

Luís Ferreira Pires
Slimane Hammoudi
Bran Selic (Eds.)

Communications in Computer and Information Science

880

Model-Driven Engineering and Software Development

5th International Conference, MODELSWARD 2017
Porto, Portugal, February 19–21, 2017
Revised Selected Papers

Communications in Computer and Information Science

880

Commenced Publication in 2007

Founding and Former Series Editors:

Phoebe Chen, Alfredo Cuzzocrea, Xiaoyong Du, Orhun Kara, Ting Liu,
Dominik Ślęzak, and Xiaokang Yang

Editorial Board

Simone Diniz Junqueira Barbosa

*Pontifical Catholic University of Rio de Janeiro (PUC-Rio),
Rio de Janeiro, Brazil*

Joaquim Filipe

Polytechnic Institute of Setúbal, Setúbal, Portugal

Igor Kotenko

*St. Petersburg Institute for Informatics and Automation of the Russian
Academy of Sciences, St. Petersburg, Russia*

Krishna M. Sivalingam

Indian Institute of Technology Madras, Chennai, India

Takashi Washio

Osaka University, Osaka, Japan

Junsong Yuan

University at Buffalo, The State University of New York, Buffalo, USA

Lizhu Zhou

Tsinghua University, Beijing, China

More information about this series at <http://www.springer.com/series/7899>

Luís Ferreira Pires · Slimane Hammoudi
Bran Selic (Eds.)

Model-Driven Engineering and Software Development

5th International Conference, MODELSWARD 2017
Porto, Portugal, February 19–21, 2017
Revised Selected Papers

Editors

Luís Ferreira Pires
University of Twente
Enschede
The Netherlands

Bran Selic
Malina Software Corp.
Nepean, ON
Canada

Slimane Hammoudi
Siège du Groupe ESEO
Angers
France

ISSN 1865-0929 ISSN 1865-0937 (electronic)
Communications in Computer and Information Science
ISBN 978-3-319-94763-1 ISBN 978-3-319-94764-8 (eBook)
<https://doi.org/10.1007/978-3-319-94764-8>

Library of Congress Control Number: 2018947447

© Springer International Publishing AG, part of Springer Nature 2018

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors, and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Printed on acid-free paper

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

Preface

The present volume contains extended versions of a set of selected papers from the 5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2017), held in Porto, Portugal, during February 19–21, 2017.

These papers were selected by the event chairs and their selection is based on a number of criteria that include the classifications and comments provided by the Program Committee members, the session chairs' assessment, as well as the program chairs' overview of all papers in the technical program. The authors of selected papers were then invited to submit a revised and extended version of their papers having at least 30% additional new material.

The purpose of the International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2017, was to provide a platform for researchers, engineers, academics as well as industrial professionals from all over the world to present their research results and development activities in using models and model-driven engineering techniques for software development. Model-driven development (MDD) is an approach to the development of IT systems in which models take a central role, not only for purposes of analysis and documentation but also for their construction. MDD has emerged from a number of modeling initiatives, most prominently the model-driven architecture (MDA) adopted by the Object Management Group (OMG).

The papers selected to be included in this book contribute to the development of highly relevant research trends in model-driven engineering and software development, including:

- Methodologies for MDD development and exploitation
- Model-based testing
- Model simulation
- Domain-specific modeling
- Code generation from models
- New MDD tools
- Multi-model management
- Model evolution
- Industrial applications of model-based methods and technologies

We would like to thank all the authors for their contributions and also the reviewers who helped ensure the quality of this publication.

February 2017

Luis Ferreira Pires
Slimane Hammoudi
Bran Selic

Organization

Conference Chair

Bran Selic

Malina Software Corp., Canada

Program Co-chairs

Luis Ferreira Pires

University of Twente, The Netherlands

Slimane Hammoudi

ESEO, MODESTE, France

Program Committee

Silvia Abrahão

Universitat Politecnica de Valencia, Spain

Guglielmo De Angelis

CNR, IASI, Italy

Keijiro Araki

Kyushu University, Japan

Marco Autili

University of L'Aquila, Italy

Omar Badreddin

University of Texas El Paso, USA

Mira Balaban

Ben-Gurion University of the Negev, Israel

Daniel Balasubramanian

Vanderbilt University, USA

Bernhard Bauer

University of Augsburg, Germany

Martin Becker

Fraunhofer IESE, Germany

Luca Berardinelli

Vienna University of Technology, Austria

Antonia Bertolino

Italian National Research Council, CNR, Italy

Lorenzo Bettini

Università di Firenze, Italy

Paolo Bocciarelli

University of Rome Tor Vergata, Italy

Jan Bosch

Chalmers University of Technology, Sweden

Marco Brambilla

Politecnico di Milano, Italy

Mark van den Brand

Eindhoven University of Technology, The Netherlands

Antonio Brogi

Università di Pisa, Italy

Achim D. Brucker

SAP Research, Germany

Philipp Brune

University of Applied Sciences Neu-Ulm, Germany

Christian Bunse

University of Applied Sciences Stralsund, Germany

Dumitru Burdescu

University of Craiova, Romania

Olena Chebanyuk

National Aviation University, Ukraine

Dickson Chiu

The University of Hong Kong, Hong Kong, SAR

China

Antonio Cicchetti

Malardalen University, Sweden

Bernard Coulette

Université Toulouse Jean Jaurès, France

Kevin Daimi

University of Detroit Mercy, USA

Andrea D'Ambrogio

Università di Roma Tor Vergata, Italy

Birgit Demuth

TU Dresden, Germany

Enrico Denti	Università di Bologna, Italy
Zinovy Diskin	McMaster University and University of Waterloo, Canada
Dimitris Dranidis	CITY College, International Faculty of the University of Sheffield, Greece
Schahram Dustdar	Vienna University of Technology, Austria
Sophie Ebersold	IRIT, France
Holger Eichelberger	Universität Hildesheim, Germany
Maria Jose Escalona	University of Seville, Spain
Rik Eshuis	Eindhoven University of Technology, The Netherlands
Angelina Espinoza	Universidad Autónoma Metropolitana, Iztapalapa, Spain
Vladimir Estivill-Castro	Griffith University, Australia
Anne Etien	University of Lille 1, Inria, CNRS, France
Dirk Fahland	Eindhoven University of Technology, The Netherlands
João Faria	University of Porto, Portugal
Bernd Fischer	Stellenbosch University, South Africa
Stephan Flake	S&N CQM Consulting & Services GmbH, Germany
Francois Fouquet	University of Luxembourg, Luxembourg
Piero Fraternali	Politecnico di Milano, Italy
Jicheng Fu	University of Central Oklahoma, USA
Kurt Geihs	University of Kassel, Germany
Sébastien Gérard	CEA, France
Paola Giannini	University of Piemonte Orientale, Italy
Stefania Gnesi	CNR, Italy
Cesar Gonzalez-Perez	Institute of Heritage Sciences, Spanish National Research Council, Spain
Carmine Gravino	University of Salerno, Italy
Slimane Hammoudi	ESEO, MODESTE, France
Klaus Havelund	Nasa/Jet Propulsion Laboratory, USA
Jose R. Hilera	University of Alcalá, Spain
Pavel Hruby	DXC Technology, Denmark
Marianne Huchard	Université de Montpellier, France
Emilio Insfran	Universitat Politècnica de València, Spain
Stefan Jablonski	University of Bayreuth, Germany
Slinger Jansen	Utrecht University, The Netherlands
Ricardo Jardim-Gonçalves	Universidade Nova de Lisboa, Portugal
George Kakarontzas	Technological Educational Institute of Thessaly, Greece
Teemu Kanstren	VTT, Finland
Jun Kong	North Dakota State University, USA
Jochen Kuester	University of Applied Sciences in Bielefeld, Germany
Uirá Kulesza	Federal University of Rio Grande do Norte, Brazil
Anna-Lena Lamprecht	Utrecht University, The Netherlands
Kun Chang Lee	Sungkyunkwan University, South Korea

Claudia Linnhoff-Popien	Ludwig-Maximilians-Universität Munich, Germany
Francesca Lonetti	National Research Council Pisa, Italy
Roberto Lopez-Herrejon	École de Technologie Supérieure, Canada
David Lorenz	Open University, Israel
Der-Chyuan Lou	Chang Gung University, Taiwan
Frederic Mallet	Université Nice Sophia Antipolis, France
Eda Marchetti	ISTI-CNR, Italy
Beatriz Marin	Universidad Diego Portales, Chile
Steve McKeever	Uppsala University, Sweden
Dragan Milicev	University of Belgrade, Serbia
Dugki Min	Konkuk University, South Korea
Valérie Monfort	LAMIH Valenciennes UMR CNRS 8201, France
Sascha Mueller-Feuerstein	Ansbach University of Applied Sciences, Germany
Halit Oguztüzün	Middle East Technical University, Turkey
Olaf Owe	University of Oslo, Norway
Rob Pettit	The Aerospace Corp., USA
Luis Ferreira Pires	University of Twente, The Netherlands
Elke Pulvermueller	University of Osnabrück, Germany
Gil Regev	Ecole Polytechnique Fédérale de Lausanne, Switzerland
Iris Reinhartz-Berger	University of Haifa, Israel
Wolfgang Reisig	Humboldt-Universität zu Berlin, Germany
Werner Retschitzegger	Johannes Kepler University, Austria
Colette Rolland	Université de Paris 1 Panthéon Sorbonne, France
Jose Raul Romero	University of Cordoba, Spain
Gustavo Rossi	Lifia, Argentina
Motoshi Saeki	Tokyo Institute of Technology, Japan
Francesca Saglietti	University of Erlangen-Nuremberg, Germany
Comai Sara	Politecnico di Milano, Italy
Klaus Schmid	University of Hildesheim, Germany
Jean-Guy Schneider	Swinburne University of Technology, Australia
Wieland Schwinger	Johannes Kepler University, Austria
Bran Selic	Malina Software Corp., Canada
Peter Sestoft	IT University of Copenhagen, Denmark
Marten van Sinderen	University of Twente, The Netherlands
Pnina Soffer	University of Haifa, Israel
Arnor Solberg	Sintef, Norway
Stéphane Somé	University of Ottawa, Canada
Jean-Sébastien Sottet	Luxembourg Institute for Science and Technology, Luxembourg
Alin Stefanescu	University of Bucharest, Romania
Arnon Sturm	Ben-Gurion University of the Negev, Israel
Hiroki Suguri	Miyagi University, Japan
Eugene Syriani	University of Montreal, Canada
Massimo Tivoli	University of L'Aquila, Italy
Mario Trapp	Fraunhofer IESE, Germany

Naoyasu Ubayashi	Kyushu University, Japan
Andreas Ulrich	Siemens AG, Germany
Gianluigi Viscusi	EPFL Lausanne, Switzerland
Shuai Wang	Simula Research Lab, Norway
Christiane Gresse von Wangenheim	Federal University of Santa Catarina, Brazil
Layne Watson	Virginia Polytechnic Institute and State University, USA
Jan Martijn van der Werf	Universiteit Utrecht, The Netherlands
Michael Whalen	University of Minnesota, USA
Franz Wotawa	Graz University of Technology, Austria
Husnu Yenigun	Sabanci University, Turkey
Gefei Zhang	Hochschule für Technik und Wirtschaft Berlin, Germany
Heming Zhang	Tsinghua University, China
Tian Zhang	Nanjing University, China
Chunying Zhao	Western Illinois University, USA
Haiyan Zhao	Peking University, China
Kamil Zyla	Lublin University of Technology, Poland

Additional Reviewers

Zakia Alkadri	University of Texas at El Paso, USA
Shinpei Hayashi	Tokyo Institute of Technology, Japan
Alexander Jahl	Kassel University, Germany
Giorgio Oronzo Spagnolo	ISTI CNR ITALY, Italy

Invited Speakers

Uwe Assmann	Technische Universität Dresden, Germany
Juan de Lara	Universidad Autónoma de Madrid, Spain
Frédéric Benaben	Ecole de Mines Albi-Carmaux, France

Contents

SOMMELIER: A Tool for Validating TOSCA Application Topologies	1
<i>Antonio Brogi, Antonio Di Tommaso, and Jacopo Soldani</i>	
Evaluation of XIS-Reverse, a Model-Driven Reverse Engineering Approach for Legacy Information Systems	23
<i>André Reis and Alberto Rodrigues da Silva</i>	
Formal and Virtual Multi-level Design Space Exploration	47
<i>Letitia W. Li, Daniela Genius, and Ludovic Apvrille</i>	
Automated Synthesis of a Real-Time Scheduling for Cyber-Physical Multi-core Systems	72
<i>Johannes Geismann, Robert Höttger, Lukas Krawczyk, Uwe Pohlmann, and David Schmelter</i>	
A Model Based Approach for Complex Dynamic Decision-Making.	94
<i>Souvik Barat, Vinay Kulkarni, Tony Clark, and Balbir Barn</i>	
Deterministic High-Level Executable Models Allowing Efficient Runtime Verification	119
<i>Vladimir Estivill-Castro and René Hexel</i>	
A Consistency-Preserving Editing Model for Dynamic Filtered Engineering of Model-Driven Product Lines.	145
<i>Felix Schwägerl and Bernhard Westfechtel</i>	
Model-Driven STEP Application Protocol Extensions Combined with Feature Modeling Considering Geometrical Information	173
<i>Thorsten Koch, Jörg Holtmann, and Timo Lindemann</i>	
A Model Driven Engineering Approach for Heterogeneous Model Composition.	198
<i>Fazle Rabbi, Yngve Lamo, and Lars Michael Kristensen</i>	
Generative versus Interpretive Model-Driven Development: Moving Past ‘It Depends’	222
<i>Michiel Overeem, Slinger Jansen, and Sven Fortuin</i>	
Applying Integrated Domain-Specific Modeling for Multi-concerns Development of Complex Systems	247
<i>Reinhard Pröll, Adrian Rumpold, and Bernhard Bauer</i>	

A Domain-Specific Modeling Approach for Testing Environment Emulation	272
<i>Jian Liu, John Grundy, Mohamed Abdelrazek, and Iman Avazpour</i>	
A Framework for UML-Based Component-Based Design and Code Generation for Reactive Systems.	300
<i>Van Cam Pham, Ansgar Radermacher, Sébastien Gérard, and Shuai Li</i>	
Automatic UI Generation for Aggregated Linked Data Applications by Using Sharable Application Ontologies	328
<i>Michael Hitz, Thomas Kessel, and Dennis Pfisterer</i>	
Surveying Co-evolution in Modeling Ecosystems	354
<i>Jürgen Ettlstorfer, Elisabeth Kapsammer, Wieland Schwinger, and Johannes Schönböck</i>	
Functional Decomposition for Software Architecture Evolution	377
<i>David Faitelson, Robert Heinrich, and Shmuel Tyszberowicz</i>	
Model-Driven Approach to Handle Evolutions of OLAP Requirements and Data Source Model	401
<i>Said Taktak, Jamel Feki, Abdulrahman Altalhi, and Gilles Zurfluh</i>	
Complex Event Processing for User-Centric Management of IoT Systems . . .	426
<i>Moussa Amrani, Fabian Gilson, and Vincent Englebert</i>	
Efficient Distributed Execution of Multi-component Scenario-Based Models	449
<i>Shlomi Steinberg, Joel Greenyer, Daniel Gritzner, David Harel, Guy Katz, and Assaf Marron</i>	
Modelling the World of a Smart Room for Robotic Co-working	484
<i>Uwe Aßmann, Christian Piechnick, Georg Püschel, Maria Piechnick, Jan Falkenberg, and Sebastian Werner</i>	
Author Index	507



SOMMELIER: A Tool for Validating TOSCA Application Topologies

Antonio Brogi, Antonio Di Tommaso, and Jacopo Soldani^(✉)

Department of Computer Science, University of Pisa, Pisa, Italy
soldani@di.unipi.it

Abstract. TOSCA is an OASIS standard for specifying cloud applications and automating their management. The topology of a cloud application can be described as a typed and directed graph. The latter can then be automatically processed by so-called TOSCA engines to automate the deployment and management of the described application on cloud platforms. In this paper we first illustrate the conditions ensuring the validity of a TOSCA application topology. We then introduce SOMMELIER, an open-source validator of TOSCA application topologies based on such validity conditions.

1 Introduction

Cloud computing is nowadays characterised by a lack of standardisation, with different cloud platforms providing similar offerings in different and heterogeneous ways [1]. As a result, cloud developers tend to remain locked-in a specific platform environment because it is practically unfeasible for them, due to high complexity and cost, to migrate their applications to a different platform. According to [2], to enable the creation of portable cloud applications, the application components, their relations and management should be modelled in a standardised, machine-readable format. This would also allow the automation of the deployment and management of modelled applications [3].

In this scenario, OASIS released the *Topology and Orchestration Specification for Cloud Applications* (TOSCA [4]). TOSCA permits specifying portable cloud applications and automating their management. The structure of a cloud application can be described as a typed and directed topology graph (specified in a standardised, YAML-based modelling language). In a topology graph, the nodes model the components of an application (e.g., a web application, an application server, a NoSQL database, and a NoSQL DBMS), while the edges model the relationships occurring between such components (e.g., the web application runs on the application server and it connects to the NoSQL database, and the NoSQL database is installed on the NoSQL DBMS).

TOSCA applications can then be declaratively processed by so-called TOSCA engines to automate their deployment [5]. Such a declarative processing depends on the inter-node relationships specified in the topology of an application [6]. Initially, (i) all nodes without dependencies on other nodes are deployed.

Then, (ii) the nodes whose requirements are actually satisfied (by capabilities offered by the nodes that have been deployed) are deployed, and their outgoing relationships are properly processed. Step (ii) is repeated until all the nodes in the application topology have been deployed [7].

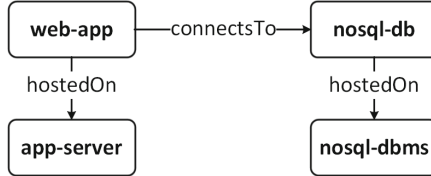


Fig. 1. A toy example of application topology

Consider, for instance, the application topology in Fig. 1, whose nodes are a web application, an application server, a NoSQL database and a NoSQL DBMS. The inter-node relationships indicate that the web application must be hosted on the application server and connected to the NoSQL database. They also indicate that the NoSQL database must be hosted on the NoSQL DBMS. Step (i) would result in first deploying the application server and the NoSQL DBMS. Step (ii) would then result in deploying the NoSQL database on the NoSQL DBMS. Step (ii) would then be repeated, and it would result in deploying the web application on the application server, and in setting up a connection from the web application to the NoSQL database.

The above (toy) example demonstrates how the declarative processing of TOSCA applications heavily depends on the inter-node relationships indicated in the topology of an application. We must also consider that the actual processing of each node and relationship relies on configuration information contained in its specification, which is put into context by indicating constraints on how it can be interconnected with other nodes and relationships [7]. This makes it crucial to ensure, at design-time, that TOSCA application topologies are valid, by also checking that all the relationships interconnecting the nodes in an application topology have been set properly.

This paper aims at providing a first design-time support for TOSCA application developers, by allowing them to validate their application topologies. In this perspective, the main contributions in this paper are twofold:

- We formalise the conditions that must hold to have valid TOSCA application topologies, by systematically mapping the interconnection constraints that can be specified in TOSCA into formal conditions that must hold to ensure the validity of a TOSCA application topology.
- We propose a first prototype of validator for TOSCA application topologies, called SOMMELIER. SOMMELIER checks whether the topology of a TOSCA application satisfies all interconnection constraints, by actually checking all the proposed validity conditions.

We believe that SOMMELIER can fruitfully help TOSCA application developers, as it allows them to automatically validate their TOSCA application topologies at design time (a task that they currently have to perform manually). Also, as SOMMELIER fully integrates with the OpenStack TOSCA parser [8], and since both SOMMELIER and the OpenStack TOSCA parser are open-source, they can lay the foundations for an open-source toolset for supporting TOSCA application developers from the design time till the run time [9].

This paper is an extended version of [10], which further motivates the need for a design-time support for validating of TOSCA application, and which includes an extended description of the prototype and of its testing.

The rest of this paper is organised as follows. Section 2 provides background on TOSCA and a motivating example further highlighting the need for validating TOSCA application topologies. Section 3 illustrates the formal conditions allowing to validate the topology of a TOSCA application. Sect. 4 provides a detailed description of SOMMELIER, by also showing its testing and how it permits validating TOSCA application topologies. Finally, Sects. 5 and 6 discuss related work and draw some concluding remarks, respectively.

2 Background and Motivations

2.1 Background: TOSCA

The OASIS standard TOSCA [4] aims at enabling the specification of portable cloud application and at automating their deployment and management. Cloud applications can be specified in a YAML-based, machine-readable modelling language. Obtained specifications can then be declaratively processed by TOSCA engines, which can automatically deploy and manage specified applications.

In TOSCA, a cloud application is specified as a `service_template`, which is in turn composed by a `topology_template`, and by the types needed to build such a `topology_template` (Fig. 2). The `topology_template` is a typed directed graph that describes the topology of a cloud application, viz., its structure. Its nodes (called `node_templates`) model the components of an application, while its edges (called `relationship_templates`) model the inter-component relationships.

The `node_templates` and `relationship_templates` are typed by means of `node_types` and `relationship_types`, respectively. A node type defines the `requirements` of a component, the `capabilities` that it can offer to satisfy the requirements of other components, its observable `attributes` and `properties`, and the `interfaces` through which it offers its management operations. Capabilities are also typed through so-called `capability_types`, which permit indicating their `attributes`, `properties` and `valid_source_types` (viz., the node types that can be satisfied by such capabilities).

A relationship type instead describes the `attributes` and `properties` of an inter-component relationship, as well as the `interfaces` through which it offers its management operations. A relationship type can also indicate constraints on the capability types that can be targeted by such type of relationships (through its clause `valid_target_types`).

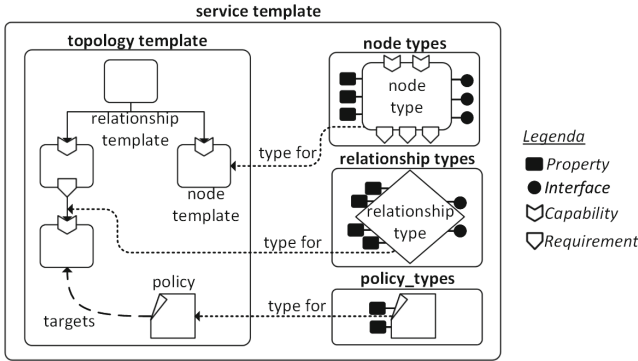


Fig. 2. TOSCA `service_template` [4].

It is worth noting that the TOSCA type system supports inheritance. A node type can extend another, to inherit all its attributes, properties, requirements, interfaces, and operations. Analogously, a relationship type or a capability type can extend another to inherit all its features.

TOSCA application specifications are given in `.tosca` document, which has then to be packaged together with all the installable and executable files needed to properly instantiate the specified applications. To enable this, TOSCA also prescribes the format (called CSAR—*Cloud Service ARchive*) to archive application specifications together with all such files.¹

2.2 Motivating Example

Consider the (toy) web-based application in Fig. 3 (modelled in TOSCA according to the Winery graphical notation [11]). The application is composed by three main components, namely a web-based GUI, a REST API and a Database, which are hosted on a WebServer, a Server and a DBMS, respectively. The GUI depends on the availability of the REST API to effectively work, and the REST API in turn depends on the availability of the Database.

Despite the application specification in Fig. 3 seems valid, there are two wrong inter-node relationships. Firstly, the REST API explicitly states that it requires a connection to the back-end Database, but the requirement connection is satisfied by a relationships of type `DependsOn`. The latter will be processed by TOSCA-compliant cloud platforms by only postponing the installation of the REST API after that of the Database. No connection from the REST API to the Database will be set up (even if the REST API explicitly requires it).

Also, the Database in our motivating application is NoSQL, and it `HostedOn` a MariaDB² DBMS. The latter is however a SQL-based relational DBMS, which is hence not capable of managing NoSQL databases.

¹ A more detailed, self-contained introduction to TOSCA can be found in [9].

² <https://mariadb.com/>.

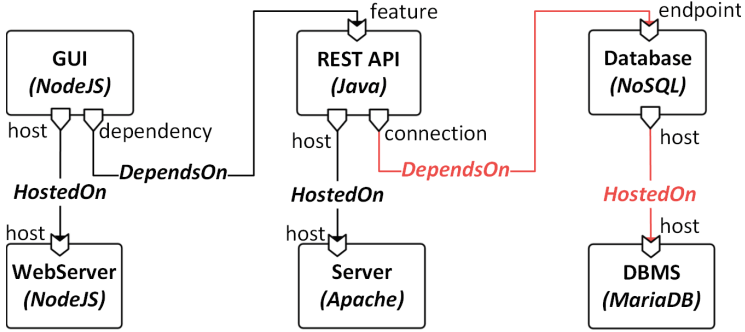


Fig. 3. Motivating example.

The above errors are hard to be manually detected, as we need to manually check that all inter-node relationships in a TOSCA application topology satisfy the interconnection constraints specified in (the types of) the source and target nodes, as well as those specified in (the type of) the employed relationships. TOSCA application developers should hence be provided with solutions for automatically validating TOSCA application topologies.

3 Validating TOSCA Application Topologies

The aim of this paper is to provide a first design-time support for TOSCA application developers, which allows them to validate the topology of an application. The topology of an application is given in the form of a `topology_template`, viz., a typed directed graph whose nodes represent the components of an application, and whose edges model the relationships occurring among such components [4]. Each relationship specifies that a requirement of the source node must be actually satisfied by (a capability of) the target node.

To check whether a TOSCA application topology is valid, we must verify that all inter-component relationships are properly settled. This in turn means that we must check all TOSCA elements forming a relationship, namely its source (viz., a requirement of a node), the relationship itself (viz., a relationship template), and its target (viz., a node or a capability of a node).

In the following, we show how to *systematically map*³ the interconnection constraints that can be specified in TOSCA to formal conditions. We first present

³ We first systematically read the TOSCA specification [4], and we excerpted all portions that describe how to specify interconnection constraints. In the following, we shall recall all such portions, and we illustrate how to directly map each of them to a formal condition that must be verified to ensure the validity of a TOSCA application topology.

the validation conditions for sources of relationships (Sect. 3.1), those for relationships themselves (Sect. 3.2), and those for targets of relationships (Sect. 3.3). We finally compose all such conditions to provide a notion of validity for application topologies (Sect. 3.4).

3.1 Validating Sources of Relationships

The source of a relationship is a requirement of a node [4]. We must hence verify that all the relationships outgoing from a requirement do not violate any constraint indicated in the requirement definition.

In this section, we first recall how requirements can be defined in TOSCA, by also explaining what the meaning of a requirement definition is (according to the TOSCA specification [4]). We then single out the formal conditions that must be verified to ensure that the interconnection constraints indicated in a requirement definition are all satisfied.

How to *define* a Requirement in TOSCA

A node type n_{type} defines the set of named requirements that can be exposed by a node template of such type. Requirements can be defined within the `requirements` of n_{type} with the grammars displayed in Fig. 4. Both grammars permit specifying the requirement name (`req_name`) and some constraints that must be fulfilled to actually satisfy the requirement under definition.

```
req_name: cap_type_name
```

(a)

```
req_name:
  capability: cap_type_name
  node: node_type_name
  relationship: rel_type_name
  occurrences: [ min_occ, max_occ ]
```

(b)

Fig. 4. (a) Simple and (b) extended grammars for *requirement definitions* [4].

The simple grammar (a) requires to indicate the name of a valid capability type that must be used to fulfill the requirement under definition. The extended grammar (b) allows to specify three additional (optional) constraints.

- `node` allows to indicate a node type that contains a capability definition that can be used to fulfill the requirement.

- **relationship** allows to indicate a relationship type that can be exploited to create an outgoing relationship template to fulfill the requirement.
- **occurrences** allows to indicate the minimum and maximum occurrences of the requirement in node templates, whose node type is that under definition.⁴

A node template n_{temp} is an instance of a node type n_{type} in a TOSCA application topology, which allows to indicate the requirements that are actually needed by the application component modelled by n_{temp} . Such requirements can be indicated within the **requirements** clause of n_{temp} , through so-called *requirement assignments*. Each requirement assignment instantiates a corresponding⁵ requirement definition in n_{type} . Figure 5 displays the grammars for requirement assignments in TOSCA.

```
req_name: node_temp_name
```

(a)

```
req_name:
node: node_temp_name | node_type_name
relationship: rel_temp_name | rel_type_name
capability: cap_name | cap_type_name
node_filter: node_filter_definition
```

(b)

Fig. 5. (a) Simple and (b) extended grammars for *requirement assignments* [4].

The simple grammar (a) only allows to indicate (the name of) the node template satisfying the requirement under assignment. This notation is only valid if the corresponding requirement definition (in the node type of the node template that is being specified) indicates at least a valid capability type that can be found in the target node template.

The extended grammar (b) not only allows to indicate (the name of) the target node template, but also to specify some additional information.

- **node** allows to indicate the target node. It can be used to provide either the name of the node template that is actually fulfilling the requirement under assignment, or the name of a node type that constrains the type of nodes that can be used to dynamically satisfy the requirement at run-time.

⁴ Since the focus of this paper is on validating inter-component dependencies in TOSCA application topologies, and since **occurrences** is not giving any constraint concerning inter-component dependencies, we shall not formalise the trivial condition to validate **occurrences**.

⁵ A node template’s requirement assignment corresponds to a node type’s requirement definitions if they have the same name **req_name**.

- **relationship** is optional, and it allows to indicate the name of a relationship template (to relate the source node to the—capability in the—target node when fulfilling the requirement), or the name of a relationship type (that constrains the type of relationships that can be used to dynamically settle a relationship between the source and target nodes at run-time).
- **node_filter** is also optional, and it allows to indicate additional constraints on the node/relationship that can be used to dynamically satisfy the requirement under assignment at run-time.

How to *validate* a TOSCA Requirement

We hereby single out the conditions that must hold to validate sources of relationships at design-time.⁶ In doing so, we exploit some shorthand notation.

Notation. *We shall write:*

- $\text{type}(\cdot)$ and $\text{name}(\cdot)$ to denote the type and name of a TOSCA element,
- $t' \geq t$ to denote that t' extends⁷ or is equal to t ,
- $e.f$ to denote the field f of the TOSCA element e (which is \perp if f is not defined in e), and
- $C(\cdot)$ and $R(\cdot)$ to denote the capabilities and the requirements defined in a node type or assigned by a node template.

Given a TOSCA application topology, the sources of its inter-component relationships are valid if each of the requirements of its node templates is associated with a node and a relationship satisfying all constraints indicated in the definition and assignment of such requirement.

Condition 1. *Let n_{temp} be a node template of type n_{type} . Then: $\forall r_a \in R(n_{temp}), \exists r_d \in R(n_{type})$:*

- (1) $\text{name}(r_a) = \text{name}(r_d) \wedge$
- (2) $r_d.\text{node} \neq \perp \Rightarrow \text{type}(r_a.\text{node}) \geq r_d.\text{node} \wedge$
- (3) $r_a.\text{capability} \neq \perp \Rightarrow \text{type}(r_a.\text{capability}) \geq r_d.\text{capability} \wedge$
- (4) $r_a.\text{capability} = \perp \Rightarrow \exists c \in C(r_a.\text{node}) : \text{type}(c) \geq r_d.\text{capability} \wedge$
- (5) $r_d.\text{relationship} \neq \perp \wedge r_a.\text{relationship} \neq \perp \Rightarrow$
 $\text{type}(r_a.\text{relationship}) \geq r_d.\text{relationship}$

The first check (Condition 1.1) ensures that, for each requirement assigned r_a in a node template n_{temp} (of type n_{type}), there exists a corresponding requirement definition r_d in n_{type} .

We can then check that no requirement assignment r_a is violating the constraints indicated by the corresponding requirement definition r_d :

⁶ Since we focus on design-time, we shall not consider all constraints on how to automatically complete the topology of a TOSCA application (viz., those constraining the types of node and relationship that can be used to automatically fulfill a requirement at run-time). Anyway, our approach can help driving the automatic completion of TOSCA application topologies, as well as to double-check that automatically completed topologies are valid.

⁷ Given that t and t' are TOSCA types, t' extends t if t' is (directly or indirectly) derived from t [4].

- A requirement definition r_d may indicate the node type that can be used to validly satisfy a requirement (through the optional field `node`). Hence, if `node` is specified in r_d , the type of the node template targeted by each corresponding requirement assignment r_a has to extend or to be equal to that indicated in `node` (Condition 1.2.)
- Each requirement definition specifies the name of a capability type that can be used to validly satisfy the requirement. This is ensured if each corresponding requirement assignment r_a directly targets a capability whose type extends or is equal to that indicated in the requirement definition (Condition 1.3). When a requirement assignment is instead only indicating the target node template (without indicating any of its capabilities), we must check whether such node template is offering at least one type-compatible capability (Condition 1.4).
- A requirement definition r_d may also indicate the relationship type that can be validly exploited to settle a relationship template outgoing from the requirement (through the optional field `relationship`). Hence, if `relationship` is specified in r_d , the type of a relationship template outgoing from a corresponding requirement assignment r_a (if any) extends or is equal to that indicated in `relationship` (Condition 1.5).

3.2 Validating Relationships

Inter-component relationships are indicated as typed relationship templates [4]. We must hence verify that the relationship templates in a TOSCA application topology are instantiating the corresponding relationship types without violating any of their interconnection requirements.

In this section, we first recall how to specify relationship types in TOSCA, by also explaining what the meaning of a relationship type definition is (according to the TOSCA specification [4]). We then single out the formal conditions that must be verified to ensure that no relationship template is violating the constraints given by the corresponding relationship type.

How to *define* a Relationship in TOSCA

TOSCA allows to define a relationship type with the grammar shown in Fig. 6. The latter permits indicating the name of the relationship type under definition (`rel_type_name`), and its features and interconnection constraints.

- `derived_from` is optional, and it allows to indicate (the name of) a parent relationship type⁸. If indicated, the relationship type under definition inherits all the features and constraints of the parent relationship type, and it can override some of them [7]. For instance, if the relationship type under definition does not specify a new list of `valid_target_types`, then it takes that of the parent relationship type. Otherwise, the parent's list of `valid_target_types` is overridden by that specified in the relationship type under definition.

⁸ All TOSCA relationship types should be derived (directly or indirectly) from the `tosca.relationships.Root` relationship type [4].

```

rel_type_name:
  derived_from: parent_rel_type_name
  version: version_number
  description: rel_description
  properties: property_definitions
  attributes: attribute_definitions
  interfaces: interface_definitions
  valid_target_types: [ cap_type_names ]

```

Fig. 6. Grammar for *relationship types* [4].

- **version** and **description** are optional, and they permit versioning and describing (in natural language) a relationship type.
- **properties** and **attributes** are also optional, and they allow to specify the desired and actual state of a relationship, respectively.
- **interfaces** is optional, and it allows to indicate the management operations that can be offered by relationship templates whose relationship type is that under definition.
- **valid_target_types** is optional, and it allows to list the capability types that can be validly used as targets of relationship templates whose relationship type is that under definition.

How to *validate* a TOSCA Relationship

We hereby single out the conditions ensuring that a relationship template is not violating any of the interconnection constraints indicated by the corresponding relationship type. In this perspective, it is worth highlighting that the only interconnection constraints that can be specified while defining a relationship type are those concerning its **valid_target_types** (which can be indicated inline or inherited from the parent relationship type).

Notation. We shall denote with $\mathbb{T}(\cdot)$ the set of capability types that are valid targets for a relationship type. Given a relationship type rel_{type} :

- If $rel_{type}.valid_target_types \neq \perp$, then $\mathbb{T}(rel_{type})$ is the set containing all types in $rel_{type}.valid_target_types$,
- otherwise, if $rel_{type}.derived_from \neq \perp$, then $\mathbb{T}(rel_{type})$ is the set containing all types in $\mathbb{T}(rel_{type}.derived_from)$,
- otherwise, $\mathbb{T}(rel_{type})$ is the set containing all capability types (meaning that all capability types are valid targets for rel_{type}).

The relationship templates instantiated in a TOSCA application topology are valid if all their targets are valid, viz., the targets do not violate the constraints indicated in the corresponding relationship types.

Condition 2. Let rel_{temp} be a relationship template of type rel_{type} . If there exists a requirement assignment r_a that is source of rel_{temp} , then:

- (1) $r_a.\text{capability} \neq \perp \Rightarrow$
 $\exists c_{type} \in \mathbb{T}(\text{rel}_{type}) : \text{type}(r_a.\text{capability}) \geq c_{type} \wedge$
- (2) $r_a.\text{capability} = \perp \Rightarrow$
 $\exists c_a \in \mathbb{C}(r_a.\text{node}) : \exists c_{type} \in \mathbb{T}(\text{rel}_{type}) : \text{type}(c_a) \geq c_{type}$

The above conditions ensure that, if a relationship template targets a specific capability⁹, then such capability must be type-compatible with at least one of those indicated in `valid_target_types` (Condition 2.1). If a relationship template instead targets a node template (but not concretely pointing to any of its capabilities), then such node template must offer at least one type-compatible capability (Condition 2.2).

3.3 Validating Targets of Relationships

The target of a relationship can be either a capability or a node template. In the latter case, the targeted node template must offer at least a capability that can satisfy the source requirement [4].

In this section, we first recall how to specify capabilities in TOSCA, by also explaining what their meaning is (according to the TOSCA specification [4]). We then single out the formal conditions that must be verified to ensure that no relationship is violating the constraints given by its target.

How to *define* a Capability in TOSCA

Capability types can be defined in TOSCA according to the grammar shown in Fig. 7. The latter permits indicating the name of the capability type under definition (`cap_type_name`), and its features and interconnection constraints.

```
cap_type_name:
  derived_from: parent_cap_type_name
  version: version_number
  description: capability_description
  properties: property_definitions
  attributes: attribute_definitions
  valid_source_types: [ node_type_names ]
```

Fig. 7. Grammar for *capability types* [4].

- `derived_from` is optional, and it allows to indicate (the name of) a parent capability type¹⁰. If indicated, the relationship type under definition inherits

⁹ Please recall that a relationship is outgoing from a requirement assignment r_a . The latter can either indicate the specific capability satisfying r_a , or a node template offering (at least) a capability satisfying r_a (see Fig. 5).

¹⁰ All TOSCA capability types should be derived (directly or indirectly) from the `tosca.capability.Root` capability type [4].

all the features and constraints of the parent relationship type, and it can override some of them [7]. For instance, if the capability type under definition does not specify a new list of `valid_source_types`, then it takes that of the parent capability type. Otherwise, the parent's list of `valid_source_types` is overridden by that specified in the capability type under definition.

- `version` and `description` are optional, and they permit versioning and describing (in natural language) a capability type.
- `properties` and `attributes` are optional, and they allow to indicate the desired and actual state of a capability, respectively.
- `valid_source_types` is optional, and it allows to list the node types that can be validly used as sources of relationships whose target capability is of the type under definition.

Capability types are then referred by node types. Each node type indeed defines (within its field `capabilities`) the set of named capabilities that can be exposed by node templates of such type. The grammars for capability definitions are displayed in Fig. 8.

```
cap_name: cap_type_name
```

(a)

```
cap_name:
  type: cap_type_name
  description: capability_description
  properties: property_definitions
  attributes: attribute_definitions
  valid_source_types: [ node_type_names ]
```

(b)

Fig. 8. (a) Simple and (b) extended grammars for *capability definitions* [4].

Both the simple grammar (a) and the extended grammar (b) allow to indicate the name (`cap_name`) of the capability under definition, and its type. The extended grammar (b) also allows to specify some optional fields (viz., `description`, `properties`, `attributes`, and `valid_source_types`), whose meaning is analogous to that of their homonym fields in the grammar for specifying capability types (Fig. 7).

Node templates are instances of node types, and they also instantiate the capabilities they define. Node templates can also assign concrete values to the properties and attributes of such capabilities, to provide additional information concerning their desired and actual state, respectively. Such a kind of capability assignments can be provided with the field `capabilities` of a node template, with the grammar in Fig. 9.


```

cap_name :
  properties: property_assignments
  attributes: attribute_assignments

```

Fig. 9. Grammar for *capability assignments* [4].

How to *validate* a TOSCA Capability

We hereby single out the conditions ensuring that a relationship template is not violating any of the interconnection constraints indicated by the capability it targets. The interconnection constraints concern the `valid_source_types`, and they can be indicated while specifying defining a capability type or while defining a capability in a node type. In the former case, the constraints can be indicated inline in the specification of capability type, or they can be inherited by the parent capability type.

Notation. We shall denote with $S(\cdot)$ the set of node types that are valid sources for a capability type. Given a capability type c_{type} :

- If $c_{type}.valid_source_types \neq \perp$, then $S(c_{type})$ is the set containing all node types in $c_{type}.valid_source_types$,
- otherwise, if $c_{type}.derived_from \neq \perp$, then $S(c_{type})$ is the set containing all node types in $S(c_{type}.derived_from)$,
- otherwise, $S(c_{type})$ is the set containing all node types (meaning that all node types are valid targets for c_{type}).

TOSCA application topologies are such that (requirement assignments of) multiple node templates can be sources of relationship templates targeting the same capability assignment c_a . All such node templates must not violate the interconnection constraints indicated by the capability type of c_a (Condition 3.1), nor those indicated in its corresponding capability definition (Condition 3.2).

Condition 3. Let c_a be a capability assignment, whose corresponding type and definition are c_{type} and c_d . For each node template n_{temp} having a requirement assignment r_a such that $r_a.capability = c_a$:

- (1) $\exists n_{type} \in S(c_{type}): type(n_{temp}) \geq n_{type} \wedge$
- (2) $\exists n_{type} \in S(c_d): type(n_{temp}) \geq n_{type}$

3.4 Valid TOSCA Application Topologies

In the previous sections we have singled out the formal conditions that must hold to ensure the validity of sources, instances, and targets of relationships in a TOSCA application topology. For the convenience of readers, all such conditions are recapped in Table 1. We below define the notion of validity for a TOSCA application topology, by gathering Conditions 1, 2, and 3 in a single definition.

Table 1. Formal conditions ensuring the validity of sources, instances, and targets of relationships in a TOSCA application topology.

Validating sources of relationships (Condition 1)	Let n_{temp} be a node template of type n_{type} . Then $\forall r_a \in \mathbf{R}(n_{temp}), \exists r_d \in \mathbf{R}(n_{type})$: (1.1) $\text{name}(r_a) = \text{name}(r_d) \wedge$ (1.2) $r_d.\text{node} \neq \perp \Rightarrow \text{type}(r_a.\text{node}) \geq r_d.\text{node} \wedge$ (1.3) $r_a.\text{capability} \neq \perp \Rightarrow \text{type}(r_a.\text{capability}) \geq r_d.\text{capability} \wedge$ (1.4) $r_a.\text{capability} = \perp \Rightarrow$ $\exists c \in \mathbf{C}(r_a.\text{node}): \text{type}(c) \geq r_d.\text{capability} \wedge$ (1.5) $r_d.\text{relationship} \neq \perp \wedge r_a.\text{relationship} \neq \perp \Rightarrow$ $\text{type}(r_a.\text{relationship}) \geq r_d.\text{relationship}$
Validating relationships (Condition 2)	Let rel_{temp} be a relationship template, and let rel_{type} be its relationship type. If there exists a requirement assignment r_a that is source of rel_{temp} , then: (2.1) $r_a.\text{capability} \neq \perp \Rightarrow$ $\exists c_{type} \in \mathbf{T}(rel_{type}): \text{type}(r_a.\text{capability}) \geq c_{type} \wedge$ (2.2) $r_a.\text{capability} = \perp \Rightarrow$ $\exists c_a \in \mathbf{C}(r_a.\text{node}), c_{type} \in \mathbf{T}(rel_{type}): \text{type}(c_a) \geq c_{type}$
Validating targets of relationships (Condition 3)	Let c_a be a capability assignment, whose corresponding type and definition are c_{type} and c_d . For each node template n_{temp} having a requirement assignment r_a such that $r_a.\text{capability} = c_a$: (3.1) $\exists n_{type} \in \mathbf{S}(c_{type}): \text{type}(n_{temp}) \geq n_{type} \wedge$ (3.2) $\exists n_{type} \in \mathbf{S}(c_d): \text{type}(n_{temp}) \geq n_{type}$

Definition 1. A TOSCA application topology is valid if all its node templates and relationship templates satisfy Conditions 1, 2, and 3.

4 Prototype Implementation

In this section, we present SOMMELIER, a Python prototype of validator for TOSCA application topologies (based on the formal conditions discussed in Sect. 3. The prototype of SOMMELIER is open-source¹¹, and it is fully integrated with the OpenStack TOSCA parser [8].

4.1 SOMMELIER

SOMMELIER validates TOSCA application topologies as illustrated in Fig. 10:

- ❶ SOMMELIER takes as input a CSAR archive or a .tosca document. The input contains the application topology to be validated.
- ❷ SOMMELIER forwards the input to the OpenStack TOSCA parser [8].

¹¹ The source code of SOMMELIER is publicly available on GitHub at <https://github.com/di-unipi-socc/Sommelier>.

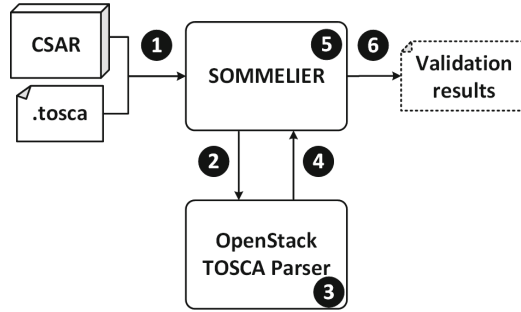


Fig. 10. Workflow followed by SOMMELIER while checking the validity of a TOSCA application topology [10].

- ③ The OpenStack TOSCA parser checks whether the specified application is syntactically correct, viz., whether all its elements have been specified by respecting the grammar of TOSCA, as well as each template has been defined by respecting the structure indicated by its corresponding type. If this is the case, the TOSCA parser of OpenStack generates a representation of the TOSCA application in Python, according to its object-model [8].
- ④ The OpenStack TOSCA parser returns the Python representation of representing the TOSCA application to SOMMELIER.
- ⑤ SOMMELIER validates the topology of the TOSCA application. More precisely, SOMMELIER check whether all the conditions listed in Table 1 are satisfied by all the nodes and relationships of the input topology. The result is a Python dictionary¹² structured as indicated in Fig. 11. The dictionary associates each requirement of each node template with a list containing all validation errors affecting its outgoing relationship. Errors are in turn represented with lists, which head is the error code (e.g., 1.2, if Condition 1.2 is violated) and which remaining elements provide additional information on the error (e.g., type/-name of the target node, which is not satisfying Condition 1.2).
- ⑥ SOMMELIER displays the results of the validation, viz., it states that the analysed topology is valid, or it provides the list of all violations that make the topology not valid.

Example. In this example, we show how SOMMELIER can be fruitfully exploited to validate TOSCA application topologies. An instance of SOMMELIER can be run with the following command line instruction:

```
$ python sommelier.py --template-file=template-file-path
```

(where `template-file-path` indicates the absolute path for retrieving the TOSCA file to be validated).

Figure 12 shows two concrete runs of SOMMELIER, to which we passed as input two TOSCA application specifications available in the GitHub repository

¹² <https://docs.python.org/3/tutorial/datastructures.html>.

```

{
  node_template1_name : {
    req1_name : [
      [ error1_code , error1_info ],
      [ error2_code , error2_info ],
      ...
    ]
    req2_name : [ ... ],
    ...
  },
  node_template2_name : { ... } ,
  ...
}

```

Fig. 11. Structure of the Python dictionary containing the results of the validation [10].

```

$ python sommelier.py --template-file=./tosca-parser/toscaparser/
tests/data/topology_template/tosca_elk.yaml

The application topology is valid.

```

(a)

```

$ python sommelier.py --template-file=./tosca-parser/toscaparser/
tests/data/topology_template/transactionssubsystems.yaml

NODE_TEMPLATE: app
REQUIREMENT: host
1.2 - NODE_TYPE_NOT_COHERENT: The type "tosca.nodes.WebServer"
of the target node "webserv" is not valid (as it differs from
that indicated in the requirement definition).

```

(b)

Fig. 12. Example of runs of SOMMELIER [10].

of the OpenStack TOSCA parser, viz., *tosca_elk.yaml* and *transactionssystem.yaml*.¹³ *tosca_elk.yaml* turned out to be valid (Fig. 12(a)).

The topology of *transactionssystem.yaml* instead resulted to be not valid. As shown in Fig. 12(b), this is because the relationship outgoing from requirement *host* of the node template *app* is violating Condition 1.2 (since it targets the node template *webserv*, whose type is not compatible with those indicated in the requirement definition corresponding to *host*). Without a design-time support

¹³ The files are developed and maintained by the community around the OpenStack TOSCA parser. They are publicly available at <https://github.com/openstack/tosca-parser/tree/master/toscaparser/tests/data>. We hereby consider the version of the files available on November 13th, 2016.

like that offered by SOMMELIER, such a kind of issues would have been hard to detected, as the validation should have been performed manually. \square

4.2 Unit Testing of SOMMELIER

We developed a battery of unit tests covering 99% of the source code of SOMMELIER, to double-check that SOMMELIER was capable of recognising all possible violations of all conditions in Table 1. More precisely, we developed a set of non-valid TOSCA application specification, each containing a violation of a condition in Table 1. Each specification was obtained by injecting an error in the valid application topology specified in *tosca_ellk.yaml* (Fig. 12(b)).

To run the unit tests, we must first clone the *master* branch of the GitHub repository of SOMMELIER in a host folder. This can be done by running the following command:

```
$ git clone https://github.com/di-unipi-socc/Sommelier.git
```

We can then execute all unit tests by running the following command in the newly created `sommelier` folder¹⁴:

```
$ coverage run --source topologyvalidator \
    -m unittest discover
```

We can then display the source code coverage by running the command in Fig. 13.

\$ coverage report			
Name	Stmts	Miss	Cover
topologyvalidator.py	227	2	99%

Fig. 13. Source code coverage of SOMMELIER.

5 Related Work

Validation techniques permit checking whether software systems fulfill a set of specified requirements [12]. Such techniques are crucial nowadays, as software systems are more and more involved in our everyday life, and ensuring that they fulfill desired requirements is imperative [13].

The OASIS standard TOSCA [4] recognises the importance of validation. TOSCA allows to indicate a set of constraints on how to interconnect application components, which have to be fulfilled when building the topology of a cloud application. The design-time support for verifying such constraints is however currently limited, and this makes the validation of TOSCA applications a cumbersome and time-consuming process.

¹⁴ To run `coverage`, the `coverage-py` Python library must be installed on the host. The latter can be installed by executing `sudo pip install coverage`.

OpenStack recently developed a TOSCA parser [8], which can be used to verify the syntactical correctness of TOSCA application specifications. The OpenStack TOSCA Parser indeed allows to check whether all the elements in a TOSCA specification have been provided in their proper section (e.g., relationship types in the `relationship_types` section, relationship templates in the `relationship_templates` section), and that all templates have been specified coherently with the structure indicated by the corresponding types (e.g., the properties indicated by a relationship template are also defined in the corresponding relationship type, numeric properties contain numeric values). Even if it can be used to check that the actual values assigned to the fields of a template are of the proper type, the OpenStack TOSCA Parser does not check their “meaningfulness”. For instance, the OpenStack TOSCA Parser can fruitfully be exploited to check that the `node` field of a requirement assignment contains the name of a node template, but it does not provide any information on whether the indicated node template satisfies the interconnection constraints defined in the requirement definition. The objective of this paper (and of SOMMELIER) is precisely to allow to check such a kind of interconnection constraints.

Similar considerations apply to other TOSCA parsers currently available, e.g., the brooklyn-tosca parser [14], that employed in SeaClouds [15], or the TOSCA parser in Alien4Cloud [16].

The OASIS standard TOSCA is also available in an older, XML-based version [17]. TOSCA XML still allows to specify the topology of a cloud application and interconnection constraints on application components. It also allows to define management plans, which can be specified as workflows that orchestrate the management operations of the components forming an application.

Winery and OpenTOSCA are two open-source tools allowing to edit and execute cloud applications specified in TOSCA XML. Despite both tools are provided with parsers verifying the syntactical correctness of TOSCA XML applications, a support for validating the interconnection forming the topology of an application is currently lacking.

A first approach exploiting some of the interconnection constraints that can be expressed in TOSCA XML is that in [18], which proposes a solution for automatically completing TOSCA XML application topologies. The solution is based on the idea of exploiting the interconnection constraints to select (from a set of available components) the components that can be used to satisfy dangling requirements. [18] however differs from our approach mainly due to its objectives. It indeed relies on TOSCA XML, and it only considers some of the interconnection constraints that can be specified in TOSCA XML, viz., those indicating which capability types and relationship types can be used to satisfy a requirement. Our objective is instead to enable a full validation of TOSCA application topologies, by systematically mapping all the interconnection constraints that can be specified in TOSCA to formal conditions that must be ensured when building application topologies.

There also exist approaches for validating the management plans indicated in TOSCA XML application specifications. Such approaches are based on manage-

ment protocols [19, 20], a compositional modelling for specifying the management behaviour of application components. The behaviour of the components forming a TOSCA XML application can then be combined (according to the topology of the application) to automatically derive the management behaviour of the application. The latter allows to automate various analyses, including the validation of management plans. At the same time, even if the topology of an application is fundamental to derive its management behaviour, it is always assumed to be valid (because of the lack of a design-time support for validating TOSCA XML application topologies).

In summary, even if TOSCA allows to indicate interconnection constraints that must be fulfilled when building application topologies, there is currently a lack of a design-time support for checking such constraints. The contributions presented in this paper can help solving this lack, as we systematically map the interconnection constraints that can be expressed in TOSCA to formal conditions, and we exploit such conditions to provide a first support for validating TOSCA application topologies.

It is finally worth highlighting that our approach follows the baselines of existing approaches for validating multi-component systems at design time, e.g., [21–23], or for automatically synthesizing them, e.g., [24, 25]. Similarly to all such approaches, we start from the specification of a multi-component system, and we try to enforce that all its components are properly interconnected (viz., that no interconnection violates the constraints imposed by its source and target components, and by the interconnection itself). The main difference between such approaches and ours is given by the context. While [21–25] target the valid composition of the functionalities offered by a set of components, we focus on ensuring that the dependencies between the components forming a TOSCA application are properly specified, as such information is the basis for orchestrating the management of a TOSCA application.

6 Conclusions

The OASIS standard TOSCA permits specifying cloud applications, and automating their management. TOSCA permits describing the structure of an application as a topology graph, which is then exploited by TOSCA-compliant cloud platforms to automate the management of the components forming an application. As the automated deployment of TOSCA applications is based on their topologies [7], it is fundamental to ensure—at design time—that such topologies are valid.

TOSCA allows to indicate the interconnection constraints that must be fulfilled when building the topology of a cloud application (e.g., a node can indicate which types of nodes and relationships can be used to satisfy its requirements). In this paper we have systematically mapped such constraints to formal conditions that must hold to ensure the validity of a TOSCA application topology (see Table 1). We have also introduced a first prototype of validator based on such conditions, viz., SOMMELIER.

SOMMELIER is open-source and it is already integrated with the open-source TOSCA parser developed by the OpenStack community [8]. SOMMELIER and the OpenStack TOSCA parser can put the basis for the development of a full-fledged design time support for TOSCA application developers.

In this perspective, SOMMELIER can be fruitfully exploited for validating TOSCA application topologies while they are being developed, e.g., by improving the functionalities of existing graphical editors (such as that in the SeaClouds platform [15], for instance). Its output could indeed be exploited for suggesting how to fix errors (e.g., by highlighting a misplaced requirement, and by suggesting which nodes can be used to actually satisfy it) or to drive the development itself (e.g., by impeding developers to wrongly interconnecting nodes). The integration of SOMMELIER with an existing graphical editor is left for future work. This is in line with the research directions indicated in [9], in particular with the development of tools to support TOSCA developers from the design time till the run time.

Notice that the version of TOSCA considered in this paper [4] is a simplified profile of the former, XML-based version of TOSCA [17]. All the interconnection constraints that can be indicated in the considered version of TOSCA can also be specified in TOSCA XML. We hence plan to check whether the proposed conditions for validating TOSCA applications can be exploited also for validating applications specified in TOSCA XML (or whether they need to be extended). We also plan to implement an extended version of SOMMELIER capable of validating TOSCA XML application topologies, and to integrate with the OpenTOSCA open-source environment [5, 11].

Notice also that both versions of TOSCA also allow to indicate non-functional requirements of application components (such as scalability policies or QoS requirements). All such non-functional requirements should also be enforced, hence requiring TOSCA application specifications should be validated from a non-functional perspective. We plan to devise a technique for carrying out such a validation, and to include it within SOMMELIER.

Acknowledgements. We would like to thank Luca Rinaldi for his valuable help in preparing the battery of unit tests for SOMMELIER.

References

1. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M.: A view of cloud computing. *Commun. ACM* **53**, 50–58 (2010)
2. Binz, T., Breitenbücher, U., Kopp, O., Leymann, F.: TOSCA: portable automated deployment and management of cloud applications. In: Bouguettaya, A., Sheng, Q., Daniel, F. (eds.) *Advanced Web Services*, pp. 527–549. Springer, New York (2014). https://doi.org/10.1007/978-1-4614-7535-4_22

3. Brogi, A., Carrasco, J., Cubo, J., D'Andria, F., Ibrahim, A., Pimentel, E., Soldani, J.: EU Project SeaClouds - adaptive management of service-based applications across multiple clouds. In: CLOSER 2014 - Proceedings of the 4th International Conference on Cloud Computing and Services Science, pp. 758–763. SciTePress (2014)
4. OASIS: TOSCA Simple Profile in YAML, Version 1.0 (2016). <http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/TOSCA-Simple-Profile-YAML-v1.0.pdf>
5. Binz, T., et al.: OpenTOSCA – a runtime for TOSCA-based cloud applications. In: Basu, S., Pautasso, C., Zhang, L., Fu, X. (eds.) ICSSOC 2013. LNCS, vol. 8274, pp. 692–695. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-45005-1_62
6. Endres, C., Breitenbücher, U., Falkenthal, M., Kopp, O., Leymann, F., Wettinger, J.: Declarative vs. imperative: two modeling patterns for the automated deployment of applications. In: Proceedings of the 9th International Conference on Pervasive Patterns and Applications, pp. 22–27. Xpert Publishing Services (XPS) (2017)
7. OASIS: Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer (2013). <http://docs.oasis-open.org/tosca/tosca-primer/v1.0/tosca-primer-v1.0.pdf>
8. OpenStack: TOSCA Parser (2016). <https://github.com/openstack/tosca-parser>
9. Brogi, A., Soldani, J., Wang, P.: TOSCA in a nutshell: promises and perspectives. In: Villari, M., Zimmermann, W., Lau, K.-K. (eds.) ESOC 2014. LNCS, vol. 8745, pp. 171–186. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44879-3_13
10. Brogi, A., Di Tommaso, A., Soldani, J.: Validating TOSCA application topologies. In: Pires, L.F., Hammoudi, S., Selic, B. (eds.) Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development, MODEL-SWARD 2017, Porto, Portugal, 19–21 February 2017, pp. 667–678. SciTePress (2017)
11. Kopp, O., Binz, T., Breitenbücher, U., Leymann, F.: Winery – a modeling tool for TOSCA-based cloud applications. In: Basu, S., Pautasso, C., Zhang, L., Fu, X. (eds.) ICSSOC 2013. LNCS, vol. 8274, pp. 700–704. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-45005-1_64
12. Geraci, A.: IEEE Standard Computer Dictionary: Compilation of IEEE Standard Computer Glossaries. IEEE Press, New York (1991)
13. Marchetti, E.: Foreword of the thematic track: ICT verification and validation. In: Proceedings of the 9th International Conference on the Quality of Information and Communications Technology, QUATIC 2014, pp. 208–209. IEEE (2014)
14. Brooklyn-tosca (2016). <https://github.com/cloudsoft/brooklyn-tosca>
15. Brogi, A., et al.: SeaClouds: an open reference architecture for multi-cloud governance. In: Tekinerdogan, B., Zdun, U., Babar, A. (eds.) ECSA 2016. LNCS, vol. 9839, pp. 334–338. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48992-6_25
16. Alien4cloud (2016). <https://github.com/alien4cloud/alien4cloud>
17. OASIS: Topology and Orchestration Specification for Cloud Applications (2013). <http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.pdf>
18. Hirmer, P., Breitenbücher, U., Binz, T., Leymann, F.: Automatic topology completion of TOSCA-based cloud applications. In: INFORMATIK 2014. LNI, vol. 232, pp. 247–258. Gesellschaft für Informatik (GI) (2014)

19. Brogi, A., Canciani, A., Soldani, J.: Modelling and analysing cloud application management. In: Dustdar, S., Leymann, F., Villari, M. (eds.) ESOCC 2015. LNCS, vol. 9306, pp. 19–33. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24072-5_2
20. Brogi, A., Canciani, A., Soldani, J., Wang, P.: A petri net-based approach to model and analyze the management of cloud applications. In: Koutny, M., Desel, J., Kleijn, J. (eds.) Transactions on Petri Nets and Other Models of Concurrency XI. LNCS, vol. 9930, pp. 28–48. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53401-4_2
21. Speck, A., Pulvermuller, E., Jerger, M., Franczyk, B.: Component composition validation. *Int. J. Appl. Math. Comput. Sci.* **12**, 581–590 (2002)
22. Caporuscio, M., Inverardi, P., Pelliccione, P.: Compositional verification of middleware-based software architecture descriptions. In: Proceedings of the 26th International Conference on Software Engineering, ICSE 2004, pp. 221–230. IEEE Computer Society (2004)
23. Wu, Y., Chen, M.-H., Offutt, J.: UML-based integration testing for component-based software. In: Erdogmus, H., Weng, T. (eds.) ICCBSS 2003. LNCS, vol. 2580, pp. 251–260. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36465-X_24
24. Autili, M., Inverardi, P., Navarra, A., Tivoli, M.: SYNTHESIS: a tool for automatically assembling correct and distributed component-based systems. In: Proceedings of the 29th International Conference on Software Engineering, ICSE 2007, pp. 784–787. IEEE Computer Society (2007)
25. Pelliccione, P., Tivoli, M., Bucchiarone, A., Polini, A.: An architectural approach to the correct and automatic assembly of evolving component-based systems. *J. Syst. Softw.* **81**, 2237–2251 (2008)



Evaluation of XIS-Reverse, a Model-Driven Reverse Engineering Approach for Legacy Information Systems

André Reis^(✉) and Alberto Rodrigues da Silva

INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal
{andre.filipe.reis,alberto.silva}@tecnico.ulisboa.pt

Abstract. Companies have been struggling to manage and maintain their legacy information systems because upgrading said systems has been a complex challenge. Many times, requirements changes are difficult to be properly managed, leading to legacy information system requirements deterioration. To overcome or reduce such problems we propose the XIS-Reverse, a software reverse engineering approach. XIS-Reverse is a model-driven reverse engineering approach that takes database artefacts and user preferences as input, and generates high-level models and specifications of these legacy information systems. This paper presents the evaluation of XIS-Reverse using two real-world information systems, provides an assessment of its interoperability with an existent framework and discusses its main challenges and benefits.

Keywords: Model-driven engineering
Model-driven reverse engineering · Model-driven reengineering
Database · Legacy system

1 Introduction

One of the main reasons software projects tend to fail is the difficulty to manage their requirements, mainly due to the fact that requirements changes are difficult to be managed [1]. Without a proper way to manage requirements, software projects may have consequences, namely excessive development and management costs, the development of a system which does not meet stakeholders needs, and so on. Although new methods to collect, analyze, document and maintain requirements have been appearing, their software requirements specifications are still mainly written in natural language [1]. Those kinds of specifications are usually hard to keep up to date while the software applications are being developed, leading to deterioration. To overcome or reduce such problems, software reverse engineering approaches can be used.

Reverse engineering was initially used in hardware analysis, but it quickly extended its scope to software systems [2]. Then, following the huge expansion and advent of software from the end of the 80s, the reverse engineering topic has

been mainly used in the context of legacy information systems, which are often still responsible for running crucial and critical operations for companies [3].

Reverse engineering can be defined as the process of examining an already implemented software system to create a higher abstraction level representation in a different form [2].

The main objective of such representations is to provide a better understanding of the software system's current state. These can be used to correct (e.g. fix bugs), update (e.g. alignment with updated user requirements), upgrade (e.g. add new capabilities), or even completely reengineer the system under study [3]. These operations are happening now more than ever due to new user requirements and expectations, adaptation to emerging business models, updated legislation, new technology innovation and prevention of system structure deterioration [4]. Since reverse engineering an information system is a time-consuming and error-prone process, any reverse engineering solution that increases the automation level of the process will benefit the users of such complex task, and thus facilitate its larger adoption.

Model-driven engineering (MDE) approaches are increasingly gaining acceptance in the software engineering field to tackle software complexity and to improve software productivity [5,6]. These approaches promote the systematic use of models, raising the level of abstraction at which software is specified and increasing the automation level of software development applying model transformations. Although most of the MDE approaches use forward engineering techniques to, for instance, transform higher-level models into source code, MDE can also be used to perform the opposite transformation using reverse engineering techniques (Model-Driven Reverse Engineering (MDRE)) [3].

XIS-Reverse [7] aims to mitigate requirements deterioration and maintenance of legacy information systems, reducing human effort and improving productivity. Such goals can be achieved using reverse engineering techniques based on a model-driven approach, producing high-level specifications of information systems through model transformations. This is accomplished using and extending the Sparx Systems Enterprise Architect (EA) tool with those transformations.

This paper extends the previous work that introduced the XIS-Reverse approach [7] with the following novel contributions: (i) an extensive discussion of the relevance of this approach based on the evaluation of two real-world cases studies with large databases (e.g., case study B involves more than 150 data entities and more than 200 associations); (ii) a discussion showing how to combine the XIS-Reverse approach with forward engineering approaches, namely the XIS-Web approach [8], by showing that the extracted models (with the XIS-Reverse) can then be involved in models validation, model-to-model and model-to-text transformations; (iii) finally, a comparison of the XIS-Reverse with other approaches and a discussion of the related work.

Furthermore, regarding the main contributions of XIS-Reverse, we have to highlight the following aspects: (i) semi-automatic heuristics that can identify certain relationships between entities, namely implicit generalizations and aggregations (specialization of associations), and also (ii) the possibility to extract values from the source database to enrich the target models or specifications.

All that combined enhances the understanding of each entity’s role in the produced models, and consequently the comprehension of the information system.

XIS-Reverse was developed in sixteen months following the Action Research methodology [9]. Over time it was necessary to evaluate the level of detail and correctness of the XIS-Reverse extracted specifications. This evaluation process was done in an iterative way. Initially, this was done testing only a subset of the approach, namely the domain entities extraction with simple case studies. However, throughout this period, we got the chance to test XIS-Reverse using real-world applications, increasing the relevance of XIS-Reverse’s results.

The outline of this paper is as follows. Section 2 presents the context. Section 3 gives an overview of the XIS-Reverse. Section 4 presents and analyses the evaluation performed to XIS-Reverse using two real-world applications. Section 5 presents and analyses the interoperability evaluation of XIS-Reverse with an existent framework. Section 6 analyses and compares this proposal with the related work. Finally, Sect. 7 summarizes the main conclusions of this work along with some future work perspectives.

2 Background

This research has been developed at the Instituto Superior Técnico, Universidade de Lisboa, in the scope of the MDDLingo¹ and the RSLingo² initiatives.

MDDLingo is an umbrella researching initiative that aggregates several projects around MDE topics, namely involving the definition of a family of languages, also known as XIS*. This set of modelling languages derives from the XIS-UML profile [10], involving namely XIS-Mobile [11, 12], XIS-CMS [13] or XIS-Web [8]. XIS-UML is a set of coherent constructs defined as an UML profile that allows a high-level and visual modeling way to design business information systems. In general these languages include the following views: Entities (which includes Domain and Business Entities views), UseCases (containing Actors and Use Cases views), Architectural and User-Interfaces (composed by Interaction Space and Navigation Space views).

Figure 1 illustrates a simple XIS* Domain view which aggregates domain classes (XisEntity), their attributes (XisEntityAttribute) and relationships (XisEntityAssociation and XisEntityInheritance).

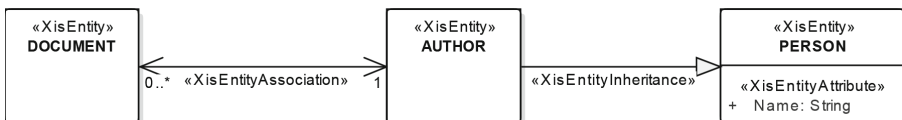


Fig. 1. Example of a XIS* Domain view.

¹ <https://github.com/MDDLingo>.

² <https://github.com/RSLingo>.

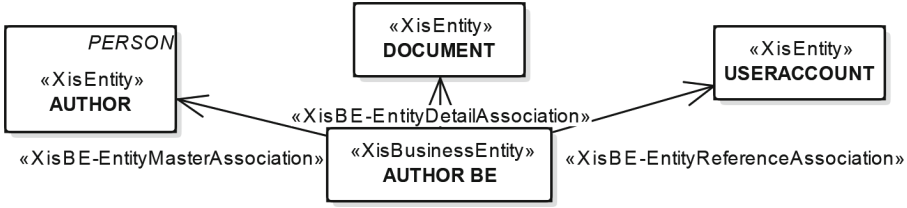


Fig. 2. Example of a XIS* BusinessEntities view.

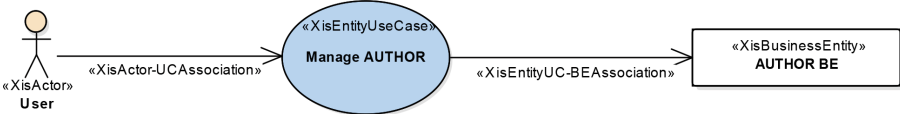


Fig. 3. Example of a XIS* UseCases view.

Figure 2 shows a BusinessEntities view, which allows to define higher-level entities (XisBusinessEntity), that aggregate XisEntities and that in the context of a given use case can be easily manipulated.

Figure 3 shows the UseCases View. This view details the operations an actor can perform over the business entities when interacting with the system [11].

RSLingo is a general approach defined to rigorously specify and validate software requirements using lightweight Natural Language Processing techniques to (partially) translate informal requirements into a rigorous representation provided by a language specially designed for Requirements Engineering. Over time, following the RSLingo’s approach, several projects have been developed, namely RSLingo4Privacy [14] and RSLingo’s RSL³ [15]. Moreover, RSLingo’s RSL is a controlled natural language (restricted use of a natural language grammar and a set of standardized terms to be used in a restricted grammar) to help the production of software requirements specifications in a more systematic, rigorous and consistent way [15]. Such specifications are usually specified as a set of .rsl files, and later they can be validated and used by different types of users such as requirement engineers, business analysts, or domain experts [15]. The most relevant RSLingo’s RSL concepts regarding our research are: Data Entities, Data Entity Views, User Stories, Functional Requirements, Goals, Business Processes and Terms.

3 XIS-Reverse Overview

The XIS-Reverse [7] is a MDRE approach that allows to extract high-level specifications from legacy application artefacts.

As illustrated in Fig. 4, the XIS-Reverse approach starts by extracting the application data model from an available database, and from that and from the

³ <https://github.com/RSLingo/RSL>.

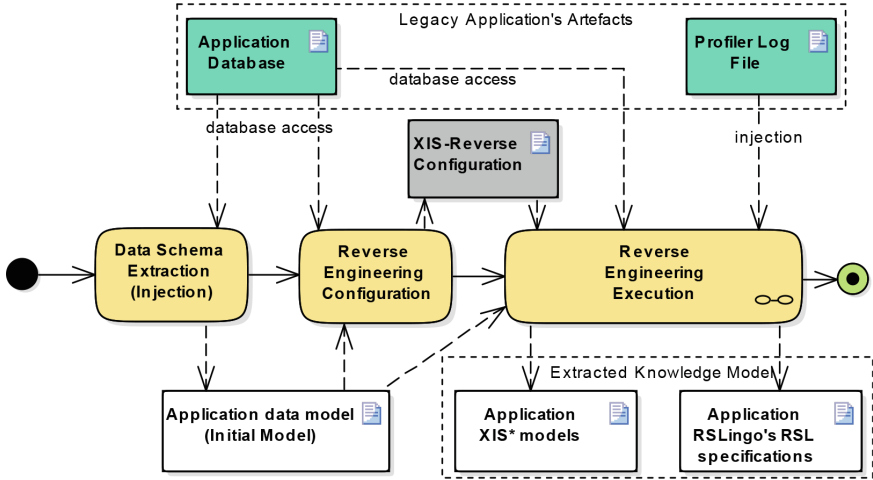


Fig. 4. Overview of the XIS-Reverse approach.

user configuration (second stage), the reverse engineering execution takes place, by applying several reverse engineering heuristics on those artefacts, and then generating the extracted knowledge in the form of models and specifications.

XIS-Reverse was implemented on top of the Sparx Systems EA⁴, as an EA plug-in. The XIS-Reverse's first stage (data schema extraction) relies on the native capability of EA to reverse engineer a database schema through an ODBC connection. Then, the following stages (reverse engineering configuration and execution) are supported by the XIS-Reverse tool (available from GitHub⁵). The configuration stage provides a user interface (see Fig. 5) that can be split into 4 different areas:

- **Input** - to specify input artefacts, namely the application data model, database name and additional artefacts, namely a database access or a profiler log file;
- **Output** - to select additional output representations, namely XIS-Web and RSLingo's RSL;
- **Transformation Rules Guidance** - to provide configuration points to the following features: Simple Principal Entities (to identify aggregations); Attribute Values Extraction (to extract attribute values); and Generalization Discovery (to detect implicit generalizations);
- **Appearance** - to improve the readability of the produced specifications.

Although the number of available input technologies and output specifications can be extended, for now our approach is able to produce XIS-Web language models [8] and RSLingo's RSL specifications [15] from Microsoft SQL Server databases.

