minimum spanning tree algorithm that combines arithmetic and relational operators, and shows how a valid theorem can produce spurious counterexamples. Section 4 gives our new semantics, and Section 5 explains its implementation in boolean circuits. Finally, Section 7 presents related work on the topic of partial functions in logic, compares our approach with the existing ones, and discusses alternatives for solving the issue of overflows in Alloy.

2 Prototypical Overflow Anomalies

While a wraparound semantics for integer overflow is consistent and easily explained, its lack of correspondence to unbounded arithmetic produces a variety of anomalies. Most obviously, the expected properties of arithmetic do not necessarily hold: for example, that the sum of two positive integers is positive (Fig. 1.a).

```
check {
  all a, b: Int |
    a > 0 && b > 0 => a.plus[b] > 0
} for 3 Int
```

counterexample

```
Int = {-4, -3, ..., 2, 3}
a = 3; b = 1;
a.plus[b] = - 4
```

(a) Sum of two positive integers is not necessarily positive.

```
check {
  all s: set univ |
    some s iff #s > 0
} for 4 but 3 Int
```

counterexamples

```
Int = {-4, -3, ..., 2, 3}
s = {S0, S1, S2, S3}
#s = -4
```

(b) Overflow anomaly involving cardinality of sets.

Fig. 1. Prototypical overflow anomalies in the previous version of Alloy

More surprisingly, expected properties of the cardinality operator may not hold. For example, the Alloy formula `some s` is defined to be true when the set `s` contains some elements. One would expect this to be equivalent to stating that the set has a non-zero cardinality (Fig. 1.b). And yet this property will not hold if the cardinality expression `#s` overflows, since it may wrap around, so that a set with enough elements is assigned a negative cardinality.

Of course, in practice, Alloy is more often used for analyzing software designs than for exploring mathematical theorems, and so properties of this kind are rarely stated explicitly. But such properties are often relied upon implicitly, and consequently, when they fail to hold, the spurious counterexamples that are produced are even harder to comprehend. Such a case arises in the the example discussed in the next section, where a test for an undirected graph being treelike is expressed by saying that there should be one fewer edge than nodes. Clearly, when using such a formulation, the user would rather not consider the effects of wraparound in counting nodes or edges.

3 Motivating Example

Consider checking Prim's algorithm [7, §23.2], a greedy algorithm that finds a minimum spanning tree (MST) for a connected graph with positive integral weights. Alloy is for the most part well-suited to this task, since it makes good use of Alloy's quantifiers and relational operators, including transitive closure. The need to sum integer weights, however, is potentially problematic, due to Alloy's bounded treatment of integers.[1]

[1] An alternative approach would be to use an analysis that includes arithmetic without imposing bounds. It is not clear, however, whether such an approach could be fully automated, since the logics that are sufficiently expressive to include both arithmetic and relational operators do not have decision procedures, and those (such as SMT) that do offer decision procedures for arithmetic are not expressive enough. In this paper, we are not arguing that such an approach cannot work. But, either way, exploring ways to mitigate the effects of bounding arithmetic has immediate benefit for users of Alloy, and may prove useful for other tools that impose ad hoc bounds.

```
open util/ordering[Time]

sig Time {}

sig Node {covered: set Time}

sig Edge {
  weight: Int,
  nodes: set Node,
  chosen: set Time
} {
  weight >= 0 and #nodes = 2
}

pred cutting (e: Edge, t: Time) {
  (some e.nodes & covered.t) and (some e.nodes & (Node - covered.t))
}

pred step (t, t': Time) {
  -- stutter if done, else choose a minimal edge from a covered to an uncovered node
  covered.t = Node =>
         chosen.t' = chosen.t and covered.t' = covered.t
  else some e: Edge {
         cutting[e,t] and (no e2: Edge | cutting[e2,t] and e2.weight < e.weight)
         chosen.t' = chosen.t + e
         covered.t' = covered.t + e.nodes}
}

fact prim {
  -- initially just one node marked
  one covered.first and no chosen.first
  -- steps according to algorithm
  all t: Time - last | step[t, t.next]
  -- run is complete
  covered.last = Node
}

pred spanningTree (edges: set Edge) {
  -- empty if only 1 node and 0 edges, otherwise covers set of nodes
  (one Node and no Edge) => no edges else edges.nodes = Node
  -- connected and a tree
  #edges = (#Node).minus[1]
  let adj = {a, b: Node | some e: edges | a + b in e.nodes} |
    Node -> Node in *adj
}

correct: check { spanningTree [chosen.last] } for 5 but 10 Edge, 5 Int

smallest: check {
  no edges: set Edge {
    spanningTree[edges]
    (sum e: edges | e.weight) < (sum e: chosen.last | e.weight)}
} for 5 but 10 Edge, 5 Int
```

Fig. 2. Alloy model for bounded verification of Prim's algorithm that finds a minimum spanning tree for a weighted connected graph

Figure 2 shows an Alloy representation of the problem. The sets (signatures in Alloy) `Node` and `Edge` (lines 5–5 and 7–13) represent the nodes and edges of a graph. Each edge has a weight (line 8) and connects a set of nodes (line 9); weights are non-negative and edges connect exactly two nodes (line 12).

This model uses the *event-based idiom* [10, §6.2.4] to model sequential execution. The `Time` signature (line 3) is introduced to model discrete time instants, and fields `covered` (line 5) and `chosen` (line 10) track which nodes and edges have been covered and selected respectively at each time. Initially (line 31) an arbitrary node is covered and no edges have been chosen. In each subsequent time step (line 33), the state changes according to the algorithm. The algorithm terminates (line 35) when the set of all nodes has been covered.

At each step, a 'cutting edge' (that is, one that connects a covered and a non-covered node) is selected such that there is no other cutting edge with a smaller weight (line 24). The edge is marked as chosen (line 25), and its nodes as covered (line 26)[2]. If the node set has already been covered (line 21), instead no change is made (line 22), and the algorithm stutters.[3]

Correctness entails two properties, namely that: (1) at the end, the set of covered edges forms a spanning tree (line 47), and (2) there is no other spanning tree with lower total weight (lines 49–53). The auxiliary predicate (`spanningTree`, lines 38–45) defines whether a given set of edges forms a spanning tree, and states that, unless the graph has no edges and only one node, the edges cover all nodes of the graph (line 40), the number of given edges is one less than the number of nodes (line 42), and that all nodes are connected by the given set of edges (lines 43–44).

If we run the previous version of the Alloy Analyzer to check these two properties, the `smallest` check fails. In each of the reported counterexamples, the expression `sum e: edges | e.weight` (representing the sum of weights in the alternative tree, line 52) overflows and wraps around, and thus appears (incorrectly) to have a lower total weight than the tree constructed.[4] In the latest version of the Alloy Analyzer that incorporates the approach described in this paper, the check, as expected, yields no counterexamples for a scope of up to 5 nodes, up to 10 edges and integers ranging from -16 to 15.

4 Approach

Our goal is to give a semantics to formulas whose arithmetic expressions might involve out-of-domain applications, such as the addition of two integers that ideally would require a value that cannot be represented. In contrast to traditional

[2] For a field `f` modeling a time-dependent state component, the expression `f.t` represents the value of `f` at time `t`.

[3] An implementation would, of course, terminate rather than stuttering. Ensuring that traces can be extended to a fixed length allows better symmetry breaking to be employed, dramatically improving performance.

[4] One might think that this overflow could be avoided by adding guards, for example that the total computed weight in the alternative tree is not negative. This does not work, since the sum can wrap around all the way back into positive territory.

approaches to the treatment of partial functions, the out-of-domain applications arise here not from any intrinsic property of the system being modeled, but rather from a limitation of the analysis.[5] Consequently, whereas it would be appropriate in more traditional settings to produce a counterexample when an out-of-bounds application occurs, in this setting, we aim to mask such counterexamples, since they do not indicate problems with the model per se.

First, a standard three-valued logic [13] is adopted, in which elementary formulas involving out-of-bounds arithmetic applications are given the third logical value of 'undefined' ($\perp$), and undefinedness is propagated through the logical connectives in the expected way (so that, for example, 'false and undefined' evaluates to false). But the semantics of quantifiers diverges from the standard treatment: the meaning of a quantified formula is adjusted so that the bound variable ranges only over values that would yield a body that evaluates to true or false. Thus bindings that would result in an undefined quantification are masked, and quantified formulas are never undefined. Since every top level formula in an Alloy model is quantified[6] this means that counterexamples (and, in the case of simulation, instances) never involve undefined terms.

This semantics cannot be implemented directly, since the analysis does not explicitly enumerate values of bound variables, but instead uses a translation to boolean satisfiability (SAT) [21]. A scheme is therefore needed in which the formula is translated compositionally to a SAT formula. To achieve this, a boolean formula is created to represent whether or not an arithmetic expression is undefined. This is then propagated to elementary subformulas in an unconventional way which ensures the high-level semantics of quantifiers given above.

To understand this intuitively, it may help to think of all the quantifiers being eliminated by explicit unrolling, and the entire formula being put in disjunctive normal form, as a collection of clauses, each consisting of a conjunction of elementary subformulas. The goal is to ensure that when an arithmetic term is undefined, the clause containing it evaluates to false and is effectively dropped.

We therefore have given two semantics: the high level semantics that the user needs to understand, and the low level semantics that justifies the analysis. This lower level semantics is then implemented by a translation to boolean circuits.

4.1 User-Level Semantics

As explained above, the key idea of our approach is to change the semantics of quantifiers so that the quantification domain is restricted to those values for which the body of the quantifier is defined (determined by the **def** function):

$$[\![\texttt{all } x\texttt{: Int } |\ p(x)]\!] = \forall x \in \texttt{Int} \bullet \textbf{def}[\![p(x)]\!] \implies p(x)$$
$$[\![\texttt{some } x\texttt{: Int } |\ p(x)]\!] = \exists x \in \texttt{Int} \bullet \textbf{def}[\![p(x)]\!] \wedge p(x)$$

[5] Note that this discussions concern only the partial function applications arising from arithmetic operators; partial functions over uninterpreted types are treated differently in Alloy, and counterexamples involving their application are never masked.

[6] The fields and signatures of an Alloy model are always implicitly bound in an outermost existential quantifier, which is eliminated in analysis by skolemization.

Integer expressions (i.e. those employing Alloy's arithmetic operators) are undefined if any argument is undefined or the evaluation results in overflow:

$$\textbf{def}[\![\alpha(i_1, \ldots, i_n)]\!] = (i_1 \neq \bot) \wedge \cdots \wedge (i_n \neq \bot) \wedge \neg([\![\alpha(i_1, \ldots, i_n)]\!] \text{ overflows})$$

Integer predicates are boolean formulas that relate one or more integer expressions. In Alloy, the only integer predicates are the integer comparison operators. They are also undefined if any argument is undefined:

$$\textbf{def}[\![\rho(i_1, \ldots, i_n)]\!] = (i_1 \neq \bot) \wedge \cdots \wedge (i_n \neq \bot)$$

A formula is defined if it evaluates to either true or false when three-valued logic truth tables of propositional operators are used (e.g. [13, Table A.1]):

$$\textbf{def}[\![\texttt{and}(p, q)]\!] = (p \wedge_3 q) \neq \bot \qquad \textbf{def}[\![\texttt{implies}(p, q)]\!] = (p \Rightarrow_3 q) \neq \bot$$
$$\textbf{def}[\![\texttt{or}(p, q)]\!] = (p \vee_3 q) \neq \bot \qquad \textbf{def}[\![\texttt{not}(p)]\!] = (\neg_3 p) \neq \bot$$

Finally, quantifiers are always defined:

$$\textbf{def}[\![\texttt{all } x \mid p(x)]\!] = \texttt{true} \qquad \textbf{def}[\![\texttt{some } x \mid p(x)]\!] = \texttt{true}$$

Note that the semantics of the rest of the Alloy logic (in particular, of the relational operators) remains unchanged.

4.2 Implementation-Level Semantics

A direct implementation of the user-level semantics in Alloy would entail a three-valued logic, and the translation to SAT would thus require 2 bits for a single boolean variable (to represent the 3 possible values), a substantial change to the existing Alloy engine. Furthermore, such a change would likely adversely affect the analysis performance of models that do not use integer arithmetic. In this section, we show how the same semantics can be achieved using the existing Alloy engine, merely by adjusting the translation of elementary integer functions and integer predicates.

To make all formulas denote (and thus to avoid the need for a third boolean value), a truth value must be assigned to an integer predicate even when some of its arguments are undefined. A common approach [8,17] is to assign the value `false`. For example, the sentence e_1`<`e_2 will be true *iff* both e_1 and e_2 are defined and e_1 is less than e_2 (and similarly for e_1`>=`e_2):

$$[\![\ \texttt{lt}(e_1,e_2)\ ]\!] = e_1 < e_2 \ \wedge\ e_1{\downarrow} \ \wedge\ e_2{\downarrow} \qquad [\![\ \texttt{gte}(e_1,e_2)\ ]\!] = e_1 \geq e_2 \ \wedge\ e_1{\downarrow} \ \wedge\ e_2{\downarrow}$$

(using the syntactic shortcuts $e{\downarrow} \equiv e \neq \bot$, and $e{\uparrow} \equiv e = \bot$).

Negation presents a challenge. Following the high-level semantics, negation of an integer predicate (e.g., `!(`e_1`<`e_2`)`) is still undefined if any argument is undefined. Therefore, under the low-level semantics, `!(`e_1`<`e_2`)` must also, despite the negation, evaluate to false if either e_1 or e_2 is undefined (and thus have exactly the same semantics as $e_1 \geq e_2$). To achieve this behavior, the *polarity* [11] of each expression must be known. Polarity is easily determined by the structure of the enclosing negations. Evaluation of a binary integer predicate can be then formulated as (ignoring the stack of enclosing quantifiers for the moment):

(a) Semantic Domains

```
Formula      = BoolConst | IntPred(IntExpr, ..., IntExpr) |
               BoolPred(Formula, ..., Formula) |
               QuantFormula(VarDecl, Formula)
IntExpr      = IntConst | IntVar | IntFunc(IntExpr, ..., IntExpr)
BoolConst    = true | false
IntConst     = ⊥ | 0 | -1 | 1 | -2 | 2 | ...
QuantFormula = all | some
BoolPred     = not1 | and2 | or2 | implies2 | iff2
IntPred      = eq2 | neq2 | gt2 | gte2 | lt2 | lte2
IntFunc      = neg1 | plus2 | minus2 | times2 | div2 | mod2 |
               shl2 | shr2 | sha2 | bitand2 | bitor2 | bitxor2
Store        = {var: IntVar; val: IntConst; quant: QuantFormula;
                polarity: BoolConst; parent: Store}
```

(b) Symbols

$\bot$ ∈ `IntConst`	(undefined integer)	b_i ∈ `BoolConst`	(boolean constants)
i_i ∈ `IntConst`	(integer constants)	p_i ∈ `Formula`	(boolean formulas)
e_i ∈ `IntExpr`	(integer expressions)	β_i ∈ `BoolPred`	(boolean predicates)
ρ_i ∈ `IntPred`	(integer predicates)	x_i ∈ `IntVar`	(integer variables)
α_i ∈ `IntFunc`	(arithmetic functions)	q_i ∈ `QuantFormula`	(quantified formula)

(c) Stores

σ : `Store` (environment of nested quantifiers and variable bindings)

Fig. 3. Overview of semantic domains, symbols, and stores to be used. Subscripts in function and predicate names indicate their arities.

aeval : `IntExpr` → `Store` → `IntConst`

$$
\begin{aligned}
&\textbf{aeval}[\![i]\!]\sigma &&= i\\
&\textbf{aeval}[\![x]\!]\{x_\sigma, i_\sigma, q, b, \sigma_\mathsf{p}\} &&= \textbf{if } x_\sigma = x \textbf{ then } i_\sigma \textbf{ else } \textbf{aeval}[\![x]\!]\sigma_\mathsf{p}\\
&\textbf{aeval}[\![\alpha(i_1, \ldots, i_n)]\!]\sigma &&= \begin{cases} \bot & \textbf{if } i_i = \bot \text{ or ... or } i_n = \bot \\ \bot & \textbf{if } \alpha(i_1, \ldots, i_n) \text{ overflows} \\ \alpha(i_1, \ldots, i_n) & \text{otherwise} \end{cases}\\
&\textbf{aeval}[\![\alpha(e_1, \ldots, e_n)]\!]\sigma &&= \textbf{aeval}[\![\alpha(\textbf{aeval}[\![e_1]\!]\sigma, \ldots, \textbf{aeval}[\![e_n]\!]\sigma)]\!]\sigma
\end{aligned}
$$

Fig. 4. Evaluation of arithmetic operations (**aeval**). If any operand of an arithmetic operation is undefined, the result is undefined too.

beval : `Formula` → `Store` → `BoolConst`

$$
\begin{aligned}
&\textbf{beval}[\![b]\!]\sigma &&= b\\
&\textbf{beval}[\![\rho(e_1, e_2)]\!]\sigma &&= \textbf{ieval}[\![\rho(e_1, e_2)]\!]\sigma\\
&\textbf{beval}[\![\mathtt{not}(p)]\!]\{x, i, q, b, \sigma_\mathsf{p}\} &&= \neg\, \textbf{beval}[\![p]\!]\{x, i, q, \neg b, \sigma_\mathsf{p}\}\\
&\textbf{beval}[\![\beta(p_1, \ldots, p_2)]\!]\sigma &&= \beta(\textbf{beval}[\![p_1]\!]\sigma, \ldots, \textbf{beval}[\![p_2]\!]\sigma)\\
&\textbf{beval}[\![\mathtt{all}\ x\mathtt{:\ Int\ |}\ p]\!]\sigma &&= \bigwedge_{i \in \mathrm{Int}} \textbf{beval}[\![p]\!]\{x, i, \mathtt{all}, \mathtt{true}, \sigma\}\\
&\textbf{beval}[\![\mathtt{some}\ x\mathtt{:\ Int\ |}\ p]\!]\sigma &&= \bigvee_{i \in \mathrm{Int}} \textbf{beval}[\![p]\!]\{x, i, \mathtt{some}, \mathtt{true}, \sigma\}
\end{aligned}
$$

Fig. 5. Evaluation of boolean formulas. The new semantics (together with the **ieval** function, Fig. 6) ensures that quantifiers quantify over only those values that do not cause any overflows.

$$
\begin{array}{l}
\textbf{ieval} : \texttt{IntPred} \to \texttt{Store} \to \texttt{BoolConst} \\
\hline
\quad \textbf{ieval}[\![\rho(e_1,\, e_2)]\!]\sigma \quad = \textbf{let}\ b\ = \rho(\textbf{aeval}[\![e_1]\!]\sigma, \textbf{aeval}[\![e_2]\!]\sigma)\textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{ensureDef}[\![b, \{e_1, e_2\}]\!]\sigma \\
\\
\textbf{ensureDef} : \texttt{BoolConst} \to \{\texttt{IntExpr}\} \to \texttt{Store} \to \texttt{BoolConst} \\
\hline
\quad \textbf{ensureDef}[\![b,\, e_{\texttt{in}}]\!]\{x,\, i,\, q,\, b_{\texttt{pol}},\, \sigma_{\texttt{p}}\} = \textbf{let}\ e_{\text{univ}} = \{\, e \mid e \in e_{\texttt{in}} \wedge \textbf{isUnivQuant}[\![e]\!]\sigma \,\}\ \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{let}\ e_{\text{ext}}\ = e_{\texttt{in}} \setminus e_{\text{univ}}\ \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{let}\ b_{\texttt{def}}\ = (e_{\text{ext}} = \varnothing) \vee \bigwedge_{e \in e_{\text{ext}}} (\textbf{aeval}[\![e]\!]\sigma \neq \bot)\ \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{let}\ b_{\texttt{undef}} = (e_{\text{univ}} \neq \varnothing) \wedge \bigvee_{e \in e_{\text{univ}}} (\textbf{aeval}[\![e]\!]\sigma = \bot)\ \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{if}\ b_{\texttt{pol}}\ \ \textbf{then}\ (b \vee b_{\texttt{undef}}) \wedge b_{\texttt{def}} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ (b \vee \neg b_{\texttt{def}}) \wedge \neg b_{\texttt{undef}} \\
\textbf{isUnivQuant} : \texttt{IntExpr} \to \texttt{Store} \to \texttt{BoolConst} \\
\hline
\quad \textbf{isUnivQuant}[\![e]\!]\{\} \quad = \texttt{false} \\
\quad \textbf{isUnivQuant}[\![e]\!]\{x,\, i,\, q,\, b,\, \sigma_{\texttt{p}}\} \quad = \textbf{if}\ x \in \textbf{vars}[\![e]\!]\ \ \textbf{then}\ q\ = \texttt{all} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else isUnivQuant}[\![e]\!]\sigma_{\texttt{p}} \\
\textbf{vars} : \texttt{IntExpr} \to \{\texttt{IntVar}\} \\
\hline
\quad \textbf{vars}[\![i]\!] \quad = \varnothing \\
\quad \textbf{vars}[\![x]\!] \quad = \{x\} \\
\quad \textbf{vars}[\![\alpha(e_1,\, \ldots,\, e_n)]\!] \quad = \textbf{vars}[\![e_1]\!] \cup \ldots \cup \textbf{vars}[\![e_n]\!]
\end{array}
$$

Fig. 6. Evaluation of integer predicates. If any argument of an integer predicate is undefined, the result is `true` if the expression is in a universally quantified context, otherwise it is `false`.

$$
\begin{array}{rl}
[\![\rho(e_1, e_2)]\!] = \textbf{if}\ \text{polarity is positive} & \textbf{then}\ \rho([\![e_1]\!], [\![e_2]\!]) \wedge [\![e_1]\!]{\downarrow} \wedge [\![e_2]\!]{\downarrow} \\
& \textbf{else}\ \rho([\![e_1]\!], [\![e_2]\!]) \vee \neg([\![e_1]\!]{\downarrow} \wedge [\![e_2]\!]{\downarrow})
\end{array}
$$

The polarity approach is not *compositional*, since the meaning of the negation of a formula is not simply the logical negation of the meaning of that formula. For that reason, this approach violates the law of the excluded middle, which, fortunately, will not be problematic, since the violation would only be observable for variable bindings that result in overflow and such bindings are excluded by the semantics (see Sec. 4.4).

The semantics are formally defined in Figs. 3–6. Expressions and formulas are interpreted in the context of a store that holds, for each variable bound in an enclosing quantifier: (a) the value of the variable in the particular binding, (b) whether the quantifier is universal or existential, and (c) its current polarity.

Evaluation of integer expressions (**aeval**) and boolean formulas (**beval**) has the same effect as evaluation in the user-level semantics; it is elaborated differently here simply to account for the need to pass the store. Every time a negation is seen, the inner formula is interpreted in a store in which the polarity is negated. Quantifiers are unfolded, with the body interpreted in a new nested store with polarity set to `true`. For the evaluation of top-level formulas, an empty existential environment is presented.

Evaluation of integer predicates (**ieval**) is where the crucial differences lie. Whereas in the user-level semantics predicates evaluate to true, false and undefined, in this implementation semantics predicates evaluate only to true or false.

When a predicate would have been undefined in the user-level semantics, its meaning will be either true or false, chosen in such a way as to ensure that the associated binding becomes irrelevant. This choice is represented by the auxiliary function **ensureDef**, which determines the truth value based on the current polarity and the stack of enclosing quantifiers.

For the existential case, the goal is to ensure that a predicate evaluates to false when any argument is undefined. However, when such a predicate contains a universally quantified variable, to achieve the desired semantics of the universal quantifier (which is to ignore cases where the body is undefined), it is enough to simply make the predicate evaluate to true instead. Therefore, all expressions with universally quantified variables are identified first (e_{univ}) and a definedness condition for them ($b_{\texttt{undef}}$) is computed as a disjunction of either being undefined. For all other arguments (e_{ext}) the definedness condition ($b_{\texttt{def}}$) is a conjunction of all being defined (as before). Finally, based on the value of the polarity flag ($b_{\texttt{pol}}$), the two conditions are attached to the base result (b).

4.3 Correspondence between the Two Semantics

To show that our low-level semantics correctly implements the high-level user semantics, it is enough to establish a correspondence between the two definitions of quantifiers (the low-level semantics only introduced a change to the semantics of quantifiers). Following directly from the two definitions, this is equivalent to proving that whenever an expression $p(x)$ is undefined by the laws of three-valued logic (i.e., $\textbf{def}[\![p(x)]\!]$ is false), if x is *universally* quantified then $\textbf{beval}[\![p(x)]\!]$ evaluates to true, else it evaluates to false.

This hypothesis would traditionally be proved by a structural induction on expressions. Instead of giving a complete proof (which would exceed the scope of this paper), we explain several interesting base cases instead.

As said earlier, the low-level evaluation of integer predicates is where the crucial differences lie. Let us therefore consider the case when $p(x)$ is an integer predicate, $\rho(e_1(x),\ e_2(x))$. Furthermore, let us assume that $e_1(x)$ is undefined, which makes $p(x)$ undefined as well. In this context, polarity is positive, and the value of $\textbf{beval}[\![\rho(e_1(x),\ e_2(x))]\!]$ becomes the value of **ensureDef**. There are two cases to consider: (1) if x is *universally* quantified, e_{univ} contains both e_1 and e_2, $b_{\texttt{undef}}$ becomes true, $b_{\texttt{def}}$ is true by default, so the result is also true regardless of the base value b; (2) if x is *existentially* quantified, e_{ext} contains both e_1 and e_2, $b_{\texttt{def}}$ becomes false, $b_{\texttt{undef}}$ is false by default, so the result is also false, as expected.

Let us now assume that $p(x)$ is a negation of an integer predicate, $p(x) = \neg\rho(e_1(x), e_2(x))$, and that $e_1(x)$ is again undefined. Despite the negation, $p(x)$ is *still* undefined, so the low-level evaluation should behave exactly as in the previous case. The result of $\textbf{beval}[\![p(x)]\!]$ now becomes a negation of the value returned by **ensureDef**, which, in contrast, now evaluates in a context where the polarity is negative. Following exactly the same derivation as before, it can be shown that **ensureDef** now returns false for the universal case, and true for the existential case (because of the negative polarity), so the end result of $\textbf{beval}[\![p(x)]\!]$ remains the same, as expected.

4.4 The Law of the Excluded Middle

We mentioned earlier that our non-compositional rule for negation breaks the law of the excluded middle. Usually, this is not a problem.

Consider checking the theorem that all integers when multiplied by two are either less than zero or not less than zero:

```
check { all x: Int | x.mul[2] < 0 or !(x.mul[2] < 0) } for 3 Int
```

If we run the Alloy Analyzer with overflow prevention turned on, this sentence is interpreted as "*for all integers x s.t. x times two does not overflow, x times two is either less than zero or not less than zero*", and thus no counterexample is found, which is consistent with classical logic.

In a sense, however, the violation of the law is visible if truth is associated with whether or not a check yields a counterexample at all. For example, a check of whether 4 plus 5 is *equal* to 6 plus 3 for the bitwidth of 4 (`Int = {-8, ..., 7}`) does not return a counterexample, but neither does a check of whether 4 plus 5 is *different* from 6 plus 3.

```
check { 4.plus[5]  = 6.plus[3] } for 4 Int -- no counterexample found
check { 4.plus[5] != 6.plus[3] } for 4 Int -- no counterexample found
```

Though this might at first appear confusing, it is consistent with our design goal: indeed, for a bitwidth of 4, there is no non-overflowing instance in which 4 plus 5 is either equal to or different from 6 plus 3.

5 Implementation in Circuits

The core task of finding satisfying models for a relational formula is delegated to Kodkod [20]. Kodkod is a bounded constraint solver for relational first-order logic. It works by translating a given relational formula (together with bounds) into an equivalent propositional formula and using an of-the-shelf SAT solver to check its satisfiability.

Detecting arithmetic overflows at the level of relational logic would be difficult, and probably inefficient. We therefore implemented our approach at the level of the translation to propositional logic, as an extension to Kodkod. Even though the goal is now to translate the input formula into a digital circuit (instead of evaluating it to a boolean constant), we only had to modify Kodkod's translation of appropriate terms directly following the denotational semantics presented in this paper. In summary, we changed:

- *the translation of arithmetic operations* to generate an additional one-bit overflow circuit which is set iff the operation overflows. We used textbook overflow circuits for all arithmetic operations supported by Kodkod;
- *the way the environment gets updated* so that it additionally keeps track of the polarity and the quantification stack;
- *the translation of integer comparison predicates* so that the original circuit representing the comparison result is extended to include the definedness conditions, exactly as defined above.

6 Evaluation

Finding suitable models for evaluating the new approach is difficult, because most Alloy models do not involve arithmetic, in part because of the problem of overflow that motivated this work.

To evaluate the approach of this paper, we took a previously published model of a flash filesystem [15] which uses arithmetic operations and whose analysis is non-trivial, and compared its execution under the old (Alloy4) and new (Alloy4.2) analysis schemes. This model involves both assertions (that certain properties hold) and simulations (that produce sample scenarios). First, we checked that there are no new spurious counterexamples, and that none of the expected valid scenarios are lost. This was not the focus of our evaluation, however, since the design of the analysis ensures it. Rather, our concern was that the addition of new clauses to the SAT formula generated by the Analyzer might increase translation and solving time.

The new translation always results in a larger SAT formula, because extra clauses are needed to rule out models that overflow. One might imagine that adding clauses would cause the solving time to increase. On the other hand, the additional clauses might result in a smaller search space, and thus potentially reduce the search time.

We ran all checks that were present in the "concrete" module of the model. The first 10 (`run1` through `run10`) are simulations (which all find an instance), and the remaining 6 (`check1` through `check6`) are checks, which, with the exception of `check5`, produce no counterexamples. For each check, we measured both the translation and solving time, as shown in Table 1. As expected, in some cases the analysis runs faster, and sometimes it takes longer. In total, with the overflow prevention turned on, the entire analysis finished in about 8 hours, as opposed to almost 12 hours that the same analysis took otherwise.

Table 1. Analysis times of all checks found in the "concrete" module of a flash filesystem from [15]. All values are in seconds, except the values in the "speedup" row which are in percents. "old" stands for the previous version of Alloy, whereas "new" stands for the new version with overflow prevention turned on.

	run1		run2		run3		run4		run5		run6		run7		run8		run9	
old	1.2	0.9	2.1	0.4	0.8	0.2	12.9	2.3	5.9	0.5	12.7	1.0	11.9	1.1	9.0	1.0	12.5	1.0
new	1.2	0.8	1.6	0.4	0.8	0.3	13.4	8.7	6.2	0.5	12.6	0.8	12.1	1.5	9.1	1.0	12.7	2.6
abs diff	0	0.1	0.5	0	0	-0.1	-0.5	-6.4	-0.3	0	0.1	0.2	-0.2	-0.4	-0.1	0	-0.2	-1.6
speedup	0	11.1	23.8	0	0	-50.0	-3.9	-278.3	-5.1	0	0.8	20.0	-1.7	-36.4	-1.1	0	-1.6	-160.0

	run10		check1		check2		check3		check4		check5		check6		**total**
old	25.7	14.8	20.0	39.6	12.1	2190.7	12.0	30673.3	12.5	3713.2	12.3	3.0	74.3	5782.6	42663.5
new	25.9	12.5	20.2	12.6	12.2	1670.4	12.2	16741.9	12.7	3526.9	12.5	1.3	73.9	7083.5	29304.5
abs diff	-0.2	2.3	-0.2	27	-0.1	520.3	-0.2	13931.4	-0.2	186.3	-0.2	1.7	0.4	-1300.9	13359.0
speedup	-0.8	15.5	-1.0	68.2	-0.8	23.8	-1.7	45.4	-1.6	5.0	-1.6	56.7	0.5	-22.5	31.3

7 Related Work

The problem addressed in this paper is an instance of the more general problem of handling partial functions in logic. The most important difference, however, is that, in our case, the out-of-bound function applications arise due to deficiencies in the analysis, rather than from the inherent semantics of the logic. Requiring the user to introduce guards in the formal description itself to mitigate the effects of undefinedness is therefore not acceptable.

Despite this fundamental difference, our approach shares some features of several previously explored approaches.

The *Logic of Partial Functions* (LPF) was proposed for reasoning about the development of programs [13,14], and was adopted in VDM [12]. In this approach, not only integer predicates but also boolean formulas may be non-denoting, so truth tables extended to a three-valued logic are needed. This allows guards for definedness to be treated intuitively; thus, for example, even when "`x`" is equal to zero, formula `x!=0 => x/x=1`, evaluates to `true` in spite of `x/x=1` being undefined. Our approach uses this three-valued logic for determining whether the body of a quantified formula is undefined, but the meaning of the formula as a whole is treated differently – masking the binding that produces undefinedness rather than interpreting the quantification in the same three-valued logic.

Our implementation-level semantics adopts the *traditional approach to partial functions* (a term coined by Farmer [8]), in which all formulas must be denoting but functions may be partial. Farmer's approach, however, leaves open whether, given an undefined `a`, `!(a=a)` and `a!=a` have different meanings — an issue that in the standard setting is hard to resolve because of the competing concerns of compositionality and preserving complementarity of predicates. In our case, the non-compositional choice fits nicely with the user-level semantics.

Like the Alloy Analyzer, SMT [6] solvers can also be used for model finding. They all support unbounded integer arithmetic, so the problem of overflows does not arise. However, using Alloy over SMT-based tools has certain benefits, most notably the expressiveness of the Alloy relational language. There are higher-level languages that build on SMT technologies (e.g. Dafny [16]), but for a task similar to verifying Prim's algorithm, such tools are typically not fully automatic, and demand that the user provide intermediate lemmas.

Model-based languages such as B [3] and Z [18], being designed for specifying programs, make extensive use of partial functions. Both are based on set theory, and model functions as relations. Whereas in Alloy out-of-bounds applications of partial functions over uninterpreted types result in the empty set, in B such an application results in an unknown (but determined) value [19]. The initial specification of the Z notation [18] left the handling of partial functions open.

Several different approaches have been proposed (see [5] for a survey); in the end, it appears that the same approach as in B has evolved to be the norm [19]. In both Z and B, integers are unbounded, and so the problems of integer overflow do not arise. On the other side, the tools for discharging proof obligations (e.g. Rodin [4]) are typically less automated than the Alloy Analyzer.

References

1. Alloy: A language and tool for relational models, http://alloy.mit.edu/alloy
2. User posts about arithmetic overflows on Alloy community forum, http://alloy.mit.edu/community/search/node/overflow
3. Abrial, J.R., Hoare, A.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (2005)
4. Abrial, J.-R., Butler, M.J., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. STTT 12(6) (2010)
5. Arthan, R., Road, L.: Undefinedness in Z: Issues for Specification and Proof. In: CADE-13 Workshop on Mechanization of Partial Functions. Springer (1996)
6. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa (2010)
7. Cormen, T.H., Stein, C., Rivest, R.L., Leiserson, C.E.: Introduction to Algorithms, 2nd edn. McGraw-Hill Higher Education (2001)
8. Farmer, W.M.: Reasoning about partial functions with the aid of a computer. Erkenntnis 43 (1995)
9. Gries, D., Schneider, F.: A logical approach to discrete math. Texts and monographs in computer science. Springer (1993)
10. Jackson, D.: Software Abstractions: Logic, language, and analysis. MIT Press (2006)
11. Girard, J.-Y.: Linear logic. Theoretical Computer Science 50(1) (1987)
12. Jones, C.B.: Systematic software development using VDM, 2nd edn. Prentice-Hall, Inc., Upper Saddle River (1990)
13. Jones, C.B.: Reasoning about partial functions in the formal development of programs. Electron. Notes Theor. Comput. Sci. 145 (January 2006)
14. Jones, C.B., Lovert, M.J.: Semantic Models for a Logic of Partial Functions. Int. J. Software and Informatics 5(1-2) (2011)
15. Kang, E., Jackson, D.: Formal Modeling and Analysis of a Flash Filesystem in Alloy. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) ABZ 2008. LNCS, vol. 5238, pp. 294–308. Springer, Heidelberg (2008)
16. Leino, K.R.M.: Dafny: An Automatic Program Verifier for Functional Correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS, vol. 6355, pp. 348–370. Springer, Heidelberg (2010)
17. Parnas, D.L.: Predicate logic for software engineering. IEEE Trans. Softw. Eng. 19 (September 1993)
18. Spivey, J.M.: Understanding Z: a specification language and its formal semantics. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press (1988)
19. Stoddart, B., Dunne, S., Galloway, A.: Undefined Expressions and Logic in Z and B. Formal Methods in System Design 15 (1999)
20. Torlak, E.: A Constraint Solver for Software Engineering: Finding Models and Cores of Large Relational Specifications. PhD thesis, MIT (2008)
21. Torlak, E., Jackson, D.: Kodkod: A Relational Model Finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer, Heidelberg (2007)

Extending Alloy with Partial Instances

Vajih Montaghami and Derek Rayside

University of Waterloo
{vmontagh,drayside}@uwaterloo.ca

Abstract. Kodkod, the backend of Alloy4, incorporates new features for solving models where part of the solution, that is, a *partial instance*, is already known. Although Kodkod has had this functionality for some time, it is not explicitly available to the modeller through the Alloy language syntax. We propose an extension to the Alloy language to make partial instances explicitly available to the Alloy user. Explicit partial instances are helpful for the Alloy user in a number of capacities, including test-driven development, regression testing, modelling by example, and combined modelling and meta-modelling. The proposed syntax also gives the modeller explicit access to the performance benefits of Kodkod's partial instance features.

1 Introduction

Five years ago, while introducing Kodkod [10,9], Torlak & Jackson wrote that *Alloy's main deficiency as a general-purpose problem description language is its lack of support for partial instances* [10]. (Kodkod is the backend of Alloy4.) This statement is still true for the majority of Alloy users today: despite Kodkod's support for partial instances, the Alloy language has not yet been extended to explicitly support them. In this paper, we propose a syntactic extension to the Alloy language that exposes this functionality of Kodkod. We also discuss several reasons why Alloy users might find this functionality useful. While Torlak & Jackson [10] demonstrate that Kodkod performs well on problems with partial instances, they do not describe the software engineering benefits of integrating partial instances with Alloy models.

Figure 1 introduces our syntax extension by describing three instances of a linked list: simple, single, and cyclic. In the simple instance, the line Node = head + middle + tail says that there are exactly three node atoms and their names are head, middle, and tail. The next two lines give exact bounds for the next and val relations in terms of these atoms and the integers. The single and cyclic instances are defined in a similar manner.

The inst block gives the Alloy user direct access to Kodkod's partial instance feature. Previously, if the specifier wished to specify an instance, then she would have had to do it implicitly either by constraint or by a constant function. Consider the phrase val = n→0 which gives an exact bound for the val relation in the single instance of Figure 1. In Alloy4, the specifier could have achieved a similar semantic result with the constraint fact {val = n→0} or by commenting out the

J. Derrick et al. (Eds.): ABZ 2012, LNCS 7316, pp. 122–135, 2012.

val relation declaration and introducing a constant function of the same name: fun val[] : Node→Int {n→0}. As we describe below, our new syntax extension affords the specifier greater clarity and modularity, and corresponds to a more consistently efficient translation.

```
1 sig Node { next : lone Node, val : one Int }
2 inst simple { Node = head + middle + tail, -- introduce three atoms
3               next = head→middle + middle→tail, -- exact bound for next relation
4               val = head→0 + middle→1 + tail→2 } -- exact bound for val relation
5 inst single { Node = n, no next, val = n→0 }
6 inst cyclic { Node = a + b, next = a→b + b→a, val = a→0 + b→1}
```

Fig. 1. Alloy model of a linked list with instances expressed in proposed syntax

Paper organization. Section 2 describes four ways in which partial instances benefit the Alloy user: test-driven development, regression testing, modelling by example, and combined modelling and meta-modelling. Section 3 describes our proposed extension to Alloy. Section 4 presents two experiments that demonstrate the increased computational efficiency of directly exposing Kodkod's partial instance feature when compared to adoption of traditional Alloy syntax. Section 5 considers two other possible ways to make Kodkod's partial instance feature available to Alloy users, and argues that our main proposal is preferable. Section 6 concludes.

2 Using Alloy with Partial Instances

We explore four use cases that demonstrate the utility of adding partial instances to the Alloy surface syntax: test-driven development, regression testing, modelling by example, and combined modelling and meta-modelling.

2.1 Test-Driven Development

Partial instances enables modellers to apply the test-driven development [2] methodology to their Alloy models. Consider the following example scenario. When we teach Alloy to senior undergraduates, the first in-class exercise is to write invariants for a binary tree. The lecturer, who has a computer running Alloy, displays the skeletal Alloy model listed in Figure 2.

The lecturer runs the simulation, the class looks at the result and tells the lecturer in plain language what is wrong with the displayed instance, and then the lecturer translates that plain language into formal constraints within the wellFormedTree predicate.

During this initial exercise, it is common for students to identify an instance of the model where some node y is both the left and right child of some node x. When

```
1 sig Node { left, right : lone Node, val : one Int}
2 pred wellFormedTree[] { } -- to be filled in by students
3 run wellFormedTree for 3
```

Fig. 2. A skeletal Alloy model of a binary tree

this occurs, the students usually give a constraint such as 'the left and right children cannot be equal,' which the lecturer translates as all n : Node | n.left !=n.right. The students tend to be satisfied with this translation, but the astute reader will notice that this formalization prevents leaf nodes, forcing the tree to be cyclic (*i.e.*, a leaf node has no left child and no right child, and clearly the empty set is equal to the empty set). The students typically do not realize this overconstraint for fifteen or twenty minutes.

Had the students been following test-driven development with partial instances, they may have realized the folly of the proposed formalization sooner. Suppose that the students had first written the two simple partial instances in Figure 3. Figure 3a lists a tree of a single node that the students expect to be legal. Figure 3b lists a tree with self-loops that the students expect to be illegal. When the wellFormedTree predicate is empty at the beginning of the lecture the illegal self-loops test fails. When the bogus constraint n.left != n.right is added then the singleton tree test fails. Having concrete tests (partial instances) to detect errors in the program (model) is the essence of test-driven development.

(a)
```
1 inst SingletonTree { Node = n, no left, no right, val = n→0 }
2 run wellFormedTree for SingletonTree expect 1
```

(b)
```
1 inst IllegalSelfLoops { Node = n, left = n→n, right = n→n }
2 run wellFormedTree for IllegalSelfLoops expect 0
```

Fig. 3. Two partial instances of a binary tree: (a) a legal singleton tree, and (b) a tree with illegal self-loops

A difference between test-driven development for imperative code versus that for declarative logic models is the role of positive and negative examples. With imperative code, the programmer writes positive test cases for empty procedures (or code stubs) that initially fails. With declarative logic models, a positive example (such as a singleton tree) will succeeds with an empty wellFormedTree predicate. Only once the predicate becomes overconstrained will the positive example fail. In contrast, negative examples will fail with the empty predicate, and will only pass with a properly constrained predicate. Consider, for example, the negative example of a node that is its own child in Figure 3b. If wellFormedTree is underconstrained (*e.g.*, empty) then this test will fail. Thus, the programmer builds up a procedure to construct positive examples, the modeller builds up a predicate to rule out negative examples.

2.2 Regression Testing of Alloy Models

Like programs, specifications evolve: requirements change, extra properties need to be checked, refactoring for readability, and so on. As with programs, some form of regression testing can provide assurance that the specification (or program) still corresponds to programmer intent.

For an Alloy specification with associated safety properties, partial instances can be used in regression testing to detect over-constrained models. When a model becomes over-constrained, the safety properties will still hold; however, the modeller might be unaware of over-constraints. Regression testing of Alloy instances can be effective in detecting these occurrences.

The user following a TDD approach can have their initial tests do double duty as regression tests.

2.3 Modelling by Example

The idea of modelling by example [7] is that the system induces logical constraints through a dialogue of examples with the user. The user begins by providing some prototypical instances to the system, and then the system responds with other instances that the user classifies as either valid or invalid. As the dialogue continues the system refines a general formula that includes the positive examples and excludes the negative examples.

A modelling by example system would be substantially facilitated by having explicit syntactic support for partial instances in Alloy.

2.4 Combined Modelling and Meta-modelling

Alloy is sometimes used to define new modelling languages. We will refer to such activity as 'meta-modelling.' Let L name the Alloy model that describes the new language, and let M name an Alloy model that describes a model written in the new language. At present, there is often no mechanical connection between L and M. Our facility for adding partial instances to Alloy makes it easier to have L and M tightly integrated. We examine the work of Cai & Sullivan *et alia* as a case study to illustrate these points.

In a series of papers over the last ten years Cai & Sullivan *et alia* have been exploring formal techniques for assessing modularity in software design [8,3,5,4]. This is a serious, high-quality research effort that (we claim) illustrates some of the shortcomings of the current Alloy surface syntax that our proposal for integrating partial instances addresses.

Cai & Sullivan have written their meta-model (L) in Z [3]. This meta-model is then implicitly encoded in the Java source of their tool Simon. Given a model of a software design in their language, Simon produces a specialized Alloy model (M) that is used to check modularity properties of the proposed software design. There is no mechanically analyzed connection between L and M.

We have translated the Cai & Sullivan meta-model from Z to Alloy and used our partial instance feature to write some of Cai & Sullivan's specific models

```
1  inst IrwinMatrixDesignSpace {
2    AugmentedConstraintNetwork = ACN,
3    Variable = Density + Struct + Alg,
4    Value = dense+sparse + links+array + traverse+lookup + other,
5    domain = Density→(dense+sparse) + Struct→(links+array+other) +
6      Alg→(lookup+traverse+other),
7    dominates = ACN →((Struct→Density)+(Alg→Density)),
8    solutions = ACN →Solution,
9  }{−−Appended facts have access to atom names introduced in inst block
10   all s : Solution | {
11    let x = {p : Variable, q : Value | some b : s.bindings | p=b.var and q=b.val} |{
12      (Struct→links) in x ⇒ (Density→sparse) in x
13      (Struct→array) in x ⇒ (Density→dense) in x
14      (Alg→lookup) in x ⇒ (Struct→array) in x
15      (Alg→traverse) in x ⇒ (Struct→links) in x
16   }}
17 }
18 run createMatrixACN for IrwinMatrixDesignSpace
```

Fig. 4. Partial instance encoding of Irwin *et alia*'s description [6] of the design space for a matrix manipulation program

as partial instances of this meta-model. Figure 4 lists our encoding of Cai & Sullivan's study of Irwin *et alia*'s example of designing a program to store and manipulate a matrix [6]. There are three variables (decisions) in this design space (line 3): the density of the matrix, the underlying data structure used to encode the matrix, and the algorithm used to manipulate that structure. More specifically, the matrix may be dense or sparse, the structure may be a linked list or an array (or other), and the algorithm may be either 'lookup' or 'traversal' (or other) (lines 4–6). In the vocabulary of Cai & Sullivan, the density decision dominates the data structure and algorithm decisions (line 7). The intuition here is that one selects the data structure and algorithm depending on whether the density is expected to be dense or sparse. Additionally, the partial instance block is followed by a list of facts (lines 9–17) that constrain valid solutions of the design space to those where the algorithm and data structure are natural matches for the matrix density and each other. A fact appended to a partial instance block can make use of the atom names introduced in that block.

We now have a mechanically analyzed connection between the Cai & Sullivan meta-model and the specific model of the design space of a matrix manipulation program. We have greater certainty that the properties we have checked on the meta-model also hold of the model (partial instance).

Clafer [1] is a language that is designed to support combined modelling and meta-modelling.

3 Language Extension

We propose to add an inst block to the Alloy language, allowing the user to specify a partial instance, as illustrated above in Figures 1, 2, 3, and 4. The partial instances in those examples only use exact bounds; Kodkod and our syntax also support lower and upper bounds as well, using the in and includes keywords, respectively. Lower bound is a set of tuples that a relation must have, and upper bound is the one that realation might have [9].

These inst blocks are given names and used in Alloy commands. Whereas now a user might write run p for 3, they will now write run p for i, indicating that predicate p is to be simulated in the context of partial instance i.

An inst block, like a sig block, may have an appended fact. For inst blocks, the appended fact is only expected to be true when that inst block is part of the command being executed. The purpose of this appended fact is to give the specifier an opportunity to write constraints that mention the atom names introduced in the inst block — these names are not available elsewhere in the model.

(a) Grammar

⟨*iBlk*⟩ := 'inst' id ('extends' id)? '{' ⟨*iSt*⟩[,⟨*iSt*⟩]* '}' ('{' ⟨*frml*⟩ '}')?

⟨*iSt*⟩ := ⟨*n*⟩
| 'exactly' ⟨*n*⟩ ⟨*var*⟩
| ⟨*var*⟩ '=' ⟨*iXpr*⟩
| ⟨*var*⟩ 'in' ⟨*iXpr*⟩
| ⟨*var*⟩ 'include' ⟨*iXpr*⟩
| ⟨*var*⟩ 'include' ⟨*iXpr*⟩ 'moreover' ⟨*iXpr*⟩
| 'no' ⟨*var*⟩

⟨*iXpr*⟩ := ⟨*iXpr*⟩ '->' ⟨*iXpr*⟩
| ⟨*iXpr*⟩ '+' ⟨*iXpr*⟩
| '(' ⟨*iXpr*⟩ ')'
| ⟨*atm*⟩

(b) Preliminary type definitions

⟨*prb*⟩ := ⟨*univ*⟩ ⟨*iSt*⟩* ⟨*frml*⟩*

⟨*univ*⟩ := {⟨*atm*⟩[,⟨*atm*⟩]*}

⟨*tpl*⟩ := ⟨*atm*[,*atm*]*⟩

⟨*cnst*⟩ := {tpl[,tpl]*} | {} [×{}]*

⟨*var*⟩ := id

⟨*atm*⟩ := id

⟨*sig*⟩ := ⟨*var*⟩

⟨*sigs*⟩ := ⟨*sig*⟩*

⟨*n*⟩ := int

Fig. 5. Grammar and preliminary type definitions

Figure 5a lists the grammar for our proposed extension to the Alloy language to support partial instances. An iBk has a name, a list of iSts and optionally an appended fact. Each iSt alternative that contains a var bounds either a signature or a field (whichever is named by the var). The one iSt alternative that does not name a var provides the default number of atoms for each signature. A relation (signature or field) name can only appear on the left-hand side of at most one iSt in each iBk.

An iSt that names a signature on its left-hand side introduces atom names on its right-hand side. These atom names can then be used to describe the bounds on fields. An iXpr is an expression that describes a set of tuples using the normal Alloy union (+) and cross-product (→) operators along with the names of the atoms. If the user wishes to specify both an upper and lower bound for relation r, they can write an iSt like r include x + y moreover p + q, which specifies a lower bound of x + y and an upper bound of x + y + p + q.

One partial instance block may extend another. For example, the partial instance in Figure 10 extends the partial instance in Figure 4. The semantics of partial instance extension are simply concatenation and conjunction. Let p name the base partial instance block; let q name the extending partial instance block; and let r name the result of applying the extension to q. The text of r is the concatenation of the text of p with the text of q. The appended fact of r is the conjunction of p's appended fact with q's appended fact. The result r must follow the same well-formedness guidelines as p and q: no relation can be named on the left-hand side of more than one statement. This restriction keeps both regular semantics and extension semantics simple, as it prevents statements from interfering with each other (notwithstanding quantitative statements that interact with named statements in a well-defined manner as formalized below).

$evr : sig \to univ$
$U : iBlk \to sigs \to evr$
$G : iSt^* \to sigs \to evr \to univ$ $\quad G' : iSt \to sigs \to evr$
$X : iSt^* \to sigs \to evr \to evr$ $\quad X' : iSt \to sigs \to evr \to evr$
$N : iSt^* \to sigs \to evr$ $\quad N' : iSt \to sigs \to evr$
$K : sig \to int \to univ$
$Q : iXpr \to univ$

$$
\begin{array}{ll}
U[\![iBlk, sigs]\!] & := G[\![iSt_1 \ldots iSt_n, sigs, X[\![iSt^*, sigs, N[\![iSt^*, sigs, \emptyset]\!]]\!]]\!] \\
G[\![iSt^*, sig, evr]\!] & := G[\![iSt_1 \ldots iSt_n, sigs, evr]\!] \\
G[\![iSt_1 \ldots iSt_n, sigs, evr]\!] & := G[\![iSt_2 \ldots iSt_n, sigs, evr \ +\!+ \ G'[\![iSt_1, sigs]\!]]\!] \\
G[\![[], sigs, evr]\!] & := evr \\
G'[\![v \ [=|\mathbf{in}|\mathbf{include}] \ p, sigs]\!] & := \{(a,b) | a \in sigs \wedge a = v \wedge b \in Q[\![p]\!]\} \\
G'[\![v \ \mathbf{include} \ p \ \mathbf{moreover} \ q, sigs]\!] & := \{(a,b) | a \in sigs \wedge a = v \wedge b \in Q[\![p]\!] \cup Q[\![q]\!]\} \\
X[\![iSt^*, sigs, evr]\!] & := X[\![iSt_1 \ldots iSt_n, sigs, evr]\!] \\
X[\![iSt_1 \ldots iSt_n, sigs, evr]\!] & := X[\![iSt_2 \ldots iSt_n, sigs, evr \ +\!+ \ X'[\![iSt_1, sigs]\!]]\!] \\
X[\![[], sigs, evr]\!] & := evr \\
X'[\![\mathbf{exactly} \ n \ v, sigs]\!] & := \{(a,b) | a \in sigs \wedge \ a = v \wedge b \in K[\![v,n]\!]\} \\
N[\![iSt^*, sigs, evr]\!] & := N[\![iSt_1 \ldots iSt_n, sigs, evr]\!] \\
N[\![iSt_1 \ldots iSt_n, sigs, evr]\!] & := N[\![iSt_2 \ldots iSt_n, sigs, evr \ +\!+ \ N'[\![iSt_1, sigs]\!]]\!] \\
N[\![[], sigs, evr]\!] & := evr \\
N'[\![n, sigs]\!] & := \{(a,b) | a \in sigs \wedge b \in K[\![a,n]\!]\} \\
K[\![v,n]\!] & := \{\langle ToString(v) +' \ \$' + ToString(n-1)\rangle\} \cup K[\![v, n-1]\!] \\
K[\![v,0]\!] & := \langle\rangle \\
Q[\![p]\!] & := \{\langle ToString(p)\rangle\} \\
Q[\![p+q]\!] & := Q[\![p]\!] \cup Q[\![q]\!] \\
Q[\![p \to q]\!] & := \langle\rangle
\end{array}
$$

Fig. 6. Universe construction

3.1 Semantics

We define the semantics of the partial instance block as an extension of the Kodkod semantics [9]. The Kodkod semantics take a universe and relation bounds as inputs. The purpose of the partial instance block is for the user to specify the universe and relation bounds.

Figure 6 describes how the universe is constructed from a partial instance block by the U function, which in turn makes use of the N, X, and G functions. Preliminary type definitions are given above in Figure 5b. First the N function constructs a universe in which each sig has the default number of atoms. The X function takes this default universe and returns a universe that complies with

the exactly statements in the partial instance block. Finally, the G function adds atoms named in upper and lower bound statements. All of these functions take as input a set of the sigs declared in the model. This set of sig names is used to distinguish statements that might introduce atoms (which name a sig on the left-hand side) from statements that bound relations (which name a field on the left-hand side).

Once the universe is constructed (Figure 6), then the bounds can be constructed (Figure 7). Figure 7 starts by redefining the top-level function P from the Kodkod semantics [9] to indicate that the universe and the relation bounds are generated from the partial instance block.

$P : problem \to binding \to boolean$ — top-level function, re-defined from [9]
$F : formula \to binding \to boolean$ — formulas, definition given in [9]
$S : iSt^* \to sigs \to evr \to binding \to boolean$ — list of inst statements
$S' : iSt \to sigs \to evr \to binding \to boolean$ — individual inst statement
$C : iXpr \to univ \to cnst$ — expressions
$W : var \to sigs \to evr \to univ$ —

$$P[\![sigs.U[\![iBk, sigs]\!]\ iSt_1 \ldots iSt_n\ frml^*]\!]_b := S[\![iSt_1 \ldots iSt_n, sigs, U[\![iBk, sigs]\!]]\!]_b \wedge F[\![frml^*]\!]_b$$
$$S[\![iSt_1 \ldots iSt_n, sigs, evr]\!]_b := S[\![iSt_2 \ldots iSt_n, evr, sigs]\!]_b \wedge S'[\![iSt_1, evr, sigs]\!]_b$$
$$S[\![], evr, sigs]\!]_b := true$$
$$S'[\![\textbf{exactly}\ n\ v, evr, sigs]\!]_b := W[\![v, sigs, evr]\!] \subseteq b(v) \subseteq W[\![v, sigs, evr]\!]$$
$$S'[\![v\textbf{=}p, evr, sigs]\!]_b := C[\![p, sigs.evr]\!] \subseteq b(v) \subseteq C[\![p, sigs.evr]\!]$$
$$S'[\![v\ \textbf{in}\ p, evr, sigs]\!]_b := C[\![\emptyset, sigs.evr]\!] \subseteq b(v) \subseteq C[\![p, sigs.evr]\!]$$
$$S'[\![v\ \textbf{include}\ p, evr, sigs]\!]_b := C[\![p, sigs.evr]\!] \subseteq b(v) \subseteq W[\![v, sigs, evr]\!]$$
$$S'[\![v\ \textbf{include}\ p\ \textbf{moreover}\ q, evr, sigs]\!]_b := C[\![p, sigs.evr]\!] \subseteq b(v) \subseteq C[\![p+q, sigs.evr]\!]$$
$$S'[\![\textbf{no}\ v]\!]_b := b(v) = \emptyset$$
$$C[\![p+q, univ]\!] := C[\![p, univ]\!] \cup C[\![q, univ]\!]$$
$$C[\![p \to q, univ]\!] := \{\langle p_1, \ldots, p_n, q_1, \ldots, q_m\rangle | \langle p_1, \ldots, p_n\rangle \in C[\![p, univ]\!] \wedge \langle q_1, \ldots, q_m\rangle \in C[\![q, univ]\!]\}$$
$$C[\![p, univ]\!] := \{p' | p' \in univ\ \wedge\ ToString(p') = p\}$$
$$W[\![v, sigs, evr]\!] := \{\langle p_1, \ldots, p_n\rangle | (v \in sigs\ \implies\ p_1 \in v.evr) \wedge (v \notin sigs\ \implies\ p_i \in v_i.evr)\}$$

Fig. 7. Bounds construction (building on formalization of [9])

4 Experiments

We performed two experiments to evaluate the computational efficiency of the proposed partial instance block: a micro-benchmark to characterize the maximum possible improvement, and our combined modelling and meta-modelling case-study based on Cai & Sullivan's work (§2.4). All tests are done on Intel i7-2600K CPU at 3.40GHz with 16GB memory. The performance results are essentially the same with both Minisat and Sat4J, although we report only the Sat4J results here.

We compared using the partial instance block to two alternative specification styles in two different versions of Alloy 4.2. The two different styles were constraining relations with facts and using constant functions instead of relations. Constant functions are just expressions that are inlined at their point of use. They add clauses but not variables to the generated SAT formula. Alloy 4.x includes some inference capability to translate constraints on relations as bounds. In response to a draft of this paper the Alloy development team improved this

inference capability. We refer to this enhanced version as A4.2′, and to the version of Alloy 4.2 from January 2012 as A4.2. We refer to our version of Alloy with the partial instance block as A4.2*i*.

4.1 Micro Benchmark

We devised a micro-benchmark to illustrate the upper bound on the potential performance improvements of exposing Kodkod's partial instance features through our new syntax. Our micro-benchmark has a single signature S and a single binary relation r that maps S to S. For our partial instance, we want to introduce some named atoms of sig S, and then define relation r to be a fully connected graph (*i.e.*, map every S atom to every other S atom).

Figure 8 lists examples of these partial instance models in the three different syntaxes: *(a)* constraining relation r with a fact; *(b)* replacing relation r with a constant function named r; and *(c)* using our new partial instance syntax. The example listings in Figure 8 show these models where signature S has two atoms (S0 and S1). For the plots in Figure 9, we generated these models with signature S having up to seventy-five atoms. The cardinality of relation r is proportional to the square of the cardinality of signature S (as one would expect from a fully connected graph).

(a) By Fact	*(b)* By Constant-Function	*(c)* By Inst-Block
one sig S0,S1 **extends** S{} **fact** {r=S0→S1 + S1→S0} **pred** f[]{**all** s:S | S **in** s.^r} **run** f	**one sig** S0,S1 **extends** S{} **fun** r[]:S→S{S0→S1 + S1→S0} **pred** f[]{**all** s:S | S **in** s.^r} **run** f	**inst** b { S=S0 + S1, r=S0→S1 + S1→S0} **pred** f[]{**all** s:S | S **in** s.^r} **run** f **for** b

Fig. 8. Example models for micro-benchmark experiment

Figure 9 shows graphs characterizing how the translations of the three syntactic approaches shown in Figure 8 scale on different measures: *(a)* total number of variables in the resulting boolean (SAT) formula; *(b)* number of primary variables in the resulting boolean (SAT) formula; *(c)* time taken by Kodkod to translate the Alloy model to SAT; and *(d)* time taken by the SAT solver to find a solution. We make a number of observations from the data in Figure 9:

1. The inference capability of A4.2 is incomplete: it is unable to deduce that the constraints on r can be translated as bounds rather than as variables and clauses. Therefore the number of variables and the translation and solving times grow exponentially.
2. All other strategies show very little growth as the number of atoms increases.
3. The improved inference in A4.2′ is effective (A4.2′ Fact column).
4. The number of SAT variables produced by the constant function encoding, the improved inference, and the partial instance strategies is the same (low).
5. The partial instance encoding has the fastest translation and solving times (by a narrow margin).

The main conclusion of Figure 9 is that if the specifier chooses to use constant functions instead of relations or writes their facts in a manner that Alloy can infer bounds from, then there is little performance gain from the partial instance block. However, the partial instance block does provide the best performance, and does so without the specifier having to worry about whether their writing style is comprehensible to Alloy's bounds inference facility.

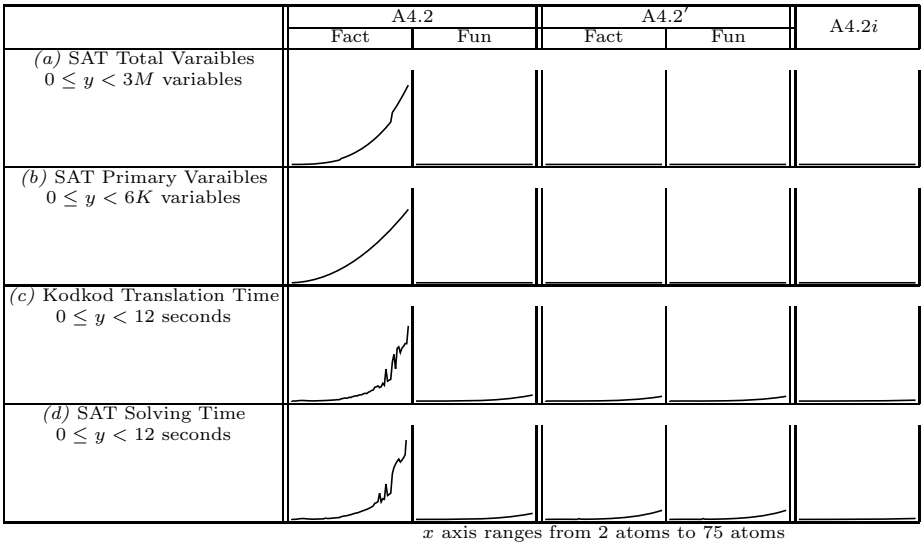


Fig. 9. Results of micro-benchmarks

4.2 Staged Evaluation

The proposed partial instance feature offers Alloy users the opportunity to stage evaluation of their models, which might potentially save time when certain parts of the model are not changing and other parts are. Consider, for example, the model in Figure 4 that describes the design space of a program to manipulate matrices. The partial instance of Figure 4 is written in terms of Cai & Sullivan's meta-model, which has (design) variables, values, bindings of variables to values, and 'states'. (The 'states' are of a design automaton, which is a concept they use to analyze design spaces that we do not explain here.)

Suppose that the user wishes to experiment with the constraints written in the appended fact of Figure 4. These constraints do not affect the space of valid binding atoms. Therefore, the user could stage the evaluation of the model by saving the legal bindings in a partial instance, such as in Figure 10. Subsequent simulations would not have to re-solve this part of the model.

```
1 inst IrwinMatrixDesignSpace_WithBindings extends IrwinMatrixDesignSpace {
2   Binding = B0+B1+B2+B3+B4+B5+B6+B7,
3   var = (B0+B1)→Struct + (B2+B3+B4)→Density + (B5+B6+B7)→Alg,
4   val = B0→dense + B1→sparse + B2→links + B3→array + B4→other +
5         B5→traverse + B6→lookup + B7→other }
6 run createMatrixACN for IrwinMatrixDesignSpace_WithBindings
```

Fig. 10. Iwrin matrix design space partial instance (Figure 4) extended with binding atoms generated by a previous simulation

Figure 11 characterizes the potential performance improvements from staged evaluation using the Irwin matrix design space example of Cai & Sullivan. The translation time for the model from Figure 10 is over ten times faster than the translation time for the model from Figure 4, and the solving time is three times faster, for an overall improvement of seven times. Obviously the speedup to be gained from staged evaluation depends on the particulars of the model in question; other models will likely produce different results than this one.

Figure 11 also shows performance results for A4.2 and A4.2′ simulating a model equivalent to Figure 4 (*i.e.*, not staged). In this particular case there is no significant difference between A4.2 and A4.2′. We suspect that this is the case because the domain relation is constrained piecewise across a number of appended facts. All of these piecewise constraints add up to an exact bound on domain, but a fairly sophisticated whole-model analysis would be needed to deduce that. A4.2*i* results in four times faster solving time than A4.2′ for the model in Figure 4, at the expense of a 10% slowdown in translation time.

	Total Vars	Pri. Vars	Clauses	Translation time (ms)	Solving time (ms)
A4.2*i* (Fig. 4)	59,694	773	162,642	12,742	6,744
A4.2*i* (Fig. 10 — staged)	20,060	503	37,148	986	2,174
A4.2	59,953	768	162,417	11,976	27,415
A4.2′	59,953	768	162,417	11,188	27,730

Fig. 11. Performance improvements from staged evaluation

5 Alternatives Considered

In this section we consider some alternative approaches for specifying partial instances in Alloy and argue for the approach proposed in this paper.

5.1 Static Analysis

Alloy 4.x already includes the capability to infer when constraints might be encoded as Kodkod bounds rather than as SAT clauses. Although it is not yet

perfect, this capability will continue to improve. Given this capability, no extra syntax is needed to realize the main performance benefits of Kodkod's partial instance feature.

We argue that there are software engineering benefits to our new syntax beyond the performance gains that it affords. The proposed syntax makes it easy for the specifier to run different commands with different instances, or to run commands with no partial instance (the norm in Alloy now). Writing a partial instance implicitly via constraints in the traditional Alloy syntax makes it difficult to switch from running a command with a partial instance to running a command without a partial instance. For example, to run the fragment in Figure 8a without a partial instance, we would want to remove the keyword abstract from the signature S and remove the sub-signatures S1, S2, S3. With the partial instance block syntax, one does not have to edit the text of the model to run it in these different ways. A number of our use cases described above depend on this affordance of the new syntax.

5.2 Syntactic Alternatives for the Partial Instance Block

There are a variety of different ways in which one could specify the body of a partial instance block. We consider the proposal described above to be a 'relational' style because each statement specifies a different relation.

Alternatively, one could imagine an 'object-oriented' syntax in which relations are defined piecewise with respect to individual atoms. Figure 12a lists a small example of this syntax. The same example is listed in the relational style in Figure 12c. The object-oriented style syntax is intuitively appealing for some examples; however, its piecewise nature makes the bound being defined unclear: does Figure 12a define a lower bound or an exact bound for relation r?

Another alternative syntax is 'set-oriented' style, shown in Figure 12b. This style is concise and consistent with common mathematical notation, but it does not conform to the existing Alloy expression grammar.

Our proposed relational style syntax (Figure 12c) conforms to the existing Alloy expression grammar and has a clear and uniform way to specify lower, exact, and upper bounds.

(a) object-oriented style	*(b)* set-oriented style	*(c)* relational style
sig S{r: S} **inst** i{S=S1+S2+S3, S1.r=S2, S2.r=S3}	**sig** S{r: S} **inst** i{S={S1,S2,S3}, r={S1→S2,S2→S3}}	**sig** S{r: S} **inst** i{S=S1+S2+S3, r=S1→S2+S2→S3}

Fig. 12. Syntactic alternatives for the body of the partial instance block

6 Conclusion

Explicit partial instances could be used in Alloy to efficiently specify constraints on allowable solutions (their intended usage in Kodkod); for test-driven development of Alloy models; for regression testing of Alloy models; to support new ideas such as modelling by example; and for combined modelling and meta-modelling. While Alloy currently has an inference mechanism that makes use of Kodkod's partial instance functionality behind the scenes, these engineering benefits are substantially facilitated by explicit syntactic support for partial instances.

There is more than one possible way to expose Kodkod's partial instance feature to the Alloy user. We have explored a number of alternatives and recommend a new named block with statements written in a relational style. This recommendation is backwards compatible with existing Alloy models and the existing Alloy expression grammar; it affords the user a uniform way to express exact, upper, and lower bounds; it combines with Alloy commands in a modular fashion; and it has an easy and efficient translation to Kodkod.

Kodkod has supported partial instances for five years, and that is one of its main improvements over the backend of Alloy3. It's time that Alloy users had the opportunity to take full advantage of this functionality.

Acknowledgments. We thank Daniel Jackson, Aleksandar Milicevic, Steven Stewart, Emina Torlak, and the anonymous referees for comments on previous drafts of this paper. We also thank Aleksandar Milicevic for his assistance with the Alloy code base and for improving the inference capabilities of the Alloy analyzer.

This work was supported in part by the National Science and Engineering Research Council of Canada (NSERC).

References

1. Bąk, K., Czarnecki, K., Wąsowski, A.: Feature and Meta-Models in Clafer: Mixed, Specialized, and Coupled. In: Malloy, B., Staab, S., van den Brand, M. (eds.) SLE 2010. LNCS, vol. 6563, pp. 102–122. Springer, Heidelberg (2011)
2. Beck, K.: Test-Driven Development. Addison-Wesley, Reading (2003)
3. Cai, Y.: Modularity in Design: Formal Modeling and Automated Analysis. Ph.D. thesis, University of Virginia (August 2006)
4. Cai, Y., Huynh, S., Xie, T.: A framework and tool supports for testing modularity of software design. In: Egyed, A., Fischer, B. (eds.) Proc. 22nd ASE, Atlanta, GA, pp. 441–444 (November 2007)
5. Cai, Y., Sullivan, K.: Modularity analysis of logical design models. In: Easterbrook, S., Uchitel, S. (eds.) Proc. 21st ASE, Tokyo, Japan (September 2006)
6. Irwin, J., Loingtier, J.M., Gilbert, J.R., Kiczales, G., Lamping, J., Mendhekar, A., Shpeisman, T.: Aspect-Oriented Programming of Sparse Matrix Code. In: Sun, Z., Reynders, J.V.W., Tholburn, M. (eds.) ISCOPE 1997. LNCS, vol. 1343, pp. 249–256. Springer, Heidelberg (1997)
7. Mendel, L.: Modeling by Example. Master's thesis, MIT (September 2007)

8. Sullivan, K.J., Griswold, W.G., Cai, Y., Hallen, B.: The structure and value of modularity in software design. In: Proc. 9th FSE, Vienna, Austria, pp. 99–108 (September 2001)
9. Torlak, E.: A Constraint Solver for Software Engineering: Finding Models and Cores of Large Relational Specifications. Ph.D. thesis, MIT (2009)
10. Torlak, E., Jackson, D.: Kodkod: A Relational Model Finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer, Heidelberg (2007)

Toward a More Complete Alloy*,**

Timothy Nelson[1], Daniel J. Dougherty[1],
Kathi Fisler[1], and Shriram Krishnamurthi[2]

[1] Worcester Polytechnic Institute
[2] Brown University

Abstract. Many model-finding tools, such as Alloy, charge users with providing bounds on the sizes of models. It would be preferable to automatically compute sufficient upper-bounds whenever possible. The Bernays-Schönfinkel-Ramsey fragment of first-order logic can relieve users of this burden in some cases: its sentences are satisfiable iff they are satisfied in a finite model, whose size is computable from the input problem.

Researchers have observed, however, that the class of sentences for which such a theorem holds is richer in a many-sorted framework—which Alloy inhabits—than in the one-sorted case. This paper studies this phenomenon in the general setting of order-sorted logic supporting overloading and empty sorts. We establish a syntactic condition generalizing the Bernays-Schönfinkel-Ramsey form that ensures the Finite Model Property. We give a linear-time algorithm for deciding this condition and a polynomial-time algorithm for computing the bound on model sizes. As a consequence, model-finding is a complete decision procedure for sentences in this class. Our work has been incorporated into Margrave, a tool for policy analysis, and applies in real-world situations.

1 Introduction

The undecidability of first-order logic poses a challenge to using Alloy for verification: analysis performed under bounds may not be complete. While incompleteness is unavoidable for some classes of formulas, there are also classes for which analysis is complete under domains of finite size. Alloy asks users to specify domain-size bounds, but does not help users determine whether their bounds suffice for completeness. Ideally, tools such as Alloy would provide such feedback or, better still, compute sufficient bounds automatically when possible.

Sufficient-bounds results are long-established for classical first-order logic. Alloy's logic, however, is different in ways that impact computing bounds. Alloy signatures yield first-order logic with *sorts*: the class of many-sorted first-order logic formulas with sufficient bounds properly includes that for the unsorted case. Existing results on sufficient bounds for many-sorted logic, however, make

* This research is partially supported by the NSF.

** An expanded version of this paper, with complete proofs, is available at `http://tinyurl.com/osepl-tr-pdf`

J. Derrick et al. (Eds.): ABZ 2012, LNCS 7316, pp. 136–149, 2012.

assumptions that are not valid for many Alloy specifications: Alloy allows sorts to be empty, and also allows sorts to overlap. These features, which are critical for modeling realistic systems, require an extended theory of bounds-computation. This paper presents the theory and algorithms for computing sufficient bounds for a substantial class of Alloy formulas.

We actively use our results within our Margrave tool (www.margrave-tool.org) for analyzing policies (such as access-control, firewall, and routing policies). One of our standard policy examples—from a deployed conference-paper manager—requires the results in this paper. Margrave uses the presented algorithms to compute how many papers are required for complete reasoning. In other examples, Margrave computes sufficient bounds on *some* sorts (even when others cannot be bounded). This can help a user decide how to allocate the computational resources of model finding. Margrave is built upon Kodkod [20], the backend model-finder for the Alloy Analyzer. For Alloy users, we provide an implementation online (sortedtermcount.appspot.com) that takes a formula σ in Alloy notation, checks whether σ lies in our class of decidable formulas, and computes sufficient sizes for whichever Alloy signatures we can bound.

2 Overview of Results

The Bernays-Schönfinkel-Ramsey class, sometimes called "Effectively Propositional Logic" (EPL), comprises the set of first-order sentences of the form

$$\exists x_1 \ldots \exists x_n \forall y_1 \ldots \forall y_m \; . \; \varphi$$

where φ is quantifier-free and has no function symbols. The satisfiability problem for this class is decidable: Bernays and Schönfinkel [2] and Ramsey [19] showed that such a sentence has a model if and only if it has a model of size bounded by n plus the number of constants in φ. When such a *finite model property* holds, satisfiability-testing reduces to exhaustive search for a model within bounded domains. Furthermore, the search need only consider models whose elements are constants. In effect, satisfiability for these formulas reduces to propositional satisfiability.

The EPL results assume that all variables quantify over the same domain. Alloy uses a *sorted* first-order logic, in which values come from several domains (Alloy signatures) and each variable is associated with a particular domain. Sorts provide additional information that model finders and theorem provers can exploit in the search for models [10,11,13,16]. More strikingly, the class of sentences with the finite model property is richer in a sorted framework [1,6,9]. The following simple example illustrates the interplay between sorts and bounds for completeness. Consider the class of unsorted sentences of the form

$$\forall y_1 \exists x \forall y_2 \; . \; \varphi.$$

Satisfiability is undecidable for this prefix class [3]. In contrast, this sorted verion

$$\sigma \equiv \forall y_1^A \exists x^B \forall y_2^A \; . \; \varphi \tag{1}$$

is better behaved. Suppose that φ contains constants, say n_A constants of sort A and n_B of sort B, but no function symbols. If we were to postulate that sort A is a subsort of sort B, then if σ has any models at all then it has a model whose size at sort A is bounded by n_A and whose size at sort B is bounded by $(2n_A + n_B)$. On the other hand, if our signature declared that B were a subsort of A, then some σ would only have infinite models. In considering subsort relationships, this example illustrates *order-sorted* logic, in which there is a partial order on the sorts rather than an assumption that all sorts are disjoint. We give a formal treatment of this example below, as Example 15.

Alloy's use of order-sorted logic, as well as its allowance of empty sorts, demand new methods for computing sufficient bounds. To illustrate why, we first consider a standard approach to establishing the finite-model property. Let σ be a sentence in unsorted first-order logic.

1. By Skolemization, there is a universal sentence σ_{sk} equi-satisfiable with σ. The language of σ_{sk} is richer than that of σ, since constants and function symbols have been introduced on behalf of existential quantifiers of σ.
2. Any potential model $\mathcal{M}$ for σ_{sk} has a *Skolem hull* [4] consisting of the interpretation in the model of the ground terms of the language. The set of ground terms is called the *Herbrand universe*. The Skolem hull forms a submodel of $\mathcal{M}$ in which every element is named by a term in the language.
3. A fundamental classical theorem is that the truth of universal sentences is preserved under submodel. Thus, if the signature of σ_{sk} has only finitely many terms, that is, if the Herbrand universe is finite, then σ has the finite-model property.

When the language has only a single sort, the only way to guarantee that the Herbrand universe is finite is to have no function symbols (other than constants). In that setting, the sentences whose Skolemization produces no function symbols comprise the EPL class. The many-sorted setting is more lenient. Consider for example a sentence σ whose Skolemization leads to a language with simply a constant a of sort A, a function f of sort $A \to B$. Then the only ground terms that can be constructed are a and $f(a)$ (terms such as $f(f(a))$ are not well-sorted). This suggests—correctly—that a richer classes of finite-model results are available.

But there are technical obstacles to generalizing the above argument. In particular,

- When empty sorts are allowed, the Skolem form of σ is not equi-satisfiable with σ. For example the sentence $(\forall y^A.y = y) \vee (\exists x^B.x \neq x)$ is true in models where the sort B is empty. Skolemization, with a new constant b of sort B, yields the sentence $(\forall y^A.y = y) \vee (b \neq b)$ which is is unsatisfiable. Section 5 addresses this issue formally.
- When sorts are not assumed to be disjoint (the order-sorted setting), not every element in the Skolem hull of a model is named by a term. Indeed the Skolem hull of $\mathcal{M}$ can be infinite even when a finite submodel of $\mathcal{M}$ *does* exist. Example 9 in Section 5 illustrates this case.

Contributions. This paper adapts the approaches in the standard argument to accommodate ordered sorts and empty sorts. In doing so, it enables automatic bounds computation for additional Alloy formulas through the following contributions:

- We identify (Definition 11) a syntactically-determined class of sentences extending EPL, comprising *Order-Sorted Effectively Propositional Logic (OS-EPL)*, for which the Finite Model Property holds (Theorem 10, Section 6).
- We present a linear-time algorithm (Corollary 16) for membership in OS-EPL. We present a cubic-time algorithm (Theorem 17) for computing an upper bound on the size of models required for testing satisfiability. It is interesting to note that the bound itself can be exponential in the size of the sentence (Section 7), even though it can be computed in polynomial time.

We view identification of the OS-EPL class as a contribution to a taxonomy of decidability classes in order-sorted logic. In the presence of possibly-empty sorts, sentences do not always have equivalent prenex-normal forms, so we cannot attempt a decidability classification in terms of quantifier prefix as in [3]. As Section 6 shows, our decidability criterion is based entirely on the signature of the Skolemization of the given formula. This signature can be viewed as a generalization of the idea of quantifier prefix, as it implicitly records the pattern of nesting between universal and existential quantification.

2.1 A Sample Application

The PLT Scheme application Continue [12] automates many conference-management tasks. Margrave has been helpful in developing and analyzing Continue; here we hint at some of the ways that the algorithms in this paper have improved some of these analyses.

The access-control policy of a conference can be represented as a first-order theory, over a language representing facts about the world and access-control decisions such as *permit* and *deny*. A user will query the policy to verify or falsify properties of the system; Margrave's mode of interaction is to generate models, or "scenarios" for situations being explored by the user. For example, a certain policy rule might say "The conference administrator can advance the conference out of the `bidding` phase if every reviewer has bid on some paper." This rule gives rise to the sentence

$$permit(s, advancePhase, conference) \longleftarrow Admin(s) \wedge (phase = Bidding) \wedge \forall u^{User} \exists p^{Paper} \,.\, bidOn(u, p)).$$

The set of such policy rules, together with assumed facts about the application domain, comprise a background theory for analysis. Now suppose one wants to verify the property: "The conference chair can modify user passwords." It suffices to determine that there are no models of *the negation of* the formula

$$\forall r^{User} \,.\, permit(chair, modifyPassword, r)$$

together with the background theory. The question, of course, is determining an upper bound on the scope of the search. The fact that the formula being explored by the user is purely existential is of little help by itself, since the entire policy theory is part of the satisfiability query. Besides rules such as the permit rule quoted above, the language also includes such function symbols as $paperPhase : Paper \to PaperPhase$ and $decision : Paper \to Decision$.

In the absence of sensitivity to sorts this theory would not submit to a finite-model discipline. But in fact, for the Continue theory and the associated query above Margrave automatically computes sufficient bounds:

```
Conference:7  PaperPhase:12 Object:14 ConferencePhase:10
Action:16 User:9 Resource:6 univ:59 Paper:6
```

Without the support of the finite-model algorithms of this paper the user would—à la Alloy—have to instruct the tool to restrict attention to a finite search space that was presumably arrived at in an *ad-hoc* manner.

3 Related Work

The decidability of the satisfiability problem for the $\exists\forall$ class in pure logic is a classical result of Bernays and Schönfinkel [2] in the absence of equality, extended by Ramsey [19] to allow equality. The problem is known to be EXPTIME-complete [14].

Goguen and Meseguer did seminal work [8] on order-sorted algebra; order sorted predicate logic was first considered by Oberschelp [18]. Harrison was one of the first to observe that many-sortedness can not only yield efficiencies in deduction but can also support new decidability results. In unpublished notes [9] he presents some examples of this phenomenon, and suggests searching for typed analogs of classical decidability classes, as we have done here. Order-sorted signatures (without relation symbols) can be viewed as tree automata, so the question of whether the set of closed terms is finite can be answered using standard automata techniques. We believe that the algorithm in this paper for counting terms is new.

Fontaine and Gribomont [6], working in "flat" many-sorted logic (*i.e.*, without subsorting) prove that if there are no functions having result sort A and σ is a universal sentence then σ has a model if and only if it has a model in which the size of A is bounded by the number of constants of sort A. This result is used to eliminate quantifiers in certain verification conditions. This theorem has application even when not all sorts are finite and can be used in a setting where some functions and predicates are interpreted.

Claessen and Sorensson [5] have integrated a *sort inference* algorithm into the Paradox model-finder that deduces sort information for unsorted problems and, under certain conditions, can bound the size of domains for certain sorts and improve the performance of the instantiation procedure. Order-sorting is not used, and there are restrictions on the use of equality.