⁴ <http://www.sparxsystems.com/products/ea>.

⁵ <https://github.com/MDDLingo/xis-reverse>.

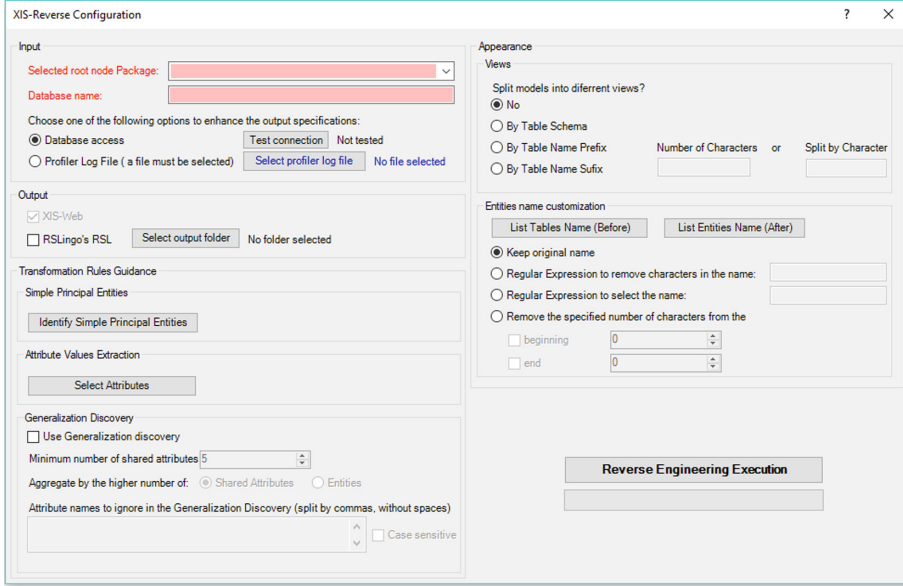


Fig. 5. Main configuration panel of the tool.

Finally, the reverse engineering execution stage is supported by Model-to-Model (M2M) transformations that use the application data model and user configurations to generate XIS* models and RSLingo’s RSL specifications. Then, the user can analyze the produced artefacts and introduce some refinements, such as changing automatically identified relationships into different ones in the Entities view, enhancing the Use-Cases views, etc.

4 Evaluation

In this section, two case studies are introduced and used to assess the XIS-Reverse approach.

Both applications were supported by SQL Server databases and our experiments only considered database access in order to enhance the output specifications since it is harder to generate a profiler log file that adequately represents the normal usage of such applications.

To assess the overall results of each Case Study we divided this evaluation into three levels of configuration scenarios: Without configuration, Blind configuration and Semi-guided configuration. Within each scenario we extracted: number of XisEntities (including explicit and implicit superclasses); number of XisAssociations (also including Aggregations and Many-to-many associations); number of Aggregations; number of Many-to-many associations; number of explicit and implicit subclasses and superclasses; number of XisBusinessEntities; number of each XisBusinessEntity associations and number of XisEntityUseCases.

Moreover, we defined some heuristics to evaluate the obtained results in a deeper way, namely in terms of aggregation associations and implicit generalizations. However, those heuristics will not be applied to the Case Study B due to privacy constraints.

Regarding aggregations, we defined two rules. The first one requires having an updated domain model in the available application requirements, in which entity associations are classified (e.g. one-to-one or aggregation associations). The second one requires having every entity manually classified as a main entity (e.g. relevant entity in the domain), configuration entity (e.g. “kind of” entity) or association entity (e.g. entity whose main purpose is to link two or more entities).

Rule-1: Number of Associations Well Classified in Terms of Aggregations. We assess this rule by applying the concepts of a confusion matrix to the results (after the experiment), thus we count the number of: (1) actual aggregations that were correctly classified as aggregations (true positive); (2) non-aggregations that were incorrectly classified as aggregations (false positive); (3) aggregations that were incorrectly marked as non-aggregations (false negative); (4) all the remaining associations correctly classified as non-aggregations (true negative).

Rule-2: Number of Configuration Entities that do not Aggregate Main Nor Association Entities. We assess this rule by counting how many of those did and did not aggregate main or association entities (after the experiment).

Regarding generalizations, we want to extract implicit generalizations which maximize both the number of subclasses found (variable x) and the number of inherited attributes (variable y), based on the following function:

$$Reis(x, y) = 0.5x + 0.5y \quad (1)$$

To better explain the Reis function and its variables, a simple domain model illustrated in Fig. 6 will be used.

Variable x is determined by the number of subclasses found (after the experiment), divided by the maximum number of subclasses that could be found (number of entities without generalization and with at least 1 attribute (before the experiment), such as A, B, C, F and G (5) in Fig. 6).

Taking into account that generalizations with the exact number of two subclasses will always maximize the number of superclasses that can be found, and thus, maximize also the number of inherited attributes:

Variable y is determined by the sum of all the superclass attributes found (after the experiment), divided by the sum of the maximum number of attributes that could be inherited (the sum of the maximum number of attributes every pair of entities can share (before the experiment), taking into account all pairs of entities that can be grouped, by the descending order of attributes number, such as 3 in Fig. 6, since pairs A-B share at most 2 attributes and C-F share at most 1 attribute, for example).

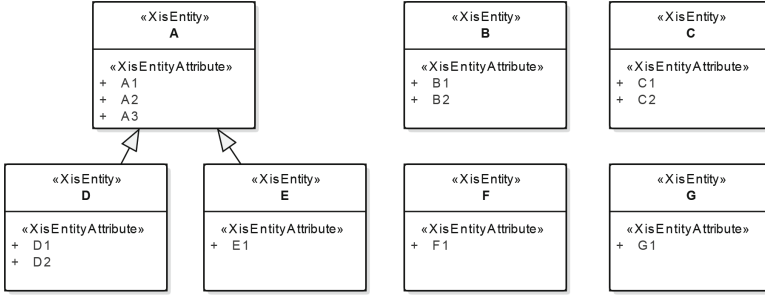


Fig. 6. Support example to explain the Reis function.

4.1 Case Study A: ProjectIT-Enterprise

The ProjectIT⁶ [16] initiative aggregates several research topics, such as software engineering and software development. The main goal behind this initiative is to provide a complete software development workbench, with support for project management, requirements engineering, analysis, design and code generation features. Moreover, within this initiative, a collaborative tool with Web interface was developed. This web application, called ProjectIT-Enterprise [17,18], provides a mechanism to process definition, collaborative support for teamwork, emphasizing project management, project-process alignment, workflows and documents management.

Although ProjectIT-Enterprise was mainly used and tested in an academic and research scope, it is mature, with well-defined concepts and requirements. Since we had the chance to use it, we decided to perform an exhaustive experiment to assess the XIS-Reverse.

Regarding the aggregation rules, since we had access to the domain model specifications and database of this application (Fig. 7), and it was granted that there were no significant updates in the database since this specification was defined, we used the said specification to evaluate against our experiment (required for Rule-1). With that, we established a mapping between every database table and the corresponding entity in the domain model (Table 1) and then, with some domain knowledge, we classified those entities/tables as main entities, configuration entities and association entities (required for Rule-2). This mapping and classification will be used during the evaluation to compare the extracted specification (using the XIS-Reverse) with the aforementioned domain model, shown in Fig. 7.

Moreover, we also identified the direct relationships between the main entities in the domain, defined as foreign key constraints in the application database (checked symbols in Fig. 7).

Taking into account Fig. 7, from the total of 8 direct relationships identified, 7 were aggregations (relaxing the composition definition) and 1 was a one-to-one relationship (not an aggregation).

⁶ <http://isg.inesc-id.pt/alb/ProjectIT>.

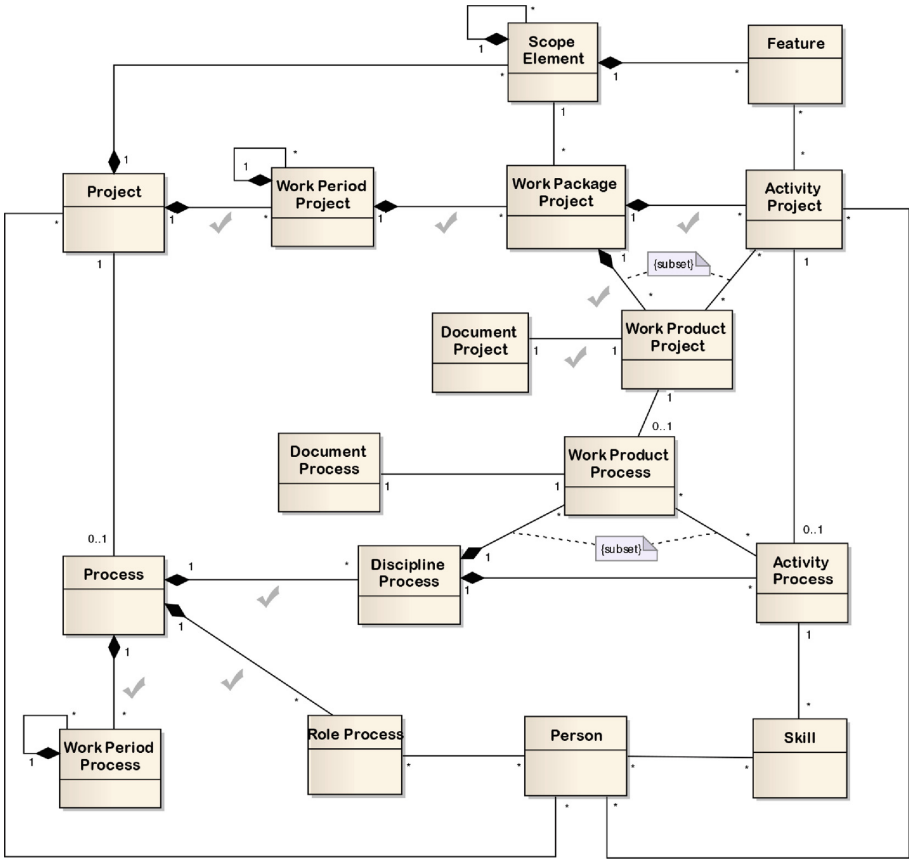


Fig. 7. Case Study A - domain model of the project dimension (adapted from [17]).

Table 2 shows the results of applying the aggregation rules using the XIS-Reverse. These results are analyzed below, in each of the configuration scenarios.

Furthermore, to support each scenario, Table 3 presents the overall picture of the extracted elements.

Scenario A: Extraction Without Configuration. In this scenario, we only used the minimum required configurations, namely select the root node package, provide the database name and select database access to enhance the specifications. With that, our aim was to extract and analyze of the simplest scenario used with the XIS-Reverse, and then to compare the obtained results with the scenarios that use configurations (Scenario-B and Scenario-C).

The first execution of this configuration scenario allowed to identify a problem in our approach, namely the identification of many-to-many associations (rule E-1 [7]). This problem occurred due to the generic definition of such heuristic that did not take into account composite primary keys. Moreover, that issue

Table 1. Case Study A - equivalence between application database and domain model.

Application database table	Domain model entity	Manual classification
ActivityEffort	-	Main
ActivityMembers	-	Association
ActivityProcess	Activity Process	Main
ActivityProcessSkills	-	Association
ActivityProject	Activity Project	Main
ActivityProjectTemplate	-	Main
ActivityProjectTemplateSkills	-	Association
Country	-	Configuration
DisciplineProcess	Discipline Process	Main
DisciplineProjectTemplate	-	Main
DocumentProcess	Document Process	Main
DocumentProject	Document Project	Main
DocumentProjectTemplate	-	Main
PrivacyLevel	-	Configuration
Process	Process	Main
ProcessDefinition	-	Main
Project	Project	Main
ProjectMembers	-	Association
RoleActivities	-	Association
RoleProcess	Role Process	Main
RoleSkills	-	Association
Skill	Skill	Configuration
State	-	Configuration
TimePeriod	-	Configuration
UserProfile	Person	Main
UserSkills	-	Association
WorkPackage	Work Package Project	Main
WorkPackageMembers	-	Association
WorkPeriodProcess	Work Period Process	Main
WorkPeriodProject	Work Period Project	Main
WorkPeriodProjectTemplate	-	Main
WorkProductProcess	Work Product Process	Main
WorkProductProject	Work Product Project	Main
WorkProductProjectTemplate	-	Main

Table 2. Case Study A - evaluation of aggregation scenarios.

Scenarios	Results					
	Rule-1				Rule-2	
	True positive	False positive	False negative	True negative	Configs. without aggregations	Configs. with aggregations
A: Without Configs.	6	1	1	0	4	1
B: M = 20	0	0	7	1	5	0
B: M = 10	2	1	5	0	5	0
B: M = 5	3	1	4	0	4	1
C: Semi-guided	6	1	1	0	5	0

occurred in cases that an entity had at least 2 primary keys (which only one of them was a foreign key), there was only one attribute and that attribute had a foreign key constraint. Taking that into account, we redefined that heuristic (updated listing available on GitHub (see footnote 5)).

After updating that heuristic, a new execution was performed in which 34 XisEntities were found with 45 XisEntityAssociations established, from which 42 were classified as aggregations. Regarding our aggregation evaluation Rule-1, from the 8 direct relationships, 6 aggregations were well identified. However, one aggregation was misinterpreted as a simple XisEntityAssociation and the one-to-one relationship was wrongly classified as an aggregation. The first problem occurred due to the difference of rows' number of each entity in the database, and since that difference goes against the rule EA-2-b ([7]) and there is no available configuration able to correct this problem, this type of issue had to be solved manually. On the other hand, the second problem was due to the absence of a Unique Index property in that foreign key, which was probably forgotten or relaxed.

Moreover, following Rule-2, from all configuration entities, only the Skill entity had aggregation associations with main or association entities. This can be solved by classifying this entity as Simple Principal Entity during the configuration stage.

There were no many-to-many associations identified, neither explicit generalizations. Thus, to improve the quality of the obtained specifications in this scenario, the main configurations that make sense to explore, in the following configuration scenarios, are the identification of Simple Principal Entities and Generalization discovery.

Scenario B: Extraction with Blind Configuration. After the previous configuration results, the goal in this scenario is to improve the results using a

Table 3. Case Study A - overall results of the reverse engineering.

Element/scenario	A	B M = 20	B M = 10	B M = 5	C
XisEntities	34	34	34	34	38
XisEntityAssociations	45	45	45	45	45
XisEntityAssociations (Aggregations)	42	6	13	32	38
XisEntityAssociations (Many-to-many)	0	0	0	0	0
Explicit subclasses	0	0	0	0	0
Explicit superclasses	0	0	0	0	0
Implicit subclasses	0	0	0	0	8
Implicit superclasses	0	0	0	0	4
XisBusinessEntities	14	31	28	23	15
XisBusinessEntities Master Associations	14	31	28	23	15
XisBusinessEntities Detail Associations	42	6	13	32	38
XisBusinessEntities Reference Associations	12	36	27	21	13
XisEntityUseCases	14	31	28	23	15

trial and error approach. In this section we cover two distinct situations, namely aggregations and then generalizations.

Aggregation

This situation is focused on the assessment of the obtained results using different Simple Principal Entity configurations in a blind way, following the defined evaluation heuristics.

Regarding the Simple Principal Entities selection menu [7], we started by using the default value (20) to filter entities by the maximum number of rows that a table can have in the database. And then selected all those entities.

Moreover, a similar process is used in the following situations but with different numbers. To simplify we denote this number as M.

- **M = 20** - With this configuration, 6 aggregations were identified. However, regarding the Rule-1, none of the 6 aggregations that were correctly classified in the Scenario-A were now correctly identified, thus the number of False Negatives increased to 7. Moreover, the one-to-one association wrongly identified as aggregation in the Scenario-A, was not classified as an aggregation this time. Regarding the Rule-2, none of the configuration entities aggregated a main entity or an association entity. Due to the low number of identified aggregations, it only makes sense to test again with a lower M number.
- **M = 10** - With this configuration, 13 aggregations were found. In terms of the Rule-1, 2 aggregations were correctly identified, thus the number of False Negatives decreased to 5. This time, the one-to-one association was wrongly classified as an aggregation, since this time DocumentProject entity was not selected as Simple Principal Entity (number of False Positives is 1). Moreover, following the Rule-2, the result was the same as in the previous test. With

- this new value for M , the number of identified aggregations is still less than half of the total number of entities. Thus, we will decrease M once again.
- $M = 5$ - With this configuration, the number of aggregations increased to 32. Following Rule-1, the number of correctly identified aggregations increased by one, thus there were still 4 aggregations wrongly identified as simple XisEntityAssociations (False Negatives). The number of False Positives remained the same. Regarding Rule-2, this time one entity (Skill) had aggregation associations with main or association entities, likewise in the scenario without configurations. Moreover, since the number of entities with 5 or less attributes ($M = 5$) is only one (Process), it does not make sense to try lower M values, because the results would be the same as we got in the scenario without configurations.

With these results, we can say that without domain knowledge about the ProjectIT-Enterprise, in terms of aggregations, we would get the best result without using the Simple Principal Entities configuration in a blind way.

However, we think that the tests with this kind of configuration did not show interesting results, mainly due to the reduced application usage, which was reflected in the low amount of main entities rows, such as project and process. And, since our heuristic assumes that the number of rows of aggregated entities is greater or equal to the number of rows of the entities that aggregate them, and that the number of rows of Simple Principal Entities is usually a lot smaller, compared with others, we conclude that this configuration did not benefit the obtained results in this case study.

Generalization

This situation is focused on the identification of implicit generalizations and the assessment of the obtained results. In order to perform this evaluation we will activate Generalization discovery and use its configuration points [7]. Moreover, since our generalization evaluation heuristic tries to maximize both the number of subclasses and the number of inherited attributes, we will use our two options to aggregate entities (the third configuration in the Generalization Discovery) every iteration, i.e. every time we change other configuration points.

Table 4, summarizes the main results during this evaluation, taking into account, for each configuration, the number of generated subclasses (a), the sum of inherited attributes (b) and the application of such values in our evaluation function. From Scenario-A we could extract that the maximum number of subclasses that can be found is 31, and the sum of the maximum number of attributes that can be inherited is 51, so we can rewrite our function as:

$$Reis'(a, b) = 0.5\left(\frac{a}{31}\right) + 0.5\left(\frac{b}{51}\right) \quad (2)$$

We started our evaluation by using the default value for the minimum number of shared attributes (5). And then, from the obtained results we decided which configuration should be used in the next iteration.

Overall, from the first iteration, we got reasonable results ($Reis' = 0.18$), namely 4 subclasses and 12 inherited attributes were found. Then, due to a lower

Table 4. Case Study A - Scenario B - evaluation of generalization simulations.

Minimum # of shared attributes	Aggregated by	Ignored names	Subclasses (a)	Inherited attributes (b)	Reis'
5	Attributes	-	4	12	0.18
5	Entities	-	4	12	0.18
4	Attributes	-	4	12	0.18
4	Entities	-	5	10	0.18
3	Attributes	-	8	18	0.31
3	Entities	-	10	12	0.28
3	Attributes	Name, description	4	8	0.14
3	Entities	Name, description	4	8	0.14
2	Attributes	Name, description	6	10	0.19
2	Entities	Name, description	8	8	0.21

number of subclasses found, it only made sense to iterate with lower numbers for the minimum number of shared attributes. We, reduced to 4, and we got slightly similar results, so we decided to reduce again to 3, from which we got better results (Reis' = 0.31), namely 8 subclasses found and 18 inherited attributes, by aggregating our entities by the higher number of shared attributes. Then, we reduced to 2 and a combinational explosion happened, generating no results. However, we analyzed the inherited attributes from the last configuration successfully used, and we noticed that two attributes ("name" and "description") were inherited by almost every superclass, which led us to the decision of doing more iterations, this time ignoring such attributes. During these iterations, the best result we got (Reis' = 0.21) was slightly higher than the firsts we got, but in no way closer to our best one.

Scenario C: Extraction with Semi-guided Configuration. The main goal of this scenario is an attempt to improve the results obtained from the two previous configuration scenarios, by introducing some domain knowledge in the configuration parameters.

In terms of aggregations, we solely identified the configuration entities as Simple Principal Entities, from the previously generated Table 1. And as expected, with that configuration, we got the best results of all the configuration scenarios used, specifically by improving results of Rule-2. As stated before, no matter how much domain knowledge the user has, the False Negative and the False Positive problems identified can only be solved manually, since none of the available configurations can correct such issues.

Regarding generalization discovery, even with a good domain knowledge (e.g. average number of entity attributes), the user would always need to perform a similar approach as the blind one, to get good results in terms of implicit generalizations.

To get the best results overall, we had to select the known configuration entities as Simple Principal Entities and activate Generalization Discovery with at least 3 shared attributes, ordered by the higher number of shared attributes.

4.2 Case Study B: Social Security Application

During this research period, we worked in the TT-MDD-Mindbury/2015 project, whose main goal was to develop a real-world Social Security application on top of an existent legacy one. This was a good opportunity to apply our approach to a real-world application, already with some usage by its users. Due to privacy concerns for the client and the development company, details about this application must be kept confidential. For that reason, the analysis and evaluation of this case study are not as detailed as the Case Study A, only highlighting the most relevant results.

This project lasted for 12 months, allowing us to acquire a deep domain knowledge of this application and such knowledge is used to guide the reverse engineering process for this application, to reduce the number of iterations. We evaluate this application based on this knowledge, since the available documentation of this application was outdated.

In terms of its database, this application was designed with 187 tables, and most of its tables shared a common set of attributes, namely timestamps for those entities, etc. Thus, using the XIS-Reverse approach to find implicit generalizations (further XisEntityInheritance transformation rule [7]), can easily lead to a combinatorial explosion.

This evaluation only stresses 3 different configuration scenarios, namely a scenario without configuration; one blind configuration scenario, selecting every entity with 20 or less rows as Simple Principal Entity; and a semi-guided configuration scenario, based on the selection of Simple Principal Entities selection and on Generalization discovery.

Table 5, shows the overall results obtained from these three configuration scenarios based on the previously defined metrics.

Scenario A: Extraction Without Configuration. We found that many entities, that we could classify as Simple Principal Entities, were wrongly aggregating main entities and no explicit generalization was found. The obtained results were not accurate due to said fact and in general were wrong in terms of aggregations.

Scenario B: Extraction with Blind Configuration. This scenario was executed in order to evaluate if a user could get better results by selecting Simple Principal Entities in a blind way. In this experiment, every entity with 20 or less rows in the database was selected as Simple Principal Entity (132 entities). With our domain knowledge, we were able to identify that from those 132 entities, 14 were wrongly selected and 12 Simple Principal Entities were not selected since they had more than 20 rows (e.g. country). However the quality of the results

Table 5. Case Study B - overall results of the reverse engineering.

Element/scenario	A	B	C
XisEntities	168	168	176
XisEntityAssociations	224	224	224
XisEntityAssociations (Aggregations)	126	38	21
XisEntityAssociations (Many-to-many)	19	19	19
Explicit subclasses	0	0	0
Explicit superclasses	0	0	0
Implicit subclasses	0	0	16
Implicit superclasses	0	0	8
XisBusinessEntities	140	156	161
XisBusinessEntities Master Associations	140	156	161
XisBusinessEntities Detail Associations	82	23	15
XisBusinessEntities Reference Associations	124	190	215
XisEntityUseCases	140	156	161

increased drastically since, overall, most of the Simple Principal Entities (around 91%) were well identified using this approach.

Scenario C: Extraction with Semi-guided Configuration. With the domain knowledge, we had the ability to improve these results even more, by identifying every Simple Principal Entity (with the help of the available filters, due to the large number of entities to select). We knew the average number of attributes per entity and that some entities shared some properties, so we could reduce the number of iterations to obtain results in terms of implicit generalizations. With that, following a semi-guided configuration we got better results, not only in terms of aggregations that made more sense, but also in terms of implicit generalizations found.

Furthermore, during this research, this case study was used several times to support the evaluation of the development iterations. One of the issues that we encountered by using this complex system, besides the time to realize reverse engineering, was the combinatorial explosion that a Generalization discovery could easily trigger while comparing every entity and their attributes.

This issue can happen when there are a large number of entities which share identical attributes, leading to a large set of entities to be compared which can exponentially increase the amount of time and memory required to find generalizations. During our experiments, the aforementioned combinatorial explosion was usually stopped due to memory constraints of the EA application (usual Windows application constraints), which led to a crash of the application. All facts considered, the only solution to execute this feature in large domains (like this one), is to ignore some of the most used attribute names.

5 Interoperability with XIS* Frameworks

In this section, an analysis of the XIS-Reverse interoperability with a XIS* framework is performed, to assess how well both tools can work together. Figure 8 illustrates the main goal of this evaluation, which is to successfully use XIS-Reverse to generate XIS* models given a Legacy Application, and use those models to generate a New Application using a XIS* framework.

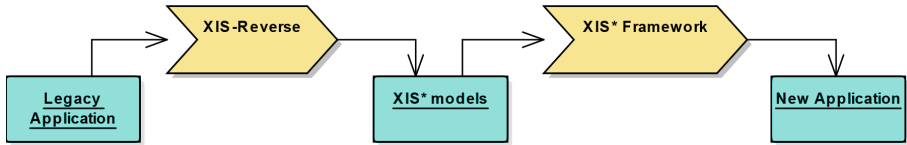


Fig. 8. Interoperability with XIS* frameworks.

For this evaluation we used the XIS-Web technology [8], since it is the most recent XIS* technology. In a nutshell, this framework supports the XIS-Web forward engineering process, which is applied to XIS-Web models. This is accomplished by following three steps: (1) Models validation; (2) M2M transformation; and (3) Model-to-Text (M2T) transformation. Step-1 uses a set of rules (built-in the framework) to validate the XIS-Web models. Then, step-2 generates the user-interface views, namely the Interaction Space and Navigation Space views. Finally, step-3 generates the target application's source code.

To perform this evaluation, we used the most recent version of the XIS-Web framework ⁷. Then, using the XIS-Web specifications from the ProjectIT-Enterprise (Sect. 4.1), that were extracted with the XIS-Reverse tool, we did our evaluation step by step:

Models Validation - In this step, we got one relevant error in the first validation, namely that XisEntities had to have at least one XisEntityAttribute. Since this error can easily occur, for example in subclasses, we believe that the only viable solution for this mismatch is to update the XIS-Web framework in order to omit this rule. Additionally, to bypass this problem, we added a fictitious XisEntityAttribute to those XisEntities and the second validation was successful.

M2M Transformation - Then we executed the model generation, which successfully produced the contents of the Interaction Space and the Navigation Space views.

M2T Transformation - In this step, we also got an error during code generation. The error stated that the same file was generated several times, i.e. some files were overwritten. This problem occurred due to the limitation of the XIS-Web framework to generate Interaction Space and the Navigation Space views

⁷ <https://github.com/MDDLingo/xis-web>.

for XisEntities with many-to-many relationships. Once again, the most logical solution to this problem is to extend the XIS-Web framework in order to support that scenario. However, to bypass this problem, we removed each many-to-many relationship and, for each one of them, we added an association entity (XisEntity linked to each of those entities through a one-to-many relationship) and after rerunning the M2T transformation, the M2T transformation was successful.

From these results, we can conclude that we can successfully use the XIS* specifications generated by the XIS-Reverse tool with the XIS-Web tool. Which means that it is possible to automate the generation of a new application by applying those tools to a legacy application.

6 Related Work Discussion

The continuous development of reverse engineering methodologies and tools has been crucial to mature this topic. More recently, MDE started to be applied to reverse engineering (MDRE), promoting a more systematic and flexible process. Likewise, MDRE approaches have been extended in order to completely reengineer a source application into a new target application through model-driven reengineering techniques.

This section overviews the most relevant research studies, covering data schema extraction and reverse engineering of databases. Moreover, those contributions are also compared with our approach.

6.1 Data Schema Extraction

The main properties of the research work analyzed in this subsection are shown in Table 6. This table specifies for each approach, its input, the existence of data schema extractors, if it extracts all table properties and its output. The last row categorizes our approach.

Gra2MoL [19] Text-to-Model (T2M) language and MoDisco [3] framework have been specially tailored for data schema extraction (model injection).

Gra2MoL is a domain specific language (DSL) to write transformations between any textual artefact which conforms to a grammar (e.g. source code) and a model which conforms to a target metamodel. On the other hand, MoDisco is a Java framework intended to facilitate the implementation of MDRE approaches. Regarding data schema extraction, MoDisco facilitates the implementation of discoverers (model injectors), and it currently provides discoverers for Java, JSP and XML. A drawback of both approaches is that they would require the definition of such transformations and discoverers, respectively, to extract the database schema.

Schemol [20] is another tool for injecting models. This tool allows injecting data stored into the database by specifying transformations that express the correspondence between the source database schema and the target metamodel.

Furthermore, in terms of database model injection (to ease this T2M transformation), it is possible to transform a database schema into a graphical representation using a variety of commercial and academic tools. DB-MAIN [21] and

Table 6. Classification of some related works on data schema extraction (adapted from [7]).

Approach	Input	Output	Schema extractors	Properties
Gra2Mol [19]	Any textual artefact	Any model	No	n.a.
MoDisco [3]	Several sources	Any model	No	n.a.
Schemol [20]	Data stored into DB	Any model	No	n.a.
DB-MAIN [21]	Several sources	GER	Yes (e.g. ODBC)	Yes
SQL2XMI [22]	SQL DDL schema	UML in XMI	Yes (only MySQL)	No
EA (XIS-Reverse)	Several sources	UML	Yes (e.g. ODBC)	Yes

SQL2XMI [22] are two examples of such academic tools. Firstly, DB-MAIN is a toolbox that supports database reverse engineering, and includes legacy database schemas extractors, through several sources such as ODBC drivers or SQL files. Secondly, SQL2XMI is entitled as a lightweight transformation of data models from SQL Schemas to UML-ER expressed in XMI. To our knowledge, this tool is still limited compared with DB-MAIN since it does not infer entity types or cardinalities, and for now it is only compatible with the MySQL implementation of the SQL data definition language (DDL).

To sum up, given that a set of existing tools already supports data schema extraction from several sources, without additional specification of transformations, we took advantage of such tools, specifically EA.

6.2 Reverse Engineering

A reverse engineering approach can be classified in several ways. Table 7 gives a properties summary of the analyzed research works. These properties include: analysis type, output, supported XIS-Reverse contributions (A - Aggregations Extraction, G - Implicit Generalization Extraction and V - Attribute Values Extraction), existence of tools automating the approach, automation level of those tools (regards to the reverse engineering stage), and tool extensibility.

Regarding reverse engineering techniques, several approaches have been proposed, which are usually distinguished by the particular application artefact used as main information source. The most relevant research works, following such distinction, are described below.

Schema analysis [23] is mainly focused on spotting similarities in names, value domains and representative patterns. This technique may help identify missing constructs (e.g. foreign keys). Additionally, the XIS-Reverse approach was influenced by the manual process specified in [23]. Making possible to semi-automatically and automatically identify generalizations and many-to-many associations, respectively.

Data analysis [24,25] uses content mined from a database. Firstly it can be used to analyze the database normalization and secondly to verify hypothetical constructs suggested by other techniques. Given the combinatorial explosion that can affect the first approach, data analysis technique is mainly used with the

Table 7. Classification of some works on reverse engineering (adapted from [7]).

Approach	Analysis	Output	Contrib.	Tools	Auto.	Ext.
[23]	Schema	OMT class diagram	G	No	n.a.	n.a.
[24]	Data	Extended ER	n.a.	No	n.a.	n.a.
NoWARs [25]	Data	Conceptual schema	n.a.	Yes	Semi	n.a.
RAINBOW [26]	Screen	ER model.	n.a.	Yes	n.a.	n.a.
[27]	Program (static)	Extended ER	n.a.	No	n.a.	n.a.
[28]	Program (static)	Object-Oriented class diagram	n.a.	No	n.a.	n.a.
[29]	Program (dynamic)	n.a.	n.a.	Yes	n.a.	n.a.
[30]	Program (dynamic)	n.a.	n.a.	Yes	n.a.	n.a.
Modisco [3]	Schema and program (static)	KDM or UML	n.a.	Yes	Auto.	Yes
Relational Web [31]	Schema	UML	n.a.	Yes	Auto.	Yes
DB-MAIN [21]	Schema and program (static)	GER	G	Yes	Semi	Yes
XIS-Reverse	Schema, data and program	XIS* and RSLingo's RSL	A;G;V	Yes	Semi	Yes

purpose of the second approach. In addition, our approach uses this technique in a similar way, however it is applied to classify associations.

Screen analysis [26] state that user interfaces can also be sources of useful information. These user-oriented views over a database may display spatial structures, meaningful names and, at runtime, data population and errors combined with data-flow analysis may provide information about data structures and schema properties; our approach did not consider this kind of analysis.

Static [27, 28] and Dynamic [29, 30] program analysis can easily give valuable information about field structure and meaningful names, or identifying complex constraint checking and foreign keys after a complex analysis. A main challenge of dynamic program analysis is the extraction of highly dynamic interactions between a program and a database. The analysis of SQL statements is one of the most powerful variants of source code analysis. Our approach uses static program analysis in the profiler log file, aiming to classify associations.

Additionally, a set of approaches, concerning the application of MDE, are also taken into account. Our analysis focused on their injection and reverse engineering stages.

As previously introduced, MoDisco MDRE framework [3] has a huge potential regarding the support of reverse engineering activities due to its generic and extensible properties. Besides its legacy application discoverers (model injectors), MoDisco also allows the definition of transformations and generators, responsible for restructuring and forward engineering tasks over the system models, respectively. Our approach could be implemented extending this framework, however that would require the definition from scratch of all the three stages (discoverers, transformations and generators) needed to produce the desired artefacts.

Polo et al. propose a method and a tool, called Relational Web [31] specially designed for database reengineering. The starting point is a relational database, whose physical schema is reverse engineered into a class diagram representing its conceptual schema. In the restructuring stage, the class diagram is manipulated by the user and then passed as input to the forward engineering step. Moreover, this tool supports the definition of new database managers to be used as input

and the implementation of new code generators. On one hand, this approach only uses as input the physical schema, and user knowledge, and its tool does not take advantage of the existing MDE techniques nor technologies. On the other hand, this approach defined foreign key's semantic extraction techniques, to identify inheritance relationships and associations, that were adapted and extended by our approach, in order to identify aggregations.

As previously stated, DB-MAIN [21] is a toolbox that offers a complete functionality to apply database reverse engineering. Regarding reverse engineering, DB-MAIN includes features such as extractors of legacy database schemas, transformations between schemas, data and code analysis tools, among others. This tool is one of the most mature ones, used for database reverse engineering, meaning that it includes several features that have been the result of a great number of research contributions from Namur University. DB-MAIN supports a lot of common transformations and extraction tools thus, a user with such tools can handle almost any needed transformation to create a good conceptual schema. However, this tool requires the user to apply all the needed transformations, meaning that the degree of automation achieved in our approach is higher. Also, DB-MAIN supports generalization representation, but once again it must be identified by the user.

Regarding the main contributions of this paper, we do not find any other approach that specializes associations (e.g. distinguishing between associations and aggregations), nor any approach that allows the extraction of attribute values and their representation into the target conceptual schema.

7 Conclusion

XIS-Reverse approach allows to automatically extract high-level models and specifications from legacy applications. This approach benefits from a flexible set of configuration points and new features not found in prior work, that allows to produce more detailed models and specifications, that overall will help the user get a better understanding of the application domain.

In terms of aggregations detection, at least when using a system with a good amount of data (usage), our heuristics can correctly identify those relationships and assist the user in obtaining better results by specifying Simple Principal Entities (with and without domain knowledge).

Regarding implicit generalizations discovery, our approach proved its ability to extract accurate results. However, it does not benefit much from user domain knowledge since several experiments with different configurations scenarios must be executed to find the best results. In the extreme scenario, if the user has a good understanding of each attribute of each entity, this feature should be disabled, since the identification of generalizations can easily be done manually.

Although not evaluated, we assume that extraction of attribute values, independently of the user domain knowledge level, can benefit the user if values from certain entity attributes can be extracted, giving him or her a better understanding of the entity role in the domain.

Overall, the evaluation results of XIS-Reverse’s novel features validate that XIS-Reverse can increase the knowledge and understanding extracted from a legacy information system.

Additionally, in terms of interoperability with the XIS-Web framework, despite the aforementioned errors (Sect. 5), which were probably found due to the size and complexity of the case study, overall, the produced XIS-Web specifications with the XIS-Reverse tool, are suitable to be used with the XIS-Web framework.

Regarding future work and considering the extensibility of the proposed process, we would like to evaluate the similarity between the combined results of pipelining XIS-Reverse with XIS-Web processes to generate a new application, and then compare it with the original legacy application. Additionally, we would like to extend the XIS-Reverse approach to support new input and output technologies, include more types of analysis in the reverse engineering process (e.g. screen), and use a divide-and-conquer approach to manage the complexity of identifying implicit generalizations (e.g. splitting sets of entities by their schema).

References

1. Garcia, J.M.E.: Requirements change management based on web usage mining. Ph.D. thesis, University of Porto (2016)
2. Chikofsky, E.J., Cross, J.H.: Reverse engineering and design recovery: a taxonomy. *IEEE Softw.* **7**(1), 13–17 (1990)
3. Bruneliere, H., Cabot, J., Dupé, G., Madiot, F.: MoDisco: a model driven reverse engineering framework. *Inf. Softw. Technol.* **56**(8), 1012–1032 (2014)
4. Canfora, G., Di Penta, M., Cerulo, L.: Achievements and challenges in software reverse engineering. *Commun. ACM* **54**(4), 142–151 (2011)
5. Ruiz, F., et al.: An approach for model-driven data reengineering. Ph.D. dissertation, University of Murcia (2016)
6. da Silva, A.R.: Model-driven engineering: a survey supported by the unified conceptual model. *Comput. Lang. Syst. Struct.* **43**, 139–155 (2015)
7. Reis, A., da Silva, A.R.: XIS-Reverse: a model-driven reverse engineering approach for legacy information systems. In: *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development, MODELSWARD*, vol. 1, pp. 196–207 (2017)
8. Seixas, J.: The XIS-web technology: a model-driven development approach for responsive web applications. M.Sc. dissertation, Instituto Superior Técnico, University of Lisbon (2016)
9. Baskerville, R.L.: Investigating information systems with action research. *Commun. AIS* **2**(3es), 4 (1999)
10. da Silva, A.R., Saraiva, J., Silva, R., Martins, C.: XIS-UML profile for extreme modeling interactive systems. In: *2007 Fourth International Workshop on Model-Based Methodologies for Pervasive and Embedded Software, MOMPES 2007*, pp. 55–66. *IEEE* (2007)
11. Ribeiro, A., da Silva, A.R.: XIS-mobile: a DSL for mobile applications. In: *29th Annual ACM Symposium on Applied Computing*, pp. 1316–1323. *ACM* (2014)
12. Ribeiro, A., da Silva, A.R.: Evaluation of XIS-Mobile, a domain specific language for mobile application development. *J. Softw. Eng. Appl.* **7**(11), 906 (2014)

13. Filipe, P., Ribeiro, A., da Silva, A.R.: XIS-CMS: towards a model-driven approach for developing platform-independent CMS-specific modules. In: MODELSWARD. SCITEPRESS (2016)
14. Caramujo, J., da Silva, A.R.: Analyzing privacy policies based on a privacy-aware profile: the facebook and linkedin case studies. In: 2015 IEEE 17th Conference on Business Informatics (CBI), vol. 1, pp. 77–84. IEEE (2015)
15. da Silva, A.R.: Linguistic patterns and styles for requirements specification: the RSL/business-level language. In: Proceedings of the European Conference on Pattern Languages of Programs. ACM (2017)
16. da Silva, A.R., Saraiva, J., Ferreira, D., Silva, R., Videira, C.: Integration of RE and MDE paradigms: the ProjectIT approach and tools. *IET Softw.* **1**(6), 294–314 (2007)
17. Pinto, M.A.P.: Gestão de Projectos com Processos Ágeis. M.Sc. dissertation, Instituto Superior Técnico, University of Lisbon (2010)
18. Martins, P.V., da Silva, A.R.: ProjectIT-enterprise: a software process improvement framework. In: Industrial Proceedings of the 17th EuroSPI Conference, pp. 257–266 (2010)
19. Izquierdo, J.L.C., Cuadrado, J.S., Molina, J.G.: Gra2MoL: a domain specific transformation language for bridging grammarware to modelware in software modernization. In: Workshop on Model-Driven Software Evolution (2008)
20. Díaz, O., Puente, G., Izquierdo, J.L.C., Molina, J.G.: Harvesting models from web 2.0 databases. *Softw. Syst. Model.* **12**(1), 15–34 (2013)
21. Hainaut, J.-L., et al.: Database evolution: the DB-MAIN approach. In: Loucopoulos, P. (ed.) *ER 1994*. LNCS, vol. 881, pp. 112–131. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58786-1_76
22. Alalfi, M.H., Cordy, J.R., Dean, T.R.: SQL2XMI: reverse engineering of UML-ER diagrams from relational database schemas. In: 15th Working Conference on Reverse Engineering, pp. 187–191. IEEE (2008)
23. Premerlani, W.J., Blaha, M.R.: An approach for reverse engineering of relational databases. In: 1993 Proceedings of Working Conference on Reverse Engineering, pp. 151–160. IEEE (1993)
24. Chiang, R.H., Barron, T.M., Storey, V.C.: Reverse engineering of relational databases: extraction of an EER model from a relational database. *Data Knowl. Eng.* **12**(2), 107–142 (1994)
25. Pannurat, N., Kerdprasop, N., Kerdprasop, K.: Database reverse engineering based on association rule mining. arXiv preprint [arXiv:1004.3272](https://arxiv.org/abs/1004.3272) (2010)
26. Ramdoyal, R., Cleve, A., Hainaut, J.-L.: Reverse engineering user interfaces for interactive database conceptual analysis. In: Pernici, B. (ed.) *CAiSE 2010*. LNCS, vol. 6051, pp. 332–347. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13094-6_27
27. Petit, J.-M., Kouloumdjian, J., Boulicaut, J.-F., Toumani, F.: Using queries to improve database reverse engineering. In: Loucopoulos, P. (ed.) *ER 1994*. LNCS, vol. 881, pp. 369–386. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58786-1_91
28. Di Lucca, G.A., Fasolino, A.R., De Carlini, U.: Recovering class diagrams from data-intensive legacy systems. In: 2000 Proceedings of the International Conference on Software Maintenance, pp. 52–63. IEEE (2000)
29. Cleve, A., Meurisse, J.-R., Hainaut, J.-L.: Database semantics recovery through analysis of dynamic SQL statements. In: Spaccapietra, S. (ed.) *Journal on Data Semantics XV*. LNCS, vol. 6720, pp. 130–157. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22630-4_5

30. Cleve, A., Noughi, N., Hainaut, J.-L.: Dynamic program analysis for database reverse engineering. In: Lämmel, R., Saraiva, J., Visser, J. (eds.) GTTSE 2011. LNCS, vol. 7680, pp. 297–321. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35992-7_8
31. Polo, M., García-Rodríguez, I., Piattini, M.: An MDA-based approach for database re-engineering. *J. Softw. Maint. Evol.: Res. Pract.* **19**(6), 383–417 (2007)



Formal and Virtual Multi-level Design Space Exploration

Letitia W. Li^{1,3}(✉), Daniela Genius²(✉), and Ludovic Apvrille¹(✉)

¹ Télécom ParisTech, Université Paris-Saclay, Biot, France
{letitia.li,ludovic.apvrille}@telecom-paristech.fr

² Sorbonne Universités, UPMC Paris 06, LIP6, CNRS UMR 7606, Paris, France
daniela.genius@lip6.fr

³ Institut VEDECOM, Versailles, France

Abstract. With the growing complexity of embedded systems, a systematic design process and tool are vital to help designers assure that their design meets specifications. The design of an embedded system evolves through multiple modeling phases, with varying levels of abstraction. A modeling toolkit should also support the various evaluations needed at each stage, in the form of simulation, formal verification, and performance evaluation. This chapter introduces our model-based engineering process with the supporting toolkit TTool, with two main design stages occurring at a different level of abstraction. A system-level design space exploration selects the architecture and partitions functions into hardware and software. The subsequent software design phase then designs and assesses the detailed functionality of the system, and evaluates the partitioning choices. We illustrate the design phases and supported evaluations with a Smart Card case study.

Keywords: Virtual prototyping · Embedded systems
System-level design · Telecommunications

1 Introduction

A systematic design methodology with supporting toolkit can help designers with the modeling and evaluation of the system, and involves supporting multiple design phases at varying levels of abstraction and different evaluation tools. The design of embedded systems is complicated by the need to design both its hardware and software components. Their design methodology can therefore be separated into two main phases.

A system-level design space exploration divides functions into hardware and software, based on system performance, safety and security requirements. Next, the software design phase includes the development of the detailed system functionality, and generation of code. However, since partitioning decisions are taken at a high level of abstraction – e.g., with highly abstracted hardware components –, it might be useful to validate – and possibly reconsider – partitioning choices during the software and hardware development phase.

Several works of research and tools have addressed system-level partitioning and evaluation of hardware platforms during the software development stage. However, the lack of integration between partitioning and system development makes it difficult to reconsider partitioning choices. Also, it is very common practice to test/execute software components on the local host, and to integrate them later on the target. Consequently, errors due to the interaction between hardware and software are discovered very late in the development cycle – e.g., during the integration phase. Unfortunately, these errors may lead to reconsidering partitioning decisions. To minimize time needed for a designer to re-do a partitioning, a toolkit should ideally minimize the manual work needed to take in consideration those errors and better connect the two abstraction levels.

Thus, our work focuses on the development of a fully integrated model-driven approach to handle both partitioning and software development. Our contribution supports both the selection of candidate hardware and software architectures, and a software development approach that allows designers to evaluate the relevance of the previously selected architectures early in the development process. Automated model transformation and verification techniques - formal verification, simulation, virtual prototyping - are supported for that purpose. Our contribution presents an easy-to-comprehend methodology integrating these two stages contained within a single modeling framework (*TTool*) [1]. Previous work [2] described our design process at multiple levels, but lacked detailed automated performance analysis regarding performance metrics such as latency. In Sect. 2, we present the related work of other system-level design toolkits. Section 3 describes our overall methodology. Section 4 details the Smart Card case study that is then used to exemplify the high-level design space exploration (Sect. 5) and the software component design (Sect. 6). Finally, we present discussion and perspectives on future work in Sect. 7.

2 Design Techniques for Embedded Systems

Many frameworks have been proposed for the design of embedded systems. They offer modeling capabilities at different levels of abstraction and using various approaches, such as Platform-Based Design, Model-Driven Engineering, etc. These tools offers model edition capabilities and can verify models with different simulation and verification tools. Some of them also target executable code generation.

2.1 Design Space Exploration Approaches (with Simulation and Formal Techniques)

Ptolemy [3, 4] proposes a modeling environment for the integration of diverse execution models, in particular hardware and software components. If design space exploration can be performed with Ptolemy, its first intent is the simulation of the modeled systems. The co-simulation facility of Ptolemy II is demonstrated in [5]. Their approach relies on both a System-C architecture model and

a functional model. The paper describes how to use different abstraction levels to model systems.

Virtual prototyping of MPSoC is often hampered by slow simulation. Among approaches generating SystemC code, the virtual prototyping of [6] generates code for the TLM (transactional) level, which is more efficient to simulate but less detailed. A team from KIT [7] proposes a methodology for fast parallel simulation, which is based on TLM and though with lost accuracy, even if clock cycles can be taken into account. MPSoCSim [8], recently presented, proposes OVP processor models to simulate NoC-based System-on-chip. Bus simulation is TLM2.0 based. We chose to perform our simulations on cycle accurate bit accurate level and use a simulator based on fully static scheduling [9], which makes it 10 to 20 times faster than the SystemC event-based simulator.

Capella [10] relies on Arcadia, a comprehensive model-based engineering method. It is intended to check the feasibility of customer requirements, called *needs*, for very large systems. Capella provides architecture diagrams allocating functions to components, and advanced mechanisms to model bit-precise data structures. Capella is however more business focused, and lacks formal verification capabilities.

In POLIS [11], applications are described as a network of state machines. Each element of the network can be mapped on a hardware or a software node. This approach is more oriented towards application modeling, even if hardware components are closely associated to the mapping process. Metropolis [12], an extension of POLIS, targets heterogeneous systems, and architectural and application constraints are closely interwoven. Metropolis is based on a meta-model of a *network of concurrent objects*, with a formal semantics. Applications are described in detail and simulated with the help of instruction set simulators (ISS). This approach is more oriented towards application modeling, even if hardware components are closely associated to the mapping process. While our approach uses Model-Driven Engineering, Metropolis uses Platform-Based Design.

Sesame [13] proposes modeling and simulation features at several abstraction levels for Multiprocessor System-on-Chip architectures. Pre-existing virtual components are combined to form a complex hardware architecture. Models' semantics vary according to the levels of abstraction, ranging from Kahn process networks (KPN [14]) to data flow for model refinement, and to discrete events for simulation. Currently, Sesame is limited to the allocation of processing resources to application processes. It models neither memory mapping nor the choice of the communication architecture.

The ARTEMIS [15] project originates from heterogeneous platforms in the context of research on multimedia applications in particular. It is strongly based on the Y-chart approach [16]. Application and architecture are clearly separated: the application produces an event trace at simulation time, which is then read in by the architecture model. However, behavior depending on timers and interrupts cannot be taken into account.

MARTE [17] shares many commonalities with our approach, in terms of the capacity to separately model communications from the pair application-

architecture. However, it intrinsically lacks a separation between control and message exchange. Even if the UML profile for MARTE adds capabilities to model Real Time and Embedded Systems, it does not specifically support architectural exploration. Other works based on UML/MARTE, such as Gaspard2 [18], are dedicated to both hardware and software synthesis, relying on a refinement process based on user interaction to progressively lower the level of abstraction of input models. However, such a refinement does not completely separate the application (software synthesis) or architecture (hardware synthesis) models from communication.

Saxena and Karsai [19] introduce an abstract design space exploration framework, and its integration into design space exploration solvers, thus paving the way for customized embedded systems explorations. They define metrics (e.g., memory size) that are related to WCET. On the contrary, DIPLODOCUS does not assume any WCET, but closely evaluates possible scenarios with simulation and formal verification techniques.

The capacity of languages and models to support abstractions for designing embedded systems is discussed in [20]. In particular, MARTE is evaluated against the Y-Chart scheme. Our papers enhance their discussion with the refinement between two abstraction levels (partitioning and prototyping).

2.2 Code Generation Approaches

Rhapsody [21] can automatically generate software, but not hardware descriptions from SysML. MDGen from Sodius [22] adds timing and hardware specific artifacts such as clock/reset lines automatically to Rhapsody models, generates synthesizable, cycle-accurate SystemC implementations, and automates exploration of architectures.

The Architecture Analysis & Design Language (AADL [23]), a standard from the International Society of Automotive engineer (SAE), allows the use of formal methods for safety-critical real-time systems in avionics, automotive among other domains. It comprises a textual and a graphical representation but does not a priori contain tool support for code generation, even if specific contributions proposes code generator for specific domains, e.g. for avionics systems. In that case, the generated code can be executed for within a specific platforms, for instance for ARINC653 systems. Similar to our environment, a processor model can have different underlying implementations and its characteristics can easily be changed at the modeling stage. Recently, [24] developed a model-based formal integration framework which endows AADL with a language for expressing timing relationships.

Bombieri et al. [25] propose a method ranging from system specification to code generation, with an intermediate HW/SW partitioning stage. Their method is compliant with SW components, device driver generation, a software wrapper – e.g., to handle interrupts – and High-level synthesis for HW components. While being more advanced on code generation issues, simulation and formal verification, as well as iterations between partitioning and prototyping is not addressed as deeply as in our contribution.

Batori [26] proposes a design methodology for telecommunication applications. From use cases, the method proposes several formalisms to capture the application structure (“interaction model”) and behavior (Finite State Machine) and for its deployment from which executable code can be generated. The platform seems limited to specific components (“Runes component”) – we could call it Specific Platform-Based design – and no design exploration seems possible. Additionally, the code generation process targets a real platform, and not a prototyping environment.

As we explain in the next section, our approach combines both HW/SW partitioning and software development and prototyping, with formal verification and simulation offered for most views and abstraction levels, including safety, performance and security evaluation.

3 Methodology

3.1 Modeling Phases

The advantages of our methodology lie in its support of multiple phases of the design process, and its ability to evaluate a design with a diverse range of tools. These advantages have allowed our methodology to be applied for the modeling of a wide range of real-world systems, including automotive systems, telecommunications, security protocols, etc. [27–29]. Our method relies on a set of UML/SysML views supported with the same environment/toolkit (as shown in Fig. 1. The method is organized as follows:

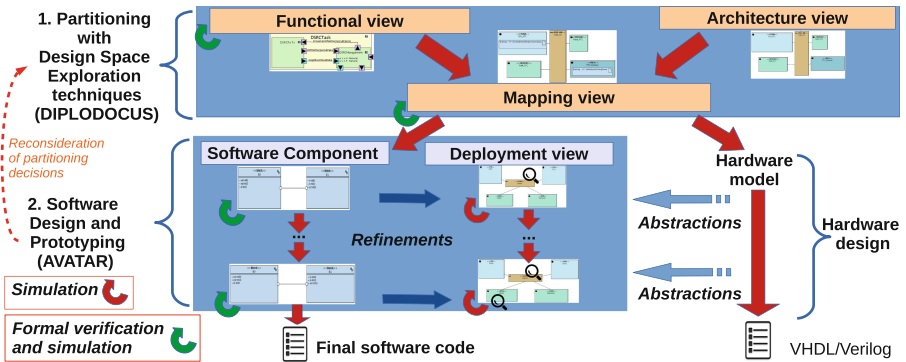


Fig. 1. Overall approach.

1. We start with system-level hardware/software partitioning based on design space exploration techniques. This phase contains three sub-phases: modeling of the functions to be realized by the system (“functional view”), modeling of the candidate architecture (“architecture view”) expressed as an assembly of highly abstracted hardware nodes, and the mapping phase (“mapping

- view”). A function mapped over a processor is considered a software function. On the contrary, a function mapped over a hardware accelerator corresponds to a custom ASIC. At this stage, we are concerned mostly with how communications and function affect the performance of a mapping, so we do not need to concern ourselves with the exact behavior of internal task behavior or contents of communications. Logical communication between functions are also expected to be mapped on a “communication path” consisting of buses, bridges, memories, Direct Memory Access controllers, networks-on-chip, etc.
2. Once a mapping has been decided, i.e., the system is fully partitioned between software and hardware functions, the design of the software and the hardware can start. Our approach offers software modeling while taking into account hardware parameters. Thus, a software component view is used to build the system software architecture and behavior, and a deployment view displays how the software components relate to the hardware components. The model of software and hardware components is more refined than in partitioning, which means that simulations and proofs are much more complex and take more time.

TTool [1], a free and open-source toolkit, supports the entire method with SysML diagrams. TTool includes UML/SysML diagram editors, compilers to perform model-to-specifications transformations, model-checkers and simulation engines.

3.2 Simulation, Verification and Prototyping

During the methodological phases, simulation and formal verification help to determine if safety, performance and security requirements are fulfilled. TTool offers a press-button approach for performing these proofs. Model transformations translate the SysML models into an intermediate form that is sent into the underlying simulation and formal verification toolkits - some of them are built into TTool, while others are third party toolkits. In all cases, backtracing to UML/SysML models is performed to better inform the users about the verification results.

During functional modeling – our highest abstraction level –, verification intends to identify general safety properties (e.g., absence of deadlock situations). At the mapping stage, verification intends to check if performance and security requirements are met. As researchers demonstrate the increasing number of hacks on embedded systems, it becomes important to detect their security flaws before mass-production. The security of communications depends on the architecture, as we explain in Subject. 6.2.

During software design, software components can be verified independently from any hardware architecture in terms of safety and security. For example, when designing a component implementing a security protocol, the reachability of the states and absence of security vulnerabilities can be verified. TTool support for integrated formal verification tools helps a designer ensure the safety and security of his/her design.

When the software components are more refined, it becomes important to evaluate performance. Since the target system is commonly not yet available, our approach offer two facilities. (i) A deployment view is used to map software components over hardware nodes. Their semantics is much more concrete – i.e., less abstract – than the one used for partitioning. (ii) A press-button approach can transform the deployment view into a SoCLib specification built upon virtual component models [30].

SoCLib is a public domain library of models written in SystemC, targeting shared-memory architectures based on the *Virtual Component Interconnect* protocol [31]. Hardware is described at several abstraction levels: TLM-DT (Transaction level with distributed time), CABA (Cycle/Bit Accurate), and RTL (Register Transfer Level). SoCLib also contains a set of performance evaluation tools [32,33]. CABA level simulation allows measurement of cache miss rates, latency of memory accesses and of any transactions on the interconnect, fill state of the buffers, taking/releasing of locks etc. in the context of video streaming and telecommunication applications [33].

A variety of low level performance measuring tools exist for SoCLib, as described in [32,34]. However, such approaches are purely based on simulating on the virtual prototype i.e. at a low level of abstraction, and lack the possibility to formally verify the application model and give it precise semantics. Moreover, they are more accessible to researchers than to engineers, nowadays very much at ease in the UML/SysML world. Hardware elements – i.e. *topcells* – are either described by hand, which is error-prone, or generated, making them not easily readable.

Since SoCLib hardware models are much more precise than partitioning models, precise timing and hardware mechanisms – e.g. cache miss – can be evaluated. If the performance results differ too greatly from the results obtained during the design space exploration stage – e.g., a cache miss ratio – then, the design space exploration shall be performed again to assess if the selected architecture is still the best according to the system requirements. If not, the definition of software components may be (re)designed. Once the iterations over the high-level design space exploration and the low level virtual prototyping of software components finished, software code can be generated from the most refined software model.

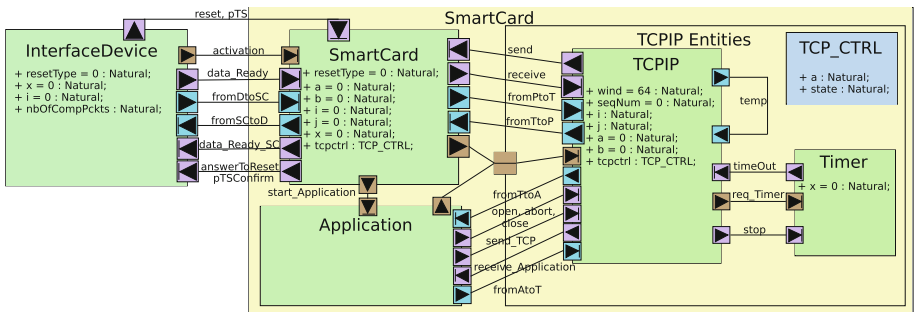


Fig. 2. Functional view (DIPLODOCUS) of the SmartCard application.

4 Case Study

Our methodology is illustrated by a “Smart Card” application. The smart card is meant to be plugged into a reader that exchanges information with the interior of the application by TCP formatted packets. The data transfer can be aborted, for example, because the smart card was unplugged. The reader (*InterfaceDevice*) signals the smart card to start, while the smart card controller handles the initialization of the other functions (e.g., the application and the network stack). In the next sections, we present modeling and analysis of the Smart Card application at the different design phases.

5 Partitioning with DIPLODOCUS

5.1 Models

The HW/SW partitioning phase, implemented in the DIPLODOCUS profile of TTool, models the abstract, high-level functionality of a system [35] and general architecture. It follows the Y-chart approach (as shown in the upper right section of Fig. 1), first modeling the abstract functional tasks (Application View), candidate architectures (Architectural View), and finally mapping tasks to the hardware components (Mapping View) [16]. Before the next stage, simulation and formal verification ensure that our design meets performance, behavioral, and schedulability requirements.

Application Modeling, Architectural Modeling, and Mapping are presented in detail in the rest of this section, using the smart card application as an example.

Figure 2 displays the functional view built upon 5 functional blocks: *InterfaceDevice* represents the interface with the reader and the internals of the smart card. *SmartCard* features the main controller. *Application* mostly models data exchanges that can occur with the reader. *TCPIP* and its *Timer* model the network stack. Exchanges between blocks are modeled with events, requests and data exchanges.

Application View. The Application View comprises of a set of communicating tasks, as shown in Fig. 2. The behavior of tasks is described abstractly. Functional abstraction allows us to ignore the exact computations and data processing of algorithms, and considers only computation complexity and data transfer size. Each individual task describes its abstract functional behavior using communication operators, computation elements, and control elements. Data abstraction allows us to consider only the size of data sent or received, and ignore details such as type, values, or names. On the Component Design Diagram, an extension of the SysML Block Instance Diagram, the designer specifies the list of tasks, and within the task, attributes and ports indicating communication.

Architectural View. The architectural model (consider only hardware components of Fig. 3, i.e. without the artifacts) displays the underlying architecture as a network of abstract execution nodes, communication nodes, and storage

nodes. Execution nodes consist of CPUs and Hardware Accelerators, defined by parameters for simulation. All execution nodes must be described by data size, instruction execution time, and clock ratio. CPUs can further be customized with scheduling policy, task switching time, cache-miss percentage, etc. Figure 4 shows processor parameters. Communication nodes include bridges and buses. Buses connect execution and storage nodes for task communication and data storage or exchange, and bridges connect buses. Buses are characterized by their arbitration policy, data size, clock ratio, etc., and bridges are characterized by data size and clock ratio. Storage nodes are Memories, which are defined by data size and clock ratio.

Mapping View. Mapping partitions the application into software and hardware and also specifies the location of their implementation and of their communication on the architectural model. A task mapped onto a processor will be implemented in software, and a task mapped onto a hardware accelerator will be implemented in hardware. The exact physical path of a data/event write may also be specified by mapping channels to buses and bridges. More complex communication schemes can be modeled with another view, which is part of recent work [36]. The mapping of Fig. 3 shows that the InterfaceDevice is mapped to a specific hardware execution node, while TCPIP, SmartCard and Application

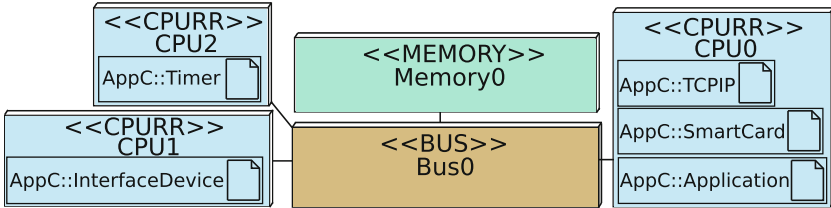


Fig. 3. Mapping view (DIPLODOCUS).

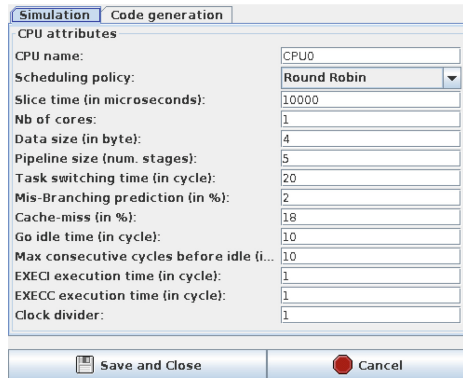


Fig. 4. Adapting processor parameters in DIPLODOCUS.

are mapped to a general purpose processor - actually, the main microcontroller of the smart card. Also, the timer is implemented with a dedicated execution node.

5.2 High-Level Simulation and Verification

Simulation of DIPLODOCUS partitioning specifications involves executing tasks on the different hardware elements. Each computation transaction executes for a variable time depending on execution cycles, CPU parameters and bus/memory behavior when transactions require data exchanges. The simulation shows performance results like bus usage, CPU usage, execution time, etc. Results are backtraced to the different views, with an example shown in Fig. 5. One can notice the high average load of the main microcontroller (91%). Also, TTool can generate a *vcd* trace to view detailed bus/CPU activity in gtkwave of a single execution sequence. TTool can also assist the user by automatically generating all possible architectures and mappings, and summarizes performance results of each possible mapping. Users are provided with the “best” architecture under specified criteria, such as minimal latency or bus/CPU load.

For a given mapping, the user can also generate the system reachability graph. The entire graph along with an enhanced excerpt is given in Fig. 6. All paths are terminated with a red state. The last actions before each red or termination state specifies the number of cycles corresponding to the path leading to that termination state. For example, the termination state “84” is preceded by an action “allCPUsTerminated<166>” which means that this system path contains 166 cycles.

TTool also makes it possible to list all termination or deadlock states (see Fig. 7): the graph contains 10 terminations states with a duration in number of cycles ranging from 20 to 247. In the shortest path, the connection was aborted after a few exchanges. On the contrary, in the longest execution path, the smart card exchanges several TCP packets. These timings are to be confirmed with more concrete software and hardware components in the design stage.

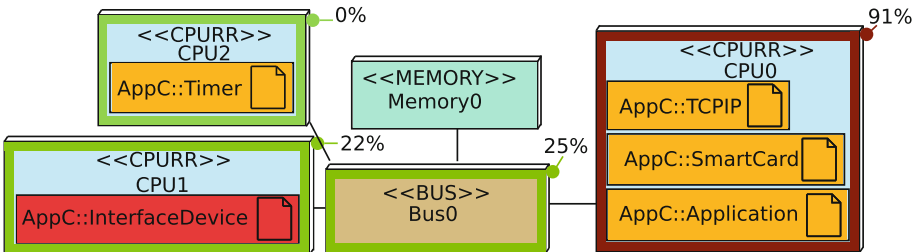


Fig. 5. Load of CPUs and buses after a simulation.

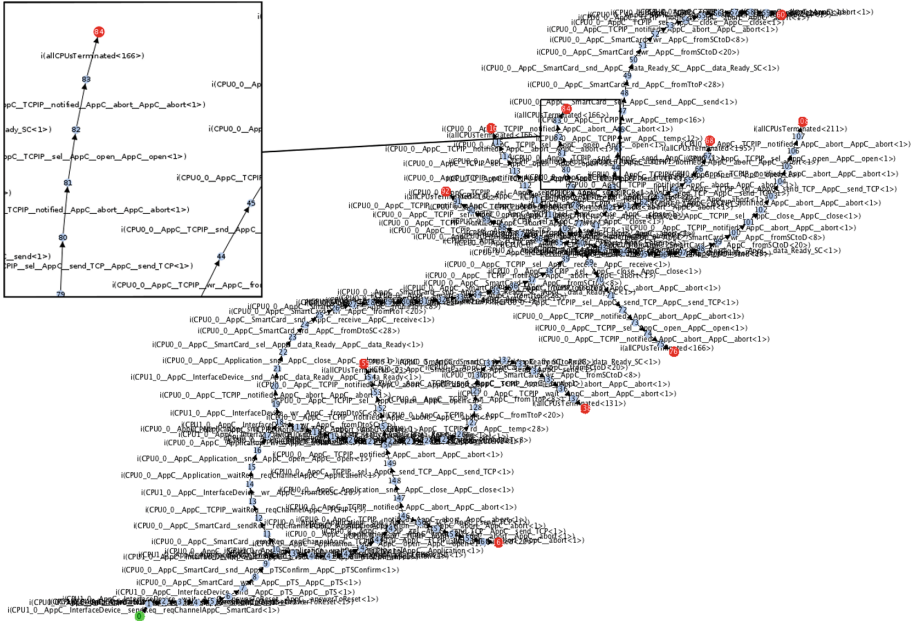


Fig. 6. Reachability graph of the mapping view. (Color figure online)

6 Software Design with AVATAR/SoCLib

Once partitioning is complete, the AVATAR methodology [37] allows the user to design the software, perform functional simulation and formal verification, and finally test the software components in a virtual prototyping environment. Where partitioning models represent an algorithm as an abstract execution spanning a duration, the software design models details of algorithms, including their attributes, int/float operations, control operators, etc.

Shortest Paths		Longest Paths	
General info.		Statistics	
States	(origin, action)		
108	(107, i{allCPUsTerminated<211>})	[0]	-- i{CPU1_0_AppC
116	(115, i{allCPUsTerminated<166>})	[0]	-- i{CPU1_0_AppC
138	(137, i{allCPUsTerminated<131>})	[0]	-- i{CPU1_0_AppC
155	(154, i{allCPUsTerminated<23>})	[0]	-- i{CPU1_0_AppC
161	(160, i{allCPUsTerminated<20>})	[0]	-- i{CPU1_0_AppC
60	(59, i{allCPUsTerminated<237>})	[0]	-- i{CPU1_0_AppC
68	(67, i{allCPUsTerminated<195>})	[0]	-- i{CPU1_0_AppC
76	(75, i{allCPUsTerminated<247>})	[0]	-- i{CPU1_0_AppC
84	(83, i{allCPUsTerminated<166>})	[0]	-- i{CPU1_0_AppC
92	(91, i{allCPUsTerminated<166>})	[0]	-- i{CPU1_0_AppC

Fig. 7. List of termination states in the reachability graph. The number of cycles on each path is given along with the last action before a termination state.

6.1 Software Components

Figure 8 shows the software components of the smart card case study modeled using an AVATAR block diagram. These modeling elements have been selected as software elements during the previous modeling stage (partitioning). Software components are grouped into the different applications running on the Smart Card using a hierarchical block called *SmartCard*.

- **Interface Device** initiates the connection and then communicates with the Smart Card.
- **SmartCard Controller** manages communication between the Interface Device, application, and TCPIP.
- **Application** communicates with the TCPIP application and sends and receives packets.
- **TCPIP** manages the TCP connection.
- **TCPPacketManager** manages packet transmission and storage.

The AVATAR model can be functionally simulated using the integrated simulator of our toolkit, which takes into account temporal operators but completely ignores hardware, operating systems and middleware. While being simulated, the model of the software components is animated. This simulation aims at identifying logical modeling bugs. Figure 9 shows the state machine of the Smart Card Controller, and Fig. 10 shows a visualization of a generated sequence diagram. Also, a reachability graph can be generated and analyzed.

6.2 Formal Verification

As previously described, TTool includes its own formal verification tools to e.g. generate a reachability graph, minimize the graphs, and check if a given reachability of liveness property is satisfied.

Alternatively, UPPAAL [38] may also be used from TTool to evaluate safety and liveness properties. UPPAAL is a model checker for networks of timed automata, the behavioral model of a system to be verified is first translated into a UPPAAL specification to be checked for desired behavior. For example, UPPAAL may verify the lack of deadlock, such as two threads both waiting for the other to send a message. Behavior may also be verified through “Reachability”, “Leads to”, and other general statements. The designer can indicate which states in the Activity Diagram or State Machine Diagram should be checked if they can be reached in any execution trace. “Leads to” allows us to verify that one state must always be followed by another. Other user-defined UPPAAL queries can check if a condition is always true, is true for at least one execution trace, or if it will be true eventually for all execution traces. These statements may be entered directly on the UPPAAL model checker, or permanently stored on the model as pragma to be verified in UPPAAL.

For example, for our case study, we can verify that the TCP Packet Manager is capable of sending the storePacket signal, that the Application can abort and thereby stop, and the Smart Card Controller can send a packet. Figure 11 shows

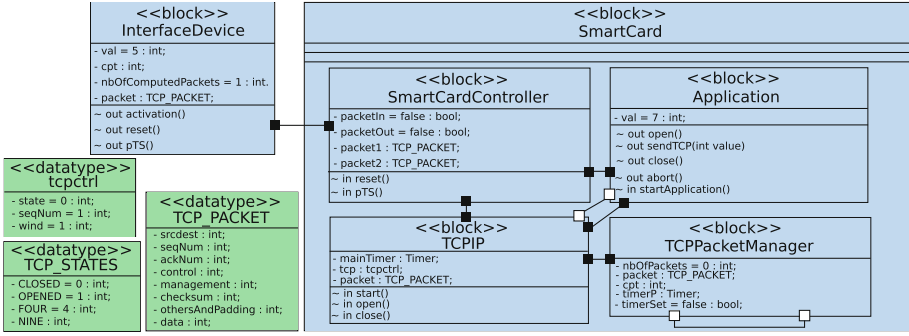


Fig. 8. Avatar block diagram.

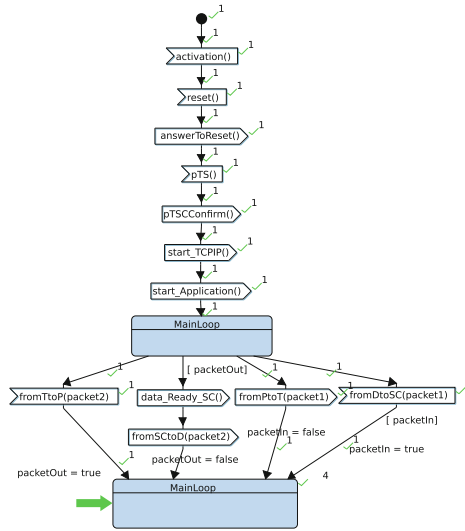


Fig. 9. High level simulation: annotated automaton.

the UPPAAL verification window which allows the user to customize which queries to execute, and then returns the results regarding whether each query is satisfied or not. In our example, the three states we queried are all reachable.

Formal verification of security is performed using ProVerif, a verification tool operating on pi-calculus specifications [39]. A ProVerif specification consists of a set of processes communicating on public and private channels. Processes can split to create concurrently executing processes, and replicate to model multiple executions (sessions) of a given protocol. Cryptographic primitives such as symmetric and asymmetric encryption or hash can be modeled through constructor and destructor functions. ProVerif assumes a Dolev-Yao attacker, which is a threat model in which anyone can read or write on any public channel, create new messages or apply known primitives.

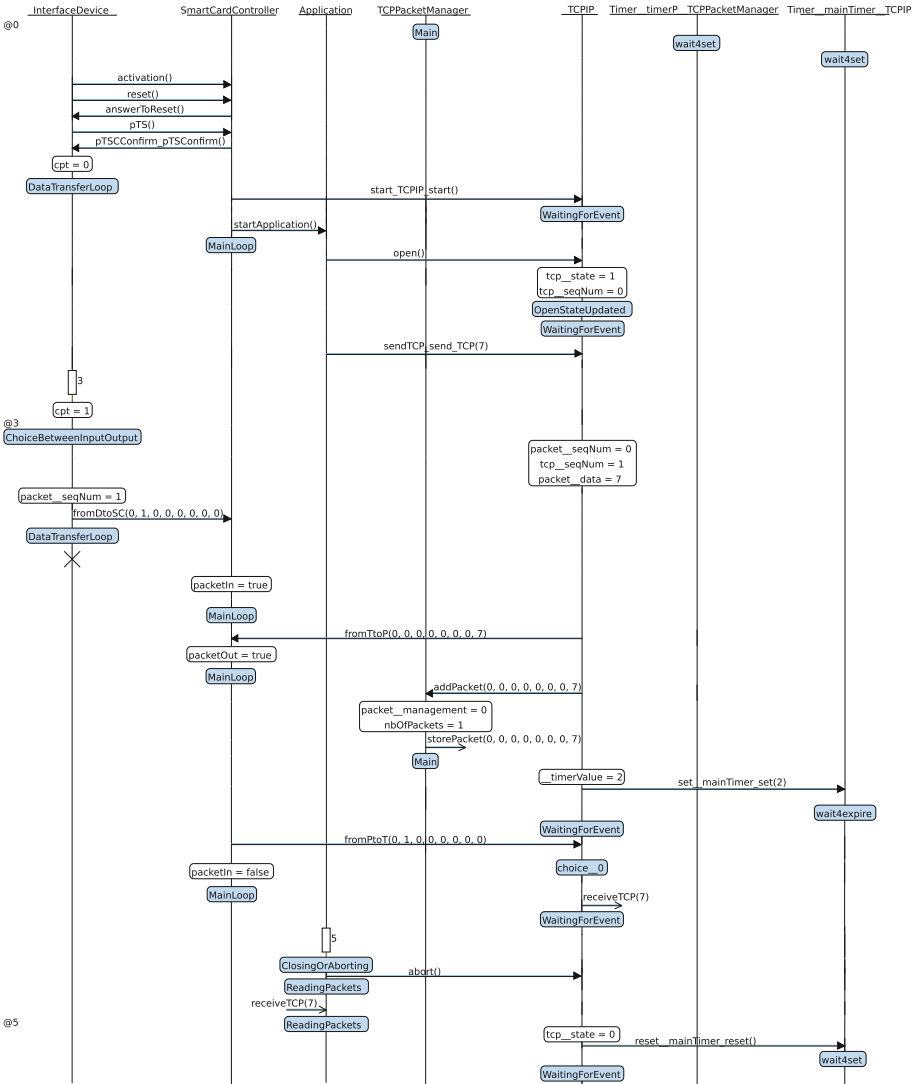


Fig. 10. High level simulation: generated sequence diagram.

ProVerif verifications query the properties of *reachability*, *confidentiality*, and *authenticity*. Reachability of an element (within the Activity Diagram or State Machine) determines if there exists an execution trace of the model in which this element is reached. Confidentiality of data refers to if the attacker can recover that data by listening and sending messages, and performing computations. Authenticity determines if the data received during a message exchange is necessarily the same as the data sent.

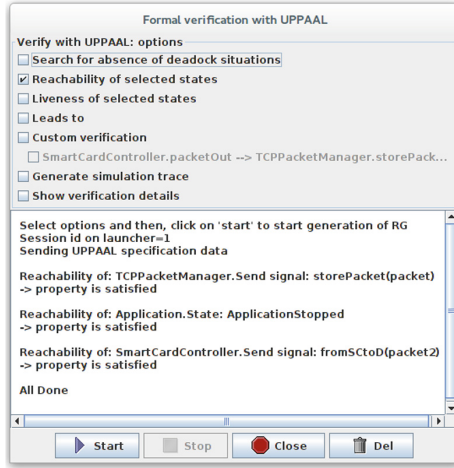


Fig. 11. UPPAAL formal verification.

In DIPLODOCUS models, security modeling and verification determines the security mechanisms required to secure critical data based on an architecture and mapping, and also impact on performance due to the added security. Certain architectural buses can be modeled to be accessible to an attacker. Abstract security operators then model the encryption/decryption of channel data and the impact of security on performance. Recent work [40] describes how the architecture and mapping selected during HW/SW Partitioning affects the security of communications, and security-related operations impacts the safety and performance of a system.

AVATAR models describe the detailed implementations of security mechanisms, and verifies the security of critical attributes [41]. The security verification determines the confidentiality of keys and specific attributes, the authenticity of encrypted exchanges over public channels accessible to an attacker, and the ability of an encryption algorithm to terminate correctly.

6.3 Prototyping

To prototype the software components with the other elements of the destination platform (hardware components, operating system), a user must first map them to a model of the target system. Mapping can be performed using the new deployment features introduced in [27]. Our toolkit thus supports use of AVATAR Deployment Diagrams. It features a set of hardware components, their interconnection, tasks, and channels.

In the partitioning phase, an architecture with two CPUs was selected. Tasks destined to become software tasks are mapped onto the CPUs, which is the case for all tasks in our example; it is also possible to realize other tasks as hardware accelerators. Now, in the prototyping phase, things are different since the

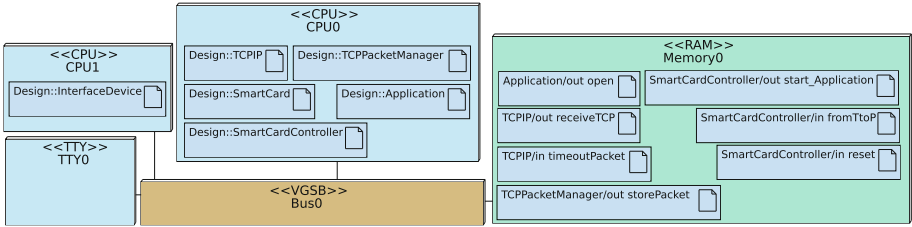


Fig. 12. Smart card deployment diagram.

AVATAR models includes only software tasks that are thus mappable only on general-purpose processors. Consequently, each hardware accelerator of the prototyping platform in SoCLib needs to be specifically developed. Which requires a significant effort. We do not consider that case because the smartcard is fully software implemented.

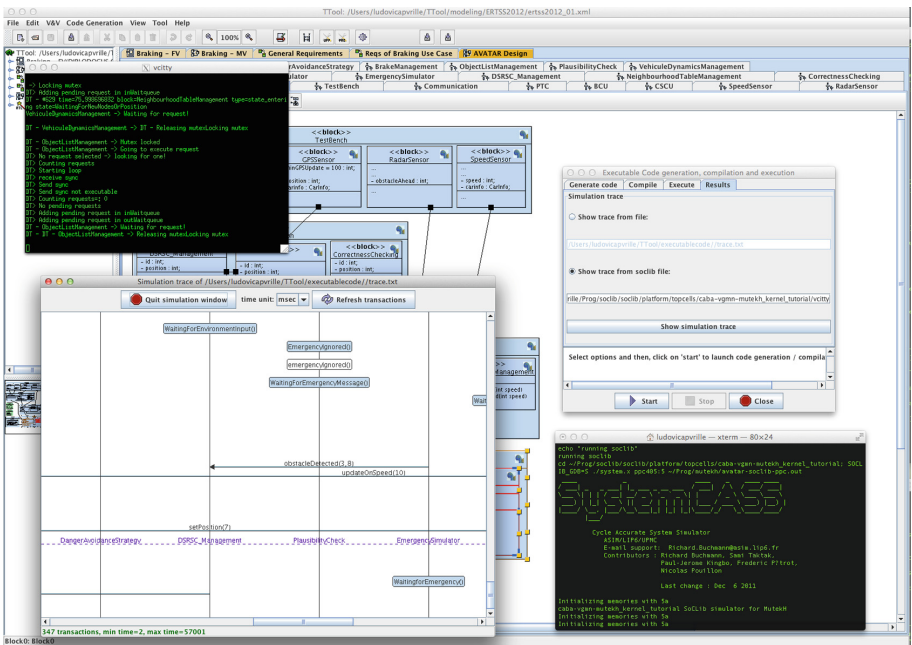


Fig. 13. AVATAR/SoCLib prototyping environment in TTool.

If the user has to explicitly model several properties pertaining to mapping, e.g. CPUs and memories parameters, the simulation infrastructure and interrupt management are added transparently to the top cell during the transformation into a SoCLib platform.

Figure 12 shows the Deployment Diagram, containing two CPUs, one memory bank and one TTY. The *InterfaceDevice* block is mapped onto CPU1, and the other five blocks are mapped onto CPU0. Each signal between AVATAR blocks is translated into a software channel mapped to on-chip RAM (for more detail, see [42]). In the case study, there are twenty-nine such signals, translated into twenty-nine SoCLib channels, which are all mapped on the single RAM, also containing the AVATAR runtime and the operating system.

From the Deployment Diagram, a SoCLib prototype is generated as described in [27]. This prototype consists of a SystemC top cell, the embedded software in the form of POSIX threads compiled for the target processors, and the embedded operating system [43]. Figure 13 from [2] shows an overview of the prototyping tool, with the simulation trace, code generation, and SocLib windows, and model in the back.

CPU attributes	
CPU name:	CPU1
Nb Of IRQs :	6
Nb of inst. cache ways:	2
Nb of inst. cache sets:	16
Nb of inst. cache words:	4
Nb of data cache ways:	2
Nb of data cache sets:	16
Nb of data cache words:	4

Save and Close Cancel

Fig. 14. Panel for varying cache associativity in SoCLib prototype.

6.4 Capturing Performance Information

We now show how performance information can be obtained by running simulations with the SoCLib virtual prototype of the SmartCard use case. The experiments shown here use a MP-SoC based on two general purpose PowerPC 405 processor cores running with 800 MHz. Later on, we plan to rely on a microcontroller, which would be more realistic for the SmartCard example. As a central interconnect, we use the VCI Generic Serial Bus (VGSB).

Although accelerated using the technique described in [9], the cycle accurate bit accurate (CABA)-level simulation is quite slow. It allows however detailed measurement of per-processor cache miss rates, latency of any transaction on the interconnect, etc. Since SoCLib hardware models are much more precise than the ones used at the design space exploration level, precise timing of the use of hardware mechanisms such as locks can be evaluated. However, these evaluations take considerable time compared to high-level simulation/evaluation.

As previously stated, the SoCLib prototype allows a designer to evaluate each processor separately, which is particularly useful for detecting unbalanced CPU loads, indicated by the Cycles per Instruction (CPI) metric.

In the following three paragraphs, we investigate three performance metrics: CPI, cache misses and latencies.

CPI. An overview of performance problems can be obtained using the numbers of Cycles per Instruction (CPI). It represents all phenomena that can slow down execution of instructions by the processors, such as memory access latency, interconnect contention, overhead due to context switching, etc.

Using these metrics, we can observe that CPU0 has a high average load – this issue was similarly noticed during the partitioning stage. Figure 15 shows that this CPU is far more challenged than CPU1 containing only the InterfaceDevice. The reason for this is due to the fact that implementing the semantics of synchronous channels requires a central request manager. Requests are stored in waiting queues for synchronous communications, in order to be canceled when they became obsolete. Requests that observe a delay before execution have to be waken up. Future work will address a better distribution of this functionality, called the *AVATAR runtime*, over the entire MPSoC architecture.

We also observe that adding cache associativity does not automatically improve the CPI. The application is characterized by an uneven mixture of small accesses (for example, *open* or *abort signals* which take one byte), and accesses to data of *packet* type which, as can be seen in the *Data Type* Block of the Block Diagram of Fig. 8, are composed of eight integers.

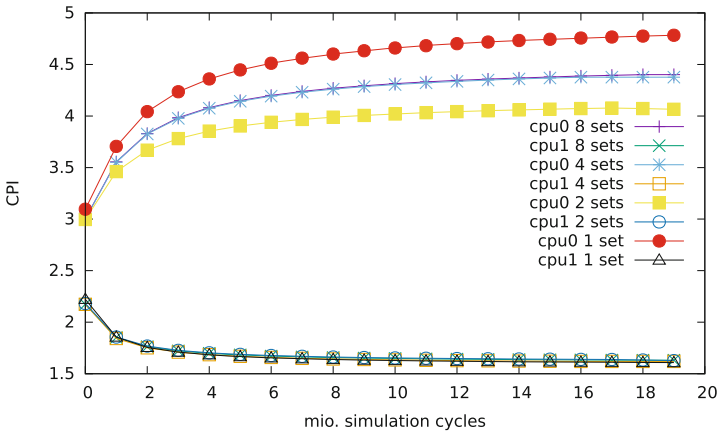


Fig. 15. CPI measured at CABA level.

Cache Misses. One important parameter of the CPU used in DIPLODOCUS partitioning is the overall cache miss rate, which is initially estimated to be 18% in DIPLODOCUS (see line *Cache-miss* in Fig. 4). While the estimate of

cache misses includes both data and instruction cache misses, we measure them separately. Instruction cache miss rates will be higher for the cache of CPU0 because the central request manager runs on this CPU, as noted in the previous paragraph.

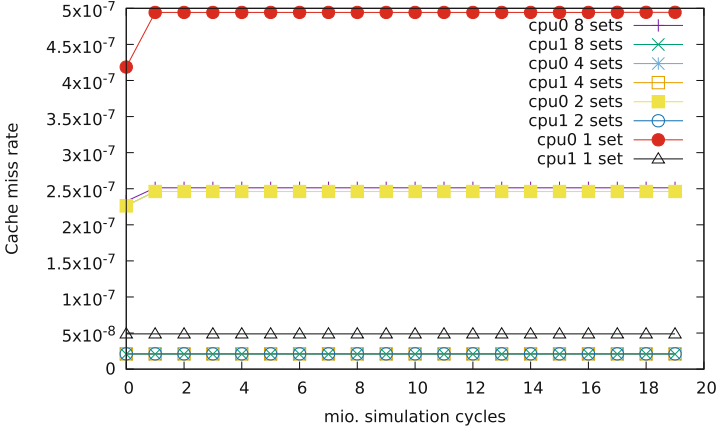


Fig. 16. Data cache miss rates measured at CABA level.

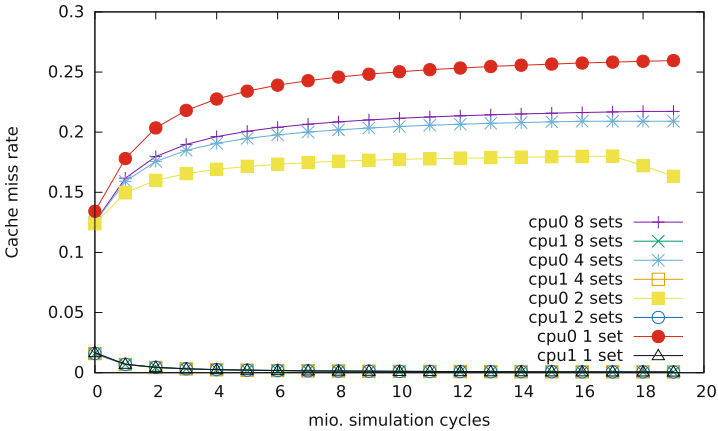


Fig. 17. Instruction cache miss rates measured at CABA level.

We vary associativity of both caches for the same cache size. Figure 14 shows the processor parameters of the Deployment Diagram. Parameters are the degree of associativity (*instruction/data cache ways*, in the Figure set to 2), the number of lines in the cache (*instruction/data cache sets*, here set to 16) and the number of words in a line (*instruction/data cache words*, here 4). Figure 16 shows the

data cache miss rates, and Fig. 17 shows instruction cache miss rates for set associativities of 1, 2, 4 and 8, using the same overall cache size, and same block size.

We observe clearly that from two cache sets onwards, cache misses are divided by two. The improvement is still around 30% for the instruction cache; we also note that CPU1 is much less challenged, as already shown for the CPI. In the worst case of a direct mapped cache, we have an instruction cache miss rate of 25% on the cache of CPU0, less than 1% for CPU1 (which essentially contains the interface and has a very small memory footprint). Thus, we can provide significantly more useful detail for a hardware implementation.

A first exploration presented in [2] for a different case study showed that cache misses can only be imprecisely estimated at the DIPLODOCUS model. However, that case study lacked the detailed modeling that we present here, both at the DIPLODOCUS and the AVATAR level. The Smart Card case study remedies this shortcoming.

We can now go back to the DIPLODOCUS level and customize the CPU by adapting the cache miss rate (Fig. 4): we were thus able to check that the partitioning result is still the same.

Latencies. In previous work [2], performance results were limited to those obtained using the hardware counters of the SoCLib modules. A recent update to TTool added support for automatically measuring latencies for channel transfers/between events during simulation. Activity elements can be marked as potential checkpoints on the model.

Events and channels in DIPLODOCUS both translate into signals in AVATAR. The left side of Fig. 18 shows a DIPLODOCUS activity diagram for the *Application* task, with two checkpoints set, one on the *open* and the *send_TCP* event. On the right side, it shows the timed automaton of the *Application* block. Again, we place one checkpoint on the *open* and another on the *send_TCP* signal.

The latencies panel of DIPLODOCUS is shown in Fig. 19. Our toolkit allows the user to choose which checkpoints he or she would like to analyze, and then displays the minimum, maximum, and average latencies in execution cycles between those two checkpoints.

On the MPSoC prototype for which the code is generated from AVATAR, latencies can be determined by hardware counters added to the SoCLib models. These counters allow identification of the processor that triggered the transfer, but not on which of the communication channels it took place. A recent improvement integrated the SoCLib logging mechanism presented in [34]. Thus, the MPSoC platform is enhanced with spies that can record all transfers on the interconnect, retrieve the names of software objects from the loader, and match them to the steps of the channel access protocol. This module is added to the VCI interface and does not impact performance results. Thus, we can now measure latencies on the MPSoC platform that are due to contentions on the interconnect, to the time spent waiting to obtain a lock, etc.

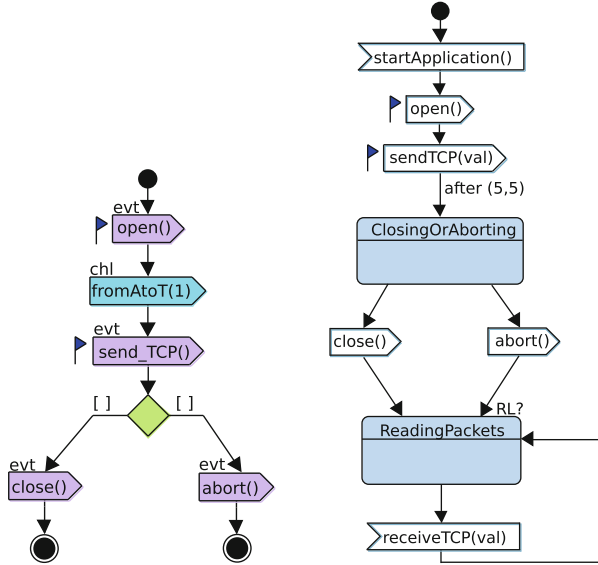


Fig. 18. Latency checkpoints (left) DIPLODOCUS (right) AVATAR.

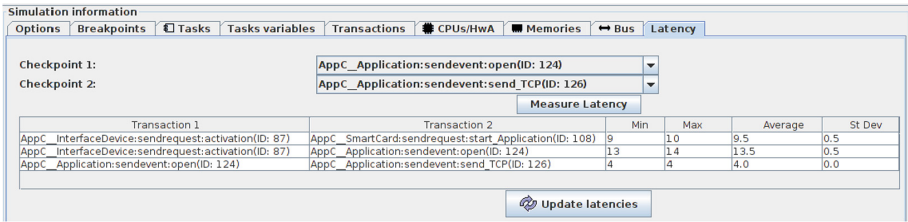


Fig. 19. DIPLODOCUS latencies panel.

Table 1 shows the latencies for selected channels corresponding to four signals in AVATAR. While *reset*, *start* and *open* signals have no parameters, *fromTtoP* conveys a packet (eight bytes in the case study).

Table 1. Latencies (milliseconds) for DIPLODOCUS simulation and SoCLib prototype.

AVATAR signal	DIPLODOCUS		MPSoC	
	Min	Max	Min	Max
SmartCardController_reset__InterfaceDevice_reset	2	2	0.56	0.64
SmartCardController_start_Application__Application_startApplication	4	4	0.56	0.58
Application_open__TCPIP_open	4	4	0.56	0.59
SmartCardController_fromTtoP__TCPIP_fromTtoP	38	75	1.6	1.7

As these first results show, there is no apparent correlation between the latency measured on the MPSoC platform and the latency obtained by DIPLODOCUS simulation. In fact for code generated from AVATAR running multiprocessor platform, cache effects, contention on the interconnect and others have to be taken into account. In particular, the storing and retrieving time of packets varies strongly. We are currently working on establishing correlations where this is possible, together with even more in-depth performance evaluation. It would be important to extend the latency measurement capability to AVATAR simulations, which should relate more closely to tests in SoCLib.

Other Performance Metrics. As can be seen in the CPU attributes window of Fig. 4, our toolkit potentially allows a designer to improve estimates of several more hardware parameters like branch misprediction rate and go idle time. Channels play a particular role: for asynchronous channels, they may overflow or otherwise be empty most of the time, slowing down or even blocking the application. Dimensioning of the channels is thus an important issue. Better understanding of the state of communication channels (fill state, evolution of read and write operations over time etc.) will be achieved by integrating new performance measuring functionality based on the work described in [32].

7 Discussion and Future Work

Our approach integrates both system-level design space exploration and the design and prototyping of refined software components in the same toolkit. Using a Smart Card case study, we show how the different metrics can easily be evaluated at the push of a button in the two abstraction levels. In particular, transformations of the software component model mapped onto a deployment diagram help precisely determine the CPI, as well as the finer metrics as cache miss rate and latencies of the application. From these evaluations, partitioning choices can be confirmed or invalidated.

We are currently working on a more complete method to determine and compare performance metrics in particular latencies at the AVATAR and DIPLODOCUS level and hope to establish correlations. Relating partitioning and software level simulations may also help us determine the accuracy of the estimates of execution duration of functions in partitioning.

The close integration of partitioning and software design facilitates the invalidation of partitioning decisions. The current backtracing to models assists the engineer in investigating how to better partition the model or to reconsider the software components. Ideally, once an invalidation has been encountered, it would be helpful for the toolkit to automatically suggest another partitioning. We propose increased automation as part of our future work, to better support designers between the different stages of the design process.

References

1. Apvrille, L.: Webpage of TTool (2015). <http://ttool.telecom-paristech.fr/>
2. Genius, D., Li, L.W., Apvrille, L.: Model-driven performance evaluation and formal verification for multi-level embedded system design. In: Conference on Model-Driven Engineering and Software Development (Modelsward 2017), Porto, Portugal (2017)
3. Buck, J., Ha, S., Lee, E.A., Messerschmitt, D.G.: Ptolemy: a framework for simulating and prototyping heterogeneous systems. In: Readings in Hardware/Software Co-design, pp. 527–543 (2002)
4. Ptolemaeus, C.: System Design, Modeling, and Simulation: Using Ptolemy II. Ptolemy.org, Berkeley (2014)
5. Kim, H., Guo, L., Lee, E.A., Sangiovanni-Vincentelli, A.: A tool integration approach for architectural exploration of aircraft electric power systems. In: IEEE Proceedings of the 1st International Conference on Cyber-Physical Systems, Networks, and Applications, pp. 38–43. IEEE (2013)
6. Zimmermann, J., Stettmann, S., Viehl, A., Bringmann, O., Rosenstiel, W.: Model-driven virtual prototyping for real-time simulation of distributed embedded systems. In: SIES, pp. 201–210. IEEE (2012)
7. Roth, C., Bucher, H., Reder, S., Buciuman, F., Sander, O., Becker, J.: A SystemC modeling and simulation methodology for fast and accurate parallel MPSoC simulation. In: 2013 26th Symposium on Integrated Circuits and Systems Design (SBCCI), pp. 1–6. IEEE (2013)
8. Real, M.M., Wehner, P., Rettkowski, J., Migliore, V., Lapotre, V., Göhringer, D., Gogniat, G.: MPSoCSim extension: an OVP simulator for the evaluation of cluster-based multi and many-core architectures. In: Proceedings of the 4th Workshop on Virtual Prototyping of Parallel and Embedded Systems (ViPES) as Part of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XVI), Samos, Greece (2016)
9. Buchmann, R., Greiner, A.: A fully static scheduling approach for fast cycle accurate SystemC simulation of MPSoCs. In: Proceedings of the ICEEC, Cairo, Egypt, pp. 35–39. IEEE (2007)
10. Polarsys: ARCADIA/CAPELLA (2008). <https://www.polarsys.org/capella/arcadia.html>
11. Lieverse, P., van der Wolf, P., Vissers, K.A., Deprettere, E.F.: A methodology for architecture exploration of heterogeneous signal processing systems. *VLSI Signal Process.* **29**, 197–207 (2001)
12. Balarin, F., Watanabe, Y., Hsieh, H., Lavagno, L., Passerone, C., Sangiovanni-Vincentelli, A.L.: Metropolis: an integrated electronic system design environment. *IEEE Comput.* **36**, 45–52 (2003)
13. Erbas, C., Cerav-Erbas, S., Pimentel, A.D.: Multiobjective optimization and evolutionary algorithms for the application mapping problem in multiprocessor system-on-chip design. *IEEE Trans. Evol. Comput.* **10**, 358–374 (2006)
14. Kahn, G.: The semantics of a simple language for parallel programming. In: Rosenfeld, J.L. (ed.) *Information Processing 1974: Proceedings of the IFIP Congress*, pp. 471–475. North-Holland, New York (1974)
15. Pimentel, A.D., Hertzberger, L.O., Lieverse, P., van der Wolf, P., Deprettere, E.F.: Exploring embedded-systems architectures with Artemis. *IEEE Comput.* **34**, 57–63 (2001)

16. Kienhuis, B., Deprettere, E.F., van der Wolf, P., Vissers, K.: A methodology to design programmable embedded systems. In: Deprettere, E.F., Teich, J., Vassiliadis, S. (eds.) SAMOS 2001. LNCS, vol. 2268, pp. 18–37. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45874-3_2
17. Vidal, J., de Lamotte, F., Gogniat, G., Soulard, P., Diguët, J.P.: A co-design approach for embedded system modeling and code generation with UML and MARTE. In: DATE 2009, pp. 226–231 (2009)
18. Gamatié, A., Beux, S.L., Piel, É., Atitallah, R.B., Etien, A., Marquet, P., Dekeyser, J.L.: A model-driven design framework for massively parallel embedded systems. *ACM Trans. Embed. Comput. Syst* **10**, 39 (2011)
19. Saxena, T., Karsai, G.: MDE-based approach for generalizing design space exploration. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010. LNCS, vol. 6394, pp. 46–60. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16145-2_4
20. Gérard, S., Espinoza, H., Terrier, F., Selic, B.: 6 modeling languages for real-time and embedded systems. In: Giese, H., Karsai, G., Lee, E., Rumpe, B., Schätz, B. (eds.) MBEERTS 2007. LNCS, vol. 6100, pp. 129–154. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16277-0_6
21. IBM Corporation: Rational Rhapsody. <https://www.ibm.com/us-en/marketplace/rational-rhapsody>
22. Sodijs Corporation: MDGen for SystemC. <http://sodijs.com/products-overview/systemc>
23. Feiler, P.H., Lewis, B.A., Vestal, S., Colbert, E.: An overview of the SAE architecture analysis & design language (AADL) standard: a basis for model-based architecture-driven embedded systems engineering. In: Dissaux, P., Filali-Amine, M., Michel, P., Vernadat, F. (eds.) IFIP WCC TC2 2004. IFIP The International Federation for Information Processing, vol. 176, pp. 3–15. Springer, Boston (2004). https://doi.org/10.1007/0-387-24590-1_1
24. Yu, H., Joshi, P., Talpin, J.P., Shukla, S.K., Shiraiishi, S.: The challenge of interoperability: model-based integration for automotive control software. In: DAC, pp. 58:1–58:6. ACM (2015)
25. Bombieri, N., Fummi, F., Vinco, S., Quaglia, D.: Automatic interface generation for component reuse in HW-SW partitioning. In: 2011 14th Euromicro Conference on Digital System Design, pp. 793–796 (2011)
26. Batori, G., Theisz, Z., Asztalos, D.: Domain specific modeling methodology for reconfigurable networked systems. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 316–330. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75209-7_22
27. Genius, D., Apvrille, L.: Virtual yet precise prototyping: an automotive case study. In: ERTSS 2016, Toulouse (2016)
28. Genius, D., Apvrille, L.: System-level design for communication-centric task farm applications. In: 12th International Symposium on Reconfigurable Communication-centric Systems-on-Chip, pp. 1–8. IEEE (2017). <https://ieeexplore.ieee.org/document/8016145/>
29. Scheppe, H., Roudier, Y., Weyl, B., Apvrille, L., Scheuermann, D.: C2x communication: securing the last meter. In: The 4th IEEE International Symposium on Wireless Vehicular Communications, WIVEC 2011, San Francisco, USA (2011)
30. SoCLib Consortium: SoCLib: an open platform for virtual prototyping of multi-processors system on chip. <http://www.soclib.fr> (2010)
31. VSI Alliance: Virtual component interface standard (OCB 2 2.0). Technical report, VSI Alliance (2000)

32. Genius, D., Pouillon, N.: Monitoring communication channels on a shared memory multi-processor system on chip. In: ReCoSoC, pp. 1–8. IEEE (2011)
33. Genius, D., Faure, E., Pouillon, N.: Mapping a telecommunication application on a multiprocessor system-on-chip. In: Gogniat, G., Milojevic, D., Morawiec, A., Erdogan, A. (eds.) Algorithm-Architecture Matching for Signal and Image Processing. LNEE, vol. 73, pp. 53–77. Springer, Dordrecht (2011). https://doi.org/10.1007/978-90-481-9965-5_3
34. Genius, D.: Measuring memory access latency for software objects in a NUMA system-on-chip architecture. In: ReCoSoC, pp. 1–8. IEEE (2013)
35. Knorreck, D., Apvrille, L., Pacalet, R.: Formal system-level design space exploration. *Concurr. Comput.: Pract. Exp.* **25**, 250–264 (2013)
36. Enrici, A., Apvrille, L., Pacalet, R.: A model-driven engineering methodology to design parallel and distributed embedded systems. *ACM Trans. Des. Autom. Electron. Syst.* **22**, 34:1–34:25 (2017)
37. Pedroza, G., Knorreck, D., Apvrille, L.: AVATAR: a SysML environment for the formal verification of safety and security properties. In: The 11th IEEE Conference on Distributed Systems and New Technologies (NOTERE 2011), Paris, France (2011)
38. Bengtsson, J., Yi, W.: Timed automata: semantics, algorithms and tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) ACPN 2003. LNCS, vol. 3098, pp. 87–124. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27755-2_3
39. Blanchet, B.: An efficient cryptographic protocol verifier based on prolog rules. In: Proceedings of the 14th IEEE Workshop on Computer Security Foundations, CSFW 2001, Washington, D.C., USA, p. 82. IEEE Computer Society (2001)
40. Li, L.W., Lugou, F., Apvrille, L.: Security-aware modeling and analysis for HW/SW partitioning. In: Conference on Model-Driven Engineering and Software Development (Modelsward 2017), Porto, Portugal (2017)
41. Lugou, F., Li, L.W., Apvrille, L., Ameur-Boulifa, R.: SysML models and model transformation for security. In: Conference on Model-Driven Engineering and Software Development (Modelsward 2016), Rome, Italy (2016)
42. Etienne Faure: Communications matérielles-logicielles dans les systèmes sur puce orientés télécommunications (HW/SW communications in telecommunication oriented MPSoC). Ph.D. thesis, UPMC (2007)
43. Becoulet, A.: MutekH. <http://www.mutekh.org>



Automated Synthesis of a Real-Time Scheduling for Cyber-Physical Multi-core Systems

Johannes Geismann^{1(✉)}, Robert Höttger², Lukas Krawczyk², Uwe Pohlmann³, and David Schmelter³

¹ Software Engineering Research Group, Paderborn University,
Zukunftsmühle 1, 33102 Paderborn, Germany
johannes.geismann@upb.de

² IDiAL Institute, Dortmund University of Applied Sciences and Arts,
Otto-Hahn-Str. 23, 44227 Dortmund, Germany
{robert.hoettger, lukas.krawczyk}@fh-dortmund.de

³ Software Engineering Department, Fraunhofer IEM,
Zukunftsmühle 1, 33102 Paderborn, Germany
{uwe.pohlmann, david.schmelter}@iem.fraunhofer.de

Abstract. Cyber-physical Systems are distributed, embedded systems that interact with their physical environment. Typically, these systems consist of several Electronic Control Units using multiple processing cores for the execution. Many systems are applied in safety-critical contexts and have to fulfill hard real-time requirements. The model-driven engineering paradigm enables system developers to consider all requirements in a systematical manner. In the software design phase, they prove the fulfillment of the requirements using model checking. When deploying the software to the executing platform, one important task is to ensure that the runtime scheduling does not violate the verified requirements by neglecting the model checking assumptions. Current model-driven approaches do not consider the problem of deriving feasible execution schedules for embedded multi-core platforms respecting hard real-time requirements. This paper extends the previous work on providing an approach for a semi-automatic synthesis of behavioral models into a deterministic real-time scheduling. We add an approach for the partitioning and mapping development tasks. This extended approach enables the utilization of parallel resources within a single ECU considering the verification assumptions by extending the open tool platform APP4MC. We evaluate our approach using an example of a distributed automotive system with hard real-time requirements specified with the MechatronicUML method.

Keywords: CPS · MDSD · Real-time scheduling · Synthesis
Model-transformation · Multi-core · Automotive · Amalthea
APP4MC

1 Introduction

Cyber-physical Systems (CPSs) are executed in physical environments, interact with each other, and are distributed over several Electronic Control Units (ECUs). Examples of CPSs are modern cars in Car-2-Car and Car-2-X scenarios. Often, these systems perform safety-critical tasks under hard real-time requirements. Heterogeneous hardware architectures consisting of interconnected multi-core ECUs are increasingly used in order to fulfill the increasing demand for computing power.

Model-driven development methods like MECHATRONICUML [7] are applied to develop the embedded software of interconnected CPSs efficiently, correctly, and to cope with the overall complexity. For this, a Platform Independent Model (PIM) is developed consisting of a component-based software architecture. Formal verification approaches like timed model checking [1] are applied to ensure the functional correctness of the modeled behavior. Afterwards, the PIM is refined to a Platform Specific Model (PSM) in order to map the PIM to the underlying multi-core platform. Especially, a scheduling needs to be derived for utilizing a multi-core platform efficiently. Moreover, the verified safety and real-time requirements need to be preserved in the scheduling. However, a systematic method to derive a feasible multi-core scheduling for interconnected CPSs that preserves verified safety and real-time requirements by design is missing.

This paper is an extended version of [14]. We present an approach that enables a step-wise, semi-automatic synthesis of behavioral models into a deterministic scheduling suited for multi-core target platforms and respects safety and real-time requirements. In addition to [14], we present in this version sophisticated techniques for grouping software parts into executable units (called partitioning) and for assigning these units to the execution cores respecting all real-time requirements by design. We embed our approach in the MECHATRONICUML [7] and APP4MC [2] toolchains and evaluate our results with an automotive example. MECHATRONICUML provides a modeling language, a development process, and an Eclipse-based tooling to design software for interconnected CPSs. APP4MC focuses on the optimization of timing and scheduling in embedded multi- and many-core systems in the context of AUTOSAR [6]. Therefore, APP4MC provides and utilizes the AMALTHEA model.

In Fig. 1, we give an overview of our synthesis approach by means of a Business Process Model and Notation (BPMN) diagram. The upper BPMN pool represents the PIM modeling. First, the software architecture of the system is created (BPMN Task 1). Software components with behavior in terms of statecharts are part of this architecture. The resulting architecture is the input of our approach. Task 2 is the first contribution of this paper. Here, the so-called *segmentation* is applied. In the segmentation, the statecharts are split into small executable parts that allow parallel execution of the modeled software. Corresponding to the AUTOSAR specification [6], we call these parts *runnables*. Also, *runnable properties* like a period for periodic execution are determined which are essential to ensure semantically correct execution as we show in this paper. The lower BPMN pool represents the PSM modeling. In Task 3, the

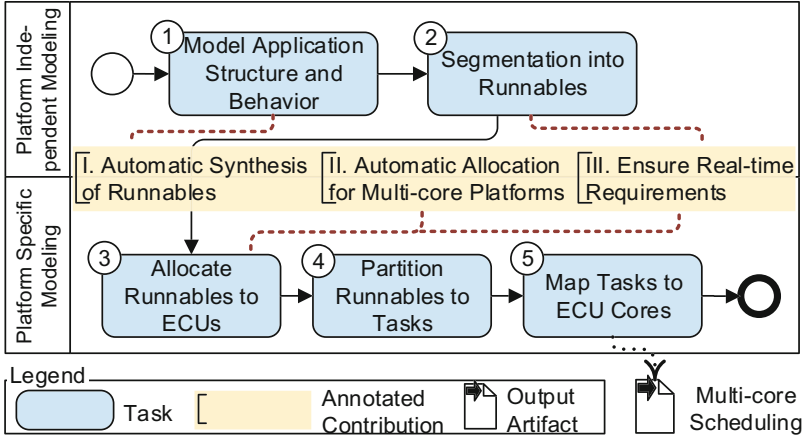


Fig. 1. Process diagram and contributions (cf. [14]).

generated runnables are automatically allocated to the distributed, interconnected ECUs. This allocation is the second contribution of this paper. In Task 4 and 5, AMALTHEA tasks are created and mapped to ECU cores by means of APP4MC’s partitioning and mapping algorithms, respectively. The detailed explanation of partitioning and mapping (cf. Sects. 3.3 and 3.4) are the main additional contribution of this long version of the paper. The overall result of the presented process is a deterministic scheduling that is suited for multi-core target platforms. In Tasks 2 and 3, we ensure the execution semantics and real-time requirements of the modeled behavior in the resulting scheduling. This is the third contribution of this paper.

For illustrating our approach, we use the running example shown in Fig. 2. The upper part of Fig. 2 depicts an autonomous overtaking scenario involving two cars. The cars communicate to coordinate the overtaking maneuver. In our example, the overtaker (red) overtakes the overtakee (green) while the overtakee guarantees that it do not accelerate during the overtaking. This scenario is safety-critical because an error in the communication can result in an unsafe overtaking maneuver. We assume that the correctness of the specified software including its real-time behavior has been formally verified on PIM level by applying model checking [16].

The remainder of this paper is structured as follows. In the next section, we introduce the MECHATRONICUML models that are relevant and used for our synthesis approach. In Sect. 3, we present our segmentation approach. Additionally, we present our allocation approach for interconnected multi-core ECUs. In Sect. 4, we evaluate our approach. In Sect. 5, we discuss related work. Finally, we conclude our paper and discuss future work in Sect. 6.

2 Modeling the Application

In this section, we give an introduction to the MECHATRONICUML modeling artifacts that we use for the software specification on PIM level. Figure 3 shows an overview of all used modeling views, artifacts, and their relations.

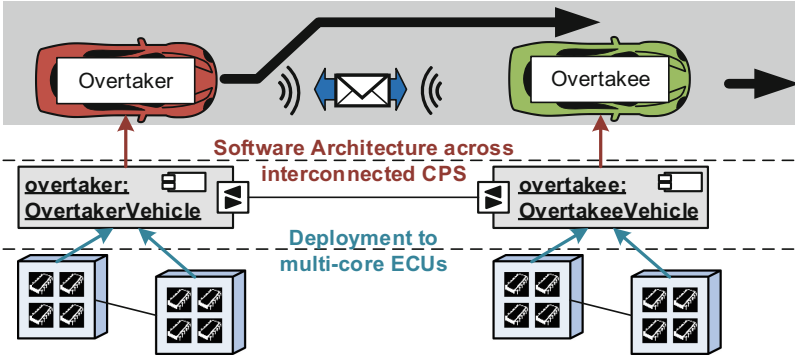


Fig. 2. Running example autonomous overtaking (cf. [14]). (Color figure online)

The Component Instance Configuration view shows the software architecture in terms of a compositional component model. In the top part, Fig. 3 shows an excerpt of the software architecture realizing the overtaking scenario. It consists of the component instances `overtaker` and `overtakee`. The component instance `overtakee` is composed of the instances `overtakeeCommunicator` and `overtakeeDriver`. Component instances have ports that can send and receive typed messages. Connector instances connect ports and have Quality of Service (QoS) assumptions like a maximum transmission time. For example, the `overtaker` sends the messages `request` and `finished` to the `overtakee` and can receive the messages `accept` or `decline` from the `overtakee`. Based on the QoS assumptions, the model checking assumes that messages are transmitted within 100 ms. Furthermore, component instances can be connected to continuous component instances that represent sensors and actuators of the CPS. For the reason of comprehensibility, we omit these components in the diagram.

The component's behavior is specified in terms of Real-time Statecharts (RTSCs) which combine UML state machines [27] and timed automata [1]. Figure 3 shows the behavior of component instance `overtakee`. RTSCs can be composed of so-called regions that again contain state machines. For instance, `CommunicatorRTSC` is composed of the regions `communicator` and `internal`. The region `communicator` represents the behavior of the communication with the `overtaker` and is composed of the states `init`, `overtaking`, and `requested`. The region `internal` represents the internal behavior of the component instance that takes the decision whether the overtaking is safe or not and is composed of the states `safe`,

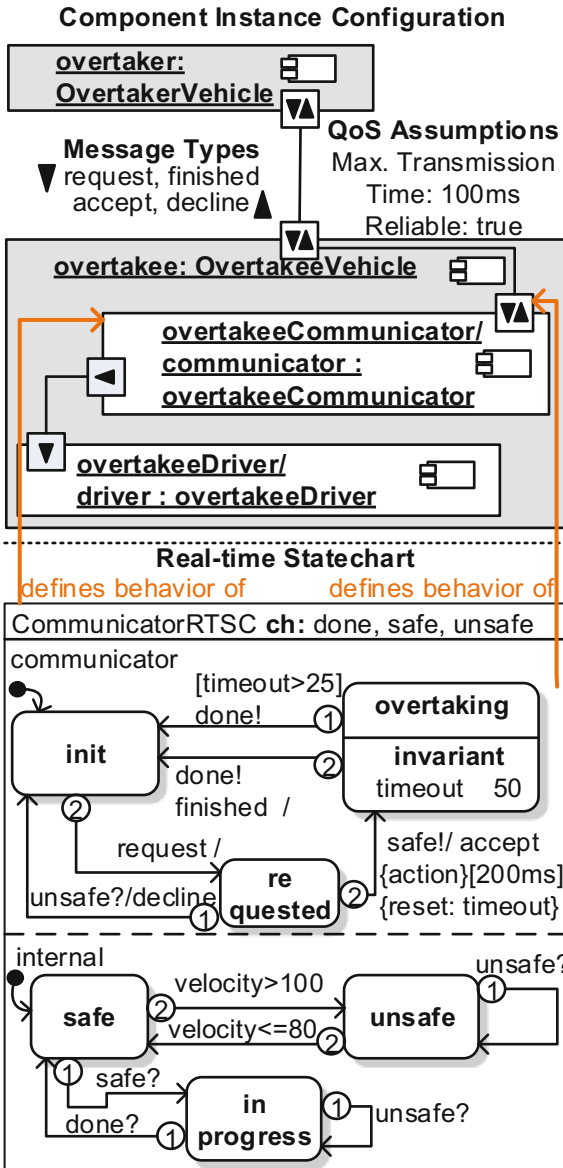


Fig. 3. Overview of software development views (cf. [14]).

unsafe, and in progress. RTSCs may share variables (e.g., velocity in region internal) and have *clocks* that measure the time and can be *reset* to zero within the statechart, e.g., *timeout* in the region communicator. Furthermore, each RTSC has exactly one currently active state. A state may contain an invariant as a real-time property, which restricts the value of the clock when the state is active. It

must be guaranteed during runtime that an invariant is never violated, e.g., the state `overtaking` has to be left before the clock `timeout` reaches 50 ms. A transition may have a guard (`[velocity > 100]`), time constraints (`[timeout > 25]`), a trigger message (`trigger /`), and a synchronization channel that restricts the firing (`sender channel! /`, `receiver channel? /`). It is *enabled*, i.e., it is able to fire, if its source state is active, its guard evaluates to true, its time constraint evaluates to true, and its trigger message is stored within the buffer. Furthermore, some transitions are connected with each other via synchronization channels; the transition from the state `requested` to the state `overtaking` in region `communicator` is synchronized with the transition from state `safe` to `overtaking` in region `internal` via the synchronization channel `safe`. Thus, these transitions may only fire jointly.

We assume that RTSCs are executed step-wise, i.e., in each step the outgoing transitions of the currently active state (and all synchronized transitions) are evaluated. If a transition is enabled, the transition with the highest priority fires and the currently activate state gets updated.

3 Software Distribution and Parallelization

In this section, we explain our proposed approach for segmentation and allocation in more detail. We assume that models for the PIM are already created and requirements are verified using model checking (cf. BPMN Task 1, Fig. 1). The remainder of this section is structured by following the development process as shown in Fig. 1. Afterwards, the *Partitioning* (Sect. 3.3) and *Mapping* (Sect. 3.4) approaches are outlined that are used to find a feasible scheduling for all runnables allocated to an ECU under consideration of diverse constraints mentioned accordingly.

3.1 Segmentation into Runnables

The segmentation defines which part of the software models are mapped to a runnable. Runnables are the smallest unit that can be executed by the system and, therefore, segmenting the PIM into runnables affects the behavior execution on the target platform directly. Additionally, WCET, period, and deadline are defined for each runnable. This step is crucial for semantically correct execution because an invariant might be violated if a runnable is executed too late. Thus, the segmentation has to fulfill the following requirements. **R1:** The segmentation has to allow parallel execution. Multi-core environments increase the performance of a system by using parallelization. Therefore, software has to be separated into runnables that can be executed in parallel. **R2:** We aim to generate as few runnables as possible without degrading the possibility of parallel execution because with an increasing number of runnables, the complexity of the partitioning step also increases, which makes it more difficult to find a feasible scheduling and may lead to a decrease in the performance of the system. **R3:** Real-time requirements must be fulfilled at runtime. On PIM level, model

checking techniques are used to ensure the fulfillment of these requirements at design time. Executing the software on a platform adds further parameters that have not been considered during the verification step on PIM level, e.g., the activation due to the concrete scheduling. Thus, a requirement for the resulting scheduling is to ensure that the semantics of the PIM is respected.

In a first step, MECHATRONICUML software models have to be split into runnables. RTSCs of the software architecture are the starting point for the segmentation. The segmentation directly addresses the first and second requirement because it defines which parts of the software can be executed in parallel. We propose to generate one runnable per region of every RTSC because it allows parallel execution of component behavior without increasing the number of runnables significantly. Furthermore, this segmentation is reasonable because each port behavior is described in exactly one region. Hence, we generate one runnable per port behavior and, therefore, the different communication protocols of a component can be executed in parallel. In addition, we generate one runnable per continuous component that is used to read sensor values periodically. Executing the runnable for a region executes one step of the corresponding RTSC, i.e., evaluating and possibly firing outgoing transitions of the currently active state.

The resulting runnables may have dependencies since they may share RTSC variables. These dependencies are important for partitioning and mapping because runnables accessing the same variable are not suitable to be executed in parallel. Corresponding to AUTOSAR, we call such variables *labels*. At first, we define labels and label-accesses of runnables. Furthermore, RTSCs may use shared variables and real-time clocks, for which labels are generated also. These label-accesses are specified for every runnable. Figure 4 shows the label accesses for the example RTSC in Fig. 3.

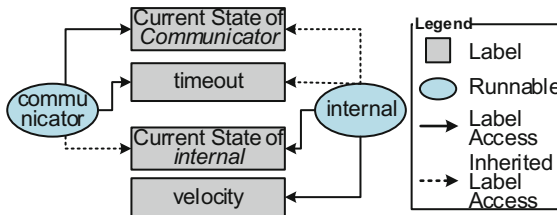


Fig. 4. Runnables have to specify label accesses (cf. [14]).

Both runnables define a label access to their current state label. The runnable for region *communicator* defines a label access to the label for the clock *timeout*. The runnable for region *internal* defines a label access to the variable *velocity*.

Additionally, both runnables specify inherited label accesses, which are needed, if synchronization channels are used. Since two transitions have to be fired jointly, we propose to extend the models and implementation for runnables

by the possibility to evaluate and fire all synchronized transitions. In Fig. 3, the transition from state `overtaking` to `init` in region `communicator` are synchronized with the transition from state `in progress` to `safe` via the synchronization channel `done`. Hence, both runnables inherit the label `accesses` from the other runnable.

In a second step, we derive runnable properties. Since these properties directly affect the scheduling, their correct determination is crucial for preserving model checking results at runtime. Every runnable has to provide a period, a deadline, and a WCET that are used for partitioning, mapping, and further analyses. Our approach provides an automatic technique to determine a period and deadline for each runnable. Determining a platform-specific WCET is a complex topic and out of scope of this paper. In our approach, we assume that the WCET for each runnable is determined by an appropriate method (e.g., Simple Scalar [5] or aiT [13]) and provided as an annotation for each runnable.

The period describes how frequently a runnable is executed. We provide an automatic technique to determine a period, such that all real-time requirements are fulfilled at runtime without increasing the processor utilization unnecessarily. Determining the period has to respect the semantics of the transition conditions, i.e., guards, deadlines, clock constraints, and invariants. Since a runnable is executed periodically, we have to guarantee that it is executed whenever a transition is enabled.

Based on the transition conditions, we can determine an *enabling interval* I_E which describes the time span when a transition is enabled. We determined a computation rule how I_E can be computed for all combinations of transition conditions. In general, we define $I_E = I_{max} - I_{min}$, where I_{min} is the first point in time and I_{max} is the last point in time when all transition conditions validate to true. As an example, consider the combination of a clock constraint and a state invariant, e.g., the transition from state `overtaking` to `init` in region `communicator` with priority 1 in Fig. 3. The transition has a clock constraint that is enabled when the clock timeout is greater than 25 ms. Additionally, the state `overtaking` has an invariant that is valid when the clock timeout is less or equal 50 ms. Figure 5 shows the time frames when each constraint validates to true. Hence, I_{min} is at 25 ms and I_{max} is at 50 ms. Thus, the valid enabling interval I_E has a length of 25 ms.

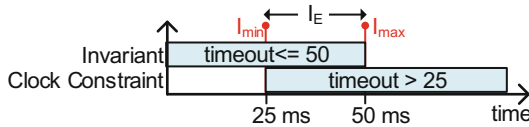


Fig. 5. Finding the enabling interval of a transition (cf. [14]).

If several clock constraints are used, we can generalize I_{min} to the infimum of all *greater-or-equal* constraints and I_{max} to the supremum of all *less-or-equal* constraints. Similar to this, we defined for all other transition conditions a similar

computation. Since guards can depend on sensor values, guards also depend on the period of the runnable of the corresponding continuous component. Thus, guards have to be considered in the computation of I_{min} and I_{max} .

It is crucial that the runnable is executed during I_E for each transition because an enabled transition might become disabled again before firing. Otherwise, the assumptions used during model checking would be neglected. Thus, based on I_E we determine a period for the runnable. For this, we set the period to half of the length of the shortest enabling interval I_E . Figure 6 illustrates that a well-chosen period is essential to guarantee the firing of an enabled transition. It shows two different cases of the execution for the runnable that handles the transition of the example above. Each case shows the enabling interval of the transition, the periodic activation times of the runnable, and the concrete execution of the runnable. On the left, the period is set to I_E . Here, the enabling interval of the transition is missed because the transition is evaluated too late. Therefore, the invariant of the state gets violated. On the right, the period is set to $\frac{I_E}{2}$ which ensures that the runnable is executed at least once during the enabling interval because a runnable is executed completely before it is activated again.

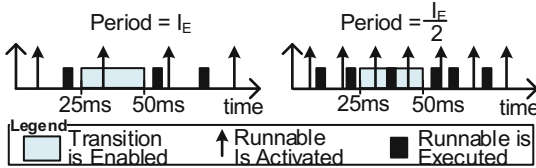


Fig. 6. Length of period affects the execution (cf. [14]).

Since the period π_r has to respect all transitions of the runnable, the period of a runnable r is defined as the minimum of all period values:

$$\pi_r = \min \left\{ \left\lceil \frac{\min(I_E)}{2} \right\rceil \mid \forall I_E \in \text{runnable} \right\}, \quad (1)$$

The current approach is limited to local (within one region) clocks and to clocks that get reset when entering the state. Otherwise, the enabling interval cannot be determined precisely. If global clocks should be supported in the future, a solution could be to apply a reachability analysis to find all possible clock zones.

Every runnable defines a deadline. Similar to the period of a runnable, the deadline depends on the execution of each transition of an RTSC since every transition can define a dedicated deadline. Consequently, the runnable has to be finished before the deadline of the firing transition expires. Thus, the deadline of a runnable is defined as the minimum deadline of all transitions that are evaluated by this runnable. If no deadline is specified, we set the deadline to the period value of the runnable, since the runnable has to be finished before it is activated again.

3.2 Allocate Runnables to ECUs

After the segmentation, we have to define which runnable is executed on which ECU (cf. BPMN Task 3, Fig. 1). Furthermore, hard real-time requirements of the communication have to be respected.

In the following, we derive two constraints that an allocation of runnables to ECUs has to fulfill: 1. A constraint regarding a necessary condition for schedulability. 2. A constraint that ensures the maximum time for communication at runtime. Based on runnable properties, the constraints are used to guarantee the maximum transmission time and schedulability of the system with regard to the real-time requirements during the allocation.

When allocating runnables to ECUs, it is required that all ECUs have enough processing capacity to execute all allocated runnables. The runnables for each allocated component decrease the available processing capacity of the ECU. We restrict the allocation regarding a necessary condition for schedulability: The amount of computing time of the executed software must not exceed the processing capacity of the ECU. We define the processing capacity of each ECU core as 1. For simplicity, we assume that all ECUs use homogeneous cores. Thus, all cores have the same processing capacity and, consequently, the processing capacity of each ECU is defined as $C_{ECU} = |ECUCores|$. The utilization factor of a runnable U_r describes how much percentage of C_{ECU} is needed to execute this runnable. We define U_r of runnable r for a specific ECU as $U_r = \frac{WCET_{r,e}}{\pi_r}$, where $WCET_{r,e}$ is the upper bound of the execution time of runnable r on ECU e and π_r is the period of runnable r . If the sum of the utilization factors of all runnables exceeds the processing capacity of the ECU, it is impossible to find a valid scheduling for a given set of runnables. Hence, this sum has to be *less* than the processing capacity of the ECU.

$$\sum_{r \in \text{Runnables}(ECU)} U_r < k * C_{ECU}, k \in [0; 1] \quad (2)$$

k is a constant factor that can be defined by the developer to adjust this constraint for her needs, e.g., to restrict the maximal processor utilization.

Another crucial aspect is the communication time between two components. The allocation affects the communicating time that is needed for communication. In MECHATRONICUML, the maximum transmission time is constrained by the QoS of a connector instance, denoted by $T_{ConInst}$, e.g., 100 ms for the communication between component instances *overtaker* and *overtakee* in Fig. 3. For the communication, we assume that each components port behavior (one region of the RTSC) is executed by one runnable: a sender runnable r_S that sends the message and a receiver runnable r_R that receives and processes the message. Additionally, we assume that a lower layer is used to handle the transmission of the message from r_S to r_R , e.g., a middleware. Based on [30], we define that delivering a message relies on time for generating and sending the message t_s , transmitting it from sender to receiver t_{trans} , and queuing it until the receiving process recognizes the message t_r . Figure 7 illustrates the derivation of t_s , t_{trans} , and t_r . When a message is sent by r_S , we assume that the middleware sends

the message directly after a task has finished. Thus, the message is processed by the middleware at least before the runnable is executed again. Hence, t_s can be estimated by the period of the runnable π_s .

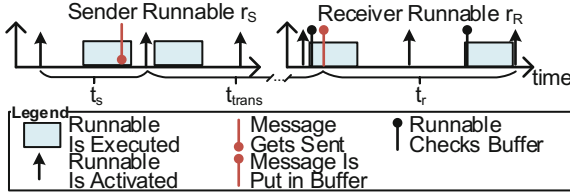


Fig. 7. Upper bound of time for sending and receiving (cf. [14]).

t_{trans} is based on the used middleware and the underlying communication protocol. We assume that an upper bound constant can be statically determined for each communication channel and used middleware. t_r describes the time it takes from the point in time when the message is put into the message buffer until runnable r_R recognizes the message. Let us assume that the message is put into the buffer immediately after r_R checked the buffer as depicted in the right part of Fig. 7. Hence, in this execution, the message is not received by the runnable. Since r_R is activated periodically, it has to be finished completely within the next period interval. Consequently, the time until the message buffer is checked again by the runnable is smaller than $2 * \pi_{receiverrunnable}$. Hence, we use this time as an upper bound for t_r and state the constraint:

$$\pi_s + t_{trans} + 2 * \pi_r \leq T_{ConInst} \quad (3)$$

Both proposed constraints (Eqs. 2 and 3) are implemented using the allocation approach of MECHATRONICUML [29], which allows specifying allocation constraints for components, e.g., which components have to be allocated to the same ECU. Thereby, we introduce additional allocation constraints in order to realize an automatic allocation of runnables. We use the heuristic that runnables that belong to the same component instance have to be allocated to the same ECU because a software component instance has a strong coherence [17]. Hence, in this step, we still allocate components to ECUs with respect to the runnable properties.

For each ECU, further actions are needed to refine the models to schedulable software: A *Partitioning* of runnables to tasks and *Mapping* these tasks to ECU cores such that all constraints are fulfilled (cf. BPMN Task 4 and Task 5, Fig. 1). Finally, the *deployment* of the software takes place which includes the generation of source code for a given multi-core scheduling.

3.3 Partitioning to Tasks

Partitioning in terms of APP4MC focuses on identifying software tasks that can potentially run at different processing cores. Therefore, runnables' activa-

tion parameters, instructions, and dependencies are considered. Publication [19] describes the corresponding algorithms. Our experience is that causality, i.e., the runnable order, is the most influencing criterion for the partitioning process. We represent the causality by modeling runnable order using directed acyclic graphs. Due to the specific demands of automotive software the used graph algorithms are extended. Such demands emerge from either communication technologies, advanced driver assistant systems, safety and security concepts, architectural approaches, or diverse design decisions and can often be reflected in specifying and considering constraints. Example for constraints are, among others, core affinities, ASIL level references, software component tags, runnable pairings or separations, or timing constraints. Considering these constraints during the software parallelization is a further contribution of this paper.

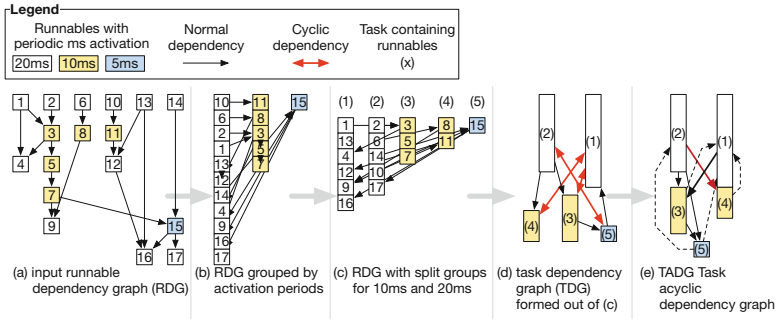


Fig. 8. Example partitioning of a Runnable Dependency Graph (RDG) to a Task Acyclic Dependency Graph (TADG).

Figure 8(a) shows a typical graph structure of runnables as rectangles and dependencies as arrows. Figure 8(b) depicts the same Runnable Dependency Graph (RDG) ordered by the runnables' periods and Fig. 8(c) shows the same graph whereas runnables for 10 ms and 20 ms are each split once. This partitioning can be configured in different ways. Here shown is a strategy to reduce cross partition dependencies and consider vertical sequences. To maximize parallelism such that runnables of the same activation rate can be computed on different cores concurrently, the vertical partitioning that considers sequences and cross partition dependencies is the prior choice. Another configuration could handle the topological level of runnables, i.e., horizontal partitioning. This latter case is preferred if the partitions are assumed to be scheduled sequentially, e.g., due to the availability of just a few cores. The two different schemes can be configured in APP4MC and have to be distinguished carefully to avoid unnecessary inter-communication overheads. Finally, (d) outlines a Task Dependency Graph (TDG) that contains the runnables from (c) with merged dependencies and (e) depicts the same graph as (d) but without any cycles denoted as Task Acyclic Dependency Graph (TADG). The mechanism to resolve cycles is described in [3].

It is important to note here, that B rectangles outline blocking periods due to shared resources, i.e., labels across cores are already in use by another running runnable on a different core. (a) further assumes having three cores dedicated for runnables with a specific period, i.e., core C0 for 20 ms, C1 for 10 ms, and C2 for 5 ms. However, this model would also be schedulable for 2 cores as shown in Fig. 9 (b).

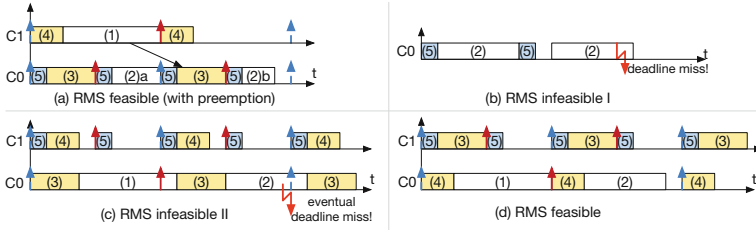


Fig. 9. Gantt charts of scheduling for the example that is shown in Fig. 8(e).

In order to have more flexibility in terms of software distribution, partitions shown in Fig. 8(c) are formed and transformed to task graphs as (e). Figure 9 provides four different task distribution scenarios (a)–(d) whereas only (d) provides a solution with no preemption and (a) is only feasible with preemption. If either task 5 is combined with task 2 or task 3 is combined with task 1, no feasible schedule can be found. The dashed vertical arrows pointing upwards outline the release of two or more tasks. The scheduling applied to the shown executions is rate monotonic scheduling (RMS). Since the periods are harmonic, the schedulability test

$$u = \sum_{i=1}^n \frac{C_i}{T_i} \leq 1 \tag{4}$$

is sufficient to form schedulable partitions (with C_i denoting the runnable’s instructions, T_i denoting the runnable’s activation rate, and n representing the number of runnables). The schedulability test during the partitioning (without consideration of the hardware topology) prior to the mapping (including consideration of the hardware topology) ensures valid and coherent solutions in order to identify the most effective software distribution scenarios.

The challenge in forming partitions like in Fig. 8(c) is not only considering causality, instructions, and activations, but also the above mentioned constraints. If, e.g., runnable 15 is paired with runnable 2 due to, e.g., tight functional relation within the braking system that is not represented by a dependency, partition (2) would have to be composed differently in order to generate a feasible schedule. Figure 10 outlines the consideration of runnable pairing constraints via runnable cumulation.

Any runnable pairing constraint merges the corresponding runnables for the graph algorithms (cumulation) and decomposes (reconstruction) to the original

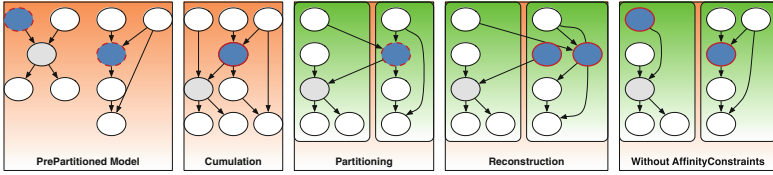


Fig. 10. Runnable cumulation mechanism to consider runnable pairing constraints.

structure after partitions have been formed. Consequently, causality, i.e., the runnable pairing positions and sequences within partitions are considered.

Other than that, if runnables 12, 9, and 16 were safety relevant, e.g., implement a braking system, and reference an according ASIL level, they would have to be separated into an independent partition in order to guarantee freedom from interference, e.g., resource blocking. Therefore, the dependencies must be carefully analyzed and possible blocking situations should be identified so that execution times can be reasoned precisely.

When taking tags, e.g., for software component instances, into account, it is desired to predefine whether and if yes how many component instances can be combined within partitions. Tags are an abstract AMALTHEA model element that can be referenced by runnables or tasks in order to group them according to the diverse users' needs.

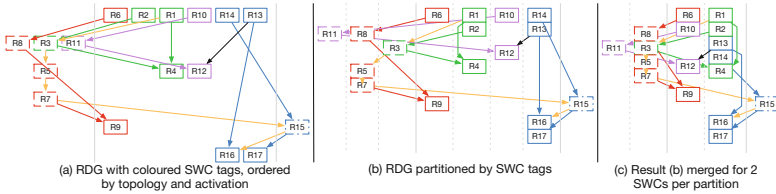


Fig. 11. Partitioning with tag consideration for software component instances (SWC).

Figure 11(a) shows the graph from Fig. 8 further extended by colors indicating a specific software component. The runnables are further topologically ordered. (a) is transformed to (b) in order to group component instance-related runnables for each activation. Each column represents a group respectively a partition. Finally, (c) shows a possible configuration for two component instances per partition. This merging process preserves causal relations within a group and aims at balancing the load across partitions. Obviously, the outcome is quite different from the result shown in Fig. 8 due to the consideration of tags.

ASIL levels and tags (e.g., for software components) are equally considered such that separate groups are formed prior to the partitioning process, which always splits the most instructions consuming partition first. Consequently, generated partitions are aligned regarding their instruction sums as much as

possible. In other words, all runnables referencing the same tag are grouped into a partition that are may split additionally if the sum of all runnables contained in this group is higher than other the instruction sums at other partitions. An important aspect of this mechanism is its influence of the overall software distribution. In order to keep the amount of generated AccessPrecedences (i.e., a dissolution of a direct cause and effect relation of two runnables) low, i.e., to keep the program’s causality at a high level, existing groups can me merged with other groups or partitions.

When being scheduled, runnables are called directly after each other if no delays are implemented between them. This may eventually result in certain runnables being executed prior to their predecessors. Such behavior can be accepted if these situations were analyzed and verified accordingly, resulting in AccessPrecedence model elements, allowing according runnables to execute with *older* values provided by their predecessors. If such AccessPrecedence is not present, system integrators must assure that runnables wait for label updates provided by their predecessors via events, interrupts, delays, or similar mechanisms.

The difference between runnable pairing and, for instance, tag groups, is that a runnable pairing also influences the position of the corresponding runnables within a task whereas groups have no direct sequencing influence.

As soon as the partitions are formed that consider all the above listed constraints, partitions are transformed to tasks and their possible mappings to processing cores are investigated as described in the following Sect. 3.4.

3.4 Mapping Tasks to Cores

Mapping in the context of APP4MC describes the process of finding a *valid* and *efficient* allocation from software elements to hardware components, i.e., of executable software (runnables or tasks) to cores, data to (distributed) memories, and communications to underlying inter-core networks. These allocations, or mappings, are considered valid if they fulfill all specified constraints, such as meeting an executables’ deadlines, providing inter-core communication channels between mapped executables, or being executed on certified hardware. Efficient mappings are achieved by optimizing the distribution w.r.t. one ore more so called quality attributes, e.g., by minimizing the overall runtime, the total energy consumption, or maximizing the reliability of a system. The APP4MC OpenMapping plugin implements this functionality and provides several mapping approaches that are based on various optimization techniques and feature multiple quality attributes. A brief description of these approaches can be found in [21].

In Sect. 3.2, we described the process of allocating runnables to ECUs. Hence, it is necessary to refine the deployment in order to further distribute the generated tasks from Sect. 3.3 onto the hardware resources of the corresponding ECU. Similar to the allocation process of runnables to ECUs, the mapping phase has to consider, among others, the execution time (or *response time*) of tasks in order to ensure that deadlines are met. Due to the heterogeneous nature of embedded

systems, the response time of a task mainly depends on the core it is mapped to. APP4MC allows specifying the number of instructions for executing tasks on a per core basis, i.e., it is necessary to determine the WCET of a task for all valid mapping targets beforehand, e.g., by means of profiling or appropriate analysis tooling. Once this information is available, the execution time $et_{e,c}$ for executing a task e on core c can be calculated as stated in Eq. 5, with $INS_{e,c}$ being the number of instructions for this concrete mapping, and IPS_c being the number of executable instructions per second. The latter is derived from the AMALTHEA HW Model using Eq. 6, with IPC_c being the executed instructions per cycle, PS_c the Prescaler (frequency scale or divider), and f_c the frequency the core operates at.

$$et_{e,c} = \frac{INS_{e,c}}{IPS_c} \quad \forall \quad e \in Tasks, c \in Cores \quad (5)$$

$$IPS_c = IPC_c \times PS_c \times f_c \quad (6)$$

For determining valid mapping targets we consider architectural constraints. Pairing- and separation constraints are treated similarly as in Sect. 3.3, enforcing or prohibiting the co-existence of a task on the same core. Architectural constraints allow annotating e.g., the required ASIL for a target core, requirements on hardware accelerators, or lockstep modes. Each task can be annotated with features that either are required (enabled) or prohibited (disabled). Accordingly, the final set of valid mapping targets is $Cores \setminus D$ with $E = \emptyset$ and $E \setminus D$ otherwise, with $Cores$ being the set of all available cores, E the set of cores with the required features, and D the set of cores with prohibited features.

Once the solution space is restricted, a mathematical model describing the mapping problem is automatically generated based on the selected approaches optimization technique. In addition to the strategies presented in [21], we have extended this model in order to support communication costs as well as penalty based constraints. Communication costs are an important aspect in distributing tasks among cores, since slow interconnections between cores and a high fragmentation of tightly coupled tasks fosters high execution times. The communication cost is extracted from either the network description or the *AccessPaths* within the ECUs AMALTHEA Hardware Model. *AccessPaths* represent communication channels between, e.g., cores and memories along with their latency. In case *AccessPaths* are not present within the model, the latency is determined by analyzing the ECUs internal network structure, i.e., by identifying all participants within the internal network, determining all possible paths between them, and evaluating their connection in terms of latency and bit width among each other. While the latter is more complex to be solved due to the exponentially rising number of paths, it provides more flexibility in finding alternative routes on, e.g., NoC architectures.

The extracted communication costs are stored in a $m \times m$ communication matrix T with m being the number of available cores, and T_{ij} the communication cost for transferring information from core i to core j . Having the software in terms of a directed acyclic graph (DAG) $G(V, E)$ with interconnected Tasks V ,

and E being a set of edges $e(t', t)$ with t' being the source task and t the target task, the matrix is used as lookup-table for determining the execution time on a core. For a simple load balancing approach [12], this is done by adding the resp. communication overheads whenever a task communicates with another over core boundaries. This overhead $comm_{t,c}$ can be determined as shown in Eq. 7

$$comm_{t,c} = Max \left(\sum_{c'=1}^m x_{t',c'} T_{c',c} : t' \in preds(t) \right) \quad (7)$$

The variable $x_{t',c'}$ is set to 1 iff a task t' is mapped to core c' , with t' being the predecessor of task t , m the total number of cores, and $preds(t)$ a function for determining the predecessors from task t . Since every task can only be mapped to one core at a time, the sum of the communication overheads always results in the overhead caused by the predecessors mapping. In case of multiple predecessors t' , getting the *max* value ensures that only the highest delay is considered.

4 Evaluation

We conducted a case study to evaluate our approach using the overtaking example. In our case study, we focused on the correctness of the synthesis. We assume the synthesis to be correct if all relevant elements are considered in the applied transformations and all computed values are correct. We based our case study on guidelines by Kitchenham et al. [20] and the Goal-Question-Metric (GQM) method [31] for the structured definition of quality metrics. We state two hypotheses to be validated by the case study. **H1**: We expect, that for the segmentation approach a feasible multi-core scheduling can be found. **H2**: We expect that applying the allocation approach, the result is a correct allocation that respects both stated constraints (cf. Eqs. 2 and 3), if such an allocation exists. We evaluated schedules for different platforms. In the following, we show the resulting tasks for one multi-core ECU of the overtaker software component instance of the running example. The segmentation of the overtaker components results in 11 runnables, 37 labels, and 39 label accesses.

We applied the segmentation to several additional component models and compared them to manually created reference models. For each model, the segmentation resulted in the expected number of runnables, labels, and label accesses. Additionally, the generated runnable properties were correct and due to the construction of period and deadline all real-time assumptions hold at runtime. Executing partitioning and mapping of APP4MC resulted in a feasible scheduling with 7 tasks. 5 tasks are mapped to one core and 2 tasks to the other. Table 1 shows the resulting tasks, their properties, and the executing ECU core. Both cores execute runnables of the component instance `overtakeeCommunicator` and `overtakeeDriver`. Hence, the execution of the software uses the benefits of parallel execution, which reduced the response time of the system. Overall, we argue that **H1** is fulfilled.

Table 1. Tasks resulting from partitioning [14].

Core	Task	Component	Period (ms)
Core 1	T3	Communicator	500
	T6	Driver	500
Core 2	T0	Driver	25
	T1	Communicator	25
	T2	Driver	12
	T4	Communicator	500
	T5	Communicator	500

For evaluating the allocation approach, we considered QoS assumptions of connectors. For each connector, the expected constraints were generated. Additionally, we used different values for the periods of the sender runnable and receiver runnable, as well as for the underlying platform model to test the cases that (A) a valid allocation with two ECUs is found, (B) a valid allocation with only one ECU is found, and (C) no valid allocation is found. For each value combination, the results are as expected. Thus, we state that **H2** is fulfilled. The case study shows that our concepts and the implementation work as expected. Due to the higher degree of automation in the whole development process, there are less manual steps in comparison to state of the art approaches. Additionally, the systems engineer needs less domain knowledge for embedded systems and scheduling. The main threats to validity are: 1. We applied our approach to a small example. 2. We assume that the partitioning and mapping of APP4MC consider all specified constraints correctly, and 3. We assume that the code generation is correct. Overall, we argue that our approach helps to increase the automation of finding a feasible scheduling for software with real-time requirements for multi-core platforms. The concepts are evaluated using MECHATRONICUML and APP4MC, but can be adopted to other approaches. We provide an Eclipse bundle that contains our implementation and model files of the running example [15].