Momtahan [15] computes a refutationally-complete upper bound on the size of a single sort (as a function of the user-provided bounds on the other sorts) for a fragment of the Alloy kernel language. The conditions defining this fragment are not directly comparable to ours, but in some respects constrain the sentences rather severely. For example existential quantification in the scope of more than one universal quantifier is usually not allowed.

Abadi *et al.* [1] identify, as we do, a decidable fragment of sorted logic that is decidable by virtue of having a finite Herbrand universe. Although they target Alloy in their examples they work in a many-sorted logic without subsorts or empty sorts; their condition for decidability is the existence of a "stratification" of the function vocabulary; they do not provide algorithms for checking the stratification condition or computing size bounds on the models.

Ge and de Moura [7] present a powerful method for deciding satisfiability modulo theories with an instantiation-based theorem prover. Given a universal (Skolemized) sentence σ they construct a system of set constraints whose least solution constitutes a set of ground terms sufficient for instantiation; satisfiability is thus decidable for the set of sentences for which this solution-set is finite (in the many-sorted setting this subsumes the Abadi *et al.* class). They do not treat empty sorts nor subsorting. They can treat certain sentences that fall outside our OS-EPL class; detection of whether a given sentence falls into their decidable class seems to require solving the associated set-constraints, as compared to our linear-time algorithm. Generally speaking they do detailed fine-grained analysis of individual sentences; we have focused on an easily recognized class of sentences.

The problem of efficiently deciding satisfiability in the EPL class is an active area of research. Our work is complementary to these efforts in that it identifies an extended class of sentences to which contemporary techniques can hopefully be applied.

A preliminary version of this work was presented at the workshop on Synthesis, Verification, and Analysis of Rich Models (SVARM), July 20-21, 2010.

4 Background: Order-Sorted Predicate Logic and Term Models

We begin by formalizing several foundational concepts that underlie the high-level argument in points 1–3 of Section 2. Naturally, we define signatures and models for order-sorted first-order logic. Finite-model properties derive from arguments that every model of a sentence has a truth-preserving submodel with only finitely-many elements. Two concepts are key to such an argument: homomorphisms between models (and hence submodels), and a *term model*, which is a particular model over the ground-terms of a sentence. Establishing that the term model is a submodel (under homomorphism) of every model of a sentence is essential to proving completeness under finite bounds; a theorem in this section captures this requirement.

The definitions and results in this section are either directly from Goguen and Meseguer's work [8] or they are the obvious extensions required to handle relations as well as functions.

Notation. We use $\langle\rangle$ for the empty sequence. If $(\mathcal{S}, \leq)$ is an ordering we extend $\leq$ to words in $\mathcal{S}^*$ and then to products, pointwise.

Signatures. An *order-sorted signature* is a triple $\mathcal{L} = (\mathcal{S}, \leq, \Sigma)$ where $(\mathcal{S}, \leq)$ is a finite poset of sorts and Σ is an indexed family of symbols, the vocabulary, comprising

- $\{\Sigma_w \mid w \in \mathcal{S}^*\}$, an $\mathcal{S}^*$-sorted family of *relation symbols,* and
- $\{\Sigma_{w,A} \mid w \in \mathcal{S}^* , A \in \mathcal{S}\}$, an $(\mathcal{S}^* \times \mathcal{S})$-sorted family of *function symbols.*

We assume that the Σ_w and $\Sigma_{w,A}$ are pairwise disjoint.

We stress that an order-sorted signature is not the same as an Alloy signature. Such a signature denotes a language of discourse: available sorts, functions, etc. along with an ordering on the sorts. In this way, an Alloy signature is a sort or a predicate within an overall order-sorted signature.

Formalizing relations and functions through words—rather than through tuples of sorts—simplifies certain definitions and eases capturing overloaded function symbols (as [8] does). Our work assumes function symbols are not overloaded (through the disjointness condition on the $\Sigma_{w,A}$); this is consistent with Alloy. Most of the results of the paper generalize to handle overloading, including our finite model theorem (Theorem 14). The one exception is our term-counting algorithm (Theorem 17), which relies on the lack of overloading to compute precise bounds; with overloading, our algorithm only promises upper bounds on the sort-sizes.

When $R \in \Sigma_w$ we say that w is the *arity* of R. When $f \in \Sigma_{w,A}$ we say that w is the *arity* of f and A is the *result sort* of f. If $\mathcal{L} = (\mathcal{S}, \leq, \Sigma)$ and $\mathcal{L}' = (\mathcal{S}, \leq, \Sigma')$ are such that for each w and A, $\Sigma_w \subseteq \Sigma'_w$ and $\Sigma_{w,A} \subseteq \Sigma'_{w,A}$ we say that $\mathcal{L}'$ is an *expansion* of $\mathcal{L}$, and that $\mathcal{L}$ is a *reduct* of $\mathcal{L}'$.

Following standard usage, a function symbol $a \in \Sigma_{\langle\rangle,A}$, taking no arguments, is referred to as a "constant" of sort A, and in concrete syntax we write simply a instead of $a()$.

The *connected components* of an ordering $(\mathcal{S}, \leq)$ are the equivalence classes for the equivalence relation generated by $\leq$. A signature $\mathcal{L} = (\mathcal{S}, \leq, \Sigma)$ is *coherent* if each pair of sorts in the same connected component has an upper bound. Henceforth we assume that our signatures are coherent. See [8] for an extended discussion of the importance of coherence. Note: in [8] the notion of coherence also requires that signatures be *regular*, a technical condition that is trivially satisfied in the absence of overloading.

The set of *formulas* is defined inductively by closing the set of atomic formulas under the propositional operators $\wedge$, $\vee$, $\Rightarrow$ and $\neg$ and the quantifiers $\exists$ and $\forall$. We will indicate quantification over a sorted variable $x \in X_A$ by $\exists x^A$ or $\forall x^A$ (where X_A is the set of variables of sort A). The notions of free and bound variable are standard; let $FV(\varphi)$ denote the set of free variables of formula φ. A *sentence* is a formula with no free variable occurrences.

Models. Fix a signature $\mathcal{L} = (\mathcal{S}, \leq, \Sigma)$. An $\mathcal{L}$-*model* $\mathcal{M}$ comprises (i) an $\mathcal{S}$-sorted family $\{\mathcal{M}_A \mid A \in S\}$ of sets, the *universe* of $\mathcal{M}$, such that $A \leq A'$ implies $\mathcal{M}_A \subseteq \mathcal{M}_{A'}$, (ii) for each $R \in \Sigma_w$ a relation $R^{\mathcal{M}_w} \subseteq \mathcal{M}_w$, and (iii) for each $f \in \Sigma_{w,A}$ a function $f^{\mathcal{M}_{w,A}} : \mathcal{M}_w \to \mathcal{M}_A$.

As described in the introduction, the first step in investigating the finite model property for a sentence is Skolemization, the process of eliminating existential quantifiers in favor of function symbols. As a consequence we need to be attentive to the ways that the language over which our models are defined can shift. If $\mathcal{M}$ is a model for $\mathcal{L} = (\mathcal{S}, \leq, \Sigma)$ and $\mathcal{L}'$ is an expansion of $\mathcal{L}$ then an *expansion* of $\mathcal{M}$ to $\mathcal{L}'$ is a model of $\mathcal{L}'$ with the same universe as $\mathcal{M}$ which agrees with $\mathcal{M}$ on the symbols in Σ.

An *environment* η over a model $\mathcal{M}$ is an $\mathcal{S}$-indexed family of finite functions $\{\eta_A : X_A \to \mathcal{M}_A \mid A \in \mathcal{S}\}$ such that $\eta_A = (\eta_{A'}) \restriction_{X_A}$ (the restriction to X_A) whenever $A \leq A'$. An environment η can be extended to terms in the usual way. When $\mathcal{M}$ is a model, φ a formula, and η an environment such that $FV(\varphi) \subseteq dom(\eta)$ the relation $\mathcal{M} \models_\eta \varphi$ is defined by the usual induction.

Homomorphism. A *homomorphism* $h : \mathcal{M} \to \mathcal{N}$ between models $\mathcal{M}$ and $\mathcal{N}$ is an $\mathcal{S}$-sorted family of functions $\{h_A : \mathcal{M}_A \to \mathcal{N}_A \mid A \in \mathcal{S}\}$ satisfying the following conditions (suppressing sort information for readability): (i) $A \leq A'$ implies $h_A = (h_{A'}) \restriction_{\mathcal{M}_A}$ (ii) $h(f^{\mathcal{M}}(a_1, \ldots, a_n)) = f^{\mathcal{N}}(h(a_1), \ldots, h(a_n))$, and (iii) $R^{\mathcal{M}}(h(a_1, \ldots, a_n))$ implies $R^{\mathcal{N}}(h(a_1), \ldots, h(a_n))$.

The Term Model. When the set of relation symbols in $\mathcal{L}$ is empty then the set of ground terms forms the universe of a model for $\mathcal{L}$, the *term algebra* [8]. We may view this as a model for an arbitrary order-sorted signature, as follows.

Fix $\mathcal{L} = (\mathcal{S}, \leq, \Sigma)$. The family $\{\mathcal{T}^{\mathcal{L}}_A \mid A \in \mathcal{S}\}$ of *ground terms* over $\mathcal{L}$ is the $\subseteq$-least family such that (i) $\mathcal{T}^{\mathcal{L}}_A \subseteq \mathcal{T}^{\mathcal{L}}_{A'}$ whenever $A \leq A'$ and (ii) if $f \in \Sigma_{w,A}$ with $w = A_1 \ldots A_n$ and for each i, $t_i \in \mathcal{T}^{\mathcal{L}}_{A_i}$ then $f(t_1, \ldots, t_n) \in \mathcal{T}^{\mathcal{L}}_A$. The ground terms determine a model $\mathcal{T}^{\mathcal{L}}$ of $\mathcal{L}$, the *term model*, by taking the interpretation of each $f \in \Sigma_{\langle A_1 \ldots A_n \rangle, A}$ to be the function taking each tuple $(t_1, \ldots, t_n) \in (\mathcal{T}^{\mathcal{L}}_{A_1} \times \cdots \times \mathcal{T}^{\mathcal{L}}_{A_n})$ to the term $f(t_1, \ldots, t_n)$, and taking the interpretation of each relation symbol to be the empty relation.

Theorem 1. *Suppose $\mathcal{L} = (\mathcal{S}, \leq, \Sigma)$ is a signature such that Σ has no relation symbols. Then for any model $\mathcal{M}$ of $\mathcal{L}$ there is a unique homomorphism from $\mathcal{T}^{\mathcal{L}}$ to $\mathcal{M}$ (i.e., $\mathcal{T}^{\mathcal{L}}$ is initial).*

Proof. Initiality of $\mathcal{T}^{\mathcal{L}}$ in the category of *algebras* was shown by Goguen and Meseguer [8]. Now, given an $\mathcal{L}$-model $\mathcal{M}$, we let $\mathcal{M}'$ be the reduct of $\mathcal{M}$ to the language $\mathcal{L}'$ obtained by removing the relation symbols. So $\mathcal{M}'$ is a $\mathcal{L}'$-algebra so that Goguen and Meseguer's theorem applies. But the unique algebra homomorphism from $\mathcal{T}^{\mathcal{L}}$ to $\mathcal{M}'$ is itself a $\mathcal{L}$-homomorphism from $\mathcal{T}^{\mathcal{L}}$ to $\mathcal{M}$, simply because each $\mathcal{T}^{\mathcal{L}}$-relation is empty, and the result follows.

5 Skolemization

A formula is in *negation-normal* form if the negation sign is applied only to atomic formulas. As for standard one-sorted logic, DeMorgan's laws for pushing negations below $\wedge$ and $\vee$, and the equivalences between $\neg\exists x^A\alpha$ and $\forall x^A\neg\alpha$ all hold, even in the presence of empty sorts. So every formula is logically equivalent to a formula in negation normal form. But the fact that models can have empty sorts changes the rules for how quantifiers may be moved within a formula. In particular the passage between $((\exists x^A\alpha)\vee\beta)$ and $\exists x^A(\alpha\vee\beta)$ (when x is not free in β) does not hold if A can be empty (and of course the dual equivalence involving $\forall$ fails as well) and so we cannot in general percolate quantifiers to the front of a formula. So we cannot restrict our attention to formulas in prenex normal form, but we will always pass to negation-normal form.

Definition 2 (Skolemization). Let φ be a negation-normal form formula over signature $\mathcal{L}=(\mathcal{S},\leq,\Sigma)$; the result of a *Skolemization-step* of φ is any formula φ' that can be obtained as follows. If $\exists x^A.\psi(x^A,x_1^{A_1},\ldots,x_n^{A_n})$ is a subformula occurrence of φ that is not in the scope of an existential quantifier, let f be a function symbol not in Σ, and let φ' be the result of replacing the occurrence of $\exists x^A.\psi(x,x_1,\ldots,x_n)$ by $\psi(f(x_1,\ldots,x_n),x_1,\ldots,x_n)$. Note that φ' is a formula in an expanded signature obtained by adding f to $\Sigma_{\langle A_1,\ldots,A_n\rangle,A}$.

A *Skolemization* of a formula φ is a sentence with no existential quantifiers, obtained from φ by a sequence of such steps.

The following lemma is straightforward.

Lemma 3. *For any σ we have $\sigma_{sk}\models\sigma$.*

In contrast to the classical case we do not have the fact that "σ satisfiable implies σ_{sk} satisfiable." That holds in one-sorted logic because we can always expand a model of σ to properly interpret the Skolem functions and make σ_{sk} true, but this expansion is not always possible in the presence of empty sorts.

Example 4. Let σ be $(\exists x^A\,.\,(x=x)\vee\exists y^B\,.\,(y=y)\,)\wedge(\forall z^A\,.\,(z\neq z))$. Then σ is satisfiable but its Skolemization $((a=a)\vee(b=b))\wedge(\forall z^A\,.\,(z\neq z))$ is not.

The phenomenon in Example 4 is essentially the only thing that can go wrong: models can be expanded to interpret Skolem functions if we do not existentially quantify over empty sorts. This points the way to recovering a weak version of the classical equi-satisfiability result which will be good enough for our present purposes.

Lemma 5. *If σ is satisfiable then there exists a formula $\sigma_\perp$ such that (i) $\sigma_\perp\models\sigma$ and (ii) ${\sigma_\perp}_{sk}$ is satisfiable.*

Proof. Suppose $\mathcal{M}\models\sigma$. The sentence $\sigma_\perp$ is obtained by replacing $\exists x^A\,.\,\alpha$ by $\perp$ precisely when $\mathcal{M}_A=\emptyset$. It is straightforward to see that $\sigma_\perp\models\sigma$. Since in $\sigma_\perp$ there is no existential quantification over sorts empty in $\mathcal{M}$ one can show that there is an expansion $\mathcal{M}^*$ of $\mathcal{M}$ to the signature of ${\sigma_\perp}_{sk}$ such that $\mathcal{M}^*\models{\sigma_\perp}_{sk}$.

6 A Finite Model Theorem for Order-Sorted Logic

Model $\mathcal{M}$ is a *submodel* of model $\mathcal{N}$ if (i) for each sort A, $\mathcal{M}_A \subseteq \mathcal{N}_A$ and (ii) each $f^{\mathcal{M}}$ and $R^{\mathcal{M}}$ are obtained as the restrictions of $f^{\mathcal{N}}$ and $R^{\mathcal{N}}$ to $\mathcal{M}$. Note that we use "submodel" in this strong sense rather than just requiring each $R^{\mathcal{M}}$ to be a subset of $R^{\mathcal{N}}$ (as is done by some authors).

If $X = \{X_A \mid A \in \mathcal{S}\}$ is a family of sets with $X_A \subseteq \mathcal{M}_A$ for each $A \in \mathcal{S}$ then we say that X is closed under a function $g : \mathcal{M}_{A_1} \times \cdots \times \mathcal{M}_{A_n} \to \mathcal{M}_A$ if whenever $(a_1, \ldots, a_n) \in X_{A_1} \times \cdots \times X_{A_n}$ we have $g(a_1, \ldots, a_n) \in X_A$. Note that this is a stronger claim than saying that the single set $\bigcup X$ is closed under g.

Lemma 6. *Let $h : \mathcal{P} \to \mathcal{M}$ be a homomorphism between models of $\mathcal{L} = (\mathcal{S}, \leq, \Sigma)$. There is a unique submodel of $\mathcal{M}$ with universe $\{h_A(\mathcal{P}_A) \mid A \in \mathcal{S}\}$.*

Proof. It is easy to check that the family $\{h_A(\mathcal{P}_A) \mid A \in \mathcal{S}\}$ is closed under the interpretations in $\mathcal{M}$ of the function symbols in Σ. So if we define the interpretations of the relation symbols in Σ to be the restriction of the interpretations in $\mathcal{M}$ the result is a submodel. Since there is no choice in the interpretations of the symbols in Σ once the universe $\{h_A(\mathcal{P}_A) \mid A \in \mathcal{S}\}$ is determined, uniqueness follows.

Next we establish the fundamental fact about preservation of universal sentences under submodel. The proof is a straightforward induction.

Theorem 7. *Let σ be a sentence that is existential-free and in negation-normal form and let $\mathcal{M}'$ be a submodel of $\mathcal{M}$. If $\mathcal{M} \models \sigma$ then $\mathcal{M}' \models \sigma$.*

Definition 8 (The *kernel* of a model). Let $\mathcal{M}$ be a model for the signature $\mathcal{L} = (\mathcal{S}, \leq, \Sigma)$. Let h be the unique homomorphism from $\mathcal{T}^{\mathcal{L}}$ to $\mathcal{M}$ (*c.f.* Theorem 1). The image of h is a submodel of $\mathcal{M}$ by Lemma 6; this is the *kernel* of $\mathcal{M}$.

The crucially important fact for us is that for the kernel $\mathcal{K}$ of $\mathcal{M}$ we have, for each sort A, the cardinality of $\mathcal{K}_A$ is bounded by the cardinality of $\mathcal{T}^{\mathcal{L}}_A$, simply because $\mathcal{K}_A$ is the image of $\mathcal{T}^{\mathcal{L}}_A$ under h.

The kernel and the Skolem hull Recall the classical treatment of Skolemization (see *e.g.*, [4]): given a model $\mathcal{M}$, let $\mathcal{M}^*$ be a model interpreting the Skolem functions that satisfies the Skolem theory (the sentences saying that the Skolem functions witness the truth of the associated existential formula). Then given a subset X of the universe of $\mathcal{M}$, the Skolem hull $\mathcal{H}_{\mathcal{M}}(X)$ is the smallest subset of the universe containing X and closed under the functions and constants of the enriched language; this determines an elementary submodel $\mathcal{H}_{\mathcal{M}}(X)$ of $\mathcal{M}^*$. In particular $\mathcal{H}_{\mathcal{M}}(\emptyset)$ can be viewed as a "minimal" submodel of $\mathcal{M}$.

But in the order-sorted setting, *the kernel of a model is not in general the same as the Skolem hull.* The latter notion, although perfectly sensible in order-sorted logic, does not play the same role of "minimal" submodel as it does in the one-sorted setting. Indeed it is possible for the kernel of a model to be finite while the Skolem hull is infinite.

Example 9. Consider $\mathcal{L} = (\{A, B\}, \emptyset, \Sigma)$ with $a \in \Sigma_{\langle\rangle,A}$ and $f \in \Sigma_{B,B}$ the only vocabulary symbols. Let $\mathcal{M}$ have $\mathcal{M}_A = \{b_0 = a^{\mathcal{M}}\}$, $\mathcal{M}_B = \{b_0, b_1, b_2, \ldots\}$, and $f^{\mathcal{M}}$ map b_i to b_{i+1}. Then the Skolem hull $\mathcal{H}(\emptyset)$ of $\mathcal{M}$ is $\mathcal{M}$ itself. Yet the kernel $\mathcal{K}$ of $\mathcal{M}$ is the model of size 1 with $\mathcal{K}_A = \{b_0\}$, $\mathcal{K}_B = \emptyset$, $f^{\mathcal{K}} = \emptyset$.

Here we present our main theorem.

Theorem 10. *Let σ be an $\mathcal{L}$-sentence whose Skolemization σ_{sk} has signature $\mathcal{L}^*$. Then σ is satisfiable if and only if σ has a model $\mathcal{H}$ such that for each sort A, the cardinality of $\mathcal{H}_A$ is no greater than the cardinality of $\mathcal{T}_A^{\mathcal{L}^*}$.*

Proof. For the non-trivial direction, suppose σ is satisfiable. By Lemma 5 there is an approximation $\sigma_\perp$ of σ such that $\sigma_{\perp sk}$ is satisfiable. Let $\mathcal{L}^{**}$ be the signature for $\sigma_{\perp sk}$; note that $\mathcal{L}^{**}$ is a reduct of $\mathcal{L}^*$ and the sentence $(\sigma_\perp)_{sk}$ is existential-free.

Let $\mathcal{M}$ be a model of $(\sigma_\perp)_{sk}$, and let $\mathcal{H}$ be the kernel of $\mathcal{M}$. Since $(\sigma_\perp)_{sk}$ is existential-free, $\mathcal{H} \models (\sigma_\perp)_{sk}$. Since $\mathcal{H}$ is a kernel we have that for each sort A, the cardinality of $\mathcal{H}_A$ is no greater than the cardinality of $\mathcal{T}_A^{\mathcal{L}^{**}}$, and thus no greater than the cardinality of $\mathcal{T}_A^{\mathcal{L}^*}$. Since $(\sigma_\perp)_{sk} \models \sigma_\perp$ and $\sigma_\perp \models \sigma$, the model $\mathcal{H}$ is the desired model of σ.

Finally we can define precisely the key notion of the paper.

Definition 11. *Order-Sorted Effectively Propositional Logic (OS-EPL)* is the class of sentences σ such that the signature of the Skolemization of σ has a finite term model.

The next section shows how to decide whether a sentence is in OS-EPL and if so, to compute the sizes of the sorts in the term model. Taken together with Theorem 10, this establishes a decision procedure for satisfiability of OS-EPL sentences.

7 Algorithms

Let $\mathcal{L} = (\mathcal{S}, \leq, \Sigma)$ be a signature. We say that sort A is *finitary* in $\mathcal{L}$ if $\mathcal{T}_A^{\mathcal{L}}$ is finite. Our membership algorithm reduces the problem of counting terms to one of asking whether a given context-free grammar yields only a finite number of strings; well-known algorithms solve the latter problem. Intuitively, the grammar captures the ground terms that can be generated from the signature.

Definition 12. Given a signature $\mathcal{L} = (\mathcal{S}, \Sigma, \leq)$ with multiple sorts, we define a grammar $G_{\mathcal{L}}$ as follows. The set of nonterminals is $\mathcal{S} \cup \{A_0\}$, where A_0 is a fresh symbol not in $\mathcal{S}$, the set of terminals is $\bigcup\{\Sigma_{w,S} \mid (w, s) \in \mathcal{S}^* \times \mathcal{S}\}$, and the set of productions comprises:

$$\begin{aligned} A_0 &\to A && \text{for each } A \in \mathcal{S} \\ B &\to f A_1 \ldots A_n && \text{whenever } f \in \Sigma_{\langle A_1 \ldots A_n\rangle, B} \\ B &\to A && \text{whenever } A \leq B \end{aligned}$$

A non-terminal X in a context-free grammar G is said to be *useful* if there exists a derivation $A_0 \Rightarrow^* \alpha X \beta \Rightarrow^* u$ where u is a string of terminals, otherwise X is *useless.* If A is a useful non-terminal and u is a string of terminals we say that A *generates* u if there is a derivation $A \Longrightarrow^* u$.

Lemma 13. *Let A be a sort of $\mathcal{L}$ and let u be a string of terminals over $\bigcup\{\Sigma_{w.S} \mid (w,s) \in \mathcal{S}^* \times \mathcal{S}\}$. Then u is a term in $\mathcal{T}_A^{\mathcal{L}}$ if and only if there is a derivation $A \Rightarrow^* u$ in $G_{\mathcal{L}}$. A sort A is inhabited by a ground term if and only if A is useful in the grammar $G_{\mathcal{L}}$. When A is useful as a sort in $L(G_{\mathcal{L}})$, the set $\mathcal{T}_A^{\mathcal{L}}$ is finite if and only if A generates only finitely many terms in $L(G_{\mathcal{L}})$. In particular the set $\mathcal{T}^{\mathcal{L}}$ is finite if and only if $L(G_{\mathcal{L}})$ is finite.*

Proof. The first claim is easy to check: it holds essentially by the construction of $G_{\mathcal{L}}$. The second claim follows from the first and the facts that the u in question are strings of terminals of $G_{\mathcal{L}}$ and we have $A_0 \Rightarrow A$ for each $A \in \mathcal{S}$.

Theorem 14. *There is an algorithm that, given an order-sorted signature $\mathcal{L}$, determines (uniformly) for each sort A, whether $\mathcal{T}_A^{\mathcal{L}}$ is finite. The algorithm runs in time linear in the total size of $\mathcal{L}$.*

Proof. By Lemma 13, $\mathcal{T}_A^{\mathcal{L}}$ is finite if and only if A generates only finitely many terms in in $L(G_{\mathcal{L}})$. There is a well-known algorithm for testing whether a non-terminal in a context-free grammar generates infinitely many terminal strings: after eliminating useless symbols from the grammar $G_{\mathcal{L}}$, form the graph whose nodes are the inhabited sorts, with an edge from B to A if and only if there is a production in $G_{\mathcal{L}}$ of the form $B \to \alpha\ A\ \beta$, that is, if and only if the set $\Sigma_{\langle A_1 \ldots A \ldots A_n \rangle, B}$ is non-empty or if $A \leq B$. Then a non-terminal A generates infinitely many terminal strings if and only if there is a path from A to a cycle. Since the size of $G_{\mathcal{L}}$ is linear in the size of $\mathcal{L}$, the overall complexity of our algorithm is linear in $\mathcal{L}$.

Example 15. Return to Equation 1 from Section 2.. After Skolemizing we have the signature with $b \in \Sigma_{\langle\rangle, B}$ and $f \in \Sigma_{A,B}$ in addition to those constants in the original signature. It is easy to check that the graph constructed for this signature has edges from the node A_0 to A and to B, and an edge from B to A. This graph is acyclic so we conclude that this class of sentences has the finite model property. On the other hand, if we were to postulate that $B \leq A$ (instead of $A \leq B$) then we cannot deduce the finite model property, since our grammar would have the production $A \to B$ in addition to $B \to A$ and the resulting graph would have a cycle.

Corollary 16. *Membership in OS-EPL is decidable in linear time.*

Proof. Let σ be given, over signature $\mathcal{L}$. We can compute the skolemization σ_{sk} of σ in linear time, and extract the signature $\mathcal{L}^*$ of σ_{sk}. The size of this signature is clearly linear in σ, so by Theorem 14, we can decide whether all sorts of $\mathcal{L}^*$ are finitary in time linear in σ.

Note that in the worst case, Σ may induce a number of terms exponential in its size. Thus we would like to avoid actually generating the terms, and merely count them if we can do so in polynomial time.

Theorem 17. *There is an algorithm that, given a signature $\mathcal{L}$, computes, in time cubic in the size of $\mathcal{L}$, the size of $\mathcal{T}_A^{\mathcal{L}}$ for each finitary sort A (returning "∞" for the non-finitary sorts).*

Space does not permit a full presentation of the algorithm: see [17] for the details. Intuitively, if a sort is finitary, its terms can be of height no greater than the number of functions in Σ. So we construct a table containing the number of terms of each height of each sort, starting with constants and then applying functions. The only complication is that when counting the ways to create a new term of height h using function f, we need to make certain that each has at least one subterm of height *exactly* $h-1$. The algorithm is implemented using dynamic programming, and the cubic bound is straightforward to establish.

Summarizing, we have the following sound and complete procedure for testing satisfiability of OS-EPL sentences. Given sentence σ, compute its Skolemization σ_{sk}; let $\mathcal{L}^*$ be the signature of σ_{sk}. If the term model $\mathcal{T}^{\mathcal{L}^*}$ is finite then we know that if σ is satisfiable then σ has a model whose universe has cardinalities as given in Theorem 10. Since these bounds are computable we can effectively decide satisfiability for such sentences.

Remark 18. The results of the algorithm in Theorem 17 can be useful even if not all sorts are finitary. Fontaine and Gribomont [6] have implemented an instantiation-based algorithm that takes advantage of the information that certain sorts are guaranteed to have finitely many ground terms. Their algorithm does not do a sophisticated test for this condition, in fact it succeeds only if there are no non-constant terms in the sort in question. Our algorithm here is simple yet will allow their methods to be applicable to a wider class of sentences.

8 Future Work

This work suggests two major lines of further inquiry. The first is the exploration of algorithms for working with OS-EPL sentences that are efficient in practice. A natural approach is to leverage insights from existing tools for model-finding and theorem-proving that are currently optimized for the traditional EPL class. The other direction is to pursue a program of classifying fragments of order-sorted logic according to decidability. Abadi *et al.* [1] suggest a taxonomy based on quantifier prefix patterns but, as pointed out in the introduction, prenex-normal form is not available when sorts are allowed to be empty. We propose that a combinatorial analysis of the signature of Skolemizations of sentences is the proper generalization of the analysis of classical quantifier prefix classes.

References

1. Abadi, A., Rabinovich, A., Sagiv, M.: Decidable fragments of many-sorted logic. Journal of Symbolic Computation 45(2), 153–172 (2010)
2. Bernays, P., Schönfinkel, M.: Zum entscheidungsproblem der mathematischen Logik. Mathematische Annalen 99, 342–372 (1928)
3. Börger, E., Grädel, E., Gurevich, Y.: The Classical Decision Problem. Perspectives in Mathematical Logic. Springer (1997)
4. Chang, C.C., Keisler, J.: Model Theory, 3rd edn. Studies in Logic and the Foundations of Mathematics, vol. 73. North-Holland (1990)
5. Claessen, K., Sorensson, N.: New techniques that improve MACE-style finite model finding. In: Proceedings of the CADE-19 Workshop on Model Computation (2003)
6. Fontaine, P., Gribomont, E.P.: Decidability of Invariant Validation for Paramaterized Systems. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 97–112. Springer, Heidelberg (2003)
7. Ge, Y., de Moura, L.: Complete Instantiation for Quantified Formulas in Satisfiabiliby Modulo Theories. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 306–320. Springer, Heidelberg (2009)
8. Goguen, J.A., Meseguer, J.: Order-Sorted Algebra I: Equational Deduction for Multiple Inheritance, Overloading, Exceptions and Partial Operations. Theor. Comput. Sci. 105(2), 217–273 (1992)
9. Harrison, J.: Exploiting sorts in expansion-based proof procedures (unpublished manuscript), `http://www.cl.cam.ac.uk/~jrh13/papers/manysorted.pdf`
10. Hooker, J., Rago, G., Chandru, V., Shrivastava, A.: Partial instantiation methods for inference in first-order logic. J. Automated Reasoning 28(4), 371–396 (2002)
11. Jereslow, R.G.: Computation-oriented reductions of predicate to propositional logic. Decision Support Systems 4, 183–197 (1988)
12. Krishnamurthi, S., Hopkins, P., McCarthy, J., Graunke, P., Pettyjohn, G., Felleisen, M.: Implementation and use of the PLT Scheme web server. Higher-Order and Symbolic Computation 20(4), 431–460 (2007)
13. Lahiri, S.K., Seshia, S.A.: The UCLID Decision Procedure. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 475–478. Springer, Heidelberg (2004)
14. Lewis, H.: Complexity results for classes of quantificational formulas. J. Comp. and Sys. Sci. 21(3), 317–353 (1980)
15. Momtahan, L.: Towards a small model theorem for data independent systems in Alloy. ENTCS 128(6), 37–52 (2005)
16. de Moura, L.M., Bjørner, N.: Deciding Effectively Propositional Logic Using DPLL and Substitution Sets. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 410–425. Springer, Heidelberg (2008)
17. Nelson, T., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: On the finite model property in order-sorted logic. Tech. rep., Worcester Polytechnic Institute (2010), `http://tinyurl.com/osepl-tr-pdf`
18. Oberschelp, A.: Order Sorted Predicate Logic. In: Bläsius, K.H., Rollinger, C.-R., Hedtstück, U. (eds.) Sorts and Types in Artificial Intelligence. LNCS, vol. 418, pp. 1–17. Springer, Heidelberg (1990)
19. Ramsey, F.P.: On a problem in formal logic. Proceedings of the London Mathematical Society 30, 264–286 (1930)
20. Torlak, E., Jackson, D.: Kodkod: A Relational Model Finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer, Heidelberg (2007)

Temporal Logic Model Checking in Alloy

Amirhossein Vakili and Nancy A. Day

Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada, N2L 3G1
{avakili,nday}@uwaterloo.ca

Abstract. The declarative and relational aspects of Alloy make it a desirable language to use for high-level modeling of transition systems. However, currently, these models must be translated to another tool to carry out full temporal logic model checking. In this article, we show how a symbolic representation of the semantics of computational tree logic with fairness constraints (CTLFC) can be written in first-order logic with the transitive closure operator, and therefore described in Alloy. Using this encoding, the question of whether a declarative model of a transition system satisfies a temporal logic formula can be solved using the Alloy Analyzer directly. Also, since a declarative description of a model may actually represent a family of transition systems, we define two distinct model checking questions on this family (existential and universal model checking) and show how these properties can be evaluated in the Alloy Analyzer.

1 Introduction

The process of model-driven engineering [1] promises many benefits for the use of models early in the development process; in general, the earlier that quality models are created, the fewer errors there will be to discover later in the process. A modelling language used early in the design process must be able to handle the lack of details available at this point in the project. Therefore, it must be able to express concepts that are abstract. However, if we wish to provide analysis support for these models to increase their quality and utility, we must be able to express the models precisely. Languages such as Alloy [2], B [3], Z [4], and ASMs [5] have many features to express abstract concepts (e.g., sets, relations, and functions) without sacrificing precision. Abstract models are usually declarative, meaning they are described as a set of constraints and do not necessarily have an operational semantics.

We are interested in the problem of analyzing temporal properties of declarative models. Chang and Jackson added finite relations and functions to a traditional state-based specification of a transition system, and developed a BDD-based model checker that analyzed these models against computational tree logic (CTL) specifications [6]. Del Castillo and Winter provided model checking support for a transition system specified as an Abstract State Machine (ASM) [5], via the translation of a class of ASMs to SMV by restricting

J. Derrick et al. (Eds.): ABZ 2012, LNCS 7316, pp. 150–163, 2012.

the range of functions to finite sets [7]. ProB [8] is a tool for analyzing finite B machines, in particular, simulation and model checking against linear temporal logic (LTL) specifications. Within Alloy, it is fairly straightforward to specify a transition relation and then iterate it to check bounded duration temporal properties [9]. None of these approaches allow us to check a full set of temporal properties against a fully declarative model of a transition system.

Describing the traditional representation of the semantics of a temporal logic with respect to a single transition system and state in first order logic is not possible because of the need for quantification over paths (a second order operator). Thus, using constraint-based first-order solvers for model checking has remained elusive. Immerman and Vardi encoded the semantics of CTL and CTL* in first order logic with transitive closure FO(TC) [10]. Their semantics has the important property that the use of transitive closure replaces the need for quantification over the paths. Our first contribution in this paper is to show that a variant of Immerman and Vardi's encoding can be used to encode CTL with fairness constraints (CTLFC) in the Alloy language. We use this symbolic encoding to create a CTLFC model checker for finite scope declarative models of transition systems directly in the Alloy Analyzer. The model checking problem is turned into a constraint solving problem. Compared to Immerman and Vardi, our encoding is linear in the size of the model, whereas in theirs the encoding requires an exponential increase in the size of the model with respect to the size of the temporal logic formula. We validate the simplicity and utility of our approach through several examples of model checking temporal logic properties of declarative models in the Alloy Analyzer.

All related work described earlier on model checking declarative models has focused on a specification of a single transition relation (possibly with nondeterminism) that uses declaratively constrained relations and functions to describe the system's behavior. In our work, the transition systems are specified in a fully declarative language, therefore, it may be the case that the model describes a family of transition relations. For example, the declarative specification "every state must reach a state that is reachable from itself" specifies more than one transition system even with 2 states: It is therefore possible to

consider multiple questions about how a family of transition relations satisfies a temporal property. In this paper, we consider two questions: 1) Universal model checking: Do all the transition relations in the family defined by the declarative model satisfy the temporal property? 2) Existential model checking: Is there a transition relation in the family defined by the declarative model that satisfies the temporal property?

These questions are important in different scenarios; e.g., the first question is relevant for black-box verification: verifying a system by using the specifications

of its subsystems rather than their implementation details. In this case, a user is interested in checking whether the system satisfies certain properties no matter how the subsystems are implemented. The second question is relevant when details to be added in the future will constrain the transition relation of interest. In this case, a user needs to know whether the abstract model can be extended into a more detailed model that satisfies the property.

Our second contribution in this paper is to show that these two distinct questions can be described as consistency problems in the Alloy Analyzer for finite scope declarative models. We show several examples that demonstrate the relevance of these two questions for abstract modeling.

2 Background

In this section, we provide a brief overview on temporal logic model checking and Alloy.

2.1 Temporal Logic Model Checking

Temporal logic model checking is a decision procedure for checking whether a transition system satisfies a temporal logic specification [11]. A transition system is a finite directed graph with a labeling function that associates a set of propositional variables to each vertex. A vertex represents a state of a system, and the propositional variables that it is labeled with represent the values of the variables in that particular state. An edge between two vertexes represents a transition from one state to another.

Definition 1. ***Transition System:*** *The transition system TS is a five tuple, $TS = (S, S_0, \sigma, P, l)$, where: S is a finite set of states; S_0, the set of initial states, is a non-empty subset of S; σ, the transition relation, is a total binary relation over S; P is a finite set of atomic propositions; l, the labeling function, is a total function from S to the power set of P.*

A computation path starting at s where $s \in S$ is a sequence of states, $s_0 \rightarrow s_1 \rightarrow \ldots$ such that $s_0 = s$ and $\forall i \geq 0 : \ \sigma(s_i, s_{i+1})$.

A specification is a set of temporal logic formulas. A temporal logic, such as CTL or CTLFC [11], has logical connectives for specifying properties over the computation paths of a transition system. Equation 1 represents the grammar for a complete fragment of CTL:

$$\varphi \ ::= \ p \mid \neg\varphi \mid \varphi \vee \varphi \mid EX\varphi \mid EG\varphi \mid \varphi EU \varphi \ , \text{ where } p \in P \tag{1}$$

The satisfiability relation for CTL, $\models$, is used to give meaning to formulas. The notation $TS, s \models \varphi$ denotes that the state s of the transition system TS satisfies the property φ and $TS, s \not\models \varphi$ is used when $TS, s \models \varphi$ does not hold. The relation $\models$ is defined by structural induction on φ:

Definition 2. ***Semantics of CTL***

$$
\begin{array}{ll}
TS, s \models p & \textit{iff } p \in l(s) \\
TS, s \models \neg\varphi & \textit{iff } TS, s \not\models \varphi \\
TS, s \models \varphi \vee \psi & \textit{iff } TS, s \models \varphi \textit{ or } TS, s \models \psi \\
TS, s \models EX\varphi & \textit{iff } \exists s' \in \sigma(s) : TS, s' \models \varphi \\
TS, s \models EG\varphi & \textit{iff there exists a path starting at } s,\ s_0 \rightarrow s_1 \rightarrow \ldots, \textit{ such} \\
 & \quad \textit{that for all } i\textit{'s } TS, s_i \models \varphi. \\
TS, s \models \varphi EU \psi & \textit{iff there exist a } j \textit{ and a path, } s_0 \rightarrow s_1 \rightarrow \ldots, \textit{ starting} \\
 & \quad \textit{at } s \textit{ such that } TS, s_j \models \psi \textit{ and for all } i \textit{ less than } j \\
 & \quad TS, s_i \models \varphi.
\end{array}
$$

The transition system TS satisfies the CTL formula φ, denoted by $TS \models \varphi$, if and only if for all $s_0 \in S_0$ we have $TS, s_0 \models \varphi$.

The syntax of a complete fragment of CTLFC is the same as Equation 1 with the addition of one connective, $E_C G$. In this connective, C is a finite set of formulas, fairness constraints, which is used to define a *fair* computation path. The computation path $s_0 \rightarrow s_1 \rightarrow \ldots$ is fair with respect to $C = \{\psi_1, ..., \psi_n\}$ iff:

$$\forall \psi \in C : \ \{i \mid TS, s_i \models \psi\} \text{ is infinite.}$$

The semantics of CTLFC is same as Definition 2 along with the semantics of $E_C G$:

$$
\begin{array}{l}
TS, s \models E_C G\varphi \textit{ iff there exists a fair computation path starting at } s, \\
\quad s_0 \rightarrow s_1 \rightarrow \ldots, \textit{ such that for all } i\textit{'s } TS, s_i \models \varphi.
\end{array}
$$

If X is a subset of S, then σ_X denotes the transition relation σ when its domain is restricted to X:

$$\sigma_X(s_1, s_2) \ \text{ iff } \ \sigma(s_1, s_2) \ \wedge \ s_1 \in X$$

In this article, $\hat{}$ denotes the transitive closure operator; for example, $\hat{}\sigma_X$ is the transitive closure of the relation σ_X. Notice that $\hat{}\sigma_X$ is $\hat{}(\sigma_X)$ and not $(\hat{}\sigma)_X$; in other words, the bounding operator has higher precedence over the transitive closure operator. Similarly, $*$ denotes the reflexive transitive closure operator.

2.2 Alloy

Alloy is a lightweight declarative relational modeling language that has static type checking [2]. The logic that Alloy provides for modeling is first-order logic with the transitive closure operator. An Alloy model consists of a set of declarations, which specify the sets, relations, and functions in a model, and a set of constraints, which are logical formulas. In general, first-order logic is undecidable; as a result, automatic consistency checking of Alloy models is not possible. The Alloy Analyzer, the main analysis tool for Alloy models, provides finite scope analysis: a user is required to fix the size of the sets in the model to constant numbers and then, the Alloy Analyzer translates the model to a propositional

CNF formula, which is then handed to a SAT solver for consistency checking. By fixing the sizes of the sets in an Alloy model, the Alloy Analyzer evaluates a model for consistency using the `run` command and validity using the `check` command. Figure 1 is a simple Alloy model of a transition system with transition relation `sigma`, and its only valid instance with four states is presented in Figure 2, where the vertexes represent the states, edges represent the transitions, and the labeling function is indicated by labeling the vertexes. In Figure 1, Lines 3-4, `S` is a set, `sigma` and `l` are functions that map each element of `S` to a subset of `S`, and to a subset of `P` respectively. Lines 1-2 are definitions of three sets, `P,p,q`, where `p` and `q` are singleton subsets of `P`. The keyword `abstract` is used to specify that every element of `P` belongs to one of its subsets. The result of this declaration is `P={p,q}`. The `fact` block is used to specify the constraints that need to be satisfied by the entities in this model.

```
1 abstract sig P {}
2 one sig p,q extends P {}
3 sig S { sigma: some S,
4  l: set P}
5 one sig S0 extends S {}
6 fact{ all s1,s2:S |
7  s1->s2 in sigma iff
8  (s1.l !in s2.l or s1=s2)
9  S0.l = P}
```

Fig. 1. A simple transition system in Alloy

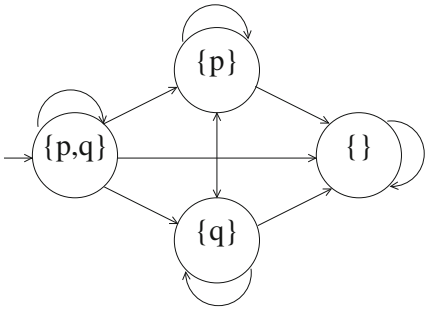


Fig. 2. A valid instance of Figure 1

3 Translating CTLFC to FO(TC)

Immerman and Vardi show how CTL and CTL* can be encoded in FO(TC) [10]. Their encoding of CTL* requires the introduction of Boolean variables into the model for every sub-formula, and as a result, the number of states of a transition system increases exponentially with respect to the size of the formula. They hypothesize that a symbolic model checking algorithm based on their encoding may be faster than previous approaches [11], however, they do not provide any implementation of their idea.

In this section, we present our translation of CTLFC to FO(TC) with a similar approach to that of Immerman and Vardi. We chose CTLFC for three reasons: 1) unlike CTL*, the encoding of CTLFC in FO(TC) does not increase the size of a transition system, 2) it is more expressive than CTL, 3) LTL model checking can be reduced to CTLFC model checking[1] [12].

Our general idea for temporal logic model checking in Alloy is to use the (reflexive) transitive closure operator to specify the necessary and sufficient conditions for the set of states that satisfy a property. The closure operator is used

[1] This translation increases the size of a transition system.

to specify the reachability relation, which is not expressible in first-order logic. We define an operator, [ ], that takes a formula as input and outputs a symbolic representation of the set of states that satisfy the input formula. This operator is defined recursively in Definition 3. The key difference from the work of Immerman and Vardi is that each formula can be defined directly; support for CTL* would require the introduction of a new Boolean variable into the transition system for each sub-formula of the property.

Definition 3. ***Translation Operator:*** *Let $TS = (S, S_0, \sigma, P, l)$ be a transition system and $C = \{\psi_1, \psi_2 \ldots, \psi_n\}$ a set of fairness constraints. The operator [ ] takes a CTLFC formula, and produces a subset of S:*

1. $[p] = \{s \in S |\ p \in l(s)\}$
2. $[\neg\varphi] = \{s \in S |\ s \notin [\varphi]\}$
3. $[\varphi \vee \psi] = [\varphi] \cup [\psi]$
4. $[EX\varphi] = \{s \in S |\ \exists t \in [\varphi] :\ \sigma(s,t)\}$
5. $[\varphi EU \psi] = \{s \in S |\ \exists t \in [\psi] :\ *(\sigma_{[\varphi]})(s,t)\}$
6. $[EG\varphi] = \{s \in S |\ \exists t \in [\varphi] :\ *(\sigma_{[\varphi]})(s,t)\ \wedge\ \hat{}(\sigma_{[\varphi]})(t,t)\}$
7. $[E_C G\varphi] = \{s \in S |\ \exists t \in [\varphi], \exists u_1 \in [\psi_1], \exists u_2 \in [\psi_2], \ldots, \exists u_n \in [\psi_n] :$
   $*(\sigma_{[\varphi]})(s,t) \wedge \hat{}(\sigma_{[\varphi]})(t,t) \wedge$
   $*(\sigma_{[\varphi]})(t,u_1) \wedge *(\sigma_{[\varphi]})(u_1,u_2) \wedge \cdots \wedge *(\sigma_{[\varphi]})(u_{n-1},u_n) \wedge *(\sigma_{[\varphi]})(u_n,t)\}$

Theorem 1. *Let $TS = (S, S_0, \sigma, P, l)$ be a transition system, C a set of fairness constraints, φ a CTLFC formula, and [.] the operator defined in Definition 3. We have:*

$$[\varphi] = \{s \in S |\ TS, s \models \varphi\}$$

Theorem 1 is proven by structural induction on φ. The proof is straightforward for the first six cases. The definition of $[E_C G\varphi]$ is based on the model checking algorithm of $E_C G$ that finds the strongly connected components (SCCs) in a transition system. The state t in the definition of $[E_C G\varphi]$ is a state that belongs to a SCC that includes a state satisfying each fairness constraint ψ_i. Due to space restrictions, the details of the proof of this theorem are available on-line[2]. A simple yet useful corollary of Theorem 1 is the following:

Corollary 1. *Let $TS = (S, S_0, \sigma, P, l)$ be a transition system, C a set of fairness constraints, φ a CTLFC formula, and [ ] the operator defined in Definition 3. We have:*

$$TS \models \varphi \text{ iff } S_0 \subseteq [\varphi]$$

4 Model Checking in Alloy

In this section, we write Definition 3 using Alloy's syntax to create an operator that takes a CTLFC formula, a transition system, a set of fairness constraints and produces the set of states that satisfy the CTLFC formula. The operator,

[2] `http://www.cs.uwaterloo.ca/~avakili/projects/`

$SET(\varphi,TS,C)$, takes a CTLFC formula, φ, a transition system, TS, and a set of fairness constraints, C as input, and produces the subset of states that satisfies the CTLFC formula. The algorithm implemented by this operator visits each sub-formula only once, and as result, it is linear with respect to the size of the CTLFC formula and the fairness constraints. This algorithm is presented in Figure 4, and it uses three helper functions, `bound`, `id`, and `loop`. Each one is described using the equivalence symbol, $\equiv$, and their corresponding Alloy code is given in Figure 3. Intuitively, `bound[R,X]` is a subset of `R` when its domain is restricted to `X`; `id[X]` is the identity relation over `X`; `loop[R]` is a subset of states that are reachable from themselves through `R`.

`bound`$[R,X]\equiv$	`id`$[X]\equiv$	`loop`$[R]\equiv$
$\{(x,y)\in R\mid x\in X\}$	$\{(x,x)\mid x\in X\}$	$\{s\mid (s,s)\in \hat{}R\}$
`fun bound[R:S->S,X:S]`	`fun id[X:S]`	`fun loop[R: S->S]`
`:S->S{ X <: R}`	`:S->S{bound[iden,X]}`	`:S{S.(^R & iden)}`

Fig. 3. Helper functions

$SET(\varphi,TS,C)$:
case φ of
1) p -> $l.p$
2) $\neg\varphi$ -> S - $SET(\varphi,TS,C)$
3) $\varphi\vee\psi$ -> $SET(\varphi,TS,C)$ + $SET(\psi,TS,C)$
4) $EX\varphi$ -> $\sigma.SET(\varphi,TS,C)$
5) $\varphi EU\psi$ -> `(*bound[`σ`,`$SET(\varphi,TS,C)$`])`.$SET(\psi,TS,C)$
6) $EG\varphi$ -> `let R=bound[`σ`,`$SET(\varphi,TS,C)$`] | (*R).loop[R]`
7) $E_CG\varphi$ -> `let R=bound[`σ`,`$SET(\varphi,TS,C)$`],`
`ids1=id[`$SET(\psi_1,TS,C)$`], ..., idsn=id[`$SET(\psi_n,TS,C)$`] |`
`(*R).(loop[R]&loop[(*R).ids1.(*R).ids2.(*R). ... .(*R).idsn.(*R)])`

Fig. 4. Translation algorithm where $C=\{\psi_1,\ldots,\psi_n\}$

According to Corollary 1, in order to check if $TS\models\varphi$ with respect to the fairness constraints C in Alloy, we add the constraint in Equation 2 to the model as an `assertion` and `check` its validity.

$$S_0 \texttt{ in } SET(\varphi,TS,C) \tag{2}$$

Example 1. In order to check whether the Alloy model of Figure 1 satisfies E_CGp, where $C=\{q,\neg q\}$, the Alloy code of Figure 5 is added to the model and the Alloy Analyzer is used to check for the validity of the assertion. In Figure 1, the definition of operator SET has been expanded to create the assertion. Since this property is not satisfied, the Alloy Analyzer outputs `Counterexample found. Assertion is invalid`; similarly, for model checking of the Alloy model of Figure 1 against $pEUq$, the Alloy code of Figure 6 is used, and the Alloy Analyzer outputs `No counterexample found. Assertion may be valid.`

```
1 assert CTLFC_MC_1{
2  let R=bound[sigma,l.p], ids1=id[l.q], ids2=id[S-l.q]|
3   S0 in (*R).(loop[R]&loop[(*R).ids1.(*R).ids2.(*R)])}
4 check CTLFC_MC_1 for exactly 4 S
```

Fig. 5. Model checking $E_C Gp$ where $C = \{q, \neg q\}$

```
1 assert CTLFC_MC_2{let R=bound[sigma,l.p]| S0 in (*R).(l.q)}
2 check CTLFC_MC_2 for exactly 4 S
```

Fig. 6. Model checking $pEUq$

To make model checking in Alloy easy and accessible, we wrote parameterized Alloy modules so that users can import the definitions of the temporal logic operators. The parameter of these modules is the set of states. We have two modules, `ctl` for model checking CTL, and `ctlfc` for model checking CTLFC. Since the number of fairness constraints is not fixed, a user needs to change some parts of the `ctlfc` module, which can be done easily. The universal path quantifiers, `AX, AG, AU, ACG`, have been defined in terms of the existential operators. The following example uses the `ctlfc` module. These module are available on-line[3].

Example 2. Figure 7 is an Alloy model of a binary counter. The `State` space of this transition system is defined by 3 `BIN`ary variables, `input`, `d1`, and `d2`. Lines 3-5 define the set `BIN={ZERO,ONE}` and the negation of a bit `fun`ction, `comp`. This model has two fairness constraints, $C = \{d_1, \neg d_2\}$, which is modeled in Line 7. Lines 8-9 state that if the variables of two states are equal, then those states are equal. Line 10 defines the `initial State`s of the system, and Lines 11-14 define the transition relation, `nextState`. Since we are interested in CTLFC model checking, the `ctlfc` module is imported having the set of `State`s as its parameter, Line 2. Suppose, we want to check that whenever $d1$ is zero, it will eventually become one. By using the temporal connectives of CTLFC from the `ctlfc` module, the property can be written as in Lines 15-16, and the Alloy Analyzer concludes the model checking problem, `CTLFC_MC` is valid.

5 Model Checking Classes of Transition Systems

A satisfiable Alloy model may have more than one valid instance. This is common when constraints that are used to model a system are not strong enough to uniquely identify the transition system; for example, Figure 8 is an Alloy model of a transition system that has more than one valid instance, namely those in Figures 9-10. Each valid interpretation represents a different transition system; as a result, the Alloy model represents a *class* of transition systems rather than a *single* system.

[3] `http://www.cs.uwaterloo.ca/~avakili/projects/`

```
1 module binary_counter
2 open temporal_logics/ctlfc[State]
3 abstract sig BIN{}
4 one sig ZERO, ONE extends BIN {}
5 fun comp[b:one BIN]:one BIN{ b=ZERO implies ONE else ZERO }
6 sig State{ input, d1, d2: BIN }
7 fact { fc1=d1.ONE and fc2=d2.ZERO
8   all s,s':State|s.input=s'.input and s.d1=s'.d1 and
9     s.d2=s'.d2 implies s=s'
10  initialState = (d1.ZERO & d2.ZERO)
11  all s,s':State| s' in nextState[s] iff
12    s.input=ZERO implies (s'.d1=s.d1 and s'.d2=s.d2)
13    else(s'.d1=comp[s.d1] and
14    (s.d1=ZERO implies s'.d2=s.d2 else s'.d2=comp[s.d2]))}
15 assert
16  MC{CTLFC_MC[ACG[implies_ctlfc[d1.ZERO,ACF[d1.ONE]]]]}
17 check MC
```

Fig. 7. Model checking $A_C G(\neg d_1 = 0 \rightarrow A_C F\ d_1 = 1)$, where $C = \{d_1, \neg d_2\}$, for a binary counter

This observation is formalized as follows: the model $\mathfrak{D}$ represents a class of transition systems, $CTS(\mathfrak{D})$:

$$CTS(\mathfrak{D}) = \{TS \mid TS \text{ is a transition system satisfying the constraints in } \mathfrak{D}\} \tag{3}$$

In Equation 3, where the model $\mathfrak{D}$ is considered as a set of transition systems, two questions can be studied: 1) *do all transition systems* in $CTS(\mathfrak{D})$ satisfy the specifications? 2) *is there a transition system* in $CTS(\mathfrak{D})$ that satisfies the specifications? We define two model checking problems for a class of transition systems that correspond to these questions:

Definition 4. ***Universal Model Checking:*** *The universal model checking of the declarative model $\mathfrak{D}$ and the temporal property φ is defined as checking whether all valid instances of $\mathfrak{D}$ satisfy φ:*

$$\mathfrak{D} \textit{ universally satisfies } \varphi \textit{ iff } \quad \forall\, TS \in CTS(\mathfrak{D}) : TS \models \varphi$$

We use $\mathfrak{D} \models_\forall \varphi$ to denote that the declarative model $\mathfrak{D}$ universally satisfies φ.

Definition 5. ***Existential Model Checking:*** *The existential model checking of the declarative model $\mathfrak{D}$ and the temporal property φ is defined as checking whether there exists a valid instance of $\mathfrak{D}$ that satisfies φ:*

$$\mathfrak{D} \textit{ existentially satisfies } \varphi \textit{ iff } \quad \exists\, TS \in CTS(\mathfrak{D}) : TS \models \varphi$$

We use $\mathfrak{D} \models_\exists \varphi$ to denote that the declarative model $\mathfrak{D}$ existentially satisfies φ.

```
1 sig S { sigma: some S,  l: set P}
2 abstract sig P {}
3 one sig p,q extends P {}
4 one sig S0 extends S {}
5 fact{ all s1,s2:S | s1.l !in s2.l implies s1->s2 in sigma
6  S0.l = P}
```

Fig. 8. A simple transition system in Alloy

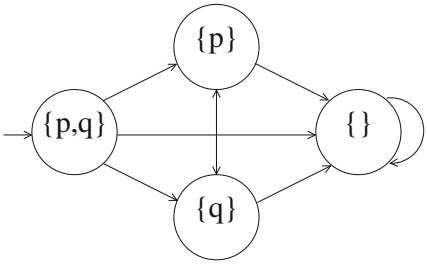


Fig. 9. A valid instance of Figure 8 that does not satisfy EGp

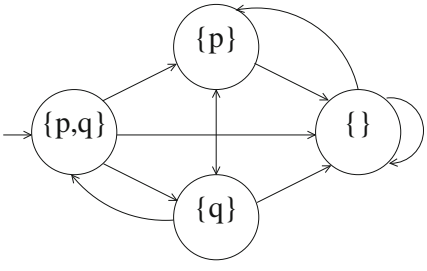


Fig. 10. A valid instance of Figure 8 that satisfies $pEUq$

The model finding capability of the Alloy Analyzer can be exploited to solve the universal and existential model checking. The model checking approach described in Section 4 solves universal model checking: since we add the constraint of Equation 2 as an assertion, the Alloy Analyzer checks whether all valid instances of the model, which in this case are transition systems, satisfy the assertion, which is the CTLFC property. If the model $\mathfrak{D}$ universally satisfies φ ($\mathfrak{D} \models_\forall \varphi$), the Alloy Analyzer outputs `valid`; otherwise, a valid interpretation of $\mathfrak{D}$ such as TS ($TS \in CTS(\mathfrak{D})$) that does not satisfy the constraint of Equation 2 ($TS \not\models \varphi$) is given as a counterexample.

For existential model checking of model $\mathfrak{D}$ against the CTLFC formula φ, the constraint of Equation 2 is added to the model as a `pred`icate and the Alloy Analyzer is used to check for the consistency of the predicate with the model. If the predicate is consistent with the model, the Alloy Analyzer outputs a valid interpretation of $\mathfrak{D}$ such as TS ($TS \in CTS(\mathfrak{D})$) that satisfies the constraint of Equation 2 ($TS \models \varphi$); otherwise, the output of the Alloy Analyzer is `inconsistent predicate`, which means $\mathfrak{D} \not\models_\exists \varphi$.

Example 3. In order to check whether the class of transition systems defined by the Alloy model of Figure 8 universally satisfies EGp, the Alloy code of Figure 11 is added to the model and the Alloy Analyzer is used to check for the validity of the assertion. Since this property is not satisfied, the Alloy Analyzer outputs the instance of Figure 9 as a counterexample; similarly, for existential model checking of the Alloy model of Figure 8 against $pEUq$, the Alloy code of Figure 12 is used, and the Alloy Analyzer outputs the transition system of Figure 10 as a valid instance.

```
1 assert MC1{
2   let R=bound[sigma,l.p]|
3     S0 in (*R).(loop[R])}
4 check MC1 for exactly 4 S
```

Fig. 11. Universal model checking of EGp

```
1 pred MC2[]{
2   let R=bound[sigma,l.p]|
3     S0 in (*R).(l.q)}
4 run MC2 for exactly 4 S
```

Fig. 12. Existential model checking of $pEUq$

6 Experimental Validation

We completed several examples to show that our method makes it possible to check CTLFC temporal logic specifications of declarative models in the Alloy Analyzer, thereby validating the simplicity and utility of our approach. We used four examples from different domains: 1) the semantics of untyped lambda calculus [13], 2) the address book from Jackson [9], 3) feature interaction (FI) between call-waiting and call-forwarding, 4) model checking a traffic light controller [14]. These models satisfy their temporal specifications. Our parameterized Alloy modules for CTL and CTLFC hide the details of model checking in Alloy for a user, so that temporal specifications can be added to models smoothly. These models are available on-line. We used the Alloy Analyzer 4.2 along with the MiniSat SAT-solver [15]. The experiments were run on an Intel Core 2 Due 2.40 GHz machine running Ubuntu 10.04 with up to 3G of user-space memory.

Table 1 presents data on the types of properties, type of model checking (universal/existential), scope size, number of signatures, number of relations, and the Alloy Analyzer time to check the property. With respect to scalability, we found that temporal specifications can be analyzed up to the size of the scopes that non-temporal specifications are often analyzed in Alloy. Thus, our method is immediately valuable to those who use Alloy for modelling and analysis now relying on the *Small Scope Hypothesis [9]*. These models are not as large as those that can be checked using a model checker such as SMV [14], however, the declarative and relational aspects of Alloy have significant advantages for creating abstract, concise models, and we now provide the ability to check temporal logic specifications directly on small scopes of these models.

Furthermore, the untyped λ-calculus example shows the value of the existential model checking question. We used existential model checking to generate a λ-term that does not have a normal form, $(\lambda x.xx)(\lambda x.xx)$, and a term that has a normal form but not necessarily every reduction path terminates, $(\lambda x.(\lambda x.xx))((\lambda x.xx)(\lambda x.xx))$. Since, a result was found for the scope 7, there was no need to do existential model checking for higher scopes. As this example suggests, one way of using existential model checking is to generate interesting instances. In general, existential model checking can help a user to have a better understanding about a declarative model of a transition system by checking the *existence* of specific instances; in other words, existential model checking can be considered as an approach for "simulating" a declarative transition system.

Table 1. Experimental results. MC: Model Checking, NS: Number of Signatures, NR: Number of Relations, SS: Scope Size, min: minute, sec: seconds.

Untyped λ-calculus		Address Book		Feature Interaction		Traffic Light Controller	
NS:6, NR:10		NS:5, NR:3		NS:6, NR:10		NS:13, NR:4	
SS	Time	SS	Time	SS	Time	SS	Time
7	8.22 sec	14	1 min 14 sec	10	14.28 sec	7	4.71 sec
		15	2 min 57 sec	11	2 min 7.6 sec	8	36.81 sec
		16	9 min 15 sec	12	20 min 51 sec	9	12 min 42 sec
		17	13 min 43 sec	13	> 1 hour	10	> 1 hour
Safety, Liveness		Safety		Safety		Safety with fairness	
Existential MC		Universal MC		Universal MC		Universal MC	

7 Related Work

The `ordering` module of Alloy can be used for bounded model checking of safety properties. This approach does not support model checking liveness properties or even safety with fairness constraints. Our approach, which is available as `ctlfc` and `ctl` modules in Alloy, supports much more sophisticated temporal properties.

A declarative relational modeling language for transition systems has been proposed by Chang and Jackson [6]. They augment the traditional languages of model checkers by sets and relations and declarative constructs to specify a transition system. Their technique is not capable of model checking a class of models, and suffers from the state-space explosion problem.

B [3] is a modeling language that has many similarities with Alloy. Models developed in B are called B *machines*, and the variables used to define the state space can be sets and relations. ProB [8] is a tool for analyzing finite B machines, in particular, model checking and automatic refinement checking of B machines. ProB provides LTL model checking. LTL properties are checked by explicit state-space search. Since each single state in a B machine represents some sets and relations, computing the set of the next states of a single state is computationally very costly. ProB also does not provide model checking for a class of transition systems.

The Abstract State Machine (ASM) method [5] is for high-level system design and analysis. The ASM method is used to specify an infinite transition system. Analysis techniques for the ASM method include theorem proving [16, 17], and model checking [7], which consists of translating an ASM to SMV by fixing the size of the scopes in the ASM.

DynAlloy is an extension to Alloy for describing the dynamic properties of systems by using actions [18]. It provides partial correctness analysis of DynAlloy models by using the Alloy Analyzer. Our work is concerned with transition systems and temporal properties.

Modal transition systems (MTSs) are generalized transition systems that are mostly used for verification of complex systems by combining over- and under-approximation for abstraction [19]. In an MTS, a user needs to specify the "must" transitions, those that are part of a system, and the "may" transitions, the ones that may become part of the transition system. Determining "must" transitions requires some analysis of the specification, and discovering how the system must work. Our approach does not need such an analysis and the systems can be completely declarative.

8 Conclusion

We have shown that every CTLFC formula can be encoded in first-order logic plus transitive closure using a similar approach to Immerman and Vardi [10]. Our encoding does not increase the size of the model, and the translation algorithm is linear with respect to the size of the CTLFC formula. We have used this translation to model check transition systems in Alloy by using the constraint solver of the Alloy Analyzer to similar scopes as are used to check non-temporal properties.

When an Alloy model of a transition system has more than one valid instance, it represents a class of transition systems. We have defined two model checking problems concerning a class of transition systems: 1) universal model checking (Definition 4) 2) existential model checking (Definition 5); further, we have used our encoding of CTLFC in Alloy, and the capability of the Alloy Analyzer in valid instance finding to solve the model checking problems that we have defined. The scalability of our approach is dominated by the SAT-solver's capability in solving constraints.

The declarative aspects of Alloy make it a very suitable language for modeling structural aspects of product. We are interested in provided more language support for specifying declarative models of transition systems to help with the readability of these models.

The witness (or counterexample) that is produced for existential (universal) model checking is a transition system; by adding labels for each sub-formula of the specification to states, a user can see why a witness (or counter-example) satisfies (does not satisfy) the specification. Developing a post-processor that takes a transition system generates SMV style counterexamples, which are computation paths, will make our approach more accessible to a non-expert user.

Even though, we do not restrict the length of computation paths in our approach as is done in bounded model checking [20], bounding the signatures of an Alloy model results in bounding the states. Bounding the signatures differently may result in discovering different errors. The relationship between the system and how its signatures are bounded can be studied to make this approach more effective for declarative models.

References

1. Selic, B.: From Model-Driven Development to Model-Driven Engineering. In: ECRTS. IEEE Computer Society (2007)
2. Jackson, D.: Alloy: a lightweight object modelling notation. ACM TOSEM 11(2), 256–290 (2002)
3. Abrial, J.R.: The B Book: Assigning Programs to Meanings. Cambridge University Press (August 1996)
4. International Organisation for Standardization: Information Technology Z Formal Specification Notation Syntax, Type System and Semantics (2000)
5. Börger, E.: The ASM Method for System Design and Analysis. A Tutorial Introduction. In: Gramlich, B. (ed.) FroCos 2005. LNCS (LNAI), vol. 3717, pp. 264–283. Springer, Heidelberg (2005)
6. Chang, F.S.H., Jackson, D.: Symbolic Model Checking of Declarative Relational Models. In: ICSE 2006, pp. 312–320 (May 2006)
7. Del Castillo, G., Winter, K.: Model Checking Support for the ASM High-Level Language. In: Graf, S. (ed.) TACAS 2000. LNCS, vol. 1785, pp. 331–346. Springer, Heidelberg (2000)
8. Leuschel, M., Butler, M.: ProB: A Model Checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 855–874. Springer, Heidelberg (2003)
9. Jackson, D.: Software Abstractions - Logic, Language, and Analysis. MIT Press (2006)
10. Immerman, N., Vardi, M.: Model Checking and Transitive-Closure Logic. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 291–302. Springer, Heidelberg (1997)
11. Clarke, E., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (1999)
12. Clarke, E.M., Grumberg, O., Hamaguchi, K.: Another Look at LTL Model Checking. Formal Methods in System Design 10, 47–71 (1997)
13. Hindley, J.R., Seldin, J.P.: An Introduction to Combinators and the λ-calculus, 2nd edn. Cambridge University Press (2008)
14. McMillan, K.L.: The SMV system (November 06, 1992)
15. Eén, N., Sörensson, N.: An Extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 333–336. Springer, Heidelberg (2004)
16. Schellhorn, G., Ahrendt, W.: Reasoning about Abstract State Machines: The WAM Case Study. Journal of Universal Computer Science 3(4), 377–413 (1997)
17. Dold, A.: A Formal Representation of Abstract State Machines Using PVS. Verifix Technical Report Ulm/6.2, Universität Ulm (July 1998)
18. Frias, M.F., Galeotti, J.P., López Pombo, C.G., Aguirre, N.M.: DynAlloy: Upgrading Alloy with Actions. In: Proceedings of ICSE 2005, pp. 442–451. ACM (2005)
19. Huth, M., Jagadeesan, R., Schmidt, D.A.: Modal Transition Systems: A Foundation for Three-Valued Program Analysis. In: Sands, D. (ed.) ESOP 2001. LNCS, vol. 2028, pp. 155–169. Springer, Heidelberg (2001)
20. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic Model Checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)

Active Attacking Multicast Key Management Protocol Using Alloy

Ting Wang and Dongyao Ji

The State Key Laboratory of Information Security
Graduate University of the Chinese Academy of Sciences
Institute of Information Engineering, Chinese Academy of Sciences,
No.89 Minzhuang Road, Haidian District, Beijing, 100195, P.R. China
whhkwt.0603@163.com

Abstract. In this paper, we use Alloy Analyzer, a fully automatic checker, to detect vulnerabilities in the multicast key management protocol proposed by Tanaka and Sato, and discover some previously unknown attacks. We model an active intruder in Alloy, and use Alloy Analyzer to test whether the active intruder can successfully attack the protocol. In this analysis, we check four critical properties that should be satisfied by any secure multicast protocol. However, none of these properties are satisfied. The protocol cannot resist the active intruder. Two unknown flaws caused by the active intruder are disclosed, and another two flaws found by CORAL are identified.

Keywords: Alloy, Multicast Key Management, Active Intruder, Security Protocol Analysis.

1 Introduction

In this paper, we describe an application of Alloy [5,6] to the analysis of a multicast key management protocol proposed by Tanaka and Sato [2]. Our model of the protocol is constructed in Alloy, which is a modeling method that includes both a modeling language based on first-order logic, and a tool, called Alloy Analyzer and based on model-finding through SAT-solving.

This protocol was first analyzed and improved by Taghdiri and Jackson [1]. However, their analysis omitted the active attack. In an active attack, the attacker, called *active intruder*, can interfere the communication. An active intruder can create, forge, replay, block and reroute messages. This active attack contrasts with a passive attack in which the attacker only eavesdrops, but it does not tamper messages.

Steel and Bundy studied the improved protocol using CORAL [3]. They found the active intruder only having the ability of replaying the network messages between KDS and its members, can find new flaws. But they did not find all active attacks caused by the active intruder.

However, there are three weak aspects of CORAL's performance [3], which debase the quality and time efficiency of the analysis. These disadvantages do

J. Derrick et al. (Eds.): ABZ 2012, LNCS 7316, pp. 164–177, 2012.

not exist in Alloy. One is the difficulty of posing conjectures, i.e. CORAL is hard to describe the property that would be checked. This difficulty may result in the inappropriately described property, and CORAL may be prone to omit flaws. This disadvantage debases the quality of analysis's result. But Alloy can overcome this problem. Alloy's ability of expression is strong and is easy to describe the property in detail.

Another weak aspect is the run time. For example, it is up to 3.5 hours to find the second attack [3]. By comparison, Alloy only uses a few seconds to find this attack, its fast speed should owe to SAT solver. The Alloy Analyzer is bundled with SAT solver. In Alloy, every analysis involves solving a constraint. The Alloy Analyzer is therefore a constraint solver for the Alloy model. In its implementation, it translates the constraint into a boolean formula and solves it using a SAT solver. In the last decade, SAT solver technology has advanced dramatically, and a state-of-the-art SAT solver [8] can often solve a formula containing thousands of boolean variables and millions of clauses.

The third weak aspect is that CORAL is unable to reason about the order in which events took place, and who was in the group at the time [3]. This disadvantage increases difficulty in analyzing counterexample. Alloy can define a **signature** as the notion of time, so the sequence of events is clear.

Due to these disadvantages in CORAL, it is necessary to use another method to analyze this protocol, and Alloy is preferred because Alloy can overcome these disadvantages easily.

On the basis of Taghdiri and Jackson's work, we propose a new model considering the behaviors of active intruder. We analyze the new model with Alloy Analyzer, and checke four critical properties, which should be satisfied by any secure multicast protocol. However, all the four properties are unsatisfied, the studied protocol cannot resist the active intruder. Two unknown flaws caused by the active intruder are disclosed, and another two flaws found by CORAL are identified[3].

The organization of this paper is as follows: Section 2 gives an overview of the improved version of the multicast key management protocol. In section 3, we describe our model of the active intruder. In section 4, we present the analysis of the model and counterexamples. Section 5 summarizes and concludes.

2 Overview of the Improved Tanaka-Sato Protocol

The protocol that Tanaka and Sato originally proposed [2], was improved by Taghdiri and Jackson [1]. The improved version will be described in the following paragraphs.

The group is partitioned into subgroups called **domains**. Each domain under the management of a trusted **key distribution server**(KDS). KDS has information about its domain membership, and KDS is responsible for processing the requests of its domain's members.

The communication between KDSs is assumed to be not only secure but also conducted under a **Reliable and Totally Ordered Multicast Protocol**

(RTOMP). Reliable multicast protocols provide retransmissions and ordering of messages from a source. Totally ordered multicast protocols guarantee that all members receive messages in the same order, ensuring consistency of shared information.

Since there is no notion of delay for the RTOMP in the protocol, we assume that there is no delay in communications via the RTOMP. Hence, if a KDS generates a new group key, all other KDSs will receive it instantly. Videlicet, at each time, all the KDSs know the same set of keys.

When a client wants to join the group, the client and KDS mutually authenticate using an authentication protocol. Having been authenticated and accepted into the group, each member shares with its KDS a key, to be called the member's **individual key**. This key is used when it is needed to send a message that could not be decrypted by others. In general, messages between KDS and a member are encrypted by the member's individual key.

However, some redundancies are caused by Taghdiri and Jackson's improvements [3]. The KDS is unnecessary to send the new group key (we name it k_0) to the new member who joins its domain, because when the member wants to send or accept messages, the member would request its KDS for the newest group key (we name it k_n), the member would use k_n rather than k_0 to encrypt or decrypt messages. So the group key k_0 is useless. Additionally, the key ID number sent in the request for the newest group key is redundant, the key ID number is the identifier of the newest group key the member owned. If the ID of the member's key is older than the newest group key the KDS owned, the KDS will send back all the newer keys. But after the improvements, the KDS only returns the newest group key rather than a set of newer group keys, the newest group key the KDS returned is independent of the ID of the newest key the member owned, so the ID number in the request is useless. We revise the model to remove these redundancies.

The protocol consists of four sub-protocols, described as follows:

- Joining the group
  1. $M_i \longrightarrow \text{KDS} : \{join, M_i\}$
  2. $\text{KDS} \longrightarrow M_i : \{Ik_{M_i}\}$

  When a host M_i wants to join the group, it sends a *join request* (message 1) to one of the KDSs and waits for confirmation. Assume there has an authentication protocol to mutually authenticate M_i and KDS, and both message 1 and message 2 are securely transported under the protocol. If the authentication is successful, M_i will join the corresponding domain, and KDS will generate a fresh individual key, Ik_{M_i}, and a new group key. KDS sends Ik_{M_i} to the new member, i.e. message 2. KDS distributes the new group key to other KDSs via the RTOMP.
- Leaving the group
  1. $M_i \longrightarrow \text{KDS} : \{leave, M_i\}_{Ik_{M_i}}$
  2. $\text{KDS} \longrightarrow M_i : \{ack.leave\}_{Ik_{M_i}}$

  When a member wants to leave the group, it sends a *leave request* (message 1) to its KDS. If the KDS approves the request, it will generate a new group key and distribute the new key to other KDSs, then it sends the confirmation

(message 2) to the member. Both message 1 and message 2 are encrypted by the member's individual key.

– Sending a message
  1. $M_i \longrightarrow \text{KDS} : \{send\}_{Ik_{M_i}}$
  2. $\text{KDS} \longrightarrow M_i : \{send, Gk_n\}_{Ik_{M_i}}$
  3. $M_i \longrightarrow \text{ALL} : \{message\}_{Gk_n}$

  When a member decides to send a message, it sends a request (message 1) to the KDS of its domain for the newest group key. If the member is inside the KDS's domain, the KDS will send back the newest key Gk_n, which carries a unique ID number n. The individual key of the member is used to encrypt the request and the reply. Then, the member uses the newest key to encrypt its message and multicasts the encrypted message.
– Receiving a message
  1. $M_i \longrightarrow \text{KDS} : \{read\}_{Ik_{M_i}}$
  2. $\text{KDS} \longrightarrow M_i : \{read, Gk_n\}_{Ik_{M_i}}$

  When a member receives a message, it sends a request (message 1) to the KDS of its domain and asks for the newest key. The corresponding KDS replies to this request with the newest key Gk_n. Only if the message is encrypted by the newest key returned by KDS, then the member can accept the message.

3 Alloy Model of the Active Intruder

In this section, we will analyze the improved protocol using Alloy. As mentioned in section 1, Taghdiri and Jackson analyzed the original protocol and proposed an improved version of the multicast key management protocol in [1]. However, their analysis overlooked the active intruder. Therefore, some flaws, which cannot be revealed by their model, may still exist in their improved protocol. To analyze their improved protocol, we construct a new model including an active intruder using Alloy.

3.1 Basic Components of the Active Intruder

As mentioned in section 2, each member shares with its KDS an individual key. The signature **Identity** models all the individual keys, and we add field **id** to signature **Member**, shown in Fig.1. **id** is regarded as the member's individual key, which uniquely identifies this member, and other members do not know this key.

Figure 1 shows the basic components of the active intruder.

When a member wants to send a message or read a received message, it will request the newest group key from its KDS. If the member is legitimate, the KDS will return the newest key which is encrypted by the member's individual key. The KDS's responses for sending messages requests are defined as **sig ResponseSend**, and reading messages requests are defined as **sig ResponseRead**. The two signatures are shown in Fig.1. Field **requester** gives the member who requests the newest group key, field **newestkey** gives the newest group key that returned by KDS. Field **encryptingKey** is the member's individual key, and used

```
sig ResponseSend {
  requester:Member,
  replaytime:set Tick,
  newestkey:Key,
  encryptingKey:Identity}
{encryptingKey=requester.id}

sig ResponseRead {
  requester:Member,
  replaytime:set Tick,
  newestkey:Key,
  encryptingKey:Identity}
{encryptingKey=requester.id}

one sig Oscar extends Member {
  learnSend:Tick->ResponseSend,
  learnRead:Tick->ResponseRead}
  {The constraint of this signature...}

sig Identity {}

sig Member {
  kds:KDS,
  ownedKeys:Tick -> Key,
  receivedMessages:Tick -> Message,
  id:Identity,
  responseSend:Tick -> ResponseSend,
  responseRead:Tick -> ResponseRead}
  {The constraint of this signature...}
```

Fig. 1. Basic components of the active intruder

to encrypt KDS's response, so KDS's response is $\{\mathbf{newestkey}\}_{\mathbf{encryptingKey}}$. Field **replaytime** gives the moments when the KDS's response is replayed.

Signature **Oscar** is treated as an active intruder. Comparing with the normal members, it owns the ability to eavesdrop, intercept and replay messages. Assume Oscar can distinguish the responses between signature **ResponseSend** and signature **ResponseRead**, and Oscar knows which member the response would be sent to. Fields **learnSend** and **learnRead** gives the set of responses that Oscar has eavesdropped at a given time. Adding two fields **responseSend** and **responseRead** to signature **Member**, the two fields give the responses that sent to this member, and these responses have been eavesdropped by Oscar at given time.

3.2 Main Operations of the Active Intruder

The main operations of the active intruder are shown in Fig.2.

The function **OscarNewestKey** explains how Oscar intercepts the responses from KDS to legitimate members and replays the previously captured responses. Argument **flag** determines the type of **m**'s request. If **flag=1** holds, **m**'s request for the newest group key is to multicast a message. If **flag**'s value is 2, **m**'s request for the newest group key is to read a received message. If **OscarCanReplay** holds, **Oscar** can carry out replay attack. **Oscar** replaces the KDS's response with the appropriate response, which misleads **m** into taking an old group key as the newest key. In the rest of the cases, Oscar would not disturb the communication between KDS and its members.

As mentioned in section 2, there are some redundancies in Taghdiri-Jackson improved version model, one of these redundancies is the key ID number sent in the request for the newest group key, because after the improvements, the KDS only returns the newest group key rather than a set of newer group keys, the newest group key the KDS returned is independent of the newest key the member owned. In our model, these redundancies are removed.

The predicate **replayAttack** is used to record the replay time. If **Oscar** replays the latest eavesdropped response at time **t**, **t** will be added to the field **replaytime** of the latest eavesdropped response. The latest eavesdropped response is returned by function **NewestResponseSend** or **NewestResponseRead**.

The predicate **GeneratedResponse** is used to create a new eavesdropped response which is added to the set of eavesdropped responses. Argument **flag** determines the type of response is **ResponseSend** or **ResponseRead**. Taking **ResponseSend** for example, the value of **flag** is 1. Member **m** cannot replay old response at time **t** if **!OscarCanReplay[m,t,1]** holds, then **m** will receive its KDS's response **msg**, it is generated by **GeneratedResponseSend[t]** and the assignments of its fields. **Oscar** eavesdrops **msg** at time **t**, and **msg** is added to **Oscar**'s set of eavesdropped responses for **m**'s requests, and it is added to **Oscar**'s set of all eavesdropped responses, too.

In Taghdiri-Jackson model [1], predicate **SendMessage** constrains progress of multicasting a message, and predicates **ReceiveMessage** and **CanReceive** constrains the progress of reading a received message. In these three predicates, function **NewerKeys** is used as KDS's responses for its members's requests for the newest group key. In our model, function **NewerKeys** is replaced by function **OscarNewestKey**, and the following expression (1), (2) and (3) in Fig. 3 are respectively added to predicates **SendMessage**, **ReceiveMessage** and **CanReceive**.