5 Related Work

Our approach is related to component-based approaches for CPS and to approaches for scheduling and safe deployment of CPS. [11, 23] survey component models in general, whereas [18] survey component models for CPS. Based on that, we state similarities and differences of approaches that consider at least partially concepts for partitioning, mapping, or deployment.

ProCom [10] provides a component model for the development of real-time systems in the automotive and telecommunication domains. ProCom provides a modeling language that is based on Final State Machines enriched by features of Timed Automata to compute (real-time related) dependencies of the model that can affect the scheduling. Additionally, ProCom provides a code synthesis [8]

that aims to preserve the semantics of ProCom at runtime. The code for every component is executed concurrently. In contrast to our approach, the resulting system is mainly event-triggered, which does not allow a static timing analysis like our approach. Since the component behavior is implemented directly in C, model checking and a model-driven segmentation is not possible. Nevertheless, in [8] a formalization of the generated code is provided.

MEMCONS (Model-based EMBEDded CONTROL Systems) [26] provides a model-driven framework for embedded systems and supports the interoperability with AUTOSAR and OSEK models. Since it follows the AUTOSAR methodology, it provides platform independent, component-based development of the system. It also provides an automatic approach for mapping tasks to multi-core ECUs. Furthermore, an analysis of timing constraints can be applied to the deployed system. In contrast to our approach, MEMCONS does not focus on verification of the PIM. Furthermore, the behavior of the software components is not specified model-driven and cannot be used for segmentation.

Further approaches focus on the modeling of (real-time) operating systems elements to improve the deployment of the software. In [24] they extend the DSL RTEPML (Real-time Embedded Platform Modeling Language) [9] to describe the behavior of the RTOS in a platform model, i.e., tasks and semaphores. Using this model for the refinement from PIM to PSM, model checking can be applied, which considers both the application behavior and the behavior of the underlying system. In contrast to our approach, concrete platform properties like the maximum transmission delay are not considered. Furthermore, distributed systems and multi-core ECUs are not taken into account. However, extending this approach to resource management on multi-core environments might be useful to improve our allocation approach.

Lukasiewicz et al. [25] present an approach to derive task priorities in event-triggered systems. The input for the algorithm is a task graph and a mapping. The task graph describes all tasks of the system and their communication. The mapping describes the assignment of tasks and messages to resources, e.g., ECUs or busses. The authors provide an algorithm to find optimal priorities for tasks in event-triggered systems. In contrast, we focus on time-triggered systems and do not consider priorities of tasks in our approach explicitly. Hence, this approach seems to be interesting to improve the task priorities in our approach.

There are also approaches regarding the semantic-preserving generation of source code for systems with real-time requirements, i.e., approaches for timed automata. In [4], code is generated for timed automata. The authors state that the code generation is platform independent since it also generates a runtime-system that handles task activation and system events. In contrast to our approach, the behavior of the tasks is not generated but implemented manually. Furthermore, the approach does not consider concepts for segmentation, partitioning, and mapping and, therefore, is not applicable for multi-core systems. In [28], the authors restrict the timed automata to deterministic features. Hence, invariants are not supported in this approach. In [22] on the other hand, they present an approach, where invariants are allowed in the specification. They do

not analyze if it all invariants can be guaranteed at runtime. In contrast to our approach, in both approaches properties of the target platform are not considered. Furthermore, both approaches do not consider distributed systems.

6 Conclusion and Outlook

In this paper, we presented a systematic approach that enables a step-wise, semi-automatic synthesis of behavioral models into a deterministic scheduling suitable for multi-core target platforms. We illustrated our approach based on an automotive, autonomous overtaking example and evaluated it based on the MECHATRONICUML and APP4MC platforms.

Firstly, we showed how runnables, runnable properties, and runnable dependencies are synthesized from RTSCs to derive a segmentation that allows parallel execution of software components. We identified limitations in our approach when using clocks across multiple states. Secondly, we introduced an approach for the allocation of runnables to interconnected multi-core ECUs. Especially, we identified and automatically derived necessary conditions an allocation has to fulfill in order to guarantee a valid scheduling. Thirdly, we introduced an approach that preserves verified real-time requirements on PIM level during the synthesis and in the resulting scheduling. In addition to [14], we presented advanced partitioning and mapping approaches considering all real-time constraints derived from former development steps. We used the APP4MC open tool platform to validate the correctness of the generated results.

In future work, we want to introduce a reachability analysis to cope with the mentioned limitations regarding clocks. Furthermore, we want to address dynamic scheduling in case of event-triggered systems. We also plan to extend the allocation constraints for ECUs that use cores with different processing capacities and by estimating the transmission time dynamically during the allocation. Finally, our goal is to combine all presented distribution and parallelization technologies along with a single example case study that provides the necessary constraints and reflects industrial needs.

Acknowledgment. This work was partially developed in the Leading-Edge Cluster ‘Intelligent Technical Systems OstWestfalenLippe’ (it’s OWL) and in the ITEA 2 AMALTHEA4public project (Nos. 01IS14029I and 01IS14029K). The IT’S OWL and the AMALTHEA4public projects are funded by the German Federal Ministry of Education and Research.

References

1. Alur, R., Dill, D.: A theory of timed automata. *Theor. Comput. Sci.* **126**(2), 183–235 (1994)
2. Amalthea: Deliverable: D3.1 concept for a partitioning/mapping/scheduling/timing-analysis tool. Technical report 3.4, Amalthea, January 2013
3. AMALTHEA4public Consortium: APP4MC Help Documentation (2017). <https://www.eclipse.org/app4mc/help/app4mc-0.8.0/index.html#section4.5.2.3>

4. Amnell, T., Fersman, E., Pettersson, P., Yi, W., Sun, H.: Code synthesis for timed automata. *Nord. J. Comput.* **9**(4), 269–300 (2002). <http://dl.acm.org/citation.cfm?id=779110.779112>
5. Austin, T., Larson, E., Ernst, D.: SimpleScalar: an infrastructure for computer system modeling. *Computer* **35**(2), 59–67 (2002)
6. AUTOSAR: Release 4.2 Overview and Revision History (2014). <http://www.autosar.org/specifications/release-42/>
7. Becker, S., et al.: The mechatronicuml design method - process and language for platform-independent modeling. Technical report tr-ri-14-337, Heinz Nixdorf Institute, Paderborn University, version 0.4, March 2014
8. Borde, E., Carlson, J.: Towards verified synthesis of ProCom, a component model for real-time embedded systems. In: Proceedings of the 14th International ACM Sigsoft Symposium on Component Based Software Engineering, CBSE 2011, pp. 129–138. ACM, New York (2011). <https://doi.org/10.1145/2000229.2000248>
9. Brun, M., Delatour, J.: Contribution to the software execution platform integration during an application deployment process. Ph.D. thesis, Ph.D. dissertation, École Centrale de Nantes, Nantes, France (2010)
10. Bureš, et al.: Procom-the progress component model reference manual. Mälardalen University, Västerås (2008)
11. Crnković, I., Sentilles, S., Vulgarakis, A., Chaudron, M.R.: A classification framework for software component models. *IEEE Trans. Softw. Eng.* **37**(5), 593–615 (2011)
12. Drozdowski, M.: Scheduling for Parallel Processing. *Computer Communications and Networks*. Springer, Berlin (2009). <https://doi.org/10.1007/978-1-84882-310-5>
13. Ferdinand, C., Heckmann, R.: aiT: worst-case execution time prediction by static program analysis. In: Jacquart, R. (ed.) Building the Information Society. IIFIP, vol. 156, pp. 377–383. Springer, Boston, MA (2004). https://doi.org/10.1007/978-1-4020-8157-6_29
14. Geismann, J., Pohlmann, U., Schmelter, D.: Towards an automated synthesis of a real-time scheduling for cyber-physical multi-core systems. In: Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development, MODELSWARD, vol. 1, pp. 285–292. INSTICC/ScitePress (2017)
15. Geismann et al.: Implementation and example models (2016). <https://trac.cs.upb.de/mechatronicuml/wiki/PaperModelsward17>, <http://workupload.com/file/rMP2kVG>
16. Gerking et al.: Domain-specific model checking for cyber-physical systems. In: Proceedings of the 12th Workshop on Model-Driven Engineering, Verification and Validation, MoDeVva 2015, vol. 1514 (2015). <http://ceur-ws.org/Vol-1514/>
17. Gill, N.S., Grover, P.S.: Component-based measurement: few useful guidelines. *SIGSOFT Softw. Eng. Notes* **28**(6), 1–6 (2003)
18. Hošek, P., Pop, T., Bureš, T., Hnětynka, P., Malohlava, M.: Comparison of component frameworks for real-time embedded systems. In: Grunke, L., Reussner, R., Plasil, F. (eds.) CBSE 2010. LNCS, vol. 6092, pp. 21–36. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13238-4_2
19. Höttger, R., Krawczyk, L., Igel, B.: Model-based automotive partitioning and mapping for embedded multicore systems. In: International Conference on Parallel, Distributed Systems and Software Engineering, ICPDSSSE 2015, vol. 2, pp. 2643–2649. World Academy of Science, Engineering and Technology (2015)
20. Kitchenham, B., et al.: Case studies for method and tool evaluation. *IEEE Softw.* **12**(4), 52–62 (1995)

21. Krawczyk, L., Wolff, C., Fruhner, D.: Automated distribution of software to multi-core hardware in model based embedded systems development. In: Dregvaite, G., Damasevicius, R. (eds.) ICIST 2015. CCIS, vol. 538, pp. 320–329. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24770-0_28
22. Kristensen, J., Mejholm, A., Pedersen, S.: Automatic translation from UPPAAL to C. Technical report, Department of Computer Science, Aalborg University (2004)
23. Lau, K.K., Wang, Z.: Software component models. *IEEE Trans. Softw. Eng.* **33**(10), 709–724 (2007)
24. Lelionnais, C., et al.: Formal behavioral modeling of real-time operating systems. In: Proceedings of the 14th International Conference on Enterprise Information Systems (ICEIS 2012), Wroclaw, Poland, vol. 2, June 2012. <https://hal.archives-ouvertes.fr/hal-01093794>
25. Lukasiwycz, F.N., et al.: Priority assignment for event-triggered systems using mathematical programming. In: Proceedings of the Conference on Design, Automation and Test in Europe, DATE 2013, EDA Consortium, San Jose, CA, USA, pp. 982–987 (2013). <http://dl.acm.org/citation.cfm?id=2485288.2485524>
26. Macher et al.: Filling the gap between automotive systems, safety, and software engineering. *e & i Elektrotechnik und Informationstechnik*, 1–7 (2015). <https://doi.org/10.1007/s00502-015-0301-x>
27. OMG: Unified Modeling Language, version 2.4.1. Superstructure Specification (2011). <http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/>
28. Opp, D., Caspar, M., Hardt, W.: Code generation for timed automata system specifications considering target platform resource-restrictions. In: Proceedings of the 7th International Conference on Computing and Information Technology 2011, pp. 144–149 (2011)
29. Pohlmann, U., Hüwe, M.: Model-driven allocation engineering. In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015), November 2015. ACM/IEEE (2015)
30. Tindell, K., et al.: Analysis of hard real-time communications. *Real-Time Syst.* **9**(2), 147–171 (1995). <https://doi.org/10.1007/BF01088855>
31. Van Solingen, R., et al.: The Goal/Question/Metric Method: A Practical Guide for Quality Improvement of Software Development. McGraw-Hill, London (1999)



A Model Based Approach for Complex Dynamic Decision-Making

Souvik Barat¹(✉), Vinay Kulkarni¹, Tony Clark², and Balbir Barn³

¹ Tata Consultancy Services Research, Pune, India
{souvik.barat, vinay.vkulkarni}@tcs.com

² Sheffield Hallam University, Sheffield, UK
t.clark@shu.ac.uk

³ Middlesex University, London, UK
b.barn@mdx.ac.uk

Abstract. Current state-of-the-practice and state-of-the-art of decision-making aids are inadequate for modern organisations that deal with significant uncertainty and business dynamism. This paper highlights the limitations of prevalent decision-making aids and proposes a model-based approach that advances the modelling abstraction and analysis machinery for complex dynamic decision-making. In particular, this paper proposes a meta-model to comprehensively represent organisation, establishes the relevance of model-based simulation technique as analysis means, introduces the advancements over actor technology to address analysis needs, and proposes a method to utilise proposed modelling abstraction, analysis technique, and analysis machinery in an effective and convenient manner. The proposed approach is illustrated using a near real-life case-study from a business process outsourcing organisation.

Keywords: Organisational decision making · Simulation
Model based approach · Conceptual model · Domain specific language

1 Introduction

Modern organisations constantly rely on decision-making to select suitable courses of action that help in achieving their goals [1]. An effective organisational decision-making calls for precise understanding of various aspects of organisation such as goals, organisational structure, operational processes and the historical data describing operational details along with execution log. The inherent characteristics of modern organisations that include the socio-technical characteristics [2], complex and dynamic organisational structure [3], significant uncertainty [4], and emergent behaviour [5] make the decision-making a complex endeavor *i.e.*, complex dynamic decision making (CDDM).

We posit that effective CDDM hinges on the availability of: (i) information required for decision-making in a structured and machine-interpretable form, (ii) suitable machineries to interpret the information, and (iii) a method to help identify the relevant information, capture it in model form, and perform *what-if* analyses. The current practice of organisational decision-making that relies heavily on human experts

typically working with primitive tools such as spreadsheets, word processors, and diagram editors *etc.* fares poorly on all the three criteria [6].

A wide range of Enterprise Modelling (EM) techniques, such as ArchiMate [7], IEM [8], MEMO [9], i* [10], BPMN [11], and System Dynamics (SD) [12], capture information of interest in a structured and/or machine interpretable form. They also support varying degree of analyses capabilities on a range of organisational aspects. However, they are found to be insufficient for CDDM [13, 14]. The actor languages and frameworks such as Kilim [15], Scala Actors [16], and Akka [17], in contrast, adopt the *actor model of computation* [18] to specify socio-technical characteristics. However, they are inadequate to express complex goal structure, organisational hierarchies, and behavioural uncertainty [13].

Therefore, it can be said that existing technological support can at best partly meet only two of the three requirements of effective CDDM *i.e.*, (i) the ability to conveniently capture the organisational goals, structure, behaviour, and their inherent characteristics and (ii) the ability to perform required analyses on available information. However, little is reported on how to use the relevant existing technologies, such as EM technologies and actor technologies, in a systematic manner for effective CDDM.

This paper presents a model-driven approach to capture necessary aspects of an organisation, such as goal, structure, and behaviour, along with their inherent characteristics, such as socio-technical characteristics and uncertainty, in a relatable and machine interpretable form and perform various *what-if* analyses leading to evidence-driven CDDM. In particular, this paper hypothesises that model-based simulation approach is an effective means to address CDDM and claims four contributions: (i) a conceptual meta-model that represents necessary and sufficient aspects of the organisation along with the inherent characteristics of CDDM, (ii) a simulation model that refines conceptual model for specific decision-making context, (iii) a pragmatic human-assisted technique to ascertain model validity, and (iv) a method to construct purposive simulatable models leading to *what-if* analyses for CDDM in a systematic manner.

The proposed conceptual meta-model caters to specification of *why, what, how, who, where* and *when* aspects [19], socio-technical characteristics as advocated in actor model of computation [18], and uncertainty [20]. The simulatable model advances the state-of-the-art actor technology [15–17] by supporting the notion of uncertainty and “time”. The proposed method refines the management view of decision-making advocated by Daft [3] while extending the modelling and model validation method advocated by Sargent [21] so as to realize a simulation based approach to CDDM.

The paper is organized as follows. Section 2 provides background by highlighting necessary tenets of CDDM and reporting brief overview of existing EM techniques and actor technologies. It also summarises notable gaps restricting adoption of EM techniques and actor technologies for CDDM. Section 3 presents model-driven simulation-based approach to CDDM. The approach is illustrated in Sect. 4 using a case study from business process outsourcing (BPO) domain. Section 5 discusses evaluation of the approach. The paper concludes with future work.

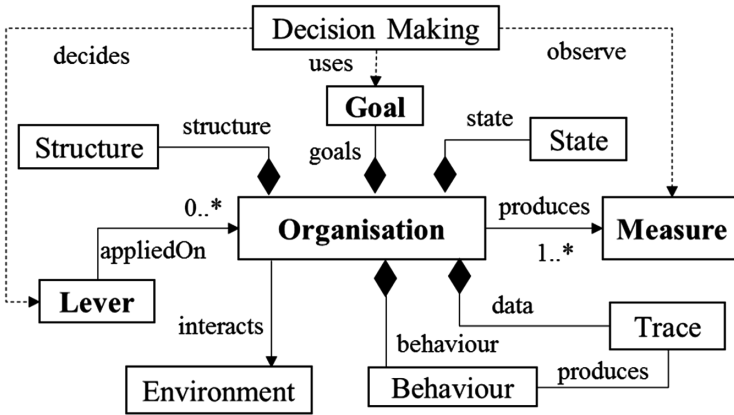


Fig. 1. Schema describing decision making concepts.

2 Background

This section presents the key requirements for affective CDDM and evaluates the state-of-the-art techniques and technologies with respect to these requirements.

2.1 CDDM Structure and Requirements

Decision-making is a continuous and indispensable activity for all organisations. It requires deep understanding of various aspects of an organisation. Zachman Framework [19] recommends six interrogative aspects namely *why*, *what*, *how*, *when*, *where*, and *who* as necessary and sufficient information to precisely understand an enterprise. Conforming to Zachman Framework, we visualize an organisation using a set of concepts as shown in the class diagram in Fig. 1 [22]. An *Organisation* has objectives or *Goals*, i.e., *Why* aspect, that it aims to achieve. A *Goal* is typically assessed by evaluating a set of performance indicators or *Measures* that are indicative of organisational effectiveness along several dimensions such as time to market, growth rate, customer satisfaction, employee happiness index, entry into new areas *etc.* Organisational effectiveness in an *Environment* (i.e., *where* aspect) is largely a function of its *Structure* (i.e., *What* and *Who* aspects) and *Behaviour* (*How* and *When* aspects). *Behaviour* induces *State* changes thus producing *Trace* (i.e., historical record of *States*) over a period of time. A *Lever* represents a possible course of action available to organisation. Typically, applying a lever results in modification of either operational parameters or *Goal* or *Behaviour* or any combination of the three thus leading to modifications to the *Trace*. Thus, decision-making is a loop involving evaluation of possible *Levers* so as to identify the most promising one – until the stated goal is achieved.

The conceptual structure of Fig. 1 though necessary is not sufficient for effective CDDM. The system of systems structure of an organisation means the decision making problem can be positioned at various levels of granularity spanning from mega to

macro to micro. This places additional demands of modularity and compositionality on the specification. As each [sub] system has own goals and the necessary wherewithal of achieving them, the specification needs to be capable of supporting *intentionality* and *autonomy*. As each of these [sub] systems operate over protracted time adapting constantly by responding to events taking place in their operating environments, the specification needs to be capable of supporting *reactive*, *temporal* and *adaptive* characteristics. Moreover, the specification must be capable of capturing the inherent *uncertainty*. Such a specification language along with its simulation engine seems necessary and sufficient infrastructure to support an iterative decision making loop wherein application of a *Lever* leads to modification of one or more *Measures* thus helping check whether a *Goal* (which is a sophisticated conditional expression over measures) is achieved or not [37]. A list of requirements of CDDM, as presented in [22], is summarised in Table 1.

Table 1. Requirements of CDDM [22].

	Requirement	Description
Aspect	Why	Goals, objectives and intentions of multiple stakeholders
	What	Structural Specification with complex hierarchy and interactions
	How	Behavioural specification with interactions
	Who	Stakeholders and human actors of the system
	Where	Information about location
	When	Temporality in behaviour and adaptation
Socio-technical Characteristics	Modularity	A system can be decomposed into multiple parts
	Compositional	Multiple parts should be composed to a consistent whole
	Reactive	Must respond appropriately to its environment
	Autonomous	Possible to produce output without any external stimulus
	Intentional	Intent defines the behaviour
	Adaptive	Adapt itself based on context and situation
	Uncertain	Precise intention and behaviour are not known a-priori
	Temporal	Indefinite time-delay between an action and its response
DC	Measure	Ability to specify what needs to be measured
	Lever	Ability to specify possible courses of action
Analysis	Machine Interpretable	Models that are interpretable by machine (<i>i.e.</i> , support for simulation/execution)
	Top-down and Bottom-up	Support for top-down and bottom-up modelling and simulation to support reductionist view and emergentism

From a methodology perspective, effective CDDM witnesses a curious dilemma. A system of systems structure involving autonomous [sub] systems indicates that organisation level *Goals* will be decomposed into various functional unit level Goals along the organisational *Structure* thus necessitating a top-down design approach. This implies that *Behaviour* of the organisation is known and hence specifiable. However,

given the complexity of modern organisations and the inherent uncertainty, it is almost impossible to know the overall behaviour of organisation. The behaviour is typically known only for highly localized contexts i.e., functional units thus suggesting a bottom-up design approach wherein the overall organisation behaviour emerges from the behaviour of its interacting functional units. As a result, the specification language and analysis techniques need to be cognizant of top-down and bottom-up approach [23, 24] as described in Table 1. Also, effective CDDM calls for a method providing help with: (i) evaluating if the desired *Goal* is achieved, (ii) identifying the most appropriate *Lever* amongst many candidates, and (iii) applying the *Lever*.

2.2 Review of State of the Art and Practice

The state-of-the-art specification and analysis techniques approach the decision-making problem in two ways namely: data-centric approach and model-centric approach. The data-centric approach makes use of sophisticated AI-based pattern recognition and predictive analysis techniques on relevant past data or *Trace* to predict future outcomes. This approach has worked well when *Trace* of an *Organisation* is comprehensive and the future is typically a linear extrapolation of the past. However, the two conditions are increasingly not being met for modern large enterprises thus leading to inappropriate decisions for emerging business context¹.

The model-centric approaches, in contrast, characterise the real organisation in the form of representative models which span across a wide spectrum. At one extreme of the spectrum are models that provide a well-defined structure for the organisational aspects of interest and rely on a variety of visualisation techniques to help humans obtain the desired understanding of the organisation. For instance, ArchiMate [7] is one such specification. At the other extreme of the spectrum are machine interpretable and/or simulatable specifications. They are capable of precise analyses for one or limited aspects. For instance, BPMN (Business Process Modelling and Notation) [11] analyses and simulates the behavioural aspect, i* [10] analyses the high level goals and objectives, and System Dynamic model simulates dynamic behaviour of the system. The multi-modelling and co-simulation environments, such as DEVS (Discrete EVent system Specifications) [25], AA4MM (Agent & Artifact for Multi-Modeling) [26], AnyLogic [27] and MEMO (Multi-perspective enterprise modeling) [9] technology, demonstrate further advancements by supporting the analysis of multiple aspects. Principally they adopt a top-down [23] approach to help analyse enterprises where the mechanistic world view holds. On the other hand, the languages and specifications advocating an actor model of computation [18] and agent-based systems [28] support emergentism [24] through bottom-up simulation. They fare better in analysis of systems comprising of adaptive and socio-technical elements.

Thus, the above mentioned techniques and technologies capture only a fragment of what ought to be captured and analysed for effective CDDM as illustrated in Table 1 [13]. In particular, the enterprise modeling languages are incapable of specifying uncertainty as well as emergent behaviour, and actor/agent languages are inadequate to

¹ <https://hbr.org/2014/09/9-habits-that-lead-to-terrible-decisions>.

conveniently express required characteristics such as the complex goal structure, organisational hierarchies, and behavioural uncertainty [22]. Moreover, EM specifications and actor based languages fall short as an intuitive and closer-to-the-problem specification as they are not designed for CDDM.

From a methodological viewpoint, the goal specification languages such as i* [10] and EKD [29] advocate a top-down method. EM languages such as ArchiMate, MEMO, and 4EM [30] advocate a top-down method and a globalized view of the system to represent the *Goal*, *Structure* and *Behaviour* of organisation in an integrated manner. BPMN [11] and SD model [12] predominantly support top-down approach and reductionist view of analyses [36]. On the other hand, actor languages and frameworks [15–17] advocate localised view, bottom-up approach, and emergentism. The reported methodological advancements also fail to support desired design principles. For example, DESIRE (DEsign Specification of Interacting REasoning components) [31] and MEMO based decision-making process [32] propose top-down model and reductionist *what-if* analysis. On the other hand, [33] advocates bottom-up approach using Belief-Desire-Intention (BDI) paradigm. Thus, there exists no single approach capable of combining top-down/bottom-up [23] design principle, reductionist/emergentism analysis techniques [24], and localized/globalized perspectives as desired. Moreover, the existing approaches are also found wanting in terms of ensuring model validity [21] and correlating with the management view of decision-making.

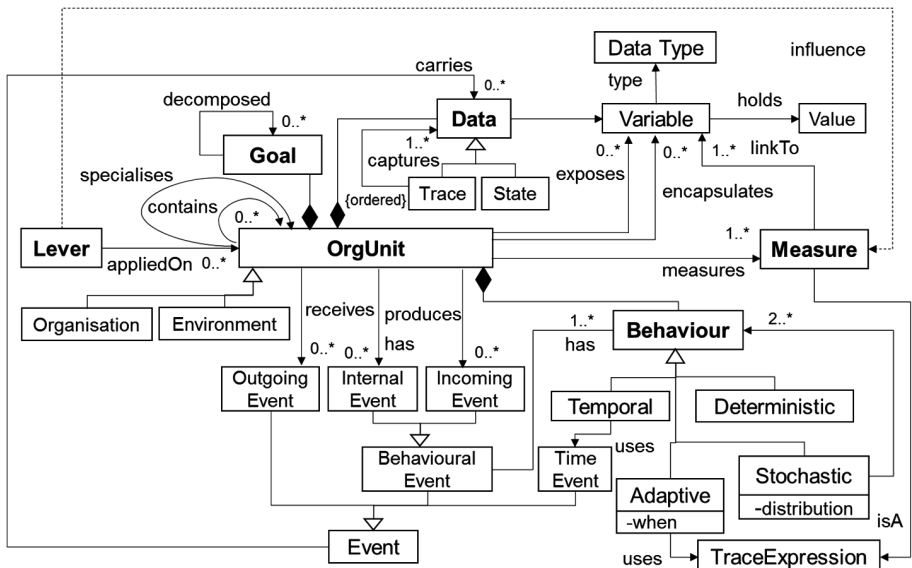


Fig. 2. CMMModel – a metamodel to represent organisation.

The next section describes our approach that addresses some of the essential specification limitations, overcomes inadequacy of analysis needs, and bridges the existing gap in methodical support.

3 Approach

Our approach to CDDM uses a model-based representation of organisation capable of supporting *what-if* simulation with a comprehensive design and analysis method providing the integration glue. In particular, we propose three artefacts that include: (i) a conceptual meta-model, termed as CMMModel, to represent relevant aspects of an organisation along with the characteristics described in Table 1, (ii) a simulatable model, termed as ESLMModel, along with simulation machinery to support analyses needed for CDDM, and (iii) a method to help construct these models so as to perform *what-if* analyses leading to evidence-driven CDDM.

3.1 Conceptual Model

The CMMModel meta-model is depicted in Fig. 2. As shown in the Figure, the key abstraction of CMMModel is *OrgUnit* that represents an autonomous self-contained functional unit having high internal coherence and low external coupling. Each *OrgUnit* has its own *Goal*, contains *Data*, deals with a set of interacting *Events*, and may have specific *Behaviour*. The Goal represents the intention or objective of an *OrgUnit*. A Goal can be decomposed into sub-Goals, sub-sub-Goals to represent hierarchical goal structure. Data captures the current *State* and sequence of historical states, *i.e.*, *Trace*, using a set of typed entity *Variables*. An *OrgUnit* may encapsulate and/or share *Data* by encapsulating and/or exposing *Variables*. *OrgUnit* responds to three kinds of *Events* namely *OutgoingEvent*, *BehaviouralEvent* and *TimeEvent*. The *OutgoingEvents* are triggered from an *OrgUnit* as part of its reactive behaviour. Each *OutgoingEvent* specifies the *Data* that it carries while reacting to an *Event*. The *BehaviouralEvent* specifies behaviour that is a response to an event and the *Data* it consumes. The *BehaviouralEvent* is further classified into two types namely *InternalEvent* and *IncomingEvent*. The *IncomingEvents* are consumed by *OrgUnit*, and the *InternalEvents* are the events that are internal to an *OrgUnit*. The *TimeEvent* is a special event that represents the concept of “Time” such as “Day”, “Month” or a “Year”.

The *Measure* and *Lever* of an *OrgUnit* represent the *Measure* that an *OrgUnit* owns and the *Lever* that are relevant for an *OrgUnit*. Essentially, a *Measure* can be represented using a set of *Variables* and the *Lever* describes the change specification of *Variables*, composition relationships, *Behavioural* specification and/or *Goals*. We visualise the notion of organisation and its environment as specialised *OrgUnit* namely *Organisation* and *Environment* as shown in Fig. 2.

By the virtue of being composable, *OrgUnit* abstraction is capable of modelling the system of systems nature of modern organisation. The composability can be specified using *contains* relationship. The meta-model advocates four kinds of *Behaviour* namely *Deterministic*, *Stochastic*, *Temporal* and *Adaptive*. The *Deterministic* behaviour describes the behaviour which is known with certainty. Essentially, the *known known* kinds of behaviour [20] can be specified using *Deterministic Behaviour*. The *Stochastic* behaviour describes uncertain *Behaviour* or *known unknown* kind of behaviour [20]. We use probabilistic distribution to specify *Stochastic Behaviour*. The *Temporal Behaviour* describes the temporal delays in interaction pattern, and the *Adaptive Behaviour* describes adaptation rules by describing *what* will change *when*.

The proposed meta-model is grounded into a set of existing concepts. The modularisation and unit hierarchy are taken from the notion of component abstraction. The goal-directed reactive and autonomous behaviour can be traced to actor behaviour [18, 34]. Defining states in terms of a type model is borrowed from UML. An event driven architecture is introduced for reactive behaviour. The concept of intentional modelling [10] is adopted to enable specification of goals. The behavioural classification and uncertainty is defined from the notion of uncertainty defined by Rumsfeld [20].

We argue that CMMModel meta-model realises the structure defined in Fig. 1 and satisfies the requirements stated in Table 1. Event definition, Data, and OrgUnit structure together specify the *what* aspect, OrgUnit help specify the *who* and *where* aspects, Goal specification specifies the *why* aspect, and Behaviour specifies the *how* and *when* aspects. The concept of OrgUnit ensures modularity and encapsulation, the Event helps to specify reactive nature, InternalEvent and TimeEvent collectively specify the autonomous behaviour, Stochastic behaviour helps in specifying uncertainty, the Temporal behaviour and TimeEvent specify the temporal behaviour, and Adaptive behaviour is capable of specifying the adaptive nature of an OrgUnit. We argue that the *contain* relationship of OrgUnit and OrgUnit specific localised Behaviour definition help in bottom-up design, whereas the *contain* relationship of OrgUnit, Goal *decomposition* relationship, and an ability to share Variables using *exposes* relationship help in top-down design. The next section introduces a specification that has capability to represent the information captured using CMMModel in a simulatable form.

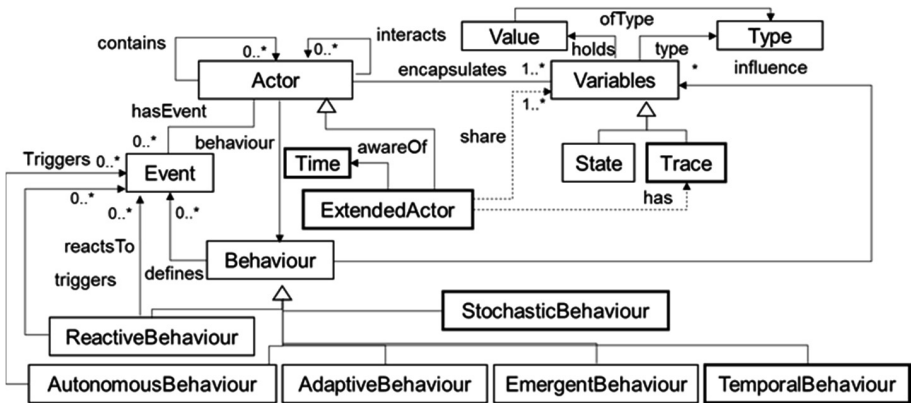


Fig. 3. ESL meta-model (ESLMMModel).

3.2 Simulatable Model

We extend the notion of traditional *actor* definition [34] to specify enterprises. The adopted concepts from actor model of computation and proposed extensions are depicted using a meta-model, termed as ESLMMModel, in Fig. 3. The extended concepts are highlighted with bolded boxes and extended associations are represented using

Table 2. Conceptual mapping from CMMModel to ESLMModel.

CMMModel	ESLMModel	CMMModel	ESLMModel
OrgUnit	ExtendedActor	Variable	Variable
Data	Variables	Trace	Actor variable
Goal	Expression over actor variables	Deterministic	DeterministicBehaviour
Event	Event	Stochastic	StochasticBehaviour
Measure	Expression over actor variables	Temporal	TemporalBehaviour
Lever	ESL specification	Adaptive	AdaptiveBehavioural

dotted lines. The *Enterprise Simulation Language* (ESL)² provides an implementation for ESLMModel.

As shown in Fig. 3, the notion of traditional *Actor* encapsulates its *State*, has specific *Behaviour* and interacts with other *Actors* using a set of *Events*. The State of an Actor is defined using a set of typed *Variables* where each Variable holds *Value*. The Behaviour of an Actor principally represents four kinds of behavioural patterns namely reactive behaviour, autonomous behaviour, adaptive behaviour and emergent behaviour. ESLMModel represents supported behavioural patterns using four kinds of Behaviour namely *ReactiveBehaviour*, *AutonomousBehaviour*, *AdaptiveBehaviour* and *EmergentBehaviour*.

The ESL extends the notion of traditional Actor along four dimensions: (i) representation of historical state information or *Trace*, (ii) the notion of “Time”, (iii) the notion of shared Variables that breaks pure encapsulation without compromising the correctness of state space of an actor, and (iv) the notion of uncertainty. The extensions (i), (ii) and (iii) are introduced using a specialised Actor entity named *ExtendedActor* and the extension (iv) is introduced as a specialised behavioural type named *StochasticBehaviour* in the ESLMModel (see Fig. 3). The notion of “Time” helps specify temporal behaviour that we represent using a specialised Behaviour named *TemporalBehaviour* in ESLMModel.

ESL provides standard language constructs namely assignment, expression evaluation, loop, recursion, message passing, *etc.*, to express Deterministic Behaviour. Stochastic Behaviour is expressed using ‘*probably(p) x y*’ construct that evaluates to *x* in *p%* of cases and otherwise to *y*. *ReactiveBehaviour* reacts to an Event or a set of Events, *AutonomousBehaviour* is typically triggered based on state Variables and/or Time, and *AdaptiveBehaviour* has a conditional expression over State and Trace Variables. The *EmergentBehaviour*, on the other hand, remains unspecified.

We propose a set of transformation rules to derive ESL specification from CMMModel. The OrgUnit and its specialisation, *i.e.*, *Organisation* and *Environment*, map onto ExtendedActor, interactions among OrgUnits map onto event specifications, and OrgUnit Variables map onto Variables of ExtendedActor. Measure maps onto Variables of ExtendedActors, Goal maps onto an expression over Variables of ExtendedActors, and the behavioural descriptions of OrgUnit map onto the behavioural specifications of ExtendedActors. The conceptual mapping from CMMModel to

² <https://www.gitbook.com/book/tonyclark/esl/details>.

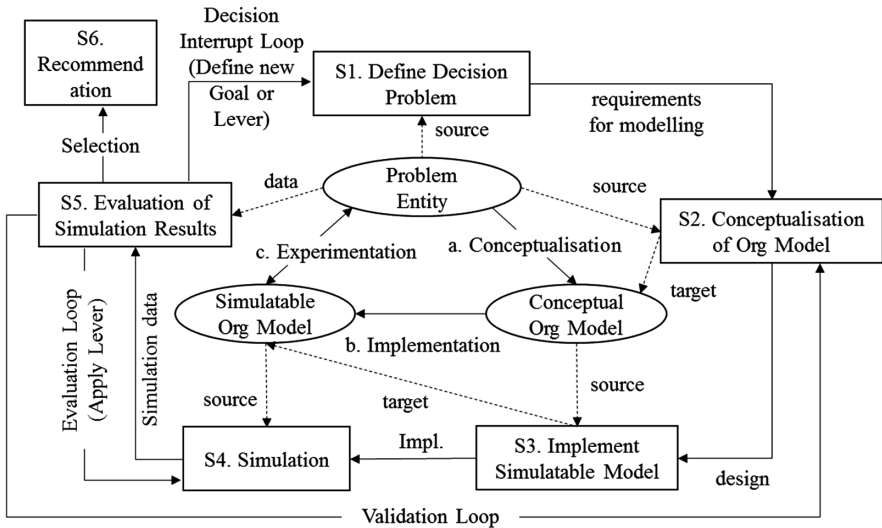


Fig. 4. Overview of modelling and simulation method.

ESLMMModel is illustrated in Table 2. Next section describes a method to construct models using CMMModel, transform the constructed model into ESL specification, and perform *what-if* analysis in a systematic manner.

3.3 Method

We propose an integrated and iterative method to effective CDDM that comprises of three essential activities: (i) construction of a simulatable model from available information of an organisation, (ii) ascertain model validity, and (iii) simulate model for *what-if* analyses leading to evidence-driven CDDM. The proposed method contains six steps namely *Define Decision Problem* [S1], *Conceptualisation of Organisation Model* [S2], *Implement Simulatable Model* [S3], *Simulation* [S4], *Evaluation of Simulation Results* [S5], and *Recommendation* [S6] as shown in Fig. 4. Step S1 formalises the decision problem and defines the scope for *what-if* scenario playing by describing the Goals, Measures and Levers of an Organisation. Step S2 conceptualises a purposive model that represents a real system for decision problem defined in S1. Step S3 transforms the conceptual model into a simulatable model. Step S4 simulates the scenario defined in step S1. Step S5 evaluates the simulation results with step S6 providing recommendations.

Conceptually the proposed method realises the modelling and validation method proposed by Sargent in [21] (henceforth referred as *M&V Method*) and adopts decision-making techniques recommended in management science [3]. From M&V Method, we adopt the notion of three representations namely *problem entity*, *conceptual model* and *computerized model*, and a two-step model construction process that includes *Conceptualisation* and *Implementation* steps to transform a real-life problem into valid analysis model as shown in Fig. 4. We also adopt the *operational validity*

[21] described in M&V Method to ascertain model validity. From management science, we adopt an iterative exploration of decision alternatives as recommended in [3] and the concept of decision interrupts [35] to explore decision alternatives that emerges while evaluating other decision alternatives.

In agreement with M&V Method, we consider the problem entity is the real organisation, the conceptual model is a purpose specific conceptual model that is necessary and sufficient to represent it for decision-making, and the computerised model is a machine interpretable equivalent of the conceptual model, *i.e.*, simulatable model. From a method perspective, the *Conceptualisation* step constructs a conceptual model from problem entity description (typically described in natural language), and *Implementation* step transforms the conceptual model into a simulatable model so as to use model-based simulation. The detailed activities of five method steps of Fig. 4 are illustrated below:

Conceptualise Organisation Model [S1]: A decision problem typically starts with a high-level *Goal* or objective of an organisation. It should be possible to decompose a high-level Goal into sub-Goals, sub-sub-Goals etc., to the desired level of granularity. It should be possible to identify a set of variables that need to be observed in order to determine whether the finest-level goal is met or not, *i.e.*, *Measures*. It should be possible to identify a set of course of actions or *Levers* that may influence the given set of Measures. The method step *Define decision problem* defines the Goals, Measures and Levers of an Organisation from problem entity description using three sub-steps namely *Goal Definition*, *Measure Identification* and *Lever Identification*.

The *Goal Definition* sub-step uses a top-down approach to define goals and goal decomposition structure. *Measure Identification* sub-step identifies *Measures* for all leaf-level Goals of constructed goal model. We use i^* specification to visualise the goals of a decision problem. We represent Goals using the *Soft Goal* of i^* notation, Measure using i^* *Task* of i^* notation, and *Goal-to-Measure* relationships using Task-Goal dependency relationship of i^* notation [10].

The sub-step *Identify Levers* focuses on two activities: (i) identify a set of Levers that may impact identified Measures, and (iii) formulate a table, termed as *decision table*, by considering the identified Levers as rows and Measures as illustrated in Fig. 7 in Sect. 4.

Conceptualisation of Organisation Model [S2]: This step captures the *Structure*, *Behaviour*, *State* and *Trace* of an organisation and overlays the Goals, Measure and Levers identified in method step S1 using OrgUnit abstraction defined in CMMModel (as depicted in Fig. 2). Essentially this method step performs four activities namely (i) *Identify OrgUnits*, (ii) *Define OrgUnit*, (iii) *Define GM-L*, and (iv) *Specify Behaviour*. Activity *Identify OrgUnits* identifies prospective OrgUnits such as organisational units, sub-units, stakeholders, resources, and environment from problem entity. Activity *Define OrgUnit* forms OrgUnits by specifying *Variables* to represent State and Trace information, and the *Events* that help interacts with other OrgUnits. It also identifies containment relationship to describe composition and decomposition relationships of identified OrgUnits. In general, the activity *Identify OrgUnit* starts with organisation as an OrgUnit, and iterates over activity *Identify OrgUnit* and activity *Define OrgUnit* by navigating the decomposition and/or composition relationships.

Essentially, it uses a middle-out approach that combines top-down and bottom-up design principles.

The activity *Define GM-L* identifies the Goals that an OrgUnit owns, the *Measures* that it can produce, and the Levers that can be applied on it. The activity *Specify Behaviour* captures the behavioural specification of identified OrgUnits.

Implement Simulation Model [S3]: This method step converts a Conceptual Organisation model defined using CMMModel into machine interpretable specification, i.e., ESL specification. Essentially, S3 transforms all OrgUnits into ExtendedActors by applying transformation rules defined in Table 2.

Simulation [S4]: We use ESL based simulation to analyse *what-if* scenario formulated in method step S1. This step simulates the simulatable organisation model (with or without Lever), observes Measures from a simulation run, and captures results in a row of *decision table* formulated in method step S1.

Evaluation of Simulation Results [S5]: This step evaluates simulation results captured in decision table. Human expert interprets the simulation results triggering one of the following possibilities: (i) initiate a *Validation Loop* that iterates method steps S2-S3-S4-S5 in case simulation results of known scenario don't match the expected outcome (i.e., *operation validity* is not satisfied), (ii) explore next Lever of a decision table by triggering an *Evaluation Loop* that iterates method steps S5-S4-S5, (iii) select the best possible Lever once all levers are evaluated through simulation (i.e., S5 to S6 transition), (iv) identify a new Lever i.e., add a new entry in decision table and reiterate the overall method using *Decision Interrupt Loop* described in Fig. 4.

Recommendation [S6]: This step recommends one or more Levers that can be implemented in real organisation.

3.4 Validation

Our method uses a validation loop that iterates over method steps S5-S2-S3-S4-S5 and compares experimental results with real or predicted data to ascertain model validity. We consider operational graphics [21], i.e., graphical representation of *Measures* as a basis for evaluation, and rely on human experts to certify the validity. For model validation, we rely solely on operational validity through manual certification of simulation results of known scenarios. Other validation techniques, such as data validity or conceptual validity, while being effort and time intensive, provide no additional certainty as discussed in [21]. We next illustrate the proposed method using a real-life decision-making scenario.

4 Illustration

This section presents a problem entity from business process outsourcing (BPO) industry and illustrates the execution of proposed method along with their outcomes.

4.1 Problem Entity

In BPO, a class of organisations, termed as *customers*, outsource their business processes to another set of organisations, which is termed as *vendors*. Customers outsource their business process for a variety of reasons such as reducing *Cost* (C), increasing *Efficiency* (E), bringing about a major transformation, *i.e.*, *Delight* (D). The vendors offer value-added services to their customers and earn revenues while servicing outsourced business processes. Considering the accruable business benefits of vendors, the outsourced business processes are classified into three broad buckets namely *Sunrise* (SR), *Steady* (ST) and *Sunset* (ST). The Transcript Entry process of Healthcare verticals is one of the early adopters of BPO and has derived almost all potential benefits accruable from outsourcing (known as *Sunset*). On the other hand, IT Infrastructure Management process being a late adopter of BPO, has a large unrealized potential to be tapped (known as *Sunrise*). And there are processes such as Help Desk, Account Opening, Monthly Alerts *etc.*, that fall somewhere in between the two extremes as regards benefits accrued from BPO (known as *Steady*). Thus, the outsourced business processes of the BPO industry can be described using a 3×3 matrix as depicted in Fig. 5 [22].

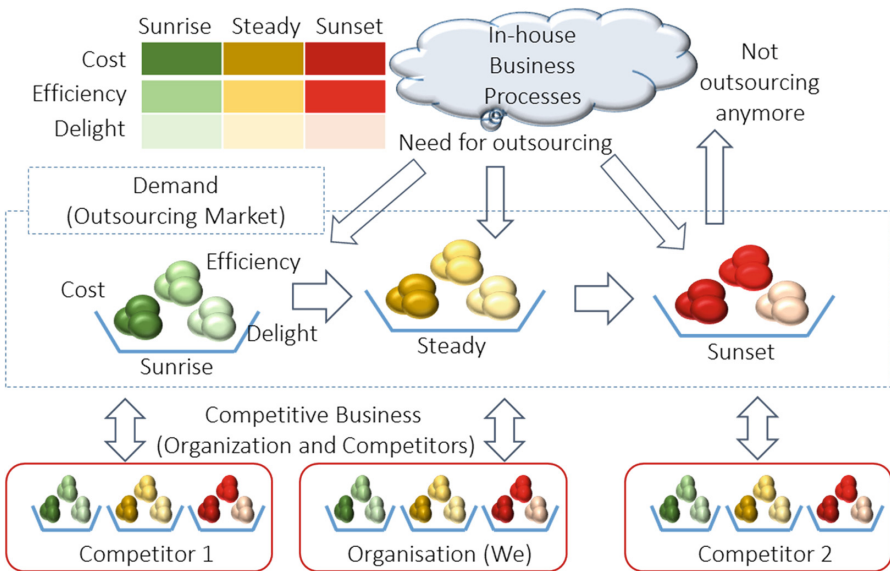


Fig. 5. Overview of business process outsourcing scenario [22].

The business-as-usual (BAU) operational process of a BPO is largely limited to a set of interactions between customers and vendors. A *customer* publishes RFP (Request For Proposal) with an intension to outsource a business *process*. Interested *vendors* bid for RFP. Typically, factors such as *Quadrant* (*i.e.* ranking as per independent agency such as analysts), *FTE Count Range* (*i.e.* Full Time Employees to be deployed on the

outsourced process), *Billing Rate Range* (i.e. per hour rate of FTE), *Organisation Size* (the number of employee) and *Track Record* (i.e., familiarity with the processes being outsourced), influence who wins the bid. The soft issues such as *Market Influence* (i.e. perception of the market as regards delivery certainty with acceptable quality), the rapport with the vendor *etc.*, also play a part in bid evaluation. In addition to these known factors there could be some uncertainty in bid evaluation criteria (in other words, bid evaluation criteria can't be fully known a-priori).

It is common observation that BPO outsourced business process engagements come up for renewal after few years (typically 3 to 5 years). A customer may renew the contract with the existing vendor on modified terms (typically advantageous to the customer) or may opt for rebidding. Factors influencing the renewal decision are reduction offered in *FTE Count*, *Billing Rate*, number and degree of escalations, perception that the external agent has as regards ability to meet the process engagement requirements, inherence uncertainty, etc. Contracts that fail to get renewed become candidates for later bidding. Figure 5 [22] shows an overview of BPO industry. The interaction pattern between customer and vendor is depicted in Fig. 6 [22].

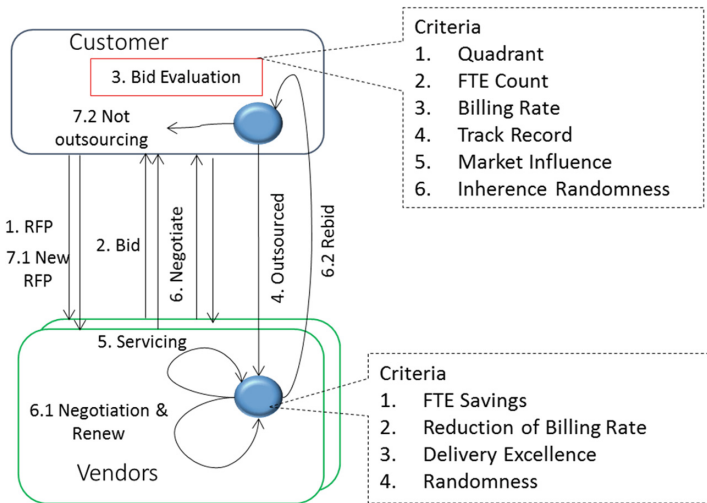
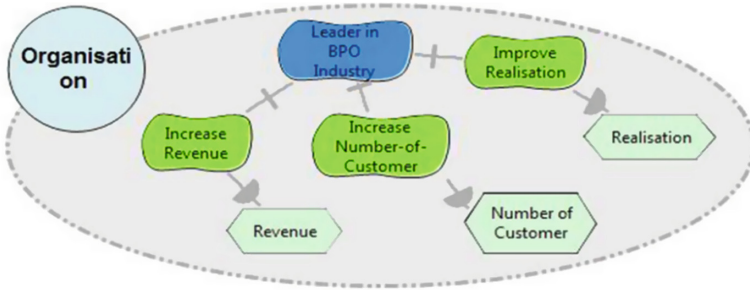


Fig. 6. Interactions and behaviours [22].

Given the above scope or a problem entity, the vendors mostly explore the decision-making problems that include: Will continuation with the current strategy keep “Me” viable ‘n’ years hence? What alternative strategies are available? How effective will a given strategy be? By when will a given strategy start showing positive impact? *Etc.*



(a): Goal decomposition and Measures

Lever	Revenue	Number of Customer	Realisation
No Lever			
Improve Skillset of Existing Employee			
Robotic Process Automation (RPA)			

(b): Decision Table

Fig. 7. Output of method step define decision problem.

In this paper, we consider a BPO vendor who would like be the leader in BPO industry with respect to the revenue, market share, and *realisation* (where the term *realisation* represents the revenue earned by each employee per hour). The next subsections describe the execution of method steps depicted in Fig. 4 and their outcomes.

4.2 Define Decision Problem

The proposed method starts with a method step *Define Decision Problem* [S1] that formulates goal models and a decision table. We consider, a vendor, termed as “WE” vendor, aims to be the “Leader in BPO Industry”. The method step S1 decomposes “Leader in BPO Industry” Goal of “WE” vendor into three sub-Goals namely “Increase Revenue”, “Increase Number-of-Customer”, and “Improve Realisation”. It identifies three Measures namely “Revenue”, “Number of Customers”, and “Realisation” to assess three leaf-level Goals. The primary goal, goal decomposition structure and associated Measures are depicted in Fig. 7(a).

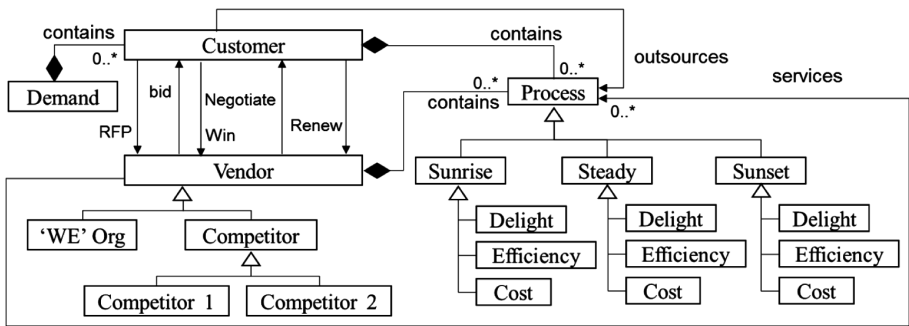
The method step S1 also identifies possible Levers that may influence the Measures and thus Goals. In this paper, we consider two Levers namely “Improve skillset of existing employee” and “Introduce Robotic Process Automation (RPA)” as illustration. Identified Levers and Measures are shown in a form of decision table in Fig. 7(b).

4.3 Conceptualisation of Organisational Model

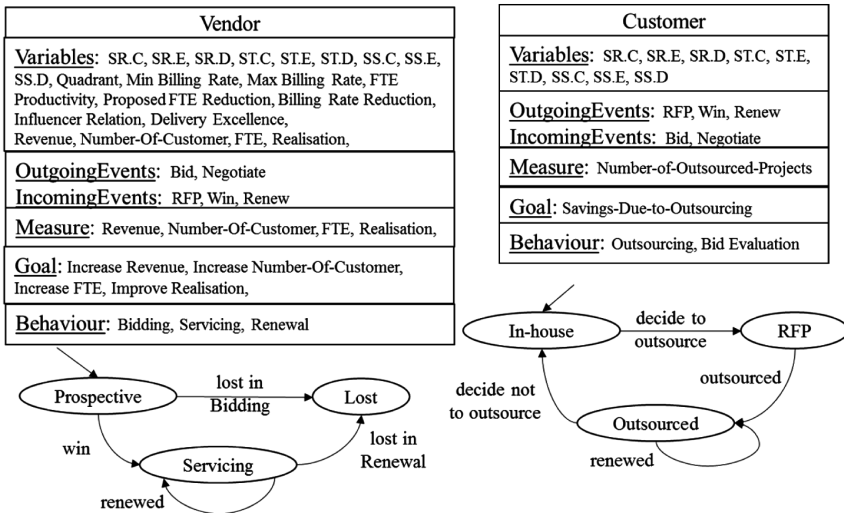
Method step S2 iteratively forms *Conceptual Organisation Model* from *problem entity* using four activities namely *Identify OrgUnit*, *Define OrgUnit*, *Define GM-L* and *Define Behaviour*. The activity *Identify OrgUnit* initially identifies three key OrgUnits namely “Customer”, “Vendor”, and “Process”. The next activity *Define OrgUnit* captures

structural relationships, Variables, and Event definitions of three OrgUnits. The Variable, IncomingEvent and OutgoingEvent of Vendor and Customer OrgUnits are illustrated in Fig. 8(b). Essentially the Vendors OrgUnit has a set of Variables to represent portfolio baskets (i.e., flattened out 3×3 matrix), the characteristics Variables such as *Quadrant*, *Min Billing Rate*, *Max Billing Rate*, *FTE Productivity*, *Proposed FTE Reduction* (during process engagement renewal time), *Proposed Billing Rate Reduction* (during project renewal time), *Influencer Relationship*, *Delivery Excellence* of the vendor OrgUnit. The OrgUnit also captures the state Variables that indicate Measure of Vendor OrgUnit such as *Revenue*, *Number-Of-Customer*, and *Realisation*.

The outcome of the iterative loop involving two activities namely *Identify OrgUnit* and *Define OrgUnit* is depicted using a class diagram in Fig. 8(a). As shown in the figure, several new OrgUnits are identified and elaborated over iterations. The “Process” OrgUnit is specialised into nine OrgUnits to represent business processes



(a) Structure of of BPO Industry using CMMModel



(b) Definition of Vendor and Customer OrgUnits

Fig. 8. Conceptual organisation model of BPO industry.

described using a 3×3 matrix of Fig. 5. The Vendor is specialised into two entities namely “WE” vendor and “Competitor” vendor. The “WE” vendor represents a vendor under consideration, and the “Competitor” vendor represents the competitor vendors of “WE” vendor. There could be several competitors who adopt a range of strategies to compete in BPO industry. We consider two types of competitors namely “Competitor 1” and “Competitor 2” as shown in Fig. 8(a). The other relationships such as Customer “contains” various kinds of Processes, Vendor “outsources” Processes, Vendor “contains” a set of Processes and Vendor “services” Processes are defined in this method step. The interactions patterns between Customer and Vendors are also become explicit in this method step. The relationships and interaction patterns between OrgUnits are illustrated in Fig. 8(a).

The next activity *Define GM-L* defines the Goal and Measures of identified OrgUnits, and map them with the Goals and Measures of problem entity that are identified in method step S1. In this example, the “WE” vendor owns the goals, measures and leavers defined in S1 method step. The generic Goals of Vendor and Customer are depicted in Fig. 8(b).

The remaining activity of the method step Conceptualisation of Organisation Model [S2] is *Define Behaviour*. This activity iterates over identified OrgUnits to define their behaviours. The typical Behaviours of Vendor and Customer are depicted in the form of state-machines in Fig. 8(b).

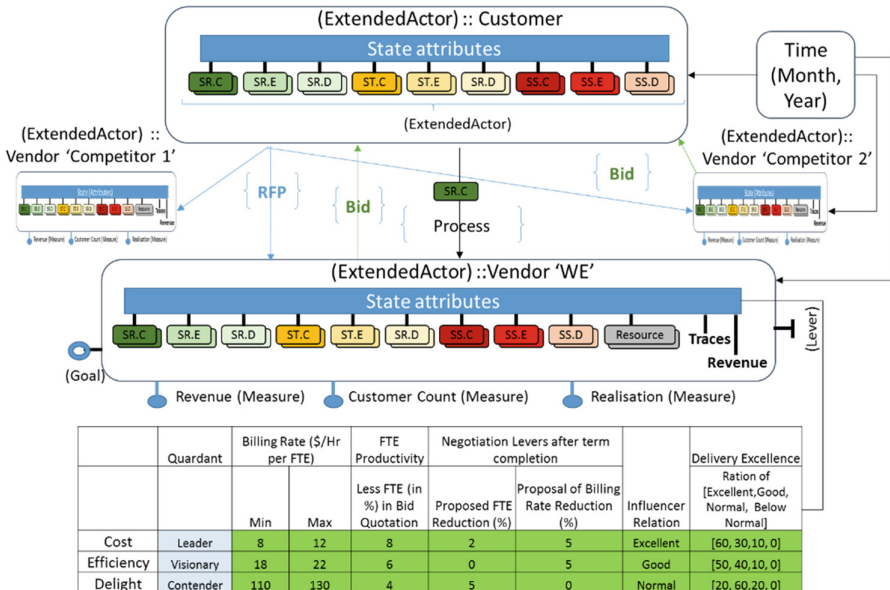


Fig. 9. Realisation of BPO scenario using ESLMModel [22].

4.4 Implement Simulatable Model

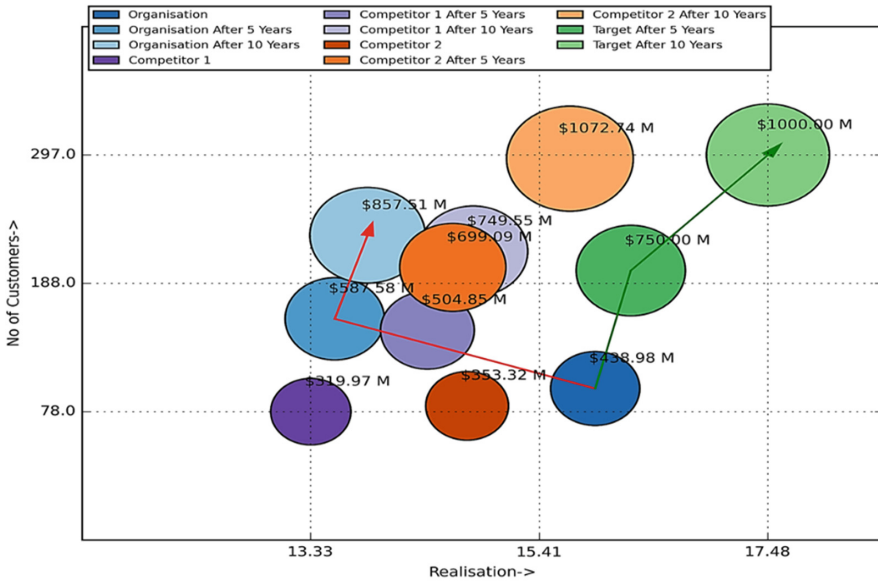
Method step *Implement Simulatable Model* (manually) translates the information captured in method step S1 and method step S2 that collectively describe the Goal, Measure, Lever, Structure, Behaviour, State and Traces of OrgUnits into ESL specification by applying the transformation rules defined in Table 2.

A representative ESLModel that contains two key ExtendedActors namely Customer and Vendor is shown in Fig. 9 [22]. The Customer ExtendedActor comprises nine variables where each variables represents a bag of outsourced process of specific type from the business process classification *i.e.*, {SR, ST, SS} X {C, E, D}. The vendor ExtendedActor comprises Variables of Vendor OrgUnit that include State variables, Trace variables and the variables that represent Measures (as shown in Fig. 9). The Customer and Vendor ExtendedActor also implement the state-machines depicted in Fig. 8(b).

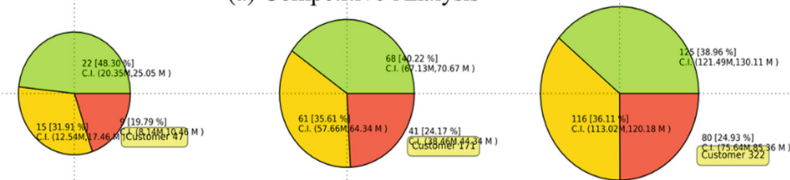
The table in Fig. 9 shows the initial characteristics of “WE” ExtendedActor. We make these Variables configurable to attenuate their values, thus these Variables also act as Lever specification in this example. As shown in the figure, a Vendor is equipped with a set of negotiation levers namely, the range of Billing Rate, range of FTE Productivity (percent reduction possible in number of full time employees), range of FTE Reduction (reduction possible during renewal of a contract), range of Billing Rate Reduction (reduction possible in billing rate during renewal of a contract), Influence Relation and Delivery Excellence. The Influence Relation is a qualitative characteristic that is quantified using four weighted labels namely ‘Excellent’, ‘Good’, ‘Normal’ and ‘Not Good’. Value of Delivery Excellence attribute is a probability distribution. For instance, “WE” ExtendedActor is confident of delivering ‘Excellent’ quality on 60% of Cost kind of BPO projects won. The values for ‘Good’, ‘Normal’ and ‘Below Normal’ quality for this kind of BPO projects are 30%, 10% and 0% respectively. Therefore, one can model different kinds of vendors by setting appropriate values to the initial setting. The “Competitor” ExtendedActors are also modelled on the same lines as “WE” ExtendedActor.

The Customer ExtendedActor raises RFP events for outsourcing project. Each RFP event is characterized by the kind of process being outsourced (*i.e.*, SR or ST or SS), the objective for outsourcing (*i.e.*, C or E or D), size of the process in terms of FTE count, and the desired billing rate. Interested vendors respond to the RFP event by picking suitable values from their characteristics at random. Bid evaluation function is a weighted aggregate of the various elements of RFP response and a random value to capture effect of inherent uncertainty. The vendor with the best evaluated value wins the outsourcing process which gets executed as defined by the characteristics of the particular vendor. Essentially, an outsourcing process ExtendedActor moves from customer ExtendedActor to a vendor ExtendedActor (*i.e.*, from customer basket to vendor portfolio basket) as shown in Fig. 9. The existence of an outsourcing process in a vendor portfolio impacts vendor’s State variable (and thus Measures) as outsourcing process contributes the Revenue, the customer count and Realisation. It also impacts the track record and market influences over the time.

The decision to renew existing contract is specified on similar lines but with a different set of characteristic variables influencing the decision. Essentially the



(a) Competitive Analysis



(b) Distribution of Sunrise, Steady and Sunset at “Now”, After 5 “Years” and after 10 “Years”

Fig. 10. Simulation results when “WE” vendor continues as-is strategy. (Color figure online)

autonomous outsourcing process ExtendedActor raises Renew event after 3 to 5 “Year” timeframe. Here too, the evaluation is cognizance of incomplete and uncertain knowledge renewability criteria.

4.5 Simulation

We use ESL simulator to simulate the business-as-usual operations of the “WE” vendor and its competitors. The simulation progresses with simulation ticks where each tick represents a “Month”. The outcome of simulation runs depicting possible states of “WE” vendor and its competitors at “Now”, after 5 “Years” and after 10 “Years” is shown in Fig. 10(a). As can be seen, the initial revenue of “WE” (represented using shades of ‘blue’ ellipses) is 438.98 MUSD from 90 customers with a realization of nearly 15.5 USD per hour per FTE. Corresponding numbers for competitor 1 and competitor 2 respectively are <319.97, 78, 13.33> (depicted using shades of ‘violet’ ellipses) and <352.32, 79, 15.1> (depicted using shades of brown ellipses). In short, at present “WE” vendor is doing much better than competition.

The graph, also shows the goals of “WE” vendor that aim to deliver <750, 200, 17> after 5 “Year” and <1000, 290, 18> after 10 “Year” (depicted using green ellipses). As can be seen, by continuing to operate the same way the “WE” vendor will be delivering <587.58, 160, 13.5> after 5 “Years” and <857.51, 215, 14> after 10 “Year” (as directed by red line in Fig. 10(a)) thus missing both the targets by a considerable margin.

Table 3. Decision Table.

Lever	Revenue (MUSD)		Number of customers		Realisation	
	After 5 Years	After 10 Years	After 5 Years	After 10 Years	After 5 Years	After 10 Years
No lever	587.58	857.51	160	215	13.55	14
Improve existing resource	820.63	1165.80	195	287	15.2	15.4
Robotic Process Automation (RPA)	899.3	1309.87	201	301	15.3	15.7

More importantly, competitor 2 will be overtaking “WE” vendor after 5 “Years” and both the competitors will be significantly ahead of “WE” vendor after 10 “Years”.

Clearly, “WE” vendor cannot afford to continue with its current way of operation. A detailed analysis on portfolio of Sunrise, Steady and Sunset kinds of business processes, as shown in Fig. 10(b), indicates significant percentage of current revenue of “WE” vendor is from sunset kinds of outsourced processes (shown in red colour in Fig. 10(b)). Over time this market is going to shrink considerably as compare to the steady (depicted using yellow colour) as well as the sunrise (depicted using yellow green) business processes. Thus “WE” vendor needs to bring about a change in its characteristics so as to be able to win more bids in this demand situation.

4.6 Validation, Evaluation of Simulation Results and Recommendation

As part of model validation, we simulated the BPO specification by considering a known set of Vendors and Customers with fixed number of outsourced Processes. Essentially we initialised Vendors and Customers to known states, simulated the specification for 2 “Years” and correlated observed simulation results with existing operational data to ascertain the validity of the constructed models.

After ensuring the operation validity of BPO specification, we explored two Levers as described in Fig. 5 (b) and captured observed Measure values in the decision table as depicted in Table 3. Figure 11 and the decision table depicted in Table 3 show the comparative analysis of two Levers. With the Lever 1, the “WE” vendor is able to beat revenue target while failing to meet the number of customers and realization targets, whereas the ‘WE’ vendor is able to beat both revenue and number of customer targets while failing to meet the realization target narrowly with Lever 2. This clearly shows that the Lever 2 works well for “WE” vendor in the competitive environment described in this section.

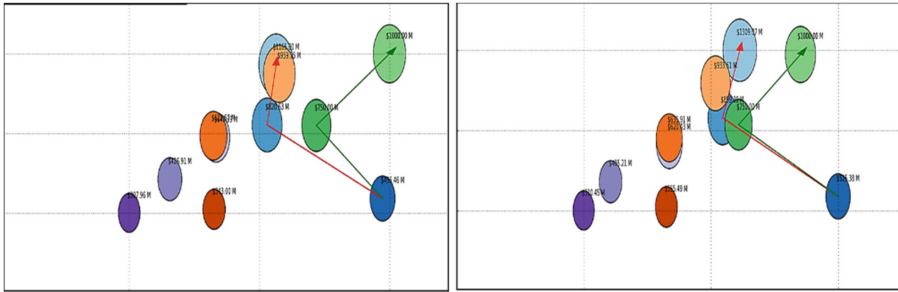


Fig. 11. Comparative study of Lever 1 and Lever 2.

Table 4. Evaluation summary.

Requirement	EM specification	Actor Lang.	Proposed approach	Enabling concepts in CMMModel
Why	✓	⊥	✓	Goal
What	✓	✓	✓	OrgUnit
How	✓	✓	✓	Event and behaviour
Who	✓	⊥	✓	OrgUnit
Where	✓	⊥	⊥	OrgUnit
When	✓	⊥	⊥	Time event
Modular	✓	✓	✓	OrgUni
Compositional	⊥	✓	✓	Composition relationship
Reactive	⊥	✓	✓	IncomingEvent, OutgoingEvent
Autonomous	×	✓	✓	InternalEvent
Intentional	✓	✓	✓	Goal
Adaptive	⊥	✓	✓	Adaptive behaviour
Uncertainty	×	⊥	✓	Stochastic behaviour
Temporal	⊥	×	✓	Temporal behaviour
Measure Spec	⊥	⊥	✓	Measure
Lever Spec	⊥	⊥	✓	Lever
Top-down/Bottom-up	Top-down	Bottom-up	Hybrid	Composition relationship, shared state variable

Legends: ✓ : Supports adequately, ⊥ can be specified with difficulties, × : not supported

5 Evaluation

For the kind of decision-making problem illustrated in this paper, industry practice relies extensively on spreadsheets, documents and diagrams. Such an approach typically represents the influence of Levers onto Measures in terms of static algebraic

equations. However, value of a Lever and influence of a Lever onto a set of Measures can vary over time. This behaviour cannot be captured using spreadsheets. Neither there is any support for encoding stochastic behaviour.

The proposed approach enables modelling of a system of systems using a set of hierarchically composable OrgUnits each listening/responding/raising events of interest. Each individual system or OrgUnit encapsulates state (*i.e.*, a set of *State* variables), trace (*i.e.*, events it has responded to and raised till now) and behaviour (*i.e.*, encoding of individual reactions). They interact with each other by sending messages resulting into emergent behaviour (*i.e.*, the behaviour of system of system emerges from interactions of OrgUnits or systems). The proposed approach further helps in addressing the scalability issue by reducing the numerous message passing between OrgUnits through shared variables. Therefore, we claim the proposed approach provides primitives for creating models that closely mimic reality.

An evaluation of two prominent decision-making aids, *i.e.*, EM based approach and pure actor language based approach, along with presented approach is summarised in Table 4. As shown in the table, an EM based approach and an actor language based approach are complementary in nature. The former one supports aspect (*i.e.*, *why*, *what*, *how*, *etc.*) specification and a top-down simulation approach, whereas actor language based approach is more effective for representing socio-technical characteristics and bottom-up simulation approach. But, it is not convenient for aspect specification. The proposed approach bridges the gaps between two classes of specifications by supporting comprehensive aspect specification and socio-technical characteristics as shown in Table 4. Moreover the explicit support for uncertainty, temporal behaviour, and the bottom-up and top-down combination make proposed approach suitable for CDDM.

6 Conclusion

Effective decision-making is a challenge that all modern organisations face. It requires deep understanding of aspects such as organisational goals, structure, operational processes. Large size, socio-technical characteristics, and increasing business dynamics make the decision-making a challenging task for the decision makers.

This paper argued that the efficacy of a complex dynamic decision-making (CDDM) chiefly depends on the three factors: (i) the availability of necessary and sufficient information in a machine-interpretable form, (ii) suitable machineries to process available information, and (iii) a method to capture information in a desired form and perform *what-if* analyses in a systematic manner. The paper presented an analysis of existing techniques and technologies to support a claim that the current state of the art decision making aids are inadequate for an affective CDDM and highlighted the gaps. Key aspects of this analysis point to the lacunae and inadequacy of support for representing necessary aspects of an organisation in a systematic manner, unavailability of appropriate concepts to represent the decision-making constructs, such as *Goal*, *Measure*, and *Lever*, and inability to handle inherent uncertainty. Importantly, the analysis also highlights the nonexistence of a suitable method supporting model construction, model validation and perform *what-if* analysis for effective CDDM.

To address these gaps, this paper contributed an approach that includes a meta-model to represent necessary and sufficient information in the form of a conceptual model (*i.e.*, CMMModel), a meta-model to represent information in a simulatable form (*i.e.*, ESLMModel) and a method. The meta-model CMMModel mitigates the identified specification gaps between the available technological capabilities and needs for CDDM (as highlighted in Table 1). The meta-model ESLMModel realises CMMModel while addressing the analyses needs of CDDM. These models are supported and used by the proposed method that uses a top-down approach for defining goals, measure and levers (the GM-L structure), a middle-out approach for defining structural aspect of an organisation, and a bottom-up approach for behavioural specification, addresses methodical needs. The method, principally, combines a modelling and validation method defined by Sargent [21] and a management sciences view for decision-making advocated by Daft [3]. The method is evaluated through an industry scale case study from the BPO domain.

As part of future research, we intend to validate the proposed approach using real business scenarios as well as proposing further extensions to CMMModel for introducing game theoretic approaches in simulations for CDDM. Other avenues of exploration include the use of constrained natural language to describe a problem entity so that a tool chain can be defined to automate production of the problem entity, conceptual model and the simulatable model. We expect the transformation chain to be human guided in the first instance.