Taking predicate **SendMessage** for example, if **m** wants to multicast a message, it will ask its KDS for the newest group key, either it receives its KDS's response, or Oscar's replay response. If **m** receives the former, Oscar will eavesdrop this response, explained by predicate **GeneratedResponse[m,t,1]**. If **m** receives the latter, Oscar will carry out a replay attack, explained by predicate **replayAttack[m,t,1]**.

```
fun OscarNewestKey(m:Member,t:Tick,flag:Int):Key{
  (flag=1 && OscarCanReplay[m,t,1])=>
  NewestResponseSend[m,t].newestkey&(m.kds).keys[t]
  else (flag=2 && OscarCanReplay[m,t,2])=>
  NewestResponseRead[m,t].newestkey&(m.kds).keys[t]
  else NewerKeys[m,m.kds,t]
}

pred replayAttack(m:Member,t:Tick,flag:Int){
  (flag=1 &&
   OscarCanReplay[m,t,1] &&
   t in NewestResponseSend[m,t].replaytime) ||
  (flag=2 &&
   OscarCanReplay[m,t,2] &&
   t in NewestResponseRead[m,t].replaytime)
}

pred GeneratedResponse(m:Member,t:Tick,flag:Int){
  (flag=1 &&
   !OscarCanReplay[m,t,1] &&
   let msg=GeneratedResponseSend[t]|{
    msg.requester=m
    msg.newestkey=OscarNewestKey[m,t,1]
    msg.encryptingKey=m.id
    m.responseSend[t]=
      m.responseSend[ord/prev[t]]+ msg
    Oscar.learnSend[t]=
      Oscar.learnSend[ord/prev[t]]+msg})||
  (flag=2 &&
   !OscarCanReplay[m,t,2] &&
   let msg1=GeneratedResponseRead[t]|{
    msg1.requester=m
    msg1.newestkey=OscarNewestKey[m,t,2]
    msg1.encryptingKey=m.id
    m.responseRead[t]=
      m.responseRead[ord/prev[t]]+msg1
    Oscar.learnRead[t]=
      Oscar.learnRead[ord/prev[t]]+msg1})
}
```

Fig. 2. Main operations of the active intruder

```
GeneratedResponse[m,t,1]||replayAttack[m,t,1]                    (1)
GeneratedResponse[m,t,2]||replayAttack[m,t,2]                    (2)
OscarCanReplay[m,t,2]=> t in NewestResponseRead[m,t].replaytime  (3)
```

Fig. 3. Eavesdrop or Replay

4 Analysis of the Model

We implement the model in the Alloy Analyzer 4.1.10 [7]. The Alloy Analyzer is used to automatically check some properties of the model, i.e. **InsiderCanSend**, **InsiderCanRead**, **OutsiderCantRead** and **OutsiderCantSend**. Any one of the four properties is unsatisfied, then either the basic ability of legitimate members cannot be guaranteed, or the confidentiality of messages is violated. For example, a group contain n legitimate members, $m_1, m_2, \cdots, m_n$. If the active intruder utilizes the attack in 4.1 to attack m_1, other legitimate members will cannot read the messages sent by m_1. m_1 can be treated as a "mute". If m_1 is under the attack in 4.2, m_1 will cannot read the messages sent form other legitimate members. m_1 can be considered as a "deaf person". m_1 is a legitimate member, it should own the basic ability of sending and reading messages. But the two attacks make m_1 lose this basic ability. If the attack in 4.3 is carried out, some legitimate members will read the counterfeit messages sent from the attacker. If the active intruder carries out the attack in 4.4, the attacker will read the confidential message which violates the confidentiality. Therefore, the safety of the four properties are significant.

4.1 Counterexample of Assertion InsiderCanSend

InsiderCanSend: This assertion claims that messages sent by an insider of the group can be accepted by other insiders of the group.

For this assertion, Oscar only eavesdrops and replays the responses for sending message requests. The Alloy Analyzer finds one type of counterexample represented as a figure. There are many relation lines in this figure. To make the figure concise, we delete some inessential lines and merge some lines, the abbreviated one is Fig. 4. For example, the abbreviated figure uses $t1$ and $t2$ to replace *Tick*1 and *Tick*2, respectively. Additionally, the previous graph's four lines that $members[t1]$, *memb-ers*$[t2]$, $members[t3]$ and $members[t4]$ are merged into one line $members[t1-t4]$ in Fig. 4. In Fig. 4, all lines are the type: $A \xrightarrow{C} B$, it means C is a field of **sig** A, and B is C's value.

In the counterexample, there is one KDS, four members respectively named *Member*0, *Member*1, *Member*2 and *Oscar*, two messages that are *Message*0 and *Message*1, seven ticks of time and their seriation is t0, t1, t2, t3, t4, t5 and t6, two group keys *Key*0 and *Key*1, and one *ResponseSend* is significant. The KDS of the four members are both *KDS*. We detailedly explain the Fig. 4 as follows:

- $KDS \xrightarrow{members[t1-t3]} Oscar$: it shows that Oscar is in the group from t1 to t3, what it means is Oscar joins at t1 then leaves at t4.
  Lines $KDS \xrightarrow{members[t1-t6]} Member0$, $KDS \xrightarrow{members[t1-t6]} Member1$ and $KDS \xrightarrow{members[t4-t6]} Member2$ can be explained in the same way.
- $KDS \xrightarrow{keys[t1-t6]} Key0$: KDS obtains Key0 from t1 to t6, and Key0 is created at t1. Line $KDS \xrightarrow{keys[t4-t6]} Key1$ can be explained in the same way.

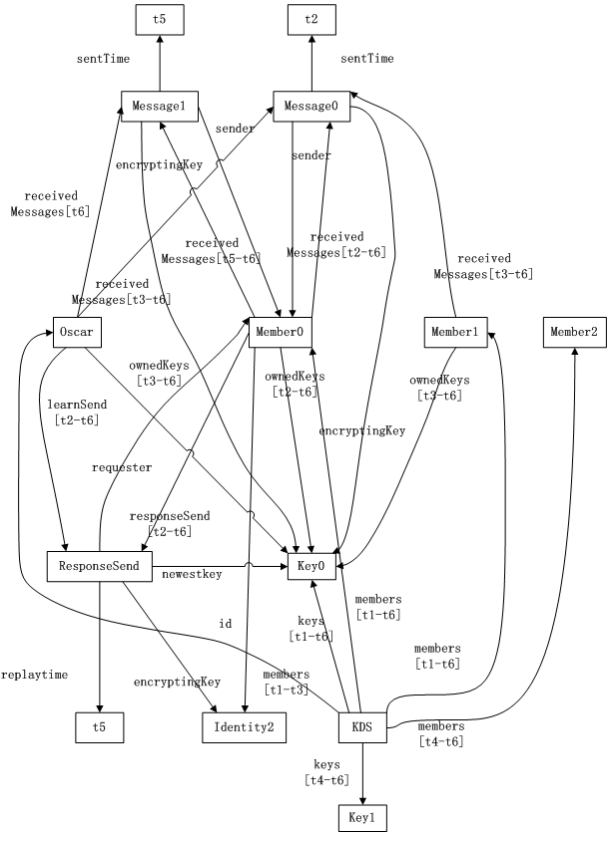


Fig. 4. The abbreviated figure of InsiderCanSend's counterexample

- $Oscar \xrightarrow{ownedKeys[t3-t6]} Key0$: Oscar owns Key0 from t3 to t6. The constrains of signature **Member** restricts that a member gets a new group key at some time must because it wants to send or read a message, then asks its KDS for the newest group key at that time. So Oscar sends or read a message at t3. Lines $Member0 \xrightarrow{ownedKeys[t2-t6]} Key0$ and $Member1 \xrightarrow{ownedKeys[t3-t6]} Key0$ can be explained in the same way.
- $Member0 \xrightarrow{id} Identity2$: Member0's individual key is Identity2.
- $Message0 \xrightarrow{sender} Member0$: Message0's sender is Member0.
  Line $Message1 \xrightarrow{sender} Member0$ has the similar meaning.
- $Message0 \xrightarrow{sentTime} t2$: departure time of Message0 is t2.
  Line $Message1 \xrightarrow{sentTime} t5$ has the similar meaning.

- $Message0 \xrightarrow{encryptingKey} Key0$: encrypting key of Message0 is Key0. Line $Message1 \xrightarrow{encryptingKey} Key0$ has the similar meaning.
- $Oscar \xrightarrow{receivedMessages[t3-t6]} Message0$: Oscar accepts Message0 at t3. Lines $Oscar \xrightarrow{receivedMessages[t6]} Message1, Member0 \xrightarrow{receivedMessages[t2-t6]} Message0, Member0 \xrightarrow{receivedMessages[t5-t6]} Message1$ and $Member1 \xrightarrow{receivedMessages[t3-t6]} Message0$ can be explained in the same way.
- $Oscar \xrightarrow{learnSend[t2-t6]} ResponseSend$: Oscar eavesdrops ResponseSend at t2.
- $Member0 \xrightarrow{responseSend[t2-t6]} ResponseSend$: Oscar eavesdrops ResponseSend at t2, and this response responds Member0's request.
- $ResponseSend \xrightarrow{requester} Member0$: ResponseSend is Member0's KDS responds its request for the newest group key, and this response is encrypted by Member0's individual key.
- $ResponseSend \xrightarrow{replaytime} t5$: this response is replayed at t5.
- $ResponseSend \xrightarrow{newestkey} Key0$: Key0 is the newest group key in this response returned by KDS.
- $ResponseSend \xrightarrow{encryptingKey} Identity2$: encrypting key of ResponseSend is Identity2.

From the graph of this counterexample and the above-mentioned interpretation of Fig. 4, we can get the event sequence of this counterexample, shown in Fig. 5.

The interpretation of the event sequence shown in Fig. 5 is as follows. Oscar, M_0 and M_1 joined the group at the same time t1, and KDS created only one group key k_0. At t2, M_0 wanted to send message, then it asked KDS for the newest group key k_0, and it sent message0 encrypted by k_0. At the same time, Oscar eavesdropped KDS's reply, i.e. message 8. Oscar and M_1 read message0 at t3. M_2 joined the group and Oscar left the group at t4, and another group key k_1 was created. At t5, M_0 wanted to send message, again. After M_0 sent a request to KDS and waited for KDS's reply, Oscar intercepted KDS's reply and replayed message 8 to M_0, in message 21. The replay message misled M_0 into believing the old key k_0 was the newest group key, then M_0 used k_0 to encrypted message1. M_1 and M_2 received message1 at t6, then they asked KDS for the newest group key and received k_1. However, message1 was encrypted by k_0, so k_1 cannot be used to decrypt message1, then M_1 and M_2 cannot read message1. But M_1 and M_2 were inside the group at t5 when message1 was sent and they were still in the group after t5. This scenario violates the assertion **InsiderCanSend**. Oscar had owned k_0, therefore, even Oscar was outside of the group, it can read message1.

This counterexample shows that Oscar's replay attack can cause that messages sent by a legitimate insider cannot be read by other insiders.

1.	t1:	Oscar	$\longrightarrow$	KDS	: $\{join, Oscar\}$
2.	t1:	KDS	$\longrightarrow$	Oscar	: $\{Ik_{Oscar}\}$
3.	t1:	M_0	$\longrightarrow$	KDS	: $\{join, M_0\}$
4.	t1:	KDS	$\longrightarrow$	M_0	: $\{Ik_{M_0}\}$
5.	t1:	M_1	$\longrightarrow$	KDS	: $\{join, M_1\}$
6.	t1:	KDS	$\longrightarrow$	M_1	: $\{Ik_{M_1}\}$
7.	t2:	M_0	$\longrightarrow$	KDS	: $\{send\}_{Ik_{M_0}}$
8.	t2:	KDS	$\longrightarrow$	M_0	: $\{send, k_0\}_{Ik_{M_0}}$
9.	t2:	M_0	$\longrightarrow$	ALL	: $\{message0\}_{k_0}$
10.	t3:	M_1	$\longrightarrow$	KDS	: $\{read\}_{Ik_{M_1}}$
11.	t3:	KDS	$\longrightarrow$	M_1	: $\{read, k_0\}_{Ik_{M_1}}$
12.	t3:	M_1			: $\{read \quad message0\}$
13.	t3:	Oscar	$\longrightarrow$	KDS	: $\{read\}_{Ik_{Oscar}}$
14.	t3:	KDS	$\longrightarrow$	Oscar	: $\{read, k_0\}_{Ik_{Oscar}}$
15.	t3:	Oscar			: $\{read \quad message0\}$
16.	t4:	M_2	$\longrightarrow$	KDS	: $\{join, M_2\}$
17.	t4:	KDS	$\longrightarrow$	M_2	: $\{Ik_{M_2}\}$
18.	t4:	Oscar	$\longrightarrow$	KDS	: $\{leave, Oscar\}_{Ik_{Oscar}}$
19.	t4:	KDS	$\longrightarrow$	Oscar	: $\{ack.leave\}_{Ik_{Oscar}}$
20.	t5:	M_0	$\longrightarrow$	KDS	: $\{send\}_{Ik_{M_0}}$
8(21).	t5:	Oscar(KDS)	$\longrightarrow$	M_0	: $\{send, k_0\}_{Ik_{M_0}}$
22.	t5:	M_0	$\longrightarrow$	ALL	: $\{message1\}_{k_0}$
23.	t6:	M_1	$\longrightarrow$	KDS	: $\{read\}_{Ik_{M_1}}$
24.	t6:	KDS	$\longrightarrow$	M_1	: $\{read, k_1\}_{Ik_{M_1}}$
25.	t6:	M_2	$\longrightarrow$	KDS	: $\{read\}_{Ik_{M_2}}$
26.	t6:	KDS	$\longrightarrow$	M_2	: $\{read, k_1\}_{Ik_{M_2}}$
27.	t6:	Oscar			: $\{read \quad message1\}$

Fig. 5. Event sequence of InsiderCanSend's counterexample

4.2 Counterexample of Assertion InsiderCanRead

InsiderCanRead: This assertion claims that any current member of the group is able to decrypt messages sent from an insider of the group.

As mentioned in section 1 that Taghdiri and Jackson proposed an improved version of this protocol, but there still exits a flaw, late messages loss [1], violates **InsiderCanRead**. Other than this flaw, the replay attack caused another new drawback which also violates **InsiderCanRead**.

For this assertion, Oscar only eavesdrops and replays the responses for reading message requests. The Alloy Analyzer finds one new type of counterexample. From the figure of this counterexample, we can get the event sequence of this counterexample, shown in Fig. 6. M_2 was in the group at t5 when message1 was sent, and still in the group at t6 when M_2 received the encrypted message1, but M_2 cannot read message1, which violates the assertion **InsiderCanRead**.

In this counterexample, there did not have any members joined or left the group from t5 to t6, so the newest group key was not changed during this period. Therefore, there would not lose any late messages, and message1 unreadable to

$$
\begin{array}{rlcll}
1. & \text{t1:} & M_0 & \longrightarrow \text{KDS} & : \{join, M_0\} \\
2. & \text{t1:} & \text{KDS} & \longrightarrow M_0 & : \{Ik_{M_0}\} \\
3. & \text{t1:} & M_2 & \longrightarrow \text{KDS} & : \{join, M_2\} \\
4. & \text{t1:} & \text{KDS} & \longrightarrow M_2 & : \{Ik_{M_2}\} \\
5. & \text{t2:} & M_0 & \longrightarrow \text{KDS} & : \{send\}_{Ik_{M_0}} \\
6. & \text{t2:} & \text{KDS} & \longrightarrow M_0 & : \{send, k_0\}_{Ik_{M_0}} \\
7. & \text{t2:} & M_0 & \longrightarrow \text{ALL} & : \{message0\}_{k_0} \\
8. & \text{t3:} & M_2 & \longrightarrow \text{KDS} & : \{read\}_{Ik_{M_2}} \\
9. & \text{t3:} & \text{KDS} & \longrightarrow M_2 & : \{read, k_0\}_{Ik_{M_2}} \\
10. & \text{t3:} & M_2 & & : \{read \quad message0\} \\
11. & \text{t4:} & M_1 & \longrightarrow \text{KDS} & : \{join, M_1\} \\
12. & \text{t4:} & \text{KDS} & \longrightarrow M_1 & : \{Ik_{M_1}\} \\
13. & \text{t5:} & M_0 & \longrightarrow \text{KDS} & : \{send\}_{Ik_{M_0}} \\
14. & \text{t5:} & \text{KDS} & \longrightarrow M_0 & : \{send, k_1\}_{Ik_{M_0}} \\
15. & \text{t5:} & M_0 & \longrightarrow \text{ALL} & : \{message1\}_{k_1} \\
16. & \text{t6:} & M_1 & \longrightarrow \text{KDS} & : \{read\}_{Ik_{M_1}} \\
17. & \text{t6:} & \text{KDS} & \longrightarrow M_1 & : \{read, k_1\}_{Ik_{M_1}} \\
18. & \text{t6:} & M_1 & & : \{read \quad message1\} \\
19. & \text{t6:} & M_2 & \longrightarrow \text{KDS} & : \{read\}_{Ik_{M_2}} \\
9(20). & \text{t6:} & \text{Oscar(KDS)} & \longrightarrow M_2 & : \{read, k_0\}_{Ik_{M_2}}
\end{array}
$$

Fig. 6. Event sequence of InsiderCanRead's counterexample

M_2 is not caused by the late message loss. In fact, M_2 cannot read message1 is because of the replay attack that Oscar replayed the old response (message 9) at t6 when M_2 asked for the newest group key. So, this counterexample is a new vulnerability different from late message loss.

4.3 Counterexample of Assertion OutsiderCantSend

OutsiderCantSend: This assertion implies that insiders of the group cannot read messages sent from an outsiders of the group.

A type of counterexamples is found by Alloy Analyzer. The event sequence of this counterexample is in Fig. 7. At t5, Oscar sent message1 which was encrypted by k_0. At the next time, M received message1, then M asked KDS for the newest key, and KDS would return k_1, but Oscar intercepted KDS's response (message 15) and replayed message 9. So, M treated k_0 as the newest key, and accepted message1 as a valid message.

4.4 Counterexample of Assertion OutsiderCantRead

OutsiderCantRead: This assertion implies that no outsider of the group is able to read a message sent by an insider of the group.

From the counterexample analyzed in section 4.1, Oscar was outside of the group at t5 when message1 was sent and still outside of the group at t6, but

1. t1: M $\longrightarrow$ KDS : $\{join, M\}$
2. t1: KDS $\longrightarrow$ M : $\{Ik_M\}$
3. t1: Oscar $\longrightarrow$ KDS : $\{join, Oscar\}$
4. t1: KDS $\longrightarrow$ Oscar : $\{Ik_{Oscar}\}$
5. t2: Oscar $\longrightarrow$ KDS : $\{send\}_{Ik_{Oscar}}$
6. t2: KDS $\longrightarrow$ Oscar : $\{send, k_0\}_{Ik_{Oscar}}$
7. t2: Oscar $\longrightarrow$ ALL : $\{message0\}_{k_0}$
8. t3: M $\longrightarrow$ KDS : $\{read\}_{Ik_M}$
9. t3: KDS $\longrightarrow$ M : $\{read, k_0\}_{Ik_M}$
10. t3: M : $\{read \quad message0\}$
11. t4: Oscar $\longrightarrow$ KDS : $\{leave, Oscar\}_{Ik_{Oscar}}$
12. t4: KDS $\longrightarrow$ Oscar : $\{ack.leave\}_{Ik_{Oscar}}$
13. t5: Oscar $\longrightarrow$ ALL : $\{message1\}_{k_0}$
14. t6: M $\longrightarrow$ KDS : $\{read\}_{Ik_M}$

9(15). t6: Oscar(KDS) $\longrightarrow$ M : $\{read, k_0\}_{Ik_M}$

16. t6: M : $\{read \quad message1\}$

Fig. 7. Event sequence of OutsiderCantSend's counterexample

it read message1 at t6. It violates the assertion **OutsiderCantRead**. So counterexample in Fig. 4 is also a counterexample of assertion **OutsiderCantRead**.

4.5 Result

In Table 1 we sum up the results obtained by the different tools studying the protocol.

We found active attacks on every properties. However, Steel and Bundy only found attacks on the last two properties, and Taghdiri and Jackson did not find these attacks. We are the only one who found the active attacks on the first two properties.

Even though Steel and Bundy also found the last two attacks, but the running times were too long, up to 3.5 hours to find the attack on property *Outsider-CantSend* [3]. By comparison, our model's time efficiency is very high. If the scopes of check commands are appropriate, for example, "for 7 but 1 KDS,3 Member,2 ResponseSend,2 ResponseRead", our model will spend only 1 second or 2 seconds to search for these counterexamples.

Table 1. The result of comparing the three models

	Our model		Steel-Bundy		Taghdiri-Jackson	
	Attack	time	Attack	time	Attack	time
InsiderCanSend	yes	less than 2s	no	-	no	-
InsiderCanRead	yes	less than 1s	no	-	no	-
OutsiderCantSend	yes	less than 1s	yes	3.5h	no	-
OutsiderCantRead	yes	less than 2s	yes	unclear	no	-

5 Conclusion

In this paper, we presented how we used Alloy to model an active intruder and Alloy Analyzer to analyze the protocol's new model. Even though the active intruder in our model only had the capability of eavesdrop, intercept and replay messages, it successfully attacked the protocol. We found four flaws on this protocol. Two flaws were previously unknown, another two were found by Steel and Bundy, their checker is CORAL. But CORAL's performance was weak as mentioned in section 1, these disadvantages debase the quality and time efficiency of the analysis. However, CORAL's disadvantages did not exist in Alloy. Especially the time efficiency, we compared it in Table 1.

Even though Taghdiri and Jackson had analyzed this protocol using Alloy, they omitted the active attack. They did not find the four flaws caused by the active intruder. And there were some redundancies in their improved model, we removed them to make our model more concise.

Acknowledgments. This research was funded by grant 90604010 from the National Nature Science Foundation and grant 2007BC311202 of the National Key Foundation Research Plan of China.

References

1. Taghdiri, M., Jackson, D.: A Lightweight Formal Analysis of a Multicast Key Management Scheme. In: König, H., Heiner, M., Wolisz, A. (eds.) FORTE 2003. LNCS, vol. 2767, pp. 240–256. Springer, Heidelberg (2003)
2. Tanaka, S., Sato, F.: A key distribution and rekeying framework with totally ordered multicast protocols. In: Proceedings of the 15th International Conference on Information Networking, pp. 831–838 (2001)
3. Steel, G., Bundy, A.: Attacking Group Multicast Key Management Protocols Using Coral. Electr. Notes Theor. Comput. Sci., 125–144 (2005)
4. Dolev, D., Yao, A.C.: On the security of public key protocols. IEEE Transactions on Information Theory 29(2), 198–208 (1983)
5. Jackson, D.: Automating first-order relational logic. In: Proceedings of the 8th ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 130–139 (2000)
6. Jackson, D.: Software Abstractions - Logic, Language, and Analysis. The MIT Press (2006)
7. Alloy Analyzer 4, `http://alloy.mit.edu/alloy4/`
8. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: DAC: Proceedings of the 38th annual Design Automation Conference, pp. 530–535. ACM, New York (2001)

Formalizing Hybrid Systems with Event-B

Jean-Raymond Abrial[1], Wen Su[2], and Huibiao Zhu[2]

[1] Marseille, France
jrabrial@neuf.fr
[2] Software Engineering Institute, East China Normal University
{wensu,hbzhu}@sei.ecnu.edu.cn

Abstract. This paper[1] contains the development of hybrid systems in Event-B and the Rodin Platform[2]. It follows the seminal approach introduced at the turn of the century in Action Systems. Many examples illustrate our approach.

1 Introduction

Hybrid systems have been studied for many years ([6] and many more). They are very important in the development of embedded systems where a piece of software, *the controller*, is supposed to manage an external situation, *the environment*. The controller works in a discrete fashion in that it is triggered regularly by detecting the status of the environment (using some *sensors*), and then reacts by sending some information to the environment (using some *actuators*). Between two successive controller detections and actions, the environment is evolving in a *continuous way*.

The formal development of such embedded systems has then to take account of two different frameworks: the discrete framework of the controller and the continuous framework of the environment. The formal development of such *closed systems* must be able to deal with these dual frameworks: this is the purpose of hybrid system.

In this paper, we explain how such systems can be developed in Event-B [7] and the Rodin Platform [8]. The paper is organized as follows: in the next section we explain how our approach follows that developed in Action Systems [1]. Section 3 contains many examples and then we conclude.

2 Approaches

2.1 The Approach of Action System to Hybrid Systems

Background. Action System [1] has been introduced in the eighties by R.J. Back and R. Kurki-Suonio. It has been then further developed by the Finnish

[1] This work is supported in part by National Basic Research Program of China (No. 2011CB302904), National High Technology Research and Development Program of China (No. 2011AA010101 and No. 2012AA011205), National Natural Science Foundation of China (No. 61061130541 and No. 61021004).

[2] An extended version of this paper can be found in [9].

J. Derrick et al. (Eds.): ABZ 2012, LNCS 7316, pp. 178–193, 2012.

School of Formal Methods. Event-B [7] [8] is a direct follower of Action System: many concepts in Event-B have been borrowed from it. As a consequence, before attempting to formalize hybrid systems in Event-B, it seems appropriate to investigate what has been done concerning hybrid systems with Action System.

Continuous Action System. In 2000, R.J. Back and colleagues extended Action System in order to support the definition and proofs of Hybrid Systems [2] [3]. They called this extension *continuous* Action System. In this paper, for the sake of clarity, we shall name the non-continuous Action System the *classical* Action System. This extension is very simple and systematic: "a continuous action system is just a non-deterministic way of defining a collection of time dependent functions". In other words, rather than having the state of the Action System being defined by a collection of *time independent* variables ranging over some sets (as is the case in classical Action System), the state of a continuous Action System is now defined as a collection of *time dependent functions* (where time ranges over the set of non-negative reals $\mathbb{R}^+$). A continuous Action System can be related to a corresponding classical Action System as follows: if $x \in S$ is a variable in a classical Action System, then $x\_c \in \mathbb{R}^+ \rightarrow S$ is the "same" variable in the continuous Action System.

Past, Present, and Future. A "technical" variable, named *now*, ranging over $\mathbb{R}^+$ stands for the "present" time. Initially, *now* is supposed to be equal to 0. The time function x_c in a continuous Action System and its relationship to the corresponding variable x in a classical Action System is to be understood as follows:

1. The time function x_c restricted to the set $\{u \mid u \in \mathbb{R}^+ \wedge u < now\}$ denotes the *past* of the variable x.
2. The value $x\_c(now)$ denotes the present value of x (at time $t = now$) .
3. The time function x_c restricted to the set $\{u \mid u \in \mathbb{R}^+ \wedge u > now\}$ denotes the *future* of x. Of course, we are not sure about this future, it only denotes what we can expect "now" about it.

Discrete Events. As for classical Action Systems, a continuous Action System contains a finite number of *guarded actions*. In each of them, time functions such as x_c can be modified. These modifications however must obey a systematic constraint: the *past cannot be modified*, only the present and future can. Once an action has been "executed", then the variable *now* is updated: this is done by incrementing it to the *smallest value* making at least one action guard becoming true[3]. In between the present *now* and the future one assigned to it, a time function such as x_c is supposed to make progress following the expected future.

Continuous Action Systems as Hybrid Systems. As can be seen, a continuous Action System is indeed a genuine hybrid system: the events correspond to discrete actions situated in the middle of continuous behaviors. Typically, the continuous evolution corresponds to what happens in the external *environment*

[3] If there is no such events then *now* is not updated meaning that the systems evolves *for ever* as prescribed by the future of each time function.

of a system, whereas discrete actions correspond to what a *controller* can do in order to manage the environment. For example, the continuous evolution could be that of a physical train running at a certain speed and a certain acceleration (positive, equal to 0, or negative), whereas the discrete actions are those of the driver (human or automatic) changing the acceleration of the train from time to time depending on the actual speed, the actual acceleration and the actual distance of the train to a necessary stop (at a station or because another train is close to it).

Invariants. As for classical Action Systems, continuous ones must preserve a number of *invariants* to be proved on the past of each time variable. For example, in the train system mentioned above, we might prove that the *past* speed of a train is never greater than a certain maximum speed. We might also prove that no train can hit another one in front of it or not stop at a given station where it should. Such invariants can be stated as follows:

$$\forall t \cdot t < now \;\Rightarrow\; \mathrm{P}(x\_c(t))$$

where P is a predicate denoting the invariant property we want to prove. In order to prove the maintenance of this invariant when *now* is updated to a new value, say *new_now* (greater than *now*), what is to be proved is the following:

$$\forall t \cdot t \geq now \wedge t < new\_now \;\Rightarrow\; \mathrm{P}(x\_c(t))$$

Notice that we do not have to prove the property for $t < now$ since, by definition, the *past is not modified* when updating a time function.

2.2 The Proposed Approach with Event-B

Background. The approach we shall follow with Event-B is very close to that proposed by R.J. Back for continuous Action Systems. However, in studying examples described in [2] and [3], we found a number of difficulties: we had the *subjective feeling* that some proofs of simple invariant properties are more complicated than they should be. In what follows, we propose some simplifications to the original proposals made in continuous Action Systems.

Discrete Variables together with Continuous Variables. In [2] and [3] all variables (except *now*) are time functions. But sometimes these time functions are always constant functions on all considered intervals (different constants however for different intervals). An obvious simplification is to consider that such variables can be better represented as discrete variables as in classical Action Systems. By doing so, we could simplify some of the proofs.

Discrete Systems as an Abstraction of Continuous Ones. We also figured out that the presented approaches for continuous Action System did not take advantage of any refinement steps to be done during the development although this was mentioned in the conclusion as future work in [2].

The main initial steps we propose here before introducing continuous variables is based on our belief that a *discrete system is an abstraction of a continuous*

one. This is a direct consequence of the observation of what is done traditionally in mathematics since the Greeks (and probably before them): in order to measure the surface of a field you cut it into different rectangles (whose surface is easily determined) and then you *refine* this process by introducing more rectangles incorporating some parts of the field that has not been taken into account previously. This operation is then repeated many times until the remaining part of the field that has not been taken into account becomes *very small*.

In the seventeenth century Newton and Leibnitz formalized this by introducing the Calculus. Later, in the nineteenth century, a considerable effort (Cauchy, Weierstrass) has been done to make this mathematical approach completely rigorous.

Refining a Discrete Systems into a Continuous Ones. When formalizing an hybrid system with Event-B, we can start the development by considering *time independent* variables only, such as x, with the basic invariant $x \in S$. This is done together with some events modifying such discrete variables. This process can be done with different refinement steps, thus making the discrete system more precise.

At some point (when the discrete system is rich enough) we can start introducing some continuous *time dependent* variables corresponding to some discrete ones. We also introduce the variable *now*. For example, the variable x is refined (and removed) to the variable $x\_c \in \mathbb{R}^+ \nrightarrow S$. The gluing invariant between x and x_c is clearly the following: $x = x\_c(now)$. The modification of the variable x_c in an event obeys the following pattern:

$$x\_c := \lambda t \cdot t \in now \mathinner{..} new\_now \mid \mathrm{E}(t)$$

where the value new_now corresponds to the modification of the variable *now* which is updated together with x_c (that is, $now := new\_now$). As can be seen, we depart here from what was done in continuous Action System. More precisely, we update the time dependent variable x_c to the new continuous value it takes within the time interval $now \mathinner{..} new\_now$ where no discrete action takes place. Notice that new_now is in fact equal to $min(\{t \mid t \geq now \ \wedge \ \mathrm{P}(t)\})$ where P is a predicate corresponding to the disjunction of the guards of the events within which *now* is replaced by t.

The transformation of the discrete system into a continuous one can be done gradually: we can have intermediate steps where some discrete variables are not yet transformed into continuous ones. Also, as stated above, some discrete variables will so remain because they are constant in all intervals.

3 Examples

3.1 The Saw [2]

Our first example is extracted from [2]. It is a very simple introductory example. The initial state is made of a single discrete variable x taking only two values: 0 and 1.

variables: x

inv0_1: $x \in \{0, 1\}$

The events UP and DN alternatively change the value of x, initialized to 0:

INIT
begin
$x := 0$
end

UP
when
$x = 0$
then
$x := 1$
end

DN
when
$x = 1$
then
$x := 0$
end

In the refinement, we replace the variable x by a time function x_c as follows (notice the gluing invariant **inv1_3**).The variable x_c is initialized to the constant function $\{0 \mapsto 0\}$, while the variable now is initialized to 0 (the beginning of time):

variables: x_c, now

inv1_1: $x\_c \in \mathbb{R}^+ \nrightarrow \mathbb{R}$

inv1_2: $now \in \mathrm{dom}(x\_c)$

inv1_3: $x = x\_c(now)$

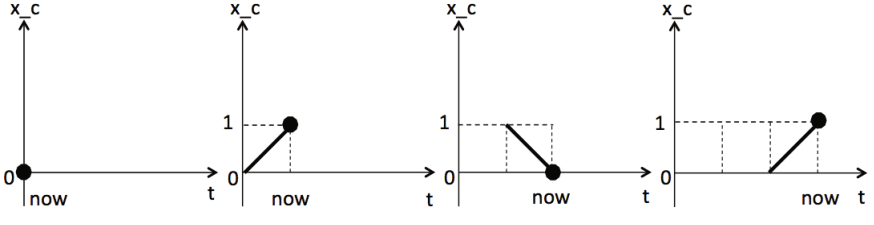
INIT
begin
$x\_c := \{0 \mapsto 0\}$
$now := 0$
end

The events UP and DN are refined as follows (notice the updating of the variable now):

UP
when
$x\_c(now) = 0$
then
$x\_c :=$
$\lambda t \cdot t \in now \,..\, now + 1 \,|\, t - now$
$now := now + 1$
end

DN
when
$x\_c(now) = 1$
then
$x\_c :=$
$\lambda t \cdot t \in now \,..\, now + 1 \,|\, 1 - t + now$
$now := now + 1$
end

The following figures show the evolution of the variable x_c initially and after various executions of the events UP and DN:



We can add several other invariants. For example we might be interested to prove that the range of the variable x_c is included in the real interval $0 \,..\, 1$:

inv1_4:	$\text{ran}(x\_c) \subseteq 0\,..\,1$

The global proof effort for the Rodin Platform [8] on this example is 17 proof obligations, all proved automatically.

3.2 Nuclear Plant Cooling [3]

The nuclear plant cooling example is taken from [3]. Here is an informal description of the problem quoted from [3]:

"The hybrid system is a temperature control system for a heat producing reactor, described by the temperature as a function of time $\theta(t)$. The reactor starts from the initial temperature θ_0 and heats up at a given rate v_r. Whenever it reaches the critical temperature θ_M, it is designed to be cooled down by inserting into the core either of two rods (rod1 or rod2), modeled by the variables $x_1(t)$ and $x_2(t)$, which are in fact clocks measuring the time elapsed between two consecutive insertions of the same rod, respectively. The cooling proceeds at the rate v_1 and v_2 depending on which rod is being used, and the cooling stops when the reactor reaches a given minimum temperature θ_m, by releasing the respective inserted rod. The rod used for cooling is then unavailable for a prescribed time T, after which it is again available for cooling. The object of the modeling is to ascertain that the reactor never reaches the critical temperature θ_M without at least one of the rods available."

In the development done in [3] the main safety proof mentioned at the end of the previous informal description (i.e. "the reactor never reaches the critical temperature θ_M without at least one of the rods available") is done directly on the time dependent variables and results in a rather heavy proof. As for the previous example, we start the Event-B development with a discrete system. We do the safety proof at this level: this results in a simple proof as expected. We then refine the system in several steps to a genuine hybrid system. We first start by defining a number of constants as introduced in the previous informal explanation: θ_m, θ_M, v_1, v_2, v_r, and T. We then define the heating time, a, needed to raise the temperature from θ_m to θ_M in the reactor and also the cooling times, b_1 and b_2, needed to decrease the temperature from θ_M to θ_m with rod1 or rod2. We suppose the following constraints on a, $b1$, $b2$, and T:

constants:	a	**axm0_1:**	$av_r = \theta_M - \theta_m$	**axm0_4:**	$2a + b_1 \geq T$
	b_1	**axm0_2:**	$b_1 v_1 = \theta_M - \theta_m$	**axm0_5:**	$2a + b_2 \geq T$
	b_2	**axm0_3:**	$b_2 v_2 = \theta_M - \theta_m$	**axm0_6:**	$a < T$

Notice axiom **axm0_6**: should it be $a \geq T$ then the cooling with a rod would always be possible because then the time, a, of temperature increasing in the reactor would be greater than or equal to the the time, T, after which a rod would be made available after being used. Axioms **axm0_4** and **axm0_5** seem a bit strange: they can be discovered as sufficient conditions while doing the proofs.

The initial model is a discrete one. We define some state variables: θ is the temperature of the reactor, $t1$ and $t2$ denote the time elapsed since rod1 or rod2 have been released. The system works with a *phase* variable (with values 0, 1, and 2). When *phase* is 0, it means that the reactor has reached the maximum temperature θ_M (invariant **inv0_5**). When *phase* is 1 or 2, it means that the reactor has reached the minimum temperature θ_m by being cooled either with rod1 or with rod2 (invariant **inv0_6**):

variables: θ, t_1, t_2, *phase*

inv0_1: $\theta \in \mathbb{R}^+$
inv0_2: $t_1 \in \mathbb{R}^+$
inv0_3: $t_2 \in \mathbb{R}^+$
inv0_4: $phase \in \{0, 1, 2\}$
inv0_5: $phase = 0 \Rightarrow \theta = \theta_M$
inv0_6: $phase \in \{1, 2\} \Rightarrow \theta = \theta_m$

The main safety invariant is the following. It states that there is always one rod available when the reactor's temperature reaches the maximum temperature θ_M:

inv0_7: $phase = 0 \Rightarrow t_1 \geq T \vee t_2 \geq T$

In this initial model, besides the initializing event, we have four events: cool_rod1, cool_rod2, release_rod1, and release_rod2. In order to simplify, we suppose that we start when the reactor has reached the maximum temperature θ_M. Here are some events (the remaining ones for rod2 are similar):

INIT
begin
$t_1 := T$
$t_2 := a$
$\theta := \theta_M$
$phase := 0$
end

cool_rod1
when
$phase = 0$
$t_1 \geq T$
then
$phase := 1$
$t_2 := t_2 + b1$
$\theta := \theta_m$
end

release_rod1
when
$phase = 1$
then
$phase := 0$
$t_1 := a$
$t_2 := t_2 + a$
$\theta := \theta_M$
end

Remember that $t1$ and $t2$ denote the time elapsed since rod1 or rod2 have been released. In event cool_rod1, the temperature goes down from θ_M to θ_m with time $b1$ by the use of rod1, so the time t2 related to rod2 is updated accordingly. In event release_rod1, which happens just after the release of rod1, the temperature is raised up to temperature θ_M with time a, thus both $t1$ and $t2$ are updated accordingly. The proof of the invariants (in particular that of the safety invariant **inv0_7**) are very simple (some additional "technical" invariants are needed).

The refinement of this discrete system into a continuous one is simple routine. In the sequel, we show how the two discrete variables t_1 and t_2 are refined to time dependent variables t_1_c and t_2_c.

variables: t_1_c, t_2_c, $phase$

inv1_1: $t_1\_c \in \mathbb{R}^+ \rightarrow\!\!\!\!\!\!\rightarrow \mathbb{R}$
inv1_2: $t_2\_c \in \mathbb{R}^+ \rightarrow\!\!\!\!\!\!\rightarrow \mathbb{R}$
inv1_3: $now \in \mathrm{dom}(t1\_c) \cap \mathrm{dom}(t2\_c)$
inv1_4: $t_1 = t_1\_c(now)$
inv1_5: $t_2 = t_2\_c(now)$

Here are the corresponding refined events:

cool_rod1
when
$phase = 0$
$t_1\_c(now) \geq T$
then
$phase := 1$
$t_2\_c := \lambda\, t \cdot t \in now \,..\, now + b1 \mid t_2\_c(now) + t - now$
$t_1\_c := \lambda\, t \cdot t \in now \,..\, now + b1 \mid t_1\_c(now)$
$\theta := \theta_m$
$now := now + b1$
end

release_rod1
when
$phase = 1$
then
$phase := 0$
$t_1\_c := \lambda\, t \cdot t \in now \,..\, now + a \mid t - now$
$t_2\_c := \lambda\, t \cdot t \in now \,..\, now + a \mid t_2\_c(now) + t - now$
$\theta := \theta_M$
$now := now + a$
end

Further refinements (not shown here) deal with making the variable θ continuous. The global proof effort for the Rodin Platform on this example is 157 proof obligations, all proved automatically.

3.3 Controlling Trains [4]

Our new example comes from the book of A. Platzer [4]. It involves one (or several) trains evolving on a single line. The goal of this system is to provide safe moves of the trains.

Preliminary Study. Each train is regularly made aware by a radio broadcasting (RBC) of a certain point situated at position m, on the line, where it should at the latest stop. In other words, the train shall never pass this point. Given the position z of the train, our main invariant is clearly the following: $m - z \geq 0$. Every ϵ second, the train controller (a piece of software) examines the situation concerning the position z, speed v, and acceleration a of the train. The controller can change the acceleration as follows: it can order a constant positive acceleration A (where A is positive), a constant negative acceleration $-b$ (where b is positive), or no acceleration. Notice that when the train has a negative acceleration $-b$ it should at last stop when $v = 0$ and thus never get a negative speed. If the train is at position z with speed v at time 0, it will circulate with speed $v + at$ at time t and its position will then be $z + vt + \frac{at^2}{2}$. In order to guarantee that the train will not pass the position m, one should be certain that the negative acceleration $-b$ will be sufficient to stop the train before the position m. Since the train should stop, we have $v + at = 0$ (with a = -b), that is $t = \frac{v}{b}$.This gives

us the following position at $z+\frac{v^2}{2b}$. This quantity should be smaller than or equal to m, that is $z+\frac{v^2}{2b} \leq m$. Here is thus our final constraints:

$$2b(m-z) \geq v^2 \tag{1}$$

It can also be said that breaking with deceleration $-b$ is able to "absorb" the kinetic energy of the train (this will give us the same result in a shorter way). This is a necessary invariant of the system. We notice that it implies the previous invariant $m-z \geq 0$. At each control time (every other ϵ seconds), the controller must ensure that the train will preserve this invariant in the next control position (in ϵ second). The speed of the train will be $v+a\epsilon$ and the position of the train will be $z+v\epsilon+\frac{a\epsilon^2}{2}$. Therefore substituting these values for v and z in (1) yields the following, $2b(m-z-v\epsilon-\frac{a\epsilon^2}{2}) \geq (v+a\epsilon)^2$, that is:

$$2b(m-z) \geq v^2 + (a\epsilon^2+2v\epsilon)(a+b) \tag{2}$$

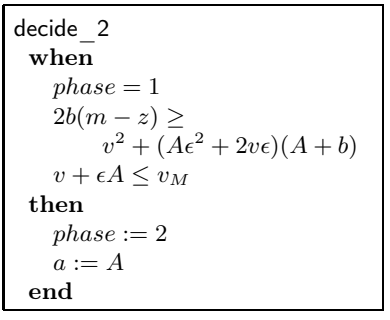
For the controller to decide that the acceleration a could be A for the next ϵ seconds, we must have: $2b(m-z) \geq v^2 + (A\epsilon^2+2v\epsilon)(A+b)$. If this is not the case (i.e. if the previous predicate is false), the controller must decide that the acceleration a must be $-b$. This decision is indeed safe since then $a+b=0$ in (2) and we already have the following invariant, $2b(m-z) \geq v^2$. At the end of the process, when the speed v is equal to 0 and when the train cannot proceed further, we have: $m-z < \frac{A\epsilon^2(A+b)}{2b}$. The final specification of this train controlling process has now become very clear. It can be stated informally as follows: move the train by accelerating or decelerating it until it reaches the speed 0 and get to a position z such that $m-z < \frac{A\epsilon^2(A+b)}{2b}$ holds. Notice that it is quite possible for the train to reach a position where its speed is 0 but with $m-z \geq \frac{A\epsilon^2(A+b)}{2b}$: it means that the train stops and restart immediately because it can move further.

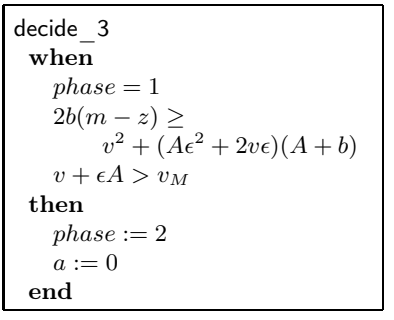
Event-B Development. The previous preliminary elementary calculations dictate the way things can be implemented with Event-B and the Rodin Platform. We first define the four constants A, b, ϵ, and m: they are all positive real numbers. The dynamic state of the system introduces variables z, v, and a (position, speed, and acceleration of the train). We also introduce a technical variable *phase* that can be 1 or 2. In phase 1, the controller will decide what to do, whereas in phase 2, the train will makes some progress or stop. The main invariants are the following (**inv1_1** to **inv1_5**).The first decision event decide_1, decelerates the train as below:

inv1_1: $z \in \mathbb{R}^+$
inv1_2: $v \in \mathbb{R}^+$
inv1_3: $a \in \{0, A, -b\}$
inv1_4: $phase \in \{1, 2\}$
inv1_5: $2b(m-z) \geq v^2$

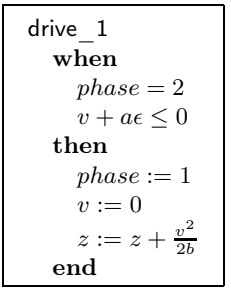
decide_1
when
$phase = 1$
$2b(m-z) < v^2 + (A\epsilon^2+2v\epsilon)(A+b)$
then
$phase := 2$
$a := -b$
end

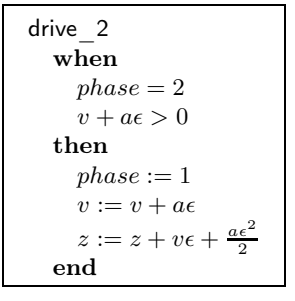
The second decision event, decide_2, accelerates the train. But at this point we introduce another constraint, namely that there is an upper constant speed limit v_M. We have thus two more decision events:





The first driving event, drive_1, stops the train before the end of ϵ seconds since otherwise the speed would become negative. The second driving event, drive_2, continues the progression of the train.





We can now refine this model by adding a second train. Nothing changes for the first train that still gets its limit point to be m. The second train with position z_2, speed v_2, and acceleration a_2 will now get its limit being z (the position of the first train) rather than m. This refinement is very easily done with Event-B and the Rodin Platform. After this second refinement, an interesting animation can be performed with the AnimB animator of the Rodin Platform: one can see the two trains accelerating and decelerating in an appropriate way. The global proof effort for the Rodin Platform on this example is 103 proof obligations, all proved automatically except two of them that are proved interactively (easy). It would be simple to refine the time independent variables z and v to time dependent variables as we have done in previous examples.

3.4 Aircraft Collision Avoidance [4] [5]

This example is taken from the book of A. Platzer [4]. It has also been developed in an independent paper by Platzer and Clarke [5]. The problem is to study an *horizontal* collision avoidance maneuver to be performed by two aircrafts flying at the same altitude. This maneuver must be performed when the two aircrafts have the possibility to "almost" collide, i.e. when the distance between them

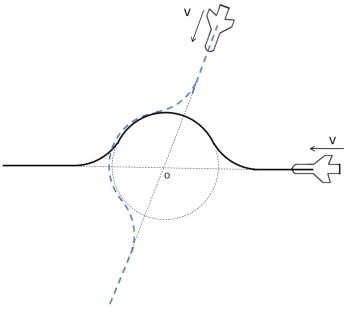
could become smaller than or equal to a predefined constant distance p. The maneuver is said to be "horizontal" as both aircrafts continue to fly at the same altitude before, during, and after executing the maneuver.

Simplified Case: Preliminary Study

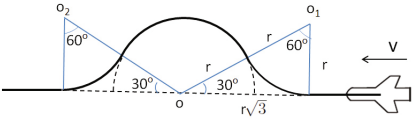
Platzer and Clarke studied a simple case with the following constraints:

1. Both aircrafts have the same linear speed v which has to be maintained during the maneuver.
2. Both aircrafts would *exactly collide* at some point o should they continue to fly without performing the maneuver.
3. Both aircrafts are situated at the same distance of the colliding point o when they decide to maneuver

The maneuver consists for both aircrafts to reach a certain circle centered in o and with a radius r (to be made precise later). Once they have reached this circle, both aircrafts follow it in the same direction until they both leave it at the same time in order to eventually return to their original direction. All this can be illustrated in the following figure:



Entering and leaving the circle as well as following the circle is always done at the same original linear speed v. In order to ensure this, both aircrafts should enter and leave the circle by using portions of external circles (called the entering circles) that are tangent to the main one and with the same radius. As a consequence, both aircrafts start following the entering circles when they are both at a distance $r\sqrt{3}$ of the circle center o. Likewise, they enter the main circle when the angle with their original trajectory is exactly $\frac{\pi}{6}$. This can be illustrated in the following figure:

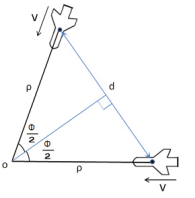


During the maneuver the distance between the two aircrafts must always be smaller than the predefined distance p. If the angle between the two trajectories

is ϕ and the common distance of both aircrafts to the circle center o is ρ then the distance d of both aircrafts is the following:

$$d = 2\rho \sin \frac{\phi}{2}$$

This is illustrated in the right figure.



The main invariant of our system is thus the following:

$$2\rho \sin \frac{\phi}{2} \geq p$$

Notice that the angle ϕ between the two trajectories will not change during the maneuver. As a consequence, the only quantity that counts in order to compute the distance and thus check whether the safety condition holds is the common distance ρ of both aircrafts to the center o. The smallest distance between the aircrafts is reached when they are both flying on the main circle (this will be formally proved below). We must have then $2r \sin \frac{\phi}{2} \geq p$. This gives us a *lower value* for r: $r \geq \frac{p}{2 \sin \frac{\phi}{2}}$. If both aircrafts decides to maneuver when they are at a distance ρ_i from the point o, this distance must be greater than the distance where they start turning (i.e. $r\sqrt{3}$). This gives us an *upper value* for r. We must have $r\sqrt{3} \leq \rho_i$. We have thus the following constraint for r (that is, r can be chosen non-deterministically between these two values): $\frac{p}{2 \sin \frac{\phi}{2}} \leq r \leq \frac{\rho_i}{\sqrt{3}}$.

Notice that if ρ_i is too small, the maneuver is impossible: it is too late. In fact, we must have the following relationship between the three constants ρ_i, p, and ϕ: $\rho_i \geq \frac{p\sqrt{3}}{2 \sin \frac{\phi}{2}}$

Simplified Case: Event-B Development. In this development, we model the behavior of one aircraft only. We can do so since in this simplified framework the behavior of the second aircraft can be deduced from that of the first one by a simple rotation with constant angle ϕ. Here are first a number of constant definitions: ρ_i, p and ϕ. They are all real numbers constrained as follows:

axm0_1: $2\rho_i \sin \frac{\phi}{2} \geq p\sqrt{3}$

The state of the first model is defined by means of the following variables: *phase*, ρ, θ, and r. Variables ρ and θ are the polar coordinates of the first aircraft. But as mentioned above, only the ρ polar cordinate is useful in order to compute the distance d between both aircrafts. As a consequence, we can discard the θ polar coordinate. The following invariants must hold between these variables. Mind the invariant **inv0_4** stating the main safety property, i.e. the distance between aircrafts is always greater than or equal to the constant p. These variables are initialized as follows and the initial agreement makes a non-deterministic choice for r:

inv0_1: $phase \in \{0, 1, 2, 3, 4, 5\}$
inv0_2: $\rho \in \mathbb{R}^+$
inv0_3: $r \in \mathbb{R}^+$
inv0_4: $2\rho \sin \frac{\phi}{2} \geq p$

INIT
begin
$\rho := \rho_i$
$phase := 0$
$r :\in \mathbb{R}^+$
end

agree
any c **where**
$phase = 0$
$p \leq 2c \sin \frac{\phi}{2}$
$c\sqrt{3} \leq \rho_i$
then
$phase := 1$
$r := c$
end

The next phases are described in the following events:

start
when
$phase = 1$
then
$phase := 2$
$\rho := r\sqrt{3}$
end

enter
when
$phase = 2$
then
$phase := 3$
$\rho := r$
end

cycle
when
$phase = 3$
then
$phase := 4$
$\rho := r$
end

leave
when
$phase = 4$
then
$phase := 5$
$\rho := r\sqrt{3}$
end

We are now going to refine this initial model by introducing the continuous time function for ρ, that is ρ_c. We also introduce the variable *now*. Our main invariant is **inv1_4** is to be compared to **inv0_4** where the variable ρ corresponded to a discrete transition.

inv1_1: $\rho\_c \in \mathbb{R}^+ \nrightarrow \mathbb{R}$
inv1_2: $now \in \mathrm{dom}(\rho\_c)$
inv1_3: $\rho = \rho\_c(now)$
inv1_4: $\forall t \cdot t \in \mathrm{dom}(\rho\_c) \Rightarrow 2\rho\_c(t) \sin \frac{\phi}{2} \geq p$

INIT
begin
$\rho\_c := \{0 \mapsto \rho_i\}$
$phase := 0$
$r :\in \mathbb{R}^+$
$now := 0$
end

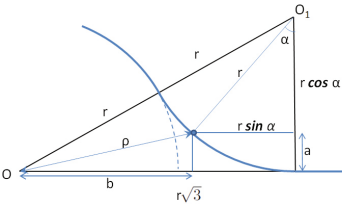
agree
any c **where**
$phase = 0$
$p \leq 2c \sin \frac{\phi}{2}$
$c\sqrt{3} \leq \rho_i$
then
$phase := 1$
$r := c$
end

The event INIT is refined in a simple way and the event agree is not modified. In the event start, the time function ρ_c is decreased from ρ_i in a linear fashion according to the constant speed v of the aircraft. The variable *now* is incremented with a time corresponding to the linear movement of the aircraft from the initial position at ρ_i to the position at $r\sqrt{3}$ where it starts turning.

start
when
$phase = 1$
then
$phase := 2$
$\rho\_c := \lambda t \cdot t \in now \,..\, now + \frac{(\rho_i - r\sqrt{3})}{v} \mid \rho_i - v(t - now)$
$now := now + \frac{(\rho_i - r\sqrt{3})}{v}$
end

The aircraft travels on the external circle centered in o_1 as shown in this figure:

$$\begin{aligned}\rho^2 &= a^2 + b^2\\ &= r^2(1-\cos\alpha)^2 + r^2(\sqrt{3}-\sin\alpha)^2\\ &= r^2(5-4\cos(\tfrac{\pi}{3}-\alpha))\\ \rho &= r\sqrt{5-4\cos(\tfrac{\pi}{3}-\alpha)}\end{aligned}$$



The part of this circle that is used corresponds to an angle of $\frac{\pi}{3}$. Since the aircraft still flies at the same linear speed v, the time it takes to turn in this circle is $\frac{\frac{\pi}{3}r}{v}$: this is therefore the quantity used to increment the variable *now* in the event enter. The computation of the new function ρ_c is a little more complicated. It is explained above (see the figure). We can notice that the quantity $\sqrt{5-4\cos(\frac{\pi}{3}-\alpha)}$ is well defined since $5-4\cos(\frac{\pi}{3}-\alpha)$ is always positive. Also, this quantity is greater than or equal to 1 (when α varies from 0 to $\frac{\pi}{3}$), so ρ is greater than or equal to r. Now the angle α can be related to the time $t - now$, which is the time elapsed on the circle to progress from the angle 0 to the angle α at linear speed v, that is: $\alpha = \frac{v(t-now)}{r}$

enter
when
$phase = 2$
then
$phase := 3$
$\rho\_c := \lambda\, t \cdot t \in now \,..\, now + \frac{\pi r}{3v} \mid r\sqrt{5-4\cos(\frac{\pi}{3} - \frac{v(t-now)}{r})}$
$now := now + \frac{\pi r}{3v}$
end

The last two phases correspond to cycling on the main circle and then leaving the circle: they are not shown here. The global proof effort for the Rodin Platform on this example is 84 proof obligations, all proved automatically.

The General Case. The general case is not very different from the simplified one. In what follows, we give a short account on this generalization. We still suppose that both aircrafts are flying *at the same linear speed* v. They are converging to a point o as in the simplified case, but this time they are not necessarily colliding at this point, but their distance might become smaller than or equal to the predefined distance p. The first thing to do is to determine the distance between both aircrafts and the way this distance evolves. The distance δ can be calculated as follows:

$$\delta^2 = \rho_2^2 \sin^2\phi + (\rho_2\cos\phi - \rho_1)^2 = \rho_1^2 + \rho_2^2 - 2\rho_1\rho_2\cos\phi$$

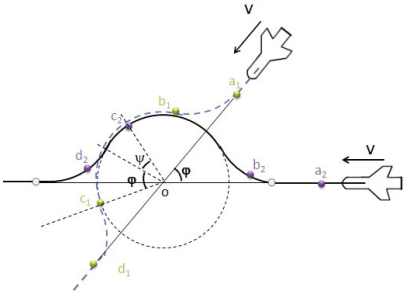
As ρ_1 and ρ_2 are moving from their initial values ρ_{i1} and ρ_{i2} at the same speed v, we have thus: $\rho_1 = \rho_{i1} - vt$ and $\rho_2 = \rho_{i2} - vt$ that is:

$$\delta^2 = (\rho_{i1} - vt)^2 + (\rho_{i2} - vt)^2 - 2(\rho_{i1} - vt)(\rho_{i2} - vt)\cos\phi$$

Thus, the derivative $\frac{d\delta^2}{dt}$ of δ^2 relative to t is the following:

$$\frac{d\delta^2}{dt} = 2v(\cos\phi - 1)(\rho_{i1} + \rho_{i2} - 2vt)$$

When t is smaller than $\frac{\rho_{i1}+\rho_{i2}}{2v}$, the derivative is negative. A minimum is thus reached when t is equal to $\frac{\rho_{i1}+\rho_{i2}}{2v}$, leading to the following minimum δ_m (in this case, we have $\rho_{i1} - vt = \frac{\rho_{i1}-\rho_{i2}}{2}$ and $\rho_{i2} - vt = \frac{\rho_{i2}-\rho_{i1}}{2}$): $\delta_m = (\rho_{i2} - \rho_{i1})\cos\frac{\phi}{2}$ (we suppose $\rho_{i2} > \rho_{i1}$) In summary, both aircrafts "almost" collide when the following holds: $\delta_m \leq p$. In order to avoid this "almost" collision, we are using the same maneuver as in the simplified case, namely to have both aircrafts following a trajectory using a circle centered in the point o as indicated in the following figure:



This time however, we have to take account of both aircrafts in order to ensure that their distance δ remains greater than or equal to p. We suppose $\rho_{i2} > \rho_{i1}$. First of all, when both aircrafts decide on the maneuver, their distance δ_i must be greater than p, that is: $\delta_i^2 = \rho_{i1}^2 + \rho_{i2}^2 - 2\rho_{i1}\rho_{i2}\cos\phi \geq p^2$.

We have to determine the radius r of the circle. The constraints are the following. The first aircraft can reach the turn: $r\sqrt{3} \leq \rho_{i1}$. The distance between both aircrafts is greater than or equal to p when they are both flying on the circle: $2r\sin\frac{\phi+\psi}{2} \geq p$ where ψ is the angular distance due to the difference of the initial positions of the aircrafts. More precisely, we have: $\psi = \frac{\rho_{i2}-\rho_{i1}}{r}$.

4 Conclusion

In this paper, we presented a way of studying hybrid systems in Event-B [7] and the Rodin Platform [8]. Our approach follows that of Action System [1] [2] [3]. It is illustrated by means of many examples taken from the literature. All of them have been developed and fully proved with the Rodin Platform. We have not studied the possible definition of the continuous parts by means of differential equations as is usually done in the hybrid system literature: this will be studied in subsequent papers. As the Rodin Platform does not support (mathematical) real numbers yet, our examples, implemented on Rodin, "cheated a bit". So far, for most of the examples, the "cheating" consisted in giving in the Rodin developments some specific integer values to the constants that are normally assigned to some real values. As a consequence, various calculations ended in

integer numbers: we have done so in the saw example (section 3.1), the nuclear plant cooling example (section 3.2), and the train example (section 3.3). In the aircraft collision avoidance example (section 3.4), this simplification could not be done as we were dealing with trigonometric functions and the square root function. So, in this case, we gave some explicit properties of these functions. For instance, $\sqrt{3}$ is left as such. We have done the same for some trigonometric values, and so on. We also sometimes added some real number "axioms" such as $\forall x \cdot x \neq 0 \Rightarrow x * (y/x) = y$ that is not true for integer numbers since "/" is the integer division.

References

1. Back, R.J., Kurki-Suonio, R.: Distributed Cooperation with Action Systems. ACM Transaction on Programming Languages and Systems 10(4), 513–554 (1988)
2. Back, R.-J., Petre, L., Porres, I.: Generalizing Action Systems to Hybrid Systems. In: Joseph, M. (ed.) FTRTFT 2000. LNCS, vol. 1926, p. 202. Springer, Heidelberg (2000)
3. Back, R.J., Cerschi Seceleanu, C., Westerholm, J.: Symbolic Simulation of Hybrid Systems. In: APSEC 2002 (2002)
4. Platzer, A.: Logical Analysis of Hybrid Systems. Springer (2010)
5. Platzer, A., Clarke, E.M.: Formal Verification of Curved Flight Collision Avoidance Maneuvers: A Case Study. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 547–562. Springer, Heidelberg (2009)
6. Alur, R., et al.: The Algorithmic Analysis of Hybrid Systems. Theoretical Computer Science 138, 3–34 (1995)
7. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
8. `http://www.event-b.org`
9. `http://research.sei.ecnu.edu.cn/reports/A_Hybrid_V7.pdf`

SMT Solvers for Rodin*

David Déharbe[1], Pascal Fontaine[2], Yoann Guyot[3], and Laurent Voisin[3]

[1] Universidade Federal do Rio Grande do Norte, Natal, RN, Brazil
david@dimap.ufrn.br
[2] University of Nancy and INRIA, Nancy, France
Pascal.Fontaine@inria.fr
[3] Systerel, France
{yoann.guyot,laurent.voisin}@systerel.fr

Abstract. Formal development in Event-B generally requires the validation of a large number of proof obligations. Some automatic tools exist to automatically discharge a significant part of them, thus augmenting the efficiency of the formal development. We here investigate the use of SMT (Satisfiability Modulo Theories) solvers in addition to the traditional tools, and detail the techniques used for the cooperation between the Rodin platform and SMT solvers.

Our contribution is the definition of two approaches to use SMT solvers, their implementation in a Rodin plug-in, and an experimental evaluation on a large sample of industrial and academic projects. Adding SMT solvers to Atelier B provers reduces to one fourth the number of sequents that need to be proved interactively.

1 Introduction

The Rodin platform [7] is an integrated design environment for the formal modeling notation Event-B [1]. Rodin is based on the Eclipse framework [18] and has an extensible architecture, where new features, or new versions of existing features, can be integrated by means of plug-ins. It supports the construction of formal models of systems as well as their refinement using the notation of Event-B, based on first-order logic, typed set theory and integer arithmetic. Event-B models should be consistent; for this purpose, Rodin generates proof obligations that need to be discharged (i.e., proved valid).

The proof obligations are represented internally as sequents, and a sequent calculus forms the basis of the verification machinery. Proof rules are applied to a sequent and produce zero, one or more new, usually simpler, sequents. A proof rule producing no sequent is called a discharging rule. The goal of the verification is to build a proof tree corresponding to the application of the proof rules, where all the leaves are discharging rules. In practice, the proof rules are generated by

* This work is partly supported by ANR project DECERT, CNPq/INRIA project SMT-SAVeS, and CNPq grants 560014/2010-4 and 573964/2008-4 (National Institute of Science and Technology for Software Engineering—INES, www.ines.org.br).

J. Derrick et al. (Eds.): ABZ 2012, LNCS 7316, pp. 194–207, 2012.

so-called *reasoners*. A reasoner is a plug-in that can either be standalone or use existing verification technologies through third-party tools.

The usability of the Rodin platform, and of formal methods in general, greatly depends on several aspects of the verification activity:

Automation. Ideally, the proof obligations are validated automatically by reasoners. If human interaction is required for discharging proof obligations (using an interactive theorem prover), productivity is negatively impacted.

Information. Validation of proof obligations should not be sensitive to irrelevant modifications of the model. When modifying the model, large parts of the proof can be preserved if the precise facts used to validate each proof obligation are recorded. Moreover, similar proof obligations can be discharged without further need of the reasoner by noticing the same proof applies, even if the proof obligations differ slightly (on irrelevant parts).

Finally, counter-examples of failed proof obligations can be very valuable to the user as hints to improve the model and the invariants.

Trust. When a prover is used, either the tool itself or its results need to be certified; otherwise the confidence in the formal development is jeopardized.

In this paper, we address the application of a verification approach that may potentially fulfill these three requirements: *Satisfiability Modulo Theory* (SMT) solvers. SMT solvers can *automatically* handle large formulas of first-order logic with respect to some background theories, or a combination thereof, such as different fragments of arithmetic (linear and non-linear, integer and real), arrays, bit vectors, etc. They have been employed successfully to handle proof obligations with tens of thousands of symbols stemming from software and hardware verification. In this paper, we propose a translation of Event-B sequents to SMT input, the difficulty lying essentially in the way sets are translated.

The SMT-LIB initiative provides a standard for the input language of SMT solvers, and, in its last version [4], a command language defining a common interface to interact with SMT solvers. We implemented a Rodin plug-in using this interface. The plug-in also extracts from the SMT solvers some additional *information* such as the relevant hypothesis. Some solvers (e.g. Z3 [9] and veriT [6]) are able to generate a comprehensive proof for validated formulas, which can be verified by a *trusted* proof checker [2]. In the longer term, besides automation, and information, trust may be obtained using a centralized proof manager.

Overview. Section 2 presents two approaches to translate Rodin sequents to the SMT-LIB notation. Section 3 illustrates both approaches through a simple example. Section 4 gives some insights on the techniques employed in SMT solvers to handle Rodin sequents and section 5 presents experimental results, based on the verification activities carried out for a variety of Event-B projects. We conclude by discussing future work.

Throughout the paper, formulas are expressed using the Event- B syntax [14], and sentences in SMT-LIB are typeset using a `typewriter` font.

2 Translating Event-B to SMT

Figure 1 gives a schematic view of the cooperation framework between Rodin and the SMT solver. For each Event-B sequent representing a proof obligation to be validated in Rodin, an SMT formula is built. SMT solvers answer the satisfiability question, so that it is necessary to take the negation of the sequent (to be validated) in order to build a formula to be refuted by the SMT solver. On success a proof and an unsatisfiable core — i.e., the set of facts necessary to prove that the formula is unsatisfiable — may be supplied to Rodin, which will extract a new Event-B proof rule out of it. If the SMT solver does not implement unsatisfiable core generation, the proof rule will assert that the full Event-B sequent is valid (and will only be useful for that specific sequent).

The SMT-LIB standard proposes several "logics" that specify the interpreted symbols that may be used in the formulas. Currently, however, none of those logics fits exactly the language of the proof obligations generated by Rodin. There exists a proposal for such a logic [13], but the existing SMT solvers do not yet implement corresponding reasoning procedures. Our pragmatic approach is thus to identify subsets of the Event-B logics that may be handled by the current tools, either directly or through some simple transformations. Translating Boolean and arithmetic constructs is mostly straightforward, since a direct syntactic translation may be undertaken for some symbols: Boolean operators and constants, relational operators, and most of arithmetic (division and exponentiation operators are currently translated as uninterpreted symbols). As an example of transformation of an Event-B sequent to an SMT formula, consider the sequent with goal $0 < n+1$ under the hypothesis $n \in \mathbb{N}$; the type environment is $\{n \oftype \mathbb{Z}\}$ and the generated SMT-LIB formula is:

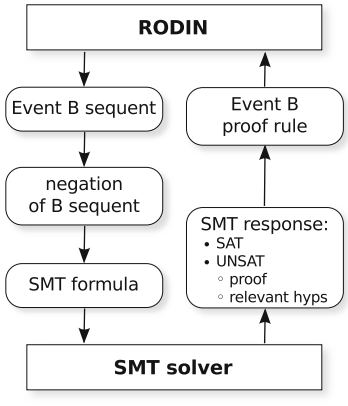


Fig. 1. Schematic view of the interaction between Rodin and SMT solvers

```
(set-logic AUFLIA)
(declare-fun n () Int)
(assert (>= n 0))
(assert (not (< 0 (+ n 1))))
(check-sat)
```

The main issue in the translation of proof obligations to SMT-LIB is the representation of the set-theoretic constructs. We present successively two approaches. The simplest one, presented shortly in the next section, is based on the representation of sets as characteristic predicates [10]. Since SMT solvers handle first-order logic, this approach does not make it possible to reason about sets of sets. The second approach removes this restriction. It uses the *ppTrans* translator, already

available in the Rodin platform; this translator removes most set-theoretic constructs from proof obligations by systematically expanding their definitions.

2.1 The λ-Based Approach

This approach implements and extends the principles proposed in [10] to handle simple sets. Essentially, a set is identified with its characteristic function. For instance the singleton $\{1\}$ is identified with $(\lambda x \oftype \mathbb{Z} \mid x = 1)$ and the empty set is identified with the polymorphic λ-expression $(\lambda x \oftype X \mid \text{FALSE})$, where X is a type variable. The union of (two) sets is a polymorphic higher-order function $(\lambda(S_1 \oftype X \to \text{BOOL}) \mapsto (S_2 \oftype X \to \text{BOOL}) \mid (\lambda x \oftype X \mid S_1(x) \lor S_2(x)))$, etc.

SMT-LIB does not provide a facility for λ-expressions, and has limited support for polymorphism. This approach requires several extensions to SMT-LIB: λ-expressions, a polymorphic sort system, and macro-definitions. Those extensions are actually implemented in the veriT parser. Consider the sequent $A \oftype \mathbb{P}(\mathbb{Z}) \vdash A \cup \varnothing = A$, the translator to this extended SMT-LIB language produces:

```
(declare-fun A (Int) Bool)
(define-fun (par (X) (union ((S1 (X Bool)) (S2 (X Bool))) (X Bool)
             (lambda ((x X)) (or (S1 x) (S2 x))))))
(define-fun (par (X) (emptyset () (X Bool) (lambda ((x X)) false))))
(assert (not (= (union A emptyset) A)))
(check-sat)
```

where `X` denotes a sort variable. The function definitions `union` and `emptyset` are inserted by the translator and are part of a corpus of definitions for most of the set-theoretic constructs (see [10,11] for details). They are divided into a list of sorted parameters, the sort of the result, and the body expressing the value of the result. The macro processor implemented in veriT transforms the goal to

```
(not (forall ((x Int)) (iff (or (A x) false) (A x))))
```

i.e., a first-order formula that may then be handled using usual SMT solving techniques. It is also possible to use veriT only as a pre-processor to produce plain SMT-LIB formulas that are amenable to verification using any SMT-LIB compliant solver.

As already mentioned, the main drawback of this approach is that sets of sets cannot be handled. It is thus restricted to simple sets and relations. Furthermore its reliance on extensions of the SMT-LIB format creates a dependence on veriT as a macro processor. The next approach lifts these restrictions.

2.2 The *ppTrans* Approach

Our second approach uses the translator *ppTrans* provided by the *Predicate Prover* available in Rodin in order to obtain first-order logic formulas which are almost free of set-theoretic elements [12]. It also separates arithmetic, Boolean and set-theoretic constructs from each other and performs simplifications. This approach makes the plug-in independent from veriT, and is more robust with

respect to the translation of relations and functions. On formulas suitable for the previous approach, the translator would however produce very similar results compared to this previous simple approach.

Besides the straightforward translations mentioned earlier, the translation from the *ppTrans* output to SMT-LIB provides some specific rules for the translation of set-theoretic constructs such as the membership operator. For instance assume the input has the following typing environment and formulas:

Typing environment	Formulas
$a \oftype S$	
$b \oftype T$	
$c \oftype U$	$a \in A$
$A \oftype \mathbb{P}(S)$	$a \mapsto b \in r$
$r \oftype \mathbb{P}(S \times T)$	$a \mapsto b \mapsto c \in s$
$s \oftype \mathbb{P}(S \times T \times U)$	

First, for each basic set found in the proof obligation, the translation produces a sort declaration in SMT-LIB. In addition, for each combination of basic sets (either through powerset or Cartesian product), an additional sort declaration is produced. Translating the typing environment produces a sort declaration for each basic set, and combination thereof found in the input:

$$
\begin{aligned}
S &\rightsquigarrow \texttt{(declare-sort S 0)} \\
T &\rightsquigarrow \texttt{(declare-sort T 0)} \\
U &\rightsquigarrow \texttt{(declare-sort U 0)} \\
\mathbb{P}(S) &\rightsquigarrow \texttt{(declare-sort PS 0)} \\
\mathbb{P}(S \times T) &\rightsquigarrow \texttt{(declare-sort PST 0)} \\
\mathbb{P}(S \times T \times U) &\rightsquigarrow \texttt{(declare-sort PSTU 0)}
\end{aligned}
$$

Second, the translation produces a function declaration for each constant:

$$
\begin{aligned}
a \oftype S &\rightsquigarrow \texttt{(declare-fun a () S)} \\
b \oftype T &\rightsquigarrow \texttt{(declare-fun b () T)} \\
c \oftype U &\rightsquigarrow \texttt{(declare-fun c () U)} \\
A \oftype \mathbb{P}(S) &\rightsquigarrow \texttt{(declare-fun A () PS)} \\
r \oftype \mathbb{P}(S \times T) &\rightsquigarrow \texttt{(declare-fun r () PST)} \\
s \oftype \mathbb{P}(S \times T \times U) &\rightsquigarrow \texttt{(declare-fun s () PSTU)}
\end{aligned}
$$

Third, for each type occurring at the right-hand side of a membership predicate, the translation produces fresh SMT function symbols:

```
(declare-fun (MS0 (S PS) Bool))
(declare-fun (MS1 (S T PST) Bool))
(declare-fun (MS2 (S T U PSTU)) Bool)
```

The Event-B atoms can then be translated as follows:

$$
\begin{aligned}
a \in A &\rightsquigarrow \texttt{(MS0 a A)} \\
a \mapsto b \in r &\rightsquigarrow \texttt{(MS1 a b r)} \\
a \mapsto b \mapsto c \in s &\rightsquigarrow \texttt{(MS2 a b c s)}
\end{aligned}
$$

Finally, the Event-B formula where all non-membership set operators have been expanded to their definition is translated to SMT-LIB. For instance, the formula $A \cup \varnothing = A$ would be translated to $\forall x \cdot (x \in A \vee x \in \varnothing) \Leftrightarrow x \in A$, which *ppTrans* simplifies to $\forall x \cdot (x \in A \vee \bot) \Leftrightarrow x \in A$, would be translated to

```
(forall ((x S)) (= (or (MS0 A x) false) (MS0 A x)))
```

While the approach presented here covers the whole Event-B mathematical language and does not require polymorphic types or specific extensions to the SMT-LIB language, the semantics of some Event-B constructs is approximated because some operators become uninterpreted in SMT-LIB (chiefly membership but also some arithmetic operators such as division and exponentiation). However, we can recover their interpretation by adding axioms to the SMT-LIB benchmark, at the risk of decreasing the performance of the SMT-solvers. Some experimentation is thus needed to find a good balance between efficiency and completeness.

Indeed, it appears experimentally that including some axioms of set theory to constrain the possible interpretations of the membership predicate greatly improves the number of proof obligations discharged. In particular, the axiom of elementary set (singleton part) is necessary for many Rodin proof obligations. The translator directly instantiates the axiom for all membership predicates. Assuming `MS` is the membership predicate associated with sorts `S` and `PS`, the translation introduces thus the following assertion:

```
(assert (forall ((x S))
          (exists ((X PS)) (and (MS x X)
                                (forall ((y S)) (=> (MS y X) (= y x)))))))
```

More implementation and optimization details are available in [12]. It is noteworthy that the plug-in based on *ppTrans* detects sequents with only simple sets (i.e., no sets of sets) and uses a translation similar to the λ-based approach in that case. Therefore, the *ppTrans* approach subsumes the λ-based approach.

3 A Small Event-B Example

As a concrete example of translation, this section presents the model of a simple job processing system consisting of a queue and a processor. The basic sets are *JOBS* (the jobs) and *STATUS* (the possible states of the processor), such that `axm1` : $STATUS = \{RUN, IDLE\}$, and `axm2` : $RUN \neq IDLE$. The state of the model has three variables: `proc` (the current status of the processor) `queue` (the jobs currently queued) and `active` (the job being processed, if any). This state is constrained by the following invariants:

`inv1` : $proc \in STATUS$ (typing)
`inv2` : $active \in JOBS$ (typing)
`inv3` : $queue \in \mathbb{P}(JOBS)$ (typing)
`inv4` : $proc = RUN \Rightarrow active \notin queue$

One of the events of the system describes that the processor takes on a new job. It is specified as follows:

Event $SCHEDULE \mathrel{\widehat{=}}$ (the processor takes on a new job)
any
j
where
grd1 : $proc = IDLE$ (the processor must be idle)
grd2 : $j \in queue$ (the job j is in the queue)
then
act1 : $queue := queue \setminus \{j\}$
act2 : $active := j$
act3 : $proc := RUN$
end

To verify that the invariant labeled inv4 is preserved by the *SCHEDULE* event, the following sequent must be proved valid:

$$\texttt{axm1}, \texttt{axm2}, \texttt{inv1}, \texttt{inv2}, \texttt{inv3}, \texttt{inv4}, \texttt{grd1}, \texttt{grd2} \vdash \underbrace{RUN}_{proc} = RUN \Rightarrow \underbrace{j}_{active} \notin \underbrace{queue \setminus \{j\}}_{queue}. \quad (1)$$

The generated proof obligations thus aim to show that the following formula is unsatisfiable:

$$STATUS = \{RUN, IDLE\} \wedge RUN \neq IDLE \wedge$$
$$proc \in STATUS \wedge active \in JOBS \wedge queue \in \mathbb{P}(JOBS) \wedge$$
$$proc = RUN \Rightarrow active \notin queue \wedge$$
$$proc = IDLE \wedge j \in queue \wedge$$
$$\neg(RUN = RUN \Rightarrow j \notin queue \setminus \{j\}).$$

This proof obligation does not contain sets of sets and the approach described in section 2.1 may be applied resulting in the SMT-LIB input presented in Figure 2. Lines 2 and 3 contain the declarations of the sorts corresponding to the basic sets introduced in the context. Lines 4–9 contain the declarations of the function symbols corresponding to the free variables of the proof obligation, and are produced using the typing environment. Note that set *queue* is represented by a unary predicate symbol. Next, the definitions of the macros corresponding to set operators $\in$ and $\setminus$ are included on lines 10–13. Line 14 is the definition of a macro that represents the singleton set $\{j\}$. Lines 15–21 are the result of the translation of the proof obligation itself.

Of course, this proof obligation is also amenable to translation using the approach described in section 2.2, and the corresponding SMT-LIB input is given in Figure 3. Since the proof obligation includes sets of *JOBS*, a corresponding sort *PJOBS* and membership predicate *MJOBS* are declared in lines 4–5. Then, the function symbols corresponding to free identifiers of the sequent are declared at lines 6–11. Finally, the hypothesis and the goal of the sequent are translated to named assertions (lines 12–18).

The sequent described in this section is very simple and is easily verified by both Atelier-B provers and SMT-solvers. Section 5 reports experiments with a large number of proof obligations and establishes a better basis to compare the effectiveness of these different verification techniques.

```
(set-logic AUFLIA)
(declare-sort STATUS 0)
(declare-sort JOBS 0)
(declare-fun RUN () STATUS)
(declare-fun IDLE () STATUS)
(declare-fun proc () STATUS)
(declare-fun active () JOBS)
(declare-fun j () JOBS)
(declare-fun queue (JOBS) Bool)
(define-fun (par (X) (in ((x X) (s (X Bool))) Bool (s x))))
(define-fun (par (X)
              (setminus ((s1 (X Bool)) (s2 (X Bool))) (X Bool)
                (lambda ((x X)) (and (s1 x) (not (s2 x)))))))
(define-fun set1 ((x JOBS)) Bool (= x j))
(assert (and (forall ((x STATUS)) (or (= x RUN) (= x IDLE)))
          (not (= RUN IDLE))
          (=> (= proc RUN) (not (in active queue)))
          (= proc IDLE)
          (in j queue)
          (not (=> (= RUN RUN)
                   (not (in j (setminus queue set1)))))))
(check-sat)
```

Fig. 2. SMT-LIB input produced using the λ-based approach

4 Solving SMT Formulas

In this section, we provide some insight about the internals of SMT solvers, in order to give to the reader an idea on the kind of formulas that can successfully be handled by SMT solvers. A very schematic view of an SMT solver is presented on Figure 4. Basically it is a decision procedure for quantifier-free formulas in a rich language coupled with an instantiation module that handles the quantifiers in the formulas by grounding the problem. For quantified logic, SMT solvers are of course not decision procedures anymore, but they work well in practice if the necessary instances are easy to find and not too numerous.

4.1 Unquantified Formulas

Historically, the first goal of SMT solvers was to provide efficient decision procedures for expressive languages, beyond pure propositional logic. Those solvers have always been based on a cooperation of a Boolean engine, nowadays typically a SAT solver (See [5] for more information on SAT solver techniques and tools), and a theory reasoner to check the satisfiability of a set of literals in the considered language. The Boolean engine generates models for the Boolean abstraction of the input formula, whereas the theory reasoner refutes the sets

```
 1 (set-logic AUFLIA)
 2 (declare-sort STATUS 0)
 3 (declare-sort JOBS 0)
 4 (declare-sort PJOBS 0)
 5 (declare-fun MJOBS (JOBS PJOBS) Bool)
 6 (declare-fun RUN () STATUS)
 7 (declare-fun IDLE () STATUS)
 8 (declare-fun proc () STATUS)
 9 (declare-fun active () JOBS)
10 (declare-fun queue () PJOBS)
11 (declare-fun j () JOBS)
12 (assert (! (forall ((x STATUS)) (or (= x RUN) (= x IDLE))) :named axm1))
13 (assert (! (not (= RUN IDLE)) :named axm2))
14 (assert (! (= proc IDLE) :named grd1))
15 (assert (! (MJOBS j queue) :named grd2))
16 (assert (! (not (=> (= RUN RUN)
17                     (not (and (MJOBS j queue)
18                               (not (= j j)))))) :named goal))
19 (check-sat)
```

Fig. 3. SMT-LIB input produced using the *ppTrans* approach

of literals corresponding to these abstract models by adding conjunctively conflict clauses to the propositional abstraction. This exchange runs until either the Boolean abstraction is sufficiently refined for the Boolean reasoner to conclude that the formula is unsatisfiable, or the theory reasoner concludes that the abstract model indeed corresponds to a model of the formula.

The theory reasoners are themselves based on a combination of decision procedures for various fragments. In our context, the relevant decision procedures are congruence closure — to handle uninterpreted predicates and functions — decision procedure for arrays (typically reduced to some kind of congruence closure), and linear arithmetic. It is possible, using the Nelson-Oppen combination method [15,19], to build a decision procedure for the union of the languages. The theory reasoner used in most SMT solvers is thus able to decide the satisfiability of literals on a language containing a mix of uninterpreted symbols, linear arithmetic symbols, and array operators.

For the theory reasoner and the SAT solver to cooperate successfully, some techniques are necessary. Among these techniques, if a set of literals is found unsatisfiable, it is most valuable to generate small conflict clauses, in order to refine the Boolean abstraction as strongly as possible. Also, theory propagation, that allows to control the decisions taken inside the SAT solver, has proved to be very worthwhile in practice (more can be found about these techniques and SMT solving in general in [3]).

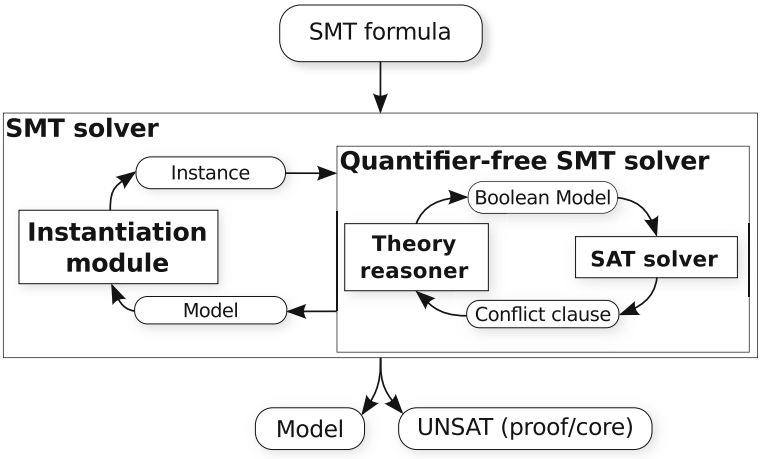


Fig. 4. Schematic view of an SMT solver

4.2 Instantiation Techniques

Automatically finding the right instances of quantified formulas is a key issue for the verification of sequents (as well as proof obligations produced in the context of a number of software verification tools). The quantifier instantiation module is responsible for producing lemmas of the form $\neg\varphi(\mathbf{t}) \vee \exists \mathbf{x}\, \varphi(\mathbf{x})$. Generating too many instances may overload the solver with useless information and exhaust computing resources. Generating too few instances will result in an "unknown", and useless, verdict. We report here how veriT copes with such quantified formulas. Several instantiation techniques are applied in turn: trigger-based, sort-based and superposition techniques.

In a quantified formula $Q\mathbf{x}\, \varphi(\mathbf{x})$, a trigger is a set of terms $T = \{t_1, \cdots t_n\}$ such that the free variables in T are the quantified variables $\mathbf{x}$ and each t_i is a subterm of the matrix $\varphi(\mathbf{x})$ of the quantified formula. Trigger-based instantiation consists in finding, in the formula, sets of ground terms T' that match T, i.e., such that there is a substitution σ on $\mathbf{x}$, where the homomorphic extension of σ over T yields T'. Each such substitution defines an instantiation of the original quantified formula. Some verification systems allow the user to specify instantiation triggers. This is not the case in Rodin, and veriT applies heuristics to annotate quantified formulas with triggers.

If the trigger-based approach does not yield any new instance, veriT resorts to sort-based instantiation. In that case, each quantified variable is instantiated with the ground terms of the formula that have the same sort.

Finally, veriT also features a module to communicate with a superposition-based first-order logic automated theorem prover, namely the E prover [17]. It is built upon automated deduction techniques such as rewriting, subsumption, and superposition and is capable of identifying the unsatisfiability of a set of

quantified and non-quantified formulas. When such a set is found satisfiable, lemmas are extracted from its output and communicated to the other reasoning modules of veriT. The E prover, like many saturation-based first-order provers, is complete for first-order logic with equality.

4.3 Unsat Core Extraction

Additionally to the satisfiability response, it is possible, in case the proof obligation is validated (i.e., when the formula given to the SMT solver is unsatisfiable), to ask for an *unsatisfiable core*. For instance, the sequent (1) discussed in Section 3 and translated into the SMT input on Figure 3 is valid independently of any hypothesis. The SMT input associates labels to the hypotheses and goal, using the reserved SMT-LIB annotation operator `!`. A solver implementing the SMT-LIB unsatisfiable core feature could thus return the list of hypotheses used to validate the goal. In the present case, it would only return the goal since no named hypothesis was used. The plug-in transmits this information to the platform through a rule stating that the goal is unsatisfiable by itself.

Once this rule has been produced, the Rodin platform uses it to discharge any similar proof objective. In particular, if we modify the current sequent without modifying any predicate of the rule (in this case for instance, by changing any irrelevant invariant), the SMT solver rule will still be applicable and the SMT solver will not need to be run again. This is very important for the end user experience: when the user modifies his model, most proofs get reused and the user does not have to wait for the solvers to run again.

The unsat core production for the veriT solver is related to the proof production feature. The solver is indeed able to produce a proof, and it has moreover a facility to prune the proof of unnecessary proof steps and hypotheses. It suffices thus to check the pruned proof and collect all hypotheses in that proof to obtain a superset of the unsat core. Although not minimal in theory, this superset often corresponds to a minimal unsat core, and thus provides the plug-in with high quality information.

5 Experimental Results

We collected a library of proof obligations from several academic (i.e., case studies from books, academic publications, tutorials,...) and industrial projects. The SMT solvers are used with a timeout of 3 seconds[1], on a dual-core Intel Core 2 Duo, cadenced at 2.93GHz, with 4GB of RAM, and running Linux Ubuntu 10.04. Figure 5 presents a summary of the results.[2] The results are detailed separately for academic and industrial projects. The second column gives the number of

[1] This timeout is unusually small for SMT solvers. Larger timeouts would provide better results, but also altering the responsiveness of the Rodin interactive platform.

[2] Notice that the z3 solver was not used at its full power since its Model Based Quantifier Instantiation feature (MBQI) was not fully functional on the latest currently available version for our system.

	Number of proof obligations	Atelier B	alt-ergo-r217	cvc3-2011-11-21	veriT-dev-r2863	veriT & E-prover	z3-3.2	SMT Portfolio (Open)	SMT Portfolio	Portfolio
Academic	2786	474	660	434	646	553	490	271	231	136
Industrial	855	126	212	178	192	177	160	83	62	8
Total	3641	600	872	612	838	730	650	354	293	144

Fig. 5. Experimental results (number of proof obligation *not* discharged by the tools)

proof obligations, the next columns the number of them not validated by the tool heading the column, i.e., the number of proof obligations requiring human interaction after automatic application of only this tool.

The column "Atelier B" gives the number of proof obligations that were *not* discharged by the prover from Atelier B. The five following columns give, for several SMT solvers, the number of proof obligations that the solvers were not able to validate. The "SMT Portfolio" column relates the number of proof obligations unproved after trying all considered SMT solvers, whereas the "SMT Portfolio (Open)" column only consider the solvers with a permissive license, (i.e., distributed with the plug-in). The "Portfolio" column gives the remaining sequents after running both the SMT solvers and the prover from Atelier B.

On Figure 6 only the proof obligations undischarged by the prover from Atelier B are considered, and we detail for each solver (or group of solvers) the number of validated formulas.

It is worth noticing that SMT solvers altogether validate more proof obligations than the Atelier B prover. But the important and strong conclusion that can be deduced from these tables is that SMT solvers complement the Atelierb B prover. From 600 proof obligations that are not validated by the prover

	Undischarged by Atelier B	alt-ergo-r217	cvc3-2011-11-21	veriT-dev-r2863	veriT & E-prover	z3-3.2	SMT Portfolio (Open)	SMT Portfolio
Academic	474	121	259	106	155	227	313	338
Industrial	126	91	99	110	105	68	118	118
Total	600	212	358	216	260	295	431	456

Fig. 6. Improvement over Atelier B (number of validated proof obligations)

from Atelier B — and that required human interaction — around 75% are discharged automatically by SMT solvers. It thus *divides by four* the amount of verification conditions requiring human interaction.

Besides this complementarity, the tools have different features that justify having a portfolio of solvers: veriT has a permissive license and produces proofs, from which it is easy to extract unsatisfiable cores; cvc3 is quite efficient, but extracting unsatisfiable cores from its output is not trivial; z3 is certainly very powerful, but has a restrictive license.

6 Conclusion

SMT solving is a formal verification technique successfully applied to various domains including verification. SMT solvers do not have built-in support for set-theoretic constructs found in Rodin sequents, but different translation approaches may be applied to map such constructs to a logic they handle. We presented two such approaches: a basic one that tackles simple sets, and another one that is furthermore able to handle more elaborate structures.

We evaluated experimentally the efficiency of SMT-solvers against proof obligations resulting from the translation of Rodin sequents. In our sample of industrial and academic projects, the use of SMT solvers on top of Atelier B provers reduces to one fourth the number of unverified sequents. This plug-in is available through the integrated software updater of Rodin (instructions at `http://wiki.event-b.org/index.php/SMT_Plug-in`).

The results are very encouraging and motivate us to progress further by implementing and evaluating new translation approaches, such as representing functions using arrays in the line of [8]. Also, as SMT solvers can provide models when a formula is satisfiable, it would be possible, with additional engineering effort, to use such models to report counter-examples in Rodin.

Cooperation of deduction tools is very error-prone, not only because it relies on the correctness of many large and complex tools, but also because of the translations. Certification of proofs in a centralized trusted proof manager would be the answer to this problem. Preliminary works in this direction exist [16].

Acknowledgement. we would like to thank the anonymous reviewers for their remarks.

References

1. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
2. Armand, M., Faure, G., Grégoire, B., Keller, C., Théry, L., Werner, B.: A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In: Jouannaud, J.-P., Shao, Z. (eds.) CPP 2011. LNCS, vol. 7086, pp. 135–150. Springer, Heidelberg (2011)

3. Barrett, C., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, ch. 26, pp. 825–885. IOS Press (February 2009)
4. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard Version 2.0 (2010)
5. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2009)
6. Bouton, T., de Oliveira, D.C.B., Déharbe, D., Fontaine, P.: veriT: An Open, Trustable and Efficient SMT-Solver. In: Schmidt, R.A. (ed.) CADE-22. LNCS, vol. 5663, pp. 151–156. Springer, Heidelberg (2009)
7. Coleman, J., Jones, C., Oliver, I., Romanovsky, A., Troubitsyna, E.: RODIN (Rigorous open Development Environment for Complex Systems). In: Fifth European Dependable Computing Conference: EDCC-5 supplementary volume, pp. 23–26 (2005)
8. Couchot, J.-F., Déharbe, D., Giorgetti, A., Ranise, S.: Scalable Automated Proving and Debugging of Set-Based Specifications. Journal of the Brazilian Computer Society 9, 17–36 (2003)
9. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
10. Déharbe, D.: Automatic Verification for a Class of Proof Obligations with SMT-Solvers. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) ABZ 2010. LNCS, vol. 5977, pp. 217–230. Springer, Heidelberg (2010)
11. Déharbe, D.: Integration of SMT-solvers in B and Event-B development environments. Science of Computer Programming (March 2011)
12. Konrad, M., Voisin, L.: Translation from Set-Theory to Predicate Calculus. Technical report, ETH Zurich (2011)
13. Kröning, D., Rümmer, P., Weissenbacher, G.: A Proposal for a Theory of Finite Sets, Lists, and Maps for the SMT-LIB Standard. In: Informal proceedings, 7th Int'l Workshop on Satisfiability Modulo Theories (SMT) at CADE 22 (2009)
14. Métayer, C., Voisin, L.: The Event-B mathematical language (2009), http://deploy-eprints.ecs.soton.ac.uk/11/4/kernel_lang.pdf
15. Nelson, G., Oppen, D.C.: Simplifications by cooperating decision procedures. ACM Transactions on Programming Languages and Systems 1(2), 245–257 (1979)
16. Schmalz, M.: The logic of Event-B, Technical report 698, ETH Zürich, Information Security (2011)
17. Schulz, S.: E - A Brainiac Theorem Prover. AI Communications 15(2/3), 111–126 (2002)
18. The Eclipse Foundation. Eclipse SDK (2009)
19. Tinelli, C., Harandi, M.T.: A new correctness proof of the Nelson–Oppen combination procedure. In: Baader, F., Schulz, K.U. (eds.) Frontiers of Combining Systems (FroCoS), Applied Logic, pp. 103–120. Kluwer Academic Publishers (March 1996)

Refinement Plans for Informed Formal Design*

Gudmund Grov[1], Andrew Ireland[2], and Maria Teresa Llano[2]

[1] University of Edinburgh, School of Informatics, Edinburgh, UK
[2] Heriot-Watt University, MACS, Edinburgh, UK

Abstract. Refinement is a powerful technique for tackling the complexities that arise when formally modelling systems. Here we focus on a posit-and-prove style of refinement, and specifically where a user requires guidance in order to overcome a failed refinement step. We take an integrated approach – combining the complementary strengths of top-down planning and bottom-up theory formation. In this paper we focus mainly on the planning perspective. Specifically, we propose a new technique called *refinement plans* which combines both modelling and reasoning perspectives. When a refinement step fails, refinement plans provide a basis for automatically generating modelling guidance by abstracting away from the details of low-level proof failures. The refinement plans described here are currently being implemented for the Event-B modelling formalism, and have been assessed on paper using case studies drawn from the literature. Longer-term, our aim is to identify refinement plans that are applicable to a range of modelling formalisms.

1 Introduction

We focus here on a layered style of formal modelling, where a design is developed as a series of abstract models – level by level concrete details are progressively introduced via provably correct *refinement* steps. There are two major approaches in achieving this style of formal modelling: the *rule-based* approach and the *posit-and-prove* approach; examples can be found in [25] and [21,1], respectively.

The work reported here aims to enhance the posit-and-prove approach. Specifically, we have developed a technique called *refinement plans* which automatically generates guidance for users within posit-and-prove formal modelling. Like many approaches to design, whether informal [13] or formal [2], our technique relies upon patterns. While we focus here on relatively small patterns, we believe this will provide a foundation upon which to explore larger refinement patterns in the future.

The novelty of our refinement plans is that they combine modelling and reasoning patterns, enabling us to computationally exploit the subtle interplay that exists between modelling and reasoning – what we call *reasoned modelling*. Our refinement plans are heuristic in nature, and can be applied *flexibly* during a development. This flexibility is achieved through *partial matching* and *proof-failure*

* An earlier version of this paper appears in the informal proceedings of *AFM'10* [23].

J. Derrick et al. (Eds.): ABZ 2012, LNCS 7316, pp. 208–222, 2012.

analysis. While we focus here on Event-B, we believe the ideas that underpin reasoned modelling are generic with respect to posit-and-prove.

The paper is structured as follows: §2 provides background on Event-B along with our previous work on automated theory formation and reasoned modelling critics. Our refinement plans mechanism is described in §3 and an example of a refinement plan is presented in §4. The current implementation of the mechanism is outlined in §5, while §6 describes related and future work.

2 Background

2.1 Event-B Refinement by Example

An Event-B development is structured into *models* and *contexts*. A context describes the static part of a system, e.g. *constants* and their *axioms*, while a model describes the dynamic part. Models are themselves composed of three components: *variables*, *events* and *invariants*. Variables represent the state of the system, events are guarded actions that update the variables and invariants are constraints on the variables. By way of illustration we now consider the Event-B model shown in Figure 1. This model is a fragment of a flash-based file system developed in [9]. The fragment shown in Figure 1 deals with the function of writing the content of a file. In the abstract model, the event *writefile* is responsible for writing the content of file *f*, *wbuffer(f)*, into *fcontent* in an atomic step. In the concrete model the content is written one page at a time into a temporary storage *fcont_tmp* (event *w_step*) before being written to the actual storage *fcontent* (event *w_end_ok*). The new events *w_start* and *w_step* are said to refine *skip*, while event *w_end_ok* refines the abstract event *writefile*.

In order to prove that the refinement is indeed correct, invariants must be provided. In the example three invariants are specified in the concrete model, the two first invariants specify the type of the new variables *fcont_tmp* and *writing*, while the last invariant specifies a property of the refinement step, that is, that when the writing process starts for a given file, the content of *fcont_tmp* is a subset or is equal to the content of *wbuffer*.

2.2 Reasoned Modelling Critics

The notion of *reasoned modelling* (REMO) was first introduced in [19], where we described REMO *critics*. These critics are motivated by the way in which proof-failure analysis typically informs the activity of modelling – and is achieved by combining common patterns of proof failure with generic modelling guidance. The mechanism builds upon the notion of *proof critics* [18], a proof patching technique developed within the context of *proof planning* [4]. The key difference is that our REMO critics exploit failure at the level of modelling and proof. As a result, we reduce the burden that users experience in manually analysing low-level proof failures, presenting them instead with high-level modelling alternatives. These ideas were further developed in [20] where an implementation via

ABSTRACT MODEL:

Variables
fcontent, w_opened_files, wbuffer, file_size, power_on, dateLastModified

Invariants
w_opened_files ⊆ files
file_size ∈ files → ℕ
power_on ∈ BOOL
fcontent ∈ files → CONTENT
dateLastModified ∈ (files ∪ directories) → DATE
wbuffer ∈ w_opened_files → CONTENT

Event writefile ≙
any f
where
f ∈ w_opened_files
power_on = TRUE
then
fcontent(f) := wbuffer(f)
dateLastModified(f) := nowdate
file_size(f) := card(wbuffer(f))
end

CONCRETE MODEL:

Variables
fcontent, w_opened_files, wbuffer, file_size, power_on, dateLastModified, writing, fcont_tmp

Invariants
writing ⊆ w_opened_files
fcont_tmp ∈ writing → CONTENT
∀ f· f∈ writing ⇒ fcont_tmp(f) ⊆ wbuffer(f)

Event w_start ≙
any f
where
f ∈ w_opened_files
f ∉ writing
power_on = TRUE
then
writing := writing ∪ { f}
fcont_tmp(f) := ∅
end

Event w_step ≙
any f i data
where
power_on = TRUE
f ∈ writing
i ∈ ℕ
data ∈ DATA
i↦ data ∈ wbuffer(f)
i ∉ dom(fcont_tmp(f))
then
fcont_tmp(f) :=
fcont_tmp(f) ∪ { i↦data}
end

Event w_end_ok ≙
refines writefile
any f
where
f ∈ writing
dom(fcont_tmp(f)) = dom(wbuffer(f))
power_on = TRUE
then
fcontent(f) := fcont_tmp(f)
dateLastModified(f) := nowdate
file_size(f) := card(fcont_tmp(f))
writing := writing \ { f}
fcont_tmp := { f} ⩤ fcont_tmp
end

Fig. 1. Event-B model of a flash file system [9]

the Remo tool, a prototype plug-in for the Eclipse-based Rodin toolset implemented in OCaml, is described. The work presented here aims to extend the REMO critics so as to generate modelling guidance at the level of refinement.

2.3 HRemo

HRemo [24] is an automatic approach to invariant discovery that builds upon HR [8], a machine learning system that performs descriptive induction to form a theory about a set of objects of interest which are described by a set of *core concepts*. Theories are constructed in HR via theory formation steps which attempt to construct new concepts, i.e. *non-core concepts*, through the use of a set of production rules and, if empirical relationships are found between concepts, formulate *conjectures* and evaluate the results. Thus, the theories HR produces contain concepts which relate the objects of interest, conjectures which relate the concepts; and proofs which explain the conjectures.

HRemo builds upon HR, animation and proof-failure analysis to automatically suggest candidate invariants of Event-B models. In particular, a set of

heuristics are used to guide the search for invariants in HR. These heuristics exploit the strong interplay between modelling and reasoning in Event-B by using the feedback provided by failed POs to make decisions about how to configure HR. Specifically, the approach consists of analysing the structure of failed POs to automate the:

1. Prioritisation in the development of conjectures about specific concepts.
2. Selection of appropriate production rules that increase the possibilities of producing the missing invariants.
3. Filtering of the final set of conjectures to be analysed as candidate invariants.

HRemo uses two classes of heuristics to constrain the search for invariants: those used in configuring HR, i.e. configuration heuristics, and those used in selecting conjectures from HR's output, i.e. selection heuristics. Using proof-failure analysis to prune the wealth of conjectures HR discovers, these heuristics have proven highly effective at identifying missing invariants. Further information about HRemo and examples of its application can be found in [24].

3 Refinement Plans

Before providing details on the structure of refinement plans, we first sketch how we envisage they will be used within a development environment such as Rodin. Given a development, our approach provides a basis for classifying refinement steps against known patterns of refinement, i.e. syntactic features of abstract and concrete models.

However, we are interested in situations where a refinement step is flawed, and thus the proof tools fail to discharge some of the POs. In such situations our approach attempts to automatically generate guidance, i.e. modelling alternatives that overcome the failure. This is achieved by firstly identifying which of the known patterns are closely aligned to the given failed refinement. As well as a refinement pattern, each refinement plan is associated with a set of *critics* – where a critic represents a common pattern of failure at the level of POs and models. Moreover, associated with each critic is generic modelling guidance as to how to overcome the failure, e.g. invariant speculation, event speculation, etc.

Table 1. Refinement pattern analysis of Event-B case studies

	control refinement				**data refinement**			
Model	**RP_1**	**RP_2**	**RP_3**	**RP_4**	**RP_5**	**RP_6**	**RP_7**	**RP_8**
Cars on a bridge [1]	✔✔		✔✔	✔				
Mondex [6]	✔	✔		✔✔✔✔		✔	✔	✔✔
Flash file system [9]		✔✔✔	✔✔		✔			
Location access ctrl. [1]		✔		✔	✔✔✔			
PLC*				✔	✔✔	✔		
Network topology [15]		✔		✔			✔✔	✔✔

*Available at `http://homepages.inf.ed.ac.uk/ggrov/`

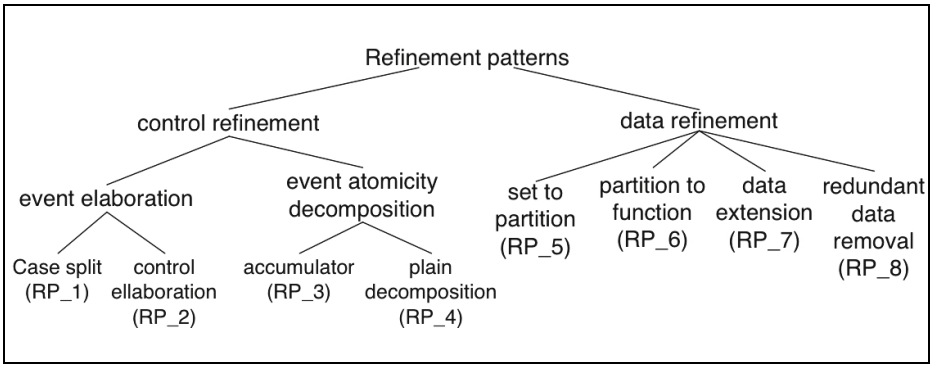


Fig. 2. A hierarchical classification of common refinement patterns

When a common pattern of failure is instantiated by a particular refinement step, the associated guidance will typically only be partially instantiated. To fully instantiate the guidance for a given flawed refinement requires in general additional search and reasoning – this is where we exploit HRemo.

Currently we have identified 8 basic refinement patterns by analysing a range of Event-B case studies from the literature. These patterns form a hierarchy as shown on Figure 2. Each leaf node denotes a distinct pattern of refinement, while the internal nodes reflect the sharing of properties between patterns. This classification provides us with a better understanding of what a user is trying to achieve in a refinement step as well as facilitates the matching process. The 8 basic patterns in Figure 2 are described briefly below:

case split: refers to refinement steps in which an abstract event is refined in the concrete model by two or more events.

control elaboration: relates to models that constrain the application of existing events based on extensions of the state and independently from the operation of new events at the concrete level.

accumulator: deals with models in which actions of an abstract atomic event are performed in the concrete model via iteration.

plain decomposition: makes reference to models in which an abstract event is refined by a sequence of new and refined events. New events are used to pre-process data used in the abstract event.

set to partition: refers to models in which an abstract variable is refined by partitioning it through a set of new variables in the concrete model.

partition to function: involves refinement steps in which an abstract partition of variables is refined into a function in the concrete model.

data extension: refers to models in which an abstract variable is refined into a concrete variable that extends the abstract data type in order to control membership of data in the variable.

redundant data removal: involves the elimination of data from the abstract level that is not being used to control the operation of any event.

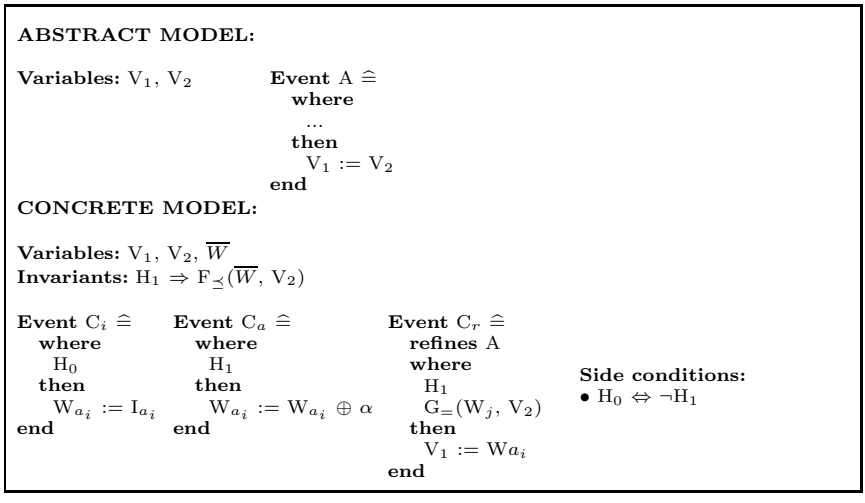


Fig. 3. Accumulator plan – Modelling pattern

The relation between this hierarchy and the case studies is given in Table 1. Currently we have explored in detail four refinement plans, i.e. *case split*, *accumulator*, *set to partition* and *partition to function*. Below in §4 we focus on the *accumulator refinement plan* and two of its associated critics.

4 The Accumulator Refinement Plan

A technique for breaking up an atomic event has been proposed by Butler and Yadav in [6] and further developed in [5,10,11]. The accumulator refinement plan has been inspired by this work. The key difference with our work is that as well as the modelling patterns, we are also interested in the deductive patterns and in providing guidance when a pattern breaks in a development.

The accumulator pattern deals with models in which actions of an abstract atomic event are performed in the concrete model via iteration. This is achieved through the use of new events that iteratively accumulate the value from the abstract action. The modelling and PO patterns of the accumulator plan are shown in Figures 3 and 4, respectively. Note that we use the Vs and Ws to denote meta-variables, and specifically we use I_{ai} to represent the initial value assigned to meta-variable W_{ai}. Note also that we use F, G and H to denote meta-predicates, where subscripts are used to restrict their instantiation, e.g. $G_=$ restricts G to be an equality. The key elements in the refinement are:

- The abstract model has an atomic event that is refined in the concrete model.
- A set of new variables $\overline{W} = \{W_1, ..., W_n\}$ are introduced.
- A subset of $\overline{W}$, W_a, which denotes accumulator variables. That is, for each $W_{a_i} \in W_a$ (where, $1 \leq i \leq n$) there is an accumulator event, i.e. the action pattern $W_{a_i} := W_{a_i} \oplus \alpha$ occurs, an initialisation event and a refined event.
- An initialisation event (C_i), accumulator event (C_a), and refined event (C_r).

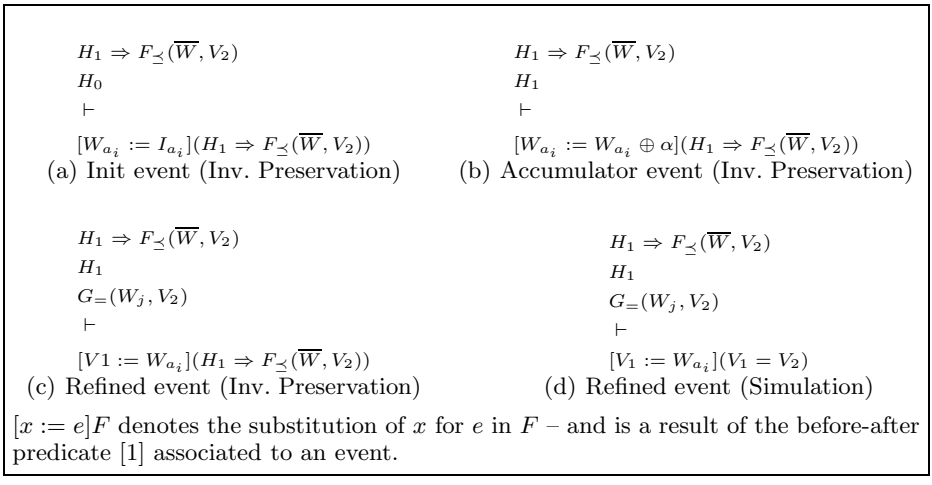


Fig. 4. Accumulator plan – PO patterns

• An invariant, $H_1 \Rightarrow F_{\preceq}(\overline{W},\ V_2)$, that explains the refinement; i.e. that the content of the accumulator variable(s) is contained within the value assigned in the abstract model – the $\preceq$ symbol generalises the containment relationship.
• The initialisation, accumulator(s) and refined events must preserve the invariant, Figures 4(a), 4(b) and 4(c), respectively.
• The refined event must simulate the abstract action, Figure 4(d).
An instance of the accumulator pattern occurs in the model presented in Figure 1, in which the action:

$$\mathit{fcontent}(f) := \mathit{wbuffer}(f)$$

within the abstract event *writefile* is achieved within the concrete model via iteration. Below we present the fragments of the events that match the modelling pattern at the concrete level:

```
Event w_start ≙
  any f
  where ...
   f ∉ writing
  then ...
   fcont_tmp(f):= ∅
   writing:=writing∪{f}
end
```

```
Event w_step ≙
  any f i data
  where ...
   f ∈ writing
  then
   fcont_tmp(f) :=
    fcont_tmp(f)∪{i↦data}
end
```

```
Event w_end_ok ≙
refines writefile
  any f
  where ...
   f ∈ writing
   dom(fcont_tmp(f))=dom(wbuffer(f))
  then ...
   fcontent(f) := fcont_tmp(f)
end
```

Note that variable *fcont_tmp* acts as the accumulator variable. Event *w_start* initialises the process by assigning the empty set to *fcont_tmp* and adding file f to the *writing* state, event *w_step* iteratively adds the content of each page to the accumulator variable, and event *w_end_ok* assigns the content of *fcont_tmp* to *fcontent* after all the pages have been written. Finally, the invariant:

$$\forall f \cdot f \in \mathit{writing} \Rightarrow \mathit{fcont\_tmp}(f) \subseteq \mathit{wbuffer}(f)$$

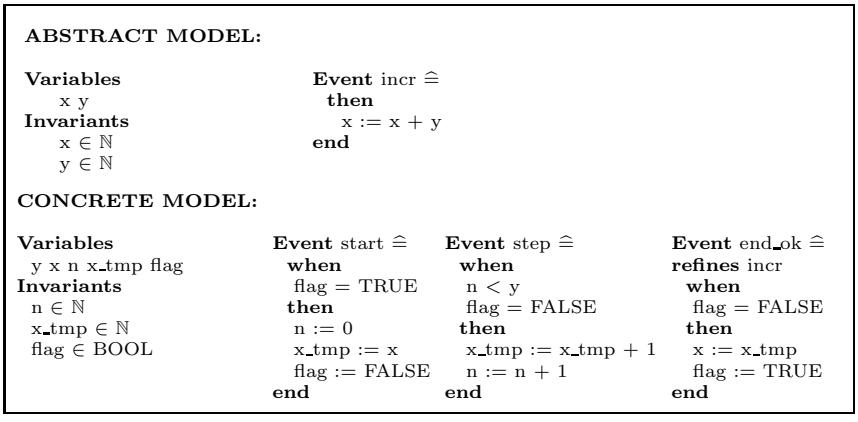


Fig. 5. Flawed accumulator plan instance – Addition example

specifies that while file f is in the writing state, the value of *wbuffer(f)* is accumulated in *fcont_tmp(f)*.

4.1 Accumulator Refinement Plan Critics

We now focus on the critics aspect of refinement plans, and how partial matching, with respect to the modelling pattern, and failure analysis are used to automatically generate modelling guidance.

We have identified a number of critics for the accumulator plan:

postGuard_speculation critic: considers the case when the guard of the refined event that ensures the accumulation process is complete is either flawed or missing.

invariant_speculation critic: handles the case when the accumulator invariant is wrong or missing.

accumulator_speculation critic: handles the case when an accumulator event refines an abstract event whose actions are performed in an atomic step.

initialisation_speculation critic: considers the case when the accumulation process does not have an initialisation phase.

loopGuard_speculation critic: deals with the case when the guard(s) that deal with the loop in the accumulator event is wrong or missing.

guard_relocation critic: deals with guards from the abstract event that need to be moved to a new event in the accumulation sequence.

Due to space constraints we only present two critics: *postGuard_speculation* and *invariant_speculation*. In order to illustrate the application of these critics we will use a simple model that adds a value to a variable. The model, taken from [9], is shown in Figure 5. The running example of the flash file system, Figure 1, is not used because it is not possible to perform the simulation of this model through the ProB animator and animation is a key component of the critics presented. We give more information about these limitations in §6.

The abstract model in Figure 5 shows an atomic event *incr* that increments the value of x by the value of y. In the concrete model the value of y is iteratively assigned in event *step* to an accumulator variable *x_tmp*, while in the event *end_ok* the value of *x_tmp* is assigned to the abstract variable x after the accumulation has finished. Event *start* initialises the accumulation. Note that variable n is a new variable used to control the accumulation process. Note also that the accumulator invariant as well as the post-guard are missing from the model; this gives rise to the following failed SIM PO associated to event *end_ok*:

$$\textbf{end\_ok/SIM PO: } \text{flag} = \text{FALSE} \vdash \text{x\_tmp} = \text{x} + \text{y}$$

At this point the *postGuard_speculation* and *invariant_speculation* critics are triggered. First the critic that deals with the guard is applied because in order to reason about the invariant, the events in the model need to be correct.

Preconditions for the Postguard_speculation Critic

P1. *An accumulator pattern is identified.*
This precondition holds for the addition model since a partial match of the accumulator pattern is detected. That is, apart from the invariant and the guard, the other key elements of the pattern are identified in the model.

P2. *The simulation PO pattern associated to the refined event fails.*
This precondition holds since the end_ok/SIM PO fails.

P3. *The post-accumulator guard is missing or it is not compatible with the guard pattern, i.e.* $G_{=}(W_j, V_2)$.
As mentioned above, the post-accumulator guard is missing from the model in Figure 5; therefore this precondition holds.

Guidance

A guard with the shape $G_{=}(W_j, V_2)$ *must be added to the refined event.*
As preconditions P1, P2 and P3 succeeded, the guard pattern is instantiated. The guidance is then to add a guard to event *end_ok* with the form:

$$G_{=}(x\_tmp, n, x, y)$$

We will revisit this guard schema below, and describe how it is instantiated. For now assume that the correct instantiation is available, i.e. $y = n$. Because the invariant is also missing, the failure persists, this triggers the invariant critic.

Preconditions for the Invariant_speculation Critic

P1. *An accumulator pattern is identified.*
This precondition succeeds as explained for the guard critic.

P2. *The SIM PO pattern associated to the refined event fails.*
This precondition holds since the end_ok/SIM PO fails. The new form of the failed PO is:

$$\text{flag} = \text{FALSE}, \text{y} = \text{n} \vdash \text{x\_tmp} = \text{x} + \text{y}$$

P3. *The post-accumulator guard is not missing and it is compatible with the guard pattern, i.e.* $G_{=}(W_j, V_2)$.
The post-accumulator guard $y = n$ is present in the refined event and is compatible with the pattern.

P4. *The accumulator invariant is missing or it is not compatible with the invariant pattern, i.e.* $H_1 \Rightarrow F_{\preceq}(\overline{W}, V_2)$.
As mentioned above, the accumulator invariant is missing from the model; therefore, this precondition holds.

Guidance
An invariant of the shape $H_1 \Rightarrow F_{\preceq}(\overline{W}, V_2)$ *must be added to the concrete model.*
As with the guard critic, preconditions P1 to P4 succeeded; therefore the invariant pattern is instantiated as follows (where due to use of natural numbers $\preceq$ is instantiated to $\leq$):

$$(flag = FALSE) \Rightarrow F_{\leq}(x\_tmp, n, x, y)$$

As can be observed the guidance currently provided is in the form of partial instantiations of the schemas. At this point, there are three options to find the correct instantiation: i) through interaction with the user, ii) through the use of proof patterns, or iii) through the use of automated theory formation (ATF).

Here we use ATF, and in particular the HRemo system to search for the missing invariants and guards. However, currently HRemo cannot be used to analyse models where the events are incorrect. This prevents us from using HRemo directly to discover missing guards. On the contrary, HRemo can be used to discover missing invariants. However, with regards to the invariant schema given above, HRemo on its own fails to find the missing invariant after 1000 theory formation steps, which give rise to 7959 conjectures. This does not imply that the invariant cannot be found, rather it means that additional search is required. In the next section we show that by combining refinement plans and event error traces with HRemo these negative issues can be effectively addressed.

4.2 Combining Modelling Patterns with HRemo

The process of finding a "correct" refinement typically involves exploring many incorrect models. Refinement plans aim at providing guidance when a failed refinement is closely aligned with a known pattern. However, as shown through the guidance obtained by the critics presented in §4.1, refinement plans are limited by the patterns observed. On the other hand, as mentioned above, HRemo also exhibits some limitations. In order to overcome these limitations we combine both approaches, in particular we extend the work presented in [24] by:

- using the ProB animator [22] to generate traces that contain undesirable states which can be used by HRemo to find missing guards, and
- using the patterns of invariants and guards available in the refinement plans to automatically tailor the search in HRemo.

As mentioned in §2.3, two type of heuristics are used by HRemo, configuration heuristics (CH) and selection heuristics (SH), when a pattern of an invariant or a guard is available then the following heuristics are applied:

Configuration Heuristics

CH1. *Prioritise core and non-core concepts expected in the invariant or guard.*
CH2. *Follow with core and non-core concepts that occur within failed POs.*
CH3. *Generate conjectures that are compatible with the type of the expected invariant or, if looking for a guard, generate only equivalence conjectures.*
CH4. *Select only production rules which will give rise to conjectures relating to the type of the expected invariant or guard.*

Equivalence conjectures are always generated since this optimises the theory formation process [8].

The selection heuristics for the search of invariants based on patterns are the same than those applied in [24]. This requires selecting conjectures where the sets of variables occurring on the left- and right-hand sides are disjoint, selecting the most general conjectures, and selecting the conjectures that discharge the failed POs and that minimise the number of additional proof failures. Note that here the selection of conjectures is focused in the core and non-core concepts that relate to the invariant pattern, as opposed to [24] which focused on core and non-core concepts from the failed POs.

In the case of missing guards the selection process differs. Through the use of the ProB animator it is possible to detect event errors which result in traces that contain undesirable states. That is, ProB can animate various refinement levels concurrently, allowing the detection of errors associated with refinement; in particular, ProB can detect violation of guard strengthening in a refined event, we exploit this animation analysis provided by ProB to tailor HRemo in the search of guards. When a trace of this type is generated we provide HRemo with the concept of *good* states, which are the steps of the trace with no guard strengthening errors associated. The selection is then focused on conjectures that express equivalences with the concept of *good*, i.e. conjectures of the form:

$$good \Leftrightarrow \phi$$

where ϕ represents the potential missing guard.

Regarding the *postGuard_speculation* and *invariant_speculation* critics, presented in §4.1, the guidance is achieved by using the partially instantiated guard and invariant schemas to tailor HRemo in the search. To illustrate, lets revisit the instantiated guard schema obtained by the *postGuard_speculation* critic:

$$G_{=}(xtmp, n, x, y)$$

based on this, we instantiate the configuration heuristics as follows:

CH1: Prioritised concepts from the guard schema: *x_tmp*, n, x and y .
CH2: Concepts from the failed POs: *flag*, $x+y$, *x_tmp*$=x+y$ and *flag=FALSE*.

CH3: Searching for a guard; thus, only equivalence conjectures are generated.
CH4: As the top-level symbol in the guard is = and the involved variables are natural numbers, the *numrelation* and *arithmetic* PRs are selected.

After 65 seconds and 1000 theory formation steps HRemo returns 1 conjecture:

$$good \Leftrightarrow y = n$$

which means that the missing guard is y = n. A similar approach is followed in the search for the missing invariant. After 45 seconds and 1000 theory formation steps HRemo returns 1 conjecture:

$$flag = FALSE \Rightarrow x\_tmp = x + n$$

which represents the missing invariant.

5 Implementation and Results

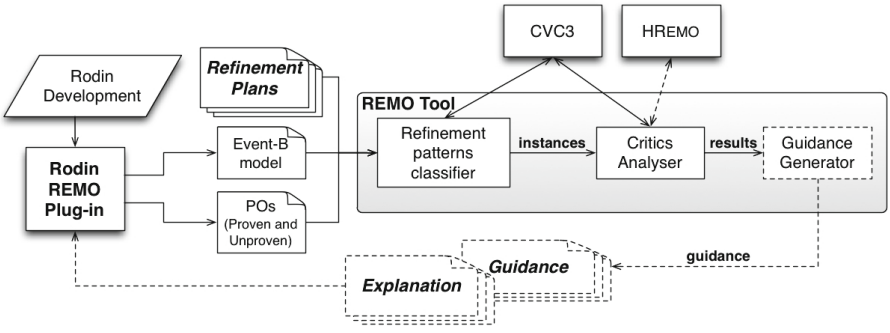
We have implemented and tested the *set to partition* and *partition to function* refinement plans. Moreover we have conducted the experiments described above with the *accumulator* plan. This implementation effort was partially integrated into the Remo toolset, which we mentioned in §2.2. An architectural view of the implementation is given in Figure 6. The prototype is partial in that the integration of the guidance from Remo back into Rodin is still under development. Note that in terms of results, our implementation is still at the experimental stage, and we are now looking to undertake more extensive testing (see §6).

6 Related and Future Work

The motivation behind the work described here is to correct a refinement which almost matches an existing pattern. Similar tools and techniques we are familiar with – such as the BART tool for classical B [26]; the ZRC refinement calculus for Z [7]; and more relevant, Event-B based tools and techniques as described in [16,17,2,12] – instead focus on automating the refinement from a given step to a more concrete step. None of the tools can handle the failure-analysis we have described here.

Our implementation of the refinement plans highlighted in this paper is ongoing. We plan to automate the link with HRemo and the external theorem prover(s) as well as to automate the communication of the results from Remo back to the user[1]. We also plan to further test and develop our existing plans, drawing upon industrial case studies arising from the DEPLOY project[2]. We are also interested in exploring the potential for using machine learning techniques to automate the discovery of new plans.

[1] One possible route is via Lopatkin's *transformation patterns* plug-in. For details see `http://wiki.event-b.org/index.php/Transformation_patterns`
[2] See `http://www.deploy-project.eu/`



The Rodin REMO plug-in provides the interface between the Rodin toolset and the REMO tool. The plug-in generates two files, i) and Event-B model, and ii) the associated POs. The refinement plans are also given as an input to the tool. The Refinement Patterns Classifier uses the refinement plans to classify the patterns of refinement used in a development. The role of the Critics Analyser is to find ways of overcoming failures. This is achieved via the critics mechanism which analyses the instances given by the classifier together with the POs. The classifier and the analyser interact with CVC3 when a precondition requires proof. The analyser also interacts with HREMO in order to search for missing/wrong invariants or guards. The results of this analysis are passed to the Guidance Generator which takes the raw results from the analyser and produces a formatted, ranked (ordered) list of alternative guidance suggestions, which will then be sent to the Rodin REMO plug-in and presented to the user. Note that the stippled lines indicate work in progress.

Fig. 6. The REMO tool architecture

Animation traces from ProB are used in the analysis phase of our work-flow. Such information about how the events relate to each other should naturally be part of the pre-conditions of plans and critics, and we will extend them with such information. Bendisposto and Leuschel [3], have developed a tool which turn ProB traces into a more abstract *flow graphs* which shows the order events may be executed[3]. We plan to add support for such "event flow" information in the preconditions, either as described in [3], or ideally extended with support for infinite systems ([3] only supports finite models), which undoubtedly will require theorem proving support.

Finally, animation is key to our approach, where the quality of the invariants produced by HREMO strongly depends on the quality of the animation traces. We believe that increasing the randomness in the production of the traces is an area where the ProB animator requires improvement. Specifically, this limitation arose during our analysis of the Mondex [6] case study.

[3] Hallerstede [14] suggests an approach achieving a similar goal, but here the user has to add more structure to the model.

7 Conclusions

We have described refinement plans, a technique which provides automatic modelling guidance for users of posit-and-prove style formal refinement. Building upon common patterns of refinement, the technique uses an automated analysis of refinement failure at the level of models and POs in order to focus the search for modelling guidance. To provide flexibility in terms of the guidance that can be generated, we have experimented with the HRemo theory formation tool. Through these experiments we have shown that combining refinement plans with HRemo improves the search for invariants and has suggested how missing guards can be discovered automatically.

Acknowledgements. Thanks to Alison Pease, Simon Colton, Julian Gutierrez, Alan Bundy and the Mathematical Reasoning Group at Edinburgh University. This work was supported by EPSRC grants EP/F037058, EP/H024204, EP/E005713, EP/E035329, EP/J001058. Maria Teresa Llano was also supported by a BAE systems studentship. Finally, we thank the anonymous ABZ reviewers for their constructive feedback.

References

1. Abrial, J.-R.: Modelling in Event-B: System and Software Engineering. Cambridge University Press (2010)
2. Abrial, J.-R., Hoang, T.S.: Using Design Patterns in Formal Methods: An Event-B Approach. In: Fitzgerald, J.S., Haxthausen, A.E., Yenigun, H. (eds.) ICTAC 2008. LNCS, vol. 5160, pp. 1–2. Springer, Heidelberg (2008)
3. Bendisposto, J., Leuschel, M.: Automatic Flow Analysis for Event-B. In: Giannakopoulou, D., Orejas, F. (eds.) FASE 2011. LNCS, vol. 6603, pp. 50–64. Springer, Heidelberg (2011)
4. Bundy, A.: A science of reasoning. In: Computational Logic: Essays in Honor of Alan Robinson. MIT Press (1991)
5. Butler, M.: Decomposition Structures for Event-B. In: Leuschel, M., Wehrheim, H. (eds.) IFM 2009. LNCS, vol. 5423, pp. 20–38. Springer, Heidelberg (2009)
6. Butler, M., Yadav, D.: An incremental development of the mondex system in Event-B. Formal Aspects of Computing 20(1) (2008)
7. Cavalcanti, A., Woodcock, J.: ZRC - A Refinement Calculus for Z. Formal Aspects of Computing 10(3) (1998)
8. Colton, S.: Automated Theory Formation in Pure Mathematics. Springer (2002)
9. Damchoom, K.: An Incremental Refinement Approach to a Development of a Flash-Based File System in Event-B. PhD thesis, University of Southampton (2010)
10. Salehi Fathabadi, A., Butler, M.: Applying Event-B Atomicity Decomposition to a Multi Media Protocol. In: de Boer, F.S., Bonsangue, M.M., Hallerstede, S., Leuschel, M. (eds.) FMCO 2009. LNCS, vol. 6286, pp. 89–104. Springer, Heidelberg (2010)
11. Salehi Fathabadi, A., Rezazadeh, A., Butler, M.: Applying Atomicity and Model Decomposition to a Space Craft System in Event-B. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 328–342. Springer, Heidelberg (2011)

12. Fürst, A.: Design Patterns in Event-B and Their Tool Support. Master's thesis, ETH Zürich (2009)
13. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
14. Hallerstede, S.: Structured Event-B Models and Proofs. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) ABZ 2010. LNCS, vol. 5977, pp. 273–286. Springer, Heidelberg (2010)
15. Hoang, T.S., Basin, D., Kuruma, H., Abrial, J.-R.: Development of a network topology discovery algorithm. DEPLOY project Repository, `http://deploy-eprints.ecs.soton.ac.uk/82/`
16. Iliasov, A.: Refinement Patterns for Rapid Development of Dependable Systems. In: EFTS. ACM Press (2007)
17. Iliasov, A.: Design Components. PhD thesis, University of Newcastle (2008)
18. Ireland, A.: The Use of Planning Critics in Mechanizing Inductive Proofs. In: Voronkov, A. (ed.) LPAR 1992. LNCS, vol. 624, pp. 178–189. Springer, Heidelberg (1992)
19. Ireland, A., Grov, G., Butler, M.: Reasoned Modelling Critics: Turning Failed Proofs into Modelling Guidance. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) ABZ 2010. LNCS, vol. 5977, pp. 189–202. Springer, Heidelberg (2010)
20. Ireland, A., Grov, G., Llano, M., Butler, M.: Reasoned modelling critics: turning failed proofs into modelling guidance. In: Science of Computer Programming. Elsevier (2011) (in Press)
21. Jones, C.B.: Systematic Software Development using VDM. Prentice Hall (1990)
22. Leuschel, M., Butler, M.: ProB: A Model Checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 855–874. Springer, Heidelberg (2003)
23. Llano, M., Grov, G., Ireland, A.: Automatic guidance for refinement based formal methods. In: AFM Workshop (2010)
24. Llano, M.T., Ireland, A., Pease, A.: Discovery of invariants through automated theory formation. In: Refine Workshop. EPTCS, vol. 55 (2011)
25. Morgan, C.: Programming from Specifications. Prentice–Hall (1990)
26. Requet, A.: BART: A Tool for Automatic Refinement. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) ABZ 2008. LNCS, vol. 5238, pp. 345–345. Springer, Heidelberg (2008)

Refinement by Interface Instantiation*

Stefan Hallerstede[1] and Thai Son Hoang[2]

[1] Aarhus University, Denmark
[2] ETH Zürich, Switzerland

Abstract. Decomposition is a technique to separate the design of a complex system into smaller sub-models, which improves scalability and team development. In the shared-variable decomposition approach for Event-B sub-models share external variables and communicate through external events which cannot be easily refined.

Our first contribution hence is a proposal for a new construct called interface that encapsulates the external variables, along with a mechanism for interface instantiation. Using the new construct and mechanism, external variables can be refined consistently. Our second contribution is an approach for verifying the correctness of Event-B extensions using the supporting Rodin tool. We illustrate our approach by proving the correctness of interface instantiation.

Keywords: Event-B, Decomposition, Refinement, External variables.

1 Introduction

Decomposition of a model into sub-models allows one to continue refining the sub-models independently of each other while preserving the properties of the full model. The decomposition method for Event-B proposed by Abrial [1] splits events between the sub-models. Variables are split correspondingly into external variables shared by the sub-models and internal variables private to each model. For all external variables, external events that mimic the effect of corresponding (internal) events of other sub-models have to be added. If we want to refine external variables, we have to provide a gluing invariant that is functional, say, $v = h(w)$ where v are the abstract variables and w the concrete variables. Abrial [1] also proposes to rewrite the external events with $v := h(w)$ so that concrete and abstract events are equivalent. Internal variables and internal events are refined as usual in Event-B [2].

We call a collection of external variables with the external invariant an *interface*. Modelling interfaces by marking the corresponding variables as being external and refining them by specifying functional invariants makes it difficult to decompose and refine a model repeatedly. Fig.1 illustrates the problem where a model M is decomposed three times and the resulting sub-models are refined. We are interested in the two sub-models M_1 and M_2 at the bottom. How do we find the shared external invariant?

* This research was carried out as part of the EU FP7-ICT research project DEPLOY (Industrial deployment of advanced system engineering methods for high dependability and productivity) `http://www.deploy-project.eu`

J. Derrick et al. (Eds.): ABZ 2012, LNCS 7316, pp. 223–237, 2012.

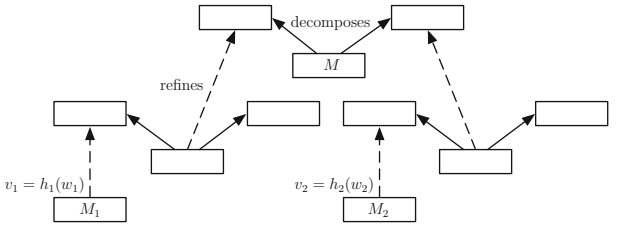


Fig. 1. Maintaining the external invariant of several sub-models

The lists of variables w_1, w_2, v_1 and v_2 are not necessarily disjoint. Let w be the list of variables occurring in w_1 or w_2 and v be the list of variables occurring in v_1 or v_2. We need to find one suitable external invariant $v = h(w)$ to be used in the sub-models M_1 and M_2. What is the shape of h? Furthermore, when refining M_2 we have to think about the necessary changes to M_1. As a consequence of the current situation, interfaces are refined to implementation level before decomposition. This complicates the use of decomposition on higher levels of abstraction. We would prefer a method where the necessary reasoning can be restricted to one place. The functional invariant h should be evident and easily maintainable also in the face of potential changes to the sub-models and the interfaces.

Using our approach of interface instantiation this can be done. Because we are treating instantiation like a special form of refinement, we can combine interface instantiation steps with refinement steps. This gives us some liberty in arranging complex refinements. We also encourage a decomposition style where a separate theory of interface instantiation is maintained. We think, that this contributes substantially to obtain models that are easier to understand and to modify. Interface instantiation supports a more incremental approach to decomposition because modifications that concern several components can be confined to only one place: the interface.

We call the very specific form of interface refinement that we use *interface instantiation*. To be useful, it should

(i) ease the proof effort compared to [1],
(ii) help to structure complex mixtures of decomposition and refinement,
(iii) work seamlessly with Event-B as it is. (It should not depend on translations.)

We argue by means of a case study that we have achieved this. The case study addresses a difficulty of relating Event-B refinement to Problem Frames elaboration [9] discussed in [5]. It has been composed from [11] and [5]. We have down-sized it in order to focus on the problem of the refinement of external variables, that is, the interfaces. We have a tool for decomposition [14] but we do not have implemented a software tool for interface instantiation. Instantiation of carrier sets has been implemented similarly internally in the ProB tool, in order to achieve better performance when model checking and constraint checking [5]. The case study as presented in [11] uses Problem Frames to achieve traceability of requirements. We have not used Problem Frames in this article because they are not required to explain interface instantiation. This also permits us to cast

the problem entirely in Event-B terminology. However, the proposed method of instantiation could be used with Problem Frames as employed in [5,11].

In the modularisation approach for Event-B presented in [8], the notion of interface has been used to capture software specifications using some interface variables and operations acting on these variables. The intention behind the use of interfaces is to separate specifications from their implementations. Our notion of interface is intended to provide efficient support for refining external variables following Abrial's decomposition method for system models. There was an earlier attempt at external variable refinement that is hinted at in the specification of the proof obligation generator for the Rodin tool [6]. This was considered too complicated and not feasible for large systems that are decomposed and refined repeatedly. We think, that our approach solves the problem. Poppleton [12] discusses external refinement based on Abrial's approach but also does not provide a practicable technique for doing so. The approach of modelling extensible records [4] also permits a form interface instantiation. A difficulty with using this approach is caused by the explicit mathematical model used for record representations and the need to specify always successor values for all fields of a record. However, extensible records could be used with our approach where it would appear useful. Behavioural interface refinement such as discussed in [13] addresses changing traces sub-models can exhibit, usually adding new events. It does not consider refinement of shared variables.

2 Event-B

Event-B models are described in terms of the two basic constructs: *contexts* and *machines*. Contexts contain the static part of a model whereas machines contain the dynamic part. Contexts may contain *carrier sets*, *constants*, *axioms*, and *theorems*, where carrier sets are similar to types [2]. Machines provide behavioural properties of Event-B models. Machines may contain *variables*, *invariants*, *theorems*, and *events*. Variables v describe the state of a machine. They are constrained by invariants $I(v)$. Theorems $L(v)$ describe consequences of the invariants, i.e., we have to prove $I(v) \Rightarrow L(v)$.

Events. Possible state changes are described by means of events. Each event is composed of a *guard* $G(t, v)$ and an *action* $A(t, v)$, where t are *parameters* the event may contain. We denote an event e by any t when $G(t, v)$ then $A(t, v)$ end in its most general form, or when $G(v)$ then $A(v)$ end if event e does not have parameters, or begin $A(v)$ end if in addition the guard equals *true*. A dedicated event of the third form is used for *initialisation*.

Assignments. The action of an event is composed of several *assignments*: $x :| Q(t, v, x')$, where x are some variables and $Q(t, v, x')$ a predicate. Variable x is assigned a value satisfying a predicate. Two variants of assignments are defined as follows: $x := B(t, v) \mathrel{\widehat{=}} x :| x' = B(t, v)$ and $x :\in B(t, v) \mathrel{\widehat{=}} x :| x' \in B(t, v)$, where $B(t, v)$ are expressions.

Refinement. A machine N can refine another machine M. We call M the *abstract* machine and N a *concrete* machine. The state of the abstract machine

is related to the state of the concrete machine by a *gluing invariant* $J(v, w)$ associated with the concrete machine N, where v are the variables of the abstract machine and w the variables of the concrete machine. Each event e of the abstract machine is *refined* by one or more concrete events f.

Decomposition. A machine M can be decomposed into several machines [1,7]. We limit the discussion here to the decomposition into two machines for the purpose of this article. Let M be a machine with variables x_1, x_3, x_5 and invariants $I(x_1, x_3, x_5)$, $I_1(x_1, x_3)$, $I_3(x_3)$ and $I_5(x_3, x_5)$. Furthermore, let e_1, e_2, e_4 and e_5 be events of M, accessing different sets of variables as follows.

$$\mathsf{e}_1 \mathrel{\widehat{=}} \mathsf{any}\ t_1\ \mathsf{where}\ G_1(t_1, x_1)\ \mathsf{then}\ x_1 :|\ S_1(t_1, x_1, x_1')\ \mathsf{end}$$
$$\mathsf{e}_2 \mathrel{\widehat{=}} \mathsf{any}\ t_2\ \mathsf{where}\ G_2(t_2, x_1, x_3)\ \mathsf{then}\ x_1, x_3 :|\ S_2(t_2, x_1, x_3, x_1', x_3')\ \mathsf{end}$$
$$\mathsf{e}_4 \mathrel{\widehat{=}} \mathsf{any}\ t_4\ \mathsf{where}\ G_4(t_4, x_3, x_5)\ \mathsf{then}\ x_3, x_5 :|\ S_4(t_4, x_3, x_5, x_3', x_5')\ \mathsf{end}$$
$$\mathsf{e}_5 \mathrel{\widehat{=}} \mathsf{any}\ t_5\ \mathsf{where}\ G_5(t_5, x_5)\ \mathsf{then}\ x_5 :|\ S(t_5, x_5, x_5')\ \mathsf{end}$$

Machine M can be decomposed into two separate machines: M_1 with events e_1 and e_2; and M_5 with events e_4 and e_5. This is illustrated in Fig.2. As a result of the decomposition, M_1 has private variables x_1 and shared variables x_3. Invariants $I_1(x_1, x_3)$ and $I_3(x_3)$ can be distributed to M_1. The resulting sub-machine M_1 has two *internal* events e_1 and e_2 and one *external* event a_4 which abstracts[1] e_4 projected on the state containing only x_3: $\mathsf{a}_4 \mathrel{\widehat{=}} \mathsf{any}\ t_4, x_5\ \mathsf{where}\ G_4(t_4, x_3, x_5)\ \mathsf{then}\ x_3 :|\ \exists x_5' \cdot S_4(t_4, x_3, x_5, x_3', x_5')\ \mathsf{end}$. Machine M_5 is similar to M_1, with two internal events e_4 and e_5; and an external event a_2 that abstracts e_2. It has the private variables x_5 and the shared variables x_3. Machines M_1 and M_5 can be developed independently with the constraints that the shared variables cannot be removed and the external events cannot be made more non-deterministic or less non-deterministic.

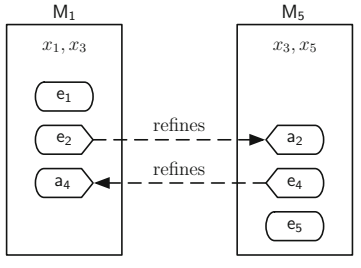


Fig. 2. Maintaining the external invariant of several sub-models

Note that invariant $I(x_1, x_3, x_5)$ are not copied to either M_1 or M_5. A possibility is to project also this invariant onto the corresponding state using existential quantifier. For example, the following can be added to M_1 as an invariant $\exists x_5 \cdot I(x_1, x_3, x_5)$.

3 Instantiation

Carrier Set and Constant Instantiation. Contexts can be extended as usual in Event-B but we allow additionally to specify expressions to instantiate constants and carriers sets. Carriers sets must be instantiated by type expressions $e(t)$ and constants can be instantiated by any expression $f(d)$.

[1] "a abstracts e" is the same as "e refines a".

context C
  sets s
  constants c
  axioms $A(s, c)$

context D
  extends C with $s = e(t)$, $c = f(d)$
  sets t
  constants d
  axioms $B(t, d)$

The equalities specifying the instantiation are treated similarly to axioms. The abstract constants and carriers sets that are instantiated remain visible. By contrast, the instantiation proposed in [2] replaces constants and carrier sets in the instantiating context. Still, they are similar to [2]: The equations of the extends-clause are used to rewrite the abstract axioms. If this changes an axiom, that axiom must be *proved* to hold in the instantiating context. Otherwise, nothing needs to be proved. This ensures that instantiation itself does not introduce new facts. The *instantiation* proof obligation is $B(t, d) \Rightarrow A(e(t), f(d))$. In summary, conventional Event-B context extension is instantiation with identity. Only abstract axioms of C with instantiated constants need to be proved as theorems in D. The other axioms are preserved by extension.

Connecting Machines to Interfaces. Interfaces are declared in contexts and used in machines by connecting a machine to the interface. The machines must see the corresponding context:

context C
  interface ii
    fields m
    constraints $P(m)$

machine M
  sees C
  connects ii

The constraints of an interface can refer to all constants and carrier sets of the surrounding context. In machine M the fields m are treated like variables and the constraints $P(m)$ like external invariants.

Interface Instantiation. Interfaces can be instantiated by specifying equalities $m = h(n)$ for replacing fields of an abstract interface m by fields of a concrete interface n. The names on the right-hand side of the equation must not occur in the abstract interface.

context C
  interface ii
    fields m
    constraints $P(m)$

context D
  extends C
  interface jj instantiates ii with $m = h(n)$
    fields n
    constraints $Q(n)$

The expression h is often composed of pair-expressions "$\cdot \mapsto \cdot$". Interfaces are not associated with proof obligations. The constraints $P(m)$ of ii are contained in interface jj as specified by the instantiation $m = h(n)$, that is, they become $P(h(n))$. Similarly to machine variables, field names of interfaces cannot be

reintroduced. Similarly to machine invariants, constraints are accumulated in by instantiation: the constraints of interface jj are $Q(n) \land P(h(n))$.

External Event Refinement. Using interface instantiation we permit refinement of external events. Consider the following external event e operating on the external variables x and its refinement f operating on the external variables y. The refinement of external variables is captured by the following relationship $x = h(y)$. Note that external events does not refer to any internal variables: it can only refer to external variables of the corresponding model.

$\mathsf{e} \mathrel{\widehat{=}} \mathsf{any}\ t\ \mathsf{when}\ G(t,x)\ \mathsf{then}\ x :|\ S(t,x,x')\ \mathsf{end}$
$\mathsf{f} \mathrel{\widehat{=}} \mathsf{any}\ u\ \mathsf{when}\ H(u,y)\ \mathsf{with}\ W(t,u,x,y,y') \land x' = h(y')\ \mathsf{then}\ y :|\ R(u,y,y')\ \mathsf{end}$

where $W(t,u,x,y,y') \land x' = h(y')$ is the witness for the refinement of e by f. It incorporates the refinement of external variables with function h.

Beside the proof obligations to prove that f is a refinement of e, we also need to prove that f is refined by e. The idea here is to prove the latter using the same given witnesses. The proof obligations are as follows (for clarity, we omit reference to possible abstract invariant $I(x)$ and other concrete invariant $J(x,y)$, which should be in the assumption of the proof obligations).

Witness Feasibility

$$x = h(y) \land G(t,x) \land S(t,x,x') \Rightarrow (\exists u, y' \cdot W(t,u,x,y,y') \land x' = h(y'))$$

In the case that h is a bijective function, the existence of y' is hence trivial, and the proof obligation can be rewritten as follows.

$$x = h(y) \land G(t,x) \land S(t,x,x') \land x' = h(y') \Rightarrow (\exists u \cdot W(t,u,x,y,y'))$$

Guard Weakening

$$x = h(y) \land G(t,x) \land S(t,x,x') \land W(t,u,x,y,y') \land x' = h(y') \Rightarrow H(u,y)$$

(Co-)Simulation

$$x = h(y) \land G(t,x) \land S(t,x,x') \land W(t,u,x,y,y') \land x' = h(y') \Rightarrow R(u,y,y')$$

Note that *invariant preservation* for the refinement of f by e can be derived from the *invariant preservation* for the refinement of e by f and the fact that we use the same witnesses.

4 Case Study: Modelling of a Cruise Control System

We present interface instantiation by means of a model of a cruise control system. A cruise control systems permits the driver of a car to select a target speed that the vehicle should attain. The system will try to maintain a vehicle speed as close as possible to the target speed. Note, that our main interest is to discuss

interface instantiation. So we will only discuss the function of the cruise control system as far as necessary for that discussion. We have modeled the system using the Rodin tool [3], emulating instantiation similarly to the approach of [5]: interfaces are represented syntactically by a lexical convention and carrier set instantiation is modelled by suitable bijections.

We want to implement a cruise control system sy0 by the three components: the controller sy4cr, the engine sy4even and the exterior sy1evsi. Fig. 3 shows the components and their interfaces.

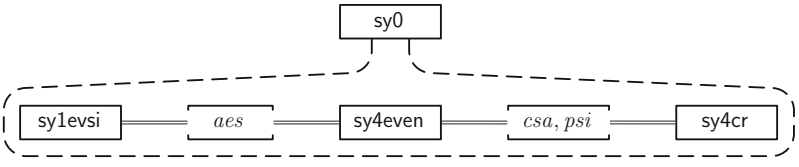


Fig. 3. Architecture of the system in terms of components and interfaces

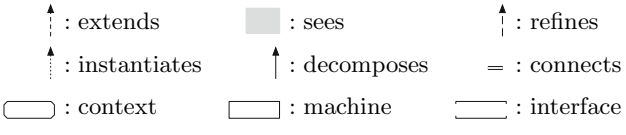


Fig. 4. Legend of used symbols

The symbols used in this figure and later figures are listed in Fig. 4. The implementations of the controller and the engine are connected by means of two interfaces: concrete speed and acceleration, *csa*, and internal pedal signals passed on from the exterior, *psi*. The interface to the exterior, *aes*, is kept abstract in the implementation.

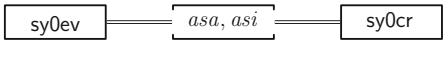


Fig. 5. Abstract components and their abstract interfaces

More abstract system models should not be forced to use the interfaces *csa* and *psi* but permit abstractions thereof (Fig. 5). The details of the interfaces should be introduced step by step, introducing the abstract interfaces *asa* and *asi* first. We prefer to refine the controller and the engine but keep the exterior abstract at first. We do not want to decide on all interfaces before decomposing system sy0: we have not decided yet on the shape of the implementation of component sy1evsi and of interface *aes*. Interface *aes* could be used to implement an interface to the exterior or it could be used for animation and visualisation [10], for instance. The problem we face is to fit the abstract components of Fig. 5 between sy0 and the implementation in Fig. 3.

4.1 The Full Model: Refinement, Decomposition and Instantiation

We present an overview of the full model and discuss specific issues in subsequent sections. Fig. 6 shows the details of the development outlined in Fig. 3. We do not discuss all aspects of the development but focus on the following three:

Section 4.2: Decomposition: introducing interfaces
Section 4.3: Mixing instantiation and refinement
Section 4.4: Repeated instantiation

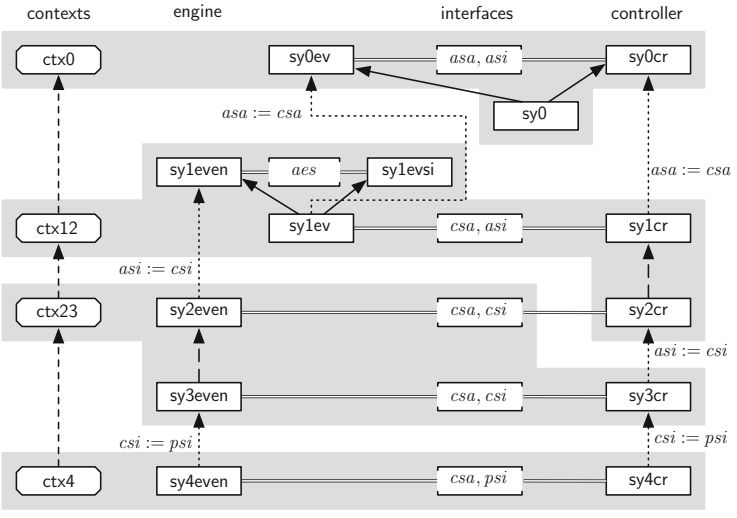


Fig. 6. Overview of system model

The separate contexts ctx0, ctx12, ctx23 and ctx4 correspond to the accompanying instantiations of carrier sets and constants.

4.2 Decomposition: Introducing Interfaces

Abstract Model. The model sy0 from which we start the development declares variables *sig*, *cs*, *vs*, *md*, *ts*, *acc* modelling external signals, internal control signals, vehicle speed, control mode, target speed and acceleration. It does not contain any interfaces. This means we can refine this model in the usual way. Context ctx0 declares constants *ES*, *CS*, *VS*, *VA*, *VRA*, etc, modelling external signals, control signals, vehicle speed, vehicle acceleration, restricted vehicle acceleration. It postulates the axiom

$$VRA \subseteq VA \tag{1}$$

We have invented constant *VRA* to make the invariant more interesting. The constants determine the possible values of the variables by means of the invariant of sy0:

$$sig \in ES \land cs \in CS \land \ldots \land md \in \{C, AC, NC\} \land (md = C \Rightarrow acc \in VRA)$$

The constant C models "cruise control active"; AC models "change of target speed"; NC models "cruise control not active". The carrier sets, e.g., K of CS

or S of VS, are not used in the machine. The reason for this is that they can only be instantiated by type expressions. However, the more common case is that we need to instantiate by some more constrained set. See, for example, the instantiations of C, AC and NC in Section 4.3.

Events of the Abstract Model. We discuss three of the events of sy0: event *chm* ("change mode") models an internal state change of the controller,

$$\mathsf{event}\ chm\ \widehat{=}\ \mathsf{begin}\ md :|\ md' \in \{C, AC, NC\} \wedge (md' = C \Rightarrow acc \in VRA)\ \mathsf{end}\,;$$

event *chaac* ("change acceleration in mode AC") models output to the engine,

$$\mathsf{event}\ chaac\ \widehat{=}\ \mathsf{when}\ md = AC\ \mathsf{then}\ acc :\in VA\ \mathsf{end}\,;$$

event *chcs* ("change control signals") models input from the engine,

$$\mathsf{event}\ chcs\ \widehat{=}\ \mathsf{begin}\ cs := fcs(sig)\ \mathsf{end}\,.$$

It would be tempting to specify in the abstract event *chcs* the assignment $cs := sig$. However, this asserts that *cs* and *sig* have the same type. Once the system is decomposed, we would have to refine them in the same way. To avoid this, we have introduced function *fcs* mapping from the type of *sig* to the type of *cs*. Models always need to be prepared for decomposition. Our method of instantiation does not change this.

Decomposition of the Abstract Model. Decomposing sy0 into sy0ev and sy0cr we have to introduce interfaces *asa* and *asi*:

interface *asa*	interface *asi*
fields *cs*	fields vs, va
constraints $cs \in CS$	constraints $vs \in VS \wedge va \in VA$

Machine sy0ev has one internal variable *sig* and connects to the two interfaces *asa* and *asi*. Machine sy0cr connects to the same interfaces and has two internal variables *ts* and *md*. We split the events in the usual way depending on which variables and fields the events refer to. Except for the use of interfaces the decomposition method of [1] works as before.

4.3 Mixing Instantiation and Refinement

Instantiation. We refine sy0cr by sy1cr by instantiating interface *asa* by *csa* while refining variable *md* by variable *nd*. The constraints and instantiation equalities of *csa* become part of the gluing invariant of sy1cr. The abstract constants VS, VA and VRA are instantiated by integer ranges: $VS = mS\,..\,MS$, $VA = mA\,..\,MA$ and $VRA = mRA\,..\,MRA$ constrained by axioms

$$\ldots \wedge mA \leq mRA \wedge mRA \leq MRA \wedge MRA \leq MA \wedge \ldots \tag{2}$$

To satisfy the instantiation proof obligation we have to verify that (2) implies (1). For clarity we introduce a new name for the interface containing the instantiated constants:

interface *csa* instantiates *asa*

Machine sy1cr and sy1ev now both need to be connected to interface *csa* replacing *asa*. The machine also need to see the extended context ctx12.

Refinement. Variable *md* is itself refined by instantiating the constants C, AC and NC, using the gluing invariant $nd \in md$, and constant instantiations $C = \{CRS, RES\}$, $AC = \{ACC, DEC\}$, $NC = \{OFF, ERR, REC\}$. Note, how closely constant instantiation and refinement are linked in the refinement of *md*. The type of the abstract variable *md* has been instantiated such that the gluing invariant becomes simply $nd \in md$.

4.4 Repeated Instantiation

First Instantiation. Continuing the development from sy1cr and sy1ev, we first instantiate interface *asi* by *csi*

interface *csi* instantiates *asi* with $cs = (ps \mapsto cis \mapsto is)$
  fields ps, cis, is
  constraints $ps \in PS \land cis \in CIS \land is \in IS$

where PS is a constant of context ctx32. The context also declares two constants PSE and PSS such that $PSE \subseteq PS \land PSS \subseteq PS \land PSE \cap PSS = \varnothing$. This is used for a first refinement of event *chm* into two events *chme* and *chmn*:

event *chme* refines *chm* $\widehat{=}$
  when $ps \in PSS \cup PSE$ then $nd :\in \{ERR, REC\}$ end
event *chmn* refines *chm* $\widehat{=}$
  when $ps \notin PSS \cup PSE$
  then $nd :| nd' \in \{CRS, RES\} \Rightarrow acc \in mRA \mathinner{..} MRA$ end

Second Instantiation. Subsequently we instantiate *csi* by *psi*

interface *psi* instantiates *csi* with $ps = (pbp \mapsto pbe \mapsto pcp \mapsto pce \mapsto pae)$
  fields $pbp, pbe, pcp, pce, pae, cis, is$
  constraints $pbp \in \mathbb{B} \land pbe \in \mathbb{B} \land pcp \in \mathbb{B} \land pce \in \mathbb{B} \land pae \in \mathbb{B}$

We instantiate the constants PSE and PSS

$$PSE = \{bp \mapsto be \mapsto cp \mapsto ce \mapsto ae | \mathbf{T} \in \{be, ce, ae\}\}$$
$$PSS = \{bp \mapsto be \mapsto cp \mapsto ce \mapsto ae | \mathbf{T} \notin \{be, ce, ae\} \land \mathbf{T} \in \{bp, cp\}\}$$

and prove $PSE \cap PSS = \varnothing$ as postulated above.

event $chmrb$ refines $chme$ $\widehat{=}$
when $\mathbf{T} \notin \{pbe, pce, pae\} \wedge \mathbf{T} = pbp$ then $nd := REC$ end
event $chmrc$ refines $chme$ $\widehat{=}$
when $\mathbf{T} \notin pbe, pce, pae \wedge \mathbf{T} = pcp$ then $nd := REC$ end
event $chmbe$ refines $chme$ $\widehat{=}$ when $\mathbf{T} = pbe$ then $nd := ERR$ end
event $chmce$ refines $chme$ $\widehat{=}$ when $\mathbf{T} = pce$ then $nd := ERR$ end
event $chmae$ refines $chme$ $\widehat{=}$ when $\mathbf{T} = pae$ then $nd := ERR$ end

This last instantiation is much more concise than the refinement suggested in [5]. We can avoid a lot of the overhead that is usually incurred by using refinement emulating instantiation. Not having dedicated tool support yet, the most elaborate proof of this development occurred when instantiating VA and VRA by integer intervals. With instantiation support in place this would have been trivial using the fact that $VRA = mRA \mathinner{..} MRA$. The difficulty in the proof of refinement emulating instantiation is caused by the need to use a bijection $\iota \in mRA \mathinner{..} MRA \rightarrowtail VRA$ so that the equation for the instantiation becomes $VRA = \iota[mRA \mathinner{..} MRA]$.

5 Correctness

We have used the Rodin tool for verifying the correctness of interface refinement. First we present a technique for verifying extensions of Event-B. We believe that it is useful beyond the use in this article for verifying the correctness of interface instantiation.

A Technique for Proving Event-B Extensions Correct. The general idea is to encode a generic model using the Rodin tool, and illustrating the extended method using the generic model. Typically, the correctness of an extension can be stated as follows: assume the consistency of some input model, then prove the consistency of some resulting model. Using the models generated by the tool, consistency of the input model are represented by the proof obligations associated with the model. To turn them into assumptions for our reasoning, we add these proof obligations as axioms to the context. Using the axioms, we prove the consistency of the resulting model.

An example of our approach is as follows. Let M be a machine with variables x, invariant $I(x)$, and an event e as follows.

$$\mathsf{e} \;\widehat{=}\; \text{any } t \text{ where } G(t,x) \text{ then } x :| S(t,x,x') \text{ end} \quad .$$

First, we model the type of variables x and parameters t using some carrier sets X and T. Subsequently, I, G and S can be declared as constants with appropriate type, i.e. $I \in \mathbb{P}(X)$, $G \in \mathbb{P}(T \times X)$, and $S \in \mathbb{P}(T \times X \times X)$. The machine M is encoded accordingly using the above context, where predicates are translated using membership ($\in$) operator[2]. For example, the invariant $I(x)$ is translated as $x \in I$. Event e hence becomes

[2] This is a well-known technique to model predicate constants or variables in Event-B using first-order logic.

$$\mathsf{e} \;\widehat{=}\; \text{any } t \text{ where } t \mapsto x \in G \text{ then } x :| \; t \mapsto x \mapsto x' \in S \text{ end}$$

The proof obligation stating that e maintains invariant $I(x)$, i.e., $I(x) \wedge G(t,x) \wedge S(t,x,x') \Rightarrow I(x')$ is encoded as an axiom in the context as follows $\forall t, x, x' \cdot x \in I \wedge t \mapsto x \in G \wedge t \mapsto x \mapsto x' \in S \Rightarrow x' \in I$.

Assume that N is a correct refinement of M, retaining the abstract variables x. In N, abstract event e is refined by concrete event f where parameter t is also retained.

$$\mathsf{f} \;\widehat{=}\; \text{any } t \text{ where } H(t,x) \text{ then } x :| \; S(t,x,x') \text{ end}$$

The fact that f is a correct refinement of e is captured by the *guard strengthening* and *simulation* proof obligations which are encoded as the following axioms: $\forall t, x \cdot x \in I \wedge t \mapsto x \in H \Rightarrow x \in G$ and $\forall t, x, x' \cdot x \in I \wedge t \mapsto x \in G \wedge t \mapsto x \mapsto x' \in R \Rightarrow t \mapsto x \mapsto x' \in S$.

A property of refinement is the preservation of invariance properties, i.e. I should be also an invariant of the concrete model, in particular maintain by the concrete event f. This can be stated and proved as a *theorem* in the context $\forall t, x, x' \cdot x \in I \wedge t \mapsto x \in H \wedge t \mapsto x \mapsto x' \in R \Rightarrow x' \in I$.

Correctness of Interface Instantiation. Using the proof method, we prove the correctness of interface instantiation as follows. Our initial machine is M as described at the end of Section 2. We decompose M into M_1 and M_5, sharing the interface U. The abstract interface U encapsulates the shared variable x_3 with invariant $I_3(x)$, and subsequently instantiated by some concrete interface V containing concrete variable y_3 as follows.

interface U	interface V instantiates U with $x_3 = h_3(y_3)$
fields x_3	fields y_3
constraints $I_3(x_3)$	constraints $J_3(y_3)$

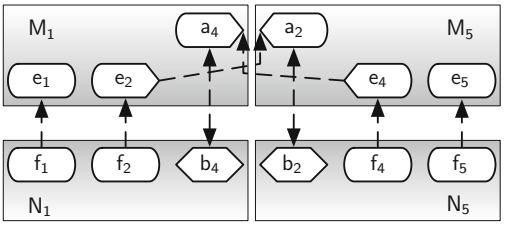


Fig. 7. Decomposition and events refinement

At the same time, M_1 and M_5 are refined into N_1 and N_5, relying on the interface instantiation. To be more precise, in N_1, internal event e_1 and e_2 are refined by f_1 and f_2, respectively. Furthermore, external event a_4 is refined *equivalently* (see Section 3) to b_4. Similarly, in N_5, f_4 and f_5 are the refinement of internal events e_4 and e_5, respectively, and b_2 is the refinement of external event a_2. The refinement relationships between the events can be depicted in Figure 7.

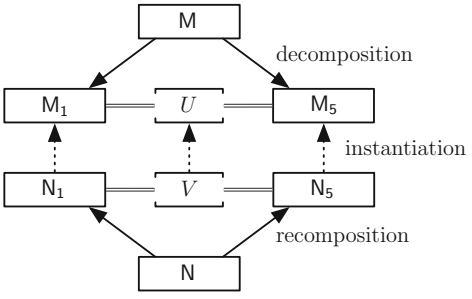


Fig. 8. Decomposition, interface instantiation, and refinement

The composition machine N comprises of internal events f_1, f_2 from N_1, and f_4 and f_5 from N_5. The summary of decomposition and interface instantiation approach is shown in Figure 8.

The correctness of our technique is guaranteed by proving that the composition N of N_1 and N_5 is indeed a refinement of the original model M, using the assumption that N_1 and N_5 refines M_1 and M_5, respectively, as described earlier. The key important aspect for the correctness of our approach is that external events are refined equivalently. This fact guarantees that the refinement relationship between a pair of internal/external events is maintained. For example, we have that the internal event f_4 is a refinement of the corresponding external event b_4.[3]

6 Conclusion

We propose in this paper the notion of interface and interface instantiation for shared-variable decomposition in Event-B. An interface is a collection of external variables and their properties which can be shared between different sub-model after a decomposition. Interface instantiation combines instantiation of carrier sets and constants with functional refinement of external variables. The encapsulation of external variables using interface offers us some flexibility in structuring the development using complex refinement and decomposition. In particular, we provide a practical method for refining external variables which is currently quite cumbersome [1].

The novelty of our approach is in the refinement of external events: we define additional proof obligations to ensure that the external events are refined *equivalently*. By contrast, in [1] equivalence is achieved by syntactical means replacing occurrences of abstract variables v by concrete terms $h(w)$. The proof obligations of our approach are similar to the standard proof obligations, even using the same refinement witnesses for proving the equivalence. We have presented a general technique for proving correctness of Event-B extensions, and showed how this is used to demonstrate the soundness of our approach. We illustrated the method by an industrial case study modelling a cruise control system.

For future work, we want to develop a theory of interface instantiation. In particular we intend to investigate the idea of having different instantiation branches of interfaces that are joined ultimately so that all machines of a model agree on their interfaces. The idea is illustrated in Fig. 9. In the figure, an abstract

[3] The development is available at `http://deploy-eprints.ecs.soton.ac.uk/364/`

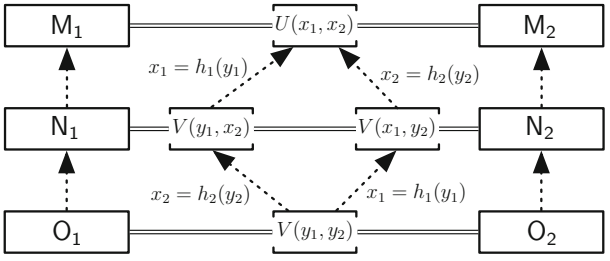


Fig. 9. A lattice of interfaces

interface U containing two abstract fields x_1 and x_2 is used by M_1 and M_2. Subsequently, U is instantiated by V_1 where x_1 is replaced by y_1 and x_2 is retained. At the same time, M_1 is refined into N_1 using V_1. Similarly, M_2 is refined by N_2 using V_2. Finally, V, an instantiation of both V_1 and V_2 can be use to refine N_1 and N_2 into O_1 and O_2, respectively. What this lattice of interfaces allows us to do is to have different order of instantiating the fields of an abstract interface in an individual sub-model. In particular, we actually abandon the compositionality of the intermediate machines (here N_1 and N_2), only to re-establish it later for the final machines O_1 and O_2, by connecting them to the same interface V.

Finally, we are looking at extending the Rodin tool [3] to support the notion of interface and interface instantiation.