References

1. Shapira, Z.: Organizational Decision Making. Cambridge University Press, Cambridge (2002)
2. McDermott, T., Rouse, W., Goodman, S., Loper, M.: Multi-level modeling of complex socio-technical systems. *Procedia Comput. Sci.* **16**, 1132–1141 (2013)
3. Daft, R.: *Organization Theory and Design*. Nelson Education, Toronto (2012)
4. Conrath, D.W.: Organizational decision making behavior under varying conditions of uncertainty. *Manag. Sci.* **13**(8), B-487 (1967)
5. O'Connor, T., Wong, H.Y.: Emergent properties (2002)
6. Locke, E.: *Handbook of Principles of Organizational Behavior: Indispensable Knowledge for Evidence-Based Management*. Wiley, Hoboken (2011)
7. Iacob, M., Jonkers, D.H., Lankhorst, M., Proper, E., Quartel, D.D.: *Archimate 2.0 Specification*, Van Haren Publishing, Zaltbommel (2012)
8. Bernus, P., Mertins, K., Schmidt, G.: *Handbook on architectures of information systems*, ISBN 3-540-64453-9 (2006)
9. Frank, U.: Multi-perspective enterprise modeling (memo) conceptual framework and modeling languages. In: HICSS. IEEE (2002)
10. Yu, E., Strohmaier, M., Deng, X.: Exploring intentional modeling and analysis for enterprise architecture. In: EDOCW (2006)
11. OMG Document, *Business Process Model and Notation* (2011). <http://www.omg.org/spec/BPMN/2.0/>. Accessed 03 Jan 2011
12. Meadows, D.H.: *Thinking in Systems: A Primer*. Chelsea Green Publishing, White River Junction (2008)

13. Barat, S., Kulkarni, V., Clark, T., Barn, B.: Enterprise modeling as a decision making aid: a systematic mapping study. In: Horkoff, J., Jeusfeld, M.A., Persson, A. (eds.) PoEM 2016. LNBIP, vol. 267, pp. 289–298. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48393-1_20
14. Sandkuhl, K., Fill, H.-G., Hoppenbrouwers, S., Krogstie, J., Leue, A., Matthes, F., Opdahl, A.L., Schwabe, G., Uludag, Ö., Winter, R.: Enterprise modelling for the masses – from elitist discipline to common practice. In: Horkoff, J., Jeusfeld, M.A., Persson, A. (eds.) PoEM 2016. LNBIP, vol. 267, pp. 225–240. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48393-1_16
15. Srinivasan, S., Mycroft, A.: Kilim: isolation-typed actors for Java. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 104–128. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70592-5_6
16. Haller, P., Odersky, M.: Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.* **410**(2), 202–220 (2009)
17. Allen, J.: *Effective Akka*. O'Reilly Media Inc., Sebastopol (2013)
18. Agha, G.A.: *Actors: A model of concurrent computation in distributed systems*. No. AI-TR-844. Massachusetts Inst of Tech Cambridge Artificial Intelligence Lab (1985)
19. Zachman, J., et al.: A framework for information systems architecture. *IBM Syst. J.* **26**(3), 276–292 (1987)
20. Rumsfeld, D.: *Known and Unknown: A Memoir*. Penguin, London (2011)
21. Sargent, R.G.: Verification and validation of simulation models. In: *Winter Simulation*, pp. 130–143, December 2005
22. Barat, S., Kulkarni, V., Clark, T., Barn, B.: A model based realisation of actor model to conceptualise an aid for complex dynamic decision-making. In: *MODELSWARD*, pp. 605–616 (2017)
23. Thomas, M., McGarry, F.: Top-down vs. bottom-up process improvement. *IEEE Softw.* **11** (4), 12–13 (1994)
24. Beckermann, A., Flohr, H., Kim, J. (eds.): *Emergence or Reduction?: Essays on the Prospects of Nonreductive Physicalism*. Walter de Gruyter, Berlin (1992)
25. Camus, B., Bourjot, C., Chevrier, V.: Combining DEVS with multi-agent concepts to design and simulate multi-models of complex systems. In: *Proceedings of the Symposium on Theory of Modeling & Simulation*, pp. 85–90 (2015)
26. Siebert, J., Ciarletta, L., Chevrier, V.: Agents and artefacts for multiple models co-evolution: building complex system simulation as a set of interacting models. In: *9th International Conference on Autonomous Agents and Multiagent Systems*, pp. 509–516 (2010)
27. Borshchev, A.: *The Big Book of Simulation Modeling: Multimethod Modeling with AnyLogic 6*. AnyLogic North America, Chicago (2013)
28. Macal, C.M., North, M.J.: Tutorial on agent-based modelling and simulation. *J. Simul.* **4**(3), 151–162 (2010)
29. Rolland, C., Selmin, N., Georges, G.: Enterprise knowledge development: the process view. *Inf. Manag.* **36**(3), 165–184 (1999)
30. Sandkuhl, K., et al.: *Enterprise Modeling. Tackling Business Challenges with the 4EM Method*, vol. 309. Springer, Heidelberg (2014). <https://doi.org/10.1007/978-3-662-43725-4>
31. van Langevelde, I., Philipsen, A., Treur, J.: Formal specification of compositional architectures. In: *10th European Conference on Artificial Intelligence* (1992)
32. Bock, A., Frank, U., Bergmann, A., Strecker, S.: Towards support for strategic decision processes using enterprise models: a critical reconstruction of strategy analysis tools. In: Horkoff, J., Jeusfeld, M.A., Persson, A. (eds.) PoEM 2016. LNBIP, vol. 267, pp. 41–56. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48393-1_4

33. Kinny, D., Georgeff, M., Rao, A.: A methodology and modelling technique for systems of BDI agents. In: Van de Velde, W., Perram, J.W. (eds.) MAAMAW 1996. LNCS, vol. 1038, pp. 56–71. Springer, Heidelberg (1996). <https://doi.org/10.1007/BFb0031846>
34. Hewitt, C.: Actor model of computation: scalable robust information systems. [arXiv:1008.1459](https://arxiv.org/abs/1008.1459)
35. Langley, A., et al.: Opening up decision making: The view from the black stool. *Organ. Sci.* **6**(3), 260–279 (1995)
36. Kulkarni, V., Barat, S., Clark, T., Barn, B.: Toward overcoming accidental complexity in organisational decision-making. In: *Model Driven Engineering Languages and Systems (MODELS)*, pp. 368–377 (2015)
37. Barat, S., Kulkarni, V., Clark, T., Barn, B.: A simulation-based aid for organisational decision-making. In: *ICSOFTEA 2016: 11th International Conference on Software Engineering and Applications* (2016)



Deterministic High-Level Executable Models Allowing Efficient Runtime Verification

Vladimir Estivill-Castro^(✉) and René Hexel

School of Information and Communication Technology,
Griffith University, Nathan, QLD 4111, Australia
{v.estivill-castro,r.hexel}@griffith.edu.au

<http://vladestivill-castro.net>, <http://www.ict.griffith.edu.au/~rhexel/>

Abstract. We present an architecture that enables run-time verification with executable models of behaviour. Our uniform modelling paradigm is logic-labelled finite-state machines (LLFSMs). Behaviours are constructed by parameterizable, loadable, and suspendable LLFSMs executed in predictable sequential schedules, but they are also verified at run-time by LLFSMs as well. Our architecture enables runtime verification (to monitor the quality of software during execution) as well as set up, tear down, and enforcement of quality behaviour during run-time. The LLFSMs models are executable and efficient because they are compiled (not interpreted). The LLFSMs can be derived from requirement engineering approaches such as behaviour trees, and also validated using test-driven development. However, in situations where software evolves incorporating elements of adaptive systems or machine learning, the software in execution may have never existed during development. We demonstrate the features of the architecture with illustrative case studies from robotics and embedded systems.

Keywords: Run-time verification

Logic-labeled finite state machines · Model-driven software development

1 Introduction

Software quality is critical to ensuring systems will not cause harm to humans (nor reduce quality of life), or economic loss [10]. The Internet-of-Things (IoT) demands more reliable software systems [40]. In early 2016, Australia Post completed its first trials of drone-delivered parcels, and although this technology will not be everyday practice for some time into the future, many embedded and robotic systems are already revolutionising transport and communication industries. Gartner estimates there are 8.4 billion smart devices in the IoT now across manufacturing, utilities, and transportation. However, insufficient software quality can cause severe malfunction of smart, embedded systems [53, 56, 58].

The opportunities for improvement in software quality are enormous: *“risks are becoming salient as our society comes to rely on autonomous or semi-autonomous computer systems to make high-stakes decisions”* [15]. The first and immediate category to deal with are AI software systems [15]: automated vehicles, home robots, and intelligent cloud services must perform correctly, even in the presence of surprising or confusing input. Recommendations emerge for researchers to focus on *“self-monitoring architectures in which a meta-level process continually observes the actions of the system, checks that its behaviour is consistent with the core intentions of the designer, and intervenes or alerts if problems are identified”* [15].

Experts suggest that the software models for the behaviour of the IoT and smart things are likely to be based on state machines [10]. These allow specifying behaviour at a higher level of abstraction than traditional programming languages, making software development faster. Use-case traces naturally map to paths through states and transitions. *Behavior Engineering* [17], a form of requirements engineering, creates these traces and then integrates them into Behavior Trees, from which finite-state machines, describing the behaviour of components, can readily be synthesised.

We will show how to use logic-labelled finite-state machines (LLFSMs) to model mechanisms that can monitor the software built using the model-driven development (MDSO) paradigm that LLFSMs offer. The precise semantics of LLFSMs makes them overcome some of the criticisms that MDSO has received [49] while enhancing its advantages. LLFSMs have been proven very effective for describing software behaviour [7] and for performing model-checking and formal verification, both in the value and the time domain [26]. LLFSMs offer a model of controlled concurrency that scales much better than comparable event-driven modelling approaches (such as UML-state charts, Behavior Trees, and teleo-reactive systems). Consequently, changing, improving, and maintaining behaviours of embedded systems and robots using LLFSMs is more cost-effective. Modelling at this high level means that the behaviour is closer to the original set of human-language requirements and therefore easier to understand. In the systems engineering and robotics communities, state-machines are ubiquitous. MDSO leads to more uniform quality; the LLFSM compiler produces efficient executables as it compiles to general, uniform code that minimises overhead. Because of the use of visual models of LLFSMs, the resulting behaviours are more transparent, and the gap between business analysts, requirement engineers, and software developers is reduced. Moreover, to scale to larger systems, LLFSMs have the capacity to incorporate Test-Driven-Development (TDD) methods and derive test suites from use-cases, incorporating such tests as LLFSMs themselves [28]. Such TDD can be managed by Continuous Integration Servers [27].

Runtime verification focusses on the design of formal languages for the specification of properties that must hold during runtime [18]. LLFSMs offer the architectural elements for runtime verification [23]. In this paper, we take matters one step further and will create software systems that can monitor the quality of other software systems as they execute, set-up, tear-down, and enforce

behaviour quality on the fly. We demonstrate the progress with concrete case studies: a network of traffic lights, a robotic vehicle and a legged robot.

We use the fact that LLFSMs are executable models analogous to state charts, but with transitions labelled by logic predicates. LLFSMs represent deterministic, executable models that enable formal specifications of requirements, including observable behaviour. We generate agents that can observe and monitor behaviour. This step enables agent technology capable of identifying undesired behaviour, raising warnings, and acting to prevent software malfunction. We use TDD and MDSO tools for the automatic construction of runtime monitoring agents that execute tests, monitor behaviour, and revise software models as they execute. Our monitoring LLFSMs raise the level by which the software is aware of its operational state, since the monitoring agents would be able to report on the behaviour of their underlying software components.

The rest of this paper is organised as follows. Section 2 discusses the three architectural elements that enable our approach. The first is the sequential scheduling of arrangements of LLFSMs that are not event-driven, but label transitions with Boolean expressions instead. The second is the capability to communicate between LLFSMs with a data-centric, in-memory middleware. The third element is the use of control/status messages, different from a publish/subscriber pattern and following a writers/readers pattern. Section 3 illustrates these architectural elements with a concrete example. This example will be used in Sect. 4 to describe our approach to runtime verification. While the first example [25] is a simple embedded system, Sect. 5 shows what can be achieved with robotic systems. Section 6 describes how to automate the generation of monitoring LLFSMs, and Sect. 7 discusses the safety and security implications by contrasting with ROSRV [35]. We benchmark our proposal here with the state of the art from literature in Sect. 8. In Sect. 9, we summarise and conclude the paper.

2 Architectural Elements

We base our architecture on executable behaviour models, represented by finite-state machines. Importantly, there are three crucial elements in this architecture.

First, transitions are labelled by Boolean expressions only (and not events), hence the name logic-labelled finite-state machine (LLFSM). LLFSMs are Communicating Extended Finite State Machines (CEFSMs) without events [43]. Significantly, the semantics is therefore not that of a software component waiting for an event triggering the transition to a new state. Instead, the components form a single thread of LLFSMs under a predefined schedule. The machine that executes (has the token) evaluates the sequence of transitions associated with its current state. This evaluation could potentially be quite sophisticated and complex (involving planning and/or reasoning), making LLFSMs, not plain, reactive architectures, but to also blend into deliberative systems [19, 25]. Control remains with one and only one component, resembling a deterministic polling system (unlike an interrupt handler). If an expression labelling a transition evaluates to `true`, the transition fires, making its target state the current state of

the LLFSM. As with ubiquitous models of state machines, states have `ONENTRY`, `ONEXIT`, and `INTERNAL` sections. Actions (code) in the `ONENTRY` section is executed only after a state change. The `ONEXIT` section is executed after a transition fires, while the `INTERNAL` section is executed only if all transitions have evaluated to *false*. After either, it becomes the next machine’s turn in the arrangement. LLFSMs have a series of mechanisms to handle composition, and to be suspended, resumed or restarted. In addition to interpreters for `Simple C`¹ and `Java`, we have efficient LLFSM compilers for `C/C++` and `Swift` under POSIX systems such as Linux or macOS, for microcontrollers, and ROS. LLFSMs are akin to UML state charts where transitions are labelled only by guards.

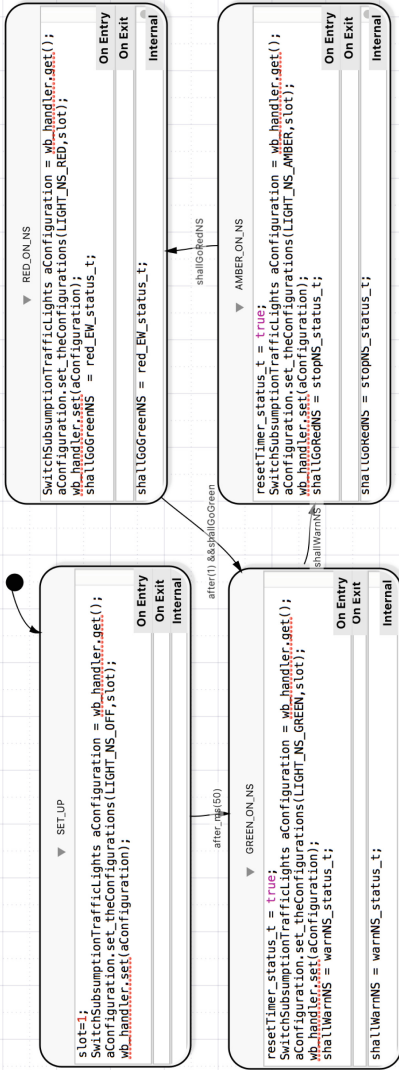
The second crucial element is the communication middleware between LLFSMs. Variables may be valid for the whole LLFSM, but states, and even each section (`ONENTRY`, `ONEXIT`, and `INTERNAL`), can have their own local variables, not shared with any other scope. However, beyond one LLFSM, variables reside in an object-oriented whiteboard, implemented in shared memory [24]. The whiteboard can be seen as a data-oriented broker, decoupling information readers and writers. However, as opposed to most robotic middlewares, where the paradigm follows a Push approach, we use a Pull approach [22]. With the LLFSM execution semantics, our `gusimplewhiteboard` implementation [24] offers fast, lock-free, atomic reader/writer semantics for multiple readers and even multiple writers. This OO implementation has proven superior in speed and reliability to other middlewares such as ROS’ system [24, 37].

The third aspect that provides a simpler, clearer semantics, while retaining modelling power and Turing-complete expressivity, is data-centric communication between components utilising *control* and *status* messages. Thus, the whiteboard implements a *blackboard control architecture* [32]. Control/status messages are an alternative to the scenarios akin to the *rendezvous* model [34, 50] in the message passing world, or a synchronous remote procedure call (RPC). By contrast, control/status messages follow the readers/writers paradigm as opposed to producer/consumer or publisher/subscriber. Typically a single class definition is assigned two message slots, `Control` for control data, and `Status` for responses (e.g., from a sensor). Reader components such as actuators and controllers use the Pull paradigm to query their corresponding messages. This decoupling enables components with long or unbounded run time, such as AI planning and reasoning, to be incorporated without interfering with a deterministic, low-latency control architecture provided by LLFSMs [25].

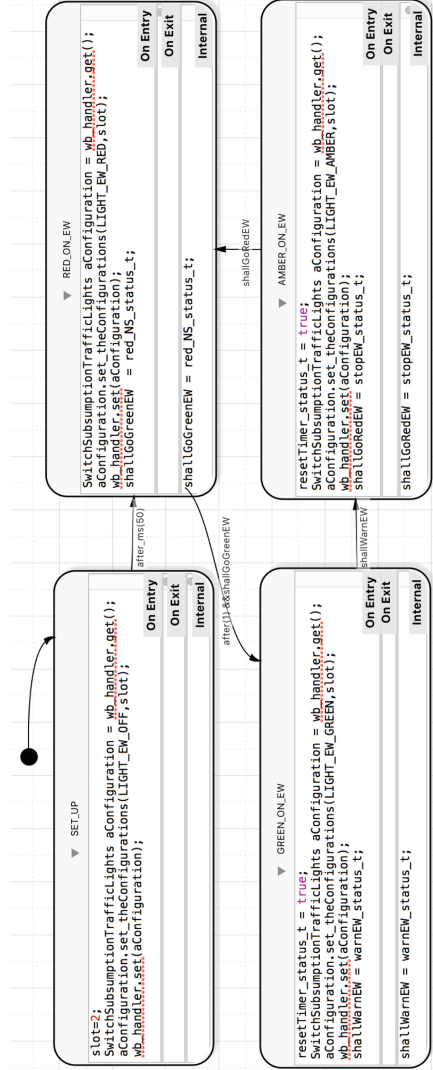
3 Illustration of Architectural Elements

We present the principles of our new software architecture with a classical example [44] of a system that controls traffic lights on an intersection between roads going North-South (NS) and East-West (EW). Requirements evolve from an initial version with no sensors, to a more advanced version with sensors in the EW-direction that, in the absence of a car, enable NS-priority (keeping the traffic

¹ `Simple C` is a subset of `C` used in some examples of `antlr` [48].



(a) North-South (NS) controlling LLFSM.



(b) East-West (EW) controlling LLFSM.

Fig. 1. Two LLFSMs for traffic lights at an intersection [23, Fig. 1]. Each machine represents a set of traffic lights for a particular direction (NS or EW). There is an initial SET_UP state, followed by three states representing the colour of the lights.

lights green in the NS-direction). The declarative requirements demonstrate the integration of reasoning and logic-programming for a reactive system [25]. The most crucial requirement, of course, is that the lights are never simultaneously green in both directions.

The complete system model consists of three LLFSMs in a single arrangement:² the Timer LLFSM and two controller LLFSMs. These two controller LLFSMs appear in Fig. 1 on the next page. These machines are part of our later example for run-time verification and are the two traffic-light behaviours for an intersection of roads going East-West and North-South. The first thing to notice is that analogous to OMT [51] and UML, these are executable models made of states and transitions. An arrangement of one or more LLFSMs constitutes a single, sequential program. That is, they have the semantics of a single thread. The token of execution rotates, in round-robin fashion, between the LLFSMs in the arrangement. States have three sections, and when the token of execution arrives to an LLFSM the corresponding machine resumes execution. It verifies it has not been suspended, and whether it has executed a transition from another state the last time it was its turn. If the current state differs from the previous one, the ONENTRY section will be executed, otherwise it is skipped. This is followed by evaluating, in sequence, the guard for each transition, and if one becomes `true`, the execution of the ONEXIT completes the turn for this LLFSM. If all guards are `false`, the turn completes by executing the INTERNAL section. Note that this sequential semantics is due to the fact that labels for the transitions are not sets of events but a sequence of Boolean expressions. Machines are compiled into loadable libraries of efficient, executable code.

Boolean expressions such as `after(1)` are analogous to the predicates that were used in *augmented* finite-state machines (AFSM) of the subsumption architecture. In fact, the LISP language for the subsumption architecture [9] is a subset of LLFSMs. Similarly, teleo-reactive programs [46] label all transitions with Boolean expressions. Consider the code in the ONENTRY section of the state RED_ON_NS.

```
SwitchSubsumptionTrafficLights aConfiguration = wb_handler.get();
aConfiguration.set_theConfigurations(LIGHT_NS_RED,slot);
wb_handler.set(aConfiguration);
shallGoGreenNS = red_EW.status.t;
```

and also the INTERNAL section

```
shallGoGreenNS = red_EW.status.t;
```

The statement in the INTERNAL section is also the last statement of the ONENTRY section and illustrates the use of a status message. The Boolean variable `shallGoGreenNS` is updated by retrieving a status message. The writer of this message is a compiled Prolog program that evaluates whether the conditions to move the North-South traffic light to green have been met [25]. Thus, if the current state is RED_ON_NS, the machine will evaluate the variable `shallGoGreenNS` and, if `false`, not carry out the transition. But before relinquishing the token of execution, the code in its INTERNAL section gets run, updating the transition-labelling variable with the current advice from the Prolog program. This shows that the models execute the reactive actions of moving to another state in their

² A GUI facade with avatars for effector and sensor hardware appears in the simulation at youtu.be/HFm6fbZ6lkg.

own time, analogous to a time-triggered approach (and definitely distinctive from the event-driven approach of UML state charts).

In our middleware, the data structures used to communicate between LLFSMs (and other processes or modules, such as Prolog programs) are essentially any C++ object with a standard C/C++ footprint in memory. This communication middleware is illustrated in this fragment of code. The statement

```
SwitchSubsumptionTrafficLights aConfiguration = wb_handler.get();
```

uses a previously declared *handler* to the middleware to retrieve the instance of `SwitchSubsumptionTrafficLights` into the object `aConfiguration`. The object-oriented nature of this middleware follows a data-centric whiteboard paradigm, and thus, all methods of the class `SwitchSubsumptionTrafficLights` are available. This is what happens with

```
aConfiguration.set_theConfigurations(LIGHT_NS_RED,slot);
```

Here, the corresponding slot for this LLFSM is updated in the data structure that the switch will use to relay commands for the traffic lights. The next statement below posts this updated data structure back to the whiteboard without any need for concurrency synchronisation as the current LLFSMs in the arrangement knows no other LLFSMs is accessing this object:

```
wb_handler.set(aConfiguration);
```

In summary, LLFSMs are models compiled into loadable executables, not interpreted. They have been compared to artefacts and modelling languages such as Behavior Trees [17], *Event-B* [1], Teleo-reactive programs [46], Executable UML [45], or SysML (UML tailored for systems engineering). For formal verification and requirements engineering, they compare favourably with Petri Nets [7] and Timed Automata [27,28]. Software construction with LLFSMs can emulate architectures based on embedded and reactive control as well as behaviour-based control, while adding feasible, formal verification [22]. In this paper, we take these elements further by enabling an architecture for runtime verification.

4 Verification and Reconfiguration

Each controller LLFSM (refer to Fig. 1) is in charge of a traffic direction and thus, minimally, each is in charge of a set of three lights (a read, green, and amber light each). Two versions of a declarative Prolog program (youtu.be/HFm6fbZ6lkg) specify when to switch lights. The Timer machine can be signalled to reset the time value. It regularly posts the time elapsed, and whether that time is greater than 5s, or greater than 30s.³ The LLFSM for the EW-set of lights (Fig. 1(b)) controls the green, amber, and red light in the EW-direction, cycling through three states such that only one light is on in each state. Thus, in the state `On_RED_EW`, in the EW-direction, only the red light is on. Symmetrically, the second controller LLFSM handles the NS-direction (Fig. 1(a)), signalling red,

³ Diagram for the Timer is 40s into the video (youtu.be/HFm6fbZ6lkg).

amber, and green in that direction, also cycling from green via amber to red, and back to green. All three LLFSMs are scheduled deterministically, and the decisions as to whether to switch state are inspections of Boolean variables. For example, `shallGoRedNS` is evaluated by obtaining the value from the whiteboard with the statement `shallGoRedNS=stopNS_status.t`; that is, the LLFSM acts as a reader in the Pull architecture of this status message, while the value is updated by a writer that periodically executes the Prolog program. The Prolog program is inside a wrapper LLFSM, synthesized from the interface provided by the Prolog program and running in another process [20, 25]. The LLFSMs define a complete and functional system composed of executable models.

Using the sensor and prioritising the NS-direction is the result of a simple, localised change, restricted to only the Prolog program. Changing between software versions requires swapping between Prolog programs. The LLFSMs can be subject to formal verification (using standard model-checking tools), as the corresponding Kripke structure can be derived directly from the model (and the number of Kripke states is small). In addition, since often expressing properties about system behaviour to perform formal verification can be complex, it is possible to create a suite of TDD tests by creating test-LLFSMs that set up, watch, and tear down behaviour [27, 28]. Such testing can validate the system before investing effort into formal verification, and also can raise the confidence of system correctness where state explosion makes formal verification impossible.

We focus on the situation where replacing one behaviour component or any of the four wrappers, at runtime, could result in a faulty system. That is, one should be able to swap between versions without faults manifesting themselves. Of course, one way is to formulate these details as a requirement and build the software accordingly. However, if the decision-making process is learnt while running (the Prolog program is composed by something akin to inductive logic programming), then no possible test could have been created originally, as the logic program would not have existed at the time. Once the logic program is available, formal verification may be infeasible (due to the complexity of the system), while testing does not prove the system is correct: it merely shows that no failures occur in a finite subset of cases. Moreover, if big data technologies and stream-data analytics were to build, online, sophisticated new rules and software to decide on the settings of the traffic lights, exhaustive testing would be infeasible. Thus, monitoring the system while in execution may actually be required, to correct the effects of traces that lead to failure, but were not discovered earlier.

4.1 Software Architecture for Run-Time Verification of LLFSMs

We propose a revolution of the subsumption architecture [8] to manage the run-time verification of a system composed of LLFSMs. Our proposal, following the subsumption architecture principles, constructs behaviour from conceptual layers of timed, finite-state machines. What we suggest here is a revolution, because we no longer assume lower layers are correct. The *timed* aspect means that we have Boolean primitives, `after(t)`, that only become true after t units of time. We, however, go beyond a mechanism to just *suppress* an input, and even beyond

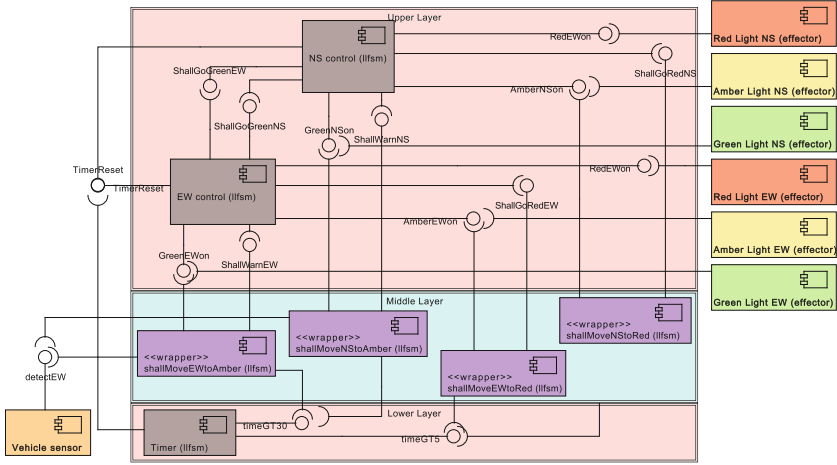


Fig. 2. Layered component diagram of the LLFSMs executable software that control the traffic lights [23, Fig. 2].

the capacity to *inhibit* the output from an LLFSM. Instead, we extend the mechanism to *suspend* [24] an LLFSM and add a mechanism that dynamically *loads* an LLFSM to join the arrangement for execution. Correspondingly, we provide mechanisms to also dynamically *unload* a faulty LLFSM and remove it from the schedule of execution.

The subsumption architecture always assumes that the lower layers are entirely correct. In stark contrast, we propose that the lower layers may, in fact, be faulty. In our proposal, higher levels act as behaviour monitors for lower layers. Realisation that a lower layer is malfunctioning, perhaps violating some requirement, is sufficient for the higher layer to take action, including one or several of the following actions.

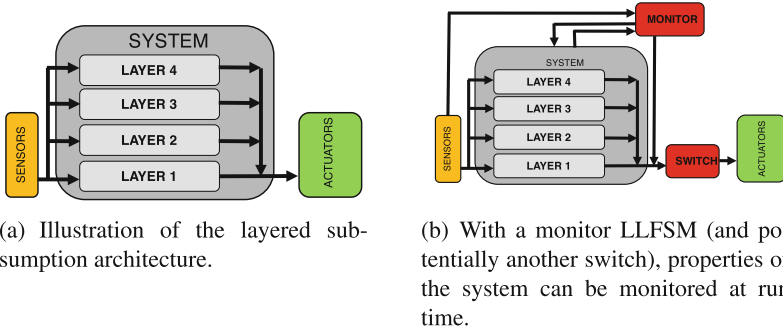


Fig. 3. Generic architecture of the safety monitor [23, Fig. 3]. (Color figure online)

1. Inhibit the output of the lower layers and replacing it with newer, safer output.
2. Provide input to lower-level machines to steer them, suspend them and/or restart them.
3. Reconfigure the arrangement by unloading some of its LLFSMs and loading non-faulty replacements.

That is, higher layers can rebuild lower layers that exhibit unstable behaviour. Figure 2 shows a component diagram (with the inputs and outputs for the traffic lights and sensor) in the layered style of the subsumption architecture [8]. The two LLFSMs of Fig. 1 appear in the upper layer in grey. Our approach is to take such a system (that receives input from sensors drawn on the left and delivers outputs to actuators) to an expanded and safer level, where a monitor (with a subsumption switch) ensures fundamental safety properties during runtime. This is illustrated by the transformation in Fig. 3, where Fig. 3(a) follows [8, p. 17, Fig. 3] to Fig. 3(b). Note that the original system can be abstracted and treated as a black box from the perspective of the two new components. The first component is a monitor LLFSM, while the second one is a subsumption `switch` [13] that can also be modelled/implemented as a (separate) logic-labelled finite-state machine. The added modules can treat the entire set of output signals of the system as inputs (“external” signals from their perspective). The added components (coloured boxes in Fig. 3(b)) are small and thus their formal verification becomes feasible. More importantly, the switch LLFSM is capable of inhibiting dangerous configuration of output signals to the actuators, replacing them with safer configurations. The monitor LLFSM can perform all the actions suggested earlier that reconfigure the running system.

Our extension creates a more uniform, layered architecture, whether or not the system is a subsumption architecture. The LLFSM for the switch⁴ simply buffers configurations of effector and actuator commands with a given priority.

If the system is a subsumption architecture, the switches already are part of the system and do not need to be replicated. The only requirement is that configurations provided by the monitors have a higher priority. Moreover, the monitoring LLFSM can have its own API, as we will discuss later.

The generality of the LLFSM approach facilitates that the monitor itself can be an LLFSM, and consequently, the monitor is also an executable model⁵. For the traffic light system, the monitoring LLFSM (Fig. 4) checks that the two green lights are never on simultaneously. The monitor will inhibit this by loading a new behaviour, which will trigger blinking amber lights in both directions (with all red and green lights turned off). Such behaviour signals malfunction to motorists and to the traffic authorities. When the monitor discovers a fault, it loads a machine that expresses a new behaviour (both lights blinking amber), and unloads the current faulty machine, loading default ones. This construction is the generic machine-monitoring pattern.

⁴ youtu.be/HFm6fbZ6lkg at 3 m 32 s.

⁵ From 3 m 40 s in the above video.

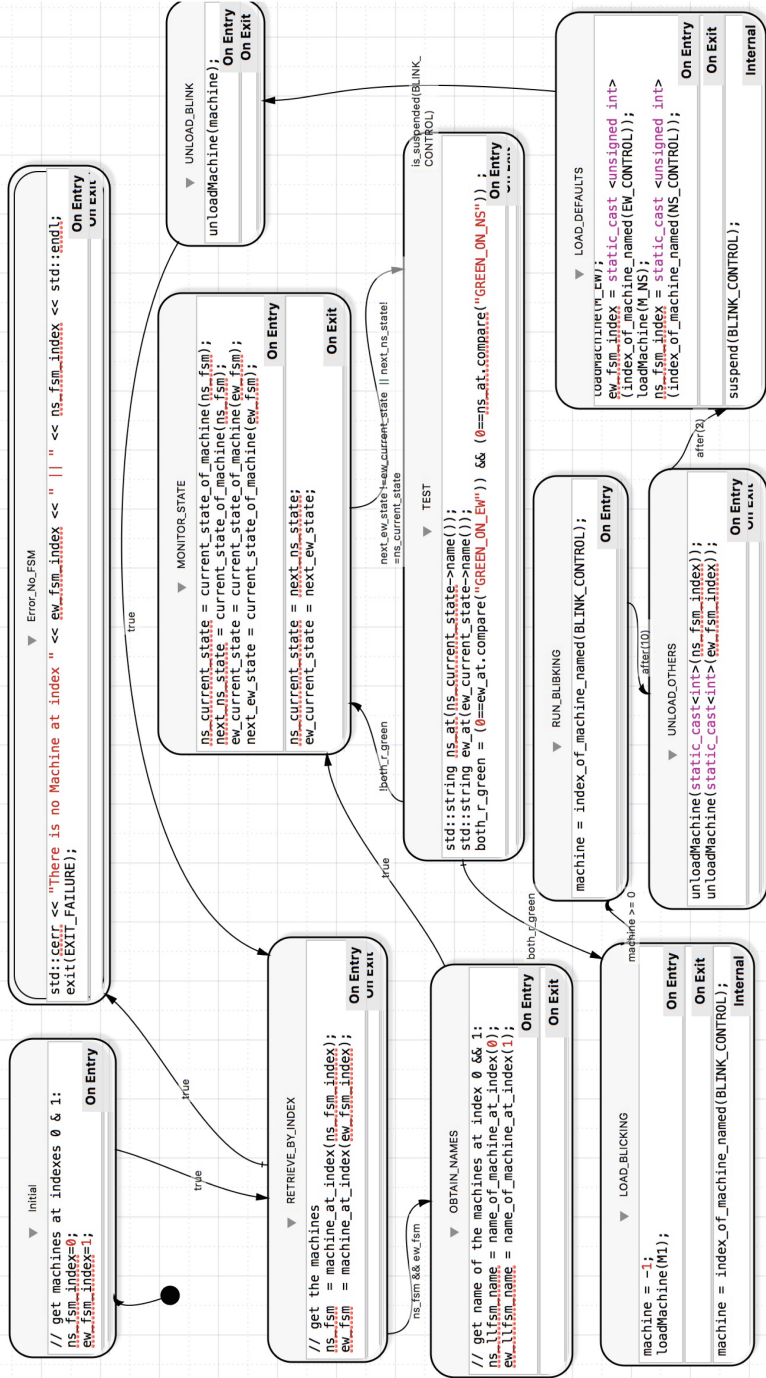


Fig. 4. The monitoring LLFSM verifying that both directions are not in the light-green state simultaneously. If this condition is detected, a blinking LLFSM is loaded.

5 Robotics Case Studies

5.1 Interaction of Behaviours

Our second case study is inspired by the presentation of the runtime verification framework `ROSRV` [35]. This framework aims at raising the level of safety in robotic systems under ROS and mainly consists of a node named `RVMaster`. It oversees all the peer-to-peer traffic in a ROS environment, blocking messages, and shielding the actual Master node (usually named `roscore`). The `ROSRV` architecture places a *Monitor* between every pair of publishers/subscribers, requiring a large number of monitors. The link between the `RVMaster` and the ROS Master is secured with a firewall.

This framework is illustrated using a simulator of the LandShark unmanned ground vehicle (UGV) robot. The examples represent situations where two modules responsible for two different tasks (although optimised for their individual responsibilities), when operating simultaneously, produce an overall deficient behaviour. One such example is a gun on the turret having a tracker for a target but when placed on the LandShark UGV body it may hit itself [35]. These scenarios are common in other robotic software, and another example discussed with the UGV simulator is combinations of turret positions and accelerations causing it to tip over [35]. Although there is no public access to the LandShark simulator, we can reproduce two of the monitoring examples using the ROS Gazebo simulation of a Komodo, a robot that is also an UGV on wheels with an articulated arm and gripper. The forbidden runtime conditions we monitor are actuator commands rather than conditions about forbidden states.

The first scenario is that certain wheel accelerations are not to be set while the arm is extended, as this causes the robot to tip. Second, certain navigation commands are not to be performed as they would take the robot into unsafe terrain. For this example the video youtu.be/MVlghB0JZ1g shows one behaviour for exploring a region that is faulty, becoming more prone to accelerate and run into barriers. However, with our runtime verification monitor, when the robot is close to the obstacles, two new behaviours are loaded, one to spin it back and one to guide it to its origin before the earlier behaviour is allowed to operate again. The methodology presented before applies here in a very similar way. We add a subsumption switch to the original system that wraps the motor commands. We add monitoring LLFSMs for the conditions. One simply uses location information directly to track the position of the robot and thus instructing the switch to inhibit motion commands to motors that would place the robot too close to the obstacles. For the other example, the monitor LLFSM (refer to Fig. 5) reads the arm position sensors, to calculate and track the centre of gravity relative to the base of the robot, adjusting a threshold value in the subsumption switch for the maximum allowed wheel acceleration.

5.2 Modular Robotics

Scalability of the Internet of Things (IoT) has also prompted modular robots [3], that is, a robot that can be composed of several physical parts. In such a system,

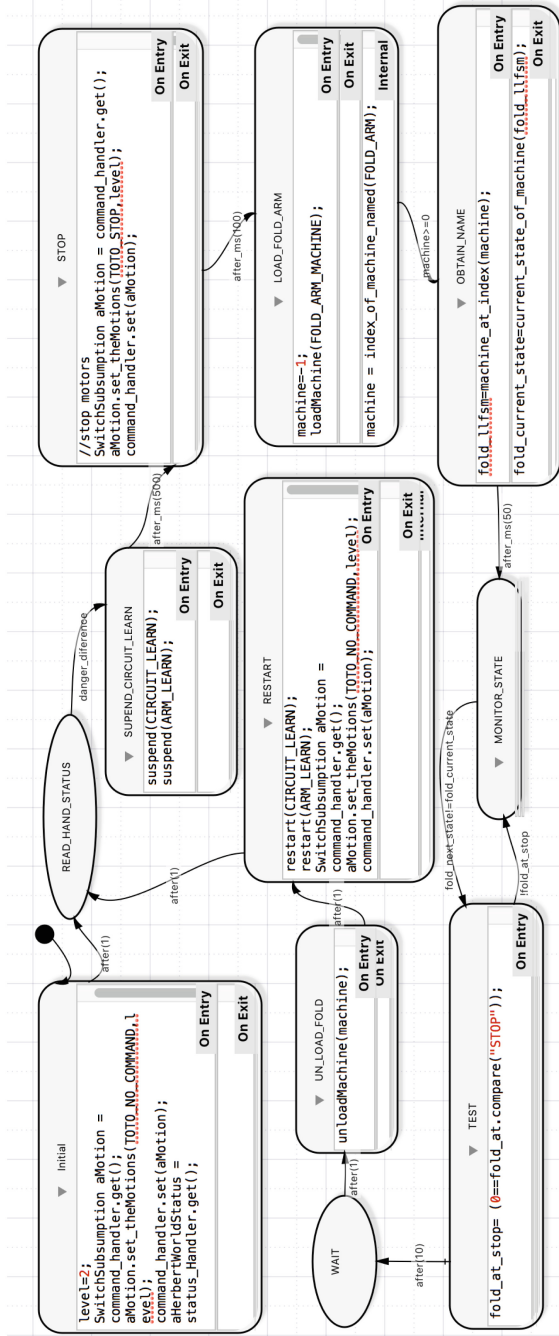
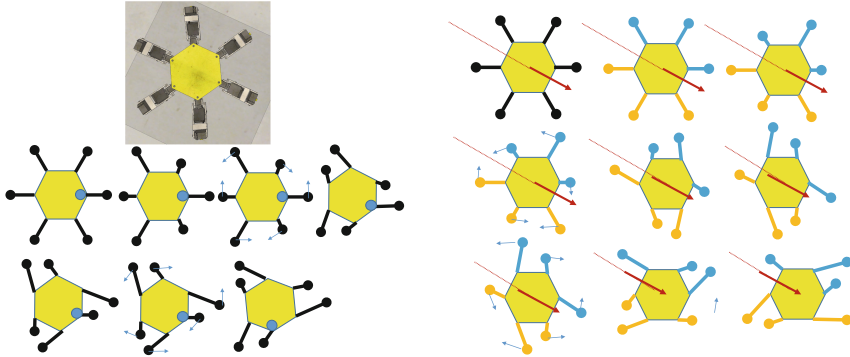


Fig. 5. Monitor LLFSM for the arm position that suspends traveling on a circuit and arm behaviour if arm’s position is dangerous. A behaviour that folds the arm replaces the exploratory behaviour of the arm.



(a) We use V-REP’s Hexapod robot to show the pattern for clockwise rotation. The longer legs are on the ground while raised legs are shown shorter.

(b) Sequence of the groups of legs on two sides of a direction of motion. Left side spins the robot clockwise while right side spins the robot counter-clockwise, but in phases between odd and even legs.

Fig. 6. Basic gaits for walking or spinning.



Fig. 7. Executable model for all legs of an n -legged robots as a parameterised LLFSM from which walking gaits and spinning gaits are composed.

the number of copies or parts of the same kind can be flexibly adjusted, not only prior to deployment, but even during operation. Therefore, it is natural to consider that the specified behaviour of such components should also be modular and would utilise MDSE [3]. We illustrate this with the parameterised behaviour that controls the repetitive and cyclic motion of an hexapod’s gait. Our presentation should be applicable to an arbitrary number of $n > 6$ legs placed around the centre of mass of the robot as if they were on a regular n -gon, it is easy to describe behaviour that spins the robot clockwise (Fig. 6(a)). In the first stage, the even numbered legs raise (shown as shorter lines). Then, odd legs use their body joint to push the robot clockwise as they actually do a counter-clockwise

turn of the body joint. Simultaneously, the even legs are raised and turn clockwise, advancing in the direction of the spin. The third phase lowers the even legs while raising the odd legs and roles are reversed between these groups of legs. In the fourth phase, it is now the legs on the ground (the even legs) that push the body by rotating counter-clockwise, while the raised ones (the odd ones) rotate clockwise. An equivalent gait for counter-clockwise rotation would simply reverse the direction of joint rotations.

The robot walks in a particular orientation using the same pattern! The fundamental movement of the legs uses the same four stage movement. However, as opposed to spinning, legs are now partitioned into two sides (Fig. 6(b)). Those on the left will be performing motions to spin clockwise, while those on the right of the center line of motion will spin counter-clockwise. The robot will walk because odd legs and even legs will have a phase shift of two stages. So the robot will ‘row’ in the direction of motion with even legs pushing back on the ground, while the odd legs are raised and move forward, again, with the odd group of legs replaced by the even in their role of pushing or advancing in the air. There are many more possible gaits. The point we are illustrating is that linear and rotational leg movements can be modelled as the fundamental parameterised motion of each leg. Figure 7 shows the fundamental four states of a leg, RAISE_LEG, LEVEL_LEG, SPIN_AGAINST_DIRECTION_OF_MOVEMENT, as well as PUSH_OPPOSITE_DIRECTION. However, deciding what is a push motion when the leg is down or what is rotating back the leg when the leg is up depends on three factors: (1) whether the hexapod is walking or spinning, (2) whether this particular leg is to the left or right of the direction of movement; and (3) if we are spinning, then whether the motion is a clockwise or counter-clockwise spin. Finally, the phase of a leg motion depends on whether it is an odd numbered leg or an even numbered leg. Figure 7 shows the parameterised executable LLFSM for the motion of a leg. The motion starts raising a leg or leveling a leg according to the group of the leg (even or odd). From there on, all legs loop through the same four states, and adjust the move when the leg is down or up according to the described calculation. A video of a hexapod driven around an area with spinning and walking can be seen at youtu.be/60FgjRvZqsc. The parameterised LLFSM in Fig. 7 are launched as concurrent, non-blocking calls with the corresponding parameters. That is, the behaviour that conforms to the gait in the case of the Hexapod invokes six instances of Fig. 7 with the appropriate actual parameters.

This example shows another feature of LLFSMs: the flexible non-blocking invocation of parameterised LLFSMs. To illustrate the run-time verification in this setting, we only show the most relevant states of the controller LLFSMs that enables driving around of the hexapod as illustrated in the video mentioned earlier. The setting of the parameters can be seen in the state RESTART_LEG_MACHINES. State NUMBER_LEGS assigns each invoked LLFSM a number (Fig. 8). State RESTART_LEG_MACHINES also calls each LLFSM without blocking. Thus, it needs to check that all such LLFSMs are running and then synchronise them, before reading a new action (and new direction) from the driver.

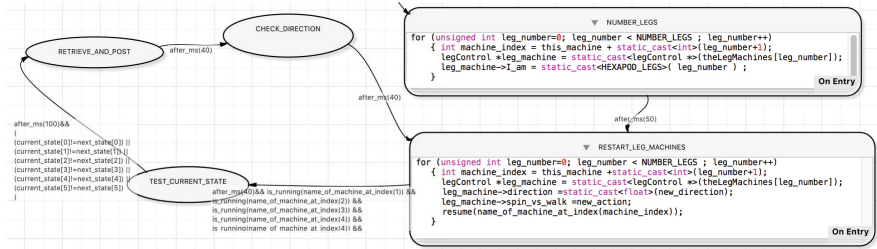


Fig. 8. Section of the LLFSM that numbers the legs. It enters a loop that checks what is the new action (and new direction) the driver of the hexapod wants to take. Machine’s parameters are set and all LLFSMs are launched concurrently without blocking this caller.

However, since each LLFSM for the legs is launched separately, there is a need to ensure synchronisation. For example, all even legs (and similarly, all odd legs) must be in the same state. This verification is rather different from static verification. In theory, one could write the corresponding temporal logic formula, but this would be particularly laborious. Nevertheless, we illustrate the virtue of LLFSMs by presenting an LLFSM that verifies this aspect at run-time. The new monitoring LLFSM will watch the state changes of the six instances of LLFSMs for the hexapod. Recall that all these are instances of the Fig. 7 LLFSM, with common parameters for the action (walking vs spinning), but with a different leg number. The monitoring LLFSM (different from the controlling LLFSM) is shown in Fig. 9. The important aspect to notice is that the transition from state `MACHINE_STATE_CHANGES` happens in any of the six LLFSMs has a state change (the transition is whether the first or the second, or the third odd labelled LLFSM controlling the leg has a change of state or the first or the second, or the third even labelled LLFSM has a change of state). But then, the transition from state `SOME_CHANGE_HAPPENED` to `ERROR` is taken if it is not the case that all machines had a change. This is also the virtue of LLFSMs’ sequential schedule, as all LLFSMs in the arrangement receive the token of execution before the monitoring LLFSM in Fig. 9 receives the execution token again. All LLFSMs are executing concurrently, and despite non-blocking calls that re-launched the leg controllers, synchronisation is achieved without further explicit coordination (as would happened with open concurrency that requires semaphores, monitors, or other explicit synchronisation mechanisms and which often renders formal verification impossible [21]). The monitoring LLFSM in Fig. 9 is not necessary to control the hexapod, but can be incorporated as a safety mechanism using the architecture described in Sect. 4.1.

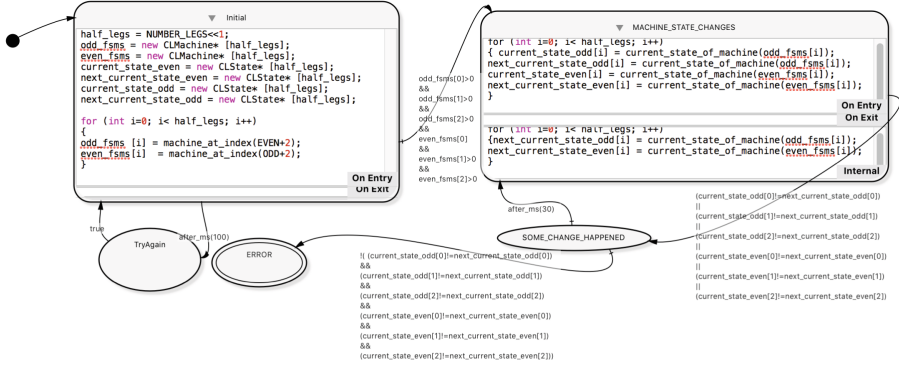


Fig. 9. A monitoring LLFSM that inspects the states of the instances of LLFSMs in Fig. 7 and checks the even group and the odd group of legs progress through their phases synchronised although resumed with no specific synchronisation.

6 Automatic Generation

Any runtime verification formalism [2, 57] could be embedded in a monitoring LLFSM because LLFSMs are Turing complete. However, we have chosen a simple mechanism that seems to fit most cases and, moreover, enables the construction of the monitoring LLFSMs from the visualisation of the system LLFSMs. The idea is to evolve LLFSMs constructed for TDD [27, 28] into monitors.

We explain our approach using the earlier example. The monitored conditions are rather simple. The LLFSM for TDD verifies that the controller LLFSMs are not simultaneously in designated states (e.g., turning all lights to green). This can also be achieved by monitoring the outputs of writer LLFSMs on the whiteboard. In the traffic light example, this would be the message to actuator lights for both green lights to be on.

Therefore, we suggest here that we can have a rather strong logic to express conditions to monitor the runtime validity of LLFSMs that are in the *System* box of Fig. 3. Moreover, the monitor LLFSM in Fig. 3 would be a model constructed completely from these logic expressions, significantly automating the implementation of such monitoring LLFSMs. First, we can describe the basic constructs of the logic to express forbidden conditions by monitoring LLFSMs. The first building blocks are formulas.

```

<formula> → <term> | (<formula> <connective> <formula>) | not(<formula>)
<term> → <state_formula> | <wb_variable_formula>
<state_formula> → <machine_name> @ <state_name>
<wb_variable_formula> → <value> == <wb_variable_name>
<connective> → ∧ | ∨
    
```

An example of the term that expresses that in the LLFSM arrangement of the traffic lights the two controlling machines cannot both be in their respective states where they set their respective lights to green is the following formula.

```
light_ns_subsumption @ GREEN_ON_NS
^
light_ew_subsumption @ GREEN_ON_EW
```

Similar formulas can be constructed for many of the safety requirements of the systems discussed in the literature of formal verification and software safety. For example, in the case of a microwave, a crucial requirement is the motor/radiation is not to be on while the door is open:

```
true == doorOpen ^ true == motorOn
```

The microwave is a widely discussed example in the literature of formal verification and model checking [4]. We point out here that from forbidden-condition formulas, the automatic construction of the LLFSM that monitors whether the formula evaluates to true (realises the forbidden condition) is rather simple. It consists of a simple loop where the information for the formula is retrieved from the whiteboard and then the formula is evaluated. Thus, our LLFSM generator only requires a parameter that indicates the period of the loop (using the `after()` construction mentioned before) and what LLFSM to activate in case the forbidden formula is realised. The designer of our runtime verification LLFSMs uses a GUI to choose states from LLFSMs to build `<state_formula>` and also to select whiteboard variables to build these formulas from. When whiteboard variables refer to objects, the GUI provides a drop-down menu to select `getters` to obtain an expression that evaluates to a basic type.

It should be clear that our logic for forbidden formulas is structurally and semantically equivalent to propositional logic. As we already mentioned, an LLFSM that checks such a formula is built by basically including the forbidden formula in a transition from a state that has read the necessary information. Such monitoring LLFSMs, although synthesised automatically are quite impenetrable to human designers. Most of the conditions or rules we have found in case studies on system safety seem to be of this form. However, we have noted that in some situations the forbidden scenario more closely corresponds to a trace of a behaviour. That is, the undesirable behaviour is not that, at a certain point in time, a certain configuration of variable values or states of sub-LLFSMs is reached in a system.

More elaborate, forbidden situations are sequences of formulas. For example, with the traffic lights, control in each direction cycles between green, amber, and red (then back to green). In this case, the forbidden behaviour can be specified by the complement of the regular expression `(green amber red)*`. Moreover, the equivalence of regular expressions and non-deterministic automata (and thus, deterministic automata) shows that we can construct monitoring LLFSMs automatically that verify that the system does not have a trace of basic formulas (about states and whiteboard variables) that belongs to a regular language where the alphabet are basic formulas. These monitoring LLFSMs are not expected to be drawn or presented for inspection by human designers, they can be rather large (even if we apply classical algorithms for DFA minimisation in the building of the corresponding monitoring LLFSM). However, the corresponding regular expressions are quite manageable by system designers. Today, for example, many

programming languages or (web) search facilities, offer tools to construct and visualise regular expressions. Thus for now, we consider this aspect less of a priority except that the architecture proposed here integrates the resulting monitoring LLFSMs quite naturally for expressing a language of forbidden traces in the running system under verification. Our `clfsm` tool enables the introspection of the running system to obtain the trace of the system’s state changes. This is another aspect in which the deterministic scheduling of arrangements of LLFSMs is an advantage, as the traces are not subject to pre-emptive scheduling if, for example, each LLFSM were to run as a separate thread.

The `spot` package allows the derivation of monitors (option `-M` for `ltl2tgba`); and we could use the `spot` libraries to automatically synthesise the monitor for our architecture directly as an LLFSM. In several robotic systems with planning and manipulation tasks, the *LTL* subset named *co-safe LTL* has been used [33] because it produces deterministic finite-automata [41]. Here again, Büchi automata can be directly modelled by LLFSMs. Our architecture can confirm co-safe *LTL* formulae, but if the formula has the modal operator for “eventually”, the monitoring LLFSM can not guarantee when such a condition is met (in the case of task planning it enables one to recognise a plan has found a goal meeting the co-safe *LTL* condition). However, we are studying a possible form of these logics or their variations for future bounded temporal logics. Note that *timed regular expressions* are equivalent to *timed automata* [4]. However, *timed automata* are non-deterministic in the sense that their execution/simulation on a computer is only one of the many execution paths. Thus, at the moment, these other formalisms to specify undesirable behaviours seem to demand a monitoring instrument that would be resource intensive.

7 Safety and Security Issues

Our architecture provides compile-time type safety because communication between LLFSMs (and from the subsumption switch to effectors and actuators) are OO-messages on the whiteboard. The only LLFSM that has access to these message types is the subsumption switch. All other LLFSMs only have access to the abstraction and interface the subsumption switch offers. Other LLFSMs cannot access effectors and actuators directly. The subsumption switch only forwards specific commands (to effectors and actuators) if such commands are placed in corresponding slots of the hierarchy by the respective LLFSMs of the system or the monitor. Our compile-time type safety is significantly more secure than `RVMaster` [35] because, for `RVMaster`, the underlying middleware is in itself ROS, lacking any security mechanisms [35]: ROS allows any node to read all the available topics and services at runtime.

In our proposal, we restrict which LLFSMs in an arrangement can perform operations such as *load*, *unload*, *suspend*, and *resume*. But monitoring LLFSMs have clearance for such operations on system LLFSMs. We are assuming that the software would need to exist in an environment isolated from penetration of malicious users who could plant such malicious LLFSMs in the paths read by the

`clfsm` instance executing the arrangement. The runtime verification here aims at safety by protecting from Byzantine faults of well-intentioned components that have evolved though potentially independent constraints and objectives, and whose synergies could cause malfunction in the system.

Evolving software modules (for learning a walk on a quadruped robot, or for tracking with a neck or turret with additional degrees of freedom), optimise their main task and thus they have a restricted range of messages for certain restricted families of effectors/actuators. We assume that system security is such that validated LLFSMs cannot be replaced with malicious ones. Moreover, the monitoring LLFSM is able to reset self-modifying modules by unloading the learnt/evolved, detrimental behaviour causing poor synergies with other modules and load a validated behaviour. In our traffic lights example, the video illustrates rebuilding at execution time the default behaviour and unloading the initial faulty behaviour. Another example is a robot learning to control its arm as it discovers the environment (see our video www.youtube.com/watch?v=_3VyISPQoEE).

The whiteboard middleware discussed earlier provides a channel to monitoring LLFSMs (monitors). Thus, monitors could receive the *suspend* command. This enables testing systems without monitoring (which could be resumed later) or running the system under different configurations of the properties that are being monitored. This facility to also configure monitoring systems during runtime has been used before [35], and in our proposal here is immediately available through the existing mechanisms of the whiteboard. Thus, it is possible to extend the subsumption architecture and the hierarchy of clearance classes by more than one level. Monitoring LLFSMs are also controllable. The suggested earlier transformation (from Fig. 3(a) to (b)) of adding a subsumption switch and a monitor (both LLFSMs) can be re-iterated several times as designers see fit, with higher levels being able to suspend, decommission, reload, and/ or reconfigure the components of the lower layers underneath.

8 Contrast with Related Work

Runtime Verification [30,38] focusses on how to monitor, analyse, and guide the execution of software, using lightweight formal methods applied during the execution of programs. Although formal validation of properties against running systems has been a long-standing concern in software engineering (for example instance dynamic typing), our suggestion here follows the current practices in testing (particularly model-based testing) when used before and during deployment of fault-tolerant systems. Note that the current practice for detecting and possibly reacting to observed behaviours satisfying or violating certain properties is to represent such properties with trace-predicate formalisms, such as finite state machines, regular expressions, context-free patterns, and linear temporal logics. LLFSMs are extremely suitable to describe verification properties and encompass all of the earlier mechanisms, as they are Turing complete [20].

Note that a large number of tools and approaches have been produced for runtime monitoring of sequential or concurrent programs in traditional coding languages such as C++, C, and Java [14]; however, essentially no work has

appeared for carrying out runtime verification using model-driven development tools. The reliability of time-triggered systems is significantly easier to determine than that of event-triggered systems [39, 42]. Time-triggered systems handle peak-load situations by design, enable software components to communicate using constant bandwidth and regular overhead even at peak load situations. By contrast, event-driven systems are inherently unpredictable, they can collapse during peak loads or event showers, and no analytical guarantees can be given for their performance [39, 42]. Surprisingly runtime-verification tools have been proposed using a modelling approach based on events [5, 12] and that their implementation is made in `Java` with unrealistic claims regarding real-time verification (but an admission of this issue is present [12, p. 141]).

Such *monitor-oriented programming* [11], in the environment of robotic systems, (in particular the Robotics Operating System ROS) requires ROSRV as an arbiter [35] of the appropriateness of message passing, introducing additional message relays and potential critical delays. Nevertheless, as discussed in the presentation, ROSRV is perhaps the closest approach related to our proposal here, but our architecture compares favourably. In ROSRV, security, scalability, and formal verification were identified as issues for further work [35]. With respect to security, ROSRV solely relies on network routing of trusted IP addresses. Moreover, ROSRV is centralised and policies and monitors need to be established for each publisher/subscriber pair, which does not scale well. The LLFSMs that act as the switch and the monitor can be formally verified in our architecture. We have also identified other advantages of our proposal, namely the specification of conditions to monitor can naturally and automatically be derived and expressed from the LLFSM models in model-driven development style.

We would argue that the subsumption architecture [8] and teleo-reactive systems are now classical mechanisms to produce reactive systems, that, in their inception, have been logic-labelled (and not event-driven), and in the case of the former, been significantly revolutionising the software architectures of robotic systems towards behaviour-based systems. In the case of the latter, several advances have been made to enable them with formal verification tools [16] or implementation tools [54]. However, teleo-reactive programs do have the danger of undefined behaviour [31].

Both, the subsumption architecture and teleo-reactive systems, suffer issues with their semantics of concurrency analogous to the issues of nested state-diagrams in UML. Issues such as state nesting [55] or other ambiguities [6, 55], have resulted in several problems with executable UML and its use in model-driven development. Most tools and approaches on formal methods based on UML must restrict themselves: for example, restrictions to the consistency and completeness of the artefact [47] or to *Practical Formal Specification's* (PFS) where events are precluded and component communications happen only through their declared inputs and outputs [36]. The community seems to largely follow Harel and Gery's executable model of hierarchical statecharts [29], which has an execution semantics akin to a *remote procedure call* (RPC) under the *Run-to-Completion Execution Model* (RTC) [52, p. 2.2.8]: that is, the system keeps

queueing events, while handling an earlier event. Such complicated semantics and runtime uncontrolled concurrency results in much higher complexity (or impossibility) of runtime verification.

9 Final Remarks

Software systems should be validated and verified prior to deployment. We are not suggesting here that because of our architecture validation, verification, and testing should be reduced. Nevertheless, current software systems evolve and adapt while in execution, and it is critical then to also ensure correctness at runtime. Artificial intelligence capabilities, such as machine learning, have matured and large software systems increasingly update their parameters, threshold values, or entire components on the fly. Software systems in operation generate large logs for big-data and analytics whose results can generate new versions to replace systems in operation. However, this logging requires a phase of batch learning, and off-line data analytics. If the adaptation, learning and analytics are incorporated with the software, the always learning system would be up-to-date with its latest experiences. However, potentially running software that none of its developers anticipated. Thus, the relevance of run-time verification.

With our approach, run-time verification excludes the system from some undesirable states, and enables to decommission LLFSMs in the arrangement. The temporary inconsistent behaviour is replaced by default safe behavior chosen by the monitoring LLFSMs. Such a replacement of one or more LLFSMs in a system could be significantly more organic, depending on particular external factors that have caused the system to evolve in particular ways, which cannot be entirely anticipated and verified. We hope that this research inspires the software engineering community to seek software systems with minimal downtime and continuous operation. Moreover, we expect this to be a fundamental quality aspect of robotics and complex, safety-critical real-time systems.

References

1. Abrial, J.R.: Modeling in Event-B – System and Software Engineering. Cambridge University Press, Cambridge (2010)
2. Alur, R., Henzinger, T.A.: Logics and models of real time: a survey. In: de Bakker, J.W., Huizing, C., de Roever, W.P., Rozenberg, G. (eds.) REX 1991. LNCS, vol. 600, pp. 74–106. Springer, Heidelberg (1992). <https://doi.org/10.1007/BFb0031988>
3. Arney, D., Fischmeister, S., Lee, I., Takashima, Y., Yim, M.: Model-based programming of modular robots. In: 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, pp. 66–74, May 2010
4. Asarin, E., Caspi, P., Maler, O.: Timed regular expressions. *J. ACM* **49**(2), 172–206 (2002)
5. Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D.: Quantified event automata: towards expressive and efficient runtime monitors. In: Gianakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 68–84. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32759-9_9

6. von der Beeck, M.: A comparison of statecharts variants. In: Langmaack, H., de Roever, W.-P., Vytopil, J. (eds.) FTRTFT 1994. LNCS, vol. 863, pp. 128–148. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58468-4_163
7. Billington, D., Estivill-Castro, V., Hexel, R., Rock, A.: Requirements engineering via non-monotonic logics and state diagrams. In: Maciaszek, L.A., Loucopoulos, P. (eds.) ENASE 2010. CCIS, vol. 230, pp. 121–135. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23391-3_9
8. Brooks, R.: A robust layered control system for a mobile robot. *IEEE J. Robot. Autom.* **2**(1), 14–23 (1986)
9. Brooks, R.: The behavior language; user’s guide. Technical report, AIM-1227, Massachusetts Institute of Technology - MIT, Artificial Intelligence Lab Publications, Department of Electronics and Computer Science (1990)
10. Bryce, R., Kuhn, R.: Software testing [guest editors’ introduction]. *Computer* **47**(2), 21–22 (2014)
11. Chen, F., Roşu, G.: Towards monitoring-oriented programming: a paradigm combining specification and implementation. *Electr. Notes Theor. Comput. Sci.* **89**(2), 108–127 (2003)
12. Colombo, C., Pace, G.J., Schneider, G.: Dynamic event-based runtime monitoring of real-time and contextual properties. In: Cofer, D., Fantechi, A. (eds.) FMICS 2008. LNCS, vol. 5596, pp. 135–149. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03240-0_13
13. Côté, C., Brosseau, Y., Létourneau, D., Raïevsky, C., Michaud, F.: Robotic software integration using MARIE. *Int. J. Adv. Rob. Syst.* **3**(1), 055–060 (2006)
14. Delgado, N., Gates, A.Q., Roach, S.: A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Trans. Softw. Eng.* **30**(12), 859–872 (2004)
15. Dietterich, T.G., Horvitz, E.J.: Rise of concerns about AI: reflections and directions. *Commun. ACM* **58**(10), 38–40 (2015)
16. Dongol, B., Hayes, I.H., Robinson, P.J.: Reasoning about goal-directed real-time teleo-reactive programs. *Formal Asp. Comput.* **26**(3), 563–589 (2014)
17. Dromey, R.G., Powell, D.: Early requirements defect detection. *TickIT J.* **4Q05**, 3–13 (2005)
18. Drusinsky, D.: Semantics and runtime monitoring of tlcharts: statechart automata with temporal logic conditioned transitions. *Electr. Notes Theor. Comput. Sci.* **113**, 3–21 (2005)
19. Estivill-Castro, V., Ferrer-Mesters, J.: Path-finding in dynamic environments with PDDL-planners. In: 16th International Conference on Advanced Robotics (ICAR), Montevideo, Uruguay, pp. 1–7 (2013)
20. Estivill-Castro, V., Hexel, R.: Arrangements of finite-state machines semantics, simulation, and model checking. In: Hammoudi, S., Ferreira Pires, L., Filipe, J., César das Neves, R. (eds.) International Conference on Model-Driven Engineering and Software Development MODELSWARD, Barcelona, Spain, 19–21 February 2013, pp. 182–189. SCITEPRESS Science and Technology Publications (2013)
21. Estivill-Castro, V., Hexel, R.: Module isolation for efficient model checking and its application to FMEA in model-driven engineering. In: ENASE 8th International Conference on Evaluation of Novel Approaches to Software Engineering, Angers Loire Valley, France, 4th–6th July 2013, pp. 218–225. INSTCC (2013)
22. Estivill-Castro, V., Hexel, R.: Simple, not simplistic – the middleware of behaviour models. In: ENASE 10 International Conference on Evaluation of Novel Approaches to Software Engineering, Barcelona, Spain. INSTCC, April 2015

23. Estivill-Castro, V., Hexel, R.: Deterministic executable models verified efficiently at runtime - an architecture for robotic and embedded systems. In: Ferreira Pires, L., Hammoudi, S., Selic, B. (eds.) Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2017, 19th–21st February 2017, pp. 29–40. SciTePress (2017)
24. Estivill-Castro, V., Hexel, R., Lusty, C.: High performance relaying of C++ objects across processes and logic-labeled finite-state machines. In: Brugali, D., Broenink, J.F., Kroeger, T., MacDonald, B.A. (eds.) SIMPAR 2014. LNCS (LNAI), vol. 8810, pp. 182–194. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11900-7_16
25. Estivill-Castro, V., Hexel, R., Ramírez Regalado, A.: Architecture for logic programming with arrangements of finite-state machines. In: Cheng, A.M.K. (ed.) First Workshop on Declarative Cyber-Physical Systems (DCPS) at Cyber-Physical Systems, pp. 1–8. IEEE, 12 April 2016
26. Estivill-Castro, V., Hexel, R., Rosenblueth, D.A.: Efficient modelling of embedded software systems and their formal verification. In: Leung, K.R., Muenchaisri, P. (eds.) The 19th Asia-Pacific Software Engineering Conference (APSEC), Hong Kong, pp. 428–433. IEEE Computer Society, December 2012
27. Estivill-Castro, V., Hexel, R., Stover, J.: Modeling, validation, and continuous integration of software behaviours for embedded systems. In: Al-Dabass, D., Romero, G., Orsoni, A., Pantelous, A. (eds.) 9th IEEE European Modelling Symposium, Madrid, Spain, 6th–8th October 2015, pp. 89–95 (2015)
28. Estivill-Castro, V., Hexel, R., Stover, J.: Models testing models in continuous integration of model-driven development. In: Cheng, A.M.K. (ed.) Proceedings of the IASTED International Symposium Software Engineering and Applications (SEA 2015), Marina del Rey, USA, 26th–27th October 2015. <https://doi.org/10.2316/P.2015.829-016>
29. Harel, D., Gery, E.: Executable object modeling with statecharts. In: Proceedings of the 18th International Conference on Software Engineering, ICSE 1996, Washington, DC, USA, pp. 246–257. IEEE Computer Society (1996)
30. Havelund, K.: Using runtime analysis to guide model checking of Java programs. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN 2000. LNCS, vol. 1885, pp. 245–264. Springer, Heidelberg (2000). https://doi.org/10.1007/10722468_15
31. Hayes, I.J.: Towards reasoning about teleo-reactive programs for robust real-time systems. In: Guelfi, N., et al. (eds.) SERENE 2008, RISE/EFTS Joint International Workshop on Software Engineering for REsilient SystEms, Newcastle Upon Tyne, UK, 17–19 November 2008, pp. 87–94. ACM (2008)
32. Hayes-Roth, B.: A blackboard architecture for control. In: Bond, A.H., Gasser, L. (eds.) Distributed Artificial Intelligence, pp. 505–540. Morgan Kaufmann Publishers Inc., San Francisco (1988)
33. He, K., Lahijanjan, M., Kavraki, L.E., Vardi, M.Y.: Towards manipulation planning with temporal logic specifications. In: 2015 IEEE International Conference on Robotics and Automation (ICRA), pp. 346–352, May 2015
34. Hoare, C.A.R.: Communicating sequential processes. *Commun. ACM* **21**(8), 666–677 (1978)
35. Huang, J., Erdogan, C., Zhang, Y., Moore, B., Luo, Q., Sundaresan, A., Rosu, G.: ROSRV: runtime verification for robots. In: Bonakdarpour, B., Smolka, S.A. (eds.) RV 2014. LNCS, vol. 8734, pp. 247–254. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11164-3_20
36. Iwu, F., Galloway, A., McDermid, J., Toyn, I.: Integrating safety and formal analyses using UML and PFS. *Reliab. Eng. Syst. Saf.* **92**, 156–170 (2007)

37. Joukoff, D., Estivill-Castro, V., Hexel, R., Lusty, C.: Fast **MAV** control by control/status OO-messages on shared-memory middleware. In: Kim, J.-H., Karray, F., Jo, J., Sincak, P., Myung, H. (eds.) Robot Intelligence Technology and Applications 4. AISC, vol. 447, pp. 195–211. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-31293-4_16
38. Kim, M., Viswanathan, M., Ben-Abdallah, H., Kannan, S., Lee, I., Sokolsky, O.: Formally specified monitoring of temporal properties. In: Proceedings of the 11th Euromicro Conference on Real-Time Systems, pp. 114–122 (1999)
39. Kopetz, H.: Should responsive systems be event-triggered or time-triggered? IEICE Trans. Inf. Syst. **76**(11), 1325 (1993)
40. Kopetz, H.: Real-Time Systems - Design Principles for Distributed Embedded Applications. Real-Time Systems Series, 2nd edn. Springer, New York (2011). <https://doi.org/10.1007/978-1-4419-8237-7>
41. Kupferman, O., Vardi, Y.M.: Model checking of safety properties. Formal Methods Syst. Des. **19**(3), 291–314 (2001)
42. Lamport, L.: Using time instead of timeout for fault-tolerant distributed systems. ACM Trans. Progr. Lang. Syst. **6**, 254–280 (1984)
43. Li, J.J., Wong, W.E.: Automatic test generation from communicating extended finite state machine (CEFSM)-based models. In: Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, (ISORC 2002), pp. 181–185 (2002)
44. Maier, D., Warren, D.S.: Computing with Logic: Logic Programming with Prolog. Benjamin-Cummings Publishing Co. Inc., Redwood City (1988)
45. Mellor, S.J., Balcer, M.: Executable UML: A Foundation for Model-Driven Architecture. Addison-Wesley Publishing Co., Reading (2002)
46. Nilsson, N.J.: Teleo-reactive programs and the triple-tower architecture. Electron. Trans. Artif. Intell. **5**(B), 99–110 (2001)
47. Pap, Z., Majzik, I., Pataricza, A., Szegi, A.: Methods of checking general safety criteria in UML statechart specifications. Reliab. Eng. Syst. Saf. **87**(1), 89–107 (2005)
48. Parr, T.: The Definitive ANTLR 4 Reference. Pragmatic Bookshelf, 2nd edn (2013)
49. Picek, R., Strahonja, V.: Model driven development-future or failure of software development. In: IIS, vol. 7, pp. 407–413 (2007)
50. Pnueli, A., de Roever, W.P., et al.: Rendezvous with ADA - a proof theoretical view. Vakgroep informatica RUU-CS-82-12, July 1982
51. Rumbaugh, J., Blaha, M.R., Lorensen, W., Eddy, F., Premerlani, W.: Object-Oriented Modelling and Design. Prentice-Hall Inc., Englewood Cliffs (1991)
52. Samek, M.: Practical UML Statecharts in C/C++: Event-Driven Programming for Embedded Systems, 2nd edn. Newnes, Newton (2008)
53. Sametinger, J., Rozenblit, J., Lysecky, R., Ott, P.: Security challenges for medical devices. Commun. ACM **58**(4), 74–82 (2015)
54. Sánchez, P., Alonso, D., Morales, J.M., Navarro, P.J.: From teleo-reactive specifications to architectural components: a model-driven approach. J. Syst. Softw. **85**(11), 2504–2518 (2012)
55. Simons, A.: On the compositional properties of UML statechart diagrams. In: Rigorous Object-Oriented Methods 2000. Electronic Workshops in Computing (eWiC), York, UK, January 2000
56. Srivastava, A.N., Schumann, J.: Software health management: a necessity for safety critical systems. Innov. Syst. Softw. Eng. **9**(4), 219–233 (2013)

57. Thati, P., Roşu, G.: Monitoring algorithms for metric temporal logic specifications. In: Fourth Workshop on Runtime Verification (RV 2004), vol. 113, pp. 145–162 (2005)
58. Weiss, M., Eidson, J., Barry, C., Broman, D., Goldin, L., Iannucci, B., Lee, E.A., Stanton, K.: Time-aware applications, computers, and communication systems (TAACCS). Technical report, Technical Note 1867, The National Institute of Standards and Technology (NIST), U.S. Department of Commerce, February 2015



A Consistency-Preserving Editing Model for Dynamic Filtered Engineering of Model-Driven Product Lines

Felix Schwägerl^(✉) and Bernhard Westfechtel

Applied Computer Science I, University of Bayreuth, 95440 Bayreuth, Germany
{felix.schwaegerl,bernhard.westfechtel}@uni-bayreuth.de

Abstract. The high cognitive complexity of model-driven software product line engineering is due to the fact that developers have to manually create, edit, and maintain multi-variant artifacts. As a solution, the adaptation of filtered editing has been proposed recently. Filtered editing can be applied in a static or in a dynamic way; in the latter case, new co-evolution problems occur when considering the evolving relationships between the historical, the variant, and the product dimension. This paper investigates, formally defines, and demonstrates by examples nine consistency constraints connected to dynamic filtered editing. Furthermore, we suggest a consistency-preserving editing model comprising four operations that synchronize a transparent multi-version repository with a single-version workspace view being presented to the user: check-out, modify, commit, and a novel operation, migrate, which prepares the workspace for the subsequent edit session. Several advantages of dynamic over static filtered or unfiltered editing are confirmed both on a theoretical and on an experimental basis.

Keywords: Model-driven software engineering
Software product line engineering · Filtered editing · Co-evolution
Uniform versioning · Variation control

1 Introduction

In *model-driven software engineering* (MDSE), software systems are developed from high level abstractions called models, which are eventually executed by interpretation or transformed into executable code [1]. A model is an instance of a metamodel, which defines the abstract syntax of the modeling language.

Software configuration management (SCM) addresses the evolution of software systems; *version control* (VC) lies at its heart [2]. Evolution may occur along several dimensions, giving rise to a refinement of the term *version* into *revisions* and *variants*, respectively. State-of-the-art VC systems organize revisions in *revision graphs* and variants in parallel *branches*.

Finally, *software product line engineering* (SPLE) is a paradigm to develop software applications based on the principle of *variability* [3]. A *platform* is

a common set of artifacts from which customized products can be efficiently derived. A variability model, e.g., *feature model* [4], describes common and discriminating features of the variants of a product line. The combination *model-driven product line engineering* (MDPLE) promises synergy effects [5], but it demands for creating, editing, and maintaining *multi-variant domain models*, which are cognitively complex tasks per se, but tend to become even more difficult as soon as evolution and collaboration occur.

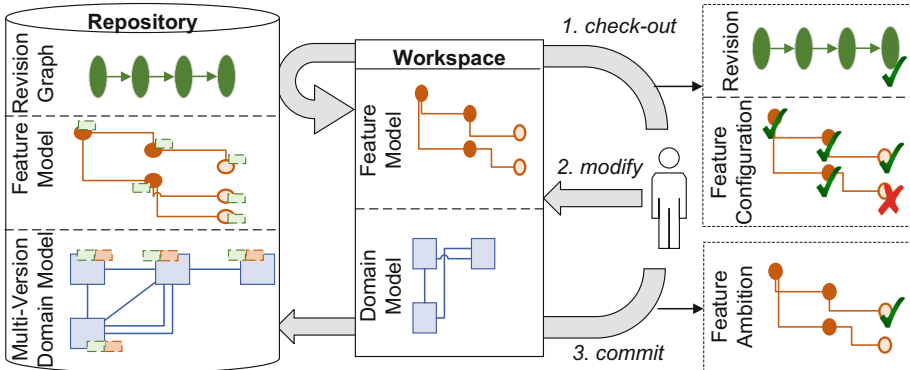


Fig. 1. Architectural and functional sketch of the conceptual framework. From [6, Fig. 1].

To address this, several approaches to *filtered software product line engineering* have recently emerged [7–9]. In accordance with the early ideas of *multi-version text editors* [10], variants not relevant for a specific change are hidden from the workspace. The multi-variant artifacts of the product line are transparent to the developer. Instead, he/she modifies a single-variant view, which is described using a *choice*, i.e., a *read filter*. The modifications are transferred back to the product line using an *ambition*, a *write filter* that defines the set of variants for which the changes apply. In this way, the change is applied *representatively* in one version – the choice –, but affects multiple versions—those included in the ambition.

The paper at hand is an extended version of [6]. The contributions presented here are based on a conceptual framework [11] whose architecture is sketched in Fig. 1. Tying on VC metaphors, a development iteration is started with *check-out* and finished with *commit*. In addition to a *revision graph*, which controls the historical evolution of the product line, a *feature model* is used for logical versioning. Selecting a version during *check-out* involves the selection of a revision and thereafter the definition of a *feature configuration*, which altogether form the choice. During *commit*, a new revision is created transparently. The logical component of the write filter is provided as a *feature ambition*, a partial selection in the feature model. The editing model is assumed to be *dynamic* in the sense

that it supports co-evolution of feature and domain model and allows to set or change the ambition throughout the edit session.

In this paper, we show that dynamic filtered editing requires well-defined *workspace operations* complying to a set of *consistency constraints*. As main contributions, we formally define these constraints and present *algorithms* that provenly preserve them. In addition to CHECKOUT and COMMIT, an extra operation is provided to MIGRATE the old feature configuration choice such that it is consistent with the evolved feature model and therefore obviates repeated check-outs.

Sect. 2 motivates the addressed co-evolution problems by a running example. Subsequently, in Sect. 3, formal foundations are explained. Sects. 4 and 5 formally present nine workspace consistency constraints and three algorithms for consistency-preserving workspace operations, respectively. Next, we supply proof that the revision graph is managed automatically and consistently. Sect. 7 presents a generalized editing model, whose amount of dynamism can be adjusted. Sect. 8 evaluates the dynamic filtered editing model experimentally. Sect. 9 presents related work. Sect. 10 concludes the paper.

2 Example Scenario

In certain situations caused by co-evolution of feature model, domain model, choice, and ambition, the dynamism implied by the dynamic filtered editing (DFE) model becomes problematic. Using the well-known graph library product line example [12], we informally sketch instances of consistency violations, which should be avoided by the consistency-preserving editing model. Furthermore, we sketch how the designated end user repairs the reported inconsistencies. For simplicity, the revision graph is faded out.

Incomplete or Inconsistent Choice. From the feature model depicted in Fig. 2(a), a version is to be selected for CHECKOUT. Then, the feature configuration shown in (b) is *incomplete*, since features *Vertices* and *Colored* do not have a selection state assigned. Therefore, the choice does not describe a unique product version. Moreover, (c) represents an *inconsistent* choice: The mandatory feature *Vertices* is deselected.

We contribute a consistency-preserving CHECKOUT operation, which ensures that user-selected choices are *complete* and *consistent*. This is also true for the feature configuration shown in (d), thus it is assumed for the subsequent steps.

Disallowed Feature Model Modification. During *modify*, the feature model may be edited, however, not arbitrarily. For example, in (e), features *Weighted* and *Directed* are made mandatory and, at the same time, arranged in an XOR group. This contradicts the semantics of feature models. A different problem is illustrated in (f): Feature *Weighted*, currently selected in the active choice (d), is deleted. In the workspace, however, elements connected to this features are still present. As a consequence, the workspace contains elements which could not be selected by any future choice.

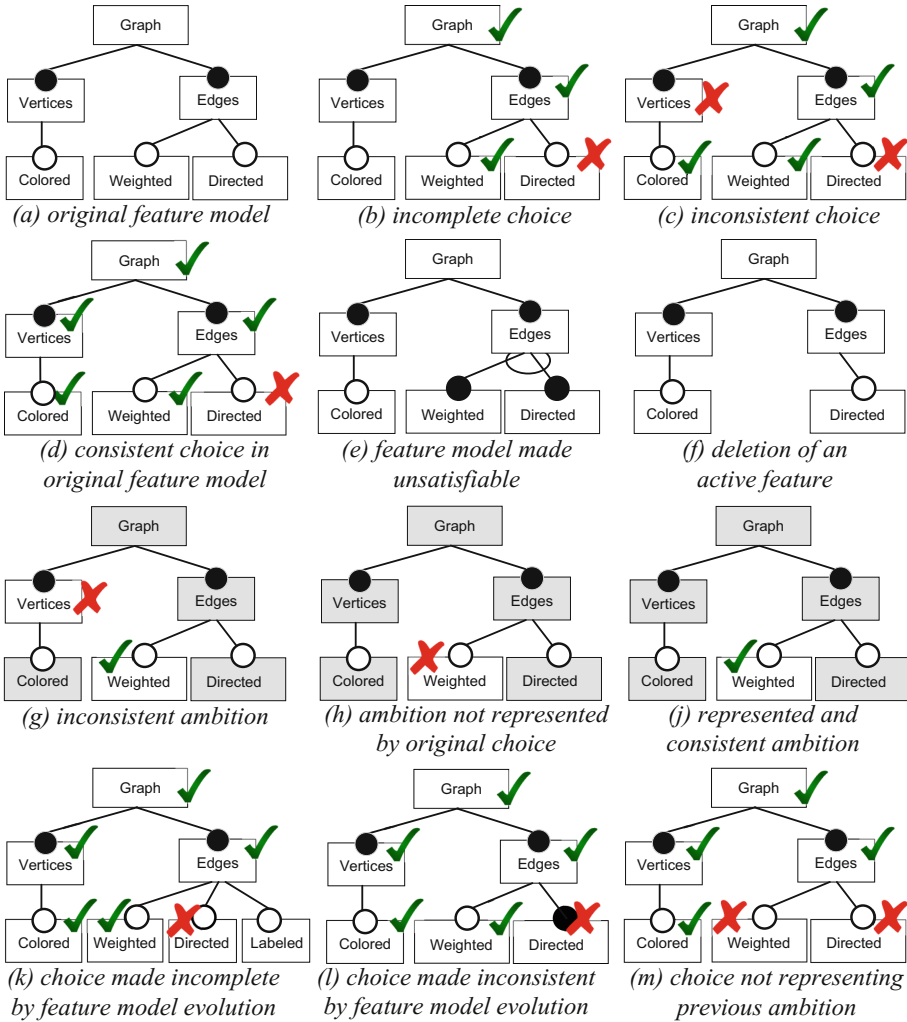


Fig. 2. Examples of consistency violations connected to feature models, choices, and ambitions.

The consistency-preserving MODIFY operation prevents both kinds of disallowed feature model modifications sketched.

Non-representative or Inconsistent Ambition. Moving further on, during COMMIT, the user is expected to define a feature ambition that delineates the scope of the change, i.e., the set of versions to which the performed change is relevant.

In Fig. 2(g), a user-specified feature ambition is depicted. Since the mandatory feature *Vertices* is bound negatively there, the ambition represents an *inconsistent* set of versions. Similarly, the ambition depicted in (h) is not in

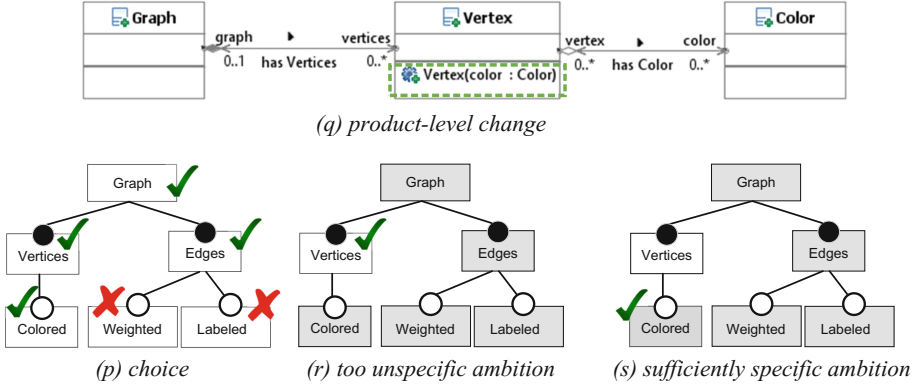


Fig. 3. Example of a non-representative product-level change, described with a too unspecific ambition. In (q), artifacts belonging to **Edges** and sub-features thereof are hidden.

line with the proposition that the choice must be a *representative* of it: Feature **Weighted**, which is positively selected in the choice (d), has a negative selection state assigned in ambition (g).

The contributed algorithm for COMMIT ensures that the ambition is *weakly consistent* (i.e., that it includes a consistent choice) and *represented by* the choice. Below, we assume that the valid ambition depicted in (j) has been selected.

Choice not Suitable for Next Iteration. Unless the user interrupts this workflow, the DFE model continues with the next iteration reusing the current choice. The choice may, however, become invalid for several reasons. First, (k) assumes that the original feature model is extended by a new feature **Labeled**, for which the original choice, however, does not define a binding. Similarly, in (l), feature **Directed** is made mandatory, but excluded from the current choice, such that this becomes invalid for the next iteration. Last, in (m), a new user-defined choice is depicted. This choice, however, disagrees with the ambition in the binding for **Weighted**, whose corresponding product artifacts are still present in the workspace. Thus, it becomes necessary to re-generate the workspace contents by check-out.

In the here presented DFE model, the user is assisted in preparing the current choice for the subsequent iteration by a new operation **MIGRATE**, which ensures that the choice becomes *complete*, *strongly consistent*, and that it is *included in* the ambition used for the preceding COMMIT.

All the problems discussed above are related exclusively to version concepts, namely version space (i.e., feature model), choice, and ambition. There exists an additional potential source of inconsistency that is, in contrast, also connected to the product space:

Too Unspecific Ambition. Figure 3 depicts an iteration based on a choice that represents a colored graph (p). The product-level-change shown in (q) consists in the introduction of a new constructor to class `Vertex`. As constructor parameter type, the existing class `Color`, whose visibility is restricted to those versions in which feature `Colored` is selected, is defined. An attempt to use (r) as ambition for this change should fail, for the following reason: The choice should be representative for all versions in which the change can be applied. Albeit, the constructor would not be valid in versions that exclude feature `Colored`, since class `Color` is not available as parameter type then. A more specific ambition, which adequately describes the set of versions in which the change is applicable, should be used; the most general, yet sufficiently specific ambition is (s).

3 Formal Foundations

Following [11, 13], we provide a formalization of the underlying conceptual framework. Internally, versioning concepts exposed at the user interface – revision graphs and feature models – are mapped to a generic base layer, the formal foundation of which is propositional logic. Workspace consistency constraints (Sect. 4) and consistency-preserving algorithms (Sect. 5) are formalized upon the base layer.

An *option* o_i represents a (logical or historical) property of a software product that is either present or absent. The *option set* is defined globally:

$$O = \{o_1, \dots, o_n\} \quad (1)$$

Internally, revisions and features are mapped to options transparently.

A *choice* is a conjunction over all options, each of which occurs in either positive or negated form:

$$c = b_1 \wedge \dots \wedge b_n, b_i \in \{o_i, \neg o_i\} \quad (2)$$

A choice can also be represented as a *binding map*, i.e., a set of *binding tuples* (o_i, s_i) , where $s_i \in \{true, false\}$, denotes the boolean selection state of an option o_i . Choices are derived from a user-based selection of a revision and a feature configuration.

Next, an *ambition* is an option binding that allows for unbound options:

$$a = b_1 \wedge \dots \wedge b_n, b_i \in \{o_i, \neg o_i, true\} \quad (3)$$

When represented as a binding map, tuples for unbound options are omitted.

Ambitions are derived from a selection in the feature model, which may leave a number of features unbound in order to describe a *set* of variants to which a change is applied. In the revision graph, the management of ambitions is automated.

A *version rule* is a boolean expression over a subset of options. The *rule base* \mathcal{R} is a conjunction of rules ρ_1, \dots, ρ_m all of which have to be satisfied by a choice in order to be consistent:

$$\mathcal{R} = \rho_1 \wedge \dots \wedge \rho_m, \rho_i \text{ is an expression over } O \quad (4)$$

Version rules are derived automatically from revision graph and feature model [11].

Preferences and defaults have been introduced to ease version selection. A *preference* is a tuple of the form $p_i = (o_i, \pi_i)$, where π_i is an *initialization expression* for option o_i . *Defaults* $d_i = (o_i, s_i)$ define a fallback selection state $s_i \in \{true, false\}$. For each option, at most one default is allowed; preferences have a higher priority.

$$\mathcal{P} = \{(o_{i_1}, \pi_{i_1}), \dots (o_{i_k}, \pi_{i_k})\}, \quad (5)$$

π_{i_j} is an expression over O

$$\mathcal{D} = \{(o_{i_1}, s_{i_1}), \dots (o_{i_l}, s_{i_l})\}, s_{i_j} \in \{true, false\} \quad (6)$$

With $\mathcal{P}\mathcal{D}c$, we denote a choice c to which preferences \mathcal{P} and defaults \mathcal{D} have been applied. The conceptual framework infers preferences and defaults transparently in order to assist the user in selections in the version space. In particular, they automate the management of the revision graph.

Each element e_i of the product space (i.e., the union of feature and domain model) carries a *visibility* v_i , a boolean expression over the variables of O . Visibilities in the feature model are composed only of revision options; visibilities in the domain model are composed of both revision and feature options. An element e_i is called *visible* under a given choice c iff applying c to its visibility v_i (written as $v_i(c)$) evaluates to *true*.

In case the visibility is to be evaluated for an ambition, which is typically not a complete option binding, *three-valued propositional logic* has to be applied. In this case, a third result value, *undefined* can occur.

The operation *filter* is applied during check-out. From a base element set E , those elements e_i that do not satisfy the choice are omitted.

$$E|_c = E \setminus \{e_i \in E \mid v_i(c) = false\} \quad (7)$$

On commit, visibilities must be updated such that inserted (deleted) elements become (in)visible in all choices included in the ambition a .

$$v'_i = \begin{cases} a & \text{if } e_i \in E_{ins} \text{ (insertion)} \\ v_i \wedge \neg a & \text{if } e_i \in E_{del} \text{ (deletion)} \\ v_i & \text{otherwise (no change)} \end{cases} \quad (8)$$

For updates to the domain model, the full ambition a is used; for updates to the feature model, bindings of feature options are omitted from the ambition.

4 Consistency Constraints for Dynamic Filtered Editing

We begin the formalization of the consistency-preserving DFE model with the description of *consistency constraints* based on the formal foundations provided in Sect. 3. We divide a development iteration up into four phases, the first of which is optional when assuming the DFE model: CHECKOUT, MODIFY, COMMIT, and MIGRATE.

The dynamic editing model is depicted in abstract form as a state chart in Fig. 4. Initially, the workspace is in state *Pending*, i.e., not populated yet. On CHECKOUT, a specific version is selected from the repository. After MODIFYing the workspace and COMMITting the changes, the user may either continue with the subsequent iteration, requiring to MIGRATE the choice, or migration is *anceled* (by the algorithm or by the user), triggering a transition back into state *Pending*. To re-populate the workspace, a new choice must be specified then.

The transitions contain preconditions for the application of the corresponding transitions. The numerical values correspond to the constraints presented in the remainder of this section. We use superscripts (ch = check-out, mo = modify, cm = commit, mi = migrate) to delineate the phases of each iteration.

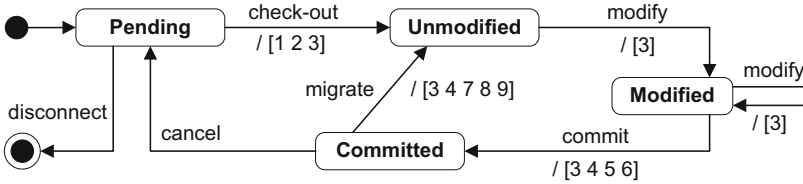


Fig. 4. Workspace operations as transitions in a state diagram.

4.1 Check-Out

In filtered editing, a choice designates a unique version to describe the workspace contents to be checked-out. Therefore, unbound options must not occur.

Constraint 1. *The option binding c^{ch} specified as choice during check-out must be complete with respect to the global option set O^{ch} defined at check-out time.*

$$\forall o \in O^{ch} : (\exists (o, s) \in c^{ch} : s \in \{true, false\}) \quad (9)$$

In the following, we assume in all constraints that choices are complete.

Moreover, the choice must comply with the rules derived by feature dependencies:

Constraint 2. *The choice c^{ch} defined at check-out time must be strongly consistent with the rule base \mathcal{R}^{ch} present at check-out time.*

$$\mathcal{R}^{ch}(c^{ch}) = true \quad (10)$$

Here, $\mathcal{R}^{ch}(c^{ch})$ denotes the evaluation of the rule base \mathcal{R}^{ch} under the choice c^{ch} .

4.2 Modify

By editing the feature model, the user indirectly modifies parts of the option set and of the rule base. It must be avoided that the user introduces rules disallowing consistent version selection in future check-outs.

Constraint 3. *After each modification to the version space, i.e., when saving the feature model, the rule base \mathcal{R}^{mo} must be satisfiable, such that there exists any strongly consistent choice:*

$$\exists c : (\mathcal{R}^{mo}(c) = true) \quad (11)$$

The aforementioned restriction, stating that active features must not be deleted, is beyond the means of formalization available for the base layer, and therefore not presented as an explicit constraint here. See Sect. 5.2.

4.3 Commit

The ambition defined at commit time describes a set of versions, which should comply with the rule base: At least one choice c must exist which agrees with a^{cm} in all common option bindings ($c \Rightarrow a^{cm}$) such that all rules hold under c .

Constraint 4. *An ambition a^{cm} specified during commit must be weakly consistent with the rule base \mathcal{R}^{cm} available at commit time:*

$$\exists c : (c \Rightarrow a^{cm}) \wedge (\mathcal{R}^{cm}(c) = true) \quad (12)$$

In the version determined by the choice, a change is applied *representatively* for the ambition. Thus, there must not be any contradiction between option bindings of the check-out time choice and the commit time ambition inferred from feature selections:

Constraint 5. *The ambition a^{cm} must be represented by the check-out choice c^{ch} .*

$$\forall (o, s) \in a^{cm} : (o, \neg s) \notin c^{ch}, \quad s \in \{true, false\} \quad (13)$$

Requiring no contradictions between choice and ambition does, however, not guarantee that the modifications performed between check-out and commit are representative at product space level. To this end, it must be ensured that the performed change – here represented as a *write set* unionized by inserted and deleted elements $E_{mod} = E_{ins} \dot{\cup} E_{del}$ – could have been equally applied in any other version contained in the ambition:

Constraint 6. *The ambition a^{cm} must be sufficiently specific to the write set E_{mod} .*

$$\forall e \in E_{mod} : (\forall e' \in P^{cm} : (e \xrightarrow{d} e') \Rightarrow v'(P^D a^{cm}) = true) \quad (14)$$

The symbols used in the equation above require further clarification. First, with $e' \in P^{cm}$, we denote any element in the check-out time product space. The premise $e \xrightarrow{d} e'$ checks whether an element of the write set *depends* on e' . Last, v' denotes the visibility of e' before commit, such that $v'(\mathcal{P}^{\mathcal{D}}a^{cm})$ evaluates to true if and only if e' is visible in all versions included in $\mathcal{P}^{\mathcal{D}}a^{cm}$, which is obtained by applying preferences and defaults¹ to the original ambition a^{cm} . Taken together, the constraint checks whether all elements on which any inserted or deleted element depends are visible in all affected versions.

The *depends* operator $e \xrightarrow{d} e'$, where $e \in E_{mod}$ and $e' \in P$, remains to be defined upon the product space base layer. Informally, e depends on e' whenever at least one of the following conditions hold:

- e is a deleted element and e equals e' . (Intuition: Elements must be visible in order to be deletable.)
- e is an inserted element and e' contains e . (Intuition: The insertion location, i.e., the container of an inserted element, must be visible.)
- e is an inserted element and e' is cross-referenced from e . (Intuition: When an inserted element represents the applied occurrence of an existing element, the latter must be visible.)

4.4 Migrate

Transitioning to post-commit time, it is assumed that the operation `MIGRATE`, to be formally defined later, produces a choice that is used for the next iteration. Therein, the same workspace can be reused in connection with the migrated choice.

Due to modifications of the rule base, the choice c^{ch} specified at check-out time may become *incomplete* with respect to the option set O^{cm} , and/or *inconsistent* with the rule base \mathcal{R}^{cm} at commit time. Such temporary inconsistencies are explicitly allowed in order to support feature model evolution. However, before starting the subsequent iteration, it is required that the version to be modified must be uniquely and consistently identified by c^{mi} .

Constraint 7. *The option binding c^{mi} describing the choice after migration must be complete with respect to the commit time option set O^{cm} :*

$$\forall o \in O^{cm} : (\exists (o, s) \in c^{mi} : s \in \{true, false\}) \quad (15)$$

Constraint 8. *The migrated choice c^{mi} must remain strongly consistent with the rule base \mathcal{R}^{cm} available at commit time:*

$$\mathcal{R}^{cm}(c^{mi}) = true \quad (16)$$

¹ In this way, a “more complete” ambition is obtained, which represents, however, the same set of product versions as a^{cm} . The options additionally included in $\mathcal{P}^{\mathcal{D}}a^{cm}$ may occur in visibilities v' , therefore $v'(\mathcal{P}^{\mathcal{D}}a^{cm})$ will less likely return *undefined*.

Apart from this, it is required that the migrated choice c^{mi} must still comply with ambition a^{cm} , which represents changes applied to the current workspace. Since all newly introduced options are mandatory to be selected or deselected for the next choice, total *inclusion* (implemented by propositional logical implication in the opposite direction) is required:

Constraint 9. *An ambition a^{cm} must include the migrated choice c^{mi} describing the workspace contents for the subsequent iteration:*

$$c^{mi} \Rightarrow a^{cm} \tag{17}$$

5 Consistency-Preserving Algorithms

In this section, we contribute detailed algorithms for the operations CHECKOUT, MODIFY, COMMIT, and MIGRATE, which are represented by transitions in Fig. 4. In addition to algorithmic descriptions, their properties are discussed, supplying proof where adequate. The algorithms contain interactive statements, which are underlined in the descriptions below. We use subscripts (r, f, d) to explicitly refer to different dimensions (revision graph, feature model, domain model) of version space and product space. Moreover, unless specified differently, we assume that all variables O , \mathcal{R} , c , etc., are initialized with the value of the corresponding variable at the end of the preceding phase.

5.1 Check-Out

The purpose of CHECKOUT is to populate an empty (i.e., Pending, cf. Fig. 4) workspace with a consistent product version uniquely defined by the user with the help of revision graph and feature model.

Algorithm 1 prompts the user for a revision selection. Using preferences and defaults introduced during COMMIT (see below), it is ensured that options of the selected revision as well as all predecessors are bound to *true*, whereas remaining options are bound to *false*, making the revision choice complete.

Next, the feature model, whose elements' visibilities exclusively refer to revision options, is filtered by the revision choice. In the filtered feature model, the user specifies a feature configuration; invisible options for deleted features are bound to *false* by corresponding defaults. The effective choice c^{ch} is calculated by union of revision and feature choice, before preferences and defaults are applied to it. Next, the feature choice is checked for completeness and strong consistency.

After a product well-formedness analysis, which is not subject of this paper, the workspace is populated with filtered versions of feature and domain model. The feature choice is memorized to enable a later re-construction of the checked-out workspace.

Algorithm 1. Consistency-preserving CHECKOUT. From [6, Algorithm 1].

procedure CHECKOUT

$r_i^{ch} \leftarrow$ option in O_r^{ch} belonging to a selected revision i

$c_r^{ch} := (r_i^{ch}, true)$

$\mathcal{P}^{\mathcal{D}} c_r^{ch} \leftarrow$ apply preferences \mathcal{P}_r and defaults \mathcal{D}_r to (c_r^{ch})

$P_F^{ch} \leftarrow P_f^{ch} |_{\mathcal{P}^{\mathcal{D}} c_r^{ch}} \quad \triangleright$ Filter the feature model; Eq. (7)

Export P_F^{ch} into the workspace

$c_f^{ch} \leftarrow$ select feature configuration in the exported filtered feature model

$c^{ch} \leftarrow c_r^{ch} \cup c_f^{ch}$

$\mathcal{P}^{\mathcal{D}} c^{ch} \leftarrow$ apply preferences \mathcal{P} and defaults \mathcal{D} to (c^{ch})

if not $(\forall o \in O^{ch} : (\exists (o, s) \in \mathcal{P}^{\mathcal{D}} c^{ch} : s \in \{true, false\}))$ **then** \triangleright Constraint 1
 return error “Choice is not complete.”

else if not $(\mathcal{R}^{ch}(\mathcal{P}^{\mathcal{D}} c^{ch}) = true)$ **then** \triangleright Constraint 2
 return error “Choice is not strongly consistent.”

$P_D^{ch} \leftarrow P_d^{ch} |_{\mathcal{P}^{\mathcal{D}} c^{ch}} \quad \triangleright$ Filter the domain model; Eq. (7)

Export (P_D^{ch}) into the workspace

Memorize c_f^{ch} for the subsequent commit

Properties. If successful, Algorithm 1 transitions the workspace into state Unmodified and produces a choice both complete and strongly consistent with respect to the check-out time rule base, such that Constraints 1 and 2 are ensured. In case the user specifies an incomplete or inconsistent choice, the action is canceled and the workspace remains in state Pending.

5.2 Modify

The consistency of the domain model is supposed to be ensured by the respective single-version editing tools employed. Feature model editing, however, is restricted in order to prevent some conflicting situations during COMMIT. We explicitly formulate these restrictions by providing algorithms that redefine the operations SAVEFEATUREMODEL and DELETEFEATURE.

Feature Model Editing. Constraint 3 must be enforced; otherwise, no consistent variant can be specified at later check-outs. To this end, we redefine the SAVEFEATUREMODEL operation of the feature model editor in Algorithm 2 in a way that only satisfiable feature models can be persisted in the workspace.

Feature Deletion. Furthermore, *deletion* of features in the workspace version of the feature model is redefined (see Algorithm 3): First, the operation is only applicable to features bound to *false* in the current choice; otherwise, the feature model would become unsatisfiable, or *choice migration* (see Sect. 5.4) would transfer the positive selection state to the choice to be derived for the next iteration, where the deleted feature and corresponding realization artifacts are supposed to be hidden.

Algorithm 2. Redefined SAVEFEATUREMODEL operation in feature model editor.

```

procedure SAVE( $P_F^{mo}$ )
  if not  $\exists c : (\mathcal{R}_f^{mo}(c) = true)$  then                                 $\triangleright$  Constraint 3
    return error "Feature model is not satisfiable."
  else
    Persist  $P_F^{mo}$  in its current state

```

Algorithm 3. Redefined DELETEFEATURE operation in feature model editor.

```

procedure DELETEFEATURE( $D$ )
   $o_D \leftarrow$  option belonging to feature  $D$  to be deleted
   $c_f^{ch} \leftarrow$  feature choice memorized during preceding check-out or migration
  if  $(o_D, true) \in c_f^{ch}$  then
    return error "Cannot delete feature active in current choice."
  else
    for all  $C \in$  children of  $D$  do
      if DELETEFEATURE( $C$ )  $\neq$  success then
        Undo all modifications related to children of  $D$ 
        return error "Error during deletion of child  $C$ ."
    set the deleted flag of  $D$  to true
     $\mathcal{D}_f^{mo} := \mathcal{D}_f^{mo} \cup \{(D, false)\}$ 

```

Second, rather than persistently deleting a feature, it is merely hidden from the user's display and thus not available in the current and future revisions of the editable feature model. Nevertheless, its feature option, which still may occur in visibilities of domain model elements, remains. To maintain *completeness* of future choices, a negative default is introduced for the feature option.

In order to maintain the hierarchical consistency of the feature model, feature deletion is recursively applied to all child features.

Properties. Constraint 3 is actively enforced by Algorithm 2. Whenever the save operation has been applied successfully, the workspace enters (or remains in) state **Modified**.

5.3 Commit

A consistency-preserving COMMIT operation is formalized in Algorithm 4. As a first step, the revision graph is handled automatically, introducing a new revision option along with a preference and a default ensuring that a single revision selection will yield complete and consistent revision choices in future (see Sect. 6). By using the latest revision as reference point, a linear version history is enforced. Besides, the user specifies a *feature ambition*.

It is ensured by corresponding checks that feature ambitions must be weakly consistent with the rule base (Constraint 4) and represented by the previous choice (Constraint 5).

Next, the checked-out workspace state is reconstructed and differentiated with its commit time version. Based on the deduced difference, it is now ensured that the specified ambition is sufficiently specific to the performed change (Constraint 6).

In case all checks are passed, inserted elements are added to the product space. Next, visibilities of inserted and deleted elements are updated as defined by Eq. 8. For visibility updates applied to the feature model and domain model, respectively, a_r^{cm} and a^{cm} are used.

Algorithm 4. Consistency-preserving COMMIT. From [6, Algorithm 2].

procedure COMMIT

$c_f^{ch} \leftarrow$ feature choice memorized during preceding check-out or migration
 $r_i \leftarrow$ option of most recently committed revision i (*head*)
 $c^{ch} \leftarrow c_f^{ch} \cup \{(r_i, true)\}$
 $\mathcal{PD} c^{ch} \leftarrow$ Apply preferences and defaults to c^{ch}
 $i + 1 \leftarrow$ new revision, successor of i , with user-specified details (commit message, etc.)
 $r_{i+1} \leftarrow$ new revision option for revision i
 $O_r^{cm} \leftarrow O_r^{ch} \cup \{r_{i+1}\}$
 $\mathcal{R}_r^{cm} \leftarrow \mathcal{R}_r^{ch} \wedge (r_{i+1} \Rightarrow r_i)$
 $\mathcal{P}_r^{cm} \leftarrow \mathcal{P}_r^{ch} \cup \{(r_i, r_{i+1})\}$
 $\mathcal{D}_r^{cm} \leftarrow \mathcal{D}_r^{ch} \cup \{(r_{i+1}, false)\}$
 $P_F^{ch} \leftarrow P_f^{ch} |_{\mathcal{PD} c^{ch}} \quad \triangleright$ Reproduce latest revision of feature model
 $P_D^{ch} \leftarrow P_d^{ch} |_{\mathcal{PD} c^{ch}} \quad \triangleright$ Reproduce latest revision of selected variant of domain model
 $P_F^{cm} \leftarrow$ Import current workspace version of the feature model
 $a_f^{cm} \leftarrow$ select feature ambition in the current workspace version of the feature model
 $a^{cm} \leftarrow a_f^{cm} \cup (r_{i+1}, true)$
if not $(\exists c : (c \Rightarrow a^{cm}) \wedge (\mathcal{R}^{cm}(c) = true))$ **then**
 return error “Ambition is not weakly consistent.” \triangleright Constraint 4
else if not $(\forall (o, s) \in a^{cm} : (o, \neg s) \notin c^{ch})$ **then**
 return error “Ambition is not represented by choice.” \triangleright Constraint 5
 $P_D^{cm} \leftarrow$ Import current workspace version of the domain model
Match and differentiate P^{cm} with P^{ch} .
 $E_{ins} \leftarrow$ inserted elements according to the difference.
 $E_{del} \leftarrow$ deleted elements according to the difference.
 $\mathcal{PD} a^{cm} \leftarrow$ Apply preferences and defaults to a^{cm}
if not $(\forall e \in E_{mod} : (\forall e' \in P_D^{cm} : (e \xrightarrow{d} e') \Rightarrow v'(\mathcal{PD} a^{cm}) = true))$ **then**
 return error “Ambition is not sufficiently specific to the change.” \triangleright Constraint 6
 $P \leftarrow P \cup E_{ins}$ \triangleright Add inserted elements to the repository
for all $e_i \in E_{ins} \cup E_{del}$ **do**
 $v_i \leftarrow v'_i$ \triangleright Update visibilities; see Eq. 8

Properties. If successful, Algorithm 4 transitions the workspace into the state Committed, while ensuring Constraints 4, 5, and 6 for the specified ambition.

Otherwise, the workspace remains in state Modified; in this case, the user may re-attempt the commit with a different ambition.

5.4 Migrate

The operation MIGRATE prepares the workspace choice for the subsequent iteration, proceeding under the assumption that the user prefers to stay in the current view. Unlike CHECKOUT and COMMIT, this operation is not triggered explicitly by the user, but automatically after COMMIT. Conversely, it makes the subsequent CHECKOUT optional, tying on the non-disruptive revision control workflow.

Algorithm 5 iterates over options that remain unbound in the choice to be migrated. If a corresponding option has been bound in the ambition, the binding is transferred to the choice. Otherwise, preferences and defaults are triggered as far as applicable, with the aim to complete c^{mi} transparently. As a “last resort”, a binding state is obtained non-deterministically. Since the new option has been ignored in the ambition, there cannot exist any reference to it in updated visibilities. Therefore, it is immaterial for the subsequent choice whether or not the option is selected. At this point, it is not known how new (and still unbound) features will be incorporated in the next iteration. Therefore, the user may choose among the set of choices describing the current workspace contents equivalently.

Algorithm 5. Consistency-preserving MIGRATE. From [6, Algorithm 3].

```

procedure MIGRATE
  for  $o \in O^{cm}$  do
    if  $(o, true) \notin c^{mi} \wedge (o, false) \notin c^{mi}$  then  $\triangleright$  Never override existing bindings
       $s^{mi} \leftarrow undefined$ 
      if  $\exists(o, s) \in a^{cm} : s \in \{true, false\}$  then
         $s^{mi} \leftarrow s$   $\triangleright$  Infer from ambition
      else if a preference  $p \in \mathcal{P}^{cm}$  is applicable to  $o$  then
         $s^{mi} \leftarrow$  apply  $p$  to  $o$ 
      else if a default  $d \in \mathcal{D}^{cm}$  is applicable to  $o$  then
         $s^{mi} \leftarrow$  apply  $d$  to  $o$ 
      else
         $s^{mi} \leftarrow$  user selection for  $o$ 
        if  $s^{mi} = undefined$  then
          return error “Operation was canceled by the user.”
         $c^{mi} \leftarrow c^{mi} \cup \{(o, s^{mi})\}$ 
    if not  $\mathcal{R}^{cm}(c^{mi}) = true$  then  $\triangleright$  Constraint 8
      return error “Cannot migrate to a consistent choice.”
    else
      Memorize  $c^{mi}$  for the subsequent commit  $\triangleright$  Obviate check-out

```

Properties. If migration succeeds, a *strongly consistent choice* (Constraint 8) is actively enforced by the algorithm. Theorems 1 and 2 prove that Constraints 7 and 9 are satisfied, respectively.

There are three possible causes for failure of this operation. First, there may be no correct solution regardless of the user selections performed². Second, the user might introduce a contradiction although a different selection would have provided a correct migrated choice. Third, the user may cancel intentionally.

If migration succeeds, the workspace immediately enters state **Unmodified**. Otherwise, entering state **Pending** triggers an exceptional check-out, forcing the user into specifying a new choice.

Theorem 1. *After having applied MIGRATE successfully, Constraint 7 is satisfied.*

Proof. The algorithm iterates over all options in O^{cm} , which equals O^{mi} as no options can be introduced between commit and migrate by any operation. In each iteration, either *true* or *false* are definitely assigned to missing bindings in c^{mi} . Therefore, after having processed all available options, c^{mi} is *complete* (as required by Constraint 7).

Theorem 2. *After having applied MIGRATE successfully, Constraint 9 is satisfied.*

Proof. Being its descendant, c^{mi} includes c^{ch} . Moreover, a^{cm} is weakly consistent with c^{ch} (cf. Constraint 5). Thus, no contradictions exist between c^{mi} and a^{cm} . Bindings for missing options are transferred from the ambition, or if not applicable, in a way that does not contradict with any ambition binding. Altogether, the migrated choice c^{mi} is included in the ambition a^{cm} (such that $c^{mi} \Rightarrow a^{cm}$, as required by Constraint 9).

6 Automated and Consistent Revision Graph Management

Above, it has been claimed that the consistency of the revision graph is guaranteed automatically, such that the user is not accosted with constraint violations in this dimension. As new revisions created during commit are appended as successor of the latest revision, the here described commit strategy makes the revision graph degenerate into a *sequence*.

Below, we supplement proof for the satisfaction of the constraints defined in Sect. 4 by the historical dimension in isolation. The remainder of this section is structured by phases, of which MODIFY has been omitted since it does not affect the revision graph.

² In such a case, newly introduced feature model rules prevent the product version available in the workspace from being reproduced by future check-outs. The performed modifications are, however, valid for different versions included in the ambition.

6.1 Check-Out

It has to be proved that the choice inferred from the selection of a single revision in Algorithm 1 satisfies the check-out time consistency constraints.

Theorem 3. *A revision choice derived at check-out is complete (Constraint 1).*

Proof. Preferences and defaults are applied in advance to filtering. On commit, a default of the form $(r_i, false)$ is introduced for each revision i , such that no unbound revision option remains after having applied all defaults.

Theorem 4. *A revision choice derived at check-out is strongly consistent (Constraint 2).*

Proof. There are two types of rules to be potentially violated: initial revision rules (r_0) and predecessor rules $(r_{i+1} \Rightarrow r_i)$.

Except for the selected revision option r_i , all revision options are bound by preferences or defaults. In Algorithm 4, it is ensured that for each invariant $r_{i+1} \Rightarrow r_i$, a preference (r_i, r_{i+1}) is created. Through repeated application of this preference, after a revision option r_i has been selected, all predecessor revisions are bound positively. For no predecessor of r_i , a negative binding will be created since defaults have a lower priority than preferences. Therefore, all predecessor rules are satisfied.

Given the premise of a linearly organized revision graph, repeated application of predecessor preferences will propagate to r_0 , regardless of which revision has been selected. Therefore, $(r_0, true)$ will occur in every binding derived this way, such that the initial revision constraint r_0 is satisfied.

6.2 Commit

During COMMIT (cf. Algorithm 4), a new revision with option r_{i+1} is introduced for the successor of the current head revision i . We prove that the derived revision ambition $(r_{i+1}, true)$ satisfies the constraints associated with the commit phase.

Theorem 5. *A revision ambition derived at commit is weakly consistent (Constraint 4).*

Proof. To be weakly inconsistent, it would be required that $(r_{i+1}, true)$ contradicts with any invariant in \mathcal{R}_r^{cm} . The only invariant in which r_{i+1} can appear is the newly introduced $r_{i+1} \Rightarrow r_i$. Though, $(r_i, false) \notin a_r^{cm}$, thus weak consistency is given.

Theorem 6. *A revision ambition derived at commit is represented by the check-out time revision choice (Constraint 5).*

Proof. Not yet existing at check-out time, r_{i+1} is unbound in the revision choice c_r^{ch} , such that no contradiction with $(r_{i+1}, true)$ can occur.

Theorem 7. *A revision ambition derived at commit is sufficiently specific to describe the historical component of a workspace change (Constraint 6).*

Proof. We represent the visibilities v of all elements as conjunctions $v_f \wedge v_r$. Furthermore, we assume that all elements e' on which modified elements $e \in E_{mod}$ may depend, have passed the choice: $v'_r(c^{ch}) = true$. After applying option binding completion, ${}^{\mathcal{P}\mathcal{D}}a_r^{cm} \supset c_r^{ch}$. As a consequence, $v'_r({}^{\mathcal{P}\mathcal{D}}a_r^{cm}) = true$ for all v'_r .

6.3 Migrate

During MIGRATE (cf. Algorithm 4), the binding $(r_{i+1}, true)$ is transferred from the ambition to the choice. We first consider the common case that the selected revision equals the *head*; for the subsequent proofs, we may therefore presume $c_r^{mi} = c_r^{ch} \cup a_r^{cm}$, hence $c_r^{cm} \Rightarrow a_r^{cm}$.

Theorem 8. *The migrated revision choice is complete (Constraint 7).*

Proof. c_r^{ch} is complete with O_r^{cm} except for the only new option r_{i+1} , which is, however, bound in the ambition a_r^{cm} and therefore transferred from there. Thus, $c_r^{mi} = c_r^{ch} \cup a_r^{cm}$ is complete.

Theorem 9. *The migrated revision choice is strongly consistent (Constraint 8).*

Proof. c_r^{ch} is consistent with respect to \mathcal{R}_r^{ch} . We may assume that $\mathcal{R}_r^{cm} = \mathcal{R}_r^{ch} \wedge (r_{i+1} \Rightarrow r_i)$. From the choice, $r_i = true$. From the ambition, $r_{i+1} = true$. Taken together, the new predecessor invariant is fulfilled ($true \Rightarrow true$).

Theorem 10. *The migrated revision choice is included in the revision ambition (Constraint 9).*

Proof. Since the binding $(r_{i+1}, true)$ is transferred from the ambition to the migrated choice, $a_r^{cm} \subset c_r^{mi}$.

In case the check-out time choice did not equal the latest revision, however, MIGRATE will definitely fail because $r_{i+1} \Rightarrow r_i$ is violated due to the negative selection state of r_i set by the revision default. In this case, an explicit CHECK-OUT, including a consistent revision selection, is enforced (cf. Fig. 4).

7 Generalized Editing Model

The functional properties of the here contributed *dynamic* editing model are now compared to the so considered conventional approach, *static filtered editing* (SFE). Concrete representatives are discussed in the related work section; we here use the vocabulary used in the own conceptual framework.

Commonalities. The filtered editing (FE) approaches considered here have in common that they operate in an *iterative* way, where each iteration is a transaction begun with *check-out* and concluded with *commit*. In between, workspace contents may be modified. The workspace is defined by a *choice* – or read filter –, which is a *unique* version selection. By an *ambition* – a *partial* write filter –, the versions affected by the change are defined. This way, the version described by the choice is *representative* for the set of versions described by the ambition.

Static Filtered Editing. In SFE, both the choice and the ambition are defined in one step at the beginning of a workspace transaction, i.e., at check-out. Typically, an ambition is defined first as a partial version selection, which is further configured top-down into a unique choice. Both choice and ambition, as well as the version space itself, are immutable during *modify*. After *commit*, the transaction is closed and the workspace is cleared; subsequent transactions must be initiated explicitly. Changes to the variability model are allowed when no workspace transaction is active.

Dynamic Filtered Editing. In DFE, the ambition is specified at *commit* time. Furthermore, the variability model is made available for modification in the workspace. This way, it is possible to introduce those features to which a change is relevant while the change is actually performed. Moreover, a new transaction is started immediately after *commit*. It is assumed that the same choice as in the previous iteration shall be used—an assumption that is obtained from generalizing the VCS workflow (and supported by the operation *migrate* presented above).

Figure 5 illustrates the different optional and mandatory phases of the respective iterations and aligns them with the constraints presented in Sect. 4.

The remainder of this section sketches – without providing formal definitions or proofs – how the editing model assumed so far can be generalized in order to support static filtered editing as well as blended forms of the editing models. The here presented framework primarily assumes DFE, but it also allows to step back to SFE in case a more restrictive workflow is desired.

7.1 Purely Static Filtered Editing

The comparison above makes obvious that SFE requires only a subset of the constraints investigated here. This is due to the missing evolution of the feature model, as well as due to the lack of a MIGRATE operation; a check-out is required in advance to each iteration. Furthermore, the order of consistency checks is different because the ambition is selected during check-out already.

The algorithms presented in Sect. 5 can be adjusted for SFE as follows:

- Ambition selection – including the automated revision graph management – is moved from COMMIT to CHECKOUT, more precisely to after filtering the feature model. Constraint 4 (weak ambition consistency) is preponed accordingly.

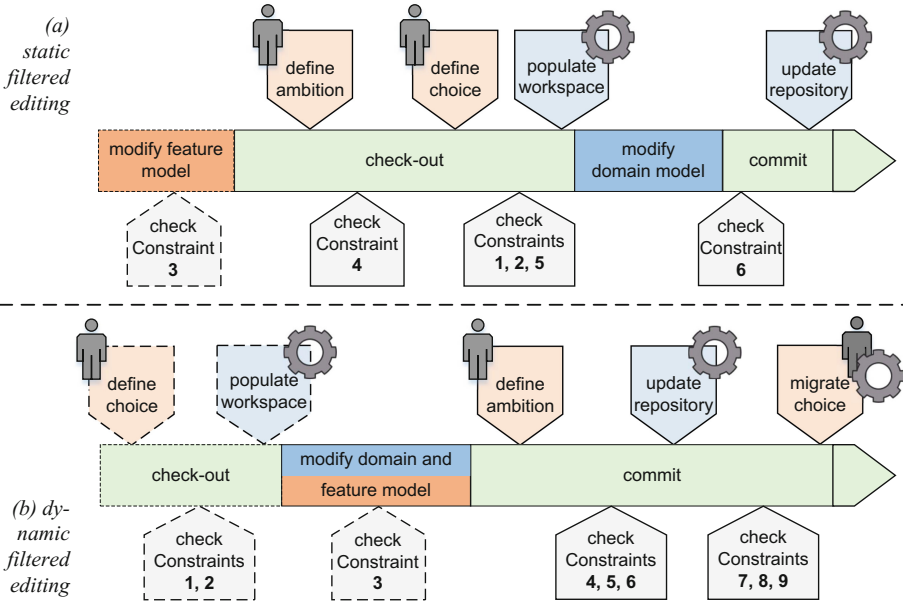


Fig. 5. Static vs. dynamic filtered editing by phases and constraints.

- As the feature model is not made available for modification, it is not exported into the workspace during CHECKOUT.
- Constraints 1 and 2 are ensured before filtering the domain model as in DFE. Directly afterwards, Constraint 5 (choice represents ambition) is checked. If this constraint fails, it is not the ambition, but the choice that has to be altered.
- During MODIFY, only the domain model can be edited.
- The only constraint that remains to be checked at COMMIT is Constraint 6. If this constraint fails, however, the user cannot be asked for a new ambition. Rather, the *write set* representing the product-level changes must be revised in order to be *sufficiently general*. This requires dedicated tool support in addition.
- MIGRATE (see Algorithm 5) is abandoned entirely.
- Feature model modification is allowed between workspace transactions in an unfiltered editing mode.

7.2 Restricted Transactions

More restrictive forms of (static or dynamic) filtered editing can be approached by tailoring DFE towards only one product dimension, the feature model or the domain model.

Feature Model Transaction. In this form of restricted transaction, only the feature model is made available in the workspace. Since this is versioned

exclusively by the revision graph, whose consistency is managed automatically (see Sect. 6), constraint validation becomes entirely transparent to the user, who is accosted only with selections in the revision graph and, e.g., with commit messages.

Domain Model Transaction. It is also desirable in many scenarios to remove the feature model from the workspace, or to make it unmodifiable. Then, the co-evolution problems motivating Constraints 3, 7, 8, and 9 become irrelevant.

A concrete workflow similar to the one implied by the SFE model can be realized by applying restricted feature model transactions and domain model transactions *alternatingly*. Furthermore, *nested* transactions are conceivable: A feature model transaction may aggregate several domain model transactions, all committed under the same historical scope.

7.3 Earlier Ambition Specification

Without negatively affecting the flexibility of DFE, the action *specify ambition* can be preponed to any point in time after *specify choice*. This allows the user to fix the scope of the intended change earlier than during commit; furthermore, the check of Constraints 4 (weak ambition consistency) and 5 (ambition represented by choice) can be applied earlier, which may prevent a subset of consistency problems. Constraint 6 (sufficiently specific ambition) still needs to be checked during commit.

7.4 The Amend Operation

DFE can also be extended in order to allow corrections of the ambition *after* commit. We semi-formally define an additional workspace operation AMEND, which behaves as follows:

- The user makes a selection in the revision graph. The chosen revision option is r_i .
- The feature model P_f for revision r_i is reconstructed.
- The user is asked to define a new feature ambition a'_f in O_f as a substitute for a_f .
- Constraint 4 is re-checked using the new ambition a'_f and \mathcal{R}_f . If the constraint is violated, the operation is aborted.
- In the visibilities of elements of the domain model, all occurrences of the term a_f are replaced by a'_f .

This operation potentially behaves less consistent than the conventional way of ambition specification. In particular, Constraints 5 and 6 are ignored.

8 Evaluation

In order to quantitatively evaluate the user-visible properties of the presented constraints and consistency-preserving algorithms, we report on two data sets

extracted from case studies that refer to standard examples from SPL literature. In addition to the product line for *graphs* [12,14] considered above, the second case study refers to a product line for *Home Automation Systems (HAS)* originally introduced in [3] and adapted in [15].

Table 1. Aggregate results quantifying the user complexity of the dynamic filtered editing model.

	Graph	HAS
Number of iterations	9	38
Feature model size	8	17
Domain model size	26	106
Average ambition complexity	$8/9 = 0.89$	$41/38 = 1.08$
% migration interactions	$1/9 = 0.11$	$5/38 = 0.13$
% explicit check-outs	$3/9 = 0.33$	$6/38 = 0.16$
% commits canceled	$0/9 = 0$	$1/38 = 0.026$

Methodology. All results were obtained by analyzing recorded version histories of previously conducted case studies. While the *graph* product line was realized by the authors themselves, the *HAS* study was conducted by a master student with MDPLE background. In order to foster an incremental style of development, requirements were communicated to the subjects in consecutive interview sessions. In the *HAS* case study, hypothetical customer feedback was given, such that it became necessary to revise or to newly define features and their corresponding domain artifacts.

The size of feature model and domain model reported below can be obtained by analyzing the repository contents manually. The four additional quantities measured demand for further explanation:

1. *Average ambition complexity:* The complexity of an ambition is defined as the number of features bound in it. Negatively bound features are treated as two bindings (as feature expressions derived from them contain an additional syntax tree element for the negation). An ambition may also have complexity 0 when no feature is bound for a universal change (case $a = true$).
2. *Migration interactions:* The ratio of editing model iterations in which at least one user interaction is required during *migration*.
3. *Explicit check-outs:* The ratio of iterations where the optional *check-out* operation is necessary in case the previously applied migration does not produce the desired choice for the next iteration.
4. *Commits canceled:* The ratio of *commit* operations canceled after consistency violations had been reported, such that further domain or feature model editing was required.

Results and Conclusions. Table 1 summarizes quantitative results extracted from both case studies. To begin with, (1) the complexity of ambitions is in average close to 1, allowing for the assumption that ambitions frequently consist of only a single feature binding; (2) *migration* happens transparently for the larger part of iterations; (3) the *check-out* operation, which has been made optional by the dynamic editing model, is necessary only in a small ratio of iterations; (4) only in one of altogether 47 iterations was it necessary to cancel the commit (in this case, the feature the subject intended to realize was accidentally missing in the feature model). In contrast, the larger part of iterations did neither require a *check-out* nor user interaction during *migrate*. When compared to SFE, the low number of commits canceled suggests that the dynamism gained by the filtered editing model is not paid with a significant loss of consistency.

Threats to Validity. The significance of the results derived from Table 1 is potentially limited by two factors. First, it was clearly communicated to the evaluators that the scope of the changes applied in one iteration should be equal, leading to comparably short-running iterations. However, in real-world scenarios, inexperienced users may accidentally realize several different features in one iteration, making it impossible to specify a valid global ambition. Second, cooperative versioning was faded out in both case studies. We expect the number of canceled commits to slightly increase with multi-user support due to problems such as doubly introduced features.

9 Related Work

This paper ties on previous publications regarding a conceptual framework to integrate SPLE, MDSE, and VC; see [6, 11, 14, 15]. Below, we refer to approaches explicitly dealing with workspace consistency problems appearing with filtered SPL editing.

Fully, Partially, and Temporarily Filtered SPL Editing. Approaches to filtered editing of (model-driven) software product lines may be categorized under *fully filtered editing*, *partially filtered editing*, and *temporarily filtered editing*.

Fully filtered multi-variant editing was influenced by early concepts of *multi-version editors* such as *P-Edit* [10]. They assume that a view – similar to the workspace in the here considered framework – is created from a multi-variant document, which corresponds to the repository here. The version available in the view is defined by a *choice* that uniquely denotes a representative of the *ambition*. The here presented approach realizes fully filtered editing. It does, however, not assume or require a specific multi-version editor, but allows arbitrary tools to be used for editing the workspace. To this end, the operations that define and interpret the choice and ambition are provided by generalized version control abstractions, which have been extended by SPLE concepts.

Partially filtered editing [8] aims at hiding variants to which the current change is immaterial, without requiring the choice to be *unique* (Constraints 1 and 7). There is only a single filter serving as choice and ambition simultaneously. Variability information referring to non-resolved configuration options is presented in the view, e.g. in the form of annotations. As a consequence, specific tools or preprocessor languages are required in order to cope with variability in the workspace.

A source-code centric approach to *temporarily filtered editing* of software product lines is described in [7]. A partial feature configuration can be specified as *write filter*. Code fragments immaterial for the so intended change are hidden. As approximation of a read filter, a *context* is derived as an extended view on the write filter. Similarly, the *FeatureMapper* [16] approach, which is based on annotative variability, offers a temporary write filter in the multi-variant view. Having selected one or more features and invoked the *record* operation, all changes performed in the MVDM are associated with a feature expression derived from the provided feature selection.

View-Based vs. Transactional Filtered Editing. An orthogonal distinction can be made between the categories *view-based* and *transactional* filtered editing. For starting and closing transactions, different metaphors are provided in the literature.

In the first case, the filter – which may be further decomposed into read and write filter – can be dynamically changed by fading in and out configuration options. Altering the filter directly influences the visible workspace contents. This is realized, e.g., in [7,8]. View-based filtered editing, however, requires specialized multi-version editors or at least a tight integration into existing editors. This makes the approach difficult to implement, particularly in model-driven development environments.

In contrast, the *transactional* approach assumes well-defined iterations during which the read filter remains equal. In the here presented approach as well as in the precursor UVM [13], transactions are opened and closed by generalized forms of the VCS metaphors *update* and *commit*. The approach presented in [9] defines similar operations, *get* and *put*. P-EDIT [10] relies on the metaphors of conditional compilation, introducing a *write* operation that closes a transaction by a specific write filter. In *FeatureMapper* [16], temporary transactions are opened and closed by starting and stopping change recording (see above); the view is inferred from a feature selection similar to feature ambitions.

Static vs. Dynamic Filtered Editing. Representatives of *static* filtered editing, e.g., UVM [13], require that the ambition is specified at check-out time; since the rule base does not evolve, constraints dealing with its evolution are unnecessary. Similarly, in [8], having a single filter requires that the scope of a change must be known beforehand, inhibiting the concurrent introduction of a feature and its realization.

The approach presented by [9] moves the specification of the write filter from check-out time to commit time, which slightly deviates from strict SFE as defined above. As the version space is not represented explicitly, no co-evolution problems may occur, and no dynamism is required for the editing model.

In the presented *dynamic* filtered editing model, the variability model may evolve during an iteration embraced by *check-out* and *commit*. In particular, new configuration options and new configuration rules may be introduced this way. The flexibility implied by this approach is – to the best of the author’s knowledge – unique in the literature. Correspondingly, the implied consistency problems described by Constraints 7, 8, and 9, have been described and analyzed for the first time. Moreover, the operation MIGRATE contributed in Algorithm 5 is novel.

Generality of the Write Set. In the list of dynamism-aware consistency constraints provided here, Constraint 6 inhibits a special role, considering not only the soundness of the version space (i.e., options, rule base, choice, and ambition), but also of its connection to the product space. Phrased in the vocabulary used in this paper, this constraint ensures that “the ambition is specific enough to reproduce the change in all affected variants”, or conversely speaking “the change is general enough to be reproduced in all variants included in the ambition”.

The potential inconsistencies that may occur by having the user inadvertently change a larger set of variants than he/she intends to do – namely the variants that contain those elements visible in the view – have been recognized in the literature previously. In [10], a distinction is made between *fixed* and *unfixed* fragments made available in the workspace. Fixed fragments are visible in all variants included in the ambition, whereas unfixed fragments are visible only in a part of the ambition that includes the choice. Unfixed fragments must be managed in a more or less restrictive way. P-EDIT graphically highlights unfixed fragments, such that the developer becomes aware of a potentially unintended modification of hidden artifacts.

The *edit isolation principle* described in [8] states that “the only variants that change in the source are those that can be reached from the view”, where “source” denotes the multi-version representation. This has been used as the central design constraint for the specification of an *update* (here: commit) operation, which does not only detect, but also repair situations in which the principle is violated. When compared to the edit isolation principle, Constraint 6 is even more restrictive since it disallows, among other, modifications that insert or destroy cross-links to invisible elements.

10 Conclusion

We have motivated, formally developed, and reflected on a consistency-preserving dynamic filtered editing model that supports the iterative creation of model-driven software product lines in a single version view by adopting version control principles.

The following implicit assumptions are underlying the DFE model: First, the users want to specify all information referring to the version space *as late as possible*; therefore, the definition of an ambition is deferred to the COMMIT phase, and modifications to the feature model need not be performed in advance to an iteration, but may be incorporated during MODIFY. Second, the user wants to be accosted with version specification tasks *as seldom as possible*, which vindicates the decision to make the CHECKOUT operation optional and to introduce MIGRATE. Third and last, the editing model should be *no more restrictive than necessary* in order to prevent the user from performing changes that cause product inconsistencies or that cannot be potentially reproduced.

Consistency is checked by explicit constraints assigned to the different editing model phases. During CHECKOUT, it is ensured that the specified feature configuration is *complete* and *consistent* with respect to the feature model. During MODIFY, changes that would make the feature model unsatisfiable are inhibited. The feature ambition defined by the user during COMMIT is checked for *weak consistency* as well as for being *represented* by the choice. Furthermore, it must be *sufficiently specific* for the performed product-level change. The newly introduced operation MIGRATE automatically produces a choice for the next iteration based on the previous choice and the ambition, in order to obviate repeated check-outs that reproduce the current workspace view. The migrated choice is checked for *completeness* and *consistency* with the evolved version of the feature model. Furthermore, it must *include* the ambition.

The correctness of the consistency-preserving operations with respect to the underlying consistency constraints has been formally proved where adequate. Furthermore, we have supplied evidence that the revision graph is managed not only automatically, but also consistently. An experimental evaluation based on aggregated data from existing version histories of two case studies confirms that the DFE model performs flexibly and non-disruptively, while the amount of inconsistencies reported is insignificantly higher than a comparative static solution to filtered editing.

The decision whether to apply *static* or *dynamic filtered editing* is related to the amount of *flexibility* (i.e., late ambition specification, co-evolution of feature model and domain model) and of *consistency guarantees* (i.e., by preventing certain co-evolution problems) required by a specific project. In the presented conceptual framework, it is assumed that DFE is the preferred style, but SFE can be adopted gradually in case a more restrictive workflow is demanded.

Several ways of generalizing the presented DFE model have been sketched. To begin with, purely static filtered editing model may be realized by moving and deleting some version selection statements and constraint checks between the algorithms. Restricted transactions may ensure that only the feature model or only the domain model are edited. Furthermore, the ambition may also be specified at an earlier point in time, which slightly increases the consistency at the expense of a more restricted editing model. Last, the AMEND operation even allows to retrospectively alter the ambition used for a previous commit, such that erroneous version specifications can be revised.

The algorithms presented in this paper have been implemented as part of the Eclipse-based filtered MDPLE tool *SuperMod* [14]. For modification of feature models, for defining and migrating feature configurations, as well as for specifying feature ambitions, dedicated tree-oriented editors and dialogs are provided.

Future work will address the syntactical correctness of single-version domain models checked-out into the workspace. It has also turned out that Constraint 6 is, in some cases, too restrictive, for instance when it concerns the graphical representation of model elements; therefore, we will introduce a mechanism to define exceptions to the corresponding constraint check. Besides, the suitability of DFE for *agile SPLE* processes will be examined.

References

1. Völter, M., Stahl, T., Bettin, J., Haase, A., Helsen, S.: Model-Driven Software Development: Technology, Engineering, Management. Wiley, Hoboken (2006)
2. Conradi, R., Westfechtel, B.: Version models for software configuration management. *ACM Comput. Surv.* **30**, 232–282 (1998)
3. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer, Berlin (2005). <https://doi.org/10.1007/3-540-28901-1>
4. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (FODA) feasibility study. Technical report CMU/SEI-90-TR-21, Carnegie-Mellon University, Software Engineering Institute (1990)
5. Gomaa, H.: Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. Addison-Wesley, Boston (2004)
6. Schwägerl, F., Westfechtel, B.: Maintaining workspace consistency in filtered editing of dynamically evolving model-driven software product lines. In: Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2017, Porto, Portugal, 19–21 February 2017, pp. 15–28. SCITEPRESS (2017)
7. Kästner, C., Trujillo, S., Apel, S.: Visualizing software product line variabilities in source code. In: Proceedings of the 2nd International SPLC Workshop on Visualization in Software Product Line Engineering (ViSPL), pp. 303–313 (2008)
8. Walkingshaw, E., Ostermann, K.: Projectional editing of variational software. In: Generative Programming: Concepts and Experiences, GPCE 2014, Vasteras, Sweden, 15–16 September 2014, pp. 29–38 (2014)
9. Stanculescu, S., Berger, T., Walkingshaw, E., Wasowski, A.: Concepts, operations and feasibility of a projection-based variation control systems. In: 2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, 2–7 October 2016, pp. 323–333. IEEE (2016)
10. Sarnak, N., Bernstein, R.L., Kruskal, V.: Creation and maintenance of multiple versions. In: Winkler, J.F.H. (ed.) Software Configuration Management. German Chapter of the ACM, vol. 30, pp. 264–275. Teubner (1988)
11. Schwägerl, F., Buchmann, T., Uhrig, S., Westfechtel, B.: Towards the integration of model-driven engineering, software product line engineering, and software configuration management. In: Hammoudi, S., Pires, L.F., Desfray, P., Filipe, J. (eds.) Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development, Angers, France, pp. 5–18. SCITEPRESS (2015)

12. Lopez-Herrejon, R.E., Batory, D.: A standard problem for evaluating product-line methodologies. In: Bosch, J. (ed.) GCSE 2001. LNCS, vol. 2186, pp. 10–24. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44800-4_2
13. Westfechtel, B., Munch, B.P., Conradi, R.: A layered architecture for uniform version management. *IEEE Trans. Softw. Eng.* **27**, 1111–1133 (2001)
14. Schwägerl, F., Buchmann, T., Westfechtel, B.: SuperMod - a model-driven tool that combines version control and software product line engineering. In: Proceedings of the 10th International Conference on Software Paradigm Trends, Colmar, Alsace, France, pp. 5–18. SCITEPRESS (2015)
15. Schwägerl, F., Buchmann, T., Westfechtel, B.: Filtered model-driven product line engineering with SuperMod: the home automation case. In: Lorenz, P., Cardoso, J., Maciaszek, L.A., van Sinderen, M. (eds.) ICSoft 2015. CCIS, vol. 586, pp. 19–41. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-30142-6_2
16. Heidenreich, F., Kopcsek, J., Wende, C.: FeatureMapper: mapping features to models. In: Companion Proceedings of the 30th International Conference on Software Engineering (ICSE 2008), pp. 943–944. ACM, New York (2008)



Model-Driven STEP Application Protocol Extensions Combined with Feature Modeling Considering Geometrical Information

Thorsten Koch^{1(✉)}, Jörg Holtmann¹, and Timo Lindemann²

¹ Software Engineering Department, Fraunhofer IEM, Paderborn, Germany
thorsten.koch@iem.fraunhofer.de

² Emmet Software Labs GmbH & Co. KG, Bad Salzuffen, Germany

Abstract. Original equipment manufacturers (OEMs) build mechatronic, variant-rich systems using components from several suppliers in industry sectors like automation. The OEMs have to integrate the different components to the overall system based on a virtual layout. For this purpose, the suppliers provide geometrical information via the standardized exchange format STEP. Beyond the geometrical information, the OEMs need additional logical and technical information for the integration task as well as the variant handling. For that reason, STEP provides an extension mechanism for extending and tailoring STEP to project-specific needs. However, extending STEP requires extending several capabilities of all involved tools, which prevents the project-specific utilization of the STEP extensions mechanism. In order to cope with this problem, we presented in previous work a model-driven approach enabling the flexible specification of STEP extensions and particularly the automatic derivation of the required capability extensions for two involved tools. Nevertheless, the OEMs still need to apply several engineering tools from different domains to consider logical as well as geometrical constraints between product variants. In this paper, we hence combine our previous approach with extended feature models that consider conventional logical and particularly geometrical information, thereby enabling a holistic product line engineering for mechatronic systems. By means of an automation production system example, we illustrate how OEMs can orchestrate their overall supply and development processes through the combination of both approaches.

Keywords: STEP · Model-driven software development
Meta-modeling · Model transformation · Product line engineering
Feature models · Geometrical constraints

1 Introduction

The development of mechatronic systems in industry sectors like automation is characterized by complex supply chains, where original equipment manufacturers (OEMs) build an overall system using physical components from several

suppliers. An example of such a system is depicted in Fig. 1. The OEM integrates this overall automation production system, a so-called *Pick & Place Unit (PPU)* [1]. The PPU encompasses the four components **Stack**, **Ramp**, **Crane**, and **Stamp**, which are delivered by suppliers. The **Stack** works as workpiece input storage and the **Ramp** acts as workpiece output storage. The **Stamp** is responsible for labeling the workpieces, and the **Crane** is responsible for transporting the workpieces by picking and placing them between the different working positions. The **Crane** transports workpieces from the **Stack** to the **Stamp**. After the **Stamp** has processed a workpiece, the **Crane** transports the workpiece finally to the **Ramp**.

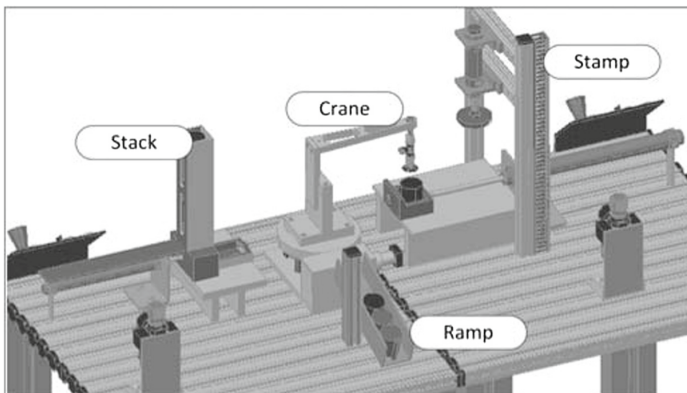


Fig. 1. Pick & Place Unit as an example for a simple automation production system [1].

Figure 2 sketches the exchange of product information between the OEM and different suppliers in the development process of a mechatronic product line like the PPU. In the course of integrating the overall system, one of the most important development tasks of the OEM is to geometrically assemble the overall system based on the particular supplier components. Prior to the actual production of the overall system, this task is performed by means of a virtual geometric layout within *computer-aided design (CAD) tools*. The suppliers geometrically design their particular components within CAD tools, too. Based on these designs, they provide geometrical information about their components via the standardized exchange format *STandard for the Exchange of Product data (STEP)* [2], such that the OEM is able to virtually layout the overall system (cf. STEP-based exchange of geometrical information in Fig. 2).

Beyond the geometrical information, the OEM needs additional technical information (e.g., the admissible payload of the Crane manufactured delivered by Supplier A and the power consumption of the Stamp delivered by Supplier B in Fig. 2) to perform his development tasks. For that reason STEP, provides an extension mechanism for extending and tailoring STEP to project-specific needs.

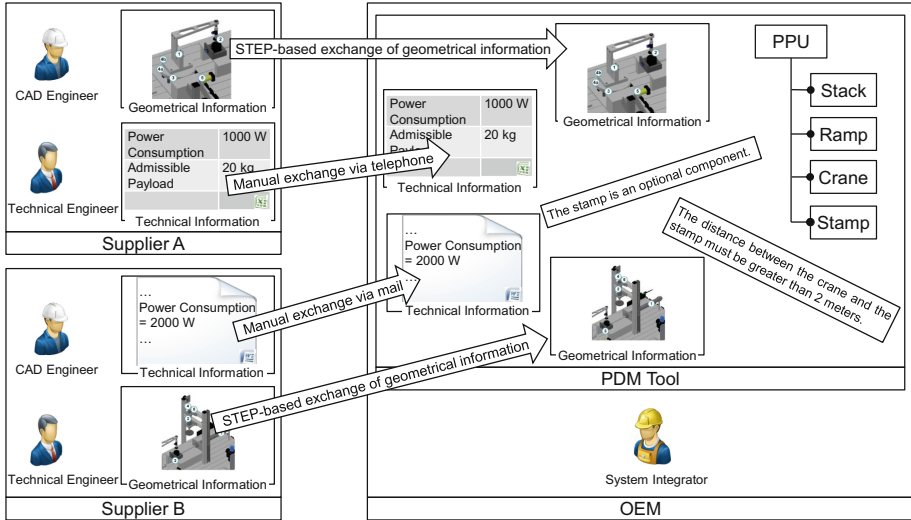


Fig. 2. Overview of the exchange of product information between the OEM and different suppliers.

Typical applications of the STEP extension mechanism have been reported in [3, 4], for example.

However, extending STEP moreover requires extending the capabilities of all involved tools for the specification, the data exchange, and the interpretation of the additional technical information. That is for one thing, all affected suppliers have to extend their CAD tools such that they are able to specify and export the additional information. For another thing, the OEM has to extend his CAD tool such that he is able to import and interpret the additional information. These tool extensions have to be implemented through plugins and application programming interfaces on the side of all involved organizations, which causes a high implementation effort. Thus, the application of the STEP extension mechanism is restricted to static, one-off, and long-term tool chains, which do not fulfill the needs of today's and future dynamic business processes (cf. the recommendations for implementing the feature “digital end-to-end engineering” for dynamic value chains in the context of Industry 4.0 [5]).

The fixedness of the STEP extension mechanism leads to a tool chain as exemplary sketched in Fig. 2. Beyond the specification of geometrical information in CAD tools and the corresponding standardized data exchange via unextended STEP, the suppliers specify the respective additional technical information outside their CAD tools. This additional information is awkwardly exported to the OEM via different communication channels (e.g., phone, office documents via mail, or electronic data interchange—EDI—formats [6]). In the example in Fig. 2, Supplier A specifies the additional product information like the power consumption and the admissible payload of the component in Excel sheets and

exchanges this information via telephone (cf. **Manual exchange via Telephone** in Fig. 2). **Supplier B** documents the power consumption in a Word document and exports the information via mail (cf. **Manual exchange via Mail** in Fig. 2). Furthermore, the OEM faces the challenge of component-wisely storing and grouping the geometrical as well as additional information within a *product data management (PDM) tool*.

In order to cope with this problem, we introduced in previous work [7] a complex application of existing meta-modeling and model transformation techniques that enables the flexible specification of STEP extensions. This particularly included the automatic derivation of the required capabilities of two involved tools for the specification, the data exchange, and the interpretation of additional technical information.