References

1. Abrial, J.-R.: Event-B: Structure and Laws (2005)
2. Abrial, J.-R.: Modeling in Event-B - System and Software Engineering. Cambridge University Press (2010)
3. Abrial, J.-R., Butler, M.J., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. STTT 12(6), 447–466 (2010)
4. Evans, N., Butler, M.: A Proposal for Records in Event-B. In: Misra, J., Nipkow, T., Karakostas, G. (eds.) FM 2006. LNCS, vol. 4085, pp. 221–235. Springer, Heidelberg (2006)
5. Gmehlich, R., Grau, K., Hallerstede, S., Leuschel, M., Lösch, F., Plagge, D.: On Fitting a Formal Method into Practice. In: Qin, S., Qiu, Z. (eds.) ICFEM 2011. LNCS, vol. 6991, pp. 195–210. Springer, Heidelberg (2011)
6. Hallerstede, S.: The Event-B Proof Obligation Generator (2005)
7. Hoang, T.S., Abrial, J.-R.: Event-B Decomposition for Parallel Programs. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) ABZ 2010. LNCS, vol. 5977, pp. 319–333. Springer, Heidelberg (2010)
8. Iliasov, A., Troubitsyna, E., Laibinis, L., Romanovsky, A., Varpaaniemi, K., Ilic, D., Latvala, T.: Supporting Reuse in Event B Development: Modularisation Approach. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) ABZ 2010. LNCS, vol. 5977, pp. 174–188. Springer, Heidelberg (2010)
9. Jackson, M.: Problem Frames: Analyzing and structuring software development problems. Addison-Wesley Longman Publishing Co., Inc. (2001)
10. Ladenberger, L., Bendisposto, J., Leuschel, M.: Visualising Event-B Models with B-Motion Studio. In: Alpuente, M., Cook, B., Joubert, C. (eds.) FMICS 2009. LNCS, vol. 5825, pp. 202–204. Springer, Heidelberg (2009)

11. Loesch, F., Gmehlich, R., Grau, K., Jones, C.B., Mazzara, M.: DEPLOY Deliverable D19: Pilot Deployment in the Automotive Sector
12. Poppleton, M.R.: The Composition of Event-B Models. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) ABZ 2008. LNCS, vol. 5238, pp. 209–222. Springer, Heidelberg (2008)
13. Schneider, S., Treharne, H.: Changing system interfaces consistently: A new refinement strategy for $CSP||B$. Sci. Comput. Program. 76(10), 837–860 (2011)
14. Silva, R., Pascal, C., Hoang, T.S., Butler, M.: Decomposition tool for event-B. Softw, Pract. Exper. 41(2), 199–208 (2011)

Discharging Proof Obligations from Atelier B Using Multiple Automated Provers*

David Mentré[1], Claude Marché[2,3],
Jean-Christophe Filliâtre[3,2], and Masashi Asuka[4]

[1] Mitsubishi Electric R&D Centre Europe, Rennes, F-35708
[2] INRIA Saclay – Île-de-France, Orsay, F-91893
[3] Lab. de Recherche en Informatique, Univ Paris-Sud, CNRS, Orsay, F-91405
[4] Advanced Technology R&D Center, Mitsubishi Electric Corp., Amagasaki, Japan

Abstract. We present a method to discharge proof obligations from Atelier B using multiple SMT solvers. It is based on a faithful modeling of B's set theory into polymorphic first-order logic. We report on two case studies demonstrating a significant improvement in the ratio of obligations that are automatically discharged.

1 Introduction

The B Method [1] is a formal approach to develop safety critical embedded systems. It is mainly used in the European railway industry [2,5]. This method allows the design of correct-by-construction programs, thanks to refinement techniques. The soundness of refinement steps is expressed by logic formulas, called *proof obligations* (PO for short), that must be proved valid. The system *Atelier B* implements the B Method and provides a dedicated theorem prover. It is mostly an automated prover for B's set theory. To discharge POs that are not proved automatically, a user interface allows interactive proof steps.

In recent years, there has been tremendous progress in the domain of *Satisfiability Modulo Theories* (SMT for short). Some SMT solvers have proved powerful in the context of extended static checking, *e.g.* Simplify for ESC/Java, Z3 for Boogie, Spec#, and VCC. A natural question is whether we would gain automation by using SMT solvers on POs generated by Atelier B. This is the question we address in this paper. We propose a technique to translate B POs into the input language of Why3 [6], an environment providing a common front-end to various external provers. Why3 implements a polymorphic first-order logic, in which we axiomatize B's set theory. A main difficulty is to make sure that this axiomatization is in a suitable form for the SMT provers to solve the generated goals.

This paper is organized as follows. Sect 2 presents the necessary background regarding B and Why3. Sect 3 exposes our technique to perform the translation from B to Why3. Sect 4 reports on experiments made with our implementation. We compare with related work in Sect 5.

* This work is partly funded by the U3CAT project (ANR-08-SEGI-021, `http://frama-c.com/u3cat/`) of the French national research organization (ANR).

J. Derrick et al. (Eds.): ABZ 2012, LNCS 7316, pp. 238–251, 2012.

```
MACHINE Timer(initial_timer_value_ms)

SEES Configuration

CONSTRAINTS initial_timer_value_ms ∈ NAT1

VARIABLES   active, remaining_time

INVARIANT active ∈ 𝔹 ∧ remaining_time ∈ NAT ∧
        (active = FALSE ⇒ remaining_time = 0) ∧
        (active = TRUE ⇒ remaining_time ≤ initial_timer_value_ms)

INITIALISATION   active := FALSE || remaining_time := 0

OPERATIONS
   start_timer = PRE active = FALSE THEN
      active := TRUE || remaining_time := initial_timer_value_ms
   END;

   decrement_timer = PRE active = TRUE THEN
      remaining_time : ( remaining_time ∈ NAT ∧
         (remaining_time$0 ≥ cycle_duration
            ⇒ remaining_time = remaining_time$0 − cycle_duration) ∧
         (remaining_time$0 < cycle_duration ⇒ remaining_time = 0) )
   END;
END
```

Fig. 1. Abstract State Machine defining a timer using B Method

2 Background

2.1 The B Environment

The B Method is organized around Abstract State Machines. Each Abstract State Machine contains a state defined through variables as well as operations allowing to modify this state. One can use Booleans, integers, and set theory to express the state of an abstract machine. For example, in Fig. 1 showing a timer defined using B Method, the state is defined through Boolean variable active and natural integer variable remaining_time. The two operations start_timer and decrement_timer allow the use of this timer by updating those variables.

Correctness properties that should be fulfilled by a machine are defined in an invariant of each machine as well as in the definition of each operation. One can use first-order logic to express those properties. In Fig. 1, the **INVARIANT** clause states that if the timer is not active, the remaining_time should be zero otherwise the remaining time should be less or equal the initial timer value. In a similar way, the specification of the decrement_timer operation states that this operation recomputes the remaining_time variable. If the value of the variable at

```
  cycle_duration = 100 ∧
  (active = TRUE ⇒ remaining_time ≤ initial_timer_value_ms) ∧
  active = TRUE ∧  1 ≤ initial_timer_value_ms ∧
  remaining_time$1 ∈ ℤ ∧ 0 ≤ remaining_time$1 ∧ remaining_time$1 ≤ 2147483647 ∧
  (cycle_duration ≤ remaining_time
      ⇒ remaining_time$1 = remaining_time − cycle_duration) ∧
  (remaining_time + 1 ≤ cycle_duration ⇒ remaining_time$1 = 0)
⇒ remaining_time$1 ≤ initial_timer_value_ms
```

Fig. 2. Example of Proof Obligation

operation entry ($0 notation) is bigger than the cycle duration, then it should be decreased by the amount of cycle duration, otherwise it should be zero. Moreover, those operations are constrained by a precondition that ensures the start_timer operation is only used when the timer is inactive while the decrement_timer operation is only used when the timer is active.

Abstract State Machines are similar to formal specifications. They are transformed into an actual implementation through the use of manual refinements that lead in one or more steps to an implementation. An implementation might import one or more other machines in order to use their operations.

The B Method ensures that correctness properties defined in the invariant or the operations are kept through the refinements and up to the final implementation. This is done through the generation of POs, following patterns defined in the B-Book [1], that must be proved valid. For example, the PO shown in Fig. 2 checks that the invariant active = **TRUE** ⇒ remaining_time ≤ initial_timer_value_ms is preserved by the decrement_timer operation. The upper part of this PO describes the effect of the operation specification (here used as an assumption for this PO), while the lower part being the property to prove under active = **TRUE** assumption. The $1 notation denotes the state of the variable after execution of the operation.

Tools are available to use the B Method in an industrial context, like Atelier B made by ClearSy company. This tool contains an editor as well as automatic and interactive provers. When developing software using the B Method, the code corresponding to specifications, refinements and implementations is entered into Atelier B. Then proof obligations are automatically generated and in a second step are proved, either automatically or under user's guidance. The amount of interactive proofs is a direct cost for a project and usually corresponds to 5% to 40% of the total amount of proof obligations for industrial projects. The PO shown in Fig. 2 is not proved by the automatic prover of Atelier B.

2.2 The Why3 System

Why3 [6] is a set of tools for program verification. Basically, it is composed of two parts, which are depicted in Fig. 3: a logical language called Why with an infrastructure to translate it to existing theorem provers; and a programming

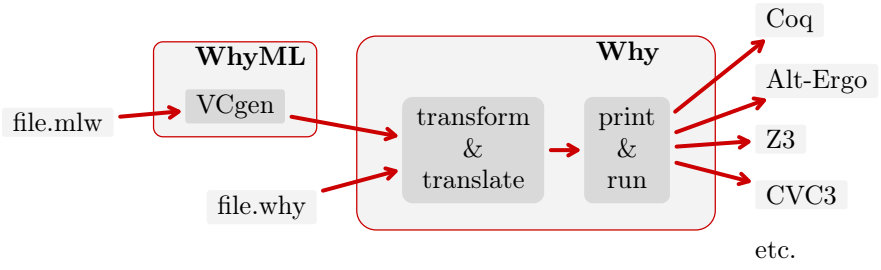


Fig. 3. Overview of Why3

```
goal g: forall active:bool,
  remaining_time remaining_time_1 initial_timer_value_ms cycle_duration : int.
  cycle_duration = 100 ∧ active = True ∧
  (active = True → remaining_time ≤ initial_timer_value_ms) ∧
  1 ≤ initial_timer_value_ms ∧
  0 ≤ remaining_time_1 ∧ remaining_time_1 ≤ 2147483647 ∧
  (cycle_duration ≤ remaining_time
      → remaining_time_1 = remaining_time − cycle_duration) ∧
  (remaining_time + 1 ≤ cycle_duration → remaining_time_1 = 0)
→ remaining_time_1 ≤ initial_timer_value_ms
```

Fig. 4. The same Proof Obligation as Fig. 2, in Why3

language called WhyML with a verification condition generator. In this paper, we are not using the programming facilities of Why3; we are only concerned with its logic, that is the right part of Fig. 3.

The logic of Why3 is a polymorphic first-order logic with recursive definitions, algebraic data types, and inductive predicates [7]. Logical declarations are organized in small units called *theories*. The purpose of Why3 is, among other things, to extract goals from theories and to translate them to the native language of external theorem provers. Such provers range from interactive proof assistants, such as Coq, to general-purpose automated theorem provers, such as Alt-Ergo, Z3, or CVC3, and even to dedicated theorem provers, such as Gappa.

Fig. 4 shows a Why3 file containing one goal, equivalent to the PO of Fig 2. Using Why3, this goal is proved valid with any of Alt-Ergo [12], Z3 [13], or CVC3 [4].

3 A Translator from B to Why3

This section details the core of our contribution: a method to translate B proof obligations into the Why3 form, so as to call the various provers available as Why3 back-end. The method is based on two components: first a modeling in Why3 of the set theory used in B (Sect. 3.1 below), second a standalone tool

```
theory Set
  type set α                                          (* abstract type for polymorphic sets *)

  predicate mem α (set α)                                                    (* membership *)

  predicate (==) (s1 s2: set α) = forall x : α. mem x s1 ↔ mem x s2      (* equality *)
  axiom extensionality: forall s1 s2: set α. s1 == s2 → s1 = s2

  predicate subset (s1 s2: set α) = forall x : α. mem x s1 → mem x s2    (* inclusion *)

  function empty : set α                                                   (* empty set *)
  axiom empty_def: forall x: α. ¬ (mem x empty)

  function union (set α) (set α) : set α                                    (* union *)
  axiom union_def: forall s1 s2: set α, x: α.
    mem x (union s1 s2) ↔ mem x s1  ∨  mem x s2
[...]
end
```

Fig. 5. Why3 theory of sets (excerpt)

that reads a B file containing proof obligations and translates it into a set of equivalent Why3 goals (Sect. 3.2). Then in Sect. 3.3 we discuss the soundness of this method.

3.1 Modeling B Set Operators as Why3 Theories

The first theory we pose is a theory of sets. An excerpt of it is shown on Fig. 5. To model the different possible types of elements, we make use of the type polymorphism of Why3, and thus declare a polymorphic type set α where the type parameter α denotes the type of elements. The type set is not defined in Why3 but only *axiomatized*. The first and essential ingredient of this axiomatization is the predicate mem which is intended to denote membership of an element in a set. Indeed, most of the other operators that we introduce afterwards are axiomatized with respect to mem, as exemplified in Fig. 5 for the (polymorphic) empty set, the union operator and the predicate subset.

In the POs generated by B, it is very common to test equality of two sets. In Why3, the built-in symbol = denotes a polymorphic equality, which is assumed to be a congruence relation on any type it is used on. However, for sets, the intended equality is not arbitrary: we want to model the fact that two sets are equal if and only if they contain the same elements (Axiom SET 4 of the B-Book [1, p. 61]). This is done by defining the predicate == of Fig. 5 just as said above, and posing an axiom of *extensionality* which states that sets that are equivalent for == are equal.

Both to exemplify our model of sets, and to define commonly used sets of integers in B, let's show how we model intervals of integers. This is done in

```
theory Interval
  use export int.Int
  use export Set

  function mk int int : set int
  axiom mem_interval: forall x a b : int. mem x (mk a b) ↔ a ≤ x ≤ b

  function integer : set int
  axiom mem_integer: forall x:int.mem x integer

  function natural : set int
  axiom mem_natural: forall x:int. mem x natural ↔ x ≥ 0
[...]
end
```

Fig. 6. Why3 theory of intervals

a new Why3 theory, importing those of sets, as shown in Fig. 6. We declare a logic function mk such that mk a b denotes the interval $[a, b]$. We also pose definitions of the B built-in sets $\mathbb{Z}$ and $\mathbb{N}$ as two constants integer and natural with appropriate axioms. We reuse Why3 computer arithmetic operators which are the same as B-Book's ones. Other set constructs are axiomatized in a similar way: relations, power sets, sequences, finite sets, etc.

We detail our model of B *relations*, as shown in Fig. 7. A relation between two sets S and T is just a set of pairs of elements of $S \times T$. Domain and range of such a relation are axiomatized with natural axioms. Partial functions in B are just particular cases of relations. The set of partial functions on some sets s and t is axiomatized in the Function theory of Fig. 7. Our axioms are designed as transcriptions of those of the B-Book [1, p. 86], independently of the case studies. We also provide a few lemmas about functions. These were added while working on the case studies. They provide a form of hint to the SMT solvers. Unlike axioms, these are logical consequences of the axiomatization. They are proved, using Why3, either automatically with SMT solvers or interactively with Coq.

The set of total functions is defined similarly. A non-trivial construct of B is function application f(x). In B, this construct is subject to the condition $x \in \mathrm{dom}(f)$ [1, p. 89]. We model this construct in Why3 using an explicit operator apply. It is axiomatized for total functions only (see the last two axioms in Fig. 7) and unspecified otherwise.

3.2 The Translation Process

Addition of the Why3 proof tool chain inside Atelier B is made after generation of proof obligations. For each B machine (specification, refinement, or implementation), Atelier B generates an internal PO file (with suffix .po). We read and translate this PO file into Why3.

```
theory Relation "Relations between two sets"
  use export Set

  type rel α β = set (α,β)

  function dom (rel α β) : set α
  axiom dom_def: forall r : rel α β, x : α. mem x (dom r) ↔ exists y : β. mem (x,y) r

  function ran (rel α β) : set β
  axiom ran_def: forall r : rel α β, y : β. mem y (ran r) ↔ exists x : α. mem (x,y) r
[...]
end

theory Function "Partial functions as relations"
  use export Relation

  function (+->) (s:set α) (t:set β) : set (rel α β)
  axiom mem_function: forall f:rel α β, s:set α, t:set β.
    mem f (s +-> t) ↔
      (forall x:α, y:β. mem (x,y) f → mem x s ∧ mem y t) ∧
      (forall x:α, y1 y2:β. mem (x,y1) f ∧ mem (x,y2) f → y1=y2)

  lemma range_function: forall f:rel α β, s:set α, t:set β, x:α, y:β.
    mem f (s +-> t) → mem (x,y) f → mem y t

  lemma function_extend_range: forall f:rel α β, s:set α, t u:set β.
    subset t u → mem f (s +-> t) → mem f (s +-> u)

  function (-->) (s:set α) (t:set β) : set (rel α β)
  axiom mem_total_functions: forall f:rel α β, s:set α, t:set β.
    mem f (s --> t) ↔ mem f (s +-> t) ∧ dom f == s

  lemma total_function_is_function: forall f:rel α β, s:set α, t:set β.
    mem f (s --> t) → mem f (s +-> t)

  function apply (rel α β) α : β
  axiom apply_def1: forall f:rel α β, s:set α, t:set β, a:α.
    mem a s ∧ mem f (s --> t) → mem (a, apply f a) f
  axiom apply_def2: forall f:rel α β, s:set α, t:set β, a:α, b:β.
    mem f (s --> t) ∧ mem (a,b) f → b = apply f a
[...]
end
```

Fig. 7. Why3 theory of relations and functions (excerpt)

Our bpo2why translator is made of three steps: the parsing of Atelier B's PO file into an abstract syntax tree, the application of a type inference algorithm on the read tree, and finally the translation of the typed tree into Why3.

```
THEORY ProofList IS
  _f(1) ∧ _f(2) ∧ _f(6) ∧ decrement_timer.2,(_f(10) ⇒ _f(11));
[...]
END
∧
THEORY Formulas IS
1  ("'Component constraints'" ∧ initial_timer_value_ms ∈ ℤ ∧
    0 ≤ initial_timer_value_ms ∧ initial_timer_value_ms ≤ 2147483647 ∧
    ¬(initial_timer_value_ms = 0) ∧ cycle_duration = 100;
2  ("'Component invariant'" ∧ active ∈ 𝔹 ∧ remaining_time ∈ ℤ ∧
    0 ≤ remaining_time ∧ remaining_time ≤ 2147483647 ∧
    (active = FALSE ⇒ remaining_time = 0) ∧
    (active = TRUE ⇒ remaining_time ≤ initial_timer_value_ms));
[...]
6  ("'decrement_timer preconditions in this component'" ∧ active = TRUE);
[...]
10  ("'Local hypotheses'" ∧ remaining_time$1 ∈ ℤ ∧ 0 ≤ remaining_time$1 ∧
    remaining_time$1 ≤ 2147483647 ∧
    (cycle_duration ≤ remaining_time ⇒
          remaining_time$1 = remaining_time − cycle_duration) ∧
    (remaining_time + 1 ≤ cycle_duration ⇒ remaining_time$1 = 0));
11  (remaining_time$1 ≤ initial_timer_value_ms)
END
∧
THEORY EnumerateX IS
  t_BOOM_MOVEMENT_ORDER = {go_up, go_down}
END
```

Fig. 8. Part of proof obligation file generated for Timer machine

The parsing step is quite usual. The format of the PO file is not publicly documented but it is generated as a text file and we have reverse-engineered it. Fig. 8 shows part of the generated PO file for the Timer machine of Fig. 1. This file contains three parts: a set of logic expressions to prove (ProofList part), a set of formulas identified by their sequence number (Formulas part) and referred as _f(n) in previous logic expressions, and a set of enumerated sets (EnumerateX part). We build an abstract syntax tree from the content of this file, using the same priority and associativity as B's operators [9]. As the B syntax is quite big (about 200 keywords and operators), we currently do not parse all of it but a significant subset[1] needed for our tests.

The type inference step decorates the abstract syntax tree with the B type of all operators and identifiers. It is necessary for a precise translation in the

[1] This subset includes ∃ and ∀ quantifiers, Boolean expressions (with ⇒, ⇔, ∧, ∨, ¬ connectors and bool operator), usual integer arithmetic expressions (+, −, ∗, / and *mod* operators, <, ≤, ≥, > comparison operators, 32 bits constants), set expressions (with ℙ, a..b, ∈, ∗, ∩, ∪ and − set operators), ℤ, ℕ and ∅ sets, operators on functions and relations (including seq, f^{-1}, ↔, ⇸, →, f[s], f(x), dom, ran, size).

```
theory B_translation
  use import bool.Bool
  use import int.Int
  use import bpo2why_prelude.Interval
[...]
  type enum_t_BOOM_MOVEMENT_ORDER = E_go_up | E_go_down
[...]
  predicate f1 (v_remaining_time_1: int) (v_remaining_time: int)
               (v_initial_timer_value_ms: int) (v_cycle_duration: int) (v_active: bool) =
    ((((mem v_initial_timer_value_ms integer)) ∧ (0 ≤ v_initial_timer_value_ms))
    ∧ (v_initial_timer_value_ms ≤ 2147483647))
    ∧ ¬(v_initial_timer_value_ms = 0)) ∧ (v_cycle_duration = 100))

  predicate f2 [...] = ((((((mem v_remaining_time integer)) ∧ (0 ≤ v_remaining_time))
    ∧ (v_remaining_time ≤ 2147483647))
    ∧ ((v_active = False) → (v_remaining_time = 0)))
    ∧ ((v_active = True) → (v_remaining_time ≤ v_initial_timer_value_ms)))
[...]
  predicate f6 [...] = (v_active = True)
[...]
  predicate f10 [...] = ((((mem v_remaining_time_1 integer))
    ∧ (0 ≤ v_remaining_time_1)) ∧ (v_remaining_time_1 ≤ 2147483647))
    ∧ ((v_cycle_duration ≤ v_remaining_time)
        → (v_remaining_time_1 = (v_remaining_time − v_cycle_duration))))
    ∧ (((v_remaining_time + 1) ≤ v_cycle_duration) → (v_remaining_time_1 = 0)))

  predicate f11 [...] = (v_remaining_time_1 ≤ v_initial_timer_value_ms)

  goal decrement_timer_2 :
    forall v_remaining_time_1: int, v_remaining_time: int,
      v_initial_timer_value_ms: int, v_cycle_duration: int, v_active: bool.
    ((f1 v_remaining_time_1 v_remaining_time v_initial_timer_value_ms v_cycle_duration v_active)
     ∧ (f2 [...]) ∧ (f6 [...]) ∧ (f10 [...]))
    →
    (f11 [...])
[...]
end
```

Fig. 9. Why3 translation of Timer proof obligation

next step. We use a classical Hindley-Milner type inference algorithm [16]. An additional issue is to support operator overloading, *e.g.* "$*$" which is both the arithmetic multiplication and the Cartesian product of two sets.

In a third step, we translate the typed abstract syntax tree into a Why3 file. This is done through a top-down traversal of the tree, translating each node into Why3 syntax and then recursively translating sub-trees of this node. This translation step uses the Why3 theories of B operators defined in Section 3.1. In case operators would have several possible translations, we use the inferred type in previous step to determine the kind of Why3 operator to use. For example, the "=" B's operator is translated into Why3's "=" if it is an integer equality or into Why3's "==" operator if it is a set equality. Enumerated sets are translated into Why3's sum types. All B's expressions in a PO file are translated, except two kinds related to enumerated sets (an enumerated set is not empty and an enumerated set is finite) as those assumptions are implicitly guaranteed by Why3's sum types. Fig. 9 shows the PO file of Fig.8 translated into Why3. We have kept the same structure as the input file, with the definition of "fn" predicates and their use in a Why3's "**goal**". All predicates are quantified over all variables used

in the PO file. The t_BOOM_MOVEMENT_ORDER enumerated set is translated into a sum type. We have used an explicit parenthesizing of expressions to avoid any priority issue. We keep the PO comments produced by Atelierb B as Why3 labels. Thanks to our modeling of B operators, we are able to translate set related expressions. For example in predicate f2 of Fig. 9, we translate the PO expression "remaining_time $\in \mathbb{Z}$" into "mem v_remaining_time integer", using the mem set operator defined in Sect. 3.1. In the same way, the symbol "integer" is the one of Fig. 6.

By default, we generate a Why3 file for each original PO file. However, when a PO file contains more than 200 proof obligations, we split the generated Why3 file into several files, each one containing at most 200 goals. We also include in those files only the "f*n*" predicates needed by goals of a given Why3 file. This approach reduces the processing time and proof context of Why3 under acceptable limits, as well as the time needed to call provers. Otherwise a single Why3 file with 1,600 goals and 1,400 predicates would take several minutes to simply load the file.

3.3 Soundness of the Translation

We claim that our translation process is sound in the sense that if the translation of a B proof obligation is a valid formula then the original one is also valid. That soundness property relies upon two things: first the modeling of B operators as presented in Sect. 3.1 must be faithful to the B-Book, second the translation mechanism given in Sect. 3.2 must be sound. Both of these ingredients are small and natural, so we are confident on their soundness. The modeling contains 3 type declarations, 35 function symbols, 5 predicate symbols, 25 axioms, and 21 lemmas[2]. The bpo2why translator is made of 2,057 lines of OCaml: 701 lines for parsing, 957 lines for type inference, and 399 lines for the translation.

However, in such a process it is easy to make a mistake when writing down axioms, which could result in an inconsistent theory in which we could prove anything. To prevent from such an inconsistency, we designed Coq *realizations* of the Why3 theories in use. Realizing theories in Coq is a feature provided by Why3. It automatically translates a given Why3 theory into a Coq module, where each abstract definition or axiom is respectively written as a concrete definition or a lemma. The latter must then be filled in by the user.

The first step is to provide a Coq definition of the type of polymorphic sets. We use the higher-order features of Coq, and define set α as a function $\alpha \rightarrow$ bool, that is a set S of elements of type α is identified with its characteristic function. The membership function is thus defined trivially as (mem x s) := (s x). From such a definition, it is straightforward to define the basic set operators empty set, union, etc. and prove that the axioms we pose are valid. However, realizing our set equality and our extensionality axiom is not an easy task. It is indeed not provable in Coq that s1==s2 implies s1=s2: pointwise equality of functions

[2] We modeled only the B constructs needed for our case studies.

does not imply equality of these functions, it is the so-called extensionality of function equality.

Thus, we pose functional extensionality as an axiom in Coq. Actually functional extensionality is not the only axiom we need. We also admit the excluded middle, because we need to have decidability of membership in a set, and finally we admit the axiom of choice to be able to realize the apply operator, which allows to construct a function from a relation. It is commonly admitted that adding these general-purpose axioms in the Coq calculus of inductive constructions is consistent, indeed by interpretation into a standard set-theoretic boolean model [3].

4 Experiments

We applied our technique on a proprietary use case called RCS3. This is a B project modeling the software controlling a railway level crossing system. This project has been entirely proved inside Atelier B, so all proof obligations are valid. While being a small project (about 3,000 lines of generated C code), it is representative of a B development with sets, sets of sets, relations, sequences, and linear integer arithmetic. The project is made of 31 machines (specification, refinement, or implementation), generating 2,247 proof obligations. Atelier B 4.0.2 automatic prover in F1 force proves 94% of them using a 10 seconds time limit, leaving 129 unproved proof obligations.

Our bpo2why translator can be applied on all generated PO files. We can then launch the Why3 tool chain on them using Alt-Ergo, CVC3, and Z3 provers. We use the following strategy to run the provers: the three provers are launched in parallel on all proof obligations, four at a time, with a 2 seconds time limit. For remaining unproved goals, we run once again the three provers with a 60 seconds time limit.

The comparison of the two proof chains is given in Fig. 10 (only machines generating proof obligations are shown). Overall, the Why3 proof tool chain proves more proof obligations than Atelier B's automatic prover (including the Timer machine previously presented). Only 19 proof obligations are not proved, corresponding to a 85% improvement. In only one machine, Automaton_context_i, the Why3 tool chain proves less proof obligations than Atelier B. This machine contains set expressions between enumerated sets. We do not know yet why such expressions are difficult for our tool chain. The 10 proof obligations in Warning_section_i machine are considered "difficult" ones. They need an elaborated mathematical proof with exhibition of witnesses (for existential quantifiers) based on properties of a bijection.

An interesting by-product of this experiment is that none of Alt-Ergo, CVC3, and Z3 automatic provers proves all proof obligations, even with a 60s time limit. For the three provers, there is at least one proof obligation which is proved by this prover and by none of the others. This result confirms the usefulness of the Why3 tool chain that targets several provers and thus allows to use them in a complementary way.

Machine	# of PO	Unproved by Atelier B	Unproved by Why3
Automaton	4	0	0
Automaton_context_i	10	**8**	9
Automaton_i	229	71	**0**
Automaton_transitions	189	7	**0**
Automaton_transitions_i	1678	25	**0**
Boom_detectors_i	16	0	0
Configuration_i	7	4	**0**
Indicators_i	12	0	0
Lamps_bells_i	4	0	0
Timer	3	1	**0**
Timer_i	10	0	0
Track_circuit	2	0	0
Track_circuit_i	1	0	0
Train_detector_i	4	0	0
Warning_section	2	0	0
Warning_section_i	59	11	**10**
Warning_section_r	17	2	**0**
Total	2247	129	**19**

Fig. 10. Comparison of Why3 tool chain with Atelier B on RCS3 use case (smaller is better)

Regarding proving time, the Why3 tool chain takes 35 min 34 s to prove all goals with the three provers using 4 cores, roughly 12 min per prover. Using F1 proving force, automatic prover of Atelier B 4.0 on one core[3] proves its proof obligations in 1 min 2 s. Using F3 force, we do not get any answer from Atelier B in 30 minutes. Discarding machine Automaton_context_i, it completes in 7 min 5 s. There is net gain of 2 proof obligations in machine Warning_section_r. Overall, Atelier B is much faster to prove the proof obligations, but Why3 produces a better result in an acceptable time. As the time needed by the user to look at unproved proof obligation is very costly, we think that any gain in automatic proofs is an effective development time gain.

Digital Watch Example. We have also applied our tool on a second example, the model of a digital watch. This model is less complex. It generates 777 proof obligations, of which 11 are not proved by Atelier B in F1 force. Using our translator and then the Why3 tool chain with Alt-Ergo, CVC3 and Z3, we can automatically prove all but one proof obligation, the remaining one being not provable (a bug in the original model). This result confirms that the Why3 tool chain improves the efficiency of proofs by exploiting the capabilities of modern SMT provers (this model contains a lot of integer arithmetic expressions).

[3] Latest Atelier B 4.0.2 is able to use all cores of a multi-core machine but we could not use this version for our tests.

5 Comparison with Related Work

Bodeveix, Filali, and Muñoz [8] formalized the semantics of B in both Coq and PVS. They define a (mostly) shallow embedding of the B notions of generalized substitutions and machines. B's set theory is not formalized at all; the native logics of Coq and PVS are used instead.

The BRILLANT [11] toolset made by Colin et al. generates B's proof obligation that can be incorporated inside the Coq proof assistant thanks to the BiCoax [10] libraries. (The BiCoax work is itself an extension of the B/PhoX [17] work based on PhoX proof assistant.) The proof obligations can then be proved manually or by Coq automatic tactics. Jacquel et al. [15] propose another deep embedding of B's set theory in Coq, whose purpose is to check using Coq that the rewrite rules used in the B prover are valid. Our Coq realization is similar to both Coq formalizations above. However our Coq model is only built for the purpose of showing the consistency, not for the purpose of performing proofs interactively with Coq.

Déharbe made a work [14] very similar to ours. Namely, Déharbe interfaces SMT solvers having an SMT-LIB interface with the Rodin development tool for Event-B. The proof obligations generated by Rodin are transformed into Boolean formulas, sets being transformed into their characteristic predicate. Déharbe's approach is limited to basic sets (*i.e.* no set of sets) while ours is able to transform all set-related expressions of the B Method. Moreover, Why3 is able to interface itself to more automatic provers, not limited by the SMT-LIB interface. For his tests, Déharbe used only one SMT solver, veriT. But even using one solver, he obtained a significant improvement in proofs, as we did.

6 Conclusion and Perspectives

In this paper, we have presented an approach and a tool to transform Atelier B's proof obligations into the Why3 proof tool chain in order to prove them using several automated provers. While being a shallow embedding of B logic into Why3 logic, we have arguments to believe that this translation is sound: mainly the translation is short and we can check axioms' correctness through Coq realization. We have applied this approach on a small but reasonably complex use case and we found a significant improvement in the number of proof obligations that are automatically proved.

This work could be improved in several ways. First of all, we could support more B operators in order to handle more complex and industrial models. The current subset of operators is the one needed to handle our use cases. Adding one B operator amounts to incrementally complete the Why3 theories, complete its Coq realization, and add a translation rule in the translator. Secondly we could try to increase the number of automatically proved proof obligations by analyzing in detail why some of them are not proved. This may amount to provide more lemmas as hints, or annotate them with *triggers*. Thirdly, we could increase our confidence level in the embedding of B into Why3 by proving B-Book's lemmas

into our Why3 framework. Fourthly we could better integrate our tool chain into Atelier B tool, for example by applying it after Atelier B automatic prover and then merging our results into Atelier B GUI. Last but not least, we could try to improve the automated provers themselves in order to better handle proof obligations generated by the B Method. E.g. an interesting theoretical question is whether the rewriting techniques used by the B prover could be combined with the satisfiability modulo theory approach.

References

1. Abrial, J.-R.: The B-Book: Assigning programs to meanings. Cambridge University Press (1996)
2. Badeau, F., Amelot, A.: Using B as a High Level Programming Language in an Industrial Project: Roissy VAL. In: Treharne, H., King, S., C. Henson, M., Schneider, S. (eds.) ZB 2005. LNCS, vol. 3455, pp. 334–354. Springer, Heidelberg (2005)
3. Barras, B.: Sets in Coq, Coq in sets. Journal of Formalized Reasoning (2010)
4. Barrett, C.W., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007)
5. Behm, P., Benoit, P., Faivre, A., Meynadier, J.-M.: Météor: A Successful Application of B in a Large Project. In: Wing, J.M., Woodcock, J. (eds.) FM 1999. LNCS, vol. 1708, pp. 369–387. Springer, Heidelberg (1999)
6. Bobot, F., Filliâtre, J.-C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Workshop on Intermediate Verification Languages (2011)
7. Bobot, F., Paskevich, A.: Expressing Polymorphic Types in a Many-Sorted Language. In: Tinelli, C., Sofronie-Stokkermans, V. (eds.) FroCos 2011. LNCS, vol. 6989, pp. 87–102. Springer, Heidelberg (2011)
8. Bodeveix, J.-P., Filali, M., Muñoz, C.: A formalization of the B-method in Coq and PVS. In: B-User Group Meeting, Formal Methods, pp. 33–49 (1999)
9. ClearSy. Language Keywords and Operators, 1.8.5 edn., `http://www.tools.clearsy.com/images/3/33/Symboles_en.pdf`
10. Colin, S., Mariano, G.: Coq, l'alpha et l'omega de la preuve pour B ? (February 2009), `http://hal.archives-ouvertes.fr/hal-00361302/PDF/bicoax.pdf`
11. Colin, S., Petit, D., Mariano, G., Poirriez, V.: BRILLANT: an open source platform for B. In: Workshop on Tool Building in Formal Methods (February 2010)
12. Conchon, S., Contejean, E., Kanig, J., Lescuyer, S.: CC(X): Semantic combination of congruence closure with solvable theories. ENTCS 198(2), 51–69 (2008)
13. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
14. Déharbe, D.: Integration of SMT-solvers in B and Event-B development environments. Science of Computer Programming (2011)
15. Jacquel, M., Berkani, K., Delahaye, D., Dubois, C.: Verifying B Proof Rules Using Deep Embedding and Automated Theorem Proving. In: Barthe, G., Pardo, A., Schneider, G. (eds.) SEFM 2011. LNCS, vol. 7041, pp. 253–268. Springer, Heidelberg (2011)
16. Milner, R.: A theory of type polymorphism in programming. Journal of Computer and System Sciences 17 (1978)
17. Rocheteau, J., Colin, S., Mariano, G., Poirriez, V.: Évaluation de l'extensibilité de PhoX: B/PhoX un assistant de preuves pour B. In: JFLA, pp. 139–153 (2004)

A Semantic Analysis of Logics That Cope with Partial Terms

Cliff B. Jones, Matthew J. Lovert, and L. Jason Steggles

School of Computing Science, Newcastle University, NE1 7RU, UK
{cliff.jones,matthew.lovert,l.j.steggles}@ncl.ac.uk

Abstract. Specifications of programs frequently involve operators and functions that are not defined over all of their (syntactic) domains. Proofs about specifications –and those to discharge proof obligations that arise in justifying steps of design– must be based on formal rules. Since classical logic deals only with defined values, some extra thought is required. There are several ways of handling terms that can fail to denote a value — this paper provides a semantically based comparison of three of the best known approaches. In addition, some pointers are given to further alternatives.

1 Introduction

This paper provides a semantic basis for terms that can fail to denote values and uses it to compare three approaches to logics for reasoning about such terms. Terms such as the head of an empty sequence (**hd** [ ]), applying a mapping outside its actual domain ($\{1 \mapsto 1\}(2)$), or even the obvious $7/0$ can be considered to fail to denote values. Of course, it would be perverse to write such naked terms deliberately but the fact is that they arise as sub-terms of quite innocent expressions. What some people call "undefined terms" are ubiquitous in reasoning about realistic program specifications and designs.

In some uses, it is tempting to try to "guard" dangerous applications by writing expressions such as:[1]

$$\forall i{:}\,\mathbb{Z} \cdot i \neq 0 \;\Rightarrow\; i/i = 1 \tag{1}$$

But there are other expressions that cannot be rewritten with such guards; consider:

$$\forall i{:}\,\mathbb{Z} \cdot (i/i = 1) \vee ((i-1)/(i-1) = 1) \tag{2}$$

Although also verging on the contrived, disjunctions where either term can be undefined –but only in the case where the other disjunct is true– arise quite naturally in specifications. The same can be said of conditions under which conjunctions and implications come into contact with "undefinedness".

[1] Assume that x/y represents integer division and that it does not yield a defined result with a zero divisor.

J. Derrick et al. (Eds.): ABZ 2012, LNCS 7316, pp. 252–265, 2012.

The issue of reasoning about such partial terms in program development has long been recognised; certainly [1] discusses the problem and the issue has since been tackled in a variety of approaches [2–19]. The topic is discussed by logicians such as in [20–27].

The first author of the current paper has long advocated the use of a non-classical "Logic of Partial Functions" (LPF) [28]. LPF is a first order predicate logic *designed* to handle non-denoting values that can arise from terms that apply partial functions and operators. LPF underlies the *Vienna Development Method* (*VDM*) [11,16,29]. A soundness proof of untyped LPF is given in [4] and of the typed version in [30].

Recently, all three authors have been looking at the issue of providing (efficient) mechanisations of LPF. One fruit of this is [31] that presents two semantic models for LPF. Also a paper on the adaptation of (semi-)decision procedures such as resolution and refutation to cope with LPF has been submitted for publication and is available as a technical report [32]. The underpinning of that research is a semantics that maps logical expressions to relations over interpretations and results. This nicely captures Blamey's [33,34] view that non-denoting terms correspond to "gaps": so $7/0$ or the head of an empty sequence map to an empty relation but i/i maps to interpretations which have a gap for $i = 0$.

In spite of the fact that the "gap" view is key to the semantic models presented later, it is convenient to first *illustrate* the three main approaches to handling partial terms being considered in this paper by using a surrogate for the "undefined" value (which, of course, can often not be computed). In these illustrations, $\perp_{\mathbb{Z}}$ is written to stand for a missing integer value and $\perp_{\mathbb{B}}$ for a missing Boolean value; then $\mathbb{B}_{\perp}$ ($\mathbb{Z}_{\perp}$) is taken to mean $\mathbb{B} \cup \{\perp_{\mathbb{B}}\}$ ($\mathbb{Z} \cup \{\perp_{\mathbb{Z}}\}$) respectively.

Essentially, the first two approaches below attempt to get by with classical logic by "catching undefinedness" before it collides with the logical operators to avoid them having any contact with non-denoting logical values. In other words providing work-arounds so that a classical (total) framework can still be used. The third approach considers using a non-classical (three-valued) logic.

The first approach (see Sect. 3) is to insist that all terms do in fact denote something (perhaps $0/0 = 42$) and is pictured in Fig. 1(a). Another approach (see Sect. 4) is to accept that terms such as i/i can fail to denote but to make any predicates (e.g. the relational operators) denote, even where their arguments fail to denote; this approach is pictured in Fig. 1(b) — existential equality $=_{\exists}$ is defined in Sect. 4.

The third approach uses a non-classical logic, notably LPF (see Sect. 5), whose attempt to "catch undefinedness" is pictured in Fig. 1(c). Here the gaps from partial terms are allowed to propagate up so that the problem *can* be "resolved" by the logical operators. Although the conditional operators of [1] do not retain properties like the commutativity of disjunctions and conjunctions, they broadly fit the picture depicted for LPF and this approach is discussed in Sect. 6.

A semantic model of the sort first presented in [31] for LPF is in fact quite convenient for comparing different approaches to handling partial terms and this is the focus of this paper. Using the definitions and ideas introduced in Sect. 2, a

$$\forall i{:}\mathbb{Z} \cdot (\overbrace{i/i}^{\in\mathbb{Z}} = 1) \vee (\overbrace{(i-1)/(i-1)}^{\in\mathbb{Z}} = 1) \quad \text{(a)}$$

$$\forall i{:}\mathbb{Z} \cdot \overbrace{(\underbrace{i/i}_{\in\mathbb{Z}_\perp} =_\exists 1) \vee (\underbrace{(i-1)/(i-1)}_{\in\mathbb{Z}_\perp} =_\exists 1)}^{\in\mathbb{B}} \quad \text{(b)}$$

$$\forall i{:}\mathbb{Z} \cdot \overbrace{\underbrace{(\underbrace{i/i}_{\in\mathbb{Z}_\perp} = 1)}_{\in\mathbb{B}_\perp} \vee \underbrace{(\underbrace{(i-1)/(i-1)}_{\in\mathbb{Z}_\perp} = 1)}_{\in\mathbb{B}_\perp}}^{\in\mathbb{B}} \quad \text{(c)}$$

Fig. 1. An illustration of where "undefinedness" can be caught: (a) a classical approach insisting that all terms denote; (b) a non-strict relational operator approach; and (c) the LPF approach

semantic model is developed for each of the three approaches to handling partial terms described above (see Sect. 3–5). These models are then used to compare and contrast the three approaches in Sect. 6, where further approaches of interest are also mentioned.

2 The Basis of the Semantics

An abstract syntax (using VDM notation [11]) is presented in Fig. 2. It is this abstract syntax that is used in the semantic models presented in this paper (although, when writing expressions in examples, concrete syntax is used for readability).

As in most logic textbooks, only a few logical operators are considered since further logical operators can be defined from this subset — and a logic is unlikely to be usable unless its operators enjoy connections such as de Morgan's laws.

$$\begin{array}{l}
Expr = Value \mid Id \mid Arith \mid Equality \mid Not \mid Or \mid Exists \\
\\
\begin{array}{ll}
Value = \mathbb{B} \mid \mathbb{Z} & Not :: \ a : Expr \\
\\
Id = Prop \mid Var & Or :: \ a : Expr \\
 & \qquad b : Expr \\
\\
Arith :: \ a \ : Expr & Exists :: \ bind : Id \\
\qquad op : - \mid / & \qquad body : Expr \\
\qquad b \ : Expr & \\
\\
Equality :: \ a : Expr & \\
\qquad b : Expr &
\end{array}
\end{array}$$

Fig. 2. The abstract syntax of the language

One predicate –equality, defined only for integer operands– and two functions –subtraction and division– suffice to illustrate the issues. Finally, quantification is only considered to be over the set of integer values and the only constant values are Booleans and integers.

Context conditions for such a language are outlined in [31] and spelt out formally in [35]. The context conditions ensure that the semantics only need be given for expressions that are well-formed thus removing the need to define semantics for ill-formed expressions such as *mk-Exists*$(x, 5)$, i.e. $\exists x \cdot 5$.

Two sorts of identifiers can occur in expressions, those for propositions (*Prop*) and those for integer variables (*Var*). The sets *Prop* and *Var* are assumed to be disjoint. It is one of the functions of the context conditions to ensure that identifiers are used appropriately. Furthermore, it is required that all integer variables are explicitly bound by quantifiers.

States ($\sigma \in \Sigma$) provide a (possibly partial) interpretation for propositional and integer variable symbols. Formally, Σ is defined as the union of two sets of maps:

$$\Sigma = \mathit{Prop} \xrightarrow{m} \mathbb{B} \mid \mathit{Var} \xrightarrow{m} \mathbb{Z}$$

where the map involving *Prop* is partial in the sense that a propositional identifier can be absent from the domain of a specific map ($\sigma \in \Sigma$) to allow for the possibility of undefined propositional identifiers. However, the *Var* map must be total since all *Var* are explicitly bound by quantifiers and in classical logic and in LPF quantification is only over a set of defined values.

The semantics is given for each of the three approaches to handling partial terms by defining a semantic function for each with the following form: $\mathcal{E}$: *Expr* $\rightarrow \mathcal{P}(\Sigma \times \mathit{Value})$.

3 Classical Logic: Making All Terms Denote

As indicated in Fig. 1(a), it is possible to get by with classical logical operators by forcing an extension of functions and operators so that they are total. To make division yield a result with a zero divisor is a challenge but it is possible to say that 7/0 yields some arbitrary integer and that perhaps no harm is done by this fiction providing that it is not possible to know which integer results. Figure 3 presents a formal semantics for this approach where division by zero is extended to return an arbitrary integer. The rest of this definition is straightforward in the sense that any feature of the language of Fig. 2 has the obvious classical meaning.

To ensure that all propositional variables do denote, the set of variable state mappings needs to be appropriately defined. Let Σ^C be the set of mappings that contain denotations for all used elements of *Prop* and *Var*:

$$\Sigma^C = \{\sigma \mid \sigma \in \Sigma \wedge \mathbf{dom}\, \sigma = \mathit{Id}\}$$

Relations are chosen as the space of denotations to facilitate comparison with the semantics in the next two sections.

$$\mathcal{E}^C : \mathit{Expr} \to \mathcal{P}(\Sigma^C \times \mathit{Value})$$

$$\begin{array}{l}
\mathcal{E}^C(e) \ \triangleq \\
\quad \textbf{cases } e \textbf{ of} \\
\quad e \in \mathit{Value} \to \{(\sigma, e) \mid \sigma \in \Sigma^C\} \\
\quad e \in \mathit{Prop} \to \{(\sigma, \sigma(e)) \mid \sigma \in \Sigma^C\} \\
\quad e \in \mathit{Var} \to \{(\sigma, \sigma(e)) \mid \sigma \in \Sigma^C\} \\
\quad \mathit{mk\text{-}Arith}(a, -, b) \to \{(\sigma, a' - b') \mid (\sigma, a') \in \mathcal{E}(a) \wedge (\sigma, b') \in \mathcal{E}(b)\} \\
\quad \mathit{mk\text{-}Arith}(a, /, b) \to \{(\sigma, a'/b') \mid (\sigma, a') \in \mathcal{E}^C(a) \wedge \\
\qquad\qquad (\sigma, b') \in \mathcal{E}^C(b) \wedge b' \neq 0\} \cup \\
\qquad\quad \{(\sigma, n) \mid (\sigma, a') \in \mathcal{E}^C(a) \wedge \\
\qquad\qquad (\sigma, b') \in \mathcal{E}^C(b) \wedge b' = 0 \wedge n \in \mathbb{Z}\} \\
\quad \mathit{mk\text{-}Equality}(a, b) \to \{(\sigma, a' = b') \mid (\sigma, a') \in \mathcal{E}^C(a) \wedge (\sigma, b') \in \mathcal{E}^C(b)\} \\
\quad \mathit{mk\text{-}Not}(p) \to \{(\sigma, \neg\, p') \mid (\sigma, p') \in \mathcal{E}^C(p)\} \\
\quad \mathit{mk\text{-}Or}(p, q) \to \{(\sigma, p' \vee q') \mid (\sigma, p') \in \mathcal{E}^C(p) \wedge (\sigma, q') \in \mathcal{E}^C(q)\} \\
\quad \mathit{mk\text{-}Exists}(x, p) \to \{(\sigma, \exists i{:}\mathbb{Z} \cdot (\sigma \dagger \{x \mapsto i\}, \textbf{true}) \in \mathcal{E}^C(p)) \mid \sigma \in \Sigma^C\} \\
\quad \textbf{end}
\end{array}$$

Fig. 3. The semantic function $\mathcal{E}^C$ — an approach to making all terms denote

Notice that this approach is total as the definition of $\mathcal{E}^C$ avoids the possibility of "gaps". In other words, for every expression e and each $\sigma \in \Sigma^C$ there exists a tuple $(\sigma, v) \in \mathcal{E}^C(e)$. This is straightforward to prove by structural induction over *Expr*. The relation, however, is not deterministic (or "functional") since it is not single-valued, i.e. $7/0 = 7/0$ can yield both true and false.

What has been done in $\mathcal{E}^C$ is to underspecify the partial division function so that it returns an arbitrary value when applied outside of its actual defined domain. An alternative approach is to overspecify the result, in other words, to define that a partial function must return a default value when applied outside of its actual defined domain, e.g. $i/0$ returns 42.[2]

The $\mathcal{E}^D$ semantics presented in Fig. 4 documents the small change needed to overspecify the partial division function. The rest of the expression cases follow as in the $\mathcal{E}^C$ semantics, if all other occurrences of $\mathcal{E}^C$ are replaced with $\mathcal{E}^D$.

It is straightforward to show the semantic function $\mathcal{E}^D$ is total and also deterministic (for any expression e it follows that $(\sigma, v_1) \in \mathcal{E}^D(e) \wedge (\sigma, v_2) \in \mathcal{E}^D(e) \Rightarrow v_1 = v_2$).

4 Classical Logic: Variant Relational Operators

Figure 1(b) indicates that there is another way to preserve the classical logic operators and that is by having non-strict relational operators denote even when their arguments fail to denote. Non-strict notions of equality include *existential equality* ($=_\exists$) and *strong equality* ($==$). Existential equality returns false when either of its operands do not denote. Strong equality differs only in the case when

[2] The answer to ... everything: *Hitchhiker's guide to the Galaxy* Douglas Adams.

$$
\begin{array}{l}
\mathcal{E}^D : \mathit{Expr} \to \mathcal{P}(\Sigma^C \times \mathit{Value}) \\
\mathcal{E}^D(e) \ \triangleq \\
\quad \textbf{cases } e \textbf{ of} \\
\quad \vdots \\
\quad \mathit{mk\text{-}Arith}(a, /, b) \to \{(\sigma, a'/b') \mid (\sigma, a') \in \mathcal{E}^D(a) \wedge \\
\qquad\qquad\qquad\qquad (\sigma, b') \in \mathcal{E}^D(b) \wedge b' \neq 0\} \cup \\
\qquad\qquad\qquad \{(\sigma, 42) \mid (\sigma, a') \in \mathcal{E}^D(a) \wedge \\
\qquad\qquad\qquad\qquad (\sigma, b') \in \mathcal{E}^D(b) \wedge b' = 0\} \\
\quad \vdots \\
\quad \textbf{end}
\end{array}
$$

Fig. 4. The semantic function $\mathcal{E}^D$ — another approach to making all terms denote

both of its operands do not denote, that is, $\perp_{\mathbb{Z}} =_{\exists} \perp_{\mathbb{Z}}$ is false but $\perp_{\mathbb{Z}} == \perp_{\mathbb{Z}}$ is true. Existential equality is the focus throughout this section.

The semantic function $\mathcal{E}^{\exists}$ is defined in Fig. 5 using a similar approach to $\mathcal{E}^C$ but replacing the case for division by the normal partial division definition and by replacing the case for equality by existential equality. Additionally any further use of $\mathcal{E}^C$ needs to be replaced with $\mathcal{E}^{\exists}$. Note that the set of variable state mappings remains as Σ^C since propositional variables are not permitted to be a source of non-denoting terms.

The $\mathcal{E}^{\exists}$ semantics is total in the sense that for every Boolean expression e and each $\sigma \in \Sigma^C$ there must be a tuple $(\sigma, v) \in \mathcal{E}^{\exists}(e)$. The $\mathcal{E}^{\exists}$ semantics is also deterministic.

$$
\begin{array}{l}
\mathcal{E}^{\exists} : \mathit{Expr} \to \mathcal{P}(\Sigma^C \times \mathit{Value}) \\
\mathcal{E}^{\exists}(e) \ \triangleq \\
\quad \textbf{cases } e \textbf{ of} \\
\quad \vdots \\
\quad \mathit{mk\text{-}Arith}(a, /, b) \ \to \{(\sigma, a'/b') \mid (\sigma, a') \in \mathcal{E}^{\exists}(a) \wedge \\
\qquad\qquad\qquad\qquad (\sigma, b') \in \mathcal{E}^{\exists}(b) \wedge b' \neq 0\} \\
\quad \mathit{mk\text{-}Equality}(a, b) \to \{(\sigma, a' = b') \mid (\sigma, a') \in \mathcal{E}^{\exists}(a) \wedge (\sigma, b') \in \mathcal{E}^{\exists}(b)\} \cup \\
\qquad\qquad\qquad\qquad \{(\sigma, \textbf{false}) \mid \sigma \in (\Sigma^C \setminus \textbf{dom}\, \mathcal{E}^{\exists}(a))\} \cup \\
\qquad\qquad\qquad\qquad \{(\sigma, \textbf{false}) \mid \sigma \in (\Sigma^C \setminus \textbf{dom}\, \mathcal{E}^{\exists}(b))\} \\
\quad \vdots \\
\quad \textbf{end}
\end{array}
$$

Fig. 5. The semantic function $\mathcal{E}^{\exists}$ for the approach of including a non-strict relational operator

5 Non-classical Logic: LPF

One way of thinking about partial functions and operators is that there are gaps in the denotations where they fail to denote: $7/0$ is not an integer; furthermore, if discussion is limited to the one strict notion of equality, $7/0 = 42$ fails to denote a Boolean value. Briefly revisiting (2), it should be clear that its truth relies on the truth of disjunctions such as $(1/1 = 1) \vee (0/0 = 1)$, which reduces to $(1 = 1) \vee (\bot_{\mathbb{Z}} = 1)$ and further to $\mathit{true} \vee \bot_{\mathbb{B}}$, since the equality is strict (i.e. undefined if either operand is undefined) and ultimately to $\bot_{\mathbb{B}}$. This unfortunately makes no sense in classical logic since its truth tables only define the logical operators for proper Boolean values.

As can be seen the LPF approach in Fig. 1(c) leaves the propositional operators to take the strain. The truth tables (disjunction, conjunction and negation) in Fig. 6 (presented in [36, §64]) illustrate how the propositional operators in LPF have been extended to handle logical values that may fail to denote. These truth tables provide the strongest possible *monotonic* extension of the familiar propositional operators with respect to the following ordering on the truth values: $\bot_{\mathbb{B}} \preceq \mathit{true}$ and $\bot_{\mathbb{B}} \preceq \mathit{false}$. The truth tables can be viewed as describing a parallel lazy evaluation of the operands, whereby a result is delivered as soon as enough information is available and such a result cannot be contradicted if a $\bot_{\mathbb{B}}$ later evaluates to a proper Boolean value.

$\vee$	**true**	$\bot_{\mathbb{B}}$	**false**
true	**true**	**true**	**true**
$\bot_{\mathbb{B}}$	**true**	$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$
false	**true**	$\bot_{\mathbb{B}}$	**false**

$\wedge$	**true**	$\bot_{\mathbb{B}}$	**false**
true	**true**	$\bot_{\mathbb{B}}$	**false**
$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$	**false**
false	**false**	**false**	**false**

$\neg$	
true	**false**
$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$
false	**true**

Δ	
true	**true**
$\bot_{\mathbb{B}}$	**false**
false	**true**

Fig. 6. The LPF truth tables for disjunction, conjunction, negation and definedness (Δ)

The quantifiers of LPF are a natural extension of the propositional operators — viewing existential quantification as an infinite disjunction (in the worst case) and universal quantification as an infinite conjunction. Thus, an existentially quantified expression in LPF is true if a witness value exists even if the quantified expression is undefined or false for some of the bound values. Such an expression is false if no witness value can be shown. Similar comments apply for universally quantified expressions.

For expressive completeness, LPF includes a definedness operator Δ whose truth table is also presented in Fig. 6. Unlike all of the other operators presented, the Δ operator is not monotone but is used only at the meta-level.

A semantics for the LPF version of the Predicate Calculus is detailed below. The abstract syntax is extended to include Δ, thus $\mathit{Expr}^L = \mathit{Expr} \mid \mathit{Delta}$ and where the abstract syntax for *Delta* is the same as for *Not*.

Since in LPF, the logical operators are extended to allow for the possibility that non-denoting logical values can be "caught", the standard definition of Σ

(given in Sect. 2) can be used for LPF, thus allowing for undefined propositional identifiers to occur in a specific σ.

The semantic function $\mathcal{E}^L$ is defined as $\mathcal{E}^C$, but with the additional and modified cases presented in Fig. 7; also any use of $\mathcal{E}^C$ needs to be replaced with $\mathcal{E}^L$ and any use of Σ^C needs to be replaced with Σ. Note that the semantics for quantifiers ensures that "gaps" are handled by non-denoting propositional expressions being absent from the domain of $\mathcal{E}^L$.

$$
\begin{array}{l}
\mathcal{E}^L : \mathit{Expr}^L \to \mathcal{P}(\Sigma \times \mathit{Value}) \\
\mathcal{E}^L(e) \ \triangleq \\
\quad \textbf{cases } e \textbf{ of} \\
\quad \vdots \\
\quad e \in \mathit{Prop} \qquad\qquad \to \{(\sigma, \sigma(e)) \mid \sigma \in \Sigma \wedge e \in \textbf{dom}\, \sigma\} \\
\quad e \in \mathit{Var} \qquad\qquad \to \{(\sigma, \sigma(e)) \mid \sigma \in \Sigma\} \\
\quad \mathit{mk\text{-}Arith}(a, /, b) \to \{(\sigma, a'/b') \mid (\sigma, a') \in \mathcal{E}^L(a) \wedge \\
\qquad\qquad\qquad\qquad\qquad (\sigma, b') \in \mathcal{E}^L(b) \wedge b' \neq 0\} \\
\quad \vdots \\
\quad \mathit{mk\text{-}Delta}(p) \qquad \to \{(\sigma, \textbf{true}) \mid \sigma \in \textbf{dom}\, \mathcal{E}^L(p)\} \cup \\
\qquad\qquad\qquad\qquad\quad \{(\sigma, \textbf{false}) \mid \sigma \in (\Sigma \setminus \textbf{dom}\, \mathcal{E}^L(p))\} \\
\quad \mathit{mk\text{-}Not}(p) \qquad \to \{(\sigma, \textbf{true}) \mid (\sigma, \textbf{false}) \in \mathcal{E}^L(p)\} \cup \\
\qquad\qquad\qquad\qquad\quad \{(\sigma, \textbf{false}) \mid (\sigma, \textbf{true}) \in \mathcal{E}^L(p)\} \\
\quad \mathit{mk\text{-}Or}(p, q) \qquad \to \{(\sigma, \textbf{true}) \mid (\sigma, \textbf{true}) \in \mathcal{E}^L(p)\} \cup \\
\qquad\qquad\qquad\qquad\quad \{(\sigma, \textbf{true}) \mid (\sigma, \textbf{true}) \in \mathcal{E}^L(q)\} \cup \\
\qquad\qquad\qquad\qquad\quad \{(\sigma, \textbf{false}) \mid (\sigma, \textbf{false}) \in \mathcal{E}^L(p) \wedge (\sigma, \textbf{false}) \in \mathcal{E}^L(q)\} \\
\quad \mathit{mk\text{-}Exists}(x, p) \to \{(\sigma, \textbf{true}) \mid \\
\qquad\qquad\qquad\qquad\qquad \sigma \in \Sigma \ \wedge \\
\qquad\qquad\qquad\qquad\qquad \textbf{true} \in \textbf{rng}\, (\{\sigma \dagger \{x \mapsto i\} \mid i{:}\mathbb{Z}\} \lhd \mathcal{E}^L(p))\} \cup \\
\qquad\qquad\qquad\qquad\quad \{(\sigma, \textbf{false}) \mid \\
\qquad\qquad\qquad\qquad\qquad \sigma \in \Sigma \ \wedge \\
\qquad\qquad\qquad\qquad\qquad \textbf{rng}\, (\{\sigma \dagger \{x \mapsto i\} \mid i{:}\mathbb{Z}\} \lhd \mathcal{E}^L(p)) = \{\textbf{false}\}\} \\
\quad \textbf{end}
\end{array}
$$

Fig. 7. The semantic function $\mathcal{E}^L$ for LPF

The "gaps" that arise from partial terms and propositional expressions in LPF are modelled by choosing *relations* as the space of *denotations* here. This is in contrast to the use of partial functions as is classical in *denotational semantics* [37]. The use of relations might suggest non-determinacy but all denotations are in fact single valued, i.e. any relation $\mathcal{E}^L(e)$ is deterministic (or "functional"), that is, for any expression e it follows that $(\sigma, v_1) \in \mathcal{E}^L(e) \wedge (\sigma, v_2) \in \mathcal{E}^L(e) \Rightarrow v_1 = v_2$.

6 Discussion

6.1 A Comparison of the Approaches Considered

The "proof of the pudding" for any logic is the ease of proof. Consider constructing a proof of (2) in classical logic; first, it is necessary to introduce some knowledge about division and subtraction, since a proof is a game with symbols, it cannot use the semantics of the arithmetic operators:

$$\forall i{:}\mathbb{Z} \cdot i = 0 \;\Rightarrow\; \neg((i-1)=0);\;\; \forall i{:}\mathbb{Z} \cdot \neg(i=0) \;\Rightarrow\; i/i = 1 \;\vdash\; \forall i{:}\mathbb{Z} \cdot (i/i=1) \vee ((i-1)/(i-1)=1)$$

A proof of the above property in classical logic is presented in Fig. 8 and it is pleasingly straightforward even though it hides the fact that the term $0/0$ with its undetermined denotation implicitly crops up in a number of places.

	from $\forall i{:}\mathbb{Z} \cdot i = 0 \;\Rightarrow\; \neg(i-1=0);\;\; \forall i{:}\mathbb{Z} \cdot \neg(i=0) \;\Rightarrow\; i/i = 1$		
1	**from** $i{:}\mathbb{Z}$		
1.1	$i = 0 \vee \neg(i=0)$		$h1, \mathbb{Z}$
1.2	**from** $i = 0$		
1.2.1	$\neg(i-1=0)$	$\Rightarrow$-E-$L(\forall$-$E(h1,h), h1.2)$	
1.2.2	$(i-1)/(i-1)=1$	$\Rightarrow$-E-$L(\forall$-$E(h1,h), 1.2.1)$	
	infer $(i/i=1) \vee ((i-1)/(i-1)=1)$	$\vee$-I-$L(1.2.2)$	
1.3	**from** $\neg(i=0)$		
1.3.1	$i/i = 1$	$\Rightarrow$-E-$L(\forall$-$E(h1,h), h1.3)$	
	infer $(i/i=1) \vee ((i-1)/(i-1)=1)$	$\vee$-I-$R(1.3.1)$	
	infer $(i/i=1) \vee ((i-1)/(i-1)=1)$	$\vee$-$E(1.1, 1.2, 1.3)$	
	infer $\forall i{:}\mathbb{Z} \cdot (i/i=1) \vee ((i-1)/(i-1)=1)$	$\forall$-$I(1)$	

Fig. 8. A proof of (2)

This prompts the question of how a proof of the same property would look in LPF. The answer is that it would be identical! The proof in Fig. 8 is a completely correct proof in LPF but the point is that nowhere is it necessary in LPF to make assumptions about the denotation of terms with zero divisors. It is also important that LPF maintains basic algebraic properties like the commutativity of conjunctions and disjunctions etc.

However, definedness does need to be established in some LPF proofs. There *are* certain constraints on inference rules in LPF. One issue is that the, so called, *law of the excluded middle*: $p \vee \neg p$, does not hold because the disjunction of two undefined Boolean values is still undefined: thus $(0/0 = 1) \vee \neg(0/0 = 1)$ is not a tautology in LPF. The non-monotone Δ operator in LPF does, however, give rise to an alternative property which is known as the *law of the excluded fourth*: $p \vee \neg p \vee \neg \Delta p$, that is, p is true, false or undefined. Futhermore, adding definedness hypotheses for all terms in some logical expression p is sufficient to make the validity of p in LPF and in classical logic coincide. One place where

Δ arises is when one wants to use what is, in classical logic, the unrestricted deduction theorem, which does not hold in LPF because knowing $\bot_{\mathbb{B}} \vdash \bot_{\mathbb{B}}$ is not the same as $\bot_{\mathbb{B}} \Rightarrow \bot_{\mathbb{B}}$. The use of Δ can provide a sound $\Rightarrow$-I rule for LPF:

$$\boxed{\Rightarrow\text{-}I}\ \frac{\Delta p; p \vdash q}{p \Rightarrow q}$$

But, Δ is not used in normal assertions as it is an operator on the meta-level. To claim definedness in a proof, the related δ operator is often used which is monotone and is equivalent to the assertion $p \vee \neg p$ (thus $\delta \bot_{\mathbb{B}} = \bot_{\mathbb{B}}$).

Interestingly, it can be argued that the law of the excluded middle doesn't necessarily hold in the $\mathcal{E}^C$ semantics where the partial division function has been underspecified! If division by zero yields a non-deterministic result then $7/0 = 0 \vee \neg(7/0 = 0)$ can be false. Since it is difficult in a logic to pin down a characterisation of "giving the same value within a context", the temptation to fix on a result such as $7/0 = 42$ becomes rather strong. Giving in to this temptation, however, leads to questions such as whether $7/0 = 5/0$ (see [38] for further discussion). The law of the excluded middle does hold in the $\mathcal{E}^D$ approach.

The approaches that handle partial terms but that allow for the classical logic operators to still be used can bring about "issues". For instance, partial functions no longer denote the obvious least fixed points [10,15] in the approach of making all terms denote. Furthermore [38] points to an issue for the underspecification approach if single element types are allowed and [14] and [10] point to another problem with the underspecification approach in having to specify, in general, the set of values where any used partial function is specified.

An obvious reservation about using multiple notions of equality is that anyone reasoning in this way has to observe different properties of the two or more notions. A strict (computational) notion of equality still needs to be written in function definitions and a non-strict notion of equality is needed to cope with partial terms. Note that in the $\mathcal{E}^{\exists}$ semantics, existential equality has replaced the strict equality. If the strict equality was to remain –in addition to the existential equality– then the $\mathcal{E}^{\exists}$ semantics would not be total for every Boolean expression, as there is still then the issue of partial terms propagating. Furthermore, although the focus here is on equality, the complications extend to include all of the other relational operators/predicates. There are also surprises in that, for example, existential inequality is *not* the negation of existential equality — they can both be false. There is an interesting formal connection between what can be proved in $\mathcal{E}^{\exists}$ and $\mathcal{E}^L$ which is explored in [39].

LPF is a candidate solution to the issue of handling partial terms. An argument that can be raised against LPF is that a large body of research and engineering has gone into classical logic which has led to a range of proof support. It is hoped that the progress reported in [32] on efficient semi-decision procedures such as resolution and refutation for LPF will lead to its wider use.

6.2 Further Approaches

A longer discussion of other approaches to handling non-denoting terms can be found in [12] but it is worth here making a few further points.

McCarthy's Conditional Operators: The propositional operators are defined by (non-strict) conditional expressions [1], for instance, the conditional disjunction operator (p **cor** q) is defined as: **if** p **then true else** q. Such a semantics is used in Raise [40, 41].

The conditional disjunction operator case of a semantic function (similar to what has been defined above for the other approaches) for McCarthy's approach $\mathcal{E}^M$ (that would use Σ in the function signature) would be defined as:

$$\begin{aligned} \mathit{mk\text{-}Or}(p,q) \rightarrow\ & \{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}^M(p)\} \cup \\ & \{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}^M(p) \wedge (\sigma, \mathbf{true}) \in \mathcal{E}^M(q)\} \cup \\ & \{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}^M(p) \wedge (\sigma, \mathbf{false}) \in \mathcal{E}^M(q)\} \end{aligned}$$

The first variable in the conditional expressions is usually referred to as the "inevitable variable" because, if it is undefined, then the entire expression is undefined since conditional expressions are strict in their first argument. This means that disjunction and conjunction are no longer commutative and, additionally, quantifiers are problematic with respect to undefined values. Thus, $\exists i{:}\{0,1\} \cdot i/i = 1$ may not have the same truth value as $1/1 = 1 \vee 0/0 = 1$. So while, (1) with the conditional implication operator can be proved in McCarthy's approach, neither the contrapositive of (1) nor (2) follow for conditionally defined operators.

The conditional form of the logical operators were used in the early IBM Vienna operational semantics definitions known as VDL (see [42, §1.1.6.2]). It was an unpreparedness to tolerate the loss of properties like commutativity of disjunction and conjunction that drove the first author of the current paper to experiment with using both the conditional and the classical operators in [43] (an idea also tried in [44] and [45]). As can be seen from [45], the distribution laws become problematic.

Avoiding Function Application: Several authors have tried to avoid writing the expression $f(x) = y$ and instead write it as $(x, y) \in f^r$, where f^r is the relation that is the "graph" of the function f. The key idea is that $(x, y) \in f^r$ is false when $x \notin \mathbf{dom}\, f^r$, for all y. This idea in not analysed in detail here (see [12, 15] for further detail), because the notation becomes rather heavy[3] but it is easy to see how it could be added to the semantics used above.

Restricting the Sets over Which Bounded Variables Range: Another solution is to restrict quantification to sets that do not contain any values outside

[3] Property 2 has to be rewritten as: $\forall i{:}\mathbb{Z} \cdot ((i,i),1) \in /^r \vee (((i-1),(i-1)),1) \in /^r$ and rewriting $g(f(x))$ requires an extra existential quantifier.

of the actual defined domains of any of the functions used. For example, (1) could be written as: $\forall i{:}\,\{i \mid i{:}\,\mathbb{Z} \wedge i \neq 0\} \cdot i/i = 1$. One could even force partial functions to become total by encoding the actual defined domain in the domain type; the application of a function with argument(s) outside of that domain could be considered to be a type error. Unfortunately, in general, the type structure becomes both clumsy and undecidable. Refer to [12, 14] for further information.

Restricting the Expressions Written: It is possible to view the relation $\mathcal{E}^L$ as total by restricting the expressions e to those for which there exists, for all $\sigma \in \Sigma$, a tuple $(\sigma, v) \in \mathcal{E}^L(e)$. As seen in [19, 46] such well definedness ("WD") restrictions can be complicated and expand exponentially in size.

7 Conclusions

This paper provides a semantic model for the most common approaches to coping with partial terms: LPF; making all terms denote values; and using non-strict relational operators. The model employs semantic functions which map logical expressions to relations over interpretations and results — this leads naturally to the view of non-denoting terms corresponding to "gaps". Each of the approaches attempts to catch "undefinedness" in a different place (see Fig. 1) and the semantic models are used to compare and contrast the different approaches. It is interesting to note that rather simple changes to the semantic models explain the different possibilities.

Acknowledgements. The authors gratefully acknowledge the funding for their research from an EPSRC grant for AI4FM and the Platform Grant TrAmS-2 as well as an EPSRC PhD Studentship. The authors also thank the referees for their feedback which we hope has helped to clarify the explanation in the paper.

References

1. McCarthy, J.: A basis for a mathematical theory for computation. In: Braffort, P., Hirschberg, D. (eds.) Computer Programming and Formal Systems, pp. 33–70. North-Holland Publishing Company (1967)
2. Owe, O.: An approach to program reasoning based on a first order logic for partial functions. Technical Report 89, Institute of Informatics, University of Oslo (February 1985)
3. Owe, O.: Partial logics reconsidered: A conservative approach. Formal Aspects of Computing 5, 208–223 (1993)
4. Cheng, J.H.: A Logic for Partial Functions. PhD thesis, University of Manchester (1986)
5. Tennent, R.: A note on undefined expression values in programming logic. Information Processing Letters 24(5) (March 1987)
6. Blikle, A.: Three-valued Predicates for Software Specification and Validation. In: Bloomfield, R., Marshall, L., Jones, R. (eds.) VDM 1988. LNCS, vol. 328, pp. 243–266. Springer, Heidelberg (1988)

7. Konikowska, B., Tarlecki, A., Blikle, A.: A Three-valued Logic for Software Specification and Validation. In: Bloomfield, R., Marshall, L., Jones, R. (eds.) VDM 1988. LNCS, vol. 328, pp. 218–242. Springer, Heidelberg (1988)
8. Jervis, C.: A Theory of Program Correctness with Three Valued Logic. PhD thesis, Leeds University (1988)
9. Spivey, J.: Understanding Z—A Specification Language and its Formal Semantics. Cambridge Tracts in Computer Science, vol. 3. Cambridge University Press (1988)
10. Schieder, B., Broy, M.: Adapting calculational logic to the undefined. The Computer Journal 42 (1999)
11. Jones, C.B.: Systematic Software Development using VDM, 2nd edn. Prentice Hall International (1990)
12. Cheng, J.H., Jones, C.B.: On the usability of logics which handle partial functions. In: Morgan, C., Woodcock, J.C.P. (eds.) 3rd Refinement Workshop, pp. 51–69. Springer (1991)
13. Müller, O., Slind, K.: Treating partiality in a logic of total functions. The Computer Journal 40(10), 640–652 (1997)
14. Gries, D., Schneider, F.B.: Avoiding the Undefined by Underspecification. In: van Leeuwen, J. (ed.) Computer Science Today. LNCS, vol. 1000, pp. 366–373. Springer, Heidelberg (1995)
15. Jones, C.B.: Reasoning about partial functions in the formal development of programs. In: Proceedings of AVoCS 2005. Electronic Notes in Theoretical Computer Science, vol. 145, pp. 3–25. Elsevier (2006)
16. Fitzgerald, J.S.: The Typed Logic of Partial Functions and the Vienna Development Method. In: Bjørner, D., Henson, M.C. (eds.) Logics of Specification Languages. EATCS Texts in Theoretical Computer Science, pp. 427–461. Springer (2007)
17. Darvas, Á., Mehta, F., Rudich, A.: Efficient Well-Definedness Checking. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 100–115. Springer, Heidelberg (2008)
18. Woodcock, J., Freitas, L.: Linking VDM and Z. In: 13th IEEE International Conference on Engineering of Complex Computer Systems, pp. 143–152 (April 2008)
19. Schmalz, M.: Term Rewriting in Logics of Partial Functions. In: Qin, S., Qiu, Z. (eds.) ICFEM 2011. LNCS, vol. 6991, pp. 633–650. Springer, Heidelberg (2011), doi:10.1007/978-3-642-24559-6-42
20. Łukasiewicz, J.: O logice trójwartościowej. Ruch Filozoficzny, 169–171 (1920) Translated as (On three-valued logic) McCall, S. (ed.) in Polish Logic, Oxford U.P, 1920–1939 (1967)
21. Wang, H.: The calculus of partial predicates and its extension to set theory. Math. Logic 7, 283–288 (1961)
22. van Fraasen, B.: Singular terms, truth-value gaps and free logic. J. Philosophy 63, 481–495 (1966)
23. Koletsos, G.: Sequent calculus and partial logic. Master's thesis, Manchester University (1976)
24. Hoogewijs, A.: Partial-predicate logic in computer science. Acta Informatica 24, 381–393 (1987)
25. Avron, A.: Foundations and proof theory of 3-valued logics. Technical Report ECS-LFCS-88-48, LFCS, Department of Computer Science, University of Edinburgh (April 1988)
26. Farmer, W.M.: A partial functions version of Church's simple theory of types. Journal of Symbolic Logic 55(3), 1269–1291 (1990)

27. MacColl, H.: A report on MacColl's three-valued logic. In: Lovett, E. (ed.) Mathematics at the Intern. Congress of Philosophy, vol. 7, pp. 157–183. Bulletin of the American Mathematical Society (1901)
28. Barringer, H., Cheng, J., Jones, C.B.: A logic covering undefinedness in program proofs. Acta Informatica 21, 251–269 (1984)
29. Bicarregui, J., Fitzgerald, J., Lindsay, P., Moore, R., Ritchie, B.: Proof in VDM: A Practitioner's Guide. In: FACIT, Springer (1994) ISBN 3-540-19813-X
30. Jones, C.B., Middelburg, C.: A typed logic of partial functions reconstructed classically. Acta Informatica 31(5), 399–430 (1994)
31. Jones, C.B., Lovert, M.J.: Semantic models for a logic of partial functions. IJSI 5, 55–76 (2011)
32. Jones, C.B., Lovert, M.J., Steggles, L.J.: Towards a mechanisation of a logic that copes with partial terms. Technical Report CS-TR-1314, Newcastle University (February 2012)
33. Blamey, S.R.: Partial Valued Logic. PhD thesis, Oxford University (1980)
34. Blamey, S.: Partial logic. In: Gabbay, D., Guenthuer, F. (eds.) Handbook of Philosophical Logic, vol. III. Reidel (1986)
35. Lovert, M.J.: A semantic model for a logic of partial functions. In: Pierce, K., Plat, N., Wolff, S., eds.: Proceedings of the 8th Overture Workshop. Number CS-TR-1224 in School of Computing Science Technical Report, Newcastle University, 33–45 (2010)
36. Kleene, S.C.: Introduction to Metamathematics. Van Nostrad (1952)
37. Stoy, J.E.: Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. MIT Press (1977)
38. Jones, C.: Partial functions and logics: A warning. Information Processing Letters 54(2), 65–67 (1995)
39. Fitzgerald, J.S., Jones, C.B.: The connection between two ways of reasoning about partial functions. IPL 107(3-4), 128–132 (2008)
40. Group, T.R.L.: The RAISE Specification Language. BCS Practitioner Series. Prentice Hall (1992) ISBN 0-13-752833-7
41. Group, T.R.M.: The RAISE Development Method. BCS Practitioner Series. Prentice Hall (1995) ISBN 0-13-752700-4
42. Walk, K., et al.: Abstract syntax and interpretation of PL/I. Technical Report TR25.098, IBM Laboratory Vienna (1969)
43. Jones, C.B.: Formal development of correct algorithms: an example based on Earley's recogniser. SIGPLAN Notices 7(1), 150–169 (1972)
44. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Englewood Cliffs (1976)
45. Gries, D., Schneider, F.B.: A Logical Approach to Discrete Math., 2nd edn. Springer (1996)
46. Mehta, F.D.: Proofs for the Working Engineer. PhD thesis, ETH Zuerich (2008)

Combining VDM with Executable Code

Claus Ballegaard Nielsen, Kenneth Lausdahl, and Peter Gorm Larsen

Department of Engineering, Aarhus University
Finlandsgade 24, DK-8200 Aarhus N, Denmark
{clausbn,kel,pgl}@iha.dk

Abstract. Formal methods have been used and successfully applied to a wide range of industrial applications for many years. However formal methods can be difficult to comprehend for outsiders and the link of formal models and external subsystems which are not modelled can be unclear. In this paper we present an approach which allows formal models to be more easily shared with external stakeholders and enables integration with external code. We demonstrate how an existing interpreter for an executable subset of VDM is extended enabling the combination of formal models with executable code. This eases the way in which a formal model can communicate with an external implementation or be used in graphical prototyping. A small case study is used to demonstrate how the approach can be utilized. In this paper the technique is used to combine VDM and Java, but the principles presented can be seen as a general approach for expanding the capabilities of formal modelling tools with interpretation capabilities.

1 Introduction

In the development of IT-based systems it is often necessary to communicate conceptual ideas to stakeholders with very little knowledge of IT. These stakeholders are typically managers and/or domain experts and generally speaking they may find it hard to understand the details of artifacts (including formal models) used inside a software development process. Normally communication between a development team and such external stakeholders is either carried out by writing documents or presenting different types of diagrams. However, if it is possible to demonstrate different kinds of aspects with a prototype, communication may be enhanced tremendously. Thus, it is helpful if some kind of graphical user interface is used to drive the input for a formal model. However, to enable this special tool support is required to combine the interpretation of a formal model with a graphical user interface.

In the same way it may not be worthwhile to formally model all parts of a system. This can either be because the formal modelling language is not capable of expressing certain aspects (e.g. user interfaces) or because legacy code already exists which may be trusted sufficiently (e.g. existing databases). Thus, for such components it may be advantageous to provide some capability for enabling interaction between the parts of the system that are formally modelled and the

J. Derrick et al. (Eds.): ABZ 2012, LNCS 7316, pp. 266–279, 2012.

parts of the system that only exist in external code. Just like for the prototyping purpose, mentioned above, special tool support is required to enable such a heterogeneous combination of the different parts.

In this paper we give with a constructive solution for solving these challenges, in an easy fashion, enabling reasonably fast support for communication with stakeholders who do not understand a formal model at all, as well as for combining a formal model with external code written in a programming language. Our implementation relies on the Overture open source tool [10], using VDM as the formal method and Java as the programming language. However, the ideas presented here can be generalised to other formal methods as well as other programming languages.

After this introduction this paper provides a brief introduction to VDM in Section 2. This is followed by Section 3 which explains how the technical solution is achieved in a fashion that should enable other tool builders to get inspiration for adding similar capabilities. Section 4 will present a small case study with a formal VDM model of a traffic infrastructure system with buses in a city, where traffic planners need to decide upon optimal planning for their customers. Section 5 provides an overview of related work and finally Section 6 presents concluding remarks.

2 The Vienna Development Method

The Vienna Development Method (VDM) is one of the longest established model-oriented formal methods for the development of computer-based systems and software [3]. The common core is based on the VDM Specification Language (VDM-SL), standardised by ISO [12], which provides facilities for the functional specification of sequential systems. The principal extensions of VDM-SL are VDM++ [4], which adds features for object-oriented modelling and thread-based concurrency, and VDM-RT for the development of real time distributed systems [14]. VDM-SL uses a structuring mechanism called modules, while VDM++ and VDM-RT use classes for encapsulating data and operations.

The type system in VDM has a number of basic types including several kinds of numeric types, Booleans and special kinds of tokens. More complex types can be defined using constructors for set, sequence and mapping types. Type membership and state variables can be restricted by invariant predicates which means that run-time type checking is required. In VDM referentially transparent functions and operations that can access persistent state information can both be defined explicitly as well as implicitly using pre- and post-condition predicates [11]. It is also worth noting that it is possible to let the definition of a function or operation be deferred to a later time by using the special `is not yet specified` key-phrase in the body. This detail is important in the work presented here. VDM is supported by an industry-strength tool set, VDMTools, owned and developed further by SCSK Systems [5] and the open source tool called Overture [10]. These tools offer syntax checking, type checking and proof obligation generation capabilities, an interpreter with debugging functionality,

code generators, a pretty printer and links to external tools for UML modelling to support round-trip engineering. In this paper we focus on an extension of the interpreter from the Overture tool [11].

3 Combining VDM Interpretation with External Code

Interpretation and debugging of an executable subset of VDM models is supported by the Overture interpreter called *VDMJ*. It includes a general debugging feature and run-time checks for pre-/post-conditions, invariants and dynamic type checking. The interpreter is Java-based and has its own internal `Value`s to hold the calculated value of all kinds of types during execution. The potential and value of VDM models can be improved greatly by a technique in the interpreter which allows the functionality to be increased beyond the standard provided by VDM, such that VDM models can be combined with external executable code. The interpreter enables (a) VDM to call directly into external code and conversely (b) Java code to control the VDMJ interpreter and execute VDM expressions in an executing model. The former method is called the *External Call Interface* (ECI), as it enables the interpreter to call external implementations, and the latter the Remote Control Interface (RCI) as it allows the interpreter to be controlled remotely. These two methods of integration allow VDM models to be connected to legacy systems, and either use external libraries or use graphical prototypes for presentation or interaction with the model. The Overture tool is bundled with a number of libraries for IO, Math and other useful functionality which are not worthwhile specifying in VDM itself; internally these libraries actually make use of the *ECI*.

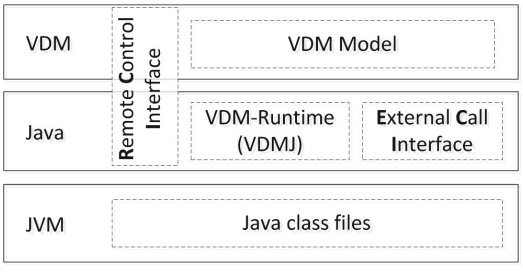


Fig. 1. Overview of the methods in relation to the specification

Fig. 1 illustrates how the *ECI* and the *RCI* can be seen in relation to Java and the VDM interpreter. The Java JVM underpins the whole system, based on this the VDM interpreter in the centre acts as the heart of the execution, while the VDM model rests on top. The executing model is capable of initiating the *ECI* which directly calls into Java. Please note that the *ECI* functionality is handled within VDMJ, but as it is not, as such, interrelated with the regular interpretation of the overlying VDM model, it has been separated in the illustration. The *RCI* initiates the interpreter and enables commands to be passed

directly to the interpreter from external Java code. Knowledge of the executing model is necessary in order to use the *RCI* effectively, therefore the VDM Remote Control pillar raises from the Java based interpreter level to the VDM model level in the illustration of Fig. 1.

An important part of using the interfaces is understanding how values of different types are passed from VDM to Java and vice versa. At run-time the interpreter uses sub-type instances of an abstract Java class called `Value` to hold the results of evaluated expressions. The basic typed VDM values, such as: `BooleanValue`, `CharacterValue` and `NumericalValue` etc., contains fields with equivalent Java type variables enabling easy conversion to basic Java types. The more complex typed values such as `SetValue`, `SeqValue`, `MapValue`, `RecordValue`, `TupleValue` are all composite values which have utility methods to extract the contained values, like the method `values()` used in `SetValue` to obtain the values of the set. The VDM union type is a type that is not present at run-time, since `int|set of int` allows either a `NumericalValue` or a `SetValue` of `NumericalValue`s to be returned. This is handled in the interpreter by using the abstract `Value` class as return type from all evaluating code. However, to do the same in Java the only class which allows both an `Integer` and `List<Integer>` is Object and therefore poses a challenge when one needs to convert from the interpreter's value system to raw Java types. A value factory is supplied by the remote interpreter to assist the creation of VDM values from Java code.

The VDM Value hierarchy is provided by a VDMJ specific Java library (specifically in `org.overturetool.vdmj.values`), which must be included in external Java libraries that integrate with the Overture interpreter. When integrating with an existing system this requirement might entail that an intermediate library is necessary, as a proxy, between the formal model and the existing legacy implementation.

3.1 The External Call Interface

The *ECI* allows the VDM model to delegate operation invocations to external implementations without any changes to the VDM syntax or any configuration of communication channels. Calling functionality defined in an external Java library directly from VDM is made possible via the VDM `is not yet specified` construct. This construct can be supplied as the body of an operation and normally implies that the definition of the operation has been deferred to a later time; it however doubles as a way of marking delegation of the functionality. The first time an operation with `is not yet specified` as its body is invoked the interpreter will attempt to delegate the operation call to an external Java method. The matching Java delegate method must have the same name and number of parameters as the VDM operation, and it must reside within a class with the same name as the VDM class or module. If the VDM operation is invoked in a model and a corresponding delegate cannot be found in a Java library a standard run-time error is issued. At run-time the interpreter searches the standard VDM libraries default directory, a project subdirectory named *lib*, for external libraries packaged as jar files. To make a VDM class name match

a Java class, placed within an Java package, the dots separating each of the package names have to be replaced with underscores. The interpreter will then convert the VDM name into a fully qualified Java class name, e.g. a VDM class named `gui_Road` will match the Java class Road in package gui: (`gui.Road`) .

To illustrate how the *ECI* can be used, List. 1.1 and List. 1.2 show a VDM class with an operation marked as `is not yet specified`, and the corresponding Java implementation, respectively. Note that a VDM class can contain a mixture of implicitly and explicitly operations as well as `is not yet specified` operations. The example illustrates how an operation in a VDM model is used for loading additional information about a road from an external data store.

List. 1.1 shows the VDM class containing the `RoadInfo` record type and the `getRoadInfo` operation. This operation has an external implementation which must return the `RoadInfo` record type. Since the body `is not yet specified` is used, the interpreter knows that it has to search for an external implementation at run-time.

```
class gui_Road
types
RoadInfo ::
  number       : RoadNumber
  maxWaypoints : nat;
operations
 public getRoadInfo : RoadNumber * int ==> RoadInfo
 getRoadInfo(num, region) == is not yet specified;
end gui_Road
```

List. 1.1. VDM++ Road class with externally specified operation

```
package gui;
import org.overturetool.vdmj.values.*;
public class Road {
public Value getRoadInfo(Value num, Value region)
throws ValueException {
String roadId= num.toString();int regionId= region.intValue();
 return
    interpreter.getFactory().createRecord("RoadInfo",roadId,
                  store.getMaxWaypoints(roadId,regionId));}}
```

List. 1.2. Road class Java implementation

List. 1.2 shows the Java implementation of the `getRoadInfo` operation. The method accepts two arguments of the generic `Value` type; one for the `num` and one for the `region`. In the method body the two arguments are converted into Java types, `num` is converted to a `string` and the `region` to an `integer`, both used to lookup information through the `store` object instance. The record type,

which has to be returned, is created by using the Value factory from the remote interpreter, (it could also be constructed by passing a string containing the VDM record creation expression to the interpreter). The factory will make a `RecordValue` object, which can then be returned to the model by the Java method.

3.2 The Remote Control Interface

The *RCI* enables control of the VDM interpreter from an external Java application. This can be utilized to control the execution of a VDM model, for instance through a graphical user interface (GUI). The *RCI* consists of two parts; the `RemoteControl` Java interface and the `RemoteInterpreter` Java class. An external Java implementation can interact with the interpreter by implementing the interface `RemoteControl` (defined in *org.overturetool.vdmj.debug.RemoteInterpreter*) which simply defines a `Run` method. Once the interpreter is started with the Remote Control enabled, it will call the `Run` method and pass an instance of the `RemoteInterpreter` object, which refers to the loaded VDM model. An implementation of the interface is shown in List. 1.3.

```
public class BuslinesRemote implements RemoteControl {
import org.overturetool.vdmj.debug.RemoteControl;
...
 private RemoteInterpreter interpreter;
 @Override
 public void run(RemoteInterpreter intptr) throws Exception {
  interpreter = intptr; }
 ...
```

List. 1.3. Implementation of the RemoteControl interface

In the Overture tool the *RCI* is enabled through the Overture Launch Configuration, where the fully qualified name of the Java class implementing the *RCI* can be supplied. The default VDM library directory will be searched for the matching implementation. When initiating the `RemoteInterpreter` the Overture tool will load the VDM model attached to the Launch Configuration into the interpreter. The composition of the *RCI* is shown in the class diagram in Fig. 2.

The functionality of the `RemoteInterpreter` can essentially be seen as having a console-like access directly to the interpreter, where VDM expressions can be given as strings to the `execute` and `valueExecute` methods. The `valueExecute` method will return a VDM `Value` instance containing the result of the execution, while the `execute` method returns a string representation of the result. As the interpreter maintains the model state between each call to the execute methods, additional methods are defined for creating variables to hold instantiations and results during continuous execution, and for stopping the interpreter. Once the `RemoteInterpreter` has been passed via the `Run` method, the external code is

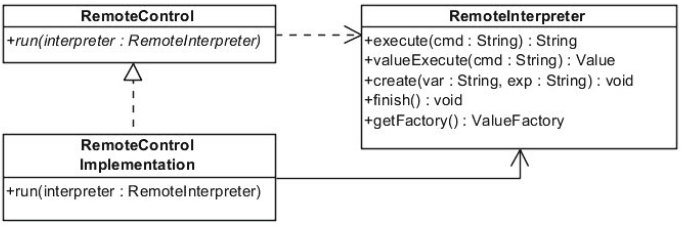


Fig. 2. The Remote Control Interface composition

in control of the interpretation and it must drive the execution by calling into the VDM model. While the `RemoteInterpreter` is in control it performs all of the run-time checks on pre/post-conditions, invariants, run-time type checking. Additionally the Overture tool debugger is still fully functional, meaning breakpoints can be used in debug mode. In the interpreter internals the main execution loop has been changed so that the control of the execution is delegated to the external code when the *RCI* is used.

```
public void init() throws ValueException {
  interpreter.create("w","new World()");
  Value ret = interpreter.execute("w.RunVDMModel()");
  if(!ret.boolValue()){
    System.err.println("World initialization failed"); }}
```

List. 1.4. Example of usage of the `RemoteInterpreter` interface

List. 1.4 shows a call to the remote interpreter `interpreter` where an instance of the class `World` is created, placed in the variable `w`, and the operation `RunVDMModel` is executed. The return value from `Run` is declared in the model to be a boolean type which is checked following the execution.

The VDM interpreter controls and executes VDM threads through a scheduler that controls the underlying Java threads. Each VDM thread is associated with a Java thread to perform VDM executions and run debug sessions. The scheduler controls the Java threads by the use of Java synchronization and VDM thread identifiers associated with the Java threads. This poses a challenge for the `RemoteInterpreter` since it will be used by threads unknown to the scheduler and thus not associated with any VDM thread. To handle this all calls through the `RemoteInterpreter` are handed to a VDM thread used specifically for executing remote commands. When a command is executed the calling thread is blocked until the VDM thread has executed the command. The result is then returned to the calling thread or, if the remote command results in an execution exception, the exception is then re-thrown in the calling thread, allowing the external code to react to it.

4 Case Study

To illustrate how the presented techniques can be used in a formal model, a case study of a simple traffic infrastructure system is used. The model is created for a group of traffic planners who are attempting to analyse the optimal planning of bus routes within a city. The various infrastructure maps of the city roads and current bus routes are already stored in an operational database and the traffic planners wish to use this existing data as input to the formal model of their infrastructure. The planners have no knowledge of formal modelling, but they want to be capable of changing the configuration of the model and to get a clear understanding of how their changes affect the city infrastructure. The model of the system describes an infrastructure with bus stops and connected roads, along which buses move to service the bus stops. The system has a constant inflow of new passengers which will board buses that are destined for their particular bus stops. The purpose of the model is to determine the efficiency of the transportation system by measuring the time passengers have to wait before they can be transported to their desired destination. These measurements will depend on the number and capacity of buses, the route of the buses and the inflow of new passengers, all of which can be adjusted in the model. At any time during execution new buses can be added and the inflow of passengers can be varied.

The requirement and abstractions of the system can be summed up as:

- Passengers arrive at a steady inflow rate at a central station, their destination is randomly chosen,
- a bus route is defined by roads, and the bus will stop at all stops it passes.
- roads are connected by waypoints, of which some are bus stops
- the roads in a bus route must be connected end to end,
- buses always drive in circles. i.e. the start and the stop of a route must be the same.

An executable VDM++ model has been created which describes the system and ensures that the modelled system conforms to the system requirements above. The relations of roads and the validity of bus routes are checked to confirm that the defined roads line up and that it is possible for the bus to follow the route. During execution the model keeps track of passengers that get annoyed with waiting, based on time passed; of the movements made by buses and passengers; and that passengers get on the right bus and off at the right stop.

To make the model more comprehensible and interactive a graphical representation has been created in Java that displays a choice between different maps, of which one can be selected and the movement of buses and passengers can be animated based on the executing model. Fig. 3 shows a screen-shot of a running animation, where a selected map is displayed with the buses indicated by squares on the roads and the waiting passengers as the circles to the right. The graphical representation is purely an overlay on top of the model, everything is continually checked and validated by the executing VDM model, from bus movements to the passenger count.

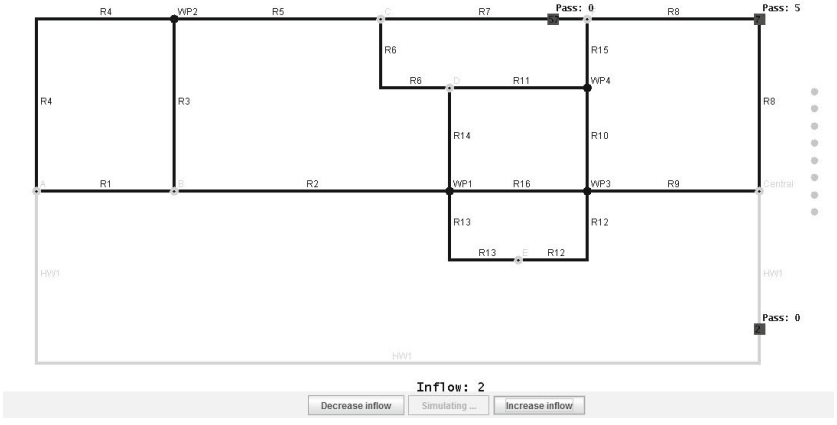


Fig. 3. Screenshot of the models graphical representation

The case study uses both the *ECI* and the *RCI*, such that both the VDM++ model and Java implementation can affect each other. During simulation the VDM model can notify the GUI of data changes which could affect the animation, and a user can load maps from the database into the model by using the GUI as well as adjusting the passenger inflow in the model. The *ECI* is used by the model to update the graphical representation each time there is a visible change in the executing model, e.g. buses moving or passenger arriving.

List. 1.5 contains selected operations from the interface *Graphics*, which is the *ECI* defined between the VDM model and the Java implementation used in the case study. The `move` operation is used for synchronization between the model and the animation, and will be discussed in further detail below, while the `busPassengerCountChanged` operation is used to updated the number of passengers on a specific bus; a count which is displayed in the GUI beside each bus.

```
class gui_Graphics
operations
 public move : () ==> ()
 move() == is not yet specified;

 public busPassengerCountChanged : nat * nat ==> ()
 busPassengerCountChanged(busid,count)== is not yet specified;
 ...
end gui_Graphics
```

List. 1.5. Specification of the External Call Interface in the case study

List. 1.6 contains the Java implementation of `busPassengerCountChanged` operation, showing how simple changes in the VDM model can be transferred

to the Java implementation. In this code segment the `model` variable is part of the Model-View-Control pattern [13] used for the GUI implementation.

```
public Value busPassengerCountChanged(Value busid,Value count)
 throws ValueException {
  long id =  busid.intValue(); long count = count.intValue());
  model.busPassengerCountChanged(id, count);
 return new VoidValue();}
```

List. 1.6. Java implementation of the busPassengerCountChanged operation in the case study

The *RCI* is used for adding waypoints, bus stops, roads and buses to the model, as well as for starting the simulation and increasing/decreasing the flow-rate of new passengers.

To emulate the infrastructure database the Java implementation uses JDBC to connect to a local database[1] that contains the existing maps and bus routes. Each of these maps can be loaded and graphically rendered prior to starting the simulation. The maps' conformity to the model rules, such as the connectivity and layout of roads, is checked by passing the data to VDM model via the *RCI*. If any invalid data that does not conform to the rules mentioned above is added to the model a run-time error is issued by the interpreter. All data used by both the VDM model and the graphics, such as the relations between waypoints, is stored in Java entity objects before being passed to the VDM model.

For example List. 1.7 shows how a road is added to the VDM model by calling an operation through the remote controlled interpreter. A string is built containing the operation and its parameters written in VDM syntax. The `Waypoint` class is an entity object with an overridden *toString()* method that outputs the data in VDM syntax.

```
public void AddRoad(Waypoint wp1, Waypoint wp2, String road,
 int length) {
 String cmd = "w.addRoad(" + wp1 + "," + wp2 + "," + road +
  "," + length + ")";
 interpreter.valueExecute(cmd);}
```

List. 1.7. Adding a road section to the VDM model using the remote controlled interpreter

The sequence diagram in Fig. 4 shows the communication and interfacing between the model and the graphical implementation. The simulation of the selected map is started by a click on the GUI, which starts the model by a call through the `Remote-Interpreter`. When the `Start` operation is called in the VDM model a clock is started in order to keep track of time. Once one time

[1] Using the H2 Database Engine, see `http://www.h2database.com`

step has passed, the VDM model calls the `Move` operation on the `Graphics` class which allows GUI to be updated. Following the graphical update, the `RemoteInterpreter` is used once again by the GUI to inform the VDM model that the next time step can be taken. This scenario continues until the simulation has completed. This advance in time, and toggling between invoking the `Move` method and `TimeStep` operation is used to ensure synchronization between the VDM model and the Java visualisation.

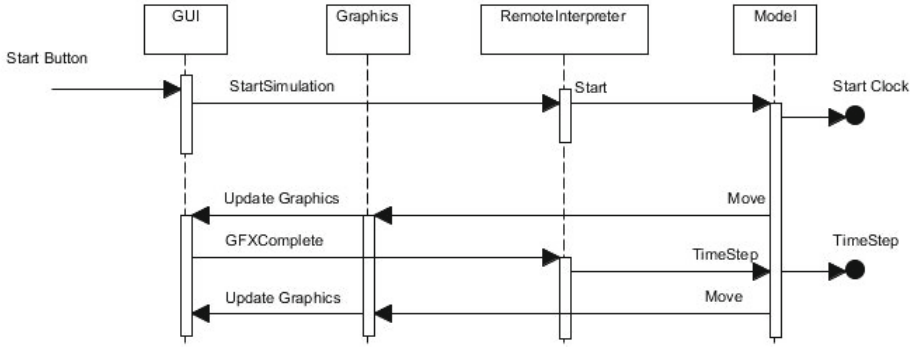


Fig. 4. Communication between model and external code via interfaces

What cannot be seen on the illustration in Fig. 4 is that, upon a time step, the data that the graphical representation is based on is updated by the VDM model through operations defined in the `Graphics` class. When a time step is taken and time progresses in the model, all the representations of buses and passengers move as well, meaning their data changes. The *ECI* allows the VDM model to notify the GUI of these changes.

5 Related Work

Frohlich [6] reports work on enabling VDMTools to execute combined specifications consisting of both specification and externally specified C++ code from Dynamic Link Libraries (DLLs). New syntax is introduced that defines a new implementation module type in which an export section can define the function signatures for external functions, and a `uselib` keyword refers to external libraries. Type conversion functions are used to transform the value types, used by the interpreter, to the values type of the C++ code, and vice versa. Our work is based on this approach, with a similar dynamic semantics but no need to change the VDM syntax. Similarly checks on invariants and pre-/post-conditions together with exceptions, including the interpreter handling external code exceptions, is supported; these are limitations of the work by Frohlich et al [6]. This approach enables a model to be combined with externally specified components while the interpreter itself stays in control during the execution. While this is good for

accessing external code, it is not sufficient for interacting with the model, as needed by an interactive GUI. Previous work has been carried out in this area for VDMTools, where interaction with the model is enabled by a CORBA interface [8]. The interface allows a CORBA client to communicate directly with the VDMTools interpreter and pass VDM expressions to be evaluated. This allows external code to interact with a running model. For type conversion between the VDM interpreter and the external code a CORBA IDL is defined which describes the different VDM values, and the CORBA Narrow functionality is used for the conversion. Using CORBA enables software components written in multiple computer languages and running on multiple computers to interact with the VDM model. However the broad heterogeneity comes at the price of increased complexity compared to a plain Java integration, as the CORBA clients can be troublesome to implement.

Comms/CPN [7] is a Standard ML library for the Coloured Petri tool package Design/CPN, which enables communication between CPN models and external processes. The Comms/CPN library enables two-way communication between the CPN model and the external process using TCP/IP, by defining generic send and receive-functions which accept a byte stream of data. Encoding/decoding functions has to be implemented to marshal data for transmission. Comms/CPN has the advantage of using TCP/IP which allows heterogeneous clients to interact with the simulator, while the send/receive approach has the weakness of potentially blocking the simulator while waiting for data transmission and it requires the external process to implement some conveying and mapping of the received data into concrete functionality e.g. an update of a graphical animation.

The successor of Design/CPN; CPN Tools, has a similar functionality called Access/CPN [15] which enables the integration of CPN models with external applications. Access/CPN has a Java interface offering an implementation of the protocol used to communicate with the CPN simulator. Communication is done via a TCP/IP stream using a custom packet format which marshals simple data types and all communication can be wrapped in a high-level simulator object, which contains methods for evaluating expressions and processing models directly in the simulator. The presented *RCI* resembles this approach, while it differs on the use of external Java libraries and class loaders for integration instead of TCP/IP, where the latter requires some lower-level configuration with address and ports.

In Event-B the tool B-Motion Studio [9] makes it possible to create visualisations via an visual editor and establish a link from it to the model using Event-B expressions as gluing code. The tool's key feature is that it allows for simpler and faster creation of graphical representations without requiring knowledge of graphical programming. Its focus on easy construction of visualisation comes at the price of flexibility, as users who might want to do advanced features, in particular parts of the visualisation, will lack the versatility provided by access to lower-lever graphical programming. To our knowledge there is no possibility of interacting with the model or visualisation through external executable code.

6 Conclusion and Future Work

We have presented an approach for combining executable formal models with external executable code to enable graphical prototyping of models and for easier integration between models and existing implementations. The aim of our approach is twofold, the first objective is to advance the understanding of models by enabling the creation of graphical representations on top of the model. The functionality and meaning of the model can then be easily conveyed to stakeholders with limited knowledge of formal models through the use of animation and interaction. The second objective is to enable formal models to be integrated with external executable code, such as external libraries or systems. The presented approach enables the model to integrate with external libraries which often have well-defined behaviour and therefore falls outside the modelling effort.

Enabling formal models to use externally specified components, within the context of VDM research, has previously been presented in work by Frohlich et al. [6]. The *ECI* builds on this but has the added advantage of the integration process being significantly simplified as there is no need to learn new syntax, wrestle C++ DLLs or implement CORBA clients. Seen from inside the VDM model the approach does not require any new syntax or constructs as everything relies on invoking operations. Only minor changes to the semantics of `is not yet specified` body is needed to allow the delegation to external libraries. This has been made possible while still keeping the run-time checks for pre/post-conditions & invariants, run-time type checking and debugging. Just as important the presented *RCI* allows external software to control the interpreter and execute statements in a running model, thereby allowing e.g. interactive GUIs to affect the model.

Our example has shown how the two presented methods can be used to create an easy understandable interactive graphical representation of the modelled system via Java GUI libraries, and how the model configuration can be built from data stored in a database. The *ECI* was used to update the graphical representation with changes in the model, in order to produce the animations, and to synchronize the time between the model and the animation. The *RCI* allowed the Java application to load different routes, start the animation and change the inflow through button presses on the GUI. It also enabled passing of data, from the database to the model, for validation of road connections and bus routes, as well as time synchronization. The Java side requires a bit more implementation effort, depending on the task to be performed. Simple invocation of Java methods is easily reached, but creating intermediate proxies to a running system or creating animated graphics might require more Java expertise. In order to provide better graphical prototyping capabilities an area of future development is in improving the tool support for graphical representations. As the presented approach requires a fair amount of Java programming to create the graphics and glue code, it would be interesting to utilize the possibilities of the approach to create visualisations easier, such as is seen in B-Motion Studio. Generation of specific visualisations as shown in the case study might be difficult to archive,

but generated GUI implementations which allow for input and output values to specific operations in a model, could be used for simpler interactions.

We believe that the approach described in this paper can be used as inspiration for tool builders of other formal methods that have implemented interpreters for executable subsets [1,2], as the principles are not specific to VDM.

Acknowledgments. We would like to thank Nick Battle and Joey Coleman for their invaluable input to this paper.

References

1. Breuer, P., Bowen, J.: Towards Correct Executable Semantics for Z. In: Bowen, J., Hall, J. (eds.) Z User Workshop, pp. 185–209. Springer (1994)
2. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J.F.: The Maude System. In: Narendran, P., Rusinowitch, M. (eds.) RTA 1999. LNCS, vol. 1631, pp. 240–243. Springer, Heidelberg (1999)
3. Fitzgerald, J.S., Larsen, P.G., Verhoef, M.: Vienna Development Method. In: Wah, B. (ed.) Wiley Encyclopedia of Computer Science and Engineering. John Wiley & Sons, Inc. (2008)
4. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs for Object–oriented Systems. Springer, New York (2005)
5. Fitzgerald, J., Larsen, P.G., Sahara, S.: VDMTools: Advances in Support for Formal Modeling in VDM. ACM Sigplan Notices 43(2), 3–11 (2008)
6. Fröhlich, B., Larsen, P.G.: Combining VDM-SL Specifications with C++ Code. In: Gaudel, M.-C., Wing, J.M. (eds.) FME 1996. LNCS, vol. 1051, pp. 179–194. Springer, Heidelberg (1996)
7. Gallasch, G., Kristensen, L.M.: Comms/CPN: A Communication Infrastructure for External Communication with Design/CPN. In: 3rd Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN 2001), pp. 75–90. DAIMI PB-554, Aarhus University (August 2001)
8. Group, T.V.T.: VDM Toolbox API. Tech. rep., CSK Systems (January 2008)
9. Ladenberger, L., Bendisposto, J., Leuschel, M.: Visualising Event-B Models with B-Motion Studio. In: Alpuente, M., Cook, B., Joubert, C. (eds.) FMICS 2009. LNCS, vol. 5825, pp. 202–204. Springer, Heidelberg (2009)
10. Larsen, P.G., Battle, N., Ferreira, M., Fitzgerald, J., Lausdahl, K., Verhoef, M.: The Overture Initiative – Integrating Tools for VDM. ACM Software Engineering Notes 35(1) (January 2010)
11. Lausdahl, K., Larsen, P.G., Battle, N.: A Deterministic Interpreter Simulating a Distributed Real Time System Using VDM. In: Qin, S., Qiu, Z. (eds.) ICFEM 2011. LNCS, vol. 6991, pp. 179–194. Springer, Heidelberg (2011)
12. Plat, N., Larsen, P.G.: An Overview of the ISO/VDM-SL Standard. Sigplan Notices 27(8), 76–82 (1992)
13. Reenskaug, T.: Models - Views - Controllers. Tech. rep., Xerox Parc (December 1979)
14. Verhoef, M., Larsen, P.G., Hooman, J.: Modeling and Validating Distributed Embedded Real-Time Systems with VDM++. In: Misra, J., Nipkow, T., Karakostas, G. (eds.) FM 2006. LNCS, vol. 4085, pp. 147–162. Springer, Heidelberg (2006)
15. Westergaard, M., Kristensen, L.M.: The Access/CPN Framework: A Tool for Interacting with the CPN Tools Simulator. In: Franceschinis, G., Wolf, K. (eds.) PETRI NETS 2009. LNCS, vol. 5606, pp. 313–322. Springer, Heidelberg (2009)

Extending the Test Template Framework to Deal with Axiomatic Descriptions, Quantifiers and Set Comprehensions

Maximiliano Cristiá[1] and Claudia Frydman[2]

[1] CIFASIS and UNR, Rosario, Argentina
[2] CIFASIS-LSIS and AMU, Marseille, France
cristia@cifasis-conicet.gov.ar, claudia.frydman@lsis.org

Abstract. The Test Template Framework (TTF) is a method for model-based testing (MBT) from Z specifications. Although the TTF covers many features of the Z notation, it does not explain how to deal with axiomatic descriptions, quantifiers and set comprehensions. In this paper we extend the TTF so it can process specifications including these features. The techniques presented here may be useful for other MBT methods for the Z notation or for other notations such as Alloy and B, since they use similar mathematical theories.

1 Introduction

The Test Template Framework (TTF) is a model-based testing (MBT) method [1, 2], used mainly for unit testing. MBT is a well-known technique aimed at testing software systems by analysing a formal model [3, 4]. MBT approaches start with a formal model or specification of the software, from which test cases are generated. These techniques have been developed and applied to models written in different formal notations such as Z [1], finite state machines and their extensions [5], B [6], algebraic specifications [7], and so on. The fundamental hypothesis behind MBT is that, as a program is correct if it satisfies its specification, then the specification is an excellent source of test cases.

Our group was the first in providing tool support for the TTF by implementing Fastest [8–10], and in extending the TTF beyond test case generation [11, 12]. Furthermore, we have applied Fastest and the TTF to several industrial-strength case studies [8, 13, 14]. The tool greatly automates tactic application, testing tree generation, testing tree simplification, and test case generation.

In 2008 we wrote a Z specification [14] of a significant portion of the ECSS-E-70-41A aerospace standard [15]. This is a medium-sized specification comprising 74 pages and more than 2,000 lines of Z. It is the largest Z specification we have written so far to test and validate Fastest. As a matter of comparison, the Tokeneer specification has only 46 lines more, while it is recognized as a full-fledged, industrial-strength formal specification [16]. The ECSS-E-70-41A formal specification comprises the minimum capability sets of 6 of the 16 services described in the standard. The model includes 25 state variables with 16 of

J. Derrick et al. (Eds.): ABZ 2012, LNCS 7316, pp. 280–293, 2012.

a relational type, of which 6 are higher-order functions and 3 are defined by referencing schema types. It also contains 28 axiomatic descriptions, some of which define operators whose domain are higher-order functions and schema types. To complicate things even more, this specification defines a number of set comprehensions and lambda expressions that influence critical outputs—for example, the report of housekeeping data of a satellite sent to ground upon request. Finally, some operations include quantified formulas.

Axiomatic descriptions, quantified formulas and set comprehensions were not considered in the original presentation of the TTF nor in Fastest. In this paper, we propose some techniques within the philosophy of the TTF and preserving a good deal of automation that extend the TTF so it can process specifications including these features. Currently, Fastest provides limited tool support for some classes of axiomatic descriptions—those referred as classes $\mathcal{C}$, $\mathcal{S}$ and $\mathcal{O}$ in Sect. 3—and it implements testing tactics for quantified formulas—those referred as WEQ, SEQ and CARD in Sect. 4. Therefore, so far, we have only been able to manually apply the techniques presented in this paper to the ECSS-E-70-41A formal specification—and automatically to some toy examples. Given that these techniques are aligned with the TTF, their full implementation will preserve the degree of automation currently featured by Fastest.

The paper is structured as follows. Section 2 describes the motivations for extending the TTF. The solution we propose for axiomatic descriptions is based on classifying them according to their intended meaning. Hence, in Sect. 3 we present a taxonomy of axiomatic descriptions and how each category should be processed. Section 4 focuses on the problem posed by quantifications, and Sect. 5 on set comprehensions and lambda expressions. Finally, in Sect. 6 we present our concluding remarks.

2 Some Extensions to the Test Template Framework

The TTF and Fastest have been thoroughly presented in many papers [2, 1, 8, 9]. In this section we focus on some difficulties appearing in the TTF when the Z specification being analysed includes axiomatic descriptions, quantifications or set comprehensions. Here we treat the TTF and Fastest as synonyms.

Given a Z specification, users have to select those operation schemas for which they want to generate test cases. As with other MBT methods, the TTF first generates test cases at the specification level, that are later refined to test the implementation corresponding to that specification [12]. In this paper we work only at the specification level. For each selected schema users indicate a set of testing tactics to be applied to it. The first testing tactic partitions the input space of the operation into a set of test specifications—i.e. test conditions or test objectives [3]. The second testing tactic partitions one or more of these test specifications, into more test specifications. The other testing tactics continue with this process. The net effect is a progressive partition of the input space of the operation into test specifications that are more restrictive than the previous ones. A test case is a witnesses satisfying the predicate of a leaf test specification.

Test specifications are Z schemas like the following one:

$$VerifyCmd_4^{SP} == [VerifyCmd_1^{DNF} \mid cmd? \in checksum \land proc1 \neq \emptyset \land proc2 \neq \emptyset \land proc1 \cap proc2 = \emptyset]$$

where *VerifyCmd* is the name of an operation selected by the user; $VerifyCmd_1^{DNF}$ is the test specification that was partitioned by applying the Standard Partitions (SP) testing tactic; *cmd?*, *proc*1 and *proc*2 are input and state variables declared in *VerifyCmd*; and *checksum* is the following axiomatic description:

$$checksum : \mathbb{P}\, FRAME$$

where *FRAME* is a given type.

Fastest generates all the test specifications automatically once users have indicated what testing tactics they want to apply to operations. Since testing tactic application means, essentially, conjoining predicates, it is not unusual to find unsatisfiable test specifications. These test specifications must be eliminated [9]. For the remaining ones, at least one test case must be generated. A test case for a given test specification is a Z schema restricting all the free variables to take one and only one value. In any MBT method, this process is intended to be as automatic as possible as hundreds of test specifications may be generated for a single specification. Fastest implements a sort of satisfiability algorithm for a significant portion of the Z Mathematical Toolkit (ZMT), that, according to our experiments, in average finds test cases for 80% of the satisfiable test specifications [8].

2.1 Axiomatic Descriptions

At this point some questions arise. Given that the satisfiability of $VerifyCmd_4^{SP}$ depends on the value of *checksum*, when should the algorithm to eliminate unsatisfiable test specifications be run? Is it reasonable for Fastest to automatically bind any value to *checksum*? What if users want an implementation for a particular value for it? Would Fastest generate test cases for that implementation or for any of its family [17, pages 36–38 and 143]? Currently, test cases are generated independently for each test specification—i.e. Fastest asks for an instantiation for each and every test specification. Can it be still done in this way in the presence of axiomatic descriptions like *checksum*? Clearly, two test cases cannot bind different values to *checksum* because they would belong to different members of the family of specifications. What if the specification includes an axiomatic description like the following one?

$$root : USER$$

Is *root* intended to be a constant or a variable? And, what if the specification includes the next one?

$$sum : \operatorname{seq}\mathbb{Z} \to \mathbb{Z}$$

$$sum\langle\rangle = 0$$
$$\forall\, s : \operatorname{seq}\mathbb{Z};\ n : \mathbb{Z} \mid s \neq \langle\rangle \bullet sum(s \frown \langle n \rangle) = n + sum\, s$$

Should a value be generated for *sum*? Or should it be treated entirely different from *checksum* and *root*? Should it be treated as an operation and, thus, test cases have to be generated?

As it can be seen, the inclusion of axiomatic descriptions poses a number of issues to be discussed in order to faithfully extend the TTF.

2.2 Quantified Formulas

Let us turn our attention to quantifications. Z provides a number of operators in the ZMT to avoid explicitly writing quantified formulas. For instance, it is convenient to replace $(\forall x : \operatorname{dom} f \bullet f\, x \neq 0)$ by $0 \notin \operatorname{ran} f$, and $(\forall x : \operatorname{dom} f \mid x \in A \bullet f\, x \neq 0)$ by $0 \notin f(\!|\ A\ |\!)$. Both implicit and explicit quantifications usually lead to loops in the implementation. So it is worth to generate test cases to test these loops by analysing these formulas. In the TTF, mathematical operators are analysed by the Standard Partition (SP) testing tactic. That is, a standard partition can be bound to, say, the $\_(\!|\ \_\ |\!)$ operator such that it will generate test specifications asking for different values of both arguments. For instance, a possible standard partition for $f(\!|\ A\ |\!)$ can be: $f = \emptyset \land A = \emptyset$; $f \neq \emptyset \land A = \emptyset$; $f = \emptyset \land A \neq \emptyset$; $f \neq \emptyset \land A \neq \emptyset \land \operatorname{dom} f \cap A = \emptyset$; and so forth. In other words, each of these partitions will exercise the potential loop implementing the operator in different ways. We would like to follow a similar approach for explicit quantified formulas. That is, we would like to have one or more testing tactics associated to quantifications that would yield test specifications that, in turn, would exercise the corresponding potential loop in different ways. For example, a quantification appearing in the ECSS-E-70-41A formalization is the following[1]:

$$\forall i : \operatorname{dom} sa? \bullet \\ sa?\, i + len?\, i \leq sizes\, m? \land cs?\, i = check(dt?\, i) \land len?\, i \leq \#(dt?\, i)$$

where $sa?$ is of type $\operatorname{seq}\mathbb{N}$, and *check* is an axiomatic description. Therefore, it would be desirable to generate test specifications that would test the implementation with $sa?$'s of different lengths.

Quantified formulas over potentially infinite sets pose a problem for any satisfiability algorithm. Hence, the approach we followed is to generate at least some test specifications where the potentially infinite set is replaced by one or more finite ones. In doing so the quantified formula is equivalent to either an unquantified conjunction or disjunction. But this brings in another issue. The first testing tactic applied by Fastest is Disjunctive Normal Form (DNF) [8]. All the other testing tactics in Fastest conjoin more atomic predicates to a given test specification. Hence, at the end, all test specifications are conjunctions of atomic predicates. Some key algorithms of Fastest rely on all test specifications having that property. Therefore, if we define testing tactics to deal with quantified formulas, they should also write the resulting predicates in DNF.

[1] Some of the names used in the formalization of the ECSS-E-70-41A standard have been changed with respect to the original specification due to space restrictions.

2.3 Set Comprehensions and Lambda Expressions

In the ECSS-E-70-41A formalization we heavily used complex set comprehensions and lambda expressions. For instance, we have the following schema:

PeriodicSampOnce

$$\Xi Housekeeping;\ \Xi Time$$
$$sOP : SID \pfun PGMODE \pfun PNAME \pfun PVAL$$
$$rS : \mathbb{P}\, SID$$
$$pSO : SID \pfun PNAME \pfun PVAL$$

$$rS = \{s : hES \mid (hRD\ s).m = p \land t = hCCI\ s + (hRD\ s).ci * dMI\}$$
$$pSO = (\lambda\, s : rS \bullet (\lambda\, p : \mathrm{dom}(hRD\ s).ns \mid (hRD\ s).ns\ p = 1 \bullet hSV\ s\ p\ t))$$
$$sOP = (\lambda\, s : rS \bullet (\lambda\, m : \{pm\} \bullet pSO\ s))$$

which is the simplest one in an operation defined by five others schemas like *PeriodicSampOnce*. Note that rS and pSO are referenced in the definition of sOP. sOP is later assembled with other similar variables declared in the other schemas to produce a single output for the operation. Therefore, the definition of the operation in which this schema participates is, essentially, an extremely complex lambda expression that is bound to an output variable. In summary, the operation has a trivial logical structure, while all its complexity lies inside the lambda expressions and set comprehensions. None of the testing tactics defined in the TTF would produce the desired results since none of them is prepared to work with bound variables. Furthermore, the implementation of these complex expressions will likely be very complex too, thus making it imperative to test it thoroughly.

Therefore, we need one or more testing tactics that generate significant test specifications for this kind of expressions. The approach we followed is to propagate the complexity inside the expressions to the outside, and then apply existing testing tactics. For example, if we have $\{x : X \mid P(x) \lor Q(x) \bullet expr(x)\}$, it can be rewritten as $\{x : X \mid P(x) \bullet expr(x)\} \cup \{x : X \mid Q(x) \bullet expr(x)\}$, making it possible to apply SP to $\cup$.

3 A Taxonomy of Axiomatic Descriptions

In Z, axiomatic descriptions can serve many purposes [17, page 143]. For example, an axiomatic description can be used just to give a name to an integer constant, or it can be used to define a function summing all the components of a sequence of integers. As we have said in Sect. 2.1, in our opinion not all the axiomatic descriptions can be treated in the same way with respect to the TTF. Therefore, we consider that a key step towards their inclusion in the TTF is to define a taxonomy for axiomatic descriptions based on their syntax—aiming at capturing their intended use and semantics. In a second step we define how each category will be processed in the TTF. The ultimate goal is making test case generation as automatic as possible in the presence of axiomatic descriptions.

3.1 Given Type Constants ($\mathcal{C}$)

If T is a given or basic type and we have:

$$x : T$$

then x is said to be a constant of type T. An example is *root* (Sect. 2.1).

Axiomatic descriptions of this kind are regarded as constants of their corresponding types. Therefore, they will be used as values for variables appearing in test specifications. Two members of $\mathcal{C}$ of the same type will be considered as different constants. For example, if *admin* is an axiomatic description of type *USER*, then $admin \neq root$ holds. However, at the same time, if an operation declares $usr? : USER$, testers can generate test specifications asking for $usr? = root$, $usr? = admin$ and $usr? \notin \{root, admin\}$. For $\mathcal{C}$ no user action is required.

3.2 Synonyms ($\mathcal{S}$)

A synonym is any axiomatic description matching any of the following:

$$\frac{x : T}{x = expr} \qquad \frac{x : T}{\forall\, y : U \bullet x(y) = expr(y)}$$

where T and U are any types and *expr* is any expression. x may depend on some y only if T is a structured type, in which case U is part of T's definition. *expr* may depend on y and, possibly, on other axiomatic descriptions. We call *expr* the definition of x. An example of this kind is the following one taken from the ECSS-E-70-41A formalization:

$$\frac{lastRepVal : (TIME \pfun PVAL) \to PVAL \to \mathbb{N} \to \operatorname{seq} PVAL}{\begin{array}{l}\forall\, h : TIME \pfun PVAL;\ v : PVAL;\ r : \mathbb{N} \bullet \\ \qquad lastRepVal\ h\ v\ r = ((\#h - r + 2\,..\,\#h) \upharpoonleft squash\ h) \frown \langle v \rangle\end{array}}$$

Axiomatic descriptions in this category can be treated in two ways:

1. Simply replace the axiomatic descriptions by their definitions when they appear in test specifications. If x is of the quantified form, replace it by its definition substituting its formal parameter by the real one. No user action is needed for $\mathcal{S}$, in this case.
2. Users may want to generate test cases for *expr* as if it were an operation. However, this is not always applicable. For example, it makes sense to do it with *lastRepVal* but it makes no sense with the following one:

$$\frac{administrators : \mathbb{P}\, USER}{administrators = \{root, admin\}}$$

In general, this decision must be left to users; Fastest should do as in 1 by default. If the user decides to generate test cases for x, then he/she can use any of the available testing tactics. However, when these axiomatic descriptions appear in a test specification they have to be processed as in 1.

3.3 Equivalences ($\mathcal{E}$)

An equivalence is any axiomatic description matching the following:

$$\begin{array}{|l} x : T \\ \hline \forall\, y : U \bullet P(x, y) \Leftrightarrow Q(y) \end{array}$$

where T and U are any types and P and Q are predicates. Q may depend also on other axiomatic descriptions. We say Q is the definition of x. *falied* is an instance of this category borrowed from the ECSS-E-70-41A formalization:

$$\begin{array}{|l} failed : \mathbb{P}((TIME \pfun PVAL) \times PVAL \times CheckDef) \\ \hline \forall\, h : TIME \pfun PVAL;\ v : PVAL;\ d : CheckDef \bullet \\ \qquad (h, v, d) \in failed \Leftrightarrow avrDelta(lastRepVal\ h\ v\ d.rep) < d.low \end{array}$$

This class is treated as $\mathcal{S}$, only considering that Q is the definition of x.

Note that if we would have defined *failed* as a set comprehension, then it would have fallen in $\mathcal{S}$ thus replacing *failed* for its definition in test specifications. Hence, later, the testing tactics defined in Sect. 5 can be applied. In either way, the expression can be properly treated.

3.4 Inductive Definitions ($\mathcal{ID}$)

We say that an axiomatic description is an inductive definition if it has the following form:

$$\begin{array}{|l} x : T \\ \hline \forall\, y_1 : U_1 \bullet x(E_1(y_1)) = expr_1 \\ \ldots\ldots\ldots \\ \forall\, y_n : U_n \bullet x(E_n(y_n)) = expr_n \end{array}$$

where T, $U_1, \ldots, U_n$ are types for which an induction principle is defined—i.e. free types, $\mathbb{N}$, $\mathrm{seq}\, X$ [17, pages 83, 114 and 123], and finite sets [18, page 59]—, $E_1, \ldots, E_n$ are n structurally different expressions of the same inductive type W, and $expr_1, \ldots, expr_n$ are expressions. Any of the quantifiers might be absent in which case the corresponding E expression will be constant. It is assumed that there are no mutually recursive definitions and no definition is infinitely recursive. An element in $\mathcal{ID}$ is *sum* in Sect. 2.1.

As with $\mathcal{S}$, elements in this category can be processed in the same two ways. The difference being that a symbolic evaluation of these axiomatic descriptions is performed when test cases are generated.

3.5 All Other Axiomatic Descriptions ($\mathcal{O}$)

Any axiomatic description not falling within any of the previous categories, belongs to this category. For instance, *checksum* in Sect. 2. An element in this category can be processed in two ways:

1. Users can provide a constant value for it. This value will be used to generate test cases for all the test specification where the axiomatic description appears. The value must help satisfy the predicate part of all axiomatic descriptions in which it appears.
2. Alternatively, Fastest can choose any value for it. Although this way of treating these axiomatic descriptions may increase the degree of automation, it can severely complicate the generation of test cases because some test specifications may become unsatisfiable, when they may not for other values. Furthermore, without any further information Fastest may choose an odd value with respect to the implementation that is going to be tested.

4 Testing Tactics for Quantifications

As we have said in Sect. 2.2, we have decided to approach the generation of test cases when quantifications are used in operations, by defining some testing tactics specially tailored to deal with such predicates. So far, the TTF had treated quantifications as atomic predicates making it very difficult, or even impossible, to generate test cases to exercise the corresponding implementation sentences—usually loops. Hence, in the following sections we introduce these new testing tactics for quantified formulas. These testing tactics can be applied only when: (i) the quantified formula includes predicates depending only on input or before-state variables, and (ii) the sets over which the bound variables ranges, depend on the same kind of variables. These restrictions are reasonable since the whole goal of the TTF is to produce a partition of the input space of the operation, which is defined by all the input and before-state variables.

4.1 Weak Existential Quantifier (WEQ)

Conceptually, this testing tactic transforms a quantification over a potentially infinite set into a quantification over a user-provided set extension. Since an existential quantification over a finite set is equivalent to a disjunction, then WEQ first transforms the existential quantification into a disjunction. Then it writes the disjunction into DNF and finally it generates as many test specifications as terms the DNF has plus one more characterized by the negation of the other predicates. Hence, in order to apply WEQ the user has to indicate the quantified predicate and a set extension for each bound variable—or a set extension for each type of the bound variables.

The example depicted in Fig. 1 helps to understand how WEQ works. Assume $M : \mathbb{P}\,\mathbb{N}$ and $H : \mathrm{seq}\,\mathbb{Z}$ are two input or before-state variables. WEQ_1 and WEQ_2 say that a test case must be generated when $x = 4$ and $y = \langle 4 \rangle$; WEQ_3 and

Original predicate	$\exists\, x : M;\ y : H \bullet x > w \wedge (y \neq \langle\rangle \Rightarrow head\ y > x)$
User-provided set extensions	$x \leftarrow \{4, 6\}, y \leftarrow \{\langle 4\rangle\}$
First transformation	$(4 > w \wedge (\langle 4\rangle\ \neq \langle\rangle \Rightarrow head\ \langle 4\rangle\ > 4))$ $\vee\ (6 > w \wedge (\langle 4\rangle\ \neq \langle\rangle \Rightarrow head\ \langle 4\rangle\ > 6))$
Second transformation (DNF)	$4 > w \wedge \neg\ \langle 4\rangle \neq \langle\rangle$ $\vee\ 4 > w \wedge head\ \langle 4\rangle\ > 4$ $\vee\ 6 > w \wedge \neg\ \langle 4\rangle \neq \langle\rangle$ $\vee\ 6 > w \wedge head\ \langle 4\rangle\ > 6$
New test specifications	$WEQ_1 \rightarrow 4 > w \wedge \langle 4\rangle = \langle\rangle$ $WEQ_2 \rightarrow 4 > w \wedge head\ \langle 4\rangle\ > 4$ $WEQ_3 \rightarrow 6 > w \wedge \langle 4\rangle = \langle\rangle$ $WEQ_4 \rightarrow 6 > w \wedge head\ \langle 4\rangle\ > 6$ $WEQ_5 \rightarrow (\exists\, x : M;\ y : H \mid x \notin \{4, 6\} \wedge y \notin \{\langle 4\rangle\} \bullet$ $x > w \wedge (y \neq \langle\rangle \Rightarrow head\ y > x))$

Fig. 1. Generating test specifications by applying WEQ

WEQ_4 say the same but with $x = 6$; and WEQ_5 says there may be other test cases to derive from the formula. Note that WEQ_1 and WEQ_3 will not produce abstract test cases since they are unsatisfiable. Also note that, in general, no satisfiability algorithm will be able to automatically generate an abstract test case for all test specifications like WEQ_5, due to the presence of the quantification over potentially infinite sets.

Likely, an existential quantification will be implemented as an iteration statement that will be abandoned when the first value satisfying its condition is found. Therefore, it is important to test this statement by making it execute zero, one or more iterations. Furthermore, it is important to test the inner clause with different values. WEQ allows all of this by letting users restrict the quantification over a suitable, finite set.

It should be noted that this tactic might not produce a partition of the test specifications—if this is unacceptable, then see the next section.

4.2 Strong Existential Quantifier (SEQ)

This tactic is a stronger form of WEQ since it always generates a partition of the test specifications where it is applied. SEQ conjoins the following predicate to the i^{th} test specification produced by WEQ, except the last one:

$$\neg\ (\exists\, x_1 : T_1, \ldots, x_n : T_n \mid x_1 \neq v_i^1 \wedge \cdots \wedge x_n \neq v_i^n \bullet P(x_1, \ldots, x_n, x))$$

where $x_1 : T_1, \ldots, x_n : T_n$ are the quantified variables and their types, $v_i^1, \ldots, v_i^n$ are the values making up the i^{th} combination of values taken from the set extensions provided by the user for each quantified variable, and P is the quantified

predicate. For instance, in the example shown in Fig. 1, SEQ would generate the following test specifications:

$$
\begin{aligned}
SEQ_1 \rightarrow\ & 4 > w \land \langle 4 \rangle = \langle\rangle \\
& \land \neg\, (\exists\, x : M;\ y : H \mid x \neq 4 \land y \neq \langle 4 \rangle \bullet x > w \land y \neq \langle\rangle \Rightarrow head\ y > x) \\
SEQ_2 \rightarrow\ & 4 > w \land head\ \langle 4 \rangle\ > 4 \\
& \land \neg\, (\exists\, x : M;\ y : H \mid x \neq 4 \land y \neq \langle 4 \rangle \bullet x > w \land y \neq \langle\rangle \Rightarrow head\ y > x) \\
SEQ_3 \rightarrow\ & 6 > w \land \langle 4 \rangle = \langle\rangle \\
& \land \neg\, (\exists\, x : M;\ y : H \mid x \neq 6 \land y \neq \langle 4 \rangle \bullet x > w \land y \neq \langle\rangle \Rightarrow head\ y > x) \\
SEQ_4 \rightarrow\ & 6 > w \land head\ \langle 4 \rangle\ > 6 \\
& \land \neg\, (\exists\, x : M;\ y : H \mid x \neq 6 \land y \neq \langle 4 \rangle \bullet x > w \land y \neq \langle\rangle \Rightarrow head\ y > x) \\
SEQ_5 \rightarrow\ & (\exists\, x : M;\ y : H \mid \\
& \qquad x \notin \{4, 6\} \land y \notin \{\langle 4 \rangle\} \bullet x > w \land y \neq \langle\rangle \Rightarrow head\ y > x)
\end{aligned}
$$

However, in general, no satisfiability method will be able to automatically generate abstract test cases for any of these test specifications due to the presence of the quantification over an infinite set. Therefore, in spite of WEQ not producing a partition, and thus potentially generating the same test case more than once, it will allow a satisfiability algorithm to find at least some test cases some times. However, SEQ is still valuable since users may provide, manually, test cases satisfying these test specifications, if WEQ is too weak for their needs.

4.3 Universal Quantifications

In order to produce a partition of a universal quantification, we propose a testing tactic, called CARD, that considers different cardinalities for the sets over which the bound variables range. Then, given a universal quantification such as:

$$\forall\, x_1 : S_1, \ldots, x_n : S_n \bullet P(x_1, \ldots, x_n, \ldots) \tag{1}$$

where $S_1, \ldots, S_x$ are sets, users may apply CARD by indicating a limit to the cardinality for each S_i with $i \in 1..n$. If these limits are $M_1, \ldots, M_n$ for $S_1, \ldots, S_n$, respectively, then CARD generates $(M_1 + 2) \times \cdots \times (M_n + 2)$ test specifications characterized by a predicate of the following form:

$$C_1 \land C_2 \land \cdots \land C_n$$

where each C_i is either $\#S_i = k_i$ with $k_i \in 0\,..\,M_i$, or $\#S_i > M_i$. Given that $\#$ can only be applied to finite sets, then Fastest understands the expression $\#S_i$ as: replace S_i in the quantified formula with a set $A_i : \mathbb{F}\, S_i$ whose cardinality verifies the corresponding restriction. This interpretation is consistent with respect to the way Fastest finds test cases, as it always proceeds by considering a finite model for each test specification [8]. The example shown in Fig. 2 helps to

Original predicate	$\forall p : active;\ h : \mathrm{dom}\, f \bullet P(p, h, \dots)$
User-provided cardinalities	$active \leftarrow 1, \mathrm{dom}\, f \leftarrow 1$
New test specifications	$CARD_1 \rightarrow \#active = 0 \land \#\,\mathrm{dom}\, f = 0$
	$CARD_2 \rightarrow \#active = 1 \land \#\,\mathrm{dom}\, f = 0$
	$CARD_3 \rightarrow \#active > 1 \land \#\,\mathrm{dom}\, f = 0$
	$CARD_4 \rightarrow \#active = 0 \land \#\,\mathrm{dom}\, f = 1$
	$CARD_5 \rightarrow \#active = 1 \land \#\,\mathrm{dom}\, f = 1$
	$CARD_6 \rightarrow \#active > 1 \land \#\,\mathrm{dom}\, f = 1$
	$CARD_7 \rightarrow \#active = 0 \land \#\,\mathrm{dom}\, f > 1$
	$CARD_8 \rightarrow \#active = 1 \land \#\,\mathrm{dom}\, f > 1$
	$CARD_9 \rightarrow \#active > 1 \land \#\,\mathrm{dom}\, f > 1$

Fig. 2. Generating test specifications by applying CARD

understand this testing tactic. For instance, when Fastest tries to find a test case for $CARD_8$ it will replace the original predicate by (assuming: $active$: $\mathbb{P}\, PROCESS$ and $f : \mathbb{N} \pfun Report$):

$$\forall p : \{p_1\};\ h : \{5, 9\} \bullet P(p, h, \dots)$$

yielding a quantification whose satisfiability is easier to determine.

In this way, CARD will produce test specifications which ultimately will execute the statement corresponding to the quantified formula, likely an iteration, a different number of times.

5 Testing Tactics for Set Comprehensions

Any lambda expression can be written as a set comprehension [17, page 58]:

$$(\lambda\, x : X \mid P(x) \bullet f(x)) \equiv \{x : X \mid P(x) \bullet x \mapsto f(x)\}$$

where f is an expression depending on x and possibly on other free variables. Therefore, the ideas presented in this section can also be applied to lambda expressions. In turn, the most general form of a set comprehension in Z is:

$$\{x : X \mid P(x) \bullet expr(x)\}$$

where P is a predicate and $expr$ is an expression, both depending on the bound variable and possibly on some free variables. The type of the set comprehension is given by the type of $expr$ [17, page 57].

Clearly, the complexity of a set comprehension lies on the complexity of both P and $expr$. As we have said in Sect. 2.3, the idea is to move the complexity of P to the outside of the set comprehension. More precisely:

1. Write P in DNF: $P_1 \vee \cdots \vee P_n$, where each P_i is a conjunction of literals.
2. Rewrite the set comprehension as a set union:

$$\{x : X \mid P(x) \bullet expr(x)\} \equiv \\ \{x : X \mid P_1(x) \bullet expr(x)\} \cup \cdots \cup \{x : X \mid P_n(x) \bullet expr(x)\}$$

3. Rewrite each term of the set union as a set intersection.
4. Apply SP to one or more $\cup$ or $\cap$.

Alternatively, apply SP inside the set comprehension rewriting it as:

$$\{x : X \mid P(x) \wedge (Q_1(x) \vee \cdots \vee Q_n(x)) \bullet expr(x)\}$$

where each Q_i is the i^{th} predicate stipulated by the corresponding standard partition. Then write $P(x) \wedge (Q_1(x) \vee \cdots \vee Q_n(x))$ in DNF and do as above. Yet another alternative is to apply SP to operators appearing in *expr* instead of or apart from P. All these can be combined as is customary in the TTF to further partition previous test specifications. The net effect is a coverage similar to the one delivered by the TTF for other constructions.

6 Concluding Remarks

There are some MBT methods, besides the TTF, for the Z notation [19–24] but none of them approaches axiomatic descriptions, quantified formulas and set comprehensions. Extending any MBT method for the Z notation to deal with these concepts is important because large specifications will include them. Then, all these MBT methods may benefit from our results. Furthermore, MBT methods for other notations such as Alloy, B and VDM may also take advantage of these results since these languages use similar mathematical theories.

Although the paper deals with three somewhat unrelated issues, there is a common, underlying theme: automation. That is, the rules proposed here to process axiomatic descriptions, quantifications and set comprehensions were devised to preserve the degree of automation currently featured by Fastest. Variants of these rules may be proposed but likely they will render the tool less automatic.

Regarding axiomatic descriptions, we conclude that they should be classified according to their intended use before processing them to produce test cases. Although the same language construct is used to define all of them, there are key differences, for example, between declarations such as *root* (Sect. 2.1) and *failed* (Sect. 3.3), making it dangerous to treat them in the same way. The taxonomy presented here may be extended, refined or modified but axiomatic descriptions cannot be treated all in the same way.

It may be argued that recent advances in decision procedures and SMT solving should be used to approach quantified formulas. We have two counterarguments to this point: (a) besides finding a witness satisfying a quantified formula, a partition based on its analysis must be generated in order to get a good coverage

of its implementation, and this cannot be done with SMT solvers; and (b) our first results on applying SMT solvers to the TTF show that these tools are not immediately or trivially useful for it [25].

Quantifiers are treated in [26] but for the VDM notation which is based on three value logic and where all sets must be finite—key differences with respect to Z. The rules proposed by Meudec: (i) take into consideration the fact that predicates or expressions can be undefined according to the VDM semantics; (ii) they lead to very long partitions even for basic quantified expressions; and (iii) apparently, these partitions cannot be controlled by the user as ours can. For all these reasons we decided to develop our own tactics to deal with quantifiers.

With regard to future work, we are working on the implementation of the results for set comprehensions and we are further investigating if SMT solvers can be used as a back-end to automate some of the results presented here.

Acknowledgements. This paper is a humble tribute to the memory of David Carrington, one of the creators of the Test Template Framework, who passed away in Australia on 7 January 2011 after a long battle with cancer.

References

1. Stocks, P., Carrington, D.: A Framework for Specification-Based Testing. IEEE Transactions on Software Engineering 22, 777–793 (1996)
2. Stocks, P.: Applying Formal Methods to Software Testing. PhD thesis, Department of Computer Science, University of Queensland (1993)
3. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann Publishers Inc., San Francisco (2006)
4. Hierons, R.M., Bogdanov, K., Bowen, J.P., Cleaveland, R., Derrick, J., Dick, J., Gheorghe, M., Harman, M., Kapoor, K., Krause, P., Lüttgen, G., Simons, A.J.H., Vilkomir, S., Woodward, M.R., Zedan, H.: Using formal specifications to support testing. ACM Comput. Surv. 41, 1–76 (2009)
5. Grieskamp, W., Gurevich, Y., Schulte, W., Veanes, M.: Generating finite state machines from abstract state machines. In: ISSTA 2002: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 112–122. ACM, New York (2002)
6. Legeard, B., Peureux, F., Utting, M.: A Comparison of the BTT and TTF Test-Generation Methods. In: Bert, D., P. Bowen, J., C. Henson, M., Robinson, K. (eds.) B 2002 and ZB 2002. LNCS, vol. 2272, pp. 309–329. Springer, Heidelberg (2002)
7. Bernot, G., Gaudel, M.C., Marre, B.: Software testing based on formal specifications: a theory and a tool. Softw. Eng. J. 6, 387–405 (1991)
8. Cristiá, M., Monetti, P.R.: Implementing and Applying the Stocks-Carrington Framework for Model-Based Testing. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM 2009. LNCS, vol. 5885, pp. 167–185. Springer, Heidelberg (2009)
9. Cristiá, M., Albertengo, P., Rodríguez Monetti, P.: Pruning testing trees in the Test Template Framework by detecting mathematical contradictions. In: Fiadeiro, J.L., Gnesi, S. (eds.) SEFM, pp. 268–277. IEEE Computer Society (2010)
10. Cristiá, M.: Fastest tool, `http://www.fceia.unr.edu.ar/~mcristia` (last access November 2011)

11. Cristiá, M., Plüss, B.: Generating natural language descriptions of Z test cases. In: Kelleher, J.D., Namee, B.M., van der Sluis, I., Belz, A., Gatt, A., Koller, A. (eds.) INLG, pp. 173–177. The Association for Computer Linguistics (2010)
12. Cristia, M., Hollmann, D., Albertengo, P., Frydman, C., Monetti, P.R.: A Language for Test Case Refinement in the Test Template Framework. In: Qin, S., Qiu, Z. (eds.) ICFEM 2011. LNCS, vol. 6991, pp. 601–616. Springer, Heidelberg (2011)
13. Cristiá, M., Santiago, V., Vijaykumar, N.: On comparing and complementing two MBT approaches. In: Vargas, F., Cota, E. (eds.) LATW, pp. 1–6. IEEE Computer Society (2010)
14. Cristiá, M., Albertengo, P., Frydman, C., Plüss, B., Rodríguez Monetti, P.: Applying the Test Template Framework to aerospace software. In: Proceedings of the 34th IEEE Annual Software Engineering Workshop, Limerik, Irland. IEEE Computer Society (2011)
15. ECSS: Space Engineering – Ground Systems and Operations: Telemetry and Telecommand Packet Utilization. Technical Report ECSS-E-70-41A, European Space Agency (2003)
16. Barnes, J., Chapman, R., Johnson, R., Widmaier, J., Cooper, D., Everett, B.: Engineering the Tokeneer enclave protection software. In: Proceedings of the IEEE International Symposium on Secure Software Engineering. IEEE (2006)
17. Spivey, J.M.: The Z notation: a reference manual. Prentice Hall International (UK) Ltd., Hertfordshire (1992)
18. Saaltink, M.: The Z/EVES mathematical toolkit version 2.2 for Z/EVES version 1.5. Technical report, ORA Canada (1997)
19. Ammann, P., Offutt, J.: Using formal methods to derive test frames in category-partition testing. In: Compass 1994: 9th Annual Conference on Computer Assurance, Gaithersburg, MD, pp. 69–80. National Institute of Standards and Technology (1994)
20. Hall, P.A.V.: Towards testing with respect to formal specification. In: Proc. Second IEE/BCS Conference on Software Engineering, Conference Publication, IEE/BCS, vol. 290, pp. 159–163 (1988)
21. Hierons, R.M., Sadeghipour, S., Singh, H.: Testing a system specified using Statecharts and Z. Information and Software Technology 43, 137–149 (2001)
22. Hierons, R.M.: Testing from a Z specification. Software Testing, Verification & Reliability 7, 19–33 (1997)
23. Hörcher, H.M., Peleska, J.: Using Formal Specifications to Support Software Testing. Software Quality Journal 4, 309–327 (1995)
24. Burton, S.: Automated Testing from Z Specifications. Technical report, Department of Computer Science – University of York (2000)
25. Cristiá, M., Frydman, C.: Applying SMT solvers to the Test Template Framework. In: Petrenko, A.K., Schlingloff, H. (eds.) Proceedings 7th Workshop on Model-Based Testing, Tallinn, Estonia, March 25. Electronic Proceedings in Theoretical Computer Science, vol. 80, pp. 28–42. Open Publishing Association (2012)
26. Meudec, C.: Automatic generation of software tests from formal specifications. PhD thesis, Queen's University of Belfast, Northern Ireland, UK (1997)

A Tool Chain for the Automatic Generation of *Circus* Specifications of Simulink Diagrams

Chris Marriott, Frank Zeyda, and Ana Cavalcanti

Department of Computer Science, University of York, UK
{chris.marriott,frank.zeyda,ana.cavalcanti}@cs.york.ac.uk

Abstract. Previous work described how to translate Simulink control law diagrams into *Circus* specifications to facilitate verification by refinement. This is not a trivial task; several tools have been developed to automate parts of the translation. This paper introduces a new tool chain that extends and integrates existing technology to cover the entire translation and cater for a larger set of diagrams. Our contributions include the integration of data types, generic definitions, and extension of the technique to model action and enabled subsystems. The tool chain has been validated using an industrial case study.

Keywords: Z, CSP, ClawZ, control law diagrams, verification.

1 Introduction

Control systems are commonly modelled using control law diagrams: a graphical notation with blocks and connecting wires. Each block represents a calculation or function, and can have state; wires connect inputs and outputs of blocks. Systems may be so complex that functionality is often defined in a number of separate diagrams, known as subsystems; these introduce a hierarchy.

As regulations for certification of safety-critical systems are being tightened, the use of formal methods is becoming increasingly encouraged. Various attempts have been made to express control law diagrams in formal languages [6, 3]. Particular attention has been given to diagrams in MATLABs Simulink [9], a *de facto* standard, especially in the automotive and avionics industries.

Circus [11] is a formal language capable of expressing state-rich concurrent systems based on Z [12], CSP [10], and a refinement calculus [5]. In [4], Cavalcanti *et al.* describe a formalised translation from Simulink diagrams to *Circus* models; it takes into account parallelism and independent flows of execution.

The main benefit of using *Circus* to encode Simulink diagrams is the ability to prove correctness of implementations through refinement. The work presented here extends the set of translatable diagrams, automates further the model generation technique of [4], and expands the set of programs we can prove correct. Code generation and diagram validation techniques that extend the static analysis in Simulink exist to satisfy different objectives from those we address here.

Our work extends and integrates a number of tools and associated techniques to support a single-click translation, and handle a larger set of diagrams.

J. Derrick et al. (Eds.): ABZ 2012, LNCS 7316, pp. 294–307, 2012.

As a result, users can produce *Circus* specifications without the need for indepth knowledge of multiple tools. We describe here the enhancements to existing tools and new methods that make this possible. In particular, we address the integration of data types and type-sensitive translation, the use of generic definitions, the use of a complex tool chain in the context of safety-critical systems, and techniques to model enabled and action subsystems.

Previously, there were two tools that supported the conversion of Simulink diagrams to *Circus* (namely ClawZ [2] and ClawCircus [13]). Each is driven individually with a significant amount of manual input. Expertise is required in Simulink, Z, *Circus*, and methods to bring these components together. With our tool chain, the amount of expert knowledge and manual input is reduced.

Additionally, the current technique does not cater for enabled and action subsystems. These are just like other subsystems in Simulink, which are defined by a sub-diagram, except they have enabling conditions that determine whether they are executed or not. They are used to control the flow of execution in a diagram and are commonly used in industrial applications. Here, we present a technique to model enabled and action subsystems in *Circus*.

The remainder of this paper is structured as follows. Section 2 presents preliminary material related to our work. Section 3 describes the translation from Simulink to *Circus*. Section 4 introduces enabled and action subsystems and describes how they can be expressed in *Circus*. Finally, Section 5 explains how the chain has been applied to a large industrial example not previously translatable, along with our conclusions and possible further work.

2 Background

This section describes Simulink diagrams, *Circus*, and existing tools.

Control Law Diagrams. An example control law diagram written in Simulink notation can be seen in Figure 1. It specifies a missile guidance subsystem used in the aerospace industry [9]. Individual blocks are boxes on the diagram, and perform their own unique function; arrows between blocks represent the communication of values. The small ovals are the inputs and outputs to the subsystem (Rm, Vc, AZ_d, ...). This example also contains an enabled subsystem (*Fuze*).

The example is used to locate an initial target position and then monitor the flight of the missile using closed-loop tracking to ensure it is reached; these calculations are performed within the custom *Guidance Processor* subsystem. The *Fuze* subsystem is used to control the detonation of the missile; it monitors the distance to the target and feeds back into another tracking subsystem.

Circus Language. Systems are specified through processes in *Circus*. Features from Z and CSP are available, including schemas, communication, parallelism and choice. Programming operators come from Morgan's refinement calculus.

The main constructs are channels, processes and actions. Channels are used to define communication events between processes. Processes contain state information and have a behaviour defined by actions. State is local, so that interaction

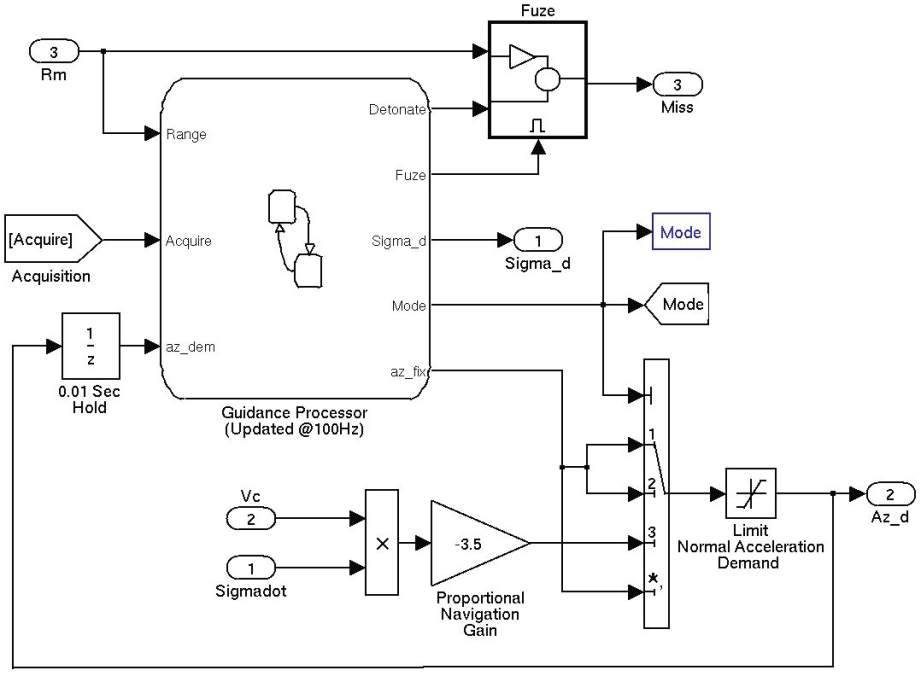


Fig. 1. Guidance Subsystem in Simulink [9]

can only occur through channels. A process can be defined explicitly, or by using operators of CSP for composition of other processes, such as parallelism.

An action can be defined as either a schema, which performs operations on the process state, a command in Dijkstra's guarded command language, or a CSP expression. Local actions are referenced by the main action, which specifies the behaviour of the process. More details about *Circus* can be found in [11].

ClawZ is a tool suite for verification of implementations of Simulink diagrams [2]. It translates diagrams into Z encoded for ProofPower-Z [7], a mechanical theorem prover. ClawZ has been used in industry and has reduced costs of verification [1].

In ClawZ models, schemas are used to define inputs, outputs, and state elements of blocks and subsystems. Only discrete-time blocks are translated because software is discrete. Schemas produced by ClawZ are defined in a library; attributes in diagrams are used to match blocks to corresponding library schemas.

Circus specifications use schemas defined by ClawZ to describe functionality; CSP describes the communication and behavioural aspects of the control law.

3 Translating Simulink Diagrams into *Circus* Specifications

This section describes our tool chain to translate Simulink diagrams into *Circus* specifications automatically. We describe all tools required and explain

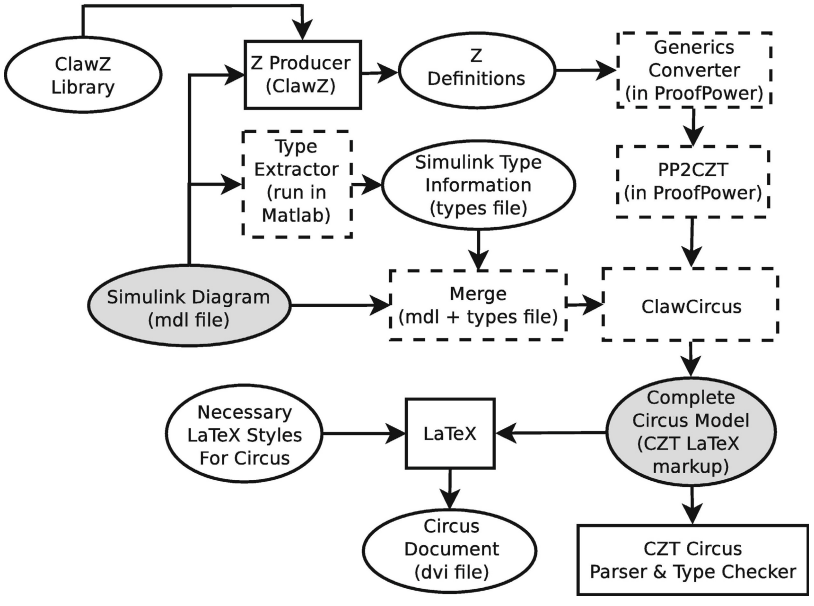


Fig. 2. Simulink to *Circus* Translation Process

how each was tailored or developed to achieve the integration in Figure 2; ovals represent files and libraries, and squares represent processes and tools. The dotted-borders indicate processes or tools adapted or developed to create the chain; the two shaded ovals are the Simulink input and *Circus* output files.

3.1 Z Producer, Generics Converter, and PP2CZT

The three processes in the top path in Figure 2 are used to produce and modify the necessary Z definitions for use in the *Circus* model. The Z Producer (part of ClawZ), is used to specify schemas for blocks automatically.

The ClawZ output is encoded for ProofPower-Z and can be used for verification once parsed. A problem arises, however, because the use of generics in ProofPower is different from that in standard Z (and *Circus*). ProofPower-Z allows partial instantiation of a generic definition: it is possible to use generic definitions in ProofPower-Z without instantiating all formal generic parameters.

The ProofPower-Z notation includes the universal type $\mathbb{U}$, which is the carrier set of a generic type ($\mathbb{U}[X] \mathrel{\widehat=} X$). The Z definitions produced by ClawZ use the universal type because of the lack of type information in the Simulink file. Data types are inferred automatically within ProofPower-Z and this remains true for the majority of definitions when treated as part of a standard Z specification. In some cases, however, inference of actual generic parameters is not possible; as an example, we consider the definition below of a Selector block.

Selector

$In1? : \mathbb{U}$

$Out1! : \mathbb{U}$

$Out1! = In1?\,(1)$

The Selector block takes a sequence of values ($In1?$) and selects a particular element ($Out1!$); in this example it is the first element. In ProofPower-Z, this schema is valid: it has an implicit formal generic parameter as there is not enough information to fully instantiate the type of $\mathbb{U}$. This parameter is not declared explicitly and the schema is not well typed according to the rules of standard Z.

To overcome this, we rewrite the definitions from ClawZ in standard Z using the new Generics Converter tool. The schema below has the same semantics as the previous example; it is a standard Z definition that introduces the type X as a formal generic parameter rather than using the universal type of ProofPower-Z. The conversion automatically infers, from the schema in the ClawZ output, that the type of the input ($In1?$) is a sequence of values. This is represented as a relation from an integer to a value of the generic type parameter.

Selector$[X]$

$In1? : \mathbb{Z} \leftrightarrow X$

$Out1! : X$

$Out1! = In1?\,(1)$

The Generic Converter traverses all definitions stored in a ProofPower-Z file and analyses their components to establish the type of definition and whether any implicit generic parameters exist. If none are found, the definition remains unchanged; however, upon finding generic parameters, the definition is re-constructed. The new definition contains the formal generic parameters explicitly.

The modified ClawZ output in standard Z is converted into CZT markup (used by the *Circus* parser) using the new PP2CZT tool within ProofPower-Z. It performs a syntactic translation of all definitions and schemas in the ClawZ output and automatically produces the CZT encoding. The translation relies on a set of mappings from the internal representation in ProofPower-Z to the text-based markup in CZT. All definitions in a ProofPower-Z document are considered individually; every component in the definition is then analysed, translated, and re-assembled in a new file to form the corresponding CZT definition.

3.2 Type Extractor and Merge

Blocks in Simulink have a set of input and output ports, each with a specific data type and dimension. Previously, the translation assumed that all components were one-dimensional, and used the ProofPower-Z $\mathbb{R}$ data type to define their types. This, however, is not a realistic assumption, and since data types in Simulink are different to those of *Circus*, a mapping between data types is necessary. Simulink uses data types such as *double, int8* and *uint8;* we represent these

in *Circus* as $\mathbb{R}$, $\mathbb{Z}$, and $\mathbb{N}$. Simulink uses multi-dimensional data such as vectors and matrices, these are represented as sequences: $seq\ X$ for vectors and $seq\ seq\ X$ for matrices. Our extension also translates boolean and complex values, and is easily extended to include custom data types.

A challenge in achieving this translation was the fact that data types and dimensions are not recorded in the Simulink (*mdl*) file. We extract them using the new Type Extractor tool. This takes the *mdl* file and produces a *types* file containing data types and dimensions for all block inputs and outputs.

This is achieved by running a custom function within MATLAB; by executing inside the MATLAB environment, we can extract attributes of diagrams not stored in the *mdl* file. This tool iterates through all blocks in the diagram and produces a new file with the same structure as the original *mdl* file.

The extracted type information is combined with the original file by our new Merge tool. The purpose of Merge is to combine two *mdl* files into one *mdlx* file. A new file is created to maintain traceability and ensure the original diagram can still be used in Simulink. The Merge tool scans both input files for matching elements in the tree structure of systems, blocks, and subsystems. Attributes from matching pairs in both *mdl* and *types* files are merged in the new *mdlx* file.

3.3 ClawCircus

The majority of the translation is achieved using the ClawCircus tool, which takes the extended Simulink file and ClawZ output and produces a *Circus* specification. A description of the tool and its implementation can be found in [13].

What we needed to do to incorporate ClawCircus in the tool chain (apart from fixing a few bugs) was to provide a way of driving it without a graphical interface. Our new ClawCircus uses a configuration file to determine its input diagram and the required translation. This is useful in the safety-critical industry to ensure traceability; all graphical interfaces are removed in our chain.

The configuration file describes which part of a Simulink diagram to translate; the requested output could be a single block, a subsystem, or the entire system. It also defines whether a subsystem is expanded or collapsed. When expanded, the translation models all internal blocks as individual processes and combines them in parallel. When collapsed, the translation does not combine the blocks in parallel, but produces a centralised single process. It also defines whether the model is simplified or not. Simplified specifications do not contain vacuous definitions to ease readability, like empty schemas or actions without behaviour. Unsimplified versions have a more uniform structure; this is useful for automation of refinement where the shape of models is important.

Configuration files are simple and do not require additional expertise to produce. A parser to interpret the configuration file is now part of ClawCircus.

3.4 LaTeX, and the *Circus* Parser and Type Checker

The *Circus* file produced is encoded in the *Circus* LaTeX markup and can be transformed into a viewable document; the type-set output makes it easier to

read. The tool chain produces two outputs: one in *Circus* LaTeX markup for the parser and type checker (Complete *Circus* Model), and a *dvi* file with the correct graphical notation for the specification (*Circus* Document).

The *Circus* Parser and Type Checker is invoked automatically to check the validity of the *Circus* specification generated. This does not validate the diagram per se, but provides some empirical evidence for the validity of the models produced by the translation. Additional tool support to analyse and refine *Circus* specifications is under development; ease of model generation crucially paves the way for those techniques to be applied effectively. Further validation of the models themselves comes from the fact that they have been used as a basis for a refinement technique formalise in [4] to verify control systems.

In summary, the tool chain eliminates the need for vast amounts of manual input and specialised knowledge. By combining a Simulink file with the corresponding configuration file, all output files are produced automatically. The specification is automatically passed through the parser and type checker with a detailed account of the entire process stored in a log file. The tool chain is automated using a script to invoke tools and manipulate files.

4 Enabled and Action Subsystems

This section describes translation enhancements to model enabled and action subsystems. Outputs from these subsystems depend on an enabling condition, which is determined by the value received on an enabling or action port. Enabled subsystems check if a value is greater than zero before being enabled, whilst action subsystems use a signal from a separate if-then-else or switch statement.

We consider, for example, the very simple diagram in Figure 3; it demonstrates the use of action subsystems, but is not the limit of our approach. In the example, an If Else block is used to control two action subsystems, which each have their own output. The If Else block takes an input (In1) and compares the value against some condition; in this example, the value must be greater than 4. If true, the If Else block outputs a boolean true to the first subsystem, and false to the other. If false, the boolean outputs to the subsystems are false and true respectively. The values from both subsystems depend on the enabling conditions; these are determined by the boolean value from the If Else block.

Both action subsystems output a constant value with a delay of one time unit. They also contain a block labelled Action Port, which is the boolean input signal from the If Else block and determines whether the subsystem is enabled.

Output blocks in enabled and action subsystems output a value, whether the subsystem is enabled or disabled; this value depends on the behaviour of the internal blocks. Typically, outputs from several subsystems are combined using a Merge block to ensure that only the value from the currently enabled subsystem is used. Subsystem outputs, however, may also be used individually, in which case, the output when enabled and disabled needs to be considered. Additionally, by considering subsystems separately, rather than their combined use with other blocks, we obtain a compositional translation strategy.

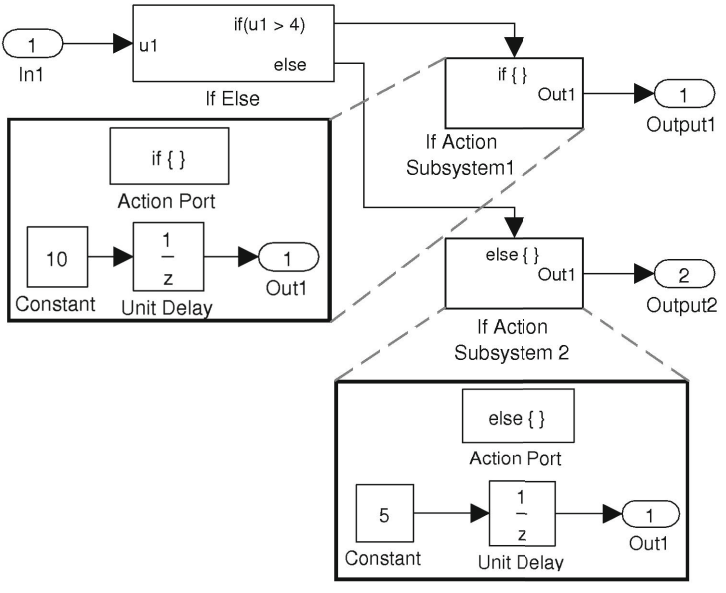


Fig. 3. Example If-Then-Else System

All blocks inside these subsystems are essentially paused when disabled. Current values must be recorded along with the enabling condition for all blocks with state inside the subsystem. It is not sufficient to use the existing behavioural definitions from ClawZ for blocks with state, as they do not include the additional components required to capture the enabling/disabling behaviour. Also, output blocks require an additional field to represent their initial value when inside an enabled or action subsystem; this is the initial output value of the subsystem.

When subsystems are enabled, blocks behave as they would normally. When disabled, blocks with state and output blocks have additional properties that describe what to do with the output value: whether to hold the last value stored, or reset the output to the initial value. Additionally, when the subsystem is re-enabled, having been in a disabled state, both enabled and action subsystems can be configured to preserve the states of all internal blocks, or reset them to their initial state. This affects blocks that define an output sequence for example, which can either pause at the last value, or reset to the first value in the sequence. The configuration is static as the held and reset properties are defined within the Simulink diagram. In summary, we have four configurations of subsystems and their blocks that give rise to different behaviours as shown in Table 1.

The remainder of this section describes how we represent the subsystems and their internal blocks in *Circus*, with the necessary Z definitions.