However, the development of mechatronic systems is not only characterized by complex supply chains, but also by high customer expectations regarding the product individualization and modularity. This results in a large variety of product variants and a lot of *logical* as well as *geometrical constraints*. For instance, the PPU example is able to handle two different kinds of workpieces: metal and plastic. If a product variant is supposed to handle metal workpieces, it requires a Crane with an admissible payload larger than 20 kg to handle the workpieces (logical constraint). Furthermore, as indicated in Fig. 2, the Crane requires a minimum distance of 2 m to other components to guarantee a safe operation (geometrical constraint). Whereas the OEMs use approaches from product line engineering (like feature modeling [8]) for the specification and validation of logical constraints in their PDM tool, they apply their CAD tool to specify and validate the geometrical constraints. This leads to awkward redundancies between both kinds of engineering tools.

In order to cope with the complex data exchange between the OEM and his suppliers on the one hand, and the mixed specification and validation of logical as well as geometrical constraints in different engineering tools on the other hand, we integrate in this paper our model-driven approach for the flexible specification of STEP extensions with another previous work [9] that extends feature models to support both logical and geometrical constraints. Based on the PPU example, we show how the OEM can orchestrate the overall supply and development process by the combination of both approaches.

The remainder of this paper is structured as follows. In the next section, we introduce fundamentals about STEP. Afterward, we present our model-driven data exchange approach integrated with extended feature models in Sect. 3 and conduct a case study in Sect. 4. Section 5 covers related work. Finally, Sect. 6 concludes this paper with a summary and an outlook on open future work.

2 ISO 10303 - Standard for the Exchange of Product Data (STEP)

The International Organization for Standardization has published the ISO 10303 - Standard for the Exchange of Product data (STEP) [2] to address the problem

of exchanging product data between different systems. The overall objective of STEP is to provide a mechanism that describes a complete and unambiguous product definition throughout the entire life-cycle of a product. Furthermore, STEP provides a system independent and computer interpretable file format for the exchange of product data between different software tools, like computer-aided design (CAD) or simulation tools [10]. However, STEP especially focuses on the representation of geometrical information.

To realize the objective of a complete and unambiguous product definition, STEP defines so-called application protocols [2]. An application protocol is a data model tailored to the specific needs of an application area. In the scope of this paper, we use the application protocol STEP AP214. Although the STEP AP214 is originally designed for the automotive domain, it is broadly used in practice, since it describes product information like sheet-metal parts of the car body, mechanical parts of the engine, and glass components. Thereby, the STEP AP214 is also suitable for the exchange of product information in the application of automation production systems.

In an application protocol, the description of product data is defined in the *EXPRESS information modeling language* [11]. EXPRESS is part of the ISO 10303 and has been defined to model geometry information. EXPRESS consists of language elements that allow unambiguous data definition and specification of constraints on the defined data. The most important EXPRESS element is the *entity* data type, which defines the objects of interest in the domain being modeled. The *entity* is characterized by its attributes and constraints. The EXPRESS information modeling language also supports various kinds of data types, including *simple types*, *aggregations types*, and *constructed types* [11].

STEP defines two different file formats for the exchange of product data: physical file [12] and XML file [13]. Whereas the XML file is an XML encoding for the product data defined by an application protocol, the physical file is a purely ASCII encoding for product data. In the scope of this paper, we use the physical file format, since it is mostly used by exchange systems today to read and write STEP data [10].

Figure 3 depicts an overview of the relationship between the EXPRESS information modeling language, a STEP application protocol, and the actual product information contained in a STEP file. The EXPRESS information modeling language has been developed prior to the Meta Object Facility (MOF) [14] standard of the OMG. However, in terms of the MOF standard, the EXPRESS information modeling language is the meta-meta-model used to specify STEP application protocols by means of a grammar. The STEP application protocol is the meta-model used to specify the structure of the product information. The STEP file is the model containing the actual product information following the structure in the application protocol.

In the remainder of this section, we use the running example of the PPU to illustrate the different parts of the STEP standard. Therefore, Listing 1 depicts an excerpt of the STEP AP214 defined by means of the EXPRESS information modeling language showing the four entities `product_context`, `product`, `line`, and `cartesian_point`. The `product_context` contains the single attribute `discipline_type`

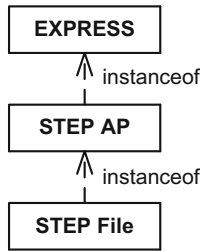


Fig. 3. Overview of the relationship between EXPRESS, STEP application protocols and STEP files [7].

of the type label. The type label represents a `STRING`. The product contains the attribute `id`, `name` and `description`; all of type `STRING`. Furthermore, it contains a list of references of `product_contexts`.

Listing 1. Exemplary excerpt of the STEP AP214 defined in EXPRESS [7].

```

1  TYPE label = STRING;
2  END_TYPE;
3
4  ENTITY product_context;
5    discipline_type : label;
6  END_ENTITY;
7
8  ENTITY product;
9    id:STRING;
10   name:STRING;
11   description:OPTIONAL STRING;
12   frame_of_reference:SET [1:?] OF product_context;
13  END_ENTITY;
  
```

Listing 2 depicts an excerpt of the physical file of Crane component of Pick&Place Unit. As mentioned before, a physical file is a pure ASCII encoded file with a simple structure. Each line of a physical file encompasses an identifier encoded `#id` and a key-value pair encoding the actual product information. For example, in Line 1 of Listing 2, the `product` is defined. The entity has the identifier `#86`, the `id` and `name` `HT_L1600`. The identifier is also used to encode cross-references between different entities. For example, the entity `product` contains a reference to the identifier `#91`.

Listing 2. Exemplary excerpt of a STEP AP214 file [7].

```

1  #86=PRODUCT('HT_L1600','HT_L1600','',(#91));
2  #91=PRODUCT_CONTEXT(' ',#93,'mechanical');
  
```

3 Flexible Specification of STEP Extensions

In this section, we present the integration of our model-driven approach for the flexible specification of STEP extensions with an extensions to feature models to support both logical and geometrical constraints. Figure 4 depicts an overview

of the approach encompassing four main contributions for the OEM and his suppliers to improve the problematic situation described in Sect. 1. First, the OEM as well as his suppliers are enabled to specify additional technical information directly in their tools (cf. -1- Specification of additional technical information in Fig. 4). For this purpose, we enable the OEM to specify a central data model that can be tailored to the specific needs of a particular development project. The central data model acts as an alternative to a PDM tool, which only has the capability to component-wisely store arbitrary artifacts (like CAD models and documents) but not to interpret model-based information from different sources. Furthermore, it contains all geometrical and technical information and is also the main artifact of our approach from which we derive the other parts using model-driven techniques. Furthermore, we provide an extension to the CAD tools of the suppliers and the product line engineering tool of the OEM based on the STEP extensions specified for the central data model. Second, we are able to derive an automatic data exchange for the involved tools (cf. -2- Automatic exchange of product information in Fig. 4). The specification of additional technical information as well as the automatic data exchange result in a machine-readable and processable representation of the product information (cf. -3- Interpretation of the additional technical information in Fig. 4). Finally, the previous contributions enable an integrated development process for mechatronic product lines (cf. -4- Integrated product line engineering in Fig. 4).

In the following section, we describe a systematic model-driven process to support the creation of the central data model. Furthermore, we present the technical details of the three process steps Automatic Derivation of the Central Data Model and Data Import, Generate CAD Extensions, and Generate Feature Model Extension in the subsequent Sects. 3.2, 3.3, and 3.4, respectively.

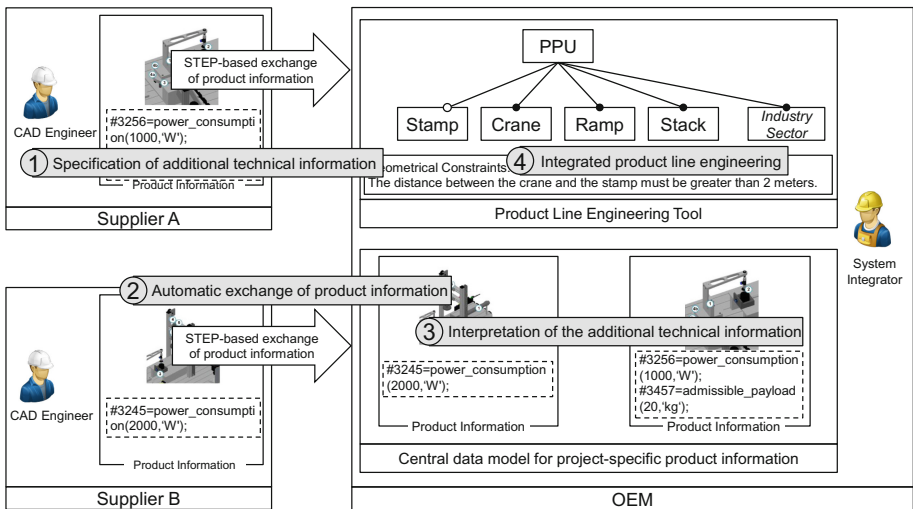


Fig. 4. Overview of our model-driven approach for the exchange of product information.

3.1 Process for the Creation of the Central Data Model

Figure 5 depicts our model-driven process to support the creation of the central data model. The process is specified by means of the Business Process Model and Notation (BPMN) [15]. The main contributions of this paper are emphasized in Fig. 5 with gray tasks and artifacts. We visualize manual steps by means of BPMN manual tasks (hand in the upper left corner of the task). Steps that we could automate are visualized by means of BPMN service tasks (cogwheel in the upper left corner of the task). Work results are specified by means of BPMN data objects (document icons), and persistent models that are subject to update and retrieval operations are specified by means of BPMN data stores (database icons).

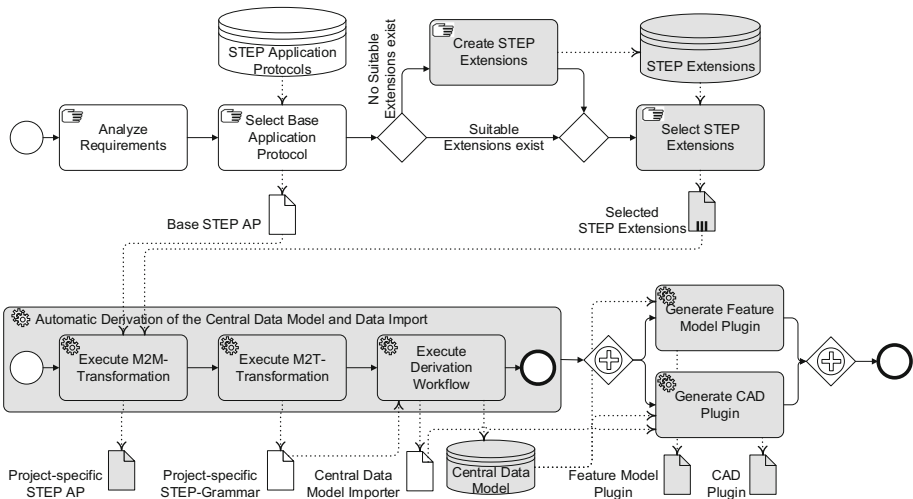


Fig. 5. Overview of the model-driven process to support the creation of the central data model.

In the following, we exemplarily perform and explain each process step depicted in Fig. 5 referring to the PPU as a running example. We design the model-driven process in such a way that the OEM has to perform it, but may need to discuss several aspects with his suppliers.

In the first process step *Analyze Requirements*, the OEM decides which information is necessary for the current development project and should be stored in the central data model. For the development of the PPU, the OEM decides that the power consumption of all used components and the admissible payload of the Crane must be stored in the central data model besides the regular geometrical information.

In the second process step *Select Base Application Protocol*, the OEM selects an application protocol from the STEP Application Protocol library that fulfills

most of the analyzed requirements and that acts as a basis for the central data model. The library only contains application protocols that are officially defined in the ISO 10303. In our running example, the OEM decides to use the STEP AP214 as the Base STEP AP.

As mentioned in Sect. 1, STEP usually does not cover all product information that is needed for the development of the overall system. Hence, the OEM uses the next two process steps Create STEP Extensions and Select STEP Extensions to enrich the selected Base STEP AP with further descriptions of product information. For this purpose, we enable the OEM to create new STEP extensions in an EXPRESS-based textual editor and to store these extensions in a STEP Extension library. Furthermore, we enable him to select existing STEP extensions from the library that satisfy his specific needs.

In our running example, the STEP Extension library already contains several STEP extensions. While reading through the descriptions of these STEP extensions, the OEM noticed that the STEP_EXTENSION POWER_CONSUMPTION depicted in Listing 3 already satisfies the requirements on the specification of a component's power consumption.

Listing 3. STEP extension for the specification of a power consumption [7].

```

1  SCHEMA STEP_EXTENSION POWER_CONSUMPTION;
2  ENTITY power_consumption;
3      component: product;
4      value: NUMBER;
5      unit: Unit;
6  END ENTITY;
7  END_SCHEMA;
```

The STEP_EXTENSION POWER_CONSUMPTION only contains the entity power_consumption. This entity refers to the entity product (cf. Sect. 2) defined in the STEP AP214. Furthermore, the entity power_consumption contains an attribute value of the type NUMBER and a reference to a unit. As mentioned before, this application protocol is sufficient to specify the description of a components power consumption in a machine-readable manner. Thus, the OEM decides to reuse this STEP extension. Since the STEP Extension library does not contain a suitable STEP extension for the specification of the admissible payload of a Crane component, the OEM defines a new STEP extension STEP_EXTENSION ADMISSIBLE_PAYLOAD as depicted in Listing 4. The structure is analogous to the previous STEP extension. After the OEM has specified the STEP extension, he stores it in the STEP Extensions library to enable its reuse in further development projects.

Listing 4. STEP extension for the specification of an admissible payload [7].

```

1  SCHEMA STEP_EXTENSION ADMISSIBLE_PAYLOAD
2  ENTITY admissible_payload;
3      component: product;
4      value: NUMBER;
5      unit: Unit;
6  END ENTITY;
7  END_SCHEMA;
```

After the selection of the required STEP extensions, the automatic derivation process of the central data model (cf. Automatic Derivation of the Central Data Model and Data Import in Fig. 5) is executed. The automatic derivation process encompasses three subprocesses: Execute M2M-Transformation, Execute M2T-Transformation, and Execute Derivation Workflow. In the first subprocess, Execute M2M-Transformation, the conceived model-to-model transformation merges the Selected STEP Extensions into the selected Base STEP AP to derive a Project-specific STEP AP. This Project-specific STEP AP contains the description of all product information that should be contained in the central data model. In the subsequent subprocess Execute M2T-Transformation, the OEM executes our developed model-to-text transformation to derive a Project-specific STEP Grammar. This grammar enables the automatic derivation of the central data model and the corresponding import capabilities as described in the subsequent section (cf. Execute Derivation Workflow in Fig. 5).

The last two process steps Generate CAD Plugin and Generate Feature Model Plugin are executed in parallel. In the process step Generate CAD Plugin, the extensions of the CAD tools for the supplier are generated. These extensions enable the specification of entities of the central data model within the user interface of the CAD tool. Furthermore, it provides a mechanism to store the product information and to export it to a physical file (cf. Sect. 2). In the process step Generate Feature Model Plugin, the extensions for the feature model tool of the OEM is developed. By means of the Feature Model Plugin, the system integrator is able to reuse the entities stored in the central data model including all geometrical and additional information during the specification of the product line. Therefore, the plugin provides user interface elements to select entities from the central data model.

Concluding, the introduction of the model-driven process, we obtain the specification capabilities of geometrical and additional technical product information by defining flexible STEP extensions. The OEM is enabled to describe a central data model by selecting an existing STEP application protocol as basis and by defining and/or selecting STEP extensions to enrich the basis STEP application protocol. The resulting project-specific STEP application protocol is further used to automatically derive the required capabilities for the data exchange between the OEM and his suppliers. Furthermore, it is used to derive extensions for existing CAD systems on the one hand, and a self-developed feature model tool on the other hand. While the extensions of CAD systems enable the specification, storage, and exchange of additional technical product information, the extensions of the feature model tool enable the reuse of additional technical product information for the specification of logical as well as geometrical constraints. Both are needed in a holistic product line engineering for mechatronic systems, like the PPU example.

3.2 Automatic Derivation of the Central Data Model and the Data Import

In this section, we describe the realization of the process step Automatic Derivation of the Central Data Model and Data Import depicted in Fig. 5. For this purpose, we recreated and developed different meta-models, models, and grammars as depicted in Fig. 6. Meta-models are depicted by means of UML classes. Grammars are depicted by means of UML classes with a small rectangle in the upper right corner. Finally, we depicted text files as UML classes with a document icon in the upper right corner and parser as UML classes with a circle in the upper right corner. As the technology icons indicate, we use the Eclipse Modeling Framework [16] to specify meta-models by means of Ecore models, and the Xtext framework [17] to define grammars. While using the Xtext framework, we are able to automatically derive a parser for a particular grammar. Besides the mentioned technologies, we use QVT-O [14] and Xtend¹ to realize model-to-model and model-to-text transformation, respectively.

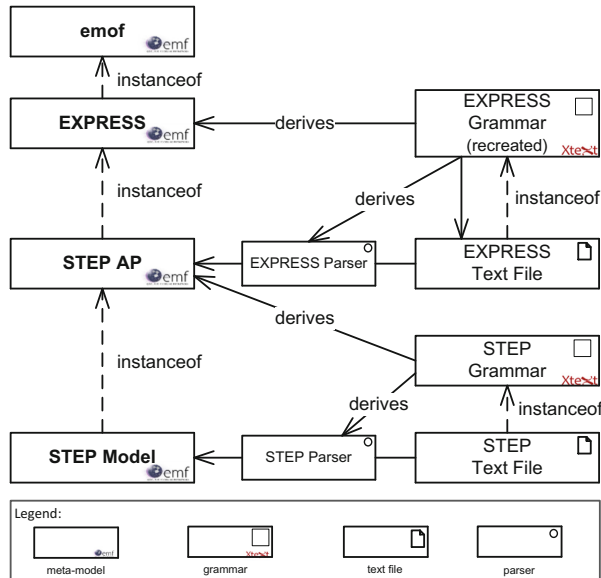


Fig. 6. Overview of the developed meta-models and their relationships [7].

As mentioned in Sect. 2, the EXPRESS information modeling language has been developed in the ISO 10303 prior to the Meta Object Facility (MOF) [14] standard of the OMG. Thus, the EXPRESS information modeling language does not comply to the OMG standard and modern model-driven development techniques are not yet applicable.

¹ <http://www.eclipse.org/xtend/>.

For this reason, we developed our own MOF-compliant meta-model of the EXPRESS information modeling language based on the Eclipse Modeling Framework and the Xtext framework. We used the Xtext framework to recreate the concrete textual syntax of the EXPRESS information modeling language by means of a grammar (cf. EXPRESS Grammar in Fig. 6). While using the generation workflow of the Xtext framework, we derive an Ecore-based meta-model of the EXPRESS information modeling language. Furthermore, we derive a EXPRESS parser that reads textual STEP application protocol files that correspond to the defined grammar.

As mentioned in Sect. 2, a STEP application protocol is defined by means of the EXPRESS information modeling language. Thus, after defining the EXPRESS grammar and deriving its meta-model as well as a corresponding parser, we are able to read and write STEP application protocols. However, in the current stage of our implementation, we are only able to process the basic EXPRESS elements *types* and *entities*. The processing of the remaining EXPRESS elements is left for future work.

A STEP application protocol only defines the structure of the product information, and not the product data itself. Hence, we apply the same technologies to create a grammar representing the product information specified in a STEP application protocol (cf. STEP Grammar in Fig. 6). Furthermore, the STEP Grammar defines the structure of the STEP Text File following the structure defined for STEP physical files (cf. Sect. 2). As depicted in Fig. 6, after the execution of the Xtext workflow, we derive a meta-model for STEP files that reflects the product information defined in the STEP application protocol.

Figure 7 depicts the execution of the automatic derivation process of the central data model for our running example by means of a UML Activity Diagram. After the OEM has performed the process step Select STEP Extensions depicted in Fig. 5, the specification of the central data model in our running example encompasses the STEP AP214 as Base STEP AP, and the two extensions STEP_ExtensionPower_Consumption : EXPRESS and STEP_ExtensionAdmissible_Payload : EXPRESS.

In the first activity M2M-Transformation, the selected Base STEP AP and the two selected STEP extensions are merged into an Project-specific STEP AP by using a model-to-model transformation realized by a QVT-O in-place transformation. This model-to-model transformation iterates over all entities in the different :EXPRESS instances and merges them into the Project-specific STEP AP. If a naming conflict occurs or some references are not yet resolved, the transformation resolves these issues.

After the execution of the merging activity, the resulting Project-specific STEP AP is transformed into an Xtext grammar by means of a model-to-text transformation realized by Xtend (cf. M2T-Transformation). The model-to-text transformation also iterates over all entities and translates them into a grammar that also fulfills the requirements of the ISO 10303-21 for the structure of the final STEP File. Listing 5 depicts an excerpt for the resulting Xtext grammar for the STEP extension shown in Fig. 4. The Xtext grammar defines the entity `power_consumption`, encompassing an ID, a desc, and as shown in Listing 3 a value,

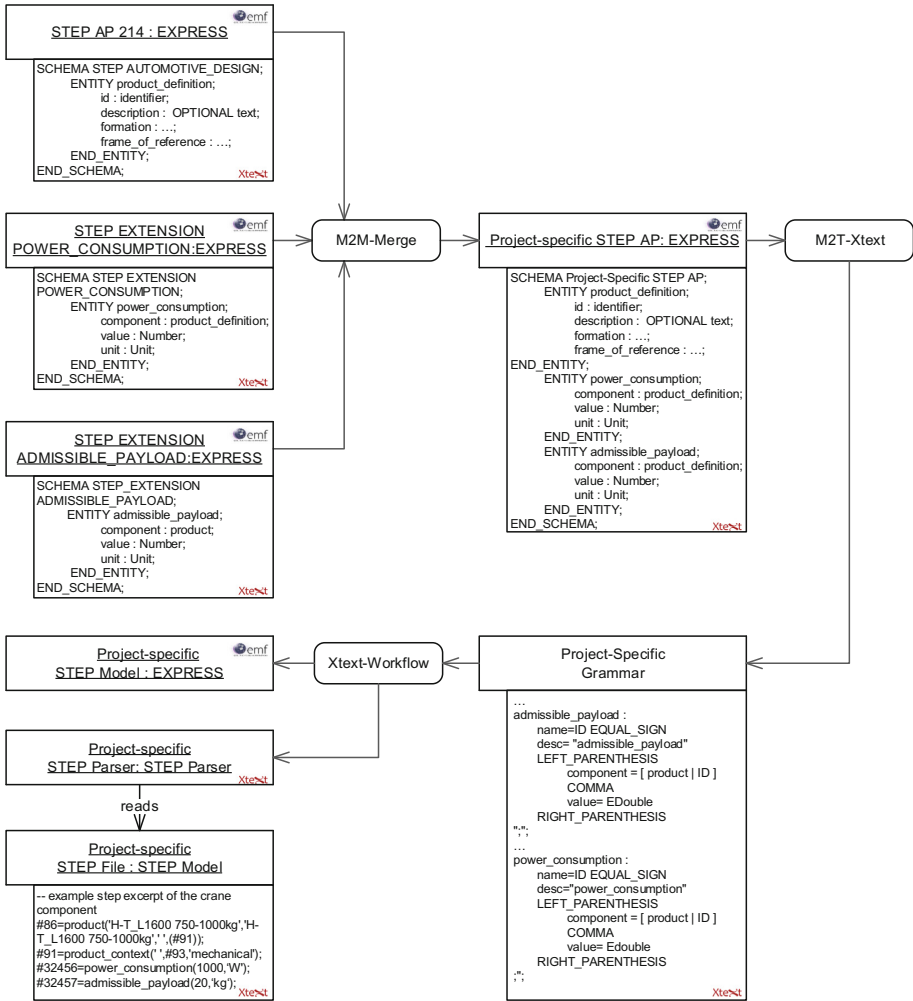


Fig. 7. Overview of the automatic derivation of the central data model and the data import [7].

and a unit. In the final STEP File, the ID corresponds to the line number and acts as an identifier. The desc attribute indicates which entity is currently parsed.

Listing 5. Excerpt of the Xtext grammar for the STEP extension shown in Listing 3 [7].

```

1 power_consumption:
2   name=ID "="
3   desc="power_consumption"
4   "("
5     component = [ product | ID ] ", "
6     value= EDouble
7   ")"
8   ", " ;
    
```

Finally, the workflow of the Xtext framework is executed and as a result, we derive the central data model (cf. Project-Specific STEP Model in Fig. 7). Furthermore, we derive an Project-specific STEP Parse that reads STEP files and creates data models conforming to the central data model.

3.3 Automatic Derivation of the CAD Plugin

In this section, we describe the realization of the process step Automatic Derivation of the CAD Plugin depicted in Fig. 5. The automatic derivation approach has been prototypically implemented for the CAD tool SolidWorks.

We started the development of the automatic derivation approach with an examination of the plugin mechanism of SolidWorks and implemented a reference plugin for an extended user-interface representing further technical product information and for exchanging this information.

After implementing the reference architecture, we generalized the reference plugin, and divided the resulting code into platform, individual, and repetitive code. The platform code is provided by the CAD tool SolidWorks to enable the development of plugins using internal functionality of SolidWorks. We encapsulated the CAD tool dependent code by writing a wrapper and refer to it as individual code. Finally, the repetitive code is used to create an extended user-interface and to create the storage functionality. Since this code only uses operations provided by our own individual code, the repetitive code is independent of the used CAD tool.

In the next development step, we developed a CAD plugin generator based on the individual code. The CAD plugin generator uses the Selected STEP Extensions as input and generates the required user-interface elements for the additional technical information. Furthermore, the CAD plugin generator also generates the required code-fragments to support the exchange of the additional information.

Figure 8 depicts the automatic derivation of the CAD Plugin for our running example. The CAD plugin generator uses the two Selected STEP Extensions STEP_ExtensionPower_Consumption : EXPRESS and STEP_ExtensionAdmissible_Payload : EXPRESS and produces the user-interface elements on the right.

3.4 Automatic Derivation of the Feature Model Plugin

In this section, we describe the realization of the process step Automatic Derivation of the Feature Model Plugin depicted in Fig. 5. The automatic derivation approach has been prototypically implemented for a self-developed product line engineering tool based on feature models. The development of the feature model plugin generator follows the same approach as presented for the CAD plugin generator in the previous section. First, we started with a reference implementation for an extended user-interface representing and accessing technical components stored in the central data model. Second, we generalized the reference implementation, and divided the resulting code into platform, individual, and repetitive code. Finally, we developed a feature model plugin generator based on

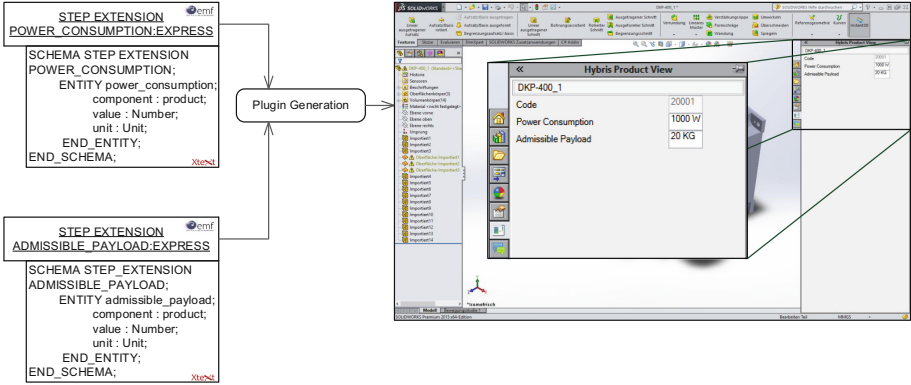


Fig. 8. Overview of the automatic derivation of the CAD plugin [7].

the individual code. The feature model plugin generator uses the Selected STEP Extensions as input and generates the required user-interface elements for the different kinds of entities including their additional technical information.

Figure 9 depicts the automatic derivation of the feature model plugin for our running example. The feature model plugin generator uses the two Selected STEP Extensions STEP_ExtensionPower_Consumption : EXPRESS and STEP_Extension Admissible_Payload : EXPRESS and generates the required user-interface elements to access the geometrical and additional product information. Furthermore, the Feature Model plugin generator also generates the required code-fragments to support the exchange of the additional information. the user-interface elements on the right.

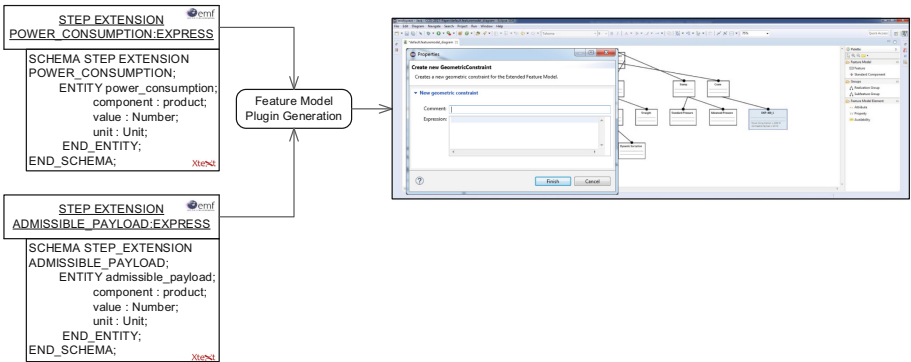


Fig. 9. Overview of the automatic derivation of the Feature Model plugin.

By integrating our model-driven approach for the flexible specification of STEP extensions with our feature model approach, we prevent the OEM from

the mixed specification and validation of logical as well as geometrical constraints in different engineering tools. Our feature model tool supports a variety of feature model extensions like feature attributes and properties [18], logical constraints between features and feature properties [19], and the distinction between features and feature realizations. Furthermore, we enable the configuration of a particular product variant and the verification of its correctness based on the information in the feature model [20,21].

Beyond the specification and verification of logical constraints, we covered the challenge of geometrical constraints in the development of mechatronic product lines and enabled their specification within our feature model tool support. Therefore, we, first, introduced a new kind of realization, a so-called *3D-Realization*. The 3D-Realization is an abstraction of a concrete technical component and encompasses all geometrical information concerning the component, like the width, the height, or the overall boundary. In our feature model plugin, we exploit the concept of the 3D-Realization to enable the access of geometrical information stored in central data model (cf. Fig. 10). Furthermore, by sub-classing the 3D-Realization for each entity in the central data model, we enable the access of the additional technical information stored in the central data model. This additional technical information could be used in logical constraints. For instance, as depicted in Listing 6, if the product variant is supposed to handle metal workpieces, it requires a Crane with an admissible payload larger than 20 kg to handle the workpieces. Thus, in the configuration of such a product variant, the additional technical information of each available Crane could be used to only allow the selection of Cranes that fulfill this constraint.

Listing 6. Example of a logical constraint for the PPU.

```
1 Metal implies Crane.admissible_payload ≥ 15
```

Second, we introduced a geometrical constraint language based on propositional logic for product line engineering [22]. Our geometrical constraint language enables the specification of so-called *assembly constraints* [23]. Assembly constraints are used to arrange components relative to each other. Typical examples are the allowed minimum and maximum distance between two components or whether two components intersect. By using our geometrical constraint language, we are able to formalize the geometrical constraints that the Crane and the Stamp must have a minimum distance of 2 m (cf. Listing 7); without that distance, the Crane would not be able to transport the workpieces and would collide with the Stamp:

Listing 7. Example of a geometrical constraint for the PPU.

```
1 MinimumDistance(Crane, Stamp) ≥ 2m
```

For the configuration of a mechatronic product variant, it is not sufficient to only configure the logical part, but also to virtually layout the product variant according to the geometrical constraints induced by the different components. For this purpose, we defined in [9] a research roadmap, which we recap in the following.

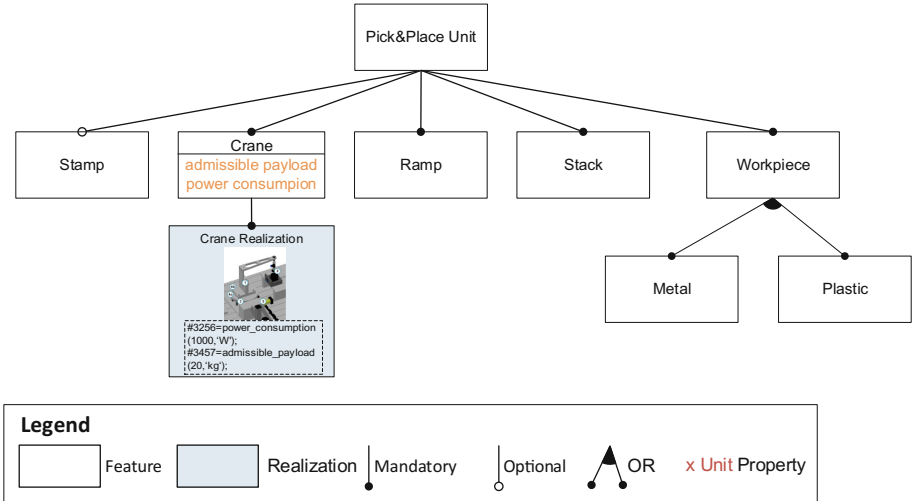


Fig. 10. Integration of feature models with the central data model.

First, we are going to integrate our tool support for the logical configuration of product variants with tool support for the virtual layout of a variant, e.g. in an e-commerce system. Figure 11 depicts the integration sketch. At first, a customer is able to select the technical components for this automation production system in our variant editor. If the product variant is correct, the information of the selected technical units is provided to the tool support for the virtual layout. The CAD model of each selected technical component is loaded from the central data model to enable the virtual layout of the product variant. To obtain the position of each technical component, the tool support for the virtual layout provides this information and stores it in the variant configuration. Thus, the variant configuration contains a complete description of the logical and geometrical configuration of a particular product variant.

Second, we are going to implement a verification of the layout against the specified geometrical constraints. Therefore, we have to implement basic algorithms that calculate the distance between two 3D-Realizations. Furthermore, we want to integrate the verification with a geometric modeling kernel [24], which is used within conventional CAD tools. By means of the geometric modeling kernel, we are able to realize operations that are more complex, for example, whether two lines or areas of two 3D-Realizations intersect. The algorithm uses the layout information stored in the variant configuration and the specified constraints as input. After reaching these two milestones, a potential customer will be able to configure a product variant and virtually layout it according to the assembly constraints.

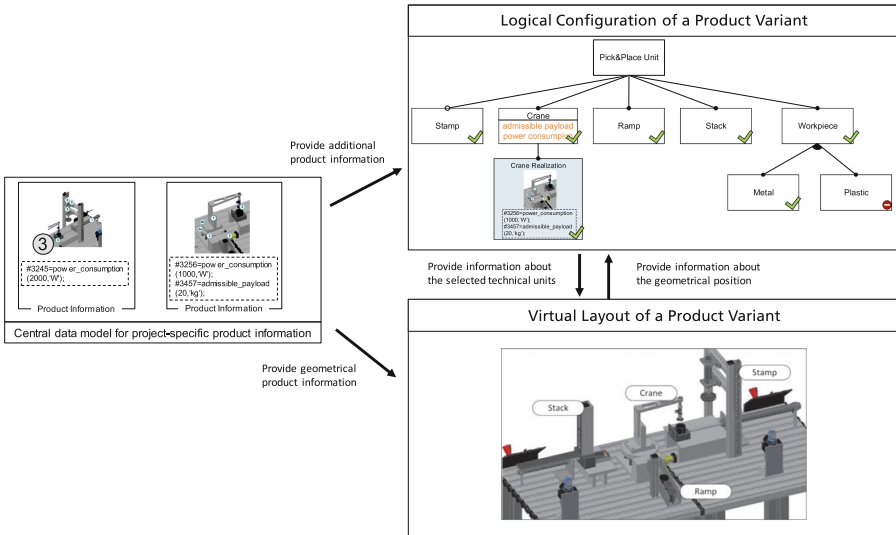


Fig. 11. Sketch for the integration of the tool support for the virtual layout and the variant configuration.

4 Case Study

In this section, we conduct a case study based on the guidelines by [25] for the evaluation of our approach. In our case study, we investigate the applicability and usefulness in practice of our approach. We perform the case study based on the running example in this paper and do not aim at generalizing the case study conclusions to all possible development projects using STEP for the exchange of product information.

4.1 Case Study Context

The objective of our case study is to evaluate whether our model-driven approach for the creation of a central data model is applicable and useful for the OEM and his suppliers, i.e., whether it reduces the manual effort in deriving tool support for the overall information exchange. For this purpose, we use the two STEP extensions of the running example and different STEP application protocols. We concentrate on the investigation of the applicability and usefulness in practice of our approach especially for the automatic derivation of the central data model and the resulting import capabilities, since the effort for extending the user-interface of the CAD tool compared to the effort of writing a correct parser is much smaller.

4.2 Setting the Hypothesis

We define two evaluation hypotheses for our case study. The first evaluation hypothesis *H1* is that our model-driven approach for the exchange of product information between the OEM and his suppliers presented in Sect. 3 reduces the manual effort in deriving tool support for the information exchange. For the evaluation of *H1*, we define response variables measuring the amounts of entities contained in the input Base STEP AP including its extensions as well as response variables measuring the code size and the generation time of the parser output. That is, we determine the number of entities contained in the selected Base STEP AP plus the number of entities contained in the used STEP extensions (response variable *H1.inputSize*), the amount of lines of code generated in particular for the parser of the central data model (*H1.outputSize*), and the time needed for generating the different code fragments (*H1.outputTime*).

The second evaluation hypothesis *H2* is that our model-driven approach for the exchange of product information produces correct models and correct parser for existing STEP application protocols like STEP AP214 and STEP AP203, and that the parsers process their input files in reasonable time. For the evaluation of *H2*, we define a response variable for measuring the number of STEP files used as input for the parser (response variable *H2.inputSize*). That is, we determine the number of files that are correctly processed without an exception (*H2.outputSize*), and the time needed for processing each file (*H2.outputTime*). To draw conclusion of the processing time, we also determine the time needed to process the same files in SolidWorks (*H2.SWTime*). We used a typical office computer² for all test runs.

4.3 Validating the Hypothesis

For the validation of the first evaluation hypothesis *H1*, we executed the model-driven approach several times with different input configurations. First, we used the STEP AP214 as Base STEP AP and the STEP AP203 as Base STEP AP without any further STEP extensions. Furthermore, we used the STEP extensions of the running example in combination with the STEP AP214 and STEP AP203. Finally, we used the STEP AP203 as an extension in combination with STEP AP214 to draw conclusions about the scalability of the approach. The determination of the number of entities contained in the Base STEP AP as well as in the used STEP extensions, the needed generation time, and the lines of code for the parser yields the results as listed in Table 1.

For the validation of the second evaluation hypothesis *H2*, we used the STEP parser generated for STEP AP214 and STEP AP203. As input files, we used free available CAD examples (<http://www.steptools.com>). This leads to 20 files corresponding to the STEP AP214 and 44 files corresponding to the STEP AP203. The determination of the number of correctly parsed files and the needed parsing time yields the results as listed in Table 2.

² Intel Core i7-4600U @2.10 GHZ, 8 GB DDR3 1066 MHz, 500 GB HDD, Windows 7 Pro 64 bit, Java JDK8u66, Eclipse 4.5.

4.4 Analyzing the Results

The results for *H1* show two aspects. First, depending on the number of entities used for the description of the central data model, the resulting parser encompasses a huge amount of source code. Without a proper tool support, no software developer would be able to produce the parser in the relatively short time. For example, the execution of the model-driven approach uses overall 915 entities and generates 357775 lines of code within 12 min (cf. first row of Table 1). Although the generation takes some time to complete, this does not affect the applicability of the approach, since the generation has to be performed only once in the whole development project. Second, the model-driven approach scales with the number of entities used in the central data model. Thus, we consider *H1* as fulfilled. The results for *H2* show that the generated parser for the STEP AP214 and STEP AP203 is able to read original STEP files, thus, we conclude that our model-driven approach generates correct parsers. Furthermore, the comparison of *H2.outputTime* and *H2.SolidWorksTime* shows that our parser is not significantly slower than the processing of SolidWorks. Thus, we consider *H2* as fulfilled. Concluding the case study, the fulfilled hypotheses indicate that our proposed model-driven approach reduces the manual effort in deriving tool-support for the creation of a central data model and the corresponding import/export capabilities. This gives rise to the assumption that our approach is applicable and useful in practice for the OEM and his suppliers.

The most important threats to validity are as follows: First, we only considered one development example and thus cannot generalize the fulfillment of the statements. Nevertheless, the example represents a typical development project and thus we do not expect large deviations for other examples. Second, the amount of lines of code generated by the approach is only a superficial metric and, therefore, might not reflect the actual development effort. Especially for small extensions like the definition of power consumption, the conceptual complexity of our approach might exceed the effort for the manual extension and/or the manual exchange of this information via another communication channel. However, the manual extension has to be performed for each development project.

Table 1. Results of the analysis for *H1* [7].

Base STEP AP	# STEP extensions	H1.inputSize (# Entities)	H1.outputSize (LOC)	H1.outputTime (Gen. Time)
STEP AP214	-	915	357775	≈12 min
STEP AP203	-	254	108144	≈2 min
STEP AP214	2	917	358237	≈16 min
STEP AP203	2	256	108626	≈3 min
STEP AP214	1 (STEP AP203)	1169	465101	≈20 min

Table 2. Results of the analysis for *H2* [7].

Base STEP AP	H2.inputSize (# files)	H2.outputSize (# correct files)	H2.outputTime (parsing time)	H2.SWTime (parsing time)
STEP AP214	20	20	∅ 57 s	∅ 48 s
STEP AP203	44	44	∅ 46 s	∅ 39 s

5 Related Work

STEP provides a standardized mechanism for representing and exchanging product data, and is therefore, considered as a promising product modeling resource. As mentioned in Sect. 1, the STEP extension mechanism has been used in several applications to describe or analyze a certain aspect of a system and to exchange the corresponding product data. For example, [3] presents an object-oriented product model based on STEP AP224, which defines a standard set of machining features. The authors used their object-oriented product model to support a computer-aided process planning (CAPP) analysis. [4] present a product data exchange using a STEP-based assembly model for the concurrent integrated design and assembly planning. Concluding this paragraph, STEP is widely used in industry and academic to organize product data in a standardized representation. However, in contrast to our approach, most approaches are tailored to one particular use case and are not reusable or interoperable.

To overcome the inflexibility and interoperability, a generic product modeling system has been proposed in [26,27]. These two approaches belong to the most related approaches using STEP to provide a generic product modeling system. However, in contrast to our approach, they do not use model-driven techniques to realize their approach, and, thus, a lot of manual effort has been done for their practical realization.

Other work has focused on applying model-driven development techniques to product data modeling in the design of mechanical systems for the purpose of collaboration and interoperability. For example, [28] present a model-driven architecture for bringing together various product data into a model-driven engineering environment. The engineering environment is used to transform, share, and export the product data enabling the collaboration between different departments and companies involved in the design of mechanical products. [29] build a bridge between STEP/EXPRESS and the Eclipse Modeling Framework. The bridge is used to transform models based on the Industry Foundation Classes (IFC), a standardized modeling language, into a format suitable for a particular CAD tool. Beyond the pure storing and sharing of STEP-based product data in a model-driven environment based on the Eclipse Modeling Framework, our approach enables the systematic extension of the STEP standard to provide the exchange of additional technical product data. Furthermore, the OMG has published a standard for a reference meta-model for the EXPRESS information modeling language [30]. This meta-model has been developed in the so-called

Mexico project. However, the standard only focuses on the meta-model for the EXPRESS information modeling language, and does not describe how existing STEP application protocols can be transformed to an instance of the reference model.

Finally, Yildiz et al. [31] present ongoing work on a model-driven approach for the specification of product information in the context of PDM. As in our work, the authors state that the initial implementation of a PDM tool, usually does not cover all information needed by the user and that the required extensions to a PDM tool are extensive to implement. Thus, they propose a model-driven approach enabling companies to specify their own business concepts for a PDM tool, resulting in tool extensions to cover the additional information. Our approach and the approach of Yildiz et al. mainly differ in their aim. We use project-specific STEP extensions for the data exchange of product information, whereas Yildiz et al. focus on the extension of the storage capabilities of PDM tools but do not consider data exchange. However, the OEM typically applies a PDM tool with a plain artifact storage mechanism nowadays, as sketched in the introduction. Thus, a complementary combination of both approaches would lead to a more holistic tool chain, as we point out in the future work.

6 Conclusion and Future Work

In previous work [7], we presented a model-driven approach for the flexible specification of STEP application protocol extensions. Our model-driven approach exploited the STEP extension mechanism to enable the specification and tailoring of STEP application protocols to project-specific needs. Furthermore, our approach included the automatic derivation of the required tool capability extensions for both the OEM and the suppliers. On the one hand, we derived a central data model as well as a STEP parser for the import and interpretation tool capability extension on the OEM side. On the other hand, we derived a plugin for the CAD tool SolidWorks for the specification and the export tool capability extension on the supplier side. Furthermore, our approach supported reusing once specified STEP application protocol extensions.

In this paper, we integrated the model-driven STEP application protocol extensions with a feature modeling approach [9] that particularly considers the constraints on the geometrical information exchanged via STEP. Beyond the STEP parser and the CAD tool export plugin, we derive a feature model plugin based on the central data model that is generated from the STEP application protocol extensions. Thereby, the feature model gains access to the logical as well as geometrical information stored in the central data model. In addition to the specification of logical constraints and validation well-known from feature models, we introduce a geometrical constraint language that enables to specify and validate assembly constraints on the geometrical information in the central data model.

Our model-driven approach significantly reduces the manual effort that had to be spent on the whole tool chain otherwise. Thereby, we enable the utilization

of STEP application protocol extensions for project-specific needs. Moreover, the possibility of reusing extensions reduces the effort on the actual specification of the particular STEP application protocol extensions if an extension was conceived in prior projects. The generality of the approach enables to handle other parts of the STEP standard beyond the one that we exemplarily extended in this paper. The integration with the feature modeling approach considering logical as well as geometrical constraints further reduces redundancies between different engineering tools. In summary, the combination of both approaches enable OEMs to orchestrate their overall supply and development processes.

The future work is twofold. First, we want to improve the creation of STEP application protocols to support the remaining EXPRESS elements like *where-clauses* and *rules* in the resulting Ecore-based meta-model by means of Object Constraint Language (OCL) [32] expressions. Second, we want to apply the feature model together with a geometric modeling kernel [24], which is used within conventional CAD tools. Thereby, we can achieve the same algorithmic power for the verification of geometrical layouts w.r.t. assembly constraints as known from CAD tools. This will enable a potential OEM customer to configure a product variant and virtually layout it according to the assembly constraints in an e-commerce-system.

Acknowledgment. This research is partially funded by the German Federal Ministry of Education and Research (BMBF) under the grant ZIM and is managed by the AiF Projekt GmbH. Furthermore, this research is partially funded by the German Federal Ministry of Education and Research (BMBF) within the Leading-Edge Cluster “Intelligent Technical Systems OstWestfalenLippe” (it’s OWL) and is managed by the Project Management Agency Karlsruhe (PTKA).

References

1. Vogel-Heuser, B., Legat, C., Folmer, J., Feldmann, S.: Researching evolution in industrial plant automation: Scenarios and documentation of the pick and place unit. Technical report, Institute of Automation and Information Systems, Technische Universität München (2014)
2. ISO: ISO 10303-1:1994: Industrial automation systems and integration - Product data representation and exchange - Part 1: Overview and fundamental principles (1994)
3. Usher, J.M.: A STEP-based object-oriented product model for process planning. *Comput. Ind. Eng.* **31**, 185–188 (1996)
4. Zha, X., Du, H.: A PDES/STEP-based model and system for concurrent integrated design and assembly planning. *Comput.-Aided Des.* **34**, 1087–1110 (2002)
5. Industrie 4.0 Working Group: Recommendations for implementing the strategic initiative INDUSTRIE 4.0. Final report (2013)
6. Min, H.: Electronic data interchange in supply chain management. In: Swamidass, P.M. (ed.) *Encyclopedia of Production and Manufacturing Management*, pp. 177–183. Springer, Boston (2000). https://doi.org/10.1007/1-4020-0612-8_284

7. Koch, T., Holtmann, J., Lindemann, T.: Flexible specification of STEP application protocol extensions and automatic derivation of tool capabilities. In: Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD) (2017)
8. Kang, K.C., Lee, K., Lee, J.: Feature-oriented product line software engineering: principles and guidelines. In: Domain Oriented Systems Development - Practices And Perspectives, pp. 29–46. Taylor & Francis (2003)
9. Koch, T., Holtmann, J., Schubert, D., Lindemann, T.: Towards feature-based product line engineering of technical systems. In: 3rd International Conference on System-Integrated Intelligence: New Challenges for Product and Production Engineering (SysInt), pp. 447–454. Elsevier (2016)
10. Kramer, T., Xu, X.: STEP in a nutshell. In: Xu, X., Nee, A. (eds.) Advanced Design and Manufacturing Based on STEP, pp. 1–22. Springer, London (2009). https://doi.org/10.1007/978-1-84882-739-4_1
11. ISO: ISO 10303–11:2004: Industrial automation systems and integration - Product data representation and exchange - Part 11: Description methods: The EXPRESS language reference manual (2004)
12. ISO: ISO 10303–21:2002: Industrial automation systems and integration - Product data representation and exchange - Part 21: Implementation methods: Clear text encoding of the exchange structure (2002)
13. ISO: ISO 10303–28:2007: Industrial automation systems and integration - Product data representation and exchange - Part 28: Implementation methods: XML representations of EXPRESS schemas and data, using XML schemas (2007)
14. OMG: Meta Object Facility (MOF) Core Specification: Version 2.5.1 (2016)
15. OMG: Business Process Model and Notation (BPMN): Version 2.0.2 (2013)
16. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley, Boston (2008)
17. Eysholdt, M., Behrens, H.: Xtext: implement your language faster than the quick and dirty way. In: OOPSLA 2010, pp. 307–309. ACM (2010)
18. Benavides, D., Trinidad, P., Ruiz-Cortés, A.: Automated reasoning on feature models. In: Pastor, O., Falcão e Cunha, J. (eds.) CAiSE 2005. LNCS, vol. 3520, pp. 491–503. Springer, Heidelberg (2005). https://doi.org/10.1007/11431855_34
19. Czarnecki, K., Eisenecker, U.: Generative Programming Methods, Tools, Applications. Addison-Wesley, Boston (2000)
20. Vierhauser, M., Grünbacher, P., Heider, W., Holl, G., Lettner, D.: Applying a consistency checking framework for heterogeneous models and artifacts in industrial product lines. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds.) MODELS 2012. LNCS, vol. 7590, pp. 531–545. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33666-9_34
21. Benavides, D., Segura, S., Trinidad, P., Ruiz-Cortés, A.: Using java CSP solvers in the automated analyses of feature models. In: Lämmel, R., Saraiva, J., Visser, J. (eds.) GTTSE 2005. LNCS, vol. 4143, pp. 399–408. Springer, Heidelberg (2006). https://doi.org/10.1007/11877028_16
22. Mannion, M.: Using first-order logic for product line model validation. In: Chastek, G.J. (ed.) SPLC 2002. LNCS, vol. 2379, pp. 176–187. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45652-X_11
23. Anderl, R., Mendgen, R.: Modelling with constraints: theoretical foundation and application. *Comput.-Aided Des.* **28**, 155–168 (1996)
24. Shah, J.J., Mantyla, M.: Parametric and Feature Based CAD/CAM: Concepts, Techniques, and Applications, 1st edn. Wiley, New York (1995)

25. Kitchenham, B., Pickard, L., Pfleeger, S.L.: Case studies for method and tool evaluation. *IEEE Softw.* **12**, 52–62 (1995)
26. Gu, P., Chan, K.: Product modelling using STEP. *Comput.-Aided Des.* **27**, 163–179 (1995)
27. Xie, S.Q., Chen, W.L.: A generic product modelling framework for rapid development of customised products. In: Xu, X., Nee, A. (eds.) *Advanced Design and Manufacturing Based on STEP*, pp. 331–352. Springer, London (2009). https://doi.org/10.1007/978-1-84882-739-4_15
28. Iraqi Houssaini, M., Kleiner, M., Roucoules, L.: Tools interoperability in engineering design using model-based engineering. In: *ASME 2012*, pp. 615–623 (2012)
29. Steel, J., Duddy, K., Drogemuller, R.: A transformation workbench for building information models. In: Cabot, J., Visser, E. (eds.) *ICMT 2011*. LNCS, vol. 6707, pp. 93–107. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21732-6_7
30. OMG: Reference Metamodel for the EXPRESS Information Modeling Language (EXPRESS): Version 1.1 (2015)
31. Yildiz, O., et al.: MDA based tool for PLM' models building and evolving. In: Grabot, B., Vallespir, B., Gomes, S., Bouras, A., Kiritsis, D. (eds.) *APMS 2014*. IAICT, vol. 438, pp. 315–322. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44739-0_39
32. OMG: Object Constraint Language (OCL): Version 2.4 (2014)



A Model Driven Engineering Approach for Heterogeneous Model Composition

Fazle Rabbi^{1,2(✉)}, Yngve Lamo¹, and Lars Michael Kristensen¹

¹ Western Norway University of Applied Sciences, Bergen, Norway
{fra,yla,lmkr}@hvl.no

² University of Oslo, Oslo, Norway

Abstract. Diagrammatic modeling plays an important role in model driven software engineering as it can be used to define domain-specific modeling languages. During the modeling phase, software designers encode domain knowledge into models. Complex models of software systems often consist of heterogeneous models representing various aspects of a system such as structure, behavior, security, and resources. In this paper, we present a formal modeling approach for the composition of heterogeneous models. We apply the technique for modeling the optimization of distributed resources using game theory.

Keywords: Model composition · Diagrammatic modeling
Epistemic game theory · Model transformation · Optimization
Distributed systems

1 Introduction

Today's software systems are more complex and involves interaction among several devices and applications using heterogeneous platforms. For modeling software systems we need to consider various aspects of systems. In order to deal with the complexity of software systems, software engineers usually separate the aspects of software systems which leads to different models. Decomposing a system based on aspects (such as structure and behavior) facilitate abstraction and provides flexibility in updating the decomposed sub-modules. However, to reason about the system as an integrated whole we need to coordinate models that are representing different aspects of a system and represent distributed systems. Requirements for integrating heterogeneous distributed systems are increasing with the rapid technological advancements. Therefore, model composition is becoming a key issue in requirements analysis and design of complex systems. This requires formalization to understand and develop satisfactory solutions. The study of integrating heterogeneous systems is a complex process consisting of information and expert knowledge management, modeling, simulation, and decision making support [1]. It becomes a challenging task to compose different software models in a coherent way as it requires combining models having different syntax.

The modeling concerns for the representation of complex systems require techniques for handling composition of heterogeneous artefacts. Efforts have been made in [2,3] to specify the integration of heterogeneous systems by the integration of heterogeneous modeling languages. They categorized the need for language integration into three groups: language aggregation, language embedding, and language inheritance. An algebraic approach for integrating languages have been studied in [4,5] where the authors used a Common Algebraic Specification Language (CASL) for the specification and development of modular software systems. However, these approaches are based on textual languages. In this paper, we focus on diagrammatic approaches for modeling complex software systems and propose a formal modeling approach to compose heterogeneous models in a coherent way. We demonstrate its application for the optimization of distributed resources. The formal modeling approach is presented by means of composition schema which provide reusable patterns for the structural composition of software models. It can be used for modeling in the small in a sense that it can deal with modeling artefacts such as models, metamodel elements and relations between them; it can also be used for modeling in the large in a sense that we can coordinate heterogeneous modeling systems with composition schema.

The formal approach for model composition presented in this paper shows how distributed systems may be coordinated to optimize their resources respecting a distributed set of local constraints. Merging of models is an important concern of analyzing distributed systems as there are situations where loosely coupled distributed systems have to deal with a set of global constraints. For instance, patients scheduled appointment time for getting a healthcare service should not conflict with scheduled appointments for other healthcare services. This paper is an extended version of a previously published article [6] where we presented an example from the healthcare domain and proposed to use epistemic game theory for optimizing the use of distributed resources and add resource constraints to game theory. In this paper, we present a formal treatment of modeling heterogeneous systems and distributed resource constraints in a coherent way.

The paper is organized as follows: In Sect. 2, we present a formal modeling approach for composing software models by means of reusable composition schema. Section 3 presents the construction of small software models with constraints. In Sect. 4 we present how the composition schema can be used for merging metamodels and models. In Sect. 5, we present an application of the proposed model composition for the optimization of distributed healthcare resources. Section 6 presents related works and Sect. 7 concludes the paper.

2 Composition Schema

The formalization of composition schema is discussed as an approach for constructing complex software models. In this formalization, we integrate several modeling artefacts to define software structures. Composition schema can be used for constructing the structure of individual software models as well as

correlating complex structures of distributed software models. As a motivation consider the following model integration problem. Consider two distributed systems for an orthopedic department and a radiology department in a hospital. To model these distributed systems we need to define different domain concepts and constraints. These two systems share some resources and we need to model the interdependency of the system with an integrated system model. There are some overlapping of concepts and constraints in these two systems and we need to specify inter-model constraints representing the global constraints governing the overall system. To model the resource allocation of the distributed systems and their optimization we wish to use a game theory model. Therefore we need to model the game theoretic perspective of the distributed resource allocations and link them with the distributed software models. To cope with this situation, we require a modeling framework that allows heterogeneous model integration and supports both modeling small and large distributed systems. In this section and subsequent sections we introduce a modeling formalism with so-called composition schema that supports such model composition requirements. Inter-model constraints of heterogeneous systems can be specified using composition schema and it provides a building block for integrating software models in a coherent way. Composition schema may have numerous applications in software engineering as various applications can be modeled by combining different combination of models and/or modeling artefacts.

2.1 Formalization

We adapt the formalization of the Diagram Predicate Framework (DPF) [7,8] by generalizing the components of diagrammatic specifications. DPF uses the concept of diagram predicates to specify constraints on a diagrammatic specification. A DPF predicate signature $\Sigma = (P^\Sigma, \alpha^\Sigma)$ consists of a collection of predicate symbols and a mapping of the predicate symbols to their arity. In DPF, a diagrammatic specification $\mathfrak{S} = (S, C^S : \Sigma)$ is given by a graph S and a set of atomic constraints $C^S : \Sigma$. The shape graph S is given by a collection of nodes, edges, source, and target maps. The atomic constraints are added to the graph S by means of graph homomorphism from the shape of a predicate to S , $\delta : \alpha^\Sigma(p) \rightarrow S$. In this paper, we generalize the concepts of diagrammatic specification and propose to use composition schema for modeling heterogeneous systems. Composition schema is formalized as a set of objects, relationship among the objects, and a set of structural constraints. The objects may be of heterogeneous classes and may consist of complex structures with internal objects; the relationships may consist of relations among the objects and/or internal objects. The objects and relationships constitute the shape of a composition schema. Similar to DPF, structural constraints are added to the shape of a composition schema by a structure preserving map from the arity of a predicate.

Definition 1 (Composition Schema). *Given a predicate signature $\Sigma = (P^\Sigma, \alpha^\Sigma)$, a composition schema \mathfrak{C} consists of*

- a set of objects Ob ,
- a set of morphisms Mor between the objects or its internal components,
- a set SC of atomic constraints on the shape of \mathfrak{C} with $p \in P^\Sigma$

where the objects may be of heterogeneous classes, each morphism f has a source object A and a target object B , written as $f: A \rightarrow B$.

A *Composition Schema* describes the structure of a model which allow us to compose modeling elements through substitution. Complex structures may be formulated with composition schemata by specifying structural constraints. In order to be a valid composition, the structural constraints must be satisfied.

We obtain a modeling language $\mathcal{M}_{\mathfrak{C}}$ for model composition by means of a composition schema. The meaning of a composition schema is given by explaining it in a semantic domain. Any model $m \in \mathcal{M}_{\mathfrak{C}}$ is mapped to the systems which obey the constraints that the model imposes. Similar to [9], we use a Set-valued semantics for the language $\mathcal{M}_{\mathfrak{C}}$. Given a set of all systems \mathcal{S} , the semantic domain is the power set $\mathcal{P}(\mathcal{S})$ and each instance $m \in \mathcal{M}_{\mathfrak{C}}$ will be mapped by a semantic mapping, \mathbf{sm} to the largest set of systems which fulfill the constraints:

$$\mathbf{sm} : \mathcal{M}_{\mathfrak{C}} \rightarrow \mathcal{P}(\mathcal{S})$$

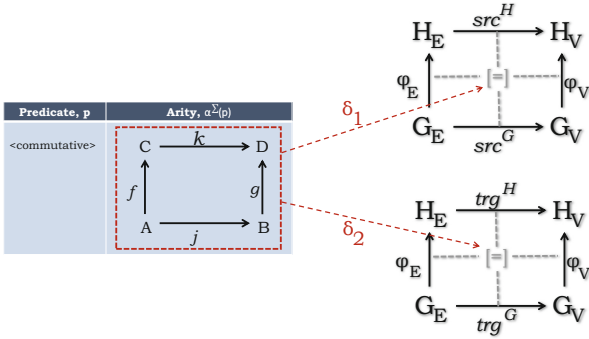
Models are usually underspecified in the early phases and therefore we need to define models with abstract information. Composition schemata are often defined with underspecification such that there is usually not a single system that realizes a model, but a larger set of realizations is allowed. Further refinement of a composition schema is achieved by replacing abstract information with concrete modeling elements. A model m_2 refines another model m_1 if $\mathbf{sm}(m_2) \subseteq \mathbf{sm}(m_1)$; thus if we add more specific information into model m_2 , it further constraints the resulting set of systems. The loose semantic approach is useful in this regard as it can be used for developing a library of schemata for model composition.

In this paper, we first show how the composition schema may be used to construct small models and later on we demonstrate how it can be used to merge heterogeneous models. In this section and in the following sections, we consistently use similar structures in the form of composition schemata. This makes it simpler to construct a new composition schema based on an existing one. Here we present a construction of typing composition for models representing the categorization of objects into classes or types which is one of the typical situations encountered in software modeling.

Definition 2 (Typing Schema). *Given a predicate signature with a predicate $\langle commutative \rangle$, a typing schema \mathfrak{C}_{typing} is specified by*

- a set of objects $Ob = \{G, H\}$ where $G = \{G_V, G_E, src^G : G_E \rightarrow G_V, trg^G : G_E \rightarrow G_V\}$ and $H = \{H_V, H_E, src^H : H_E \rightarrow H_V, trg^H : H_E \rightarrow H_V\}$ are graphs,
- a set of morphisms $Mor = \{ \varphi : G \rightarrow H \}$ where φ is given by two single valued component functions $\varphi_V : G_V \rightarrow H_V$ and $\varphi_E : G_E \rightarrow H_E$.

- a set of constraints $SC = \{ \delta_1, \delta_2 \}$ added into the shape of \mathfrak{C}_{typing} with the $\langle commutative \rangle$ predicate. The constraints are used to specify that the following two diagrams on the right are commutative:



Definition 2 defines an algebraic structure of typing schema where G, H are variables. Only directed graph structures can be assigned to these variables. The semantic of \mathfrak{C}_{typing} consists of systems with typed graphs. Given two graphs G_1 and H_1 and a map $\varphi_1 : G_1 \rightarrow H_1$, there exists a typing composition between G_1 and H_1 (i.e., G_1 is typed by H_1) if the structural constraints of \mathfrak{C}_{typing} are satisfied when G is assigned to G_1 , H is assigned to H_1 and φ is assigned to φ_1 . This is represented as $\mathfrak{C}_{typing_1} = \mathfrak{C}_{typing}(Ob = \{G := G_1, H := H_1\}, Mor = \{\varphi := \varphi_1\})$ and $\mathfrak{C}_{typing_1} \models SC(\mathfrak{C}_{typing})$. Note that \mathfrak{C}_{typing_1} is representing a refined model of \mathfrak{C}_{typing} . In the rest of this article, we use \mathfrak{C}_{A_r} to represent a refined model of \mathfrak{C}_A where r is any name.

Since a composition schema can be used inside another composition schema, it can be complicated to address the elements of a nested composition schema. To resolve this issue, we adopt the *dot notation* from object oriented programming languages to access elements of a composition schema. For instance, we can access the graph H_1 of the composition schema \mathfrak{C}_{typing_1} by using the notation $\mathfrak{C}_{typing_1}.Ob[H_1]$. Similarly, to access the map φ_1 we use the notation $\mathfrak{C}_{typing_1}.Mor[\varphi_1]$.

3 Modeling in the Small

We now show how composition schema can be used to specify a model with metamodels and constraints. For metamodel specification of a system, we adapt the formalization of DPF and present composition schema to represent DPF modeling concepts. DPF is suitable for metamodeling as in DPF, models at any level are formalized as diagrammatic specifications. In DPF, a metamodel specification consists of concepts provided in a type graph and a set of constraints specified by means of graph morphisms: Similarly, a model consists of concepts defined as a graph and the typing is specified by graph morphisms. In DPF metamodeling formalization, a model must be typed by its metamodel and must satisfy all the constraints specified in its metamodel in order to conform to its metamodel. Semantics of predicates can be given in different ways such as

the fibred manner [7] or using graph constraints [10]. The purpose of using composition schema is to provide a coherent pattern for metamodeling.

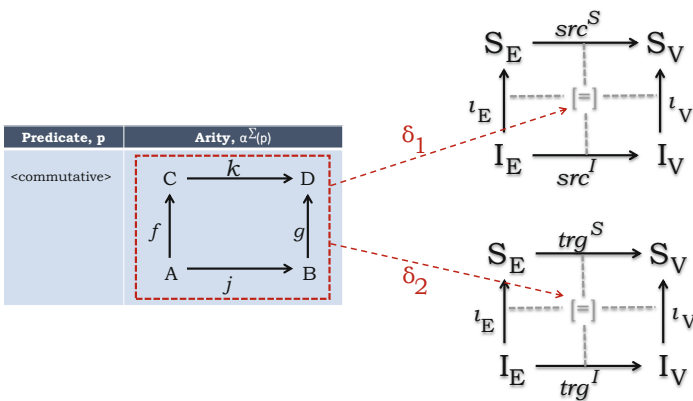
Definition 3 (Metamodel). *Given a predicate signature $\Sigma = (P^\Sigma, \alpha^\Sigma)$, a composition schema for metamodel $\mathfrak{C}_{MM} = (Ob, Mor, SC)$ is specified by*

- a set of objects Ob consisting of an underlying graph $S = \{S_V, S_E, src^S, trg^S\}$, where S_V, S_E are sets of vertices and edges; $src^S : S_E \rightarrow S_V$ and $trg^S : S_E \rightarrow S_V$ are the source and target maps of the edges,
- an empty set of morphisms Mor ,
- a set of constraints SC added into S from the predicates by graph homomorphisms.

In DPF, an instance of a metamodel is represented by a graph and a typing morphism. Similar to DPF instances, we define a model composition schema by composing a graph with a metamodel schema.

Definition 4 (Model). *Given a predicate signature with a predicate $\langle commutative \rangle$, a composition schema for model $\mathfrak{C}_M = (Ob, Mor, SC)$ is specified by*

- a set of objects Ob consisting of a graph $I = \{I_V, I_E, src^I, trg^I\}$ and a metamodel schema \mathfrak{C}_{MM} , where I_V, I_E are sets of vertices and edges; $src^I : I_E \rightarrow I_V$ and $trg^I : I_E \rightarrow I_V$ are the source and target maps of the edges,
- a set of morphisms $Mor = \{\iota : I \rightarrow S\}$ where S is the underlying graph of \mathfrak{C}_{MM} (i.e., $S = \mathfrak{C}_{MM}.Ob[S]$) and ι is given by two single valued component functions $\iota_V : I_V \rightarrow S_V$ and $\iota_E : I_E \rightarrow S_E$.
- a set of constraints $SC = \{\delta_1, \delta_2\}$ added into the shape of \mathfrak{C}_M with the $\langle commutative \rangle$ predicate. The constraints are used to specify that the following two diagrams on the right are commutative:



A valid model composition schema must satisfy all the constraints specified in its metamodel. To check that a constraint specified in a metamodel composition schema is satisfied in a given model, it is sufficient to inspect only the part

of the metamodel which is affected by the constraint. This checking is performed by a pullback construction on $\alpha^\Sigma(p) \xrightarrow{\delta} \mathfrak{C}_{MM} \xleftarrow{\mathfrak{C}_M.Mor[\iota]} \mathfrak{C}_M.Ob[I]$ for each constraint specified in the metamodel.

Example. An example of a model $\mathfrak{C}_{M_1} = \mathfrak{C}_M(Ob = \{I := I_1, \mathfrak{C}_{MM} := \mathfrak{C}_{MM_1}\}, Mor = \{\iota := \iota_1\})$ is shown in Fig. 1 consisting of a graph I_1 , a metamodel \mathfrak{C}_{MM_1} and a typing morphism ι_1 . The graph I_1 of \mathfrak{C}_{M_1} is typed by the underlying graph S_1 of \mathfrak{C}_{MM_1} . The model represents the following situation: *Bryan is a caregiver, who is involved in Ward#10; John is a patient who is admitted to Ward#10; and Bryan has data-access to John's records.* The underlying graph S_1 is constrained by the predicate $\langle composite \rangle$ by a graph morphism δ . By constraining the graph S_1 with the predicate $\langle composite \rangle$, we are essentially specifying the following constraint: *'Caregivers involved in a ward must have data-access to the patients admitted into the same ward'*. The model \mathfrak{C}_{M_1} is satisfying the constraint specified in the metamodel composition as *Bryan has data-access to John's records*; therefore, \mathfrak{C}_{M_1} is a valid model i.e., $\mathfrak{C}_{M_1} \models SC(\mathfrak{C}_M)$.

3.1 Category of Model Compositions

To define a category of model composition schemas we need to define the morphism between metamodel composition schemas. Following [7], we provide the following definition of metamodel morphism.

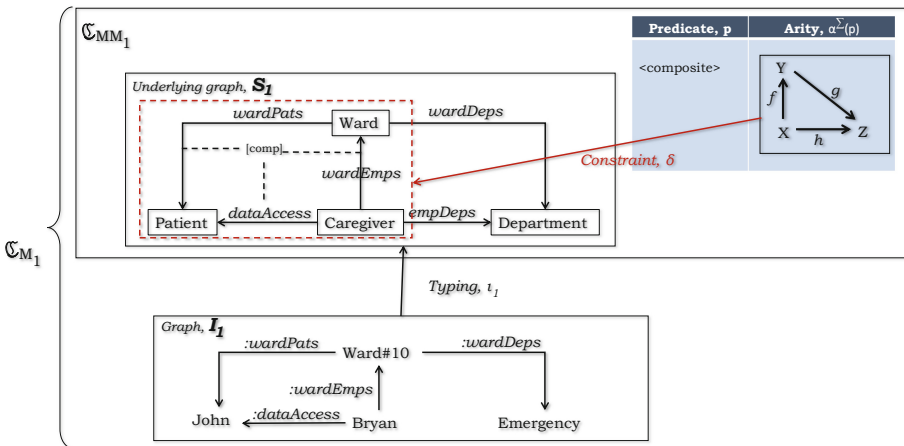
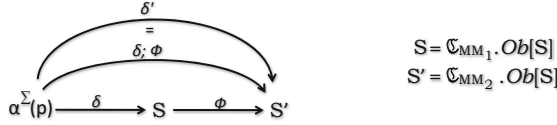


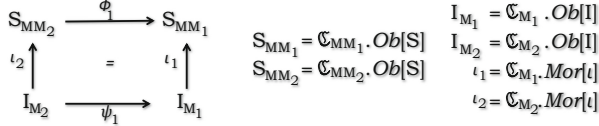
Fig. 1. Example of a model.

Definition 5 (Metamodel Morphism). Given two metamodel composition schemata \mathfrak{C}_{MM_1} and \mathfrak{C}_{MM_2} , a metamodel morphism $\phi : \mathfrak{C}_{MM_1} \rightarrow \mathfrak{C}_{MM_2}$ is a graph homomorphism $\phi : \mathfrak{C}_{MM_1} \rightarrow \mathfrak{C}_{MM_2}$ such that for each constraint $\delta \in \mathfrak{C}_{MM_1}.SC$ there exists a constraint $\delta' \in \mathfrak{C}_{MM_2}.SC$ where $\delta' = \delta; \alpha$.

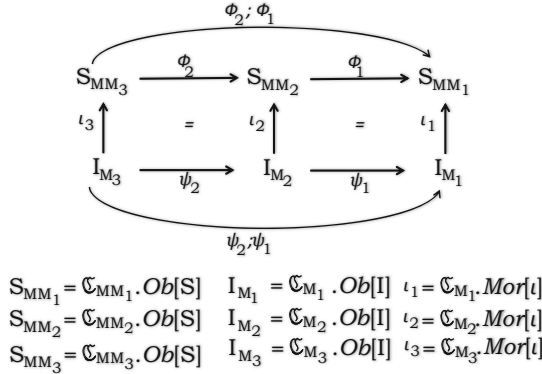


Definition 6 (Category of Model Compositions). *The category of model compositions is given by:*

- a set of objects *Ob* consisting of models,
- morphisms: any morphism $g : \mathfrak{C}_{M_2} \rightarrow \mathfrak{C}_{M_1}$ is given by a pair of morphisms (ϕ_1, ψ_1) from model \mathfrak{C}_{M_2} to \mathfrak{C}_{M_1} such that there exists a metamodel morphism from \mathfrak{C}_{MM_2} to \mathfrak{C}_{MM_1} and the following diagram commutes:



- composition: the composition $f; g : \mathfrak{C}_{M_3} \rightarrow \mathfrak{C}_{M_1}$ of two morphisms $f : \mathfrak{C}_{M_3} \rightarrow \mathfrak{C}_{M_2}$ and $g : \mathfrak{C}_{M_2} \rightarrow \mathfrak{C}_{M_1}$ is given by the composition of graph morphisms illustrated below:



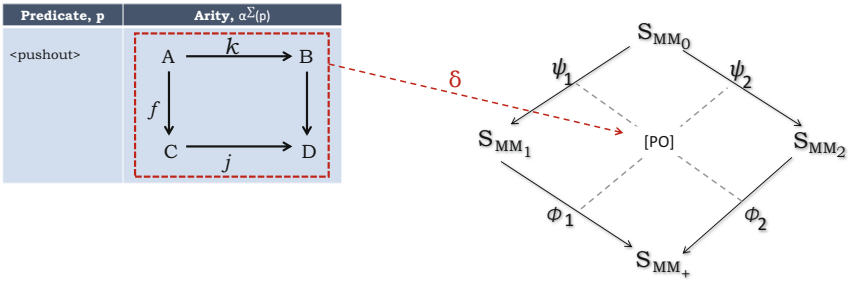
- identity: for any object \mathfrak{C}_{M_1} , the identity morphism $id : \mathfrak{C}_{M_1} \rightarrow \mathfrak{C}_{M_1}$ is given by the identity graph morphism.