4.1 Z Definitions

The Z definitions used in the *Circus* model of a diagram have to be augmented to support enabled and action subsystems. This applies to blocks with state,

Table 1. Enabled/action subsystem state and output combinations

Subsystem config.	**Block config.**	**Block output when disabled**	**Subsystem state when re-enabled**
Held	Held	Retains previous value	Retains previous state
Held	Reset	Set to initial value	Retains previous state
Reset	Held	Retains previous value	Internal block states are reset
Reset	Reset	Set to initial value	Internal block states are reset

as it is the state that is updated in different ways; to capture this we include three schemas for each block with state. These schemas describe the standard behaviour of the block, the behaviour when held, and the behaviour when reset.

As an example, we consider the Unit Delay block (as seen in Figure 1), which takes an input value, stores it in the current state, and outputs the value from the previous state; it is a single one place buffer. ClawZ uses a generic definition as it is applicable to many data types; the standard behaviour is as follows.

$$
\begin{array}{|l}
\hline
UnitDelay[X] \\
In1? : X;\ Out1! : X \\
initial\_state, state, state' : X \\
\hline
Out1! = state \land state' = In1? \\
\hline
\end{array}
$$

Consider now the situation where the Unit Delay block is inside an action or enabled subsystem and is disabled; the output is either held or reset. The ClawZ schema to describe the behaviour when held is below. The difference between this and the standard schema is in the value stored in the $state'$ component.

$$
\begin{array}{|l}
\hline
UnitDelay\_h[X] \\
In1? : X;\ Out1! : X \\
initial\_state, state, state' : X \\
\hline
Out1! = state \land state' = state \\
\hline
\end{array}
$$

The schema for the behaviour when reset, shown below, is different to the standard one as the components $state'$ and $initial\_state$ are defined to be the same.

$$
\begin{array}{|l}
\hline
UnitDelay\_r[X] \\
In1? : X;\ Out1! : X \\
initial\_state, state, state' : X \\
\hline
Out1! = state \land state' = initial\_value \\
\hline
\end{array}
$$

These three schemas define the behaviours of the block when inside an enabled or action subsystem, however, they are not sufficient for the *Circus* model.

We require additional schemas that capture the value of the current and previous enabling condition. These additional components are crucial in order to define the four scenarios in Table 1. Firstly, we define a state schema which contains a single boolean value to record the enabling condition of the block.

Enabled_State

$$enabled : \mathbb{B}$$

This state component must be updated in accordance with the current enabling condition of the subsystem. Firstly we define a frame schema for the update operation that takes a boolean input and assigns it to the *enabled* state component.

Enabled_Frame

$$\Delta Enabled\_State;\ Enabled? : \mathbb{B}$$

$$enabled' = Enabled?$$

We define three further schemas to capture the scenarios where the subsystem becomes enabled, remains enabled, and is disabled.

$$Enabling == [Enabled\_Frame \mid enabled = \mathbf{False} \land enabled' = \mathbf{True}]$$

$$RemainEnabled == [Enabled\_Frame \mid enabled = enabled' = \mathbf{True}]$$

$$Disabled == [Enabled\_Frame \mid enabled' = \mathbf{False}]$$

Using these three schemas to capture the enabling condition of a block in conjunction with the existing ClawZ block library definitions, it is possible to define block schemas for each of the four kinds of subsystem configuration in Table 1. Firstly, in the scenario where both the block and subsystem are set to hold their values when disabled and on re-enabling, we use a definition like that shown below for our example Unit Delay block. Both the *Enabling* and *RemainEnabled* schemas are combined with the *UnitDelay* schema that defines the normal behaviour. This is because when both enabled and upon re-enabling, the block values remain the same and normal behaviour continues. When the block is disabled, the *UnitDelay_h* schema is used as this specifies the held behaviour.

$$UnitDelay\_Augmented == (Enabling \land UnitDelay) \lor (RemainEnabled \land UnitDelay) \lor (Disabled \land UnitDelay\_h)$$

The second scenario is when the subsystem is set to hold the internal states upon re-enabling, and the block is set to reset to its initial value when disabled. The difference here is the *Disabled* schema is combined with the reset schema.

$$UnitDelay\_Augmented == (Enabling \land UnitDelay) \lor (RemainEnabled \land UnitDelay) \lor (Disabled \land UnitDelay\_r)$$

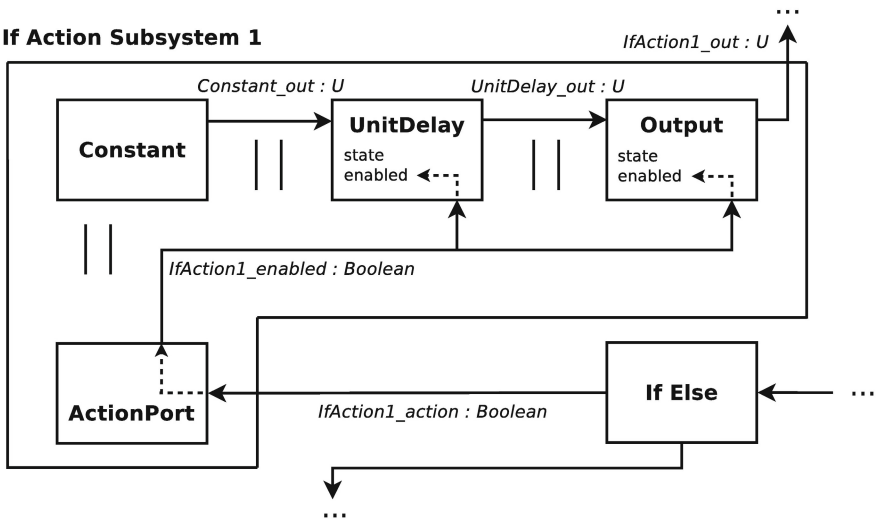


Fig. 4. *Circus* model for example action subsystem

The third behaviour is found when the subsystem resets the internal states upon re-enabling and the block holds its value when disabled. The *Enabling* schema is combined with the reset schema for the Unit Delay whilst the *Disabled* schema is associated with the held schema. Finally, the scenario where the block and subsystem reset their values. The *UnitDelay_r* schema is combined with both the *Enabling* and *Disabled* schemas.

The translation produces one of the four definitions above for each of the blocks with state inside an enabled or action subsystem, based on the properties of the subsystem and block.

4.2 *Circus* Model

This section describes *Circus* processes that model enabled and action subsystems using the Z definitions presented above. As described previously, the translation of subsystems can be done in two ways. Firstly, all blocks can be translated individually and combined using parallel composition. Alternatively, the subsystem can be defined in one centralised process. (Semantically the models are the same, however, parallelism facilitates refinement to concurrent implementations).

With a centralised process, the Z definition from ClawZ that represents the overall subsystem is lifted into a *Circus* process. This definition includes instances of the schemas for each block in the subsystem and connects the inputs and outputs together just like in the original approach. The *Action*? input to the subsystem is connected to all of the *Enabled*? inputs of the blocks. The *state* and *enabled* conditions for all blocks in the subsystem are defined as state components in the *Circus* process to ensure information is not lost between invocations.

Translations that use parallel composition of *Circus* processes for blocks in the subsystem are slightly different. To represent the *Action*? input to the subsystem,

1. For all blocks inside an action or enabled subsystem (apart from output blocks), check to see if the original ClawZ definition includes a state component. If no state exists, complete the block translation as in the previous technique.
2. For output blocks, and blocks with state, the additional frame schema described here must be combined with the original ClawZ schema according to the held and reset values of both the subsystem and the individual block.
3. Once all internal blocks are translated, the subsystem process is created:
   (a) If a centralised translation is required, instances of all internal blocks are included in the subsystem definition and are connected as per the wires in the diagram; the enabling condition is simply a component of the subsystem and accessed directly by the block schemas - there is no channel synchronisation. The state components of internal blocks are lifted to the state of the subsystem. The main action of the subsystem is a parallel composition defining the functional behaviour and the state update procedure.
   (b) A parallel translation creates individual processes for all blocks in the subsystem including the enabling/action port; the communication between processes is through channels as per the wires in the diagram. The overall subsystem process is constructed as the parallel execution of all processes that synchronise on the enabling condition and the *end_cycle* channel. The final step is to hide the internal workings of the subsystem process from the rest of the system; this is achieved by hiding all of the internal channels, leaving only the inputs and outputs of the subsystem visible.

Fig. 5. Algorithm to translate enabled and action subsystems

an additional *Circus* process is defined to pass on the enabling condition to the other blocks in the subsystem via a broadcast channel.

As a simple example, Figure 4 depicts the structure of the corresponding *Circus* model for the first subsystem in Figure 3. There, arrows represent synchronisation channels corresponding to wires in a diagram, whilst the two vertical bars inbetween processes indicate parallel composition. If Action Subsystem 1 and If Else are separate processes (which are composed in parallel to define the model of the complete diagram). The process If Action Subsystem 1 is itself defined by a parallel composition of four processes: Constant, Unit Delay, and Output correspond to the blocks in the diagram, and Action Port is the extra process defined below. The channels *Constant_out* and *UnitDelay_out* correspond to the wires. The *IfAction*1*_enabled* channel broadcasts the enabling condition received on *IfAction*1*_action* from the If Else block; all internal blocks in the subsystem synchronise on this enabling signal.

The *Circus* process *ActionPort* is below; it operates in parallel with the other processes. The *end_cycle* channel is used as a synchronisation point for all parallel processes; only once all processes have synchronised on the *end_cycle* channel can each individual process recurse or terminate accordingly.

$$\textbf{process}\ ActionPort \mathrel{\widehat=} \mu X \bullet$$
$$IfAction1\_action?x \longrightarrow IfAction1\_enabled!x \longrightarrow \textbf{Skip}\ ;\ end\_cycle \longrightarrow X$$

Processes that represent blocks inside enabled and action subsystems cannot use the standard translation with our additional state components. Most significantly, processes have to synchronise on the channel that passes the enabling condition of the enabled or action subsystem to the blocks. We extend the state of blocks with the *Enabled* flag and relate this to the underlying ClawZ schema. In our example, the *Enabled*? value is taken from the $IfAction1\_enabled$ channel.

The *Circus* model for enabled subsystems differs slightly to the action subsystem example as the enabling input is not a boolean value, but either a scalar or vector value. The *EnablingPort* process pushes the boolean value true onto the enabled channel if any input value is greater than zero, and false otherwise.

Our approach extends the existing ClawZ and *Circus* model in a uniform and structured way, and lends itself to automation. A text-based algorithm is shown in Figure 5 to demonstrate the steps required to implement the translation.

5 Conclusions and Further Work

In this paper, we address several problems in translating Simulink diagrams to *Circus* and discuss modifications and extensions to existing tools to provide an automated solution via a tool chain. A more comprehensive description of all the details discussed in Sections 2 and 3 can be found in [8].

The individual stages of the translation shown in Figure 2 have been adapted and combined to automate the process. The only part of the translation not successfully integrated in the process is ClawZ; this is due to the high level of customisation required from the user to successfully produce a ClawZ output.

The tool chain has been applied to large industrial examples, in particular, a previously non-translatable Non-linear Dynamic Inversion controller provided by QinetiQ. This non-trivial example includes nested subsystems, generic definitions, and a range of data types. The translation equates to 38,000 lines of *Circus* and completes automatically with no errors. The example presented minor bugs in tools that had not been tested with such large examples previously.

Several other examples have been used throughout the development and testing phase to ensure specific modifications and extensions are correct. These tests are small in comparison to the larger example above, however, each is challenging in its own right to test a particular part of the translation. The tool is available, with an example, from `https://svn.cs.york.ac.uk/anonsvn/clawcircus`.

As an alternative to our approach, Caspi *et al.* use the formal language Lustre to represent Simulink diagrams [3]. A tool automates their translation from Simulink to Lustre, and from Lustre to source code using the Lustre C code generator. This technique is focused on the generation of implementations with a certified code generator and has proven popular in industry. Consider, however, the situation in which the code generation technique changes; the revised generator must be re-certified. This is an expensive and time consuming process; should our technique to generate implementations change, the effort required to prove a modified refinement law is significantly less.

Chen *et al.* present a formal semantics and tool support to reason about functional and timing aspects of Simulink diagrams [6]. Their work presents a

comprehensive library of translatable blocks for both discrete and continuous time. The work is focused on the validation of diagrams with the use of the PVS theorem prover; it does not address our larger interest in program verification.

Our translation function, previously defined in [4], is specified in a compositional manner and allows us to produce an individual *Circus* process for each block or subsystem in a Simulink diagram. As *Circus* has a semantics that supports compositional refinement, piecewise development is well supported.

Future work will mechanise the translation of enabled and action subsystems based on the algorithm described. Automation of refinement techniques will allow automatic generation of models of Ada programs for verification of implementations. Work is also ongoing to integrate time-specific Simulink diagrams in *Circus* using *Circus Time*; this work will further increase the set of translatable Simulink diagrams and make the tool chain applicable to more applications.

References

1. Adams, M.M., Clayton, P.B.: ClawZ: Cost-Effective Formal Verification for Control Systems. In: Lau, K.-K., Banach, R. (eds.) ICFEM 2005. LNCS, vol. 3785, pp. 465–479. Springer, Heidelberg (2005)
2. Arthan, R., Caseley, P., O'Halloran, C., Smith, A.: ClawZ: Control laws in Z. In: ICFEM, p. 169. IEEE Computer Society (2000)
3. Caspi, P., Curic, A., Maignan, A., Sofronis, C., Tripakis, S.: Translating Discrete-Time Simulink to Lustre. In: Alur, R., Lee, I. (eds.) EMSOFT 2003. LNCS, vol. 2855, pp. 84–99. Springer, Heidelberg (2003)
4. Cavalcanti, A., Clayton, P., O'Halloran, C.: From Control Law Diagrams to Ada via *Circus*. Formal Aspects of Computing (2010)
5. Cavalcanti, A., Sampaio, A., Woodcock, J.: A refinement strategy for *Circus*. Formal Aspects of Computing 15(2), 146–181 (2003)
6. Chen, C., Dong, J.S., Sun, J.: A formal framework for modelling and validating Simulink diagrams. Formal Aspects of Computing 21(5), 451–483 (2009)
7. King, D.J., Arthan, R.D., Winnersh, I.C.L.: Development of practical verification tools. ICL Systems Journal 11, 106–122 (1996)
8. Marriott, C.: A Tool Chain for the Automatic Generation of Circus Specifications from Control Law Diagrams. Masters project thesis, Department of Computer Science, The University of York (2010)
9. The MathWorks, Inc. Simulink, `http://www.mathworks.com/products/simulink`
10. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice-Hall Series in Computer Science. Prentice-Hall (1998)
11. Woodcock, J., Cavalcanti, A.: The Semantics of *Circus*. In: Bert, D., Bowen, J. (eds.) B 2002 and ZB 2002. LNCS, vol. 2272, pp. 184–203. Springer, Heidelberg (2002)
12. Woodcock, J.C.P., Davies, J.: Using Z—Specification, Refinement, and Proof. Prentice-Hall (1996)
13. Zeyda, F., Cavalcanti, A.: Mechanised Translation of Control Law Diagrams into *Circus*. In: Leuschel, M., Wehrheim, H. (eds.) IFM 2009. LNCS, vol. 5423, pp. 151–166. Springer, Heidelberg (2009)

Verification of Hardware Interaction Properties of Software

Ramsay Taylor

Department of Computer Science, The University of Sheffield

Abstract. Many high-integrity software development processes prevent any assumptions about the system hardware, but this makes it impossible to use these techniques on software that must interact with the hardware, such as device drivers. This work takes the opposite approach: if the analyst accepts that the analysis will only be valid for a particular target system then the specification of the system can be used to infer the behaviour of the software that interacts with it. An analysis process is developed that operates on disassembled executable files and formal specifications of the target platform to produce CSP-OZ formal models of the software's behaviour. This analysis process is implemented in a prototype called Spurinna. This is demonstrated in conjunction with the verification tools Z2SAL and the SAL suite to demonstrate the verification of properties of an example program.

1 Introduction

Many projects make use of static analysis to give a measure of assurance for the safe functioning of the code. To facilitate static analysis these projects are often based on "safe" language subsets such as MISRA-C [13] and SPARK Ada [1]. These use restricted versions of common programming languages to make the code behaviour determinable without knowledge of the context. However, by restricting the language they necessarily make themselves unusable for applications that rely on the features that have been removed or restricted, and prevent analysis of requirements that are defined in terms of a particular context. Hardware control applications are a significant instance of this, and so they are the focus for the work presented here.

Hardware control and interaction is an area that is central to many safety-critical systems as it is often the device control aspect that gives them the potential to cause harm. Many restricted languages remove any feature that interacts with the hardware, since these features prevent deterministic reasoning about the code without making assumptions about the behaviour of the hardware. However, in the case of device drivers it is reasonable to make assumptions about the hardware — a specification of the hardware's behaviour is always necessary if software is to be written to control it.

Projects such as [12] and [3] have provided complete verification of system stacks, but these have required considerable manual effort that would have to be repeated for each application. The objective of this work is to provide a general

J. Derrick et al. (Eds.): ABZ 2012, LNCS 7316, pp. 308–322, 2012.

process for verifying the behaviour of low level software that is independent of the particular underlying system, and that is sufficiently automatic that it can be easily and frequently repeated as part of a software development process.

The principal contribution of this work is a technique for inferring a formal model of the behaviour of a hardware dependent system that has the following properties:

- The ability to represent the interaction of software and hardware components in the same model, and allow the verification of properties of hardware usage;
- a fully automatic implementation with no human input required after the submission of the hardware specification and the software for analysis;
- produces models of a size and complexity that can be understood by humans and is practical for the application of formal verification techniques;
- maintains traceability from the produced model back to the source program to support fault localisation and repair.

The following example of a hardware usage scenario is used in this paper:

> The device to be controlled has two ports: a control port and a data port. The control port is accessible at IO port address 0. The data port is accessed at IO port address 4. To request data the driver must write a 1 to the control port, then wait at least 10ms before the data on the data port is valid. To facilitate the timing there is a clock device available at IO port 8, which presents an integer representing time on a scale that increments once per ms. The system has an Intel i386 based processor.

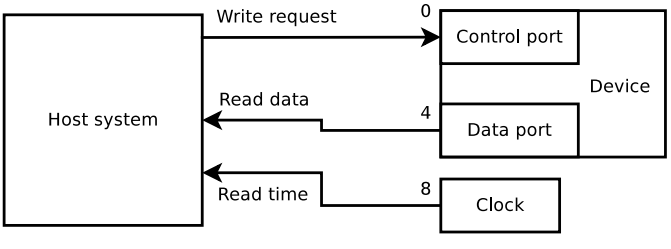


Fig. 1. The example device

A device driver that controls this device and presents the available data to a software system must ensure that the device is used in the required fashion. In this example that produces a set of specific behavioural properties that the device control software must satisfy:

- A request value of 1 must be written to the control port at address 0, before data is read from address 4.
- There must be a delay between the writing of this request and the reading of the data.

– The delay must be 10ms long. Specifically, the value present in the clock port at address 8 must have incremented by at least 10 between the writing of the request and the reading of the data

These properties make explicit statements about IO addresses, and about the sequence and content of interactions with these hardware features. The design of the formal model presented in Section 3 must support the specification of the combined software/hardware system such these details are present and properties of their use can be written easily.

1.1 Document Outline

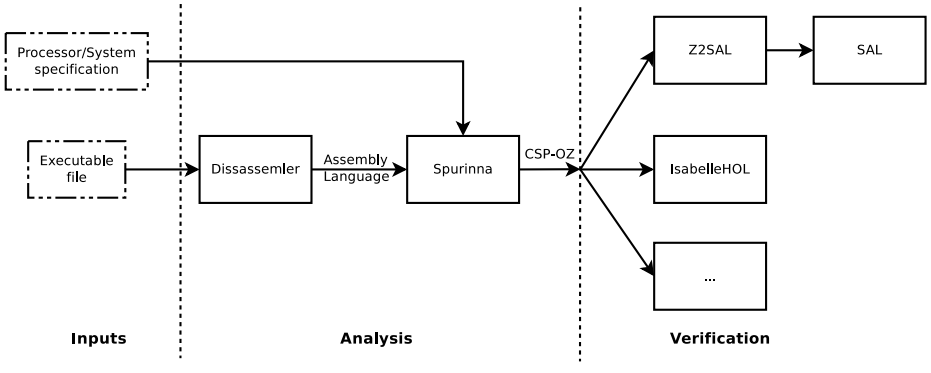


Fig. 2. The analysis process outline

The analysis process described in this paper has been implemented in a prototype tool named Spurinna[1]. The development of a formal system for interpreting executable file formats and performing disassembly was beyond the scope of this project, so GNU *objdump*[2] is used to convert the executable into assembly language and symbol information. Section 2 discusses the analysed programs and their disassembly. Spurinna takes this, and a supplied formal specification of the processor and target platform as inputs. Section 3 describes the model of the system that is required as input, the model of software behaviour that is produced by the analysis, and the design choices made to accommodate hardware details and low-level software features into a manageable, formal representation. From these inputs it is able to produce a formal model of the behaviour of the software entirely automatically. The process of automatically inferring models of this form from presented disassembly output is detailed in Section 4. This is output in CSP-OZ in LaTeX format, and can be used as input to any verification tools or techniques applicable to Z or CSP. This paper demonstrates verification of requirements using the Z2SAL tool [8] and the SAL model checking suite [7] in Section 5. Conclusions are presented in Section 6.

[1] `http://staffwww.dcs.shef.ac.uk/people/R.Taylor/Spurinna/`
[2] `http://www.gnu.org/software/binutils/`

2 Disassembly

The four stages between high-level source code and execution are *compilation* which produces assembly code, *assembly* which produces relocatable machine code object files, *linking* which collects object files into executable files and resolves the symbols in the function calls, and *loading* where the executable file is loaded into a virtual address space ready for execution.

This analysis process needs to operate on the level closest to execution, but capturing the image of the virtual address space after loading is impractical. Producing a formally-verified simulation of the loader in the target system would be ideal but the development of such a system is beyond the scope of this work. The object code produced by the assembler but before linking are also not suitable, since the linker makes a number of important decisions about the layout of the program in memory and about the resolution of symbols to absolute addresses and values. Consequently, it is the executable file that is as close as is practical and that are the source material for this analysis.

To illustrate the analysis process a program was developed to interact with the example device described in Section 1. The program was written in C and is shown in Figure 3.

In order to access the IO ports of the processor this program must use inline assembly code statements. This is a violation of the MISRA-C coding standards and is a good example of the impossibility of writing device driver code that stays within a safe language subset.

```
#define out(port, value) asm("out %1,%0" : : "dN" (port), "a" (value))
#define in(port, result) asm("in %1,%0" : "=a" (result) : "dN" (port))
#define CONTROL_REG 0
#define DATA_REG 4
#define CLOCK_REG 8
int exdev() {
  int starttime;
  int endtime;
  int now;
  int result;
  out(CONTROL_REG, 1);
  in(CLOCK_REG, starttime);
  endtime = starttime + 10;
  do {
    in(CLOCK_REG, now);
  } while(now < endtime);
  in(DATA_REG, result);
  return result;
}
```

Fig. 3. A C program that implements the device control behaviour

This program was compiled with gcc, the GNU C compiler. The resulting executable file was then disassembled with GNU *objdump* to produce the output show in Figure 4.

This shows the format of assembly instructions that are presented to the following stages of analysis, as well as the symbol information that was extracted from the executable. In this case the `exdev` function from the C program has remained identifiable, beginning at address 08048094. If the file contained multiple functions then these would be separated and identified by name. This example contains only local branch instructions but where function call instructions are present their target addresses are identified and the name of their target functions included. For example, a trivial program to identify the largest integer in a list using a helper function that compares two integers and returns the larger might contain a call instruction of the form: `call 8048180 <max>`.

```
08048094 <exdev>:
 8048094: 55                    push   %ebp
 8048095: 89 e5                 mov    %esp,%ebp
 8048097: 83 ec 10              sub    $0x10,%esp
 804809a: b8 25 00 00 00        mov    $0x25,%eax
 804809f: c7 00 01 00 00 00     movl   $0x1,(%eax)
 80480a5: b8 2b 00 00 00        mov    $0x2b,%eax
 80480aa: 8b 00                 mov    (%eax),%eax
 80480ac: 89 45 f4              mov    %eax,-0xc(%ebp)
 80480af: 8b 45 f4              mov    -0xc(%ebp),%eax
 80480b2: 83 c0 64              add    $0x64,%eax
 80480b5: 89 45 f8              mov    %eax,-0x8(%ebp)
 80480b8: b8 2b 00 00 00        mov    $0x2b,%eax
 80480bd: 8b 00                 mov    (%eax),%eax
 80480bf: 89 45 fc              mov    %eax,-0x4(%ebp)
 80480c2: eb 0a                 jmp    80480ce <exdev+0x3a>
 80480c4: b8 2b 00 00 00        mov    $0x2b,%eax
 80480c9: 8b 00                 mov    (%eax),%eax
 80480cb: 89 45 fc              mov    %eax,-0x4(%ebp)
 80480ce: 8b 45 fc              mov    -0x4(%ebp),%eax
 80480d1: 3b 45 f8              cmp    -0x8(%ebp),%eax
 80480d4: 7c ee                 jl     80480c4 <exdev+0x30>
 80480d6: b8 26 00 00 00        mov    $0x26,%eax
 80480db: 8b 00                 mov    (%eax),%eax
 80480dd: c9                    leave
 80480de: c3                    ret
```

Fig. 4. The C program after compilation, assembly, linking, and disassembly

3 Behaviour Model Structure

Many current approaches to low-level software verification, such as Separation Logic [14] are able to verify properties about programs by creating suitable, abstract models of pointers, memory addresses, and other hardware interaction that are applicable across all contemporary computer systems. This allows these approaches to explore subtleties of program construction, such as self-modifying

programs [5], that are not possible with the model presented here. However, the objective of this work is to create a model that deliberately does *not* abstract the implementation details of the hardware in any way, since it is aimed at verifying properties that make statements about specific hardware features.

The approach taken by this work is derived from the Z models of the state and operation of processors that have been produced since the 1980s [4,11]. Using process calculi also has a long history [2]. This analysis process uses both approaches and separate the control flow components of the program from the state change instructions. CSP-OZ [10] combines Object-Z [6] with CSP such that the Object-Z defines classes with state and operations on that state, while the CSP defines the possible control flow paths through those operations.

CSP-OZ specifications contain four types of component: A system state specification, operation schema that describe the state altering behaviour of events, CSP processes that define the allowed sequences of events in the system, and Object-Z classes that collect these components into an Object-Oriented framework.

Where adequate symbol information exists to identify functions in the code these are modeled as separate classes in the CSP-OZ model. This creates a model with a modularised structure that should aid comprehension.

The analysis process separates those instructions that alter control flow from those that do not. The former are referred to as *branch instructions*, while the latter are referred to as *sequential instructions*. Once the branch instructions have been separated, the remaining blocks of sequential instructions represent code that will all be executed if it is begun[3].

Branch instructions are further separated into *local* branch instructions, that alter control flow within a function, and *function call* and *function return* instructions that direct control flow to other identified functions. The distinction between the two is specified by the analysis user as part of the branch instruction set specification. Section 4.1 describes the process of separating the branch instructions from the sequential instructions to form a control flow graph. The nodes of this graph are the branch instructions, while the edges are the *sequential blocks* — the sequences of sequential instructions that contain neither a branch instruction, nor the target of a branch instruction, so are executed in sequence from start to finish. The behaviour of these sequential blocks is represented by the Z operations of the function's class in the CSP-OZ model. The local branch instructions are represented in the CSP part of the function's class definition, specifying the possible sequences of sequential blocks that can be executed. A conditional branch is encoded as an external choice between the two possible sequential blocks. To encode the decision procedure of the branch instructions two additional Z operation schema are added to the class that contain suitable

[3] Interrupts could violate this assumption, but their behaviour is ignored here as many device drivers will be operating as interrupt handlers, or with interrupts disabled. Alternatively, the impact of interrupts could be represented by making sections of the system state *volatile*, that is, its state becomes unspecified between atomic operations.

precondition invariants. These operations are prefixed to the two possible sequential blocks such that the preconditions of each choice model the decision behaviour of the instruction.

The model represents the function call and return behaviour using a more abstract, OO notation to make the inferred model more readable and more clearly resemble the structure of the original code, insofar as this can be determined from the information in the executable file. Function call and return behaviour is modelled by running the called function's class in parallel with the calling class. The calling class passes the system state along a channel to the called class, which performs its function on the system state, and then passes the state back to the calling class. The calling class synchronises on these transactions, so does not proceed until the called function has returned, and uses the Z theta notation to replace its current state with that received from the called function. Section 4.4 describes the process of combining the components together into a complete CSP-OZ representation of both the control flow and state change behaviours.

3.1 System State Specification

The formal model produced by this work must contain adequate detail of both the software and hardware behaviour to allow the properties of interest to be verified. Even a simple computer system has considerable detail that could be included, but only parts of this are relevant to the verification of a particular set of requirements. Consequently, the analysis process developed here is deliberately independent of the system specification used.

A simple specification of an Intel i386 based platform could be presented thus:

$$BIT == \{0, 1\}$$
$$INT32 == \{0..2^{32}\}$$
$$REGNAMES == \{eax, ebx, ecx, edx, esp, ebp\}$$

System

$$memory : INT32 \pfun INT32$$
$$registers : REGNAMES \to INT32$$
$$ioports : INT32 \pfun INT32$$
$$zf, cf, sf : BIT$$

For a particular Intel-based platform this schema could be augmented with invariants — perhaps identifying sections of ROM, or memory-mapped devices. The register interellations of an Intel processor, where al, ah, ax, and eax all refer to different components of the same 32bit value can be clearly represented by invariants, for example. Only a subset of the processor status flags are included here, and only a subset of the valid register names, but these are adequate for the short example used.

3.2 Sequential Instruction Templates

In addition to the system state specification, the user of this analysis process must also provide two instruction set specifications. The anaylsis process is independent of the system specification used, and is independent of the effects of the instruction set specifications but it does require a standardised format for the instruction set descriptions. The sequential instructions must be specified as *template* Z operation schemas. These are standard Z schema with a specific naming convention: the name of the schema must be the mnemonic of the instruction it represents, with a subscript containing the type signature for which this template defines behaviour.

Processor instructions are often defined with the same mnemonic having subtly different behaviour for different types of parameter. The types of parameter recognised by this analysis process are *literal*, *register*, and *register indirect* (where the value in a register is used as an address into memory, possibly with an offset). These three types are clearly identifiable in *objdump*'s output. The Intel `mov` instruction, applied to load a literal value into a register, can be specified with this template:

$$
\begin{array}{|l}
\hline
mov_{LIT\#SRC,REG\#TGT} \\
\Delta\ System \\
\hline
registers' = registers \oplus TGT \mapsto SRC \\
memory' = memory \\
\hline
\end{array}
$$

The subscript notation contains the parameters separated by commas, with the type and a placeholder name separated by the hash sign. The processing of the placeholders is described in Section 4.2.

3.3 Branch Instruction Templates

For branch instructions the `binst` collection is paramaterised with a mnemonic, and must contain a Z operation schema called *OnBranch* and, optionally, one named *NoBranch*. The *OnBranch* and *NoBranch* operation schema are prefixed to the sequential block at the target address, and the sequential block immediately following this instruction respectively. Unconditional branch instructions, such as the i386 `jmp` instruction, do not require a *NoBranch* schema, but the *OnBranch* schema may contain state change effects of the branch instruction, such as updating the program counter, if this is required for the verification.

Function call and return instructions are presented in the same way but in `callinst` and `returninst` collections, respectively. The process of converting these templates into representations of particular instruction instances is described in Section 4.4.

4 Analysis Process

The automatic analysis of a given executable to produce a model of the form described in Section 3 is broken into discrete stages that allow for as much

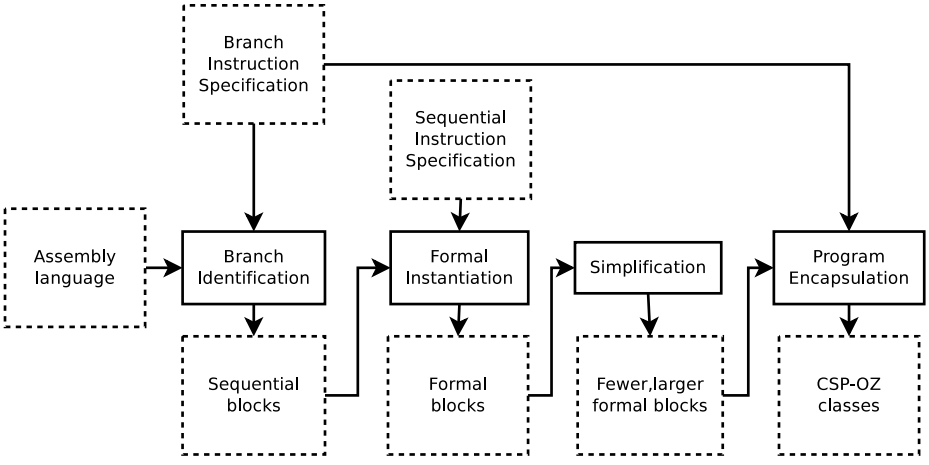


Fig. 5. The analysis workflow

parallel processing as possible. This allows problems of pure scale to be tackled most efficiently by the available resources and allows for the greatest impact of increased resources. The stages in the analysis work-flow and their inputs and outputs are shown in Figure 5.

4.1 Branch Identification

The *branch identification* stage of the analysis separates the branch instructions from the sequential instructions (as discussed in Section 3) using the supplied formal specification. Each mnemonic in the assembly language is compared to the provided branch instruction set. Where a mnemonic is identified as a branch instruction it is removed from the list of instructions, partitioning the block at that point. Additionally, if it is a local branch, the target of the branch is interpreted. If the target address falls inside an otherwise contiguous block of sequential instructions then that block is also partitioned at that address and a null, unconditional branch to the next block is inserted. This identifies that the second half can be reached by multiple routes.

This process produces a graph structure with the branch instructions forming nodes, and the blocks of sequential instructions forming edges. The unbroken lists of sequential instructions are referred to as *sequential blocks* and are named after the address of the first instruction they contain. The Z subscript convention is used, so the block starting at address $80480c4$ is named $Block_{80480c4}$. The branch instructions are also named after their locations. This naming convention retains tracability information throughout the analysis process. When a fault is identified in the completed model it is possible to locate the cause of the fault to a short block of instructions. Since these blocks represent state change with no decision making they are likely to have a clear correspondence to a small section of the original program.

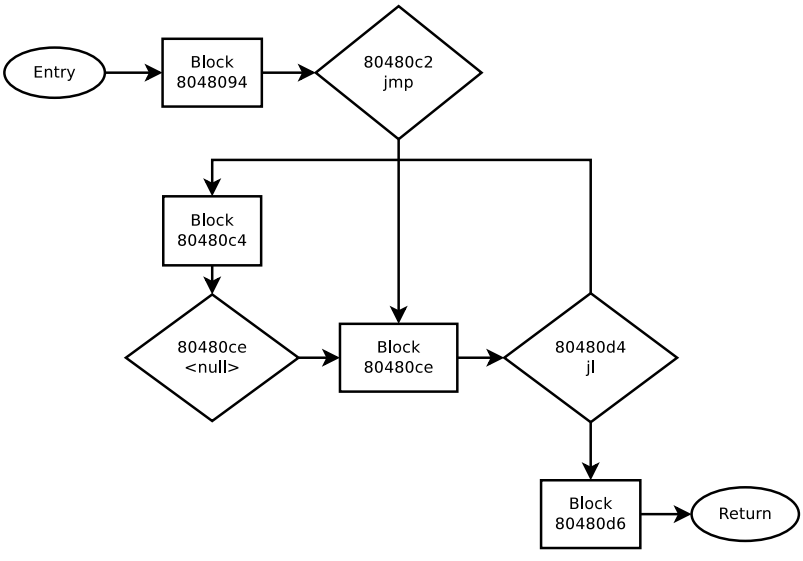


Fig. 6. Partitioning the exdev assembly code at the branch instructions

4.2 Formal Instantiation

The sequential blocks produced by the branch identification stage can be converted into formal representations of their behaviour. The instructions can be independently analysed and instantiated into Z operation schema representing their behaviour. All branching behaviour has been removed so these operation schema can be sequentially composed to produce a correct (but not minimal) representation of the system interactions of the block.

The instantiation process makes use of the template specifications described in Section 3.2. Each instruction is classified by mnemonic and by the type of the parameters present in the assembly language. The matching instruction template is identified from the mnemonic and the type signature present in the subscript of the template name. The template is then instantiated to represent a particular instruction but textually replacing the parameter placeholders with the values present in the assembly language at this point. The subscript of the name is replaced with the address of the instruction to maintain tracability.

For example the instruction: `0x80480d6: mov $0x26, %eax` has the mnemonic "`mov`" and a literal parameter, followed by a register parameter. This matches the following template:

$$mov_{LIT\#SRC,REG\#TGT}$$
$$\Delta System$$
$$registers' = registers \oplus \{TGT \mapsto SRC\}$$
$$memory' = memory$$

This is then instantiated to form:

$mov_{80480d6}$

$\Delta System$

$registers' = registers \oplus \{eax \mapsto 38\}$

$memory' = memory$

(Note: the Z convention represents integers in decimal, whilst the assembly language is in hexadecimal, so hexadecimal 26 becomes decimal 38).

4.3 Simplification

The result of the *formal instantiation* process is a series of sequential blocks that are modeled as long chains of sequentially composed Z operation schema representing each instruction. The size of these chains can quickly become unmanageable. The twelve line `exdev` function produced a 25 line assembly file with only 3 branch instructions. Some technique is needed to simplify these sequential blocks if the objective is readable formal models.

In principle, if program interruption is to be ignored, then the sequential blocks could be resolved to single Z operation schema but to do this requires some considerable formal analysis of the semantics of the operations which would be prohibitively difficult as the program size increased. This could be engineered if readability was the overriding objective. Some level of concatenation is possible for limited computation expense using the techniques outlined in [16]. The process operates by comparing two sequentially composed schemas and determining whether their composed semantics is altered by simply textually concatenating their invariants into one single operation. Since this process is text-based with only minimal parsing of the Z semantics it can be performed very quickly on large blocks of instructions.

There is a necessary choice between producing the most succinct model theoretically possible and producing a model entirely automatically. Since it is possible to apply automatic tools to the analysis of the model (for example Z2SAL, see Section 5) it can be argued that the simplification does not need to be complete if that would require excessive human effort.

4.4 Program Encapsulation

Having identified the branch instructions and sequential blocks, instantiated the formal specifications of the sequential components, and simplified the sequential blocks, the final element of the analysis process is to compose the sequential blocks into a CSP-OZ class that represents the function. The branch instructions must be instantiated to form the CSP components. The *OnBranch* and *NoBranch* schema from the conditional branch instructions are instantiated as Z operation schema where necessary. Function calls and returns must be instantiated with suitable models, as must the entry and exit of the functions.

Similarly, the analysis process records the branch that follows each sequential block as part of the internal model of the sequential block. With this information is is simple to convert unconditional branches and their target blocks into CSP statements. The branch instruction `jmp 80480ce <exdev+0x3a>` will always cause execution to transfer to virtual address `0x80480ce`. The control flow graph shows that the block beginning at `0x80480ce` ends with the branch instruction at address `0x80480d4`. The branch instructions are all represented by CSP processes named *Branch* with a subscript containing the virtual address of the instruction they represent.

$$Branch_{80480c2} = Block_{80480ce} \rightarrow Branch_{80480d4}$$

The *jl* instruction at address `0x80480d4` is a conditional branch instruction. As is discussed in Section 3, this is modeled by instantiating each possible target sequential block as a CSP arrow as before, then prefixing this arrow with a Z operation that serves to constrain the execution of the possible paths according to the conditions of the branch instruction. Finally, the two paths are conjoined with a CSP external choice operator.

exdev

$[...]$

$Branch_{80480d4} = (OnBranch_{80480d4} \rightarrow Block_{80480c4} \rightarrow Branch_{80480c9})$
$\quad \Box\ (NoBranch_{80480d4} \rightarrow Block_{80480d6} \rightarrow Branch_{80480db})$

$[...]$

$OnBranch_{80480d4}$

$\Xi System$

$sflag = 1$

$NoBranch_{80480d4}$

$\Xi System$

$sflag \neq 1$

$[...]$

As described in Section 3, function calls are modeled by executing the function in parallel, passing the system state using schema promotion, and then synchronising on the communication. All classes that use function call instructions include the *Call* and *Return* operations, which model this synchronisation. The sequence $Call \rightarrow Return$ is common to all function calls, from there the remainder of the process continues exactly as with unconditional branches: the next block is executed, and the process evolves to the next branch instruction. From the `maxint` example: `call 80480d8 <max>` becomes

$$Branch_{8048115} = (OnBranch_{8048115} \rightarrow Call \rightarrow Return \rightarrow Block_{804811a} \rightarrow Branch_{804811d}) \parallel max$$

All functions contain an *Entry* operation and a *Leave* operation that synchronise with the calling function and receive the system state, and then return then modified system state to the calling function at the end. CSP-OZ classes require a `main` process to begin execution. This begins with the *Entry* operation that receive the *System* state schema from a parallel call operation. Then the process continues with the first block and the first branch as any other branch.

All that remains is to collect these components into a CSP-OZ class, which is named according to the function name extracted by the disassembler. This produces a formal model where each function in the analysed system is contained in a CSP-OZ class.

5 An Example Verification

To demonstrate the usefulness of the inferred models the model produced for the example driver function was processed with the Z2SAL [8] tool. This produced an input file for the SAL suite of model checking tools [7]. The requirements specified in Section 1 were encoded as Linear Temporal Logic statements over this model and were verified using the SAL bounded model checker.

The Z2SAL tool does not accept CSP-OZ so the CSP-OZ had to be "flattened" to pure Z. The system state schema was augmented with a *cspstate* variable, defined with a BNF type that contains an atom for each of the processes in the CSP definition. The CSP control flow restrictions were converted to preconditions on this variable such that any given Z operation could only execute if the *cspstate* variable contained the name of a process that begins with this operation. The post condition of the operation then sets the *cspstate* variable to the name of the CSP process that follows this operation. This flattening is performed automatically by Spurinna.

Although a verification in SAL was developed this was limited to the bounded model checker, as even the small example state was too large for the symbolic model checker. To complement this, the CSP-OZ elements have been converted to a lightweight representation in Isabelle/HOL that allows properties to be verified symbolically over universally quantified state representations. Further details of this verification are presented in [17].

6 Conclusion

The principal difficulties that current techniques face when verifying hardware-dependent software are that: they have no way to determine statically the behaviour of code that interacts directly with the hardware; current techniques are necessarily detached from the hardware; and verification must fit into an industrial work-flow and not be overly dependent on expert skills, and must be applicable to large scale systems in reasonable time.

This work presents a technique that uses knowledge of the behaviour of specific hardware in order to allow the verification of its control software. This work

attempts to avoid the difficulties of analysing high level language code by taking the opposite approach: analysing code at the executable file level. While restricted languages attempt to make code sufficiently abstract that hardware details are irrelevant, the objective here is to make use of known hardware details to make high level language concerns irrelevant. The analyses operates on disassembled executables, and uses a formal specification of their target architecture as a guide to infer a model of the behaviour of the software. This should produce an interpretation of the software based on the environment in which it will run and should provide a better basis for understanding its interaction with the hardware.

The verification of properties on the inferred model has been demonstrated using Z2SAL and Isabelle/HOL. The conversion to Isabelle/HOL has not yet been automated, and still requires manual identification of the elements of the model that interact with variables of interest. A more complete and automatic embedding of the inferred models into Isabelle/HOL is intended as the continuation of this work.

The original decision to use CSP-OZ was influenced by the Syspect tool [15] that allows slicing techniques to be applied to CSP-OZ specifications. Slicing is intended specifically to highlight elements of a program that interact with particular state components, so this would address the identification problem in a larger system model. Syspect has since been expanded to model timing behaviour [9], which could also be valuable in verifying hardware control systems. It has not yet been possible to import the CSP-OZ specifications produced by Spurinna into Syspect, but this is a potential target for future work.

Acknowledgements. The author would like to thank his PhD supervisor, John Derrick, for his continous support, and also the PhD examiners, Georg Struth and Jonathan Bowen for their valuable and detailed suggestions.

References

1. Barnes, J.G.: High Integrity Software: The SPARK Approach to Safety and Security. Addison-Wesley Longman Publishing Co., Inc. (2003)
2. Birtwistle, G.: Control state in asynchronous micropipelines. In: Yakovlev, A., Nouta, R. (eds.) AINT, pp. 45–55 (2000)
3. Bogan, S.: Formal Specification of a Simple Operating System. PhD thesis, Saarland University, Computer Science Department (2008)
4. Bowen, J.P.: Formal specification and documentation of microprocessor instruction sets. Microprocessing and Microprogramming 21(15), 223–230 (1987)
5. Cai, H., Shao, Z., Vaynberg, A.: Certified self-modifying code. In: Ferrante, J., McKinley, K.S. (eds.) PLDI, pp. 66–77. ACM (2007)
6. Carrington, D.A., Duke, D., Duke, R., King, P., Rose, G.A., Smith, G.: Object-Z: An object-oriented extension to Z. In: Vuong, S. (ed.) Formal Description Techniques II, FORTE 1989, pp. 281–296. North-Holland (1990)
7. de Moura, L., Owre, S., Shankar, N.: The SAL language manual. Technical Report SRI-CSL-01-02 (Rev.2) (2003), `http://sal.csl.sri.com/doc/language-report.pdf` (accessed March 14, 2012)

8. Derrick, J., North, S., Simons, A.J.H.: Z2SAL: a translation-based model checker for Z. Formal Aspects of Computing 23, 43–71 (2011)
9. Faber, J., Linker, S., Olderog, E.-R., Quesel, J.-D.: Syspect - modelling, specifying, and verifying real-time systems with rich data. International Journal of Software and Informatics 5(1-2), 117–137 (2011) ISSN 1673-7288
10. Fischer, C.: CSP-OZ: a combination of Object-Z and CSP. In: Bowman, H., Derrick, J. (eds.) FMOODS, pp. 423–438. Chapman and Hall, London (1997)
11. Kemp, D.H.: Specification of VIPER1 in Z. Technical report, Royal Signals and Radar Establishment (1988)
12. Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: formal verification of an operating-system kernel. Commununications of the ACM 53(6), 107–115 (2010)
13. MISRA. Guidelines for the use of the C language in vehicle based software. Technical report (1998)
14. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS, pp. 55–74. IEEE Computer Society (2002)
15. Syspect. Final report of the syspect project. Technical report, Carl von Ossietzky University of Oldenburg (2006)
16. Taylor, R.: Separation of Z Operations. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) ABZ 2008. LNCS, vol. 5238, pp. 350–350. Springer, Heidelberg (2008)
17. Taylor, R.: Verification of hardware dependent software. PhD thesis, University of Sheffield (2012)

Using the Arbitrator Pattern for Dynamic Process-Instance Extension in a Work-Flow Management System

Matthes Elstermann[1], Detlef Seese[1], and Albert Fleischmann[2]

[1] Karlsruhe Institute of Technology (KIT), Institute AIFB, 76128 Karlsruhe, Germany
[2] Metasonic AG, Münchner Straße 29 - Hettenshausen, 85276 Pfaffenhofen/Germany
matthes.elstermann@kit.edu

Abstract. This paper presents the idea of using the Arbitrator Pattern, a concept from the field of robotics, and applying it to the domain of business process model interpretation, using it as a mechanism to allow dynamic adaption and extension of process instances at runtime. The idea is discussed and a formal specification is given using ASMs to detail the concept that was derived in the process of exploring the problem so far. Furthermore the non-trivial questions and issues are raised that will need to be addressed for this just started work-in-progress to advance.

Keywords: S-BPM, PASS, arbitrator pattern, dynamic model extension.

1 Basic Problem

The basic principle behind model-driven process-execution-systems (work-flow management systems) is that they use a model (most often a graph/diagram) and load it into an interpreter machine, thus forming an instance of the model. This instance, model and interpreter together, can be considered as a state machine which can be executed, or run through, until it has finished.

One such model language is the Parallel Activity Specification Schema (PASS) introduced by Albert Fleischmann in [1]. In [2] Egon Börger has presented an ASM specification for a PASS interpreter for single SBDs. The definition can also be found in [3] while [2] also gives further inside into S-BPM.

A great challenge for model based process execution system is that the models may need to change in order to cope with change requirements in the real-life processes that they are representing and supporting. With short lived process instances that is no problem. There are cases, though, where the execution of a process instance can take weeks or months. During such duration there usually is a big chance for circumstances to arise that in turn require changing at least parts of the process model ad hoc in order to satisfy the new needs without restarting whole process instances.

The problem is not new and described, e.g. in [4], and research into formal requirements for such mechanism dates at least back to [5]. An – admittedly brief – overview of the research has led to the impression that such mechanisms may

J. Derrick et al. (Eds.): ABZ 2012, LNCS 7316, pp. 323–326, 2012.

although not be explicitly be applicable to PASS models and their separated-graph nature.

The basic idea to allow for an update mechanism would be to incorporate a model exchange-mechanism or -option into the model interpreter machine. It will be assumed that a mechanism exists that guaranties validity of an extended model in the current context and that supporting tools can be realized in a way that a diagram D* will be a valid extension of D in a given currently running process context or ambient. A function *validInCurrentAmb(D*,D)* will be the placeholder. This will be one of the next research steps. The question up to discussion is whether the arbitrator pattern could be used for such a task and if yes, how this could be realized?

2 The Arbitrator Pattern

The arbitrator pattern stems from the field of robotics and was introduced by R.C. Arkin in [6] to program LEGO Mindstorm robots to interact with their not predefined environment. It allows for fast and effective programming of independent robots, but was advised against for use anywhere else, but robotic.

Its basic principle assumes a robot with input and output equipment. Instead of programming a single large complex program to control the machine there is one arbitrator deciding which of many smaller behavior programs (short "behaviors") is currently to be executed. These behaviors may contain only simple instructions like "move forward". Which behavior is currently controlling the robot is determined by a dynamically reevaluated priority that is defined based on the sensor inputs for each behavior. As soon as external events (e.g. the robot hitting a wall) require a change, the priority is shifted and the arbitrator executes a different behavior. Behaviors can be added as required given that priority-computing-functions are included.

3 The Arbitrator Pattern in S-BPM – The Resulting Specification

The idea now was to use that pattern as basis for a mechanism to allow dynamic adaption within an instance in work flow management system for PASS. In the context of S-BPM the way for a unit or subject to interact with its environment is via the reception and sending of messages which can be directed to other subjects (the environment). Furthermore internal inputs (user choices or other computations that determine actions of a subject) can, or rather should affect the priorities of behaviors (i.e.: can determine which behavior is executed).

The equivalent to the sensors of a robot here is a subject's 'message box'. And instead of controlling motors, here the arbitrator grants a subject-behavior the right to access the message facilities – to receive and send messages.

So the core concept here is not to handle a subject as single SBD-interpreter-machine, but as many interpreter machines encapsulated in an arbitrator-machine which can grant control rights for the unit/subject. Towards the outside a subject is in

principle a single unit with its *messageBox* and outgoing messages. The *BEHAVIOR(subj,state)* ASM defined by Börger in [3] probably needs to be adapted in order to really fit into the concept presented here. But for now it is assumed that the definition will work. Under normal circumstances there should be only one behavior container, containing a $BEHAVIOR_D(subj, state$. For the arbitrator pattern to work, it is assumed to be encapsulated in a container which has a priority and execution functionality:

```
BEHAVIOR_INTERPRETER_CONTAINER(D, subj, state, containerID )=
  seq
      if couldTakeControl(D, subj,state) then updatePriorityForArbitrator(subj, thisContainer)
      if  hasControl(subj, containerID) then BEHAVIOR_D(subj,state)
```

This definition should express that, when executed, the behavior container checks whether it *couldTakeControl* of the subject or not and updates the priorities. Based on that priority list the arbitrating machine will grant the access right via the *takeControl* command that should evaluate as *true* for the empowered *BEHAVIOR_INTERPRETER_CONTAINER* machine.

Upon a special change request – e.g. a special message (or event) outside the process context containing a new model (*behaviorExtensionArrived)* – the *ARBITRATOR(subj, context)* can initialize and/or start a new interpreter machine based on the received model D to take control of the subject. The condition of course, and the need for further research, is that the new model data *isValidForContext(newestBehaviorDiagram(subj), processContext)* in order to fit logically into the current process context which is given by the model the original behavior-machine is based on. The initialized machines need to be traced/collected in a location of *activeBehaviours(subj).*The source of such a special message for now is assumed to be an administrator outside the process context. More elaborate or sophisticated mechanisms are imaginable. By default a newer model simply has a higher priority for execution. Further rules to determine priority will be needed.

An attempt to specify the described mechanism with the means of ASMs is given here:

```
ARBITRATOR(subj, processContext)
      if behaviorExtensionArrived(subj, processContext, D*) then
            if isValidForContext(newestBehaviorDiagram(subj), processContext) then
      initializeNewBehaviorInterpreterContainer(newestBehaviorDiagram(subj))
      else forall  i in activeBehaviours(subj) do
            hasControl(subj, i) := false
              BEHAVIOR_INTERPRETER_CONTAINER(D, subj, state)
        choose j in activeBehaviours(subj) where priority(j) > priority(x){x != j}
            hasControl(subj, j) := true
```

This machine should execute with a certain frequency, repeating the cycle and being aware of new behaviors, reevaluating priorities and (re-)granting control to a behavior continuously. In the case of the original robotics concept, behavior changes can occur in the span of milliseconds. In a business process context, a behavior change may not need the strict real-time requirement since a change might occur only a few times.

4 Final Thoughts

The mechanism is not complicated. In robotics this simple approach allows to construct complex behavior out of simple elements. Future research will be aimed at investigating whether the application of this concept here can be useful and to find possible drawbacks (e.g. validation concerns) and challenges that would need to be addressed before actual applying this concept. Among those being the definition of valid model extensions mechanism/validator rules for PASS, followed by issues like the priority determination and the question about frequency of priority updates and actual behavior changes among other details needed to actually build a prototypical implementation as the high goal.

References

1. Fleischmann, A.: Distributed systems: software design and implementation. Springer (1994)
2. Fleischmann, A.: Subjektorientiertes Prozessmanagement: Mitarbeiter einbinden, Motivation und Prozessakzeptanz steigern. Hanser, München (2011)
3. Börger, E.: Homepage of Egon Börger (2011), `http://www.di.unipi.it/~boerger/Papers/Bpmn/SbpmBookAppendix.pdf` (Zugriff am March 19, 2012)
4. Dadam, P., Reichertt, M.: The ADEPT Project: A Decade of Research and Development for Robust and Flexible Process Support. Computer Science 23(2), 81–97 (2009)
5. Dadam, P., Reichertt, M.: ADEPTflex-Supporting Dynamic Changes of Workflows Without Losing Control. Journal of Intelligent Information Systems 10(2), 93–129 (1998)
6. Arkin, R.C.: Behavior-based robotics, 3. Aufl. MIT Press, Cambridge (2000)
7. Börger, E., Cisternino, A., Gervasi, V.: Ambient Abstract State Machines with applications. Journal of Computer and System Sciences (2011)

A Unified Processor Model for Compiler Verification and Simulation Using ASM

Roland Lezuo and Andreas Krall

Institute of Computer Languages,
Vienna University of Technology,
Argentinierstr.8,
A-1040 Wien Austria
{rlezuo,andi}@complang.tuwien.ac.at

Abstract. For safety critical embedded systems the correctness of the processor, toolchain and compiler is an important issue. Translation validation is one approach for compiler verification. A common semantic framework to represent source and target language is needed and Abstract State Machines (ASMs) are a well suited and established method. In this paper we present a method to show correctness of instruction selection by performing fully automated simulation proofs over symbolic execution traces of state transformations using an automated first-order theorem prover. We applied this approach to an industrial-strength compiler and created the ASM models in such a way that we are able to reuse them to create a cycle-accurate simulator. To achieve fast simulation we compile the ASM models to C++ and present the compilation scheme in this paper. Finally we present our preliminary results which indicate that a unified ASM model is sufficient for proving correct instruction selection and generating efficient cycle-accurate simulators.

1 Introduction

Todays safety critical systems often require application specific processors to fulfill the demanding performance and efficiency requirements. Correct behavior of the processor and the corresponding toolchain is an absolute requirement making formal specification and verification necessary. We are interested in using the same formal methods for compiler verification and simulation. Abstract State Machines are a well established method for specification and analysis of programming languages and systems providing a simple practical framework offering important features for industrial usage like decomposability and are readily understood [1].

Section 2 describes the generation of (first-order logic) proof scripts using symbolic execution of ASM models to perform translation validation [6] of instruction selection [3]. Section 3 describes our approach to generate an high-performance simulator using compilation to C++. Section 4 presents our preliminary results and concludes the paper.

J. Derrick et al. (Eds.): ABZ 2012, LNCS 7316, pp. 327–330, 2012.

2 Correctness of Instruction Selection

Zimmermann and Gaul [9] describe constructing correct compiler backends for DEC Alpha using ASMs. DEC Alpha has some nice properties making it very suitable for formal description [4]. The processor used in our project is a very long instruction word (VLIW) architecture with digital signal processor features and a non-interlocking pipeline. It supports wrap-around and saturation arithmetics, single instruction multiple data instructions, predicated execution, hardware loops and store/load with updates to the address register.

During instruction selection a (sub)tree of intermediate representation (IR) nodes is matched with a sequence of machine instructions, IR variables and temporaries (*operand*) are mapped to registers (*regmap*). We assume an infinite number of registers at this stage and allocate real registers later.

Such a translation is correct if the transformation described by the IR tree ($result_{tree}$) and the transformation induced by execution of the machine instructions ($result_{instr}$) is semantically equivalent. Semiformal this can be stated in first-order logic as: $\forall$ *operand : regmap(operand)* $\equiv$ *operand* $\Rightarrow$ *regmap(*$result_{tree}$*)* $\equiv$ $result_{instr}$. Some trees and instructions may however have side-effects (e.g. a modified memory cell) which are modeled as updates to ASM functions. For correctness the IR tree and the machine instructions must induce the same side-effects, semiformal this can be stated as $\forall$ $updates_{tree} \Rightarrow \exists$ $update_{instr}$ *:* $update_{tree}$ $\equiv$ $update_{instr}$ and vice versa.

To determine whether $result_{tree}$ and $result_{instr}$ are equivalent, ASM models (see next section for more details) using a common semantic vocabulary defining IR tree operations and machine instructions have been developed. The common semantic vocabulary is modeled as external functions in the ASM models. To generate a proof script the ASM execution engine logs an invocation of the external function f with arguments a returning result r as predicate $f(a, r)$. As concrete values for the operands are not known at instruction selection time the ASM has to be evaluated symbolically. The ASM execution engine performs the following steps when evaluation of a function f at location l results in *undef*. First create a new symbolic value s for f at l, then directly modify the definition of $f(l)$ so each evaluation of $f(l)$ returns s. The value *undef* is preserved when set explicitly, so evaluating $f(l)$ after a $f(l)$:= *undef* will result in *undef* and not in a new symbol s. Finally log the creation of the new symbol as predicate $f(l, s)$.

The resulting log is a sequence of predicates stating facts about (symbolic) values of dynamic functions (e.g. contents of registers) and invoked external functions (i.e. the common semantic vocabulary). Given a set of axioms describing relations of the semantic vocabulary and the a priori known mapping of IR operands to registers a theorem prover can now show semantic equivalence of the state transformation described by the IR tree and the machine instruction induced state transformation.

3 Fast Cycle-Accurate Simulation

Teich et al [8] have shown that ASM models can be used to generate a simulator for a processor. What they call bit-true arithmetic functions is equivalent to our common semantic vocabulary. Our simulator core is itself described in ASM notation (approx. 200 LOC). In contrast to [8] we are interested in efficient industrial-strength simulators. We initially tried the CoreASM execution engine [2] but simulating 50 CPU cycles took around 13 seconds. We considered compiling the CoreASM language to C++, but efficient compilation is difficult due to the dynamic type system (i.e. its List background). By adding type annotations to lists and restricting ourselves to a statically typed subset of the CoreASM language we were able to develop an efficient compiler.

Our compilation scheme preserves the static structure of the ASM model, and we follow the formal definition of ASM very closely. Each evaluated rule produces an update set, which is aggregated and composed as described in [2]. We support a (static) subset of the following CoreASM language elements: *seqblock, par, let, ifthenelse, :=, debuginfo, push, pop, forall, call, case, enum, derived, static, cons, nth, peek, tail, program, self* and lists with the restriction of all elements being of the same type (may be another list type). As we support the P *seq* Q rule we may need a (local) copy of the state to apply P's updates before evaluating Q. Such a copy however would be very expensive as the state contains huge functions (e.g. system main memory). That is why we introduced a so called *PseudoState* which only contains updates which should have been applied to the state already. When querying the *PseudoState* for a function f at location l a hashmap containing the pending updates is searched for $f(l)$ and if such an update is found its value is returned, if no such update can be found the global state is queried for $f(l)$.

It turned out that handling of the update sets is crucial to the performance of the simulator. As updates can not be stack allocated, and dynamic memory allocation using malloc/new would be too expensive a memory pool allocator is used. Memory management overhead is minimal as just the pointer to the next free memory cell needs to be incremented after each allocation. As soon as evaluation of the top level rules terminates (called a step in [2]) the resulting updates are applied to the global state and the memory pool is reset. This enables efficient simulation but large update sets are still troublesome for the simulator performance.

4 Preliminary Results and Conclusion

The proof generation system is capable of compiling the ANSI C Rijndael reference implementation v2.2 resulting in approx. 1650 proof scripts. About 700 scripts are successfully proven as correct. Most of the other scripts can't be proven due to missing semantic description of the involved IR nodes and machine instructions. We currently are able to handle basic copying and converting instructions (e.g. register moves), basic arithmetic operations (e.g. addition),

memory access (load and store), memory access with pointer increment, but also conditional branches with symbolically evaluated conditions.

We are able to correctly simulate all fundamental features of the CPU like instruction fetch, bundling decoding, predicated execution, hardware loops and the pipeline. Due to missing ASM models of the instruction set only one test program is executed correctly. For this case our simulator is slightly better than the manually coded simulator provided by the hardware vendor. The compiled models execute approx. 3000 times faster compared to interpretation by the CoreASM execution engine.

We have presented an approach to translation validation using ASMs and theorem proving targeting a processor architecture with many difficult to model features. We then used the very same semantic models to generate a fast cycle-accurate simulator with performance comparable to a manually coded vendor provided simulator. To achieve efficient simulation we restrained ourselves to a static subset of the CoreASM language but nonetheless found creation of the models easy.

Ongoing work is creating the missing ASM models to show the verification method is suited to prove industrial strength programs and increase the number of applications which can be simulated.

Acknowledgment. This work is supported in part by the Austrian Research Promotion Agency (FFG) and by Catena DSP GmbH. We would also like to thank Laura Kovács for her valuable help with the vampire [7,5] theorem prover.

References

1. Börger, E.: Abstract state machines: A method for high-level system design and analysis (2003)
2. Farahbod, R., Gervasi, V., Glässer, U.: CoreASM: An extensible ASM execution engine. In: Proc. of the 12th International Workshop on Abstract State Machines, pp. 153–165 (2005)
3. Fraser, C.W., Henry, R.R., Proebsting, T.A.: BURG: fast optimal instruction selection and tree parsing. ACM Sigplan Notices 27(4), 68–76 (1992)
4. Gaul, T.S.: An abstract state macine specification of the DEC-alpha processor family (1995), `ftp://www.jair.org/groups/Ealgebras/alpha.pdf`
5. Hoder, K., Kovács, L., Voronkov, A.: Interpolation and Symbol Elimination in Vampire. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS, vol. 6173, pp. 188–195. Springer, Heidelberg (2010)
6. Pnueli, A., Siegel, M., Singerman, F.: Translation validation, pp. 151–166. Springer (1998)
7. Riazanov, A., Voronkov, A.: The design and implementation of VAMPIRE. AI Commun. 15, 91–110 (2002)
8. Teich, J., Kutter, P.W., Weper, R.: Description and Simulation of Microprocessor Instruction Sets Using ASMs. In: Gurevich, Y., Kutter, P.W., Vetta, A., Thiele, L. (eds.) ASM 2000. LNCS, vol. 1912, pp. 266–286. Springer, Heidelberg (2000)
9. Zimmermann, W., Gaul, T.: On the construction of correct compiler back-ends: An ASM approach. Journal of Universal Computer Science 3, 504–567 (1997)

Modeling Synchronization/Communication Patterns in Vision-Based Robot Control Applications Using ASMs

Andrea Luzzana[1], Mattia Rossetti[1],
Paolo Righettini[2], and Patrizia Scandurra[1]

[1] Università degli Studi di Bergamo, DIIMM, Dalmine (BG), Italy
[2] Università degli Studi di Bergamo, DPT, Dalmine (BG), Italy
{andrea.luzzana,mattia.rossetti,paolo.righettini,
patrizia.scandurra}@unibg.it

Abstract. This paper proposes the use of the *Abstract State Machine* method for a rigorous foundation in modeling and validating *Vision-Based Robot Control applications*. We show how to tailor control tasks definitions and associated synchronization/communication patterns in rigorous and abstract terms by using *control state ASMs* and an extension of the classical flowchart notation to allow the definition/instantiation of recurring design solutions and to improve model traceability.

1 Introduction

Vision guided robotics is a challenging research field [4]. Open problems are the need for exchange of experiences, best practices, and high-level models of robust and flexible robot control applications with *visual servoing* (VS) functions. Recently, we considered the *Abstract State Machine* (ASM) method [3] for a systematic study and a rigorous foundation of modeling and validating Vision-Based Robot Control applications. In this paper, we show how to exploit the notion of *control state ASMs* [3], as a natural extension of Finite State Machines, to model the behavioral view of task-level control of VS applications. To this purpose, an extension of the classical flowchart notation is also proposed to denote explicitly modeling elements to be further refined, to allow the definition/instantiation of recurring design solutions (or patterns), and to improve traceability between the flowcharts and their concrete (textual) ASM specifications.

As starting point, we extracted from the code high-level and recurring synchronization/communication patterns of control tasks that could be used for the specification and analysis in ASM. We then refined these ASM abstract models to executable ASMETA/AsmetaL [2] models to validate them and run scenarios.

This paper is organized as follows. Sect. 2 summarizes the synchronization/-communication patterns at control task-level investigated in our work. Sect. 3 presents the new notation for control state ASMs, and the ASM specification of the swinging buffer communication mechanism, including its application to a concrete VS application. Finally, Sect. 4 sketches some future directions.

J. Derrick et al. (Eds.): ABZ 2012, LNCS 7316, pp. 331–335, 2012.

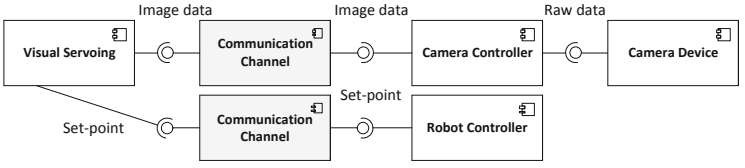


Fig. 1. A Visual Servoing Robot architecture

2 Application Domain and Background Concepts

The UML component diagram in Fig. 1 shows a possible architecture of a VS application. The vision system provides input to the robot controller acquiring and elaborating images; then, the robot controller uses joint feedback to internally stabilize the robot. Optionally, set-point computation can require acquisition of robot data, creating another communication channel.

Synchronization and Communication Issues. Basically, control tasks can be classified in *asynchronous* and *synchronous*. Asynchronous tasks are data-driven, because their elaboration starts when there are data to be consumed and ends with data transfer. Synchronous tasks are, instead, time-driven, as they are periodic and have deadlines to respect. Fig. 2 summarizes the possible communication types that we cover in our work, as collected from visual servoing and robot control applications. The analyzed solutions involve the use of swinging buffers for the communication between tasks operating at different frequencies. A swinging buffer can be viewed as an advanced circular buffer using two or more shared memory arrays instead of the single array adopted by a circular buffer. While the producer task fills up one of the buffers, the consumer empties another one. When a task reaches the end of the buffer that it is using, it starts operating from the beginning of another unused array. Since tasks works on different memory locations, no lock for the mutual exclusion is needed to access to the data on the buffer, but only for updating the read/write pointers (indexes).

Producer \ Consumer	Asynchronous	Synchronous
Asynchronous	**Asynchronous Message Passing** typical Producer-Consumer	**Swinging Buffer Communication** Visual Servoing TO Robot Controller
Synchronous	**Swinging Buffer Communication** Robot Controller TO Visual Servoing	**Swinging Buffer Communication** Sensor TO Robot Controller TO Motor

Fig. 2. Task communication types

3 ASM Models of Tasks Synchronization/Communication

We use ASMs as precise mathematical form of *ground models* [3] for specifying synchronization/communication patterns of robot control tasks. We, here, focus on the swinging buffer mechanism.

As shown in Fig. 3, we extended the classical flowchart notation for control state ASMs [3]. First, we use dashed lines for guards (or conditions) and actions (or rules) to indicate that these elements require further refinement. The textual notation `{text}` near a symbol is optional; it is useful to link the diagram to its concrete ASM specification `spec`. Specifically: for a state symbol it denotes the function name in the `spec` representing the underlying control state variable; for a guard symbol it denotes the test predicate name in the `spec`; for an action symbol it denotes the rule name in the `spec` implementing it.

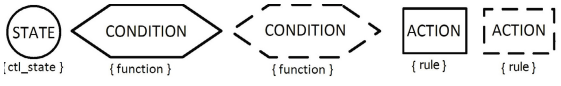


Fig. 3. Control state notation

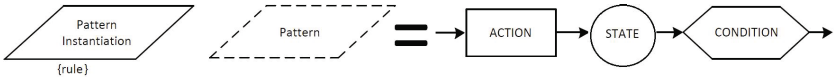


Fig. 4. Pattern notation

We also introduce two new symbols (see Fig. 4) to denote the concept of *pattern* and of *pattern instantiation* blocks, respectively. These pattern symbols are to be intended here as a placeholders for a piece of reusable ASM model, a *pattern machine* (or *pattern block*), that can be validated and verified separately and then re-used in other ASM specifications. Fig. 4 also shows the shape of such a pattern machine that includes an entering arrow followed by (at least) an action-state-condition block closed with a floating exit arrow. The circles represent the internal states of the pattern machine and it usually requires a (fresh) control state variable `ctl_state`. The entering arrow denotes always the evaluation of the guard `isUndef(_ctl_state)` that is the mandatory condition that enables the execution of the pattern machine. The floating exit arrow denotes the exit point and implies always the mandatory update `ctl_state := undef`.

Note that the most general form of composition of a complex ASM out of a simpler one is by rule replacement, so a pattern machine is defined in terms of a named rule[1] and this rule will occur as subrule of the containing machine.

Swinging Buffer Reading/Writing Patterns. As an example, Fig. 5 shows the pattern machine for an *Asynchronous-Master-Writing* operation. It implies

[1] The rule name can be specified (see Fig. 4) near the pattern instantiation symbol.

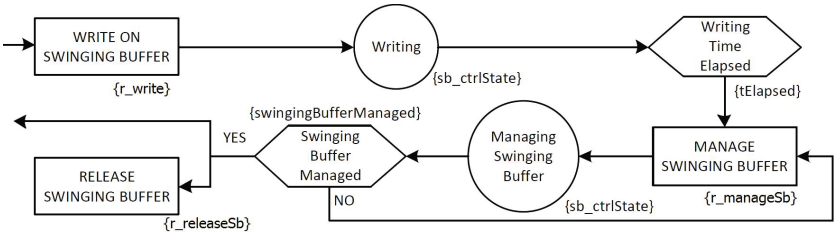


Fig. 5. Swinging Buffer - Asynchronous Master writing

first to write data on the shared memory directly (state *Writing*), without acquiring the lock for the critical section. When the writing operation is terminated (a certain period of time is passed), the swinging buffer indexes have to be updated in order to signal to the consumer that new data are ready. In the next state *Managing Swinging Buffer*, the asynchronous task try (by the iterative flowchart part) to get the lock to the critical section for updating the indexes by the action *MANAGE SWINGING BUFFER*. After the indexes are updated, the control exits by releasing the lock (*RELEASE SWINGING BUFFER*).

Application Examples. Fig. 6 shows the ASM control state for the visual servoing component in Fig. 1. The component's task is asynchronous and it is both consumer of images coming from the (synchronous) camera controller component and a producer of commands for the (synchronous) robot controller. It communicates with the two tasks through two swinging buffers and it is the master. The complete AsmetaL specification is available in [1].

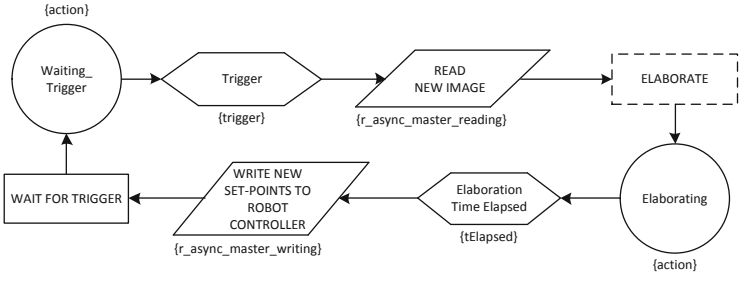


Fig. 6. Visual Servoing component

4 Future Directions

We want to collect in a library validated ASM models for recurring design solutions of VS applications and use them as patterns for modeling and validating typical control tasks of VS applications in a formal way, thus leading from the abstract models to executable (C/C++) code by a series of refinement steps.

References

1. asmeta.svn.sf.net/svnroot/asmeta/asm_examples/RTPatternLibrary
2. The ASMETA toolset website (2006), http://asmeta.sf.net/
3. Börger, E., Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer (2003)
4. Hutchinson, S., Hager, G.D., Corke, P.I.: A tutorial on visual servo control. IEEE Transactions on Robotics and Automation 12(5), 651–670 (1996)

A Reliability Prediction Method for Abstract State Machines

Raffaela Mirandola[1], Pasqualina Potena[2], and Patrizia Scandurra[2]

[1] Politecnico di Milano, DEI, Milano, Italy
mirandola@elet.polimi.it
[2] Università degli Studi di Bergamo, DIIMM, Dalmine (BG), Italy
{pasqualina.potena,patrizia.scandurra}@unibg.it

Abstract. According to the vision of *Design for Reliability*, software reliability has to be considered in all the activities within the software development life cycle. In particular, writing formal specifications, like other activities in software development, is error-prone, especially for large-scale systems. This paper presents a *reliability prediction method* for *Abstract State Machines* specifications. The method considers the internal structure of an ASM by computing its reliability based on the reliabilities calculated inductively along the call tree of the ASM rules and the structure of the rule bodies.

Keywords: ASM-based specifications reliability, ASM and system quality.

1 Introduction

Software risk comes mainly from its poor reliability, but how to effectively achieve high reliability is still a challenge today [5]. The software reliability has to be considered in all the activities within the software development life cycle.

Formal methods are more effective in achieving the completeness and accuracy of the user's requirements than informal specification methods. Formal specification and analysis, model review, prototyping (simulation) and testing techniques work together at different levels of software development to improve software reliability [6]. However, since writing formal specifications is error-prone, especially for large-scale systems, techniques that improve software reliability also during formal specification itself are required. Some methods for quality assurance of formal specifications exist, mostly based on model review, testing and model-checking (see, e.g., the work in [2]).

In this paper, we focus on reliability aspects and present a *reliability prediction method* for *Abstract State Machines* (ASMs) [4] specifications. A few previous works concerning quality assurance of ASM specifications exist (e.g., [2], and [8]). The proposed method focuses on reliability and adopts a path-based approach that considers the internal structure of an ASM and computes its reliability inductively along the call tree of the ASM rules and the structure of the rule bodies. This "predictive" analysis enables the solution of reliability problems at

J. Derrick et al. (Eds.): ABZ 2012, LNCS 7316, pp. 336–340, 2012.

the specification level, when modifications are easier and cheaper to be implemented. Our work contributes to the general goal of providing high confidence reliability evidence for critical system's parts formally specified through the use of ASMs, thus achieving a quality specification-development process based on the ASM formal method.

2 Modeling the Reliability of ASM Specifications

Inspired by the approach in [9], we here present a *path-based* reliability model of an ASM. We consider only (single-agent) basic ASMs with no shared functions. We include rule constructors for non-determinism (`choose`) and for unrestricted synchronous parallelism (`forall`), and the sequential composition (`seq`) of Turbo ASMs. We focus on *crash failures*, provoking the crash of the whole system, and consider as ASM failures the *violation of invariants* or the yielding of an *inconsistent update* set[1]. As in many existing approaches [5], we assume that these failures are independent from each other.

ASM Rule Dependency Tree (RDT). Starting from the *main rule* of an ASM M – except for the rules that are never used or are not reachable from the main rule – it is convenient to represent its internal structure by means of a tree. Indeed, the program of M is built up from *basic rules* (i.e. the `skip` rule, the function `update` rule, and (`macro`) rule call) and by rule constructors (i.e., the operators `if`, `par`, `seq`, `choose`, `forall`, and `let`). Thus, we define the $RDT = (V, E)$ of the ASM M, where the nodes V are labeled by the rules in the main rule of M, and the edges E reflect the nesting relationship among these rules. Basic rules are associated with leaf nodes, while rule constructors are associated with internal nodes. We will say that a node $j \in V$ is a *direct descendant* of $j' \in V$, if there is no node `par` rule in the path from j to j'.

Fig. 2 shows an example of ASM specification (using the AsmetaL notation of the toolset ASMETA [3]) and its corresponding RDT. Unlabeled edges have a probability value equal to 1. Note that an RDT can be defined for any named rule, but we treat such trees as subtree of the "main RDT" by collapsing the nodes corresponding to their invocations (i.e. rule call nodes).

Reliability Model Formulation. We define *move* a single computation step of an ASM, which consists of firing the updates produced by the main rule, if they do not clash. Since the main rule of an ASM has no parameters and there are no free global variables in the rule declarations of an ASM, the notion of a move does not depend on a variable assignment. A *run* of an ASM M is a finite or infinite sequence s_0, s_1, ...s_{i-1},s_i ... of states of the machine, where s_0 is an initial state and each s_i is obtained from s_{i-1} by firing the main rule. As long as the machine can make a move – from state s_{i-1} to state s_i –, the run proceeds, requiring the values of monitored functions from the environment for the next

[1] Let us recall (see definition 2.4.5 in [4]) that *consistency of updates* guarantees that the ASM locations are never simultaneously updated to different values. This fault must be removed in order to have a correct ASM specification.

machine move. If in a state the machine cannot produce a consistent update set or no update set at all, then the state is the last state in the run. Because of the non-determinism of the choose rule and of moves of the environment, an ASM can have several different runs starting from the same initial state [4].

Since the reliability of a system depends on how it will be used (usage profile or operational profile) [7], the ASM *usage profile* for the move mov_i depends on the current state s_{i-1}. The reliability of the move mov_i of an ASM M, under failures independence assumption, can be therefore defined as the product:

$$Rel^i_M = Rel^i_{I_e} \cdot Rel^i_{I_M} \cdot Rel^i_{I_{IU}} \tag{1}$$

where the reliability: 1) $Rel^i_{I_e}$ ($Rel^i_{I_M}$) is the probability that the functions modifiable by the environment (M) are *correctly updated* by the environment (M) (read: they are updated by the environment (M) with values that do not violate the invariants, if any, in which the functions appear); and, 2) $Rel^i_{I_{IU}}$ is the probability that no inconsistent update is performed (i.e., the consistency of updates is guaranteed).

Inconsistent Update Failures (IU). For the sake of space, below, we only show how to compute $Rel^i_{I_{IU}}$. We describe the model formulation only in part and provide, instead, an illustrative example.

```
module InconsistentUExample
import ... // Other module imports
signature:
dynamic controlled f, g, h: Integer
dynamic monitored z, t, e, h, q: Integer
definitions:
rule r_a =
par
   seq  f(1) := 3   g(1) := e – 1 endseq
   g(1) := t-7
endpar
rule r_b =
  if (t > 0)
    then par  g(1) := u-(h*q)  g(1) := 8 endpar
  else h(1) := 6 endif
main rule r_main =
 if(z>=4) then  r_a[]
    else par  r_b[]  g(1) := 6  endpar endif
```

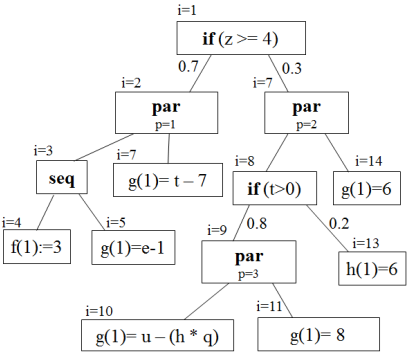


Fig. 1. The *IU* Running Example

Model Formulation: Let I be the event "the input of the ASM is correct (i.e., the monitored functions are correctly updated by the *env*)", and O the event "no inconsistent update is ever performed". Then the Rel^i_{IU} reliability is:

$$Rel^i_{IU} = P(I \cap O) = P(I)P(O|I) \tag{2}$$

where $P(I) = 1$ under failures independence assumption. $P(O|I)$ is a function (not shown here) of two main terms. The first term depends on the probability that the child nodes of the root of the *RDT* perform an inconsistent update among each other, and can be estimated, for example, using the formulas introduced in [1] for the *error propagation* probability. The second term depends on the probability $P(O_p|I)$ that the `par` rules ($1 \leq p \leq |\mathcal{P}|$) do not perform inconsistent updates, where O_p is the event "no inconsistent update is ever performed by the `par` rule p".

Similarly to the probability $P(O|I)$, the probability $P(O_p|I)$ can be obtained recursively visiting the *RDT* in postorder. $P(O_p|I)$ is a function (not shown here) of two main terms. The first term is a function of the probability that the child nodes of p perform an inconsistent update among each other. The second term depends on the probabilities that its *direct descendant* nodes of type `par` do not perform inconsistent update.

Example: Figures 2 shows an example of ASM module and its corresponding *RDT*. By visiting the *RDT* in preorder, we have associated the labels to the nodes, where the labels assume values $i \in [1, 14]$ and $p \in \{1, 2, 3\}$. The probability $P(O_p|I)$ of the `par` rule $p = 2$ depends on: 1) the probability that its child nodes (i.e., $i = 8$ and $i = 14$) perform an inconsistent update among each other; and, 2) the probability that its *direct descendant* `par` node $p = 3$ does not perform inconsistent updates. □

3 Conclusions and Future Work

We presented a reliability model for ASMs. We intend to demonstrate the tool-supportability of our method and experiment it on different ASM case studies. We want also extend it by considering multi-agent ASMs and Turbo ASMs.

References

1. Abdelmoez, W., Nassar, D.M., Shereshevsky, M., Gradetsky, N., Gunnalan, R., Ammar, H.H., Yu, B., Mili, A.: Error Propagation In Software Architectures. In: IEEE International Symposium on Software Metrics, pp. 384–393 (2004)
2. Arcaini, P., Gargantini, A., Riccobene, E.: Automatic review of Abstract State Machines by meta-property verification. In: Proc. of NASA Formal Methods Symposium (2010)
3. The ASMETA toolset website (2006), `http://asmeta.sf.net/`
4. Börger, E., Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer (2003)
5. Immonen, A., Niemelä, E.: Survey of reliability and availability prediction methods from the viewpoint of software architecture. Software and System Modeling 7(1), 49–65 (2008)
6. Liu, S., Tamai, T., Nakajima, S.: A framework for integrating formal specification, review, and testing to enhance software reliability. International Journal of Software Engineering and Knowledge Engineering 21(2), 259–288 (2011)

7. Musa, J.: Operational profiles in software-reliability engineering. IEEE Software 10(2), 14–32 (1993)
8. Ouimet, M., Lundqvist, K.: The TASM Toolset: Specification, Simulation, and Formal Verification of Real-Time Systems. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 126–130. Springer, Heidelberg (2007)
9. Yacoub, S., Cukic, B., Ammar, H.: Scenario-based reliability analysis of component-based software. In: Proc. of 10th International Symposium on Software Reliability Engineering, pp. 22–31 (1999)

A Simplified Parallel ASM Thesis

Klaus-Dieter Schewe[1] and Qing Wang[2]

[1] Software Competence Center Hagenberg
and Johannes-Kepler-University Linz, Austria
kd.schewe@scch.at, kd.schewe@faw.at, kdschewe@acm.org
[2] Research School of Computer Science, The Australian National University, Australia
qing.wang@anu.edu.au

Abstract. We present an idea how to simplify Gurevich's parallel ASM thesis. The key idea is to modify only the bounded exploration postulate from the sequential ASM thesis by allowing also non-ground comprehension terms. The idea arises from comparison with work on ASM foundations of database transformations.

Keywords: Abstract State Machine, bounded exploration, synchronous parallelism.

1 Parallel ASMs in the Light of the DB-ASM Thesis

The sequential ASM thesis refers to Gurevich's seminal work on sequential algorithms, which he defined by three simple, intuitive postulates [2]. The parallel ASM thesis refers to the generalisation by Blass and Gurevich to (synchronous) parallel algorithms [1]. In our own previous work we adapted the sequential ASM thesis to characterise database transformations in general by a variant of ASMs called DB-ASMs [3]. The core of the approach is similar to Gurevich's seminal work: provide a language-independent definition through a set of intuitive postulates, define formally an abstract machine model, and prove that the postulates are exactly captured by the machine model. On these grounds it was among others possible to tailor the model to specific data model, e.g. XML [4].

In order to remove the restriction that only sequential algorithms are exploited in the DB-ASM thesis we generalised our previous work to the case of synchronous parallel database transformations [5]. The key observations are the following:

- In the DB-ASM thesis the *sequential time postulate* was modified to support non-determinism. An additional *bounded non-determinism postulate* was added to restrict the non-determinism to choice among results of a database query. If for the time being we concentrate on deterministic algorithms, Gurevich's sequential time postulate can be kept without change. As a further consequence there would be no need for the bounded non-determinism postulate.

J. Derrick et al. (Eds.): ABZ 2012, LNCS 7316, pp. 341–344, 2012.

- In the DB-ASM thesis the *abstract state postulate* was modified to capture the finiteness of databases by meta-finite structures. Furthermore, a *background postulate* as in the parallel ASM thesis was added. The use of meta-finite states in the abstract state postulates is not more than a restriction needed for databases, and the background postulates only makes the need for certain values and operations obvious. Thus, if we want to remove the specific focus on database transformations, we have to keep Gurevich's abstract state postulate without change plus the background postulate.
- The key difference between the DB-ASM thesis and the sequential ASM thesis lies in the extension of the bounded exploration witnesses, which in the sequential ASM thesis could only consist of ground terms, whereas the *bounded exploration postulate* in the DB-ASM thesis.

Thus, if we only modify the bounded exploration postulate as in the DB-ASM thesis and add the background postulate, we obtain a different characterisation of synchronous parallel algorithms.

2 A Modified Set of Postulates

Follwing our discussion in the previous section we can define *(deterministic, synchronous) parallel algorithms* by four postulates: the sequential time postulate 1, the abstract state postulate 2, the background postulate 3, and a new bounded exploration postulate 4, in which the notion of access term is generalised.

Postulate 1 (sequential time postulate). A parallel algorithm t is associated with a non-empty set of states $\mathcal{S}_t$ together with a non-empty subset $\mathcal{I}_t$ of initial states, and a one-step transition function $\tau_t : \mathcal{S}_t \to \mathcal{S}_t$.

Postulate 2 (abstract state postulate). All states $S \in \mathcal{S}_t$ of a parallel algorithm t are structures over the same signature Σ_t, and whenever $(S, S') \in \tau_t$ holds, the states S and S' have the same base set B. The sets $\mathcal{S}_t$ and $\mathcal{I}_t$ are closed under isomorphisms, and each isomorphism σ from S_1 to S_2 is also an isomorphism from $S_1' = \tau_t(S_1)$ to $S_2' = \tau_t(S_2)$.

Postulate 3 (background postulate). Each state of a parallel algorithm t must contain an infinite set of reserve values, truth values and their connectives, the equality predicate, the undefinedness value $\bot$, and a background class $\mathcal{K}$ defined by a background signature V_K that contains at least a binary tuple constructor $(\cdot)$, a multiset constructor $\langle\cdot\rangle$, and function symbols for the following operations: pairing and projection for pairs, empty multiset $\langle\rangle$, singleton $\langle x\rangle$, binary multiset union $\uplus$, general multiset union $\biguplus x$, $AsSet$, and $\mathbf{I}x$ ("the unique") on multisets.

For the bounded exploration postulate we only have to adapt the notion of access term.

Definition 1. An *access term* is either a ground term α or a triple (f, β, α) of terms, the variables $x_1, \ldots, x_n$ in which refer to the arguments of $f \in \Sigma$. The interpretation of (f, β, α) in a state S is the set of locations

$$\{f(a_1, \ldots, a_n) \mid val_{S,\zeta}(\beta) = val_{S,\zeta}(\alpha) \text{ with } \zeta = \{x_1 \mapsto a_1, \ldots, x_n \mapsto a_n\}\}.$$

Structures S_1 and S_2 *coincide* over a set T of access terms iff the interpretation of each $\alpha \in T$ and each $(\beta, \alpha) \in T$ over S_1 and S_2 are equal.

Postulate 4 (bounded exploration postulate). For a parallel algorithm t there exists a fixed, finite set T of access terms of t (called *bounded exploration witness*) such that $\Delta(t, S_1) = \Delta(t, S_2)$ holds whenever the states S_1 and S_2 coincide over T.

Example 1. Let us look at the example of graph inversion, which was also used in [1] to motivate the postulates for parallel algorithms. Here we can assume two relations in the signature Σ, the unary relation NODE, and the binary relation EDGE. The algorithm can be simply expressed by the rule

forall x, y **with** NODE$(x) \wedge$ NODE(y)
do EDGE$(x, y) :=$ ¬EDGE(x, y) **enddo**

Here, $T = \{(Edge, Node(x) \wedge Node(y), true)\}$ is a exploration boundary witness.

Example 2. Take the rather well known LCR algorithm for leader election in a ring. Then we need a unary function UID, which maps node identifiers onto numerical values, a unary function SEND, which does the same, an incoming message function IN that maps node identifiers to numerical values or *unknown*, a unary function STATUS that maps node identifiers the values *unknown* or *leader*, a static unary function VAL mapping node identifiers to numerical values, and a constant N for the number of nodes in the ring.

In an initial state we have UID(i) = VAL(i), SEND(i) = VAL(i), IN(i) = *unknown*, and STATUS(i) = *unknown* for all i.

Then the database transformation algorithm can be specified as follows:

forall i **with** $0 \leq i \leq N - 1$
do par
IN$(i + 1 \mod N) :=$ *send*(i) ‖
if IN$(i) \neq$ *unknown* $\wedge$ IN$(i) >$ UID(i)
then SEND$(i) :=$ IN(i)
endif ‖
if IN$(i) \neq$ *unknown* $\wedge$ IN$(i) =$ UID(i)
then STATUS$(i) :=$ LEADER
endif
endpar enddo

Here, the bounded exploration witness T contains three access terms (STATUS, $\alpha(i), true$), (IN, $\alpha(i), true$) and (SEND, $\alpha(i), true$), in which $\alpha(i)$ is defined as $i \geq 0 \wedge i < N$.

3 An Expected Equivalence Result

As exemplified in [5] we expect that the proof of the DB-ASM thesis in [3] will carry over the formalisation of parallel algorithms by the four postulates in the previous section. That is, we expect to be able to prove the following:

Conjecture 1. *Each ASM $\mathcal{M}$ defines a parallel algorithm t with the same signature and background as $\mathcal{M}$, and for every parallel algorithm t there exists an equivalent ASM $\mathcal{M}$ with the same background.*

Proof (idea). As in the corresponding proofs in [2,3,5] we expect that the key will be to prove the following: For a parallel algorithm t and a state $S \in \mathcal{S}_t$ there exists a rule r_S such that $\Delta(t, S) = \Delta(r_S, S)$, and r_S only uses critical terms.

It will be easy to see that for any update $u = (f(a_1, \ldots, a_n), a_0) \in \Delta(t, S)$ the values $a_0, \ldots, a_n$ are critical and hence representable by terms involving variables from access terms in T. The interesting case is that at least one of the terms $t_0, \ldots, t_n$ is not a ground term. If none of terms $t_0, \ldots, t_n$ contain multiset operators, then (ℓ, a_0) is represented by the assignment rule $f(t_1, \ldots, t_n) := t_0$. Otherwise, without loss of generality, we can replace the terms $t_1, ..., t_n$ of an assignment rule $f(t_1, ..., t_n) := t_0$ with the variables $x_{t_1}, ..., x_{t_n}$, such that

$$\textbf{seq par } x_{t_1} := t_1 \ldots x_{t_n} := t_n \textbf{ endpar } f(x_{t_1}, ..., x_{t_n}) := t_0 \textbf{ endseq}$$

represents the update. If the outermost function symbol of term t_0 is a multiset operator ρ, , e.g. $t_0 = \rho(m)$ where $m = \langle t_0' |$ for all values $\overline{a} = (a_1, ..., a_p)$ in $\overline{y} = (y_1, ..., y_p)$ such that $val_{S,\zeta[x_1 \mapsto b_1, ..., x_k \mapsto b_k]}(\varphi(\overline{x}, \overline{y})) = true\rangle$, and $\overline{x}$ denotes a tuple of variables among $x_1, ..., x_k$, then each assignment rule $f(x_{t_1}, ..., x_{t_n}) := t_0$ can be replaced by parallel assignments $f_i(x_{t_1}, ..., x_{t_n}) := t_0'$ followed by an application of the multiset operator to obtain $f(x_{t_1}, ..., x_{t_n})$.

The rest of the proof should be almost the same as in [5].

As a consequence, the characterisation of (deterministic, synchronous) parallel algorithms by the four postulates in the previous sections would be equivalent to Blass's and Gurevich's set of postulates in [1] but simpler.

References

1. Blass, A., Gurevich, Y.: Abstract state machines capture parallel algorithms. ACM Transactions on Computational Logic 4(4), 578–651 (2003)
2. Gurevich, Y.: Sequential abstract state machines capture sequential algorithms. ACM Transactions on Computational Logic 1(1), 77–111 (2000)
3. Schewe, K.D., Wang, Q.: A customised ASM thesis for database transformations. Acta Cybernetica 19(4), 765–805 (2010)
4. Schewe, K.D., Wang, Q.: XML database transformations. Journal of Universal Computer Science 16(20), 3043–3072 (2010)
5. Schewe, K.-D., Wang, Q.: Synchronous Parallel Database Transformations. In: Lukasiewicz, T., Sali, A. (eds.) FoIKS 2012. LNCS, vol. 7153, pp. 370–383. Springer, Heidelberg (2012)

Refactoring Abstract State Machine Models

Hamed Yaghoubi Shahir[1], Roozbeh Farahbod[2], and Uwe Glässer[1]

[1] Software Technology Lab, Simon Fraser University, B.C., Canada
[2] SAP Research, Karlsruhe, Germany
{syaghoub,glaesser}@sfu.ca, roozbeh.farahbod@sap.com

Abstract. The Abstract State Machine (ASM) method proposes the concept of *ground models* for analyzing a target system based on pseudo-code-like descriptions for reasoning about system properties in terms of state machine runs over abstract data structures. This highly iterative process builds on stepwise refinement of ground models that evolve with progressing understanding of functional system requirements. Usually, as complexity increases, reorganization of a model's internal structure helps enhance its flexibility and robustness. While this approach is common practice, the underlying principles are usually left implicit. In this paper, we propose refactoring patterns to restructure abstract machine models with the goal of improving their intelligibility and maintainability.

1 Introduction

Best engineering practice calls for a system to be modeled prior to construction, so one can rigorously inspect and reason about the key system properties, making sure these are both well understood and properly established. This basic principle applies to software systems as well. Software systems design builds on abstract models that, implicitly or explicitly, reflect the underlying assumptions, requirements, design decisions and conformance criteria, and serve as a reference for implementation, integration, testing and beyond over the entire lifecycle of a software system. Arguably, software models ought to be abstract, disregarding any insignificant details as much as possible, but otherwise be precise and reliable 'blueprints' for construction, maintenance and further development. Precision is an essential quality for resolving potential ambiguities and also to uncover design flaws and weaknesses that may (and too often do) go unnoticed otherwise.

The Abstract State Machine (ASM) method [1] defines the concept of *ground model* [2] for analyzing and reasoning about dynamic properties of a system based on pseudocode-like descriptions in terms of state machine runs over abstract data structures. A common characteristic of ground models is a direct correspondence between the intuitive understanding of the system requirements to be modeled and their abstract state machine representation, this way simplifying the task of establishing correctness and completeness of a model through observation and experimentation. With no way to prove, in a strict sense, correctness or completeness of the requirements and/or the design, the transition from informal and often ambiguous descriptions to mathematical representations in the initial formalization step of a model poses a notorious problem.

J. Derrick et al. (Eds.): ABZ 2012, LNCS 7316, pp. 345–348, 2012.

In practice, modeling is a highly non-linear process with feedback loops due to progressing understanding of how to formalize abstract functional requirements. A ground model evolves over time through stepwise refinement [2], and also as a result of revisions aiming at enhancing its representation; once it has reached a stable representation, additional changes and extensions notoriously occur due to *requirements creep* over the course of the system development process, and even beyond, due to future development activities. In light of model-driven systems engineering, a model keeps evolving over its entire lifecycle, all the way from its inception to the retirement of the system it represents, thus, calling for frequent reorganization. While reorganization of models is common practice, the underlying principles are usually left implicit, albeit, there is a connection to principles for reorganizing code. In software engineering, refactoring of software is defined as "a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behaviour" [3]:

> *With refactoring the emphasis changes. You still do upfront design, but now you don't try to find the solution. Instead all you want is a reasonable solution. You know that as you build the solution, as you understand more about the problem, you realize that the best solution is different from the one you originally came up with. With refactoring this is not a problem, for it no longer is expensive to make the changes.*

In this paper we propose *refactoring patterns* to restructure ASM and CoreASM [4] models with the goal of improving intelligibility and maintainability of formal machine models in real-life system development. Building on established principles for refactoring used in software development, we explore *why* and *when* refactoring in formal modeling should occur, and also illustrate *how* to do it.

2 Pattern-Based Approach

This section describes a pattern-based approach to model-driven engineering of software intensive systems using ASM ground modeling. Patterns possess a rationale and tangibility that appeals to the human mind. They are practically proven human-devised solutions to *common* problems. Researchers and engineers do not invent patterns, but rather discover those already in use [5].

For proposing a set of patterns or a *pattern language* for a specific context, it is necessary to study existing patterns and also to explore various successfully solved examples in the given context. In this work, we have studied different categories of software patterns such as Architectural [6], Design [7], Reengineering [8], and Refactoring [3] patterns so as to explore a set of appropriate patterns that are more suitable for ASM models. In this ongoing research, we have identified a number of refactoring patterns (*extract rule*, *inline rule*, *split rule*, *merge rule*, *expand rule*, *rename rule*, *parameterize rule*, and *remove middle rule*) for ASM (or CoreASM) models. The following section provides a brief discussion of two of the patterns, namely the *extract rule* and *parameterize rule*.

2.1 Extract Rule Pattern

This pattern applies to a complex ASM rule with one or more coherent internal fragments that each can be considered a separate ASM rule. Such a rule may (and often does) diminish readability of the specification, introduce abstraction level inconsistency, and lead to duplication of specification fragments.

Problems: More precisely, this pattern addresses the following problems.

1. *Increased complexity*: The rule is too long and contains too much information which reduces its readability and increases the specification complexity.
2. *Abstraction level inconsistency*: The rule contains specification fragments with different abstraction levels, or the specification as a whole has inconsistent procedural abstraction with rules that mix different levels of abstraction.
3. *Redundancy*: The coherent specification fragments within this rule may have already been repeated in other parts of the specification, which reduces maintainability of the specification and increases the possibility of errors.

Solution: The solution is to identify these coherent fragments within the ASM rule and extract them as new ASM rules that are referred to in the original rule. Figure 1(a) illustrates the main idea by means of an example of this pattern.

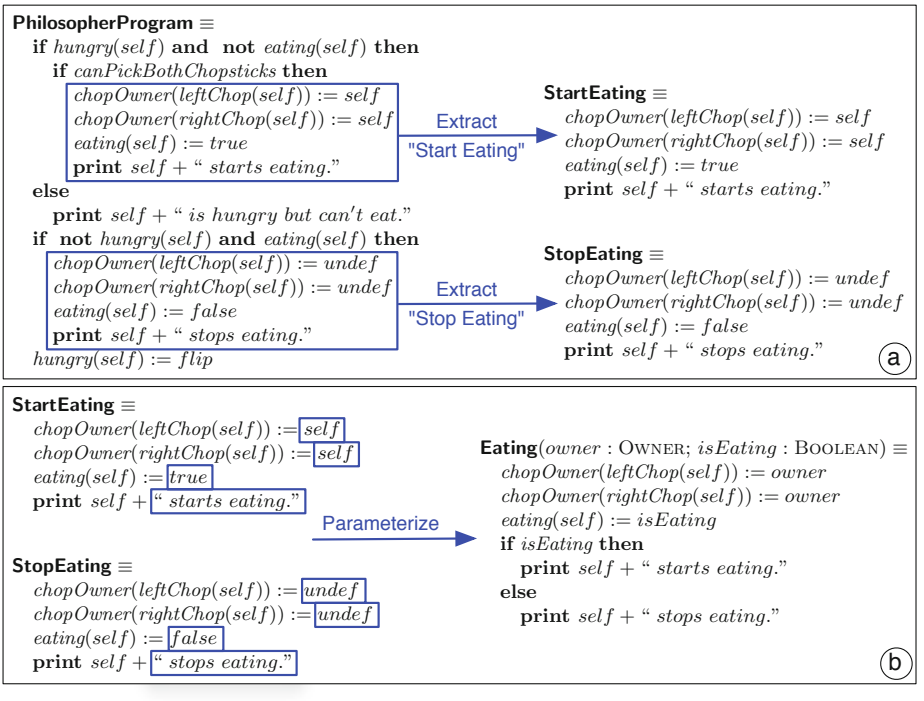


Fig. 1. a) Extract Rule Pattern, b) Parameterize Rule Pattern

2.2 Parameterize Rule Pattern

This pattern applies to ASM rules that are basically identical but apply different terms or constants that may be considered parameters of one and the same rule.

Problem: The pattern mainly addresses the problem of *redundancy* where the core concept specified by the rules is repeated in the specification, which in turn reduces maintainability and increases vulnerability to errors.

Solution: The solution is to 1) introduce a new rule $R_p(...)$ that captures the essential idea behind these rules into a parameterized ASM rule, and 2) replace all calls to the these rules with calls to R_p with the specific terms and constant values passed as arguments. Figure 1(b) illustrates the main idea by means of an example of the Parameterize Rule pattern.

3 Concluding Remarks

Ground modeling is an effective instrument for turning abstract requirements into precise formal models for requirements analysis and design. Building ground models is a non-linear process. Examples of such models in the literature typically present the final 'product' but virtually never show any intermediate models produced on the way to the final one. The work presented here focuses on the principles of restructuring models in a more systematic, pattern-based manner, resembling the use of refactoring in software engineering. We are working on a comprehensive description of a *pattern language* for refactoring ASM models, and a framework for combining refinement and refactoring. Our goal is to integrate such patterns as an advanced feature into the CoreASM tool environment.

References

1. Börger, E., Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer (2003)
2. Börger, E.: Construction and Analysis of Ground Models and their Refinements as a Foundation for Validating Computer Based Systems. Formal Aspects of Computing 19(2), 225–241 (2007)
3. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
4. Farahbod, R., Gervasi, V., Glässer, U.: Executable Formal Specifications of Complex Distributed Systems with CoreASM. In: Science of Computer Programming. Elsevier (in Press, 2012)
5. Devedzic, V.: Software Patterns. In: Handbook of Software Engineering and Knowledge Engineering, vol. 2, pp. 645–671 (2002)
6. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern Oriented Software Architecture: A System of Patterns. Wiley (1996)
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley (1995)
8. Demeyer, S., Ducasse, S., Nierstrasz, O.: Object-Oriented Reengineering Patterns. Morgan Kaufmann (2003)

Continuous Behaviour in Event-B: A Sketch

Richard Banach[1,⋆], Huibiao Zhu[2,⋆⋆], Wen Su[2], and Xiaofeng Wu[2]

[1] School of Computer Science, University of Manchester,
Oxford Road, Manchester, M13 9PL, U.K.
banach@cs.man.ac.uk
[2] Software Engineering Institute, East China Normal University,
3663 Zhongshan Road North, Shanghai 200062, P.R. China
{hbzhu,wensu,xfwu}@sei.ecnu.edu.cn

Abstract. Including provision for continuously varying behaviour as well as discrete state change is considered for Event-B. An extension of Event-B is sketched that accommodates continuous events (called pliant events) in between familiar discrete events (called mode events).

1 Introduction

In this short paper, we briefly sketch an extension of Event-B that accommodates genuinely continuous behaviour (as well as discrete state changes). The motivation for this is to enable Event-B to engage better with problems exhibiting such behaviour in an essential way, as is increasingly needed in applications. A fully worked out presentation, including more extensive discussion of semantics, proof obligations for machine consistency and for refinement, and consideration of finegraining and coarsegraining issues, will appear elsewhere. We assume familiarity with Event-B.

2 Extending Event-B with Continuous Behaviour

To adequately capture behaviour over real time, we model time as an interval $\mathcal{T}$ of the real numbers $\mathbb{R}$, with a finite left endpoint to represent the time at which the initial state of the model is created, and with a right endpoint which is finite or infinite, depending on whether the dynamics is finite or infinite. Now, the values of all variables become functions of $\mathcal{T}$. By convention, $\mathcal{T}$ partitions into a sequence of left-closed right-open intervals, $\langle [t_0 \ldots t_1), [t_1 \ldots t_2), \ldots \rangle$, the coarsest partition such that all discontinuous changes take place at some boundary point t_i. We have two kinds of variable. **Mode variables** only change discontinuously between elements of a discrete type. These are

⋆ The majority of the work reported in this paper was done while the first author was a visiting researcher at the Software Engineering Institute at East China Normal University. The support of ECNU is gratefully acknowledged.

⋆⋆ Huibiao Zhu is supported by National Basic Research Program of China (No. 2011CB302904), National High Technology Research and Development Program of China (No. 2011AA010101 and No. 2012AA011205), National Natural Science Foundation of China (No. 61061130541 and No. 61021004).

J. Derrick et al. (Eds.): ABZ 2012, LNCS 7316, pp. 349–352, 2012.

just like traditional B variables, and restricting to these recovers traditional Event-B. **Pliant variables** have types which include topologically dense sets, and which can evolve both continuously and via discrete changes. In a typical interval $[t_i \ldots t_{i+1})$, the mode variables will be constant, but the pliant variables will change continuously. However, continuity alone still allows for a wide range of mathematically pathological behaviours, so we make the following restrictions:

I Zeno: there is a constant δ_{Zeno}, such that for all i needed, $t_{i+1} - t_i \geq \delta_{\mathsf{Zeno}}$.

II Limits: for every variable x, and for every time $t \in \mathcal{T}$, the left limit $\lim_{\delta \to 0} x(t-\delta)$ written $\overrightarrow{x(t)}$ and right limit $\lim_{\delta \to 0} x(t+\delta)$, written $\overleftarrow{x(t)}$ (with $\delta > 0$) exist, and for every t, $x(t) = \overleftarrow{x(t)}$. [N. B. At the endpoint(s) of $\mathcal{T}$, any missing limit is defined to equal its counterpart.]

III Differentiability: The behaviour of every pliant variable x in the interval $[t_i \ldots t_{i+1})$ is given by the solution of a well posed initial value problem $\mathcal{D}\,xs = \phi(xs, t)$ (where xs is a relevant tuple of pliant variables and $\mathcal{D}$ is the time derivative). "Well posed" means that $\phi(xs, t)$ has Lipschitz constants which are uniformly bounded over $[t_i \ldots t_{i+1})$ bounding its variation with respect to xs, and that $\phi(xs, t)$ is measurable in t.

With I-III in place, the behaviour of every pliant variable is piecewise smooth, with the smooth variation being described by a suitable differential equation (DE).

As well as two kinds of variable, we have two kinds of event. Mode events are like traditional Event-B events. They describe discontinuous changes, though they can involve both mode and pliant variables. Their syntax is identical to traditional Event-B events, except that before-values are to be interpreted as left limits (at the moment t_i that the event occurs), and after-values are to be interpreted as the corresponding right limits. For example, a straightforward generic mode event, decorated with this limit information, could be written in the usual notation as:

StdEv
WHEN $grd(\overrightarrow{u}, \overrightarrow{i})$
ANY $\overleftarrow{u}$
WHERE $BApred(\overrightarrow{u}, \overrightarrow{i}, \overleftarrow{u})$
THEN $u := \overleftarrow{u}$
END

We also have pliant events. These involve changes to pliant variables alone, and they describe continuous change. While a mode event is a single before-/after-value pair, a pliant event is a family of before-/after-value pairs, parameterized by points in time falling within the relevant time interval $[t_i \ldots t_{i+1})$. For every member of this family, the before-value is always the value at t_i, while the after-value is the value at t, for t in the open interval $(t_i \ldots t_{i+1})$. Thus the change from before- to after-value does not take place instantaneously. Pliant events need new syntax, for which we give two variants:

PliEv
STATUS pliant
WHEN $grd(u(t_{\mathrm{L}(t)}))$
WHERE $BDApred(u(t), i(t), t)$
SOLVE $DE(u(t), i(t), t)$
END

PliEv
STATUS pliant
WHEN $grd(u(t_{\mathrm{L}(t)}))$
ANY $u(t)$
WHERE $BDApred(u(t), i(t), t)$
THEN $u := u(t)$
END

In the left hand syntax, we specify a differential equation to be solved. In the right hand syntax we just specify the continuous behaviour required directly, for those cases where this is known (since differentiating the behaviour, only to have to solve the resulting DE for it immediately afterwards is obviously wasteful).

Much of the structure of these two cases is similar, and we can discuss both of the cases together to begin with, starting from the top. After the header line we have the 'STATUS pliant' line. This introduces a new event status, the pliant status, signaling to any tool processing the syntax that a pliant event is being defined.

In the remainder of the structure we see the notation $\mathrm{L}(t) = max\{i \mid t_i \leq t\}$, which has a counterpart $\mathrm{R}(t) = min\{i \mid t_i > t\}$. These map any time t to the left and right ends of the interval containing t during a run, and are used to refer generically to the initial and final time values of the interval during which the continuous behavour is specified.

The next line is the 'WHEN' line, and contains any required facts about the initial values of the relevant state variables when the pliant transition starts; it also contains any additional guard information. Unlike the guard of a mode event, it cannot depend on any input that the pliant event needs, since any such input will last throughout the interval $(t_{\mathrm{L}(t)} \ldots t_{\mathrm{R}(t)})$, and so its value at the time instant $t_{\mathrm{L}(t)}$ has measure zero; this is of insufficient weight to influence the start of a pliant event.

At this point, the two structures start to diverge. On the left, for the case governed by a differential equation, we have a 'WHERE' line, which contains a before-during-and-after predicate *BDApred*. Since this case is predominantly governed by the differential equation, there is often little or nothing for the *BDApred* to specify, so it will often be very simple, or can be omitted entirely. On the other hand, if there are additional constraints that need to hold during the pliant event, such as facts concerning specific values of time, deadlines, or anything else, such constraints can be placed here.

The next line is where the action is, since it includes the differential equation in the 'SOLVE' clause. The differential equation specifies what the values of the state variables are to be during the interval of interest, but it does so indirectly. In general, the DE depends on the current values of the state variables and on the inputs which are received through the course of the interval of interest. That completes the description of the left hand case.

On the right, we pick up at the 'ANY' line. This works a lot like the ANY clause of a mode event, but it (typically) names a family of after-values that is time dependent, defined over the open interval $(t_{\mathrm{L}(t)} \ldots t_{\mathrm{R}(t)})$. The named values are utilised in the before-during-and-after predicate *BDApred*. Unlike the previous case, where the *BDApred* predicate is typically simple or absent, this time, the *BDApred* predicate is the entity that actually does the hard work of specifying the after-values, so, unlike previously, it will be nontrivial. The after-values are actually assigned in the 'THEN' clause on the next line, just as for normal Event-B. As usual, if the expressions to be assigned are known explicitly, then the ANY and WHERE clauses can be omitted, and the required values can be assigned directly. All this is therefore just like normal Event-B, except that everything is parameterised by time. This completes the description of the other case.

A continuous Event-B machine, with mode and pliant events as described, is said to be **well formed** iff every mode transition enables a pliant transition (but no mode

transition) on completion, every non-final pliant transition enables a mode transition during its execution (which then fires, preempting the pliant transition), and a final pliant transition either continues indefinitely (non-termination) or becomes undefined at some point (finite termination).

A run of such a machine starts with an initial mode transition which sets up the system initial system state, and then, pliant transitions alternate with mode transitions. The last transition (if there is one) is a pliant transition (whose duration may be finite or infinite).

Since time has a different character from other variables, if time is mentioned explicitly in a system model, then the name of the time variable has to be indicated to a tool such as Rodin. A convenient way of doing this is to have a 'TIME t' declaration. The value of time may be linked to the rest of the system model in the *INITIALISATION* event which may then be given a guard such as 'WHEN $t = 0$'.

An alternative approach to time utilises one or more *clocks*. The difference between a time variable and a clock variable is that a clock may run fast or slow with respect to (real) time, so its derivative must be specified during pliant events that use it. It can also be reset by mode events (specifically during initialisation). Clock variables can be declared as 'CLOCK clk'.

3 Discussion, POs

Above, we sketched the essentials of an extension of Event-B intended to cope with genuinely continuous behaviours, such as are increasingly needed in the hybrid and cyber-physical applications being developed today. That so few of these are developed using a refinement mindset is the main reason for considering Event-B here.

As with conventional Event-B, the semantics is expressed via proof obligations, which we cover briefly now. Events have to be feasible; mode events via the usual PO, pliant events via a PO that asserts a solution to the DE in some interval. Events have to preserve the invariants; mode events as usual, pliant events continuously over the course of the DE solution. Alternation between mode and pliant events is handled by POs that demand the relevant disjunctions of guards under appropriate conditions.

Properly defined refinement between machines is crucial of course. Explicit POs become much simpler if mode events must be refined by mode events (in the usual way), and if pliant events must be refined by pliant events (in a way that preserves the passage of time, and maintains the invariants during the course of the two pliant events). Relative deadlock freedom may be demanded of the mode events, and separately, of the pliant events.

One issue not present in conventional Event-B, is that relatively long lived pliant events may need to get broken up into short lived ones, in particular, when modeling the implementation of continuous behaviour by digital means utilising a high sampling frequency. To deal with this we can introduce suitable skips that momentarily interrupt the long lived pliant event. POs can be designed so that such skips do not alter the dynamics of the system, while nevertheless breaking up a long transition into short steps that can later be refined in the usual way.

Formal Verification of PLC Programs Using the B Method*

Haniel Barbosa and David Déharbe

Departamento de Informática e Matemática Aplicada, UFRN, Brazil
hanielbbarbosa@gmail.com,
deharbe@dimap.ufrn.br

Abstract. In this paper we propose an approach to verify PLC programs, a common platform to control systems in the industry. Programs written in the languages of the IEC 61131-3 standard are automatically translated to B machines and are then amenable to formal analysis of safety constraints and general structural properties of the application. This approach thus integrates formal methods into existing industrial processes.

Keywords: B method, PLC, safety critical systems, formal methods.

1 Introduction

In many industries, such as mass transport and energy, it is very common to use PLCs in control applications. Those applications are mostly programmed according to IEC 61131-3 [1], an international standard that specifies the five standard PLC programming languages, namely: LD (Ladder Diagram) and FBD (Function Block Diagram), graphical languages; IL (Instruction List) and ST (Structured Text), textual languages; and SFC (Sequential Function Chart), that shows the structure and internal organization of a PLC. It is not rare that a variation of such languages is employed too.

As the complexity of the applications increases, and as various are safety critical, it is important to ensure their reliability. Formal methods are a mean to fulfill this requirement, as testing and simulation (the *de-facto* method in many branches) can left flaws undiscovered. However, it is difficult to integrate formal methods in the industrial process since most control engineers are not familiarized with formal verification.

Some recent works have been trying to integrate formal methods and PLC verification, using different approaches. In [7], the authors created a new language combining ST and Linear Temporal Logic, ST-LTL, to ease the use of formal verification by control engineers. [6] presents a method to verify applications using Safety Function Blocks with timed-automata through model-checking and

* Project supported by ANP. CNPq grants 560014/2010-4 and 573964/2008-4 (National Institute of Science and Technology for Software Engineering—INES, www.ines.org.br).

J. Derrick et al. (Eds.): ABZ 2012, LNCS 7316, pp. 353–356, 2012.

simulation. A model-driven engineering approach is used in [5] to generate models in a FIACRE language from LD programs. To this date, these approaches are concerned only with parts of the IEC 61131-3 standard.

Our approach already handles two of the five languages of the standard, namely SFC and ST, and we are working to extend it to be fully compliant. To do so, we use the PLCopen [3] standard, which provides an interface representing all the IEC 61131-3 languages in an XML-based format. Another goal of our approach is to be capable of verifying legacy programs in numerous different PLCs. We have built an intermediary model based in the PLCopen interface that is loaded from the PLC programs and then is used to *automatically* generate a B model.

B [2] is a formal method that can be used to specify systems and through proof obligations demonstrate its correctness according to the specification, avoiding state-explosion problem. It is practical and competitive to develop safety-critical systems, with the correct methodology and tools. Using the B method we can verify safety constraints through the proof obligations and also to check structural issues, such as *deadlock freedom*, using animation tools such as ProB [4]. Thus, we increase the confidence in the PLC applications and facilitate the use of formal methods in the industry.

Next section presents details on the different phases of our method as well as an example. In the end we have some discussions and future work.

2 The Method

The method we are proposing consists of three main phases:

1. translate the information in the PLC programs into an intermediary model (from now on called "PLC model"), either from a PLC program or from an XML file in the PLCopen standard;
2. generate from it a B model that makes possible to check the structural and safety properties of the project;
3. and at last complete the formal model with these safety properties, derived from the project requirements (manually, for now).

2.1 Towards the PLC Model

The PLC model may be generated either directly from an XML in the PLCopen standard or from the programs in some hybrid language, based on the IEC 61131-3 standard. Such languages are common as adaptations to specific domain PLCs may be necessary.

We projected a compiler to analyze the programs; it deals with the elements of the standard languages and may be customized to any existing differences, to accommodate any new language. This way we can deal with legacy programs that are not strictly standard compliant. To deal with XML, we use a reader module to load the PLC model along with the uncustomized compiler.

Once the PLC model is constructed we are able to work independently from the PLC programs or the PLCopen to generate the B specification.

2.2 Generation of the B Model

A good architecture is essencial to generate a good model, as well as to define which information from which language will be responsible for which elements of the B model, since it is common to have PLCs using more than one language. As so far we are working only with SFC and ST, our architecture can handle just the elements of these two languages. When we deal with the elements of the other languages the architecture will be adapted to include them.

The architecture of the model is depicted in figure 1. This model represents a PLC that process signals required by a control application, having them as *inputs*. The PLC *outputs* are treated as local variables; it is no loss of generality to deal with them like that since we are dealing with the PLCs only as independent components. The safety requirements will concern mostly these *outputs*.

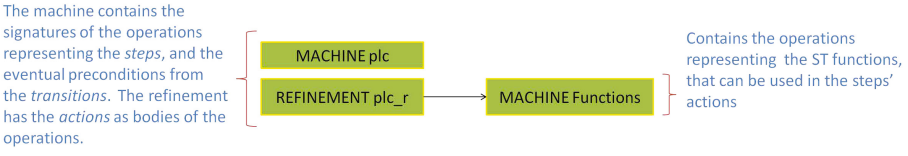


Fig. 1. Shows the architecture of the B model generated by a SFC + ST PLC

For the PLC component, the **operations** are derived from the SFC steps, as whether or not they have **preconditions** is based in the SFC transitions. The body of these **operations** and the content of their **preconditions** are the result of the translation of ST statements. The **operations** of the *Functions* **machine** are also constructed with the translated ST statements.

Figure 2 shows a little example of the generation of the B model. Due to space limitations, we do not present the whole process.

The next step is to add safety requirements. Since the PLC programs do not represent such constraints explicitly, they are manually extracted from the project requirements and inserted into the model as **invariants** of the

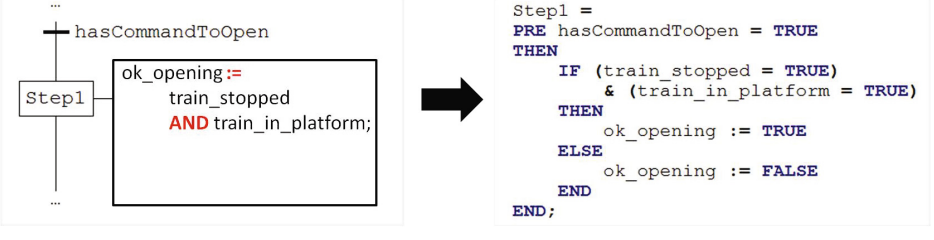


Fig. 2. Example shows a SFC step, its transition and its action (in ST) as base for the generation of a B **operation**

refinement, conditions that must always hold as the PLC actions are performed. For example, the requirement "a train must be stopped and in the platform to open its doors" is inserted as the invariant: $(ok\_opening = \text{TRUE}) \Rightarrow ((train\_stopped = \text{TRUE}) \ \& \ (train\_in\_platform = \text{TRUE}))$. Tools like AtelierB can perform automatic verification of their consistency and point out where lies any problem, guiding its treatment.

The formal model can also be evaluated with an animation tool like ProB, making possible to *model check* the model to verify structural properties (*deadlock*, *liveness*, LTL conditions...).

3 Discussions and Future Work

We have overviewed a method to carry out formal verification of the languages of the IEC 61131-3 standard for PLC programming through the automatic generation of a B specification. There is still much work to be done, but the results so far are quite satisfactory.

Future work lies most in expanding the generation of the B model to the other language. The safety constraints are still manually derived from the requirements, but we plan to automatize this process. We are about to start a case study with the company ClearSy, strongly involved with the B method and safety critical systems engineering, in a real project in the railway field to execute problem diagnosis in high speed trains.

References

1. IEC: IEC 61131-3 - Programmable controllers. International Electrotechnical Comission Standards (2003)
2. Abrial, Jr.: The B-book: assigning programs to meanings. Cambridge University Press, Cambridge (2005)
3. PLCopen: XML Formats for IEC 61131-3. PLCopen Technical Committee 6 (2009)
4. Leuschel, M., Butler, M.: ProB: A Model Checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 855–874. Springer, Heidelberg (2003)
5. Farines, J., de Queiroz, M.H., da Rocha, V.G., Carpes, A.A.M., Vernadat, F., Crégut, X.: A model-driven engineering approach to formal verification of PLC programs. In: IEEE EFTA (2011)
6. Ljungkrantz, O., Åkesson, K., Fabian, M., Yuan, C.: A Formal Specification language for PLC-based Control Logic. In: Proc. of 8th IEEE International Conference on Industrial Informatics, pp. 1067–1072 (2010)
7. Soliman, D., Frey, G.: Verification and Validation of Safety Applications based on PLcopen Safety Function Blocks using Timed Automata in Uppaal. In: Proceedings of the Second IDAC Workshop on Dependable Control of Discrete Systems (DCDS), pp. 39–44 (2009)

A Practical Event-B Refinement Method Based on a UML-Driven Development Process

Thiago C. de Sousa[1,2,⋆], Paulo Sérgio Muniz Silva[2], and Colin F. Snook[3]

[1] State University of Piaui
thiagocsousa@usp.br
[2] University of São Paulo
paulo.muniz@poli.usp.br
[3] University of Southampton
cfs@ecs.soton.ac.uk

Abstract. Event-B is a formal method that allows flexible modelling and refinement of systems. However, it is hard to convince developers to adopt it because they are not used to mathematical models and it doesn't provide any practical refinement method. On the other hand, UML has become the *de facto* standard for software modelling since it provides an easy graphical notation and nowadays it is supported by many practical process such as ICONIX. In this paper we propose a method for Event-B refinement based on a diffused UML-driven development process. So far, we have defined the steps of the method and the translation of most of the artifacts presented in ICONIX to Event-B.

Keywords: Event-B, refinement, UML, ICONIX.

1 Introduction

Event-B [1] is a state model-based formal method for modelling systems based on predicate logic and set theory where the refinement mechanism and the consistency checking are guaranteed by mathematical proof obligations. However, its use is not a common practice because mathematics are not well understood by regular systems analysts. Furthermore, Event-B doesn't provide a practical refinement method with a well defined set of steps. So, the developers get confused about some questions like: how do they should start the model?, what is the next step?, when should they add the invariants?, etc.

On the other hand, the majority of the developers works better with visual languages. So, the Unified Modelling Language (UML) has become the *lingua franca* of software development and has been supported by a lot of process and tools around the world. One of the most known and practical UML-based process is ICONIX [2], which has some important features, such as the use of only four (Use Cases, Robustness, Sequence and Class) diagrams and a very simple step-by-step to make the refinement, with regular verification checkpoints.

⋆ The first author is partially supported by CNPq.

J. Derrick et al. (Eds.): ABZ 2012, LNCS 7316, pp. 357–360, 2012.

In this work we present an approach for the integration of ICONIX with Event-B in order to provide a practical, visual and formal refinement method. More precisely, we show how the steps/stages of the ICONIX process can be used as a refinement guide. In the next section we explain our approach (a work in progress) in more details, showing the architecture overview and the main tasks of each phase. In section 3, we show some related works, and finally we reserve the last section for further discussions and the ongoing work.

2 From ICONIX to Event-B

As we can see in Figure 1, our proposal uses the first three phases of the ICONIX process, with the incorporation of an artifact in each one to describe the invariants, as a refinement method for Event-B.

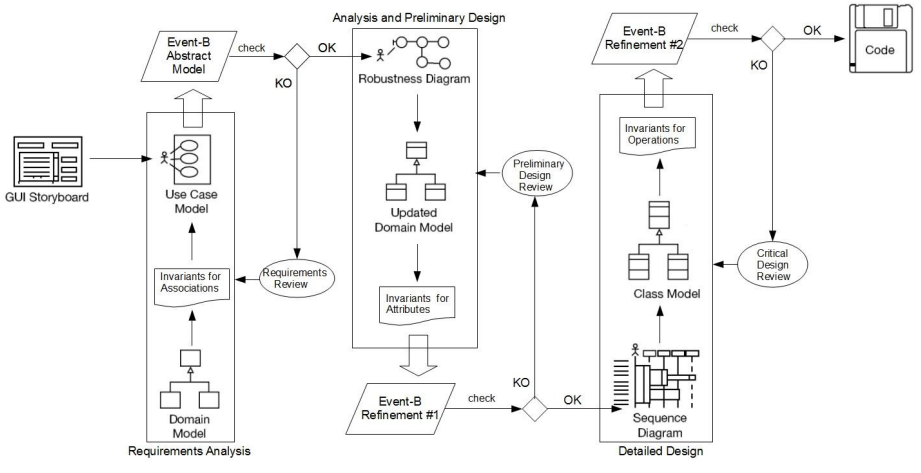


Fig. 1. The overview of our proposal

The first phase (Requirements Analysis) starts with the design of the Domain Model, which identifies the main concepts of the problem domain and their relations from extracting the nouns and noun phrases of the requirements document. After that, the invariants for the associations of the Domain Model are defined. The first stage ends with the development of informal interface prototypes (GUI Storyboard) that are used as a starting point to identify and describe the use cases textually, facilitating the design of the Use Cases Diagram. Finally, three of these artifacts (Domain Model, Invariants and Use Cases Diagram) are formally translated to Event-B (the former is mapped to sets and their relations and the latter is mapped to events) in order to compose an Event-B Abstract Model.

After the translation of all these three artifacts, the developer can use the Rodin Platform [3] to make the automatic verification of consistency among them. If there is a violation of any proof obligation that was generated during the translation, the developer must perform a review of the requirements and fix the artifacts issues. If no error is detected, the next stage can be started.

The second phase (Analysis and Preliminary Design) starts with the robustness analysis. This activity requires the examination of the narrative of the textual description of a use case and the identification of a first set of objects that participate in it, in order to ease the refinement of the abstract model and the migration to the next stage. The output artifact of this task is the Robustness Diagram, which is a hybrid between an Activity Diagram and a Class Diagram. After the design of all Robustness Diagrams (one per use case), new classes and attributes are discoveries, which are used to update the Domain Model. After that, the invariants for the attributes and new class associations of the updated Domain Model are described. Finally, these artifacts are formally translated to Event-B in order to compose the first refinement of the Event-B Model.

After the translation of all these artifacts, the developer can follow the script again and use the Rodin to check the consistency among these models as well as the correctness of the refinement. If there is a violation of any proof obligation or refinement rule that was generated during the translation, the developer must provide a review of the preliminary design and adjust the artifacts problems. If no error is identified, the next stage can be started.

The third phase (Detailed Design) starts with the design of the Sequence Diagrams, which are derived from the Robustness Diagrams and used to refine the behaviour of the Control classes, distributing methods to the Entity and Boundary classes. After that, the Domain Model is updated and refined with the methods that were allocated to each class, and transformed into a Class Model. The invariants, as well as guards (pre-conditions) and actions (post-conditions) for the methods of the Class Model, are defined. Finally, these artifacts are formally translated to Event-B in order to compose the second refinement of the Event-B Model.

After the translation of all these artifacts, the developer can use again the Rodin tool to make the formal verification and check the consistency among these artifacts and the correctness of the refinement. If there is a violation of any proof obligation or refinement rule, the developer must provide a critical review of the detailed design. If no error is detected, the next stage can be started, which can be to start coding or to continue for next refinement.

It is important to emphasize that the translation of the Domain Model, the updated Domain Model and the Class Model have the same mapping to Event-B presented by the UML-B [4]. The invariants (which are included during the process) are represented directly in the Event-B notation for now. The translation of the Use Case and Robustness Diagrams have already been defined implicitly via the equivalence of their meta-classes with elements of the Event-B language. The translation of the Sequence Diagrams is still being developed. Due to the limited space, we will not show the details of these translations.

3 Related Work

There are some works that also propose the use of UML diagrams as front-end for a formal notation and a UML-based process as a guideline to the refinement. Runde *et al* [5] and Younes and Ayed [6] present formal methods based on Sequence and Activity Diagrams, respectively. However, these approaches do not show how to represent the system static part. Chen *et al* [7] and Ahrendt *et al* [8] present sound and formal UML-based methods. However, these approaches are not based on any known methodology, which does not encourage their use.

4 Discussions and Ongoing Work

In this paper we have proposed an approach for using the phases of the ICONIX process as a guideline to an Event-B refinement in order to provide a practical and visual method and convince the developers to adopt the formal modelling approach. To achieve this goal completely, we need to finish the mapping from the Sequence Diagram to Event-B and elaborate the proof obligations and refinement rules that are needed. We are also cogitating the use of a visual language for the invariants representation in order to have a full graphical method.

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering, 1st edn. Cambridge University Press, New York (2010)
2. Rosenberg, D., Stephens, M.: Use Case Driven Object Modeling with UML: Theory and Practice. Apress (2007)
3. Abrial, J.-R., Butler, M., Hallerstede, S., Voisin, L.: An Open Extensible Tool Environment for Event-B. In: Liu, Z., Kleinberg, R.D. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 588–605. Springer, Heidelberg (2006)
4. Snook, C., Butler, M.: UML-B: Formal modeling and design aided by UML. ACM Trans. Softw. Eng. Methodol. 15, 92–122 (2006)
5. Runde, R.K., Haugen, Ø., Stølen, K.: The Pragmatics of STAIRS. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 88–114. Springer, Heidelberg (2006)
6. Ben Younes, A., Ben Ayed, L.J.: From UML Activity Diagrams to Event B for the Specification and the Verification of Workflow Applications. In: Proceedings of the IEEE International Computer Software and Applications Conference (COMPSAC 2008), pp. 643–648. IEEE Computer Society, Washington, DC (2008)
7. Chen, Z., Liu, Z., Stolz, V., Yang, L., Ravn, A.P.: A refinement driven component-based design. In: Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems, pp. 277–289. IEEE Computer Society, Washington, DC (2007)
8. Ahrendt, W., Beckert, B., Hähnle, R., Schmitt, P.H.: KeY: A Formal Method for Object-Oriented Systems. In: Bonsangue, M.M., Johnsen, E.B. (eds.) FMOODS 2007. LNCS, vol. 4468, pp. 32–43. Springer, Heidelberg (2007)

Learn and Test for Event-B – A Rodin Plugin

Ionut Dinca, Florentin Ipate, Laurentiu Mierla, and Alin Stefanescu

University of Pitesti, Department of Computer Science
Str. Targu din Vale 1, 110040 Pitesti, Romania
name.surname@upit.ro

Abstract. The Event-B method is a formal approach for reliable systems specification and verification, being supported by the Rodin platform, which includes mature plugins for theorem-proving, model-checking, or model (de)composition features. In order to complement these techniques with test generation and state model inference from Event-B models, we developed a new feature as a Rodin plugin. Our plugin implements a model-learning approach to iteratively construct an approximate automaton model together with an associated test suite. Test suite optimization is further applied according to different optimization criteria.

1 Introduction

This short tool paper presents the implementation in the Rodin platform of the general method "learn-and-test" described in our previous paper [1]. For a given Event-B model [2], the method constructs, in parallel, an *approximate automaton* model and a *test suite* for the system. The approximate model construction relies on a variant of Angluin's automata learning algorithm [3,4], adapted to finite cover automata [5]. A *finite cover automaton* represents an approximation of the system which only considers sequences of length up to an established upper bound ℓ. Crucially, the size of the cover automaton, which normally depends on ℓ, can be significantly lower than the size of the exact automaton model. In this way, by appropriately setting the value of the upper bound ℓ, the state explosion problem normally associated with constructing and checking state based models can be addressed. The proposed approach also allows for a gradual construction of the model and of the associated test suite (reusing information between iterations), which fits well with the central notion of refinement in Event-B [2].

2 Tool Overview

A bird's eye view of the tool is depicted in Fig. 1. The tool takes as input an Event-B model M and a finite bound ℓ and outputs a finite cover automaton approximating the set of feasible sequences of events of M of length up to ℓ and a test suite, i.e. a set of sequences including test data that make the sequences executable. The core procedure of "Model Learning" generates a cover automaton using a variant of automata learning from queries [3]. Simply put, a cover

J. Derrick et al. (Eds.): ABZ 2012, LNCS 7316, pp. 361–364, 2012.

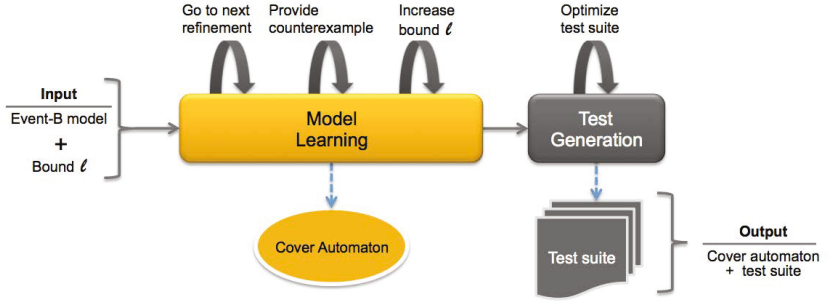


Fig. 1. Overview of the tool features

automaton for a finite set of words of length up to ℓ, is an automaton accepting all these words but also sequences that may be longer than ℓ. The cover automaton can be incrementally improved by providing more information according to the three loops in the figure. Thus, one can: (a) use the "next refinement" of the Event-B model that contains more information; or (b) "provide a counterexample" by manually or automatically providing sequences that are feasible in the Event-B model, but are not in the cover automaton or vice-versa (the counterexamples are used in the learning procedure); or (c) increase the bound ℓ and implicitly feed the learning engine with longer sequences which again will increase the precision of the finite state approximation. At any point in time, one can use the constructed cover automaton to generate tests that exercise different sequences through the Event-B model. There are many existing methods for test generation from finite state models. In our case, we use internal information from the learning procedure, which maintains a so-called "observation table" that keeps track of the learned feasible sequences. Sets of feasible sequences in this table will provide the desired test suite. Note that during the feasibility check of the sequences in Event-B, test data are also generated. The implementation of feasibility check uses a constraint-solver for Event-B available in ProB [6]. The obtained test suite satisfies strong criteria for conformance testing (usually required in the embedded system domain) and may be large. If weaker test coverage like state-, transition- or event-coverage are desired, optimization algorithms can be applied on the test suite according to the rightmost loop in Fig. 1. We implemented different optimizations as proposed by one of the co-authors in [7] using the jMetal framework which is based on genetic algorithms.

Our tool is a Rodin plugin implemented in Java (with 5,500 LOC) and can be called on any Event-B model with several levels of refinements. Installation instructions and screenshots can be found at: `http://wiki.event-b.org/index.php/MBT_plugin`. Ongoing extensions of the tool tackle not only refinement, but also different types of Event-B decompositions. Experiments with different Event-B models (publicly available on the DEPLOY repository - `http://deploy-eprints.ecs.soton.ac.uk`)

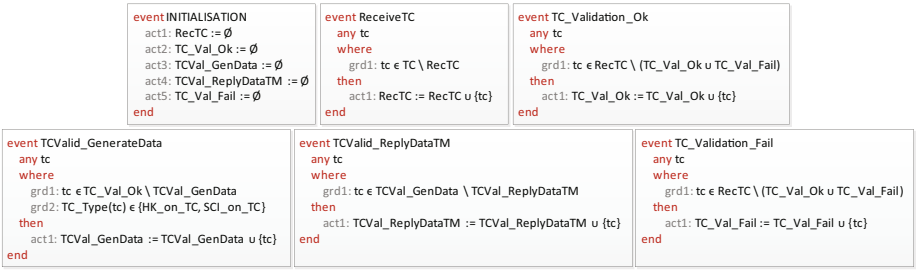


Fig. 2. The events of the abstract machine M_0 in BepiColombo Event-B model [8]

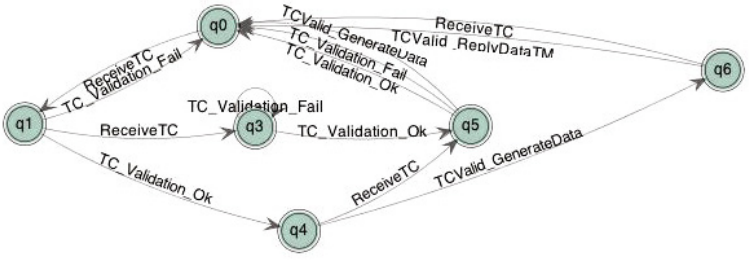


Fig. 3. The generated cover automaton for M_0 and $\ell = 4$

produced good results even for large models like BepiColombo [8] (whose third refinement exhibits 17 events and 18 variables that could induce a large explicit state space for the model). This example is discussed below.

3 The Tool Applied to an Example

An Event-B model has a context providing the data types and an abstract state machine providing the dynamic behavior. The machine has a set of events, which are the first class citizens of Event-B, that operate on a set of global variables. The modeling complexity is addressed using *refinement* as a mechanism to construct a series of more abstract models before reaching a very specific one. For instance, in a refinement step, new variables and new events can be introduced and the existing events can be made more specific.

The BepiColombo aerospace mission is one of the case study used in the DEPLOY project (`http://deploy-project.eu`). In [8], a part of BepiColombo is modeled in Event-B using several levels of refinements (combined with atomic and model decompositions which we do not address here). The main goal of the system is specified at a very abstract level, with a machine M_0. The system specification is concretized through three further refinement levels, M_1, M_2 and M_3. Fig. 2 presents the five events of M_0, plus a special event called 'Initialisation'. Each event has local parameters preceded by the keyword *any*, a guard preceded by the keyword *where*, and an action code preceded by the keyword

then. There exist also global variables (like *RecTC* of type *Set*), that are initialized in the event 'Initialisation'. Once the 'Initialisation' event is executed, the modeled system moves from one state to another by choosing one event with its guard true and executing its action code.

Given the BepiColombo Event-B model and an upper bound ℓ, we incrementally construct finite cover automata that will eventually cover all executable event sequences of length less than or equal to ℓ. Fig. 3 (plotted by our tool) illustrates the cover automaton for the first machine M_0 and $\ell = 4$, minimal by construction, having the initial state marked with q_0, transitions labeled with event names and final states marked with a double circle. Starting from the state q_0, the event sequences can be identified by following the transitions with the purpose of reaching the automaton final states, representing a subset of the communication scenarios the spacecraft system may encounter.

A conformance test suite heavily exercising the system would consist of 17 test cases. Conformance testing is a very powerful test type since it covers all states and all transitions of the automaton and also checks each state and the initial and destination states of each transition. However, for a lighter test coverage like event coverage, a test suite consists of only 2 test cases (of length up to 4): (a) `ReceiveTC(tc1), TC_Validation_Ok(tc1), TCValid_GenerateData(tc1), TCValid_ReplyDataTM(tc1)` and (b) `ReceiveTC(tc2), TC_Validation_Fail(tc2)`.

Acknowledgments. This work was supported by project DEPLOY, FP7 EC grant no. 214158, and Romanian National Authority for Scientific Research (CNCS-UEFISCDI) grant no. PN-II-ID-PCE-2011-3-0688 (project MuVet) and grant no. 7/05.08.2010.

References

1. Ipate, F., Dinca, I., Stefanescu, A.: Model learning and test generation using cover automata. Submitted to IEEE Trans. on Software Engineering (2012)
2. Abrial, J.-R.: Modeling in Event-B – System and Software Engineering. Cambridge University Press (2010)
3. Ipate, F.: Learning finite cover automata from queries. Journal of Computer and System Sciences 78(1), 221–244 (2012)
4. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. 75(2), 87–106 (1987)
5. Câmpeanu, C., Sântean, N., Yu, S.: Minimal cover-automata for finite languages. Theoret. Comput. Sci. 267(1-2), 3–16 (2001)
6. Leuschel, M., Butler, M.J.: ProB: an automated analysis toolset for the B method. Int. J. Softw. Tools Technol. Transf. 10(2), 185–203 (2008), `http://www.stups.uni-duesseldorf.de/ProB`
7. Dincă, I.: Multi-objective Test Suite Optimization for Event-B Models. In: El-Qawasmeh, D. E. (ed.) ICIEIS 2011, Part I. CCIS, vol. 251, pp. 551–565. Springer, Heidelberg (2011)
8. Salehi Fathabadi, A., Rezazadeh, A., Butler, M.: Applying Atomicity and Model Decomposition to a Space Craft System in Event-B. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 328–342. Springer, Heidelberg (2011)

Event-B Code Generation: Type Extension with Theories

Andrew Edmunds, Michael Butler, Issam Maamria, Renato Silva, and Chris Lovell

University of Southampton, UK
ae2@ecs.soton.ac.uk

Abstract. The Event-B method is a formal modelling approach; our interest is the final step, of generating code for concurrent programs, from Event-B. Our Tasking Event-B tool integrates Event-B to facilitate code generation. The theory plug-in allows mathematical extensions to be added to an Event-B development. When working at the implementation level we need to consider how to translate the newly added types and operators into code. In this paper, we augment the theory plug-in, by adding a *Translation Rules* section to the tool. This enables us to define translation rules that map Event-B formulas to code. We illustrate the approach using a small case study, where we add a theory of arrays, and specify translation rules for generating Ada code.

1 Introduction

Using Event-B [1] and Tasking Event-B, we have the ability to model and implement single and multi-tasking software systems, see [2]. It may be the case that we need some new mathematical type, and in many cases, the type will need to be implemented. In this paper we describe, using an example, how new types can be added to an Event-B Theory. We then describe the tool's new translation rule feature, which we use to define translation rules for generating Ada code. The rules describe the mapping from Event-B types, and mathematical notation, to implementable code fragments. This work has been undertaken as part of the EU DEPLOY [4] project.

The basic structural features of Event-B are contexts and machines. Contexts describe the static features of a system using sets and constants. Machines are used to describe the variable features of a system in the form of state variables and guarded events; system properties are specified using the invariants clause. Theories for Rodin are described in [3] where we introduce mathematical extensions; with this, we can add new types, operators, and rules. Rules, such as rewriting, compare a source against patterns defined in the theory rule-base. When a pattern matches with the source, the source is replaced by new elements, as determined by the pattern. In the work described in the paper, we add the ability to specify translation rules for code. This uses pattern matching, in a similar way; but, instead of initiating a substitution of new elements in place of old, we generate text for use in the main code generator. To facilitate the specification of new rules one

J. Derrick et al. (Eds.): ABZ 2012, LNCS 7316, pp. 365–368, 2012.

can introduce Type Parameters, and specify Datatypes. Users can also introduce new operators and theorems. Proof obligations are generated to verify the soundness of the rules, and the prover is augmented with the new rules, when the theory is *deployed*. In our work, we extend the theory with translator rules.

1.1 An Array Theory

The first step is to create a new theory of arrays, we introduce an array of type T. The array is a new operator, which takes a powerset of type T as an argument. The array has the following definition, using set-notation, where n is the length of the array.

$$array(s : \mathbb{P}(T)) \triangleq \{n, f \cdot n \in \mathbb{Z} \land f \in 0 \mathinner{..} (n-1) \rightarrow s | f\}$$

Since we have a low-level specification we consider implementation issues: generally, arrays are fixed-length implementations. We introduce *arrayN*, parametrized by n, which fixes the array length by stating $card(s) = n$.

$$arrayN(n : \mathbb{Z}, s : \mathbb{P}(T)) \triangleq \{a | a \in array(s) \land card(s) = n\}$$

The array constructor operator *newArray* has an integer parameter n, representing the array length; and, additionally a value x, of type T for initialising the array elements. The array construction operator has the following definition,

$$newArray(n : \mathbb{Z}, x : T) \triangleq (0 \mathinner{..} (n-1)) \times \{x\}$$

Additonally, *newArray* requires a well-definedness condition, $n \in \mathbb{N}$. For the array update, we have the definition,

$$update(a : array(T), i : \mathbb{Z}, x : T) \triangleq a \lhd\!\!+ \{i \mapsto x\}$$

update has the well-definedness condition $i \in 0 \mathinner{..} (card(a) - 1)$. We can see that array a is updated with value x at index i. The case study, which we use to illustrate the extension mechanism and the link to code, omits irrelevant detail.

2 An Event-B Model

In the following model, we make use of the array operator that we have just introduced. In the invariant, we type *cbuf* as an array of size *maxbuf* of integers. *maxbuf* is a constant defined in a seen context. The second parameter defines the element types, which in this case are integers. In the Initialisation event, we specify the size, and initial value for the array elements in the clause *act1*. In our model, we initially set *maxbuf* elements to be zero.

variables cbuf a b
invariants
@inv1 cbuf ∈ arrayN(maxbuf, ℤ)
@inv2 ...
initialisation
@act1 cbuf := newArray(maxbuf, 0)
@act2 ...

An example of the update to the array can be seen in the following *Put* event, which inserts an element into the array in action *act2*.

event ***Put*** $\triangleq$
any x
where
@grd1 $\mathtt{x} \in \mathbb{Z}$
@grd2 $\mathtt{b} \geq \mathtt{a} \Rightarrow \mathtt{b} - \mathtt{a} < \mathtt{maxbuf}$
then
@act1 $\mathtt{b} := (\mathtt{b} + 1) \bmod (\mathtt{maxbuf} + 1)$
@act2 $\mathtt{cbuf} := \mathtt{update}(\mathtt{cbuf}, \mathtt{b} \bmod \mathtt{maxbuf}, \mathtt{x})$
end

3 Adding Translation Rules

The next step is to add translation rules to the theory that defines arrays. We add the Ada *Translator Target* section, shown below, and use this to define the translation of the newly introduced operators. Metavariables (variable patterns) are introduced to facilitate type inference, and pattern matching during translation. Using the rules defined in the *Translator Rules* section, we match the patterns in the following way to determine which translation is applicable. We specify the operator to be matched on the left side of the rule (left of the $\mapsto$ operator), and the translated text, on the right side. Since there is no formal link between the pattern on the left side and text output on the right side of the rule, we use visual inspection to verify that the rule is correct.

Translator ***Target***: Ada
Metavariables
$\mathrm{s} \in \mathbb{P}(\mathrm{T})$, $\mathrm{n} \in \mathbb{Z}$, $\mathrm{a} \in \mathbb{Z} \leftrightarrow \mathrm{T}$, $\mathrm{i} \in \mathbb{Z}$, $\mathrm{x} \in \mathrm{T}$
Translator Rules
`trns1:` ...
`trns2:` $a = update(a, i, x) \mapsto a(i) := x$
`trns3:` $newArray(n, x) \mapsto$ (`others =>` x)
Type Rules
`typeTrns1:` $arrayN(n, s) \mapsto$`array`$(0 \,..\, n-1)$ `of` s

In the example shown, the array *update* operator maps to the Ada array assignment $a(i) := x$. The construction operator *newArray* provides the initial values in parameter x, and maps to the Ada clause ($others$ => x) which sets all elements (using *others*) of an array to x. In addition to translation rules, we can add type rules; these are used to map the type, as defined in the theory, to an implementable type for use in the generated code. In the type rule *typetrans1* we specify that the type $arrayN(n, s)$ should be mapped to the Ada type clause $array(0 \,..\, (n-1))\ of\ s$.

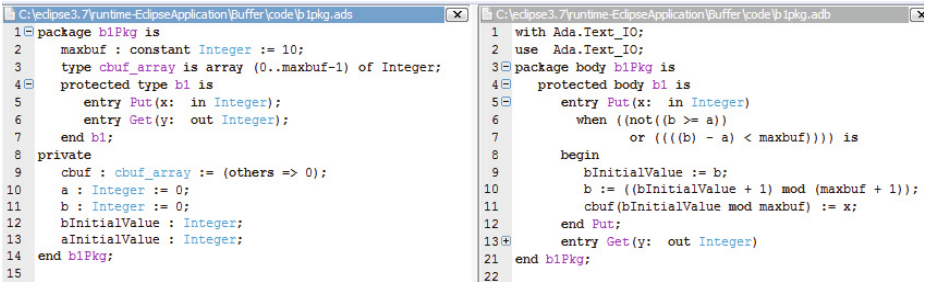


Fig. 1. Generated Ada Code

4 Conclusion

We used the our tool [5] to generate the code shown in Fig. 1. The rules allow translation of operators, and type definitions, to implementation constructs. We can now see how the Event-B relates to the generated code. In line 3 of the code, on the left hand side of the figure, we see an Ada type declaration statement. This results from applying *typeTrns1* to the invariant *inv1*. The translation of the type s, i.e. the mapping of the Event-B $\mathbb{Z}$, to the Ada *Integer* type, is handled by the pre-defined Ada theory, and not shown here. In Ada, we must 'instantiate' the *cbuf_array* type, so in line 9 we declare *cbuf* to be of type *cbuf_array*, and initialise the values. The translator makes use of the translation rule *trns3*, matching the pattern $arrayN(n, s)$ with the assignment of the initialisation action *act1*. The update rule has also been translated in line 11 on the right side of Fig. 1, which uses *trns2* applied to *act2* of the *Put* event.

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
2. Edmunds, A., Butler, M.: Tasking Event-B: An Extension to Event-B for Generating Concurrent Code. In: PLACES 2011 (February 2011)
3. Maamria, I., Butler, M., Edmunds, A., Rezazadeh, A.: On an Extensible Rule-Based Prover for Event-B. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) ABZ 2010. LNCS, vol. 5977, pp. 407–407. Springer, Heidelberg (2010)
4. The DEPLOY Project Team. Project Website, `http://www.deploy-project.eu/`
5. The Deploy Wiki Website - Code Generation Activity, `http://wiki.event-b.org/index.php/Code_Generation_Activity`

Formal Proofs for the NYCT Line 7 (Flushing) Modernization Project

Denis Sabatier, Lilian Burdy, Antoine Requet, and Jérôme Guéry

ClearSy
320, avenue Archimède
Les Pléiades III – Bât A
13857 Aix-en-Provence Cedex 3
France
denis.sabatier@clearsy.com

Abstract. The New York City Transit Authority has included formal proofs at system level as part of the safety assessment for its New York subway Line 7 modernization project, based on the CBTC from Thales Toronto. ClearSy carries out these proofs. In this paper, we describe the expected results and benefits of such proofs. We also discuss the methodology, in particular the importance of obtaining a natural language precursor for proofs. This step is paramount to find the simplest reasons why the design ensures the wanted properties.

1 Introduction

The New York City subway Line 7 (Flushing) modernization project consists in installing a Communication Based Train Control (CBTC) system and updating the existing interlocking system. The CBTC system is designed and installed by THALES Toronto. The main benefits of this modernization will be:

- Reduced headway;
- Signal and track circuit simplification;
- Extended routing possibilities, for instance in case of track failures.

The New York City Transit authority (NYCT) decided to include **formal proofs** at system level as part of the safety assessment for this project. The French company ClearSy carries out these proofs, using the Event-B method and Atelier-B toolkit.

In this paper, we discuss the methodology applied to obtain these formal proofs and the conclusions so far. The proof project is currently progressing (January 2012) and is estimated at 40% of total workload.

2 Goals and Expected Benefits

The main goal is to obtain a formal proof for the main safety properties of the system: no collision and no over-speeding. For instance, the first selected property

J. Derrick et al. (Eds.): ABZ 2012, LNCS 7316, pp. 369–372, 2012.

"no collision" is detailed as follows: **a train will never encounter any obstacle or any switch not locked in correct position** (so it is not only "no collision", the name is a simplification). So we seek to formulate a set of well defined assumptions, such that this "no collision" property can be obtained from these assumptions by pure logical reasoning only.

Those assumptions cover every relevant aspect of the system, from internal design (for example: what algorithm is used to calculate safe braking) to external conditions (for example the worst turns that motion determination can tolerate). How deep we go is decided when an assumption is considered as terminal; for instance the property "the motion determination algorithm always detects a wheel starting to slip" can be considered as a terminal assumption, or be the subject of a lower level proof relying on wheel and sensor physical properties.

2.1 Expected Benefits

The expected benefits of having such proofs are:

- Revealing all assumptions needed;
- Reaching a "proof level" confidence for the system properties.

Revealing all assumptions needed is particularly useful:

- External and environment assumptions (for instance: train operator behavior), well defined assumptions are the unique opportunity to check if they really hold and to check if they still hold in case of any change (environmental change for instance).
- Design assumptions (for instance: sensor properties, algorithm details, etc.): well defined assumptions are obviously useful at system creation and in case of system evolution.

In particular, assumptions regarding the software are intended to contribute to the elimination of errors, in spite of our not going inside the code. When we assume a well defined property for a precise piece of code, it is easier to check this property by code review or through dedicated testing (or even future proofs).

3 Methodology

We use two main steps to obtain the desired formal proofs:

1. Write a document explaining how the system ensures the desired properties. We call this **natural language "proof"**, with quotes because it is not a formal proof.
2. Write Event-B models such that the proof performed is the formal equivalent of the natural language "proof".

The first step is based on the fact that we do not use Event-B to understand **why** a property is ensured, but to **validate** that it is really ensured once the "why" is understood. We want to avoid mixing formal notation issues with domain issues. The

problem is that the available documents for a system usually describe *how* the system works, and not *why* it is designed so. This is all the difference between knowing an algorithm and knowing why it produces the wanted result.

The natural language "proof" step is a way to stabilize this understanding before writing Event-B models. Otherwise, if the understanding of the *why* changes often a lot of time is lost by modifying the Event-B models.

Natural language "proofs" are supposed to contain only well defined assumptions, only pure logical steps and resulting properties. Of course, at this stage there is not tool to check these qualities. The most difficult part is to formulate well defined assumptions. We use the following criteria:

> Well defined assumptions: in any possible scenario, it should be possible to state unambiguously if the assumption is true or false in that case.

The second step is normally more straightforward: we use Event-B to construct a proof that mimics the natural language "proof". Through Event-B [1,3] and Atelier-B [2], we obtain a powerful validation of the forecasted proofs. The methodology for ensuring good correspondence between Event-B models and the natural language "proof" is not detailed here (yet to be done); it should rely on checking mathematical objects for each notion used in the proof precursor, and checking events for each possible evolution of these notions.

3.1 Methodology for Natural Language "Proofs"

Up to now, we have worked mainly on natural language "proofs". So let's detail the methodology for these natural language "proofs". Two pitfalls must be avoided:

1. Spending too much time. The system is complex; we must understand *only the minimum needed* to reach the desired proof in order to be efficient.
2. Proving without the system designers. Otherwise it is very easy to introduce subtle biases about how the system really works, leading to a proof irrelevant for the actual system.

To find the simplest reasons why properties hold without falling into the above pitfalls, we use the following steps:

- Play scenarios trying to violate the wanted property (for instance, at top level try to play a scenario leading to a collision), in a light and fast way, until the reasons why violating the property is impossible appear.
- Once the reasons why the property is ensured have appeared, explain those reasons, at first informally then more and more rigorously, until we reach a text with only well defined assumptions and pure logical steps.

4 Results So Far

Using the above methodology, we successfully obtained a proof precursor for the first selected property "anti-collision" (that is detailed as: a train will never encounter an

obstacle or a switch not locked in correct position). Without going into too many details, the reasoning relies on the fact that the existing interlocking system is so that trains are always on "locked" routes (with all switches locked) and in the same direction (this is one of the proved sub-properties). Automated trains then calculate their positions so that their real positions are always inside what they determined (this is another one of the proved sub-properties), and a ground controller dispatches movement limits for each train such the spacing of following trains is ensured.

With this natural language "proof", we discovered that the true reasons why the property (and its sub-properties) holds are quite easy to explain once formulated and communication with domain experts is quite straightforward. However, the number of assumptions required is surprisingly immense and some of them are usually untold, or even not really known, until their importance is made explicit by the proof. The properties that each sub-part must keep appear far more clearly, which should help the designers and the testers to eliminate all errors that could impair those properties.

5 Conclusion

In the development of its own railway products, ClearSy had experiences of system level properties proving that found many errors, even after the product was already coded and testing concluded 0 defects. In many cases, the tests are connected to detailed specifications that already contain the concerned bugs; proving properties at higher level is then the only efficient solution.

This is what we are doing for the line 7 project, at a very large scale.

In industrial projects, the efforts to reach the required performance and to obtain the mandatory documents obviously come first. Extra efforts to reach higher levels of confidence (like actually proving properties, using formulated assumptions and logics) are often not mandatory in today's state of the art.

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
2. Atelier B website, `http://www.atelierb.eu/`
3. Abrial, J.R.: The B-Book. Cambridge University Press (1996)
4. Malvezzi, M., Allotta, B., Rinchi, M., Bruzzo, M., de Bernardi, P.: Odometric estimation for automatic train protection and control systems. Vehicle System Dynamics 19(4-6), 723–739 (2011)

A Pattern for Modelling Fault Tolerant Systems in Event-B

Gintautas Sulskus and Michael Poppleton

University of Southampton
{gs6g10,mrp}@ecs.soton.ac.uk

1 Introduction

Formal methods are used for the specification and verification of software and hardware systems. One class of systems interacts with the outside world through sensors and actuators, and may include nondeterminism from hardware faults or environmental inputs, making modelling more complex.

Previous work [1] by Hayes, Jackson and Jones tried to tackle these issues by primarily focusing on deriving the specification by considering the system in its environment. The authors consider an open sluice gate system which interacts with a non-deterministic environment. The sluice gate controls the flow of water for irrigation purposes, the main requirement being to keep the gate open and closed in a certain time ratio. The system consists of the controller, which actuates the gate via a motor and senses the gate's final positions (open and closed) by two sensors, positioned at the top and the bottom of the gate.

The system requirements may be classified into 3 groups: main functional requirements (e.g. the gate must move up and down), fault management requirements (e.g. in case a fault is detected, the system must shut down) and timing requirements (e.g. in case a fault is detected, the system must shut down within certain amount of time). Derived requirements are specified formally.

The aim of this work is methodological - we sought patterns in our case study development - based on [1] - that may be applicable within this class of systems.

2 A Development Pattern

Our pattern is concerned with three distinct areas: basic modelling, time and fault handling (Fig. 1). The basic model deals with non-time requirements and is based on Butler's Cookbook [2] for control systems modelling. The time module introduces a time infrastructure layer (clock, event delays and deadlines) and time related requirements via further model refinements. The fault handler (FH) is an extension of already modelled fault-free and time modules, meaning that this is not a refinement but rather additional code in already existing refinements. This fault handling can be regarded as a requirements feature, being composed onto the basic development: fault-handling behaviour and state added to existing refinements (basic model and time). This extension inevitably alters the fault-free model because of the need to modify/remove invariants which rely on monitored variables and to add the FH mode's variable checks to fault-free event guards.

J. Derrick et al. (Eds.): ABZ 2012, LNCS 7316, pp. 373–376, 2012.

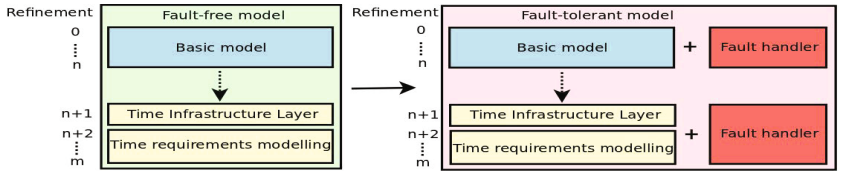


Fig. 1. General pattern structure

2.1 Basic Model

A model of a real world system that may fail, should always include a Fault Management System [3]. Introducing FMS hooks in a fault-free model does not cause much overhead and in later stages this simplifies model extension with fault tolerance by reducing the extent of possible perturbations. One of such "hooks" is FMS event FMS_pass, which checks for possible faults and if none is found, allows the system to proceed normally. The FMS never detects any errors in the fault- free model and lacks fault handling events. However, its presence is recommended since the model should reflect the actual system configuration in both hardware and software levels.

It is assumed that the fault-free model does not have any faults, and so we assume that the environment behaves as expected. To fulfil such an assumption, environment and directly related (e.g. sensing) events are constrained by specifying strict untimed guards (in this case, direct monitored variable usage is allowed) G_u.

2.2 Time

The Time area consists of two layers tackling different time-related problems (Fig. 1, boxes in yellow). The Time Infrastructure Layer (TIL) provides general time infrastructure over controller events to constrain the occurrence of events depending on time. A second layer deals with development-specific timing requirements; here, we discuss only TIL.

Our aim was to develop a generic pattern that could be used in different projects with minimal adaptation effort. Although this layer uses event order information from previous implementation layers, the focus is solely on event time constraints, expressing time required for the event to happen by usage of delays and deadlines [4]. Hence, event sequencing implementation along with other non-time constraint related functionality is out of scope and is the basic model's responsibility.

Structure. TIL is an existing template with configurable slots (Fig. 2). This template is identical regardless of the models in which it is used and does not need to be changed on reuse apart from specific slots.

Invariants. Invariants scope TIL goals, enforcing delays and deadlines after each executed event.

Fig. 2. TIL configuration with configurable areas in yellow

TIL variables. Time is expressed as an integer sequence, *activeEvents* contains timestamp entries of executed events, referring to which event types delay and deadline constraints should be applied. Variables type is $EVENT\_REL \nrightarrow \mathbb{N}$.

Events. The event guard block allows event execution only if there are no active delay constraints, or there exists at least one expired deadline. When the event is executed, it removes entries from *activeEvents*, that allowed (if delay has expired) or forced (if deadline was reached) this event to be executed. Additional entries with the current timestamp are added for every succeeding event type.

Time progress. Event *tick* represents the clock for time progress [5].

Simplicity. Event guard *event_type* is the only element that should be changed in the Machine. The main configuration is done in the Context and relates only to event sequence definitions. Set $EVENT\_TYPES$ contains all possible event types or event group types; $EVENT\_REL$ determines which event type may succeed a prior event type; sets $EVENT\_DELAYS$ and $EVENT\_DEADLINES$ define delay and deadline values for each $EVENT\_REL$ element. Hence, delays and deadlines are specific to a type of event type transition, e.g. $S \mapsto FMS$ may have different time constraints from $S \mapsto FLASH\_LIGHT$.

As in the basic model, we assume that the environment with timing requirements behaves as expected. Thus we constrain environment related events with G_t - timed requirements. The environment event guard space after timing requirement implementation is $G_u \wedge G_t$.

2.3 Fault Handling

Fault handling is based on 4 different system modes, controlled by the FMS (Fig. 3). Under normal conditions, when no faults are detected the system operates in *normal* mode and performs intended operations. This is the only mode present in the fault-free model. In case of a fault, the system enters *suspicious* mode. This mode allows the system to tolerate faults to a certain extent. In case the system manages to recover by itself, it re-enters *normal* mode, otherwise it falls into *faulty* mode and waits for external intervention. The latter mode's main purpose is to perform necessary actions e.g. to shut down, in order to prevent (further) system and environment damage and to inform the operator about the malfunction. After faults are eliminated, the operator is expected to initiate

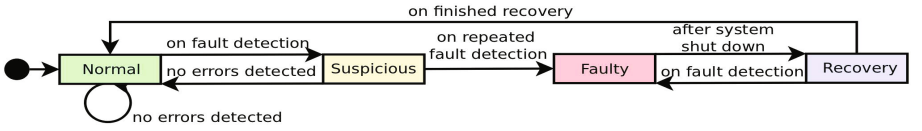


Fig. 3. Fault tolerant system modes

recovery mode, in which the system tries to return into a state from which it could start performing its normal operation.

The fault-free model with only *normal* FMS mode is easy to extend with fault tolerance simply introducing FMS and related controller events for the other three modes.

In the fault tolerant model, the environment becomes unrestricted, thus additional events are introduced for each environment event and directly related (sensing) events from the fault-tolerant model. These newly introduced events are expected to cause system fault and a deviation from a normal operation. To ensure this, events have guard block as $\neg G_u \vee \neg G_t$.

For all such relevant events $(G_u \wedge G_t)_{fault-free}$, we have fault-handling events covering $(\neg G_u \vee \neg G_t)_{fault-tolerant}$, ensuring the system is always enabled.

3 Conclusions

The case study resulted in a fully modeled sluice gate model according to our interpretation of the [1] requirements, and provided insights into possible general guidelines for control systems modelling including the problematic areas of time and fault tolerance. The described ideas and guidelines are meant for easier integration of different approaches and prevention of possible hidden difficulties. Further aim is to perform more case studies to prove pattern's feasibility. Both guards and $(\neg G_u \vee \neg G_t)$ and TIL are systematically defined, thus could be generated by tools in future. The time layer needs to be improved to make it less coupled as now it uses event sequence information from previous layers.

References

1. Hayes, I.J., Jackson, M.A., Jones, C.B.: Determining the Specification of a Control System Fromthat of Its Environment. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 154–169. Springer, Heidelberg (2003)
2. Michael, B.: Towards a Cookbook for Modelling and Refinement of Control Problems (2009)
3. Ilic, D., Troubitsyna, E., Laibinis, L., Snook, C.: Formal Development of Mechanisms for Tolerating Transient Faults (2006)
4. Sarshogh, M.R., Butler, M.: Specification and Refinement of Discrete Timing Properties in Event-B
5. Cansell, D., Mry, D., Rehm, J.: Time Constraint Patterns for Event B Development (2006)

Author Index