4 Modeling in the Large

We now address how composition schema can be used for modeling in the large, including merging metamodels and models. There are various ways of merging metamodels [11–14]. Diskin et al. presented an approach of merging metamodels by making metamodels explicit and by introducing a correspondence span between component metamodels [15, 16]. Following [15] we present a metamodel merging composition that includes an overlapping model specifying the overlap between component models.

Definition 7 (Metamodel Merging). Given a predicate signature with a predicate $\langle \text{pushout} \rangle$, a composition schema for metamodel merging $\mathfrak{C}_{\text{Meta-Merge}} = (\text{Ob}, \text{Mor}, \text{SC})$ is specified by

- a set of objects Ob consisting of two component metamodels $(\mathfrak{C}_{\text{MM}_1}, \mathfrak{C}_{\text{MM}_2})$, an overlap metamodel $\mathfrak{C}_{\text{MM}_0}$ and a merged metamodel $(\mathfrak{C}_{\text{MM}_+})$,
- a set of morphisms Mor consists of the following metamodel morphisms where $S_{\text{MM}_0} = \mathfrak{C}_{\text{MM}_0}.\text{Ob}[S]$, $S_{\text{MM}_1} = \mathfrak{C}_{\text{MM}_1}.\text{Ob}[S]$, $S_{\text{MM}_2} = \mathfrak{C}_{\text{MM}_2}.\text{Ob}[S]$, $S_{\text{MM}_+} = \mathfrak{C}_{\text{MM}_+}.\text{Ob}[S]$ are the underlying graphs of metamodels
 - $\psi_1 : S_{\text{MM}_0} \rightarrow S_{\text{MM}_1}$,
 - $\psi_2 : S_{\text{MM}_0} \rightarrow S_{\text{MM}_2}$,
 - $\varphi_1 : S_{\text{MM}_1} \rightarrow S_{\text{MM}_+}$,
 - $\varphi_2 : S_{\text{MM}_2} \rightarrow S_{\text{MM}_+}$,
- a set of constraints $\text{SC} = \{\delta\}$ added into the shape of $\mathfrak{C}_{\text{Meta-Merge}}$ with the $\langle \text{pushout} \rangle$ predicate. The constraint specifies that the underlying graph of $\mathfrak{C}_{\text{MM}_+}$ and morphisms φ_1, φ_2 are obtained by the pushout of ψ_1 and ψ_2 as illustrated below:



Inter-metamodel constraints are specified in the merged metamodel. Metamodel morphisms between $\mathfrak{C}_{\text{MM}_1}$ and $\mathfrak{C}_{\text{MM}_+}$, $\mathfrak{C}_{\text{MM}_2}$ and $\mathfrak{C}_{\text{MM}_+}$ ensure that the constraints specified in the component metamodels are preserved into the merged metamodel specification. However, it is possible that the constraints when preserved in a merged metamodel may conflict with each other or may become redundant. Some research has been done in this direction and there exists algorithms to detect these problems automatically. The reader interested in specification analysis may wish to consult [10].

Example. An example of a merged metamodel is shown in Fig. 2 consisting of two metamodels $\mathfrak{C}_{\text{MM}_1}$ and $\mathfrak{C}_{\text{MM}_2}$ that we wish to merge. $\mathfrak{C}_{\text{MM}_1}$ and $\mathfrak{C}_{\text{MM}_2}$ are representing entity models of a radiology department and an orthopedic department of a hospital. $\mathfrak{C}_{\text{MM}_0}$ is an overlap metamodel representing the overlap of concepts from $\mathfrak{C}_{\text{MM}_1}$ and $\mathfrak{C}_{\text{MM}_2}$. The merged metamodel is shown in $\mathfrak{C}_{\text{MM}_+}$. Table 1 shows the predicates of a signature Σ' used by these metamodel compositions. Below is a list of constraints specified in $\mathfrak{C}_{\text{MM}_1}$ and $\mathfrak{C}_{\text{MM}_2}$:

- **C1.** A patient must have exactly one birthdate in an instance of $\mathfrak{C}_{\text{MM}_1}$ (specified by $\langle \text{mult}(1, 1) \rangle$)

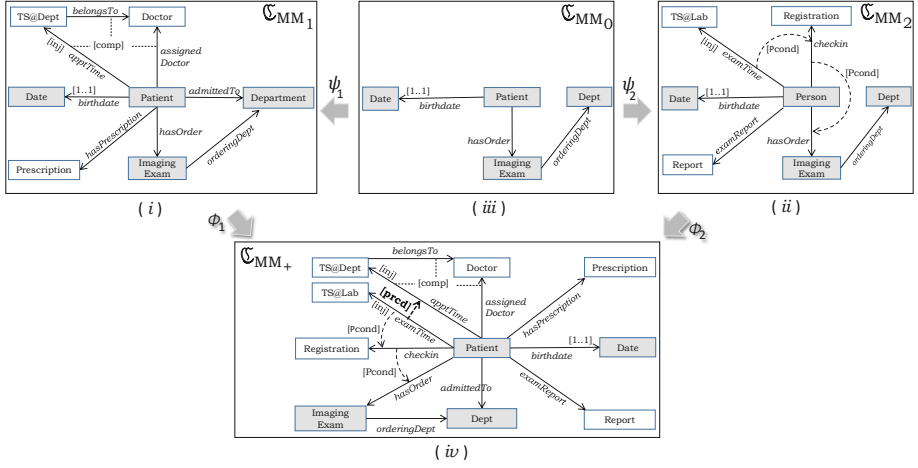


Fig. 2. Example of a merged metamodel composition [6].

Table 1. Predicates of a signature, Σ' [6].

p	Arity $\alpha_{\Sigma'}(p)$	Visualization	Semantic interpretation
$\langle \text{mult}(1,1) \rangle$	$1 \xrightarrow{f} 2$		f must have 1 instance for each instance of X
$\langle \text{pre-condition} \rangle$	$1 \xrightarrow{f} 2$ $1 \xrightarrow{g} 3$		For each instance of f there exists an instance of g with the same source node
$\langle \text{composite} \rangle$	$1 \xrightarrow{f} 2$ $1 \xrightarrow{g} 3$ $1 \xrightarrow{h} 3$		For each composition of instances f, g , there exists an instance of h
$\langle \text{precede} \rangle$	$1 \xrightarrow{f} 2$ $1 \xrightarrow{g} 3$		If there are instances of f and g with the same source node, then there exists an ordering of values between instances of Y and Z
$\langle \text{injective} \rangle$	$1 \xrightarrow{f} 2$		Instances of f never maps distinct elements of its domain to the same element of its codomain

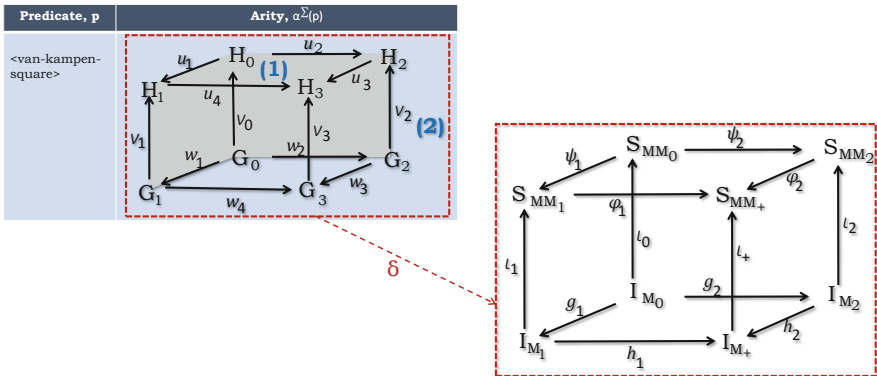
- **C2.** A person must have exactly one birthdate in an instance of \mathcal{C}_{MM_2} (specified by $\langle \text{mult}(1,1) \rangle$)
- **C3.** An appointment time-slot (i.e., $TS@Dept$) allocated to a patient in an instance of \mathcal{C}_{MM_1} must belong to that patients assigned doctor (specified by $\langle \text{composite} \rangle$)

- **C4.** A person can only check-in for an examination in an instance of \mathfrak{C}_{MM_2} if the person has an imaging order (specified by $\langle \text{pre-condition} \rangle$)
- **C5.** Only registered persons are allocated with examination time-slots (i.e., $TS@Lab$) in an instance of \mathfrak{C}_{MM_2} (specified by $\langle \text{pre-condition} \rangle$)
- **C6.** An appointment time-slot in an instance of \mathfrak{C}_{MM_1} must not be allocated to more than one patient (specified by $\langle \text{injective} \rangle$)
- **C7.** An exam time-slot in an instance of \mathfrak{C}_{MM_2} must not be allocated to more than one person (specified by $\langle \text{injective} \rangle$)

Moreover, an inter-metamodel constraint is specified in the merged meta-model. The inter-metamodel constraint (**C8**) specifies that ‘orthopedic patients time-slot at the radiology department must be preceded by the time-slot at the orthopedic department’. The $\langle \text{precede} \rangle$ predicate is used to specify this constraint. The overlap metamodel declares the common concepts of both \mathfrak{C}_{MM_1} and \mathfrak{C}_{MM_2} and the legs of the span $\mathfrak{C}_{MM_1} \xleftarrow{\psi_1} \mathfrak{C}_{MM_0} \xrightarrow{\psi_2} \mathfrak{C}_{MM_2}$ are metamodel morphisms.

Definition 8 (Model Merging). Given a predicate signature with a predicate $\langle \text{van-kampen-square} \rangle$, a model merging schema $\mathfrak{C}_{\text{Model-Merge}} = (Ob, Mor, SC)$ is specified by

- a set of objects Ob consisting of a metamodel merging schema ($\mathfrak{C}_{\text{Meta-Merge}}$), a merged model (\mathfrak{C}_{M_+}), two component models ($\mathfrak{C}_{M_1}, \mathfrak{C}_{M_2}$), and an overlap model \mathfrak{C}_{M_0} ,
- a set of morphisms Mor consists of the following where $I_{M_0} = \mathfrak{C}_{M_0}.Ob[I]$, $I_{M_1} = \mathfrak{C}_{M_1}.Ob[I]$, $I_{M_2} = \mathfrak{C}_{M_2}.Ob[I]$, $I_{M_+} = \mathfrak{C}_{M_+}.Ob[I]$ are the underlying graph of models
 - $g_1 : I_{M_0} \rightarrow I_{M_1}$,
 - $g_2 : I_{M_0} \rightarrow I_{M_2}$,
 - $h_1 : I_{M_1} \rightarrow I_{M_+}$,
 - $h_2 : I_{M_2} \rightarrow I_{M_+}$,
- a set of constraints $SC = \{ \delta \}$ added into the shape of $\mathfrak{C}_{\text{Model-Merge}}$ with the $\langle \text{van-kampen-square} \rangle$ predicate. The constraint specifies that the diagram below on the right constitute a commuting cube and the top face constitute a van Kampen square.



5 Modeling the Optimization of Resource Allocation

To illustrate the application of our model composition approach, we consider a scenario of how resources of distributed systems can be optimized. Here, we continue the example presented in Fig. 2. Figure 3 shows a merged model \mathcal{C}_{M_+} of the merged metamodel \mathcal{C}_{MM_+} of Fig. 2. The typing of the modeling elements are not explicitly represented in the figure. In the figure, *Peter* and *Barbara* are instances of *Patient*, Peter's assigned doctor is *Dr. Logan* and Barbara's assigned doctor is *Dr. Bryan*. Dr. Logan has two available time-slots: 0950 – 1010@*Logan* and 1200 – 1220@*Logan* and Dr. Bryan has one available time-slot: 1030 – 1050@*Bryan*. There are two available time-slots at the radiology department: 0945 – 1000@*Lab* and 0930 – 0945@*Lab*. In time-slot 0950 – 1010@*Logan*, 09:50 is the start time and 10:10 is the end time. The merged model \mathcal{C}_{M_+} need to be completed with resource allocation for Peter and Barbara at the radiology and orthopedic department. One can find that there are four possible choices of resource allocation for Peter and two possible choices of resource allocation for Barbara with the available resources.

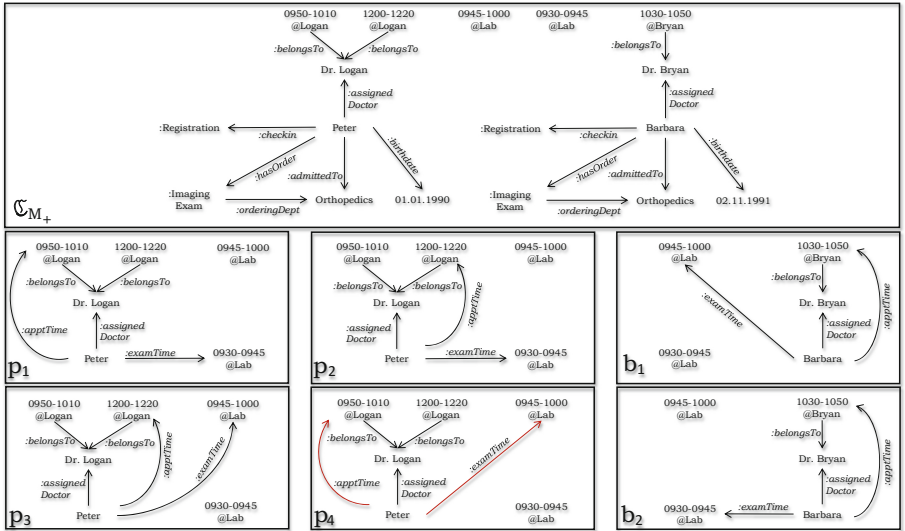


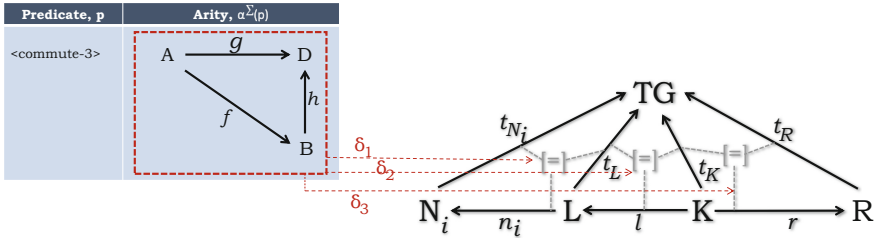
Fig. 3. A merged model \mathcal{C}_{M_+} of \mathcal{C}_{MM_+} (top) and individual resource allocation [6].

We use a transformation approach for allocating resources to the patients. We use standard double pushout approach [17] for the composition of transformation rules. In the double pushout approach, a transformation rule is defined by an input graph, an output graph, and a gluing graph to coordinate the transformation. Transformation rules can be composed in various ways such as untyped rule, typed rule, typed rule with application conditions. These variations on transformations differ in their expressive power. Transformation rules are often

described with application conditions to restrict the application intentionally. Negative application conditions are typically used in graph transformation to prohibit an infinite number of rule applications. Here we present a composition schema for transformation rule with a set of negative application conditions similar to the transformation rule proposed by Lambers et al., in [18].

Definition 9 (Typed Transformation Rule with NACs). Given a predicate signature with a 3-way commutative predicate $\langle \text{commute-3} \rangle$, a typed transformation rule with NACs $\mathfrak{C}_{Rule-N} = (Ob, Mor, SC)$ is specified by

- a set of objects Ob consisting of a type graph (TG), a matching pattern (L), a gluing graph (K), a replacement pattern (R) and a set of negative application conditions $N = \{N_i \mid i = 0 \dots n\}$,
- a set of morphisms Mor consisting of:
 - a bundle of injective graph morphisms $(n_i : L \rightarrow N_i)_{i=0 \dots n}$,
 - $K \xrightarrow{l} L$: l is an injective graph morphism,
 - $K \xrightarrow{r} R$: r is an injective graph morphism,
 - $L \xrightarrow{t_L} TG$, $K \xrightarrow{t_K} TG$, $R \xrightarrow{t_R} TG$, where $\iota_L, \iota_K, \iota_R$ are graph morphisms,
 - a bundle of graph morphisms $(t_{N_i} : N_i \rightarrow TG)_{i=0 \dots n}$
- a set of constraints $SC = \{ \delta_1, \delta_2, \delta_3 \}$ added into the shape of \mathfrak{C}_{Rule-N} with the 3-way commutative predicate $\langle \text{commute-3} \rangle$ as illustrated below:



In this transformation rule the graph L represents the input pattern or precondition, R represents the output pattern or postcondition, and K represents the common interface of L and R , i.e. their intersection which has to exist to apply the rule. In a double pushout approach, two pushouts are used to describe graph changes by first deleting graph elements and then creating new elements in the input graph. $L \setminus K$ describes the part which is to be deleted and $R \setminus K$ describes the part which is to be created by the application of the rule. For an input graph G , the rule is applied if an injective match $m : L \rightarrow G$ (i.e., structure preserving) is found from the input pattern L to the host graph G and a certain *gluing condition* [17] is satisfied. The gluing condition states that all dangling points of L , i.e., the nodes x in L such that $m(x)$ is the source or target of an edge e in $m(L \setminus K)$ must be in graph K . Figure 4 illustrates a double pushout graph transformation where m represents a match from L to G and m^* represents a comatch from R to H .

A match $m : L \rightarrow G$ of a rule satisfies a negative application condition $n_i : L \rightarrow N_i$ if there does not exist an injective morphism $q : N_i \rightarrow G$ with

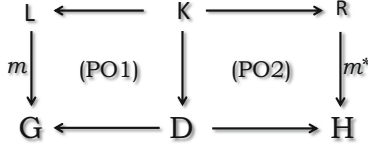


Fig. 4. Double Pushout (DPO) graph transformation [17].

$n_i; q = m$. Figure 5 illustrates a rule r_1 typed by the merged metamodel \mathcal{C}_{MM_+} . The typing information of a modeling element in r_1 appears after a colon (:). The rule specifies that a patient pt_1 is allocated with an appointment time-slot $t1$ and an exam time-slot $t2$ if $t1$ belongs to pt_1 's assigned doctor d and $t1, t2$ are not allocated to any other patient. The green color is used to indicate elements that the rule is going to produce. Negative application conditions (*NACs*) are typically used in graph transformation to prohibit an infinite number of rule applications.

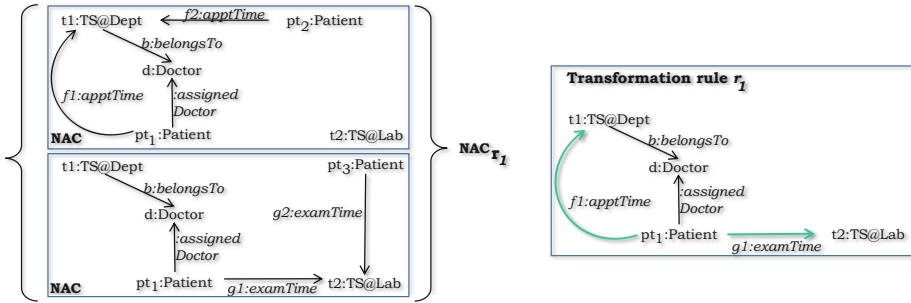


Fig. 5. Transformation rule r_1 for individual resource allocation of patients [6].

The rule can be applied over the model \mathcal{C}_{M_+} in six different ways since there are six matches. Figure 3 shows the choices of resource allocation for Peter in $p_1 - p_4$ and for Barbara in b_1, b_2 . p_1 shows a resource allocation where Peter is assigned with exam time-slot 0930 – 0945@Lab at the radiology department and appointment time-slot 0950 – 1010@Logan at the orthopedics department with Dr. Logan. This resource distribution is valid since it is not violating any constraints. Similarly, other valid resource distributions are p_2, p_3, b_1 and b_2 . However p_4 is not valid as it is violating the distributed resource constraint **C8**: the exam time-slot 0945 – 1000@Lab must be preceded by the appointment time-slot 0950 – 1010@Logan.

We apply epistemic game theory for the optimization of resources over the valid choices of resource allocation. Epistemic game theory is a broad area of research that formalizes the assumptions about rationality and mutual beliefs in a formal language to analyze games. It introduces a Bayesian perspective on decision-making in strategic situations. This model of interdependent decision

making can essentially represent a wide array of social interactions. A metamodel for modeling epistemic aspects of a system as presented in [19] is shown in Fig. 6. In this healthcare context, the players' choices are representing resource allocation options for patients. The surjective constraint imposed on the *hasChoice* relation ensures that instances of *Choice* must be associated with players. An instance of *Belief* connects the choices of a player with other players' choices denoting the choice combination of players. In game theory, *utility* represents the motivation or satisfaction of players. In an optimization problem the goal is to either maximize or minimize the utility for all the players. We use the $\langle utility(n) \rangle$ predicate to assign utility to an instance of a belief. A utility assigned to an instance of a belief denotes the utility obtained from the outcome induced by the choice combination. We use the $\langle prob(r) \rangle$ predicate to annotate an instance of a belief with probability. In our healthcare setting, patients do not have to know about each others choices. We propose to construct a strategy profile for them indirectly with the aim to optimize resource allocation respecting the preferences of patients.

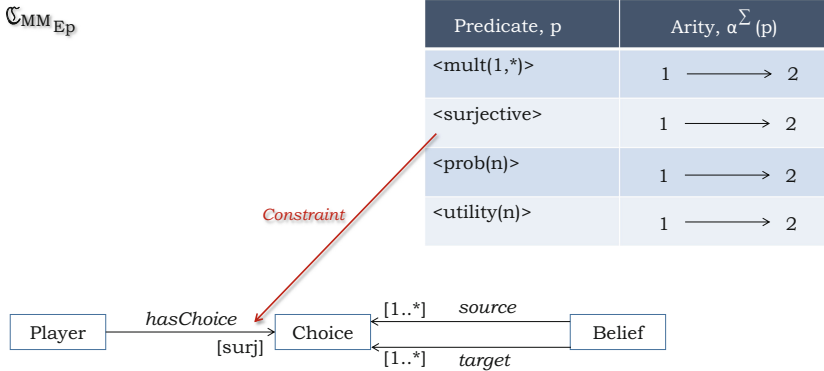


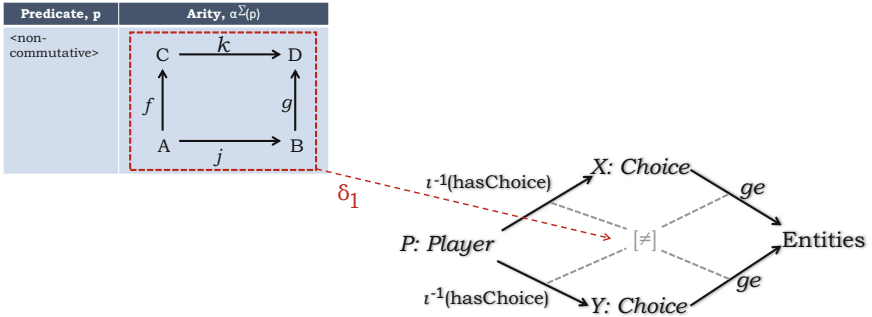
Fig. 6. A metamodel for modeling epistemic aspects [6].

Here we present a composition schema for modeling game theoretic concerns of a system. Resource allocation options are represented by a set of entity models as shown in Fig. 3. Epistemic choices in the game-theory model are coordinated with the entity models by morphisms.

Definition 10 (Composition of a Game-theory Model and Entity Models). Given a predicate signature with predicates $\langle surjective \rangle$ and $\langle non-commutative \rangle$, a composition of a game-theory model and entity models $\mathfrak{C}_{gte} = (Ob, Mor, SC)$ is specified by

- a set of objects *Ob* consisting of a game-theory model $\mathfrak{C}_{MGT} = \mathfrak{C}_M(Ob = \{I, \mathfrak{C}_{MM} := \mathfrak{C}_{MM_{Ep}}\})$ and a set of models (Entities) specified with model composition schema (see Definition 4),
- a set of morphisms $Mor = \{\mathfrak{C}_{MGT}.Ob[I] \xrightarrow{ge} Entities\}$,

- a set of constraints $SC \{ \delta_1, \delta_2 \}$ added into the shape of \mathfrak{C}_{gte} with $\langle \text{surjective} \rangle$ and $\langle \text{non-commutative} \rangle$ predicates. The constraint δ_1 specifies that if X and Y are two different choices of a player P , the following right diagram must not commute. The constraint δ_2 is added on the map ge by the $\langle \text{surjective} \rangle$ predicate and specifies that all the elements of Entities must be linked to some choice instances as illustrated below:



The graph I in the game theory model \mathfrak{C}_{MGT} of \mathfrak{C}_{gte} needs to be refined with an instance of a graph typed by the shape graph of $\mathfrak{C}_{MMEp}.Ob[S]$. Figure 7 shows a composition of a game theory model with a set of entities. The graph I is representing a game theory model where Peter and Barbara are two instances of Player. There are three options to make appointments for Peter represented by p_1, p_2 and p_3 ; two options to make appointments for Barbara represented by b_1 and b_2 . The choice instances in the game theory model are linked to the entity models via the coordinating edges ge . These game theory models can be used for optimizing the resources of distributed systems. Techniques related to the optimization may be found from [19].

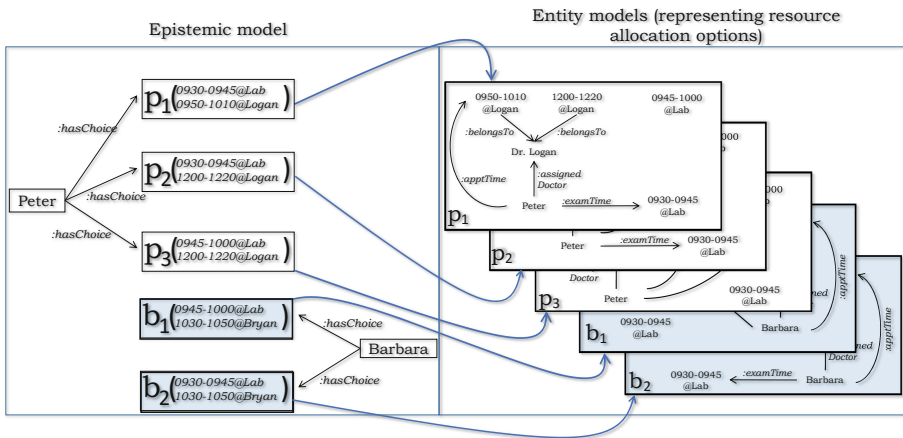


Fig. 7. Composition example of a game theory model with entities (adapted from [6]).

We show how resource allocation of distributed systems may be optimized using epistemic game theory. In our example, the patients were asked to come to the hospital at 9am. We calculate the utility based on the time spent at the hospital by the patients to get healthcare services. If a patient spends m minutes in total to get services, then her utility is m . Therefore, the longer the patients are waiting, the higher utility they get. Our target is to optimize the resource allocation for the patients so that their wait time will be reduced. Consider the option p_1 for Peter, which represents the resource allocation of 0930 – 0945@*Lab* and 0950 – 1010@*Logan*. This gives a utility of 70 as Peter is spending only 70 min at the hospital to get services. Similarly, we can calculate Barbara and Peter’s utilities and complete the epistemic model shown in Fig. 7 with belief instances by performing a Cartesian product of Barbara and Peter’s options as shown in Fig. 8. Curved arrows in Fig. 8 are representing belief instances which will be referred to as belief arrows. The belief arrow from p_1 to b_1 with an attribute 70 is representing an instance of *Belief* with utility 70.

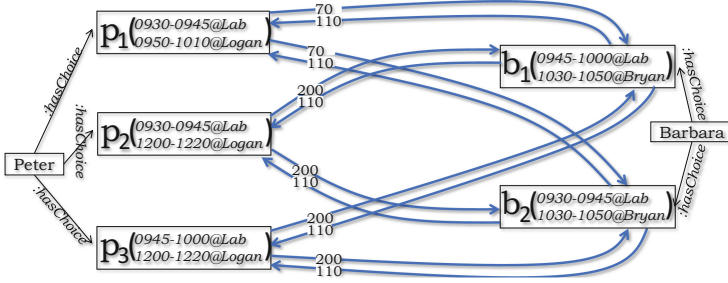


Fig. 8. Epistemic model with combination of beliefs (adapted from [6]).

An essential element of epistemic game theory analysis is the notion of belief hierarchies which are used to characterize solution concepts. We obtain belief hierarchies from an epistemic model by following belief arrows starting from any choice of a player. In our epistemic model we are only considering simple belief hierarchies. The idea of a simple belief hierarchy states that the whole belief hierarchy may be summarized by a combination of beliefs about players’ choices. A belief hierarchy is simple if it contains at most one belief of each player. The epistemic model in Fig. 8 represents six simple belief hierarchies: (i) $p_1 \xrightarrow{70} b_1 \xrightarrow{110} p_1$; (ii) $p_1 \xrightarrow{70} b_2 \xrightarrow{110} p_1$; (iii) $p_2 \xrightarrow{200} b_1 \xrightarrow{110} p_2$; (iv) $p_2 \xrightarrow{200} b_2 \xrightarrow{110} p_2$; (iii) $p_3 \xrightarrow{200} b_1 \xrightarrow{110} p_3$; (iii) $p_3 \xrightarrow{200} b_2 \xrightarrow{110} p_3$. In the healthcare context, a belief hierarchy represents the choice combination of resource allocation for patients. The belief hierarchy $p_1 \xrightarrow{70} b_1 \xrightarrow{110} p_1$ can be read as: the system believes that Peter will consider making appointments represented in p_1 if Barbara considers making appointments represented in b_1 ; and Barbara considers making appointments represented in b_1 if Peter considers making appointments represented in

p_1 . Healthcare systems need to be equipped with uncertainty management as many patients attending do not have an obvious diagnosis at presentation; also there are varieties of uncertainty in healthcare [20]. In our approach, uncertainty can be modeled by assigning probability to the belief arrows.

An important concern regarding the belief hierarchies is their consistency. A consistent belief hierarchy represents a solution concept that satisfies all the constraints specified in the metamodel specifications. Following [19] we determine how inconsistent combination of beliefs can be automatically removed from an epistemic model. If a combination of beliefs includes choices from a homogeneous metamodel, then the models are merged into a single model. For instance, if we wish to check if the combination of beliefs represented by the belief hierarchy $p_1 \xrightarrow{70} b_1 \xrightarrow{110} p_1$ is consistent, then we need to merge the models linked to p_1 and b_1 since they are typed by the merged metamodel \mathfrak{C}_{MM+} . The merged model is then checked for consistency and if it violates any constraint specified in its metamodel, we conclude that the combination of beliefs is inconsistent and should be discarded from the epistemic model.

The combination of beliefs represented by the belief hierarchy $p_1 \xrightarrow{70} b_2 \xrightarrow{110} p_1$ is inconsistent as the merged instance is violating constraint **C7** (see Sect. 4). The reason is that according to the injective constraint imposed on the reference *examTime*, patients cannot have the same time-slot. Therefore, assigning time-slot 0930 – 0945@*Lab* to both Peter and Barbara is making the merged instance inconsistent. To check that a constraint is satisfied in a given model of a metamodel, we inspect the part of a model which is affected by the constraint. This is checked by projecting out the part of the model which is affected by the constraint. This is formally defined as a pullback operation in category theory.

Similarly, we can show that the combination of beliefs represented by the belief hierarchies $p_2 \xrightarrow{200} b_2 \xrightarrow{110} p_2$, $p_3 \xrightarrow{200} b_1 \xrightarrow{110} p_3$ are inconsistent. Figure 9 shows an epistemic model with only consistent combination of beliefs.

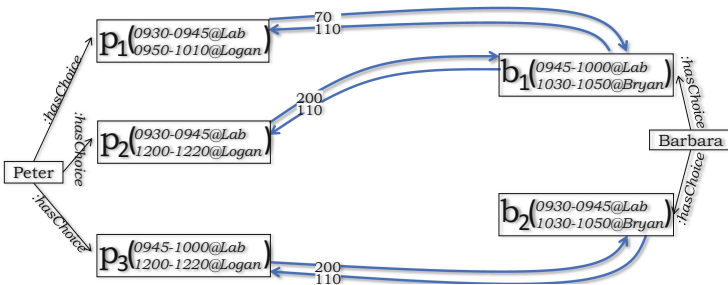


Fig. 9. Epistemic model with consistent combination of beliefs (adapted from [6]).

The epistemic choice p_2 which represents the resource allocation of 0930 – 0945@*Lab* and 1200 – 1220@*Logan* is not a rational choice since this is not optimal for any of the epistemic choices for Barbara. If Peter is allocated with

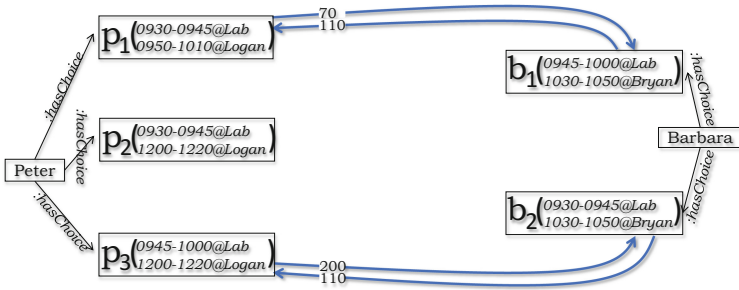


Fig. 10. Epistemic model with rational combination of beliefs (adapted from [6]).

time-slot 0930 – 0945@Lab at the radiology department, it will be optimal for Peter to see the doctor in appointment time-slot 0950 – 1010@Logan. Therefore, the epistemic choice p_2 is dominated by p_1 . The epistemic choice p_3 is a rational choice since this is optimal if Barbara is allocated with time-slot 0930–0945@Lab at the radiology department.

Figure 10 shows belief combinations with rational choices. The combination of beliefs is rational as the choices consist of the belief combinations are all rational choices with respect to the choice combination represented by the combination of beliefs.

Following [19,21], a simple belief hierarchy generated by a combination of beliefs leads to a *Nash equilibrium* if the combination of beliefs is rational. Therefore, we obtain two *Nash equilibria* from the epistemic model represented in Fig. 10:

- Allocating time-slot 0930 – 0945@Lab and 0950 – 1010@Logan for Peter is rational if time-slot 0945 – 1000@Lab and 1030 – 1050@Bryan are allocated to Barbara gives a total utility of $(70 + 110) = 180$;
- Allocating time-slot 0945 – 1000@Lab and 1200 – 1220@Logan for Peter is rational if time-slot 0930 – 0945@Lab and 1030 – 1050@Bryan are allocated to Barbara gives a total utility of $(200 + 110) = 310$.

However, the first equilibrium has a lower total utility for the system.

6 Related Work

We provide a survey of existing approaches for modeling the composition of system that includes merging models, metamodels, views and megamodeling. Our approach of modeling in the small and modeling in the large is heavily influenced by all these approaches where we have generalized the concepts and presented a formal foundation that can be used for modeling composition of heterogeneous systems.

Sabetzadeh and Easterbrook [11] presented an algebraic approach for merging multiple views. In their approach, large models are constructed by manipulating individual views representing information relevant to different development

concerns. The view merging framework proposed in [11] is based on a colimit construction. The proposed method can be applied to a variety of graphical modeling languages as the underlying syntactic structure of the views follows a graph-based formalism. The merge framework incorporates a systematic annotation scheme which can be used to identify a potential inconsistent portion of a view. While modeling a view, a modeler can express inconsistencies that arise due to stakeholders' conflicting beliefs during the requirement analysis phase. Stakeholders' beliefs about the content of views are represented using annotations denoting the degree of knowledge. The formalization of the degree of knowledge is based on knowledge orders [22,23]. The annotation scheme allows a modeler to hypothesize possible interconnections for a set of views. An implementation of the framework exists as a proof of concept in the Java tool called *iVuBlender* [24]. The framework can be used to explicitly model inconsistencies and capture typing constraints to some degree, but cannot capture constraints whose articulation rely on the semantics of the modeling language being used.

The multimodeling approach refers to a set of heterogeneous models where each model captures a specific view of the system and is represented by a set of domain concepts and local constraints [25]. Diskin et al. [25,26] presented a framework for multimodeling based on category theory. In their approach, software models are formalized as typed graphs and a multimodel consists of (i) component models; (ii) an overlap specification; and (iii) inter-metamodel constraints. A model is defined as a triple with a type graph (M_A), a graph specifying the data of the model (G_A), and a graph morphism from G_A to M_A called typing. Overlap between models is specified by a pair of model mappings with a common source model representing common concepts of the component models. The mappings show how the common concepts are represented in each of the component models. This configuration of models and mappings is called a *span* and represents an overlap specification of a multimodel. An overlap specification can be further decomposed into two graph spans: a metamodel span $M_1 \xleftarrow{r_{1M}} M_0 \xrightarrow{r_{2M}} M_2$ and a data span $G_1 \xleftarrow{r_{1G}} G_0 \xrightarrow{r_{2G}} G_2$. Component metamodels are merged automatically by performing a pushout (PO) of morphisms r_{1G} and r_{2G} in the category of graphs. Local constraints defined on the component metamodels are directly carried into the merged graph along with the maps r_{1G} and r_{2G} . Inter-metamodel (i.e., global) constraints are specified over the component models of a multimodel [25,27]. Even if the component models of a multimodel are consistent according to their metamodels, they may become inconsistent after merging. Checking global consistency is challenging as it requires building big and possibly unfeasible merged metamodels and models. König and Diskin provided a mechanism for efficiently checking global constraints by reducing the workload for matching [28].

A conceptual framework of *megamodeling* was presented in [12–14] for modeling and reasoning about large-scale software evolution processes without entering into the details of the technological space involved. A *megamodel* is used to describe MDE by explaining the concept of model, language, metamodel, and transformation. The core of an MDE megamodel centered around four relations.

The *DecomposedIn* relation (δ) in the megamodel is used to decompose a complex system into subsystems or parts. For instance, $S_1 \delta S_2$ indicates that system S_2 is a part of the system S_1 . The *DecomposedIn* relation can be represented as a δ association in a UML object diagram. The μ association (*RepresentationOf* relation) is used to represent a relationship between a model and the system under study. The notion of model is relative and a system could play the role of a model. In the megamodel, a system can play the role of a language by associating a set of systems with the relation *ElementOf* (\in). A grammar can be referred to as a model of a language. This concept leads to the definition of a metamodel. In the megamodel, metamodels are *models of languages of models*. By this definition, a metamodel ensures that a model must conform to its metamodel.

Favre and NGuyen [14] introduces the *IsTransformedIn* (τ) association in the megamodel to represent model transformation. A *IsTransformedIn* relation has a source and a target model. For instance, $m_{src} \delta m_{trg}$ indicates that the model m_{src} is transformed to model m_{trg} . The pair (m_{src}, m_{trg}) of a *IsTransformedIn* relation is called a *transformation instance*. A transformation instance can be further considered as a system and can be used as a first-class entity in the megamodel. Software evolution processes can be modeled as a graph using the megamodel where the graph is composed by a combination of τ links and other kind of links specifying a mega-pattern. A set of mega-patterns have been provided in [14] representing various kinds of evolution and co-evolution transformations. This approach of megamodeling is inherently abstract and lacking structural details of the models and intermodel relations.

In [29], Diskin et al. addressed this abstraction gap with a new type of megamodel, called a *mapping aware (MA-)megamodel* and presented a mathematical framework based on category theory. A (MA-)megamodel provides additional internal structure to a megamodel and remains abstract and independent of any particular modeling language. It provides different level of abstraction to a megamodel designer. A megamodel designer can zoom into a megamodel's nodes and edges and disassemble them into elementary building blocks. These elementary blocks are composed to build classical megamodeling constructs: conformance, overlapping, consistency, and transformation relationships. Complex megamodeling constructs are built by the composition of the same elementary blocks, e.g., bidirectional transformations and heterogeneous merge. A list of usable design patterns have been formalized in [29] for megamodel engineering including model; model mappings; model overlap; descriptive views; prescriptive views; model transformation; incremental update. The approach outlines a mathematical framework based on graph and category theory that can be used to provide formal semantics of the patterns. The patterns include declarative aspects of megamodeling operations which is being reflected in the view and transformation definition mappings.

The abstraction gap has been addressed in a different way in [30] by a linguistic architecture where models and relationships of a megamodel are linked to illustrative software artifacts. Linked resources are described with linguistic

relationships concerning dataflow, language membership, schema/type conformance, and correspondence. With the linguistic architecture, linked resources can be explored and validated. The megamodeling approach has been demonstrated for analyzing Object/XML mappings (O/X mapping).

7 Conclusion

In this paper we presented a formal framework for model composition. We studied a variety of composition patterns and formalize them by means of reusable composition schemata. We showed how composition schemata can be used for both modeling in the small and modeling in the large. The approach can be used to specify abstract specifications of modeling constructs which can then be enhanced with further refinement. To show the applicability of the proposed method, we presented an application of model composition for optimizing distributed resource allocations in a healthcare context. Distributed systems are often loosely connected and inter-dependencies are not defined into their software model. This limits the scope of optimization of distributed resources. We therefore proposed to use a model composition approach for articulating distributed resource constraints. We proposed to apply model-driven engineering and use model transformation rules to construct a game theoretic model with purpose of optimizing distributed resource allocation. We presented a diagrammatic approach for modeling the constraints and conflicting situations of a strategic game. In particular, producing resource allocation choices by applying model transformation rules is an automated process. This means that it alleviates the effort required by the modeler to manually enter the choices.

References

1. Bagdasaryan, A.: Systems theoretic techniques for modeling, control, and decision support in complex dynamic systems. CoRR abs/1008.0775 (2010)
2. Erdweg, S., Giarrusso, P.G., Rendel, T.: Language composition untangled. In: Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications, LDTA 2012, pp. 7:1–7:8. ACM, New York (2012)
3. Haber, A., Look, M., Perez, A.N., Nazari, P.M.S., Rumpe, B., Völkel, S., Wortmann, A.: Integration of heterogeneous modeling languages via extensible and composable language components. In: Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development. MODELSWARD 2015, Portugal, pp. 19–31. SCITEPRESS - Science and Technology Publications, Lda (2015)
4. Astesiano, E., Bidoit, M., Kirchner, H., Krieg-Brckner, B., Mosses, P.D., Sannella, D., Tarlecki, A.: CASL: the common algebraic specification language. *Theor. Comput. Sci.* **286**, 153–196 (2002)
5. Mossakowski, T.: Relating CASL with other specification languages: the institution level. *Theor. Comput. Sci.* **286**, 367–475 (2002)

6. Rabbi, F., Kristensen, L.M., Lamo, Y.: Optimizing distributed resource allocation using epistemic game theory: a model-driven engineering approach. In: Pires, L.F., Hammoudi, S., Selic, B. (eds.): Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2017, 19–21 February 2017, Porto, Portugal, pp. 41–52. SciTePress (2017)
7. Rutle, A.: Diagram predicate framework: a formal approach to MDE. Ph.D. thesis, Department of Informatics, University of Bergen, Norway (2010)
8. Lamo, Y., Wang, X., Mantz, F., MacCaull, W., Rutle, A.: Computer and information science 2012. In: Lee, R. (ed.) DPF Workbench: A Diagrammatic Multi-layer Domain Specific (Meta-)Modelling Environment. Studies in Computational Intelligence, vol. 429, pp. 37–52. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30454-5_3
9. Herrmann, C., Krahn, H., Rumpe, B., Schindler, M., Völkel, S.: An algebraic view on the semantics of model composition. In: Akehurst, D.H., Vogel, R., Paige, R.F. (eds.) ECMDA-FA 2007. LNCS, vol. 4530, pp. 99–113. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-72901-3_8
10. Wang, X.: Towards correct modelling and model transformation in DPF. Ph.D. thesis, Department of Informatics, University of Bergen, Norway (2016)
11. Sabetzadeh, M., Easterbrook, S.: An algebraic framework for merging incomplete and inconsistent views. In: 13th International Requirements Engineering Conference, RE 2005, pp. 306–315 (2005)
12. Favre, J.: Foundations of model (driven) (reverse) engineering: models - episode I: stories of the fidus papyrus and of the solarus. In: Bézivin, J., Heckel, R. (eds.) Language Engineering for Model-Driven Software Development, Dagstuhl Seminar Proceedings, 29 February–5 March 2004, vol. 04101. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany (2004)
13. Favre, J.M.: Foundations of meta-pyramids: languages vs. metamodels - episode ii: story of thotus the baboon1. In: Bezivin, J., Heckel, R. (eds.) Language Engineering for Model-Driven Software Development, Dagstuhl Seminar Proceedings, no. 04101. Dagstuhl, Germany, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany (2005)
14. Favre, J.M., NGuyen, T.: Towards a megamodel to model software evolution through transformations. In: Electronic Notes in Theoretical Computer Science, vol. 127, pp. 59–74 (2005). Proceedings of the Workshop on Software Evolution through Transformations: Model-Based vs. Implementation-Level Solutions (SETra 2004) (2004)
15. Diskin, Z., Xiong, Y., Czarnecki, K.: Specifying overlaps of heterogeneous models for global consistency checking. In: Proceedings of the First International Workshop on Model-Driven Interoperability. MDI 2010, pp. 42–51. ACM, New York (2010)
16. Diskin, Z.: Model synchronization: mappings, tiles, and categories. In: Fernandes, J.M., Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2009. LNCS, vol. 6491, pp. 92–165. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18023-1_3
17. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science. Springer, Heidelberg (2006). <https://doi.org/10.1007/3-540-31188-2>
18. Lambers, L., Ehrig, H., Prange, U., Orejas, F.: Embedding and confluence of graph transformations with negative application conditions. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 162–177. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87405-8_12

19. Rabbi, F., Lamo, Y., Yu, I.C.: Towards a categorical approach for meta-modelling epistemic game theory. In: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems. MODELS 2016, pp. 57–64. ACM, New York (2016)
20. Han, P.K.J., Klein, W.M.P., Arora, N.K.: Varieties of uncertainty in health care. *Med. Decis. Making* **31**, 828–838 (2011). PMID: 22067431
21. Perea, A.: *Epistemic Game Theory: Reasoning and Choice*. Cambridge University Press, Cambridge (2012)
22. Belnap, N.D.: Modern uses of multiple-valued logic. In: Dunn, J.M., Epstein, G. (eds.) *A Useful Four-Valued Logic*, vol. 2, pp. 5–37. Springer, Netherlands, Dordrecht (1977). https://doi.org/10.1007/978-94-010-1161-7_2
23. Ginsberg, M.L.: Bilattices and modal operators. *J. Log. Comput.* **1**, 41–69 (1990)
24. Sabetzadeh, M., Easterbrook, S.: iVuBlender: a tool for merging incomplete and inconsistent views. In: 13th IEEE International Conference on Requirements Engineering (RE 2005), pp. 453–454 (2005)
25. Diskin, Z., Xiong, Y., Czarnecki, K.: Specifying overlaps of heterogeneous models for global consistency checking. In: Proceedings of the First International Workshop on Model-Driven Interoperability. MDI 2010, pp. 42–51. ACM, New York (2010)
26. Diskin, Z., König, H.: Incremental consistency checking of heterogeneous multimodels. In: Milazzo, P., Varró, D., Wimmer, M. (eds.) STAF 2016. LNCS, vol. 9946, pp. 274–288. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-50230-4_21
27. Sabetzadeh, M., Nejati, S., Liaskos, S., Easterbrook, S., Chechik, M.: Consistency checking of conceptual models via model merging. In: 15th IEEE International Requirements Engineering Conference (RE 2007), pp. 221–230 (2007)
28. König, H., Diskin, Z.: Advanced local checking of global consistency in heterogeneous multimodeling. In: Wasowski, A., Lönn, H. (eds.) ECMFA 2016. LNCS, vol. 9764, pp. 19–35. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-42061-5_2
29. Diskin, Z., Kokaly, S., Maibaum, T.: Mapping-aware megamodeling: design patterns and laws. In: Erwig, M., Paige, R.F., Van Wyk, E. (eds.) SLE 2013. LNCS, vol. 8225, pp. 322–343. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-02654-1_18
30. Favre, J.-M., Lämmel, R., Varanovich, A.: Modeling the linguistic architecture of software products. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds.) MODELS 2012. LNCS, vol. 7590, pp. 151–167. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33666-9_11



Generative versus Interpretive Model-Driven Development: Moving Past 'It Depends'

Michiel Overeem¹(✉), Slinger Jansen², and Sven Fortuin²

¹ Department of Architecture and Innovation,
AFAS Software, Leusden, The Netherlands
m.overeem@afas.nl

² Department of Information and Computing Sciences, Utrecht University,
Utrecht, The Netherlands
{slinger.jansen,s.e.fortuin}@uu.nl

Abstract. Model-driven development practices are used to improve software quality and developer productivity. However, the design and implementation of an environment with which software can be produced from models is not an easy task. One part of such an environment is the model execution approach: how is the model processed and translated into running software? Experts state that code generation and model interpretation are functionally equivalent. However, a survey that we conducted among several organizations shows that there is a lack of knowledge and guidance in designing the execution approach. In this article we present the results of a literature study on the advantages of both interpretation and generation. We also show, using a case study, how these results can be utilized in the design decisions. Finally, a decision support framework is proposed that can provide the guidance and knowledge for the development of a model-driven engineering environment.

Keywords: Model-driven development · Model-driven architecture
Software architecture · Code generation
Run-time model interpretation · Decision support

1 Introduction

Model-driven development (MDD) is used by software producing organizations (SPOs) to improve software quality and developer productivity. According to

This work is a result of the AMUSE project. See <https://amuse-project.org> for more information. An earlier version of this work was published as Overeem and Jansen [1]. This article adds the motivating survey that we have conducted among sixteen organizations. The literature study is extended with literature published since the original study. The case study is expanded with more in-depth observations on the decision making process.

Díaz et al. [2] these improvements in quality and productivity are achieved because a well designed model raises the abstraction level of the software development process. The abstracted model allows for an expressiveness that can be more concise than general-purpose programming languages. Domain-specific modeling improves that even further by catering the model to a certain domain. The expressiveness causes both the increase of productivity (more can be done with less) and the quality (there will be fewer mistakes, because there is a smaller model). The models can be used in different manners, Brown [3] shows a modeling spectrum with, among others, roundtrip engineering, model-centric, and model only. We are especially interested in the model-centric approach: the model is the source of truth and the application follows from the model. The model-centric approach is implemented in Model Driven Engineering Environments (MDEE), an environment that is similar to an Integrated Development Environment (IDE) used for software development. Modelers create models using modeling languages in a specific modeling environment, just as developers write software in their IDE. These models are translated according to well defined semantics, into an application. Together these components (from modeling environment up to and including the application) form the MDEE (visualized in Fig. 1). The translation process that reads the model and produces an application is defined as the *model execution approach*, and implemented in the model execution engine.

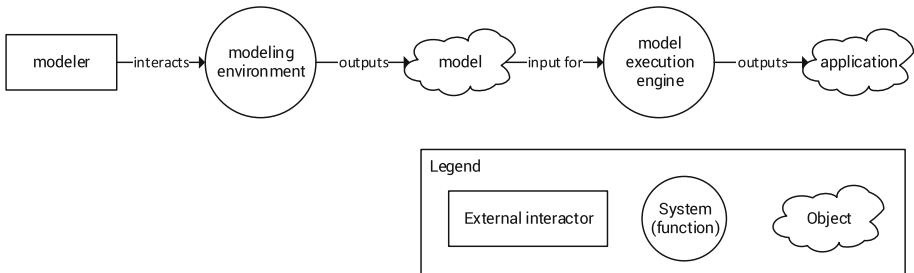


Fig. 1. A model-driven engineering environment enables a modeler to create a model in a *modeling environment*. The model is subsequently translated by the *model execution engine* (using a *model execution approach*) into an application.

Our experiences are that the development of a MDEE is by no means an easy task. The initial investment is large, because there are many technical challenges. One of these technical challenges that is of particular interest to us, is the design and development of the *model execution approach*. SPOs can choose for code generation, run-time interpretation, or a hybrid form that combines both approaches (Fig. 2). As in every design challenge, there are numerous decisions to make (with their specific trade-offs) that influence the overall quality of the MDEE.

Just like any other (architectural) design question, the design questions for the model execution approach can be answered with “it depends”. In this article we show that the design depends on desired quality characteristics and the context of the MDEE. Moreover, we show how SPOs can take these characteristics into account. It might be regarded as an implementation detail, but the model execution approach, like any other component in the system has its influence on the quality characteristics (such as run-time behavior and maintainability) of the whole system. As in any system that consists of multiple components working together, the model execution approach should not be designed individually (i.e., not be out of the context of the MDEE). The influence of the model execution approach is similar to, for example, the influence of a specific database on the quality of a data intensive system. While users may not see a difference in functionality between two different databases, the quality of the system is affected by it, for example, in terms of performance, stability, and availability. SPOs can deliver the same functionality, whether they choose code generation or run-time interpretation, but the quality of the MDEE will differ significantly.

The main research question of this article is *How can SPOs make an informed decision between a generative or interpretive model execution approach?*. In Sect. 2 we explain the different model execution approaches in more depth, and discuss the work already done in this area. We motivate our research question in Sect. 3 by presenting the results of a survey among SPOs applying model-driven development. This survey shows that there is no “one size fits all” solution. It also shows that many SPOs do not have a clear rationale for the model execution approach that is used. Therefore, decision support and clear guidance is necessary to improve the overall design and implementation of MDEEs. Section 4 discusses the results of the literature study that we have performed on the advantages and disadvantages of the generative and interpretive approach. There are many hybrid model execution approaches that combine the generative and interpretive approach. We show the preference for the two pure approaches in terms of percentages. These percentages can be used by the SPO to find the right balance in designing their own hybrid model execution approach. Section 5 describes a case study, in which we observe the design of a fitting model execution approach. We conclude that the design of a fitting model execution approach is not detached of the overall design of the MDEE. We reflect on the case study and our observations in Sect. 6. We observed three general areas of design decisions that influence the model execution approach, and we present a design support framework based on the case study. Finally, Sects. 7 and 8 evaluates and discusses the study, and presents our conclusion respectively.

2 Context and Related Work

There are several model execution approaches, many of them are a hybrid form of the two pure approaches. We discuss the two pure approaches, and describe two groups of hybrid approaches, shown in Fig. 2. The first pure model execution approach is *code generation*. During code generation a model is parsed,

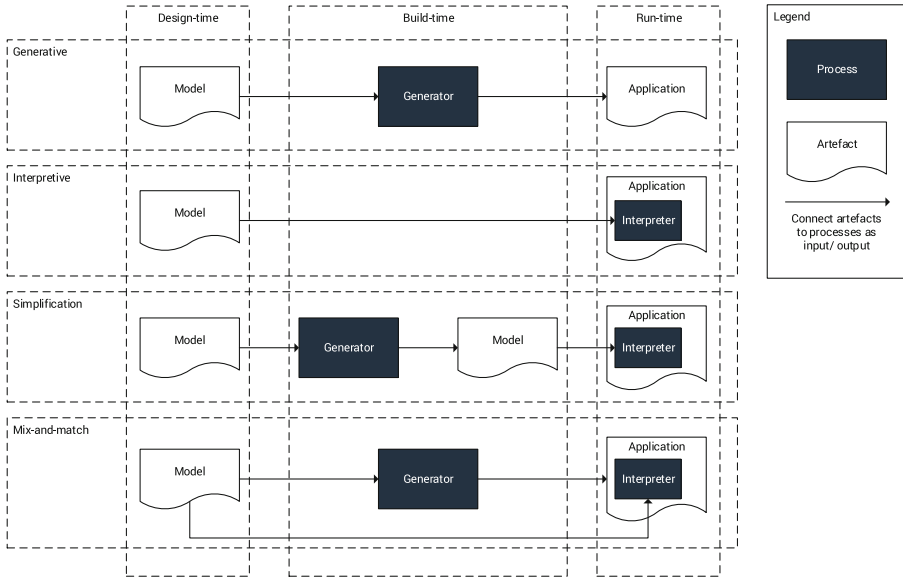


Fig. 2. The four main types of execution approaches are generation, interpretation, simplification, and mix-and-match. The darker boxes show the execution process. With the two hybrid approaches, the execution process can be split up and divided between build-time and run-time. The model is created at design-time, but is used at build-time and/or run-time.

interpreted and transformed into source code. The generated source code generally results in running software. This approach is not exclusive to MDD, and is formalized and defined by Czarnecki and Eisenecker [4] as *Generative Programming*. According to their definition, it is a paradigm based on modeling facilities to automatically manufacture customized and optimized intermediate and/or end-products. Applying generative programming within MDD results in generative MDD. Although nothing in the definition states that the output can not be changed manually before the final software product is delivered, we only regard *full* code generation that does not need manual changing the generated code. This does not imply that every part should be generated; the generated code can be combined with frameworks or base libraries, as pointed out by Kelly and Tolvanen [5].

The second pure model execution approach is *run-time interpretation*, or interpretive MDD. The idea is similar to code generation, but the timing is different: the parsing and interpretation of the model are done at run-time. There is no need to first generate source code, the running software executes its functions directly based on the model. In this case the model execution approach becomes part of the application, the application interprets the model before offering functionality based on the model. Further manual coding is not possible with this approach, because there is no time to intervene in the execution of the

software. However, as we see in Sect. 3 it is possible to combine custom code with an interpreter. The model needs to be deployed along with the running software, while in the generative approach, the model is not part of the running software.

These two approaches form the extremes of the execution spectrum, and many hybrid forms are possible. We see two groups of hybrid approaches. The first group is *simplification*: a model is transformed into a second model before deploying it for run-time interpretation. In this approach there is both a generation step and an interpretation step, instead of generating source code. The generation step transforms the model into a second model that can be interpreted at run-time. This can be achieved by transforming high-level concepts to low-level concepts, or by transforming into a model with fewer constructs. The results of this approach are manifold: (1) The interpreter is easier to develop and better maintainable, because it has to support fewer constructs. (2) The translation is less complex, so the interpreter is faster. And finally, (3) the interpreter becomes more reusable, because there can be many different models that can be transformed into the intermediate model. This approach is also used by programming languages that compile into an intermediate language that is in turn interpreted by a runtime environment, such as the approach Meijler et al. [6] discuss. They generate Java source code, but use a customized class loader that acts as a run-time interpreter.

The second group of hybrid approaches is a *match-and-mix* approach: some parts of the platform use code generation, while others use interpretation. This approach can be used both from an architectural perspective as well as from a model perspective. The MDEE could use a different approach in different components, for instance the user interface could be interpreted, while the database access layer is generated. Different approaches could also be chosen based on model dynamics, where the more stable parts can be generated into source code and the more dynamic parts are interpreted.

Figure 2 shows the four described approaches, marking out the time at which the execution takes place. In the generative approach, the execution is done at build-time, as opposed to the interpretive approach in which the execution takes place at run-time. Both the simplification and mix-and-match approach show that they have part of the execution at build-time, and part at run-time. This makes them flexible, because SPOs can decide how much happens at what time. These hybrid approaches can also be combined, the mix-and-match approach can combine the interpretive, generative, and simplification approach into a single encompassing model execution approach.

There is some work done on the challenge of designing a fitting model execution approach discussed in this article. A multi-criteria analysis of the different approaches is performed by Batouta et al. [7] with as goal the support of the decision-making. Their analysis results in a decisive statement about the best approach (based on their list of ten criteria). However, they do not take the context of the MDEE into account. Fabry et al. [8] address a number of advantages regarding the different model execution approaches, but they do not give any support for the decision-making. Zhu et al. [9] researches the decision-making within

MDD applied to game development, however, he only looks at other architectural decisions than the model execution approach within a MDEE. Code generators and the interaction with developers is researched by Guana and Stroulia [10], only without making a comparison with the interpretive approach. All of the mentioned work is incorporated in the literature study in Sect. 4.

The design of software and their architectures is a thoroughly researched topic. Capilla et al. [11] show how design decisions play a role in software architecture, and that it is important to capture them. Jansen and Bosch [12] define “software architecture as the composition of a set of architectural design decisions”, and formalize this in the Archium approach which is further extended in Ven et al. [13]. Svahnberg et al. [14] present a decision process that, based on desired quality attributes, supports a SPO in finding the architecture variant that shows the most potential. We combine the definition of software architecture as a set of design decisions with the approach to support a decision with quality attributes, and apply this to MDD. Because of this we are able to uncover the rationale of either a generative or interpretive approach, and support SPOs in their design process.

3 How SPOs Design and Develop MDEEs

We interviewed twenty-two product experts of sixteen different SPOs that develop MDEEs. All of the experts had either five or more years experience with the product or were working with the product since its start. They served in different roles at the time: twelve of them as chief executive, the others in different roles such as lead developer, business developer, and sales manager. These experts were asked questions on the design and implementation of their company’s MDEE. The SPOs were identified by an Internet search, exploiting our network, and asking interviewed product experts.

We identified 36 qualifiable case companies with representatives in Belgium, The Netherlands, or Luxembourg. For sixteen companies we found experts that were willing and able to cooperate in our research¹. The companies differ in size (ranging from ten employees to thousands of employees), in market (some operate only in The Netherlands, while others operate worldwide), and maturity (some MDEEs are almost twenty years old, while others only two years). After we processed the answers, every expert had the opportunity to correct any mistaken interpretations. The answers are summarized in Table 1.

The first topic of interest is the target users of the MDEE and its modeling language. We asked the experts what the target group of users for the MDEE are, and what kind of expertise they expect from them. Their answers resulted in four categories of users:

¹ Some needed to be excluded due to confidentiality issues or the lack of (technical) knowledge.

Table 1. Anonymized results of the survey among SPOs. The target users and the model execution approach are shown, a long with the company size (in terms of number of employees) and maturity (in terms of number of development years). The cells marked with an * identify MDEEs that support two distinct web platforms.

	<i>SPO</i> ₁	<i>SPO</i> ₂	<i>SPO</i> ₃	<i>SPO</i> ₄	<i>SPO</i> ₅	<i>SPO</i> ₆	<i>SPO</i> ₇	<i>SPO</i> ₈	<i>SPO</i> ₉	<i>SPO</i> ₁₀	<i>SPO</i> ₁₁	<i>SPO</i> ₁₂	<i>SPO</i> ₁₃	<i>SPO</i> ₁₄	<i>SPO</i> ₁₅	<i>SPO</i> ₁₆	
Company size																	
0-50 employees	•	•	•	•			•					•		•	•	•	56%
100-500 employees						•		•	•	•			•				31%
+500 employees					•						•						13%
Development years																	
0-5			•						•			•					19%
6-15	•	•		•						•						•	37%
+15					•	•	•	•			•		•	•			44%
Target platforms																	
Web	•	•	•	•	•	•	*	•	*	•	•	•	•	•	*	•	100%
Desktop				•		•							•				19%
Mobile													•				6%
Target users																	
Laymen	•			•					•			•				•	32%
Technical business users		•	•		•			•		•	•		•	•		•	56%
SQL experts				•													6%
Developers	•	•				•	•			•						•	37%
Model execution approach																	
Interpretation	•	•	•	•		•	•			•			•	•	•	•	69%
Generation	•	•				•		•	•		•	•	•			•	50%
Simplification					•				•								13%
Match-and-mix	•	•				•			•			•				•	38%

- **Laymen** are people without any technical knowledge.
- **Technical business users** are those that have some knowledge of software development, but are no developers. They are expected to have knowledge about software concepts such as data models, and data types. An informal description would be people more knowledgeable than layman, but less knowledgeable than developers.
- **SQL experts** are a specific set of users that are able to write SQL queries. They are not able to write software in other programming languages. This specific category was added after the review with *SPO*₄, because the category **developers** did not match their target description.
- **Developers** are those users that are able to write software in a programming language. MDEEs that target developers expect them to be familiar with IDEs and other programming concepts.

A third of the SPOs specifically target laymen, while the others require some form of technical knowledge of their users. There is no correlation found between

the model execution approach and the targeting of laymen. In the case study we conducted (see Sect. 5) we also observe the design of a MDEE that targets laymen. A third of the SPOs that target technical business users also target developers, their MDEEs support custom programming, because the model is not able to express all required functionality. The six SPOs that target developers all use an interpretive approach, four of them also use code generation.

Five SPOs (SPO_2 , SPO_6 , SPO_9 , SPO_{13} , and SPO_{16}) state that a reason for their model execution approach is a certain required build-time behavior. As an example, the expert of SPO_{16} states “*You can’t generate code again in an end application that is already generated. To allow workflow modeling in the end application, we were forced to make use of an interpretative solution.*”. All of the five mentioned SPOs explain that users are able to change the model, and expect that their changes are (near) instantly applied and visible in the application. Four of them use run-time interpretation, while the other one uses a simplification approach. The SPOs that use a generative approach did not mention such a requirement for build-time behavior.

All of the SPOs target a web platform, meaning that they support at least back-end and front-end applications. Three of the SPOs, however, support two different back-end platforms, one also supports mobile applications, and two others also support native desktop applications. Effectively, we can conclude that all SPOs support multiple platforms. The interpretive approach is motivated three times by the advantage of platform independence, or portability.

We have found little reasoning behind the implemented approaches, one expert even stated “*We just had to go with one of the two.*”. An expert of SPO_6 refers to an advantage in portability for interpretation, a correlation that we will see again in Sect. 4: “*By interpreting the UI and generating the remaining parts of the application, we are able to share models between different platforms.*”. A reference to resource utilization is made by an expert of SPO_{14} : “*We don’t want to regenerate an entire database every time the model changes, because this can potentially cause a lot of problems with data migration.*”. The interviews show that all approaches are used, and nearly half of them use a hybrid form. This supports our claim that the model execution approach depends on many factors and is context specific. We cannot give a simple answer such as “web platforms should use an interpreter”, Table 1 shows that other approaches are used for web platforms as well. Like Capilla et al. [11] we believe that it is important for SPOs to document the rationale behind important architectural decisions. In the next Section we will show that the model execution approach influences the quality of the MDEE, and that it is important to capture the rationale of the design.

4 Quality Characteristics of Model Execution Approaches

We started the literature study by executing a literature review on the advantages and disadvantages of both code generation and run-time interpretation. The literature review was done with the snowballing approach as described by Wohlin [15]. The snowballing approach uses references between articles as a

means to discover other relevant literature. The first step is to select a start set from which the references can be followed. This approach was chosen because the research areas to be covered in this review are broad. We expected literature from the MDD field as well as Domain-Specific Language engineering and compiler design. The second reason was that the literature that we had found in earlier explorations never mentioned the advantages or disadvantages directly, but they were often hidden in implementation details.

Our start set was created by earlier informal explorations with the Google Scholar engine, using “interpretation versus code generation” and “interpretation vs. code generation” as keywords. We selected five articles as the start set: van Deursen et al. [16], Meijler et al. [6], Mernik et al. [17], Tanković [18], and Voelter [19]. These papers represent the different research areas and have a broad research question, resulting in many references (both backwards and forwards). With this start set we executed several steps, following both backward and forward references. The found literature was included when it mentioned advantages or disadvantages on model execution approaches, and we ended up with 35 studies.

The literature was classified using the ISO standard 25010:2011 for software product quality [20]. This standard is used to assess the quality of software systems, and matches our intent to assess the quality of MDEEs. The ISO standard consists of eight categories with 31 characteristics. We found evidence for differences in quality fulfillment for five out of these eight categories, summarized in Table 2. The summary of all the found evidence is presented in Table 3. There was no evidence found for the categories *functional suitability*, *usability*, and *reliability*. The first two categories match the statement of Stahl et al. [21] “code generation and model interpretation are functionally equivalent”. For category *reliability* no evidence was found as well, which was expected. Reliability is the degree to which the system performs its functions under certain conditions. We assume the generative and interpretive approach to be functionally equivalent, and in both approaches it is possible to build a reliable functioning system.

Table 3 presents every mention of an advantage or disadvantage in relation to its source and the quality characteristics. A *G* stands for a preference of generation over interpretation, while an *I* stands for the opposite. When we encountered statements on the two approaches without a preference, we marked the corresponding cell with both *G* and *I*. The total number of preferences are used to calculate the percentage of the two alternatives with respect to the quality characteristic. The evidence we found is presented in relation to the generative and interpretive approach. This does not mean that the hybrid approaches are not mentioned by authors (as discussed in Sect. 2). However, the advantages and disadvantages we found were always in terms of the generative and interpretive aspects of an approach.

Table 2. The categories and characteristics from the software product quality model in ISO standard 25010:2011. For the emphasized items we found evidence of a preference for either code generation or model interpretation.

Category	Characteristics
Functional suitability	Functional completeness, Functional correctness, Functional appropriateness
<i>Performance efficiency</i>	<i>Time behaviour, Resource utilization, Capacity</i>
<i>Compatibility</i>	<i>Co-existence, Interoperability</i>
Usability	Appropriateness recognizability, Learnability, Operability, User error protection, User interface aesthetics, Accessibility
Reliability	Maturity, Availability, Fault tolerance, Recoverability
<i>Security</i>	<i>Confidentiality, Integrity, Non-repudiation, Accountability, Authenticity</i>
<i>Maintainability</i>	<i>Modularity, Reusability, Analysability, Modifiability, Testability</i>
<i>Portability</i>	<i>Adaptability, Installability</i>

4.1 ISO: Performance Efficiency

The characteristics in the category *Performance efficiency* describe the performance of a system: how the system utilizes resources, responds to requests, and meets the capacity requirements. For two of the characteristics evidence was found.

Time Behavior - The first characteristic for which we found evidence is the time behavior of the system. For MDEEs, this is a special characteristic, because there are two main use cases for which the response and processing time is important. The run-time time behavior describes the response time of the functionality offered in the application. However, the second important use case for which response time is important, is the translation from model to application. When a generative approach is used, the model execution approach takes up time between model changes and software updates. When an interpretive approach is used, there is no time between model changes and software updates, because the execution happens during execution of normal system functions. These two distinct use cases are confirmed by the literature that we studied: we found comments in relation to both approaches. Therefore this characteristic is split into two separate characteristics. Both build-time time behavior and run-time time behavior are used as two separate characteristics in our study.

Twenty-two out of the 32 papers mention the time behavior characteristic, it is one of the most frequently commented characteristics. Because of the possibility of doing upfront analysis during code generation, more efficient code can be generated. On the other hand, interpreters add overhead to run-time functionality and thus are slower. While that is the general sentiment, Klint [34] remarked

Table 3. The results of the literature review and basis for the ranking of the two approaches. *G* corresponds with a preference for code generation over interpretation. *I* identifies where a paper shows a preference for interpretation over generation. *G I* indicates papers did not present a preference, but did give advantages or disadvantages.

	ISO: Performance behavior			ISO: Compatibility		ISO: Security		ISO: Maintainability				ISO: Portability	
	Run-time behavior	Build-time behavior	Resource utilization	Co-existence	Interoperability	Confidentiality	Modularity	Analysability	Modifiability	Testability	Adaptability	Installability	
Batouta et al. [7]	G	I	G			G			I		G	G	
Brady and Hammond [22]	G						I	I	I	I			
Cleenewerck [23]							G	G	I				
Consel and Marlet [24]	G	I					I						
Cook et al. [25]	G		G						I			G	
Cordy [26]		I							I				
Czarnecki and Eisenecker [4]	G	I											
Daz et al. [2]	G	I							I	G			
Ertl and Gregg [27]	G	I							I		I		
Fabry et al.[8]	G		G		I				I				
Gaouar et al. [28]	G			I									
Gregg and Ertl [29]	G		G	I				I	I	I	I		
Guana and Stroulia [10]								I	I	I			
Hinkel et al. [30]										I			
Inostroza and Van Der Storm [31]							I						
Jones et al. [32]	G							I	I	G			
Jrges [33]		I		I				I					
Klint [34]	G	I	G					I	I	G	I		
Meijler et al. [6]	G	I	G							G	G	G	
Mernik et al. [17]							I		I	G		I	
Ousterhout [35]	G	I			I					G			
Pessoa et al. [36]										G	I		
Riehle et al. [37]		I											
Romer et al. [38]	G	I											
Schramm et al. [39]		I											
Stahl et al. [21]		I				G		I	I	I			
Sundharam et al. [40]								I		I	I	I	
Tankovi [18]	G	I				G					I	I	
Tankovi et al. [41]	G	I				G					I	I	
Thibault et al. [42]	G								I				
Thibault and Consel [43]		I							I				
Varr et al. [44]	G				I					I			
Voelter [19]	G	I						G	G	G	G	G	
Voelter and Visser [45]	G	I						G	G	G			
Zhu[46]	G	I	G							G			
% in favor of generation	88	0	87.5	0	0	100	20	20	15	55.5	30	50	
% in favor of interpretation	12	100	12.5	100	100	0	80	80	85	44.5	70	50	

that the overhead of interpreters will diminish with the advent in hardware. Both Ertl and Gregg [27] and Romer et al. [38] show that there is nothing that makes interpreters inherently slow.

The reduced build times that an interpretive approach gives are an advantage, such as enabling of agile development and better prototyping. This advantage is stated by Consel and Marlet [24] and Riehle et al. [37] among many others.

Resource Utilization - The general comment that code generation results in improved run-time behavior can be extended to resource utilization as well. Meijler et al. [6] state that generators can optimize for more than run-time behavior only, something that is useful in for instance embedded systems and game environments. A difference can also be seen in how generators or interpreters compete with the running application for resources. A generator might use more memory, but could be running on different hardware than the application. Interpreters are part of the application, so it could be hard to run them on different hardware. Gregg and Ertl [29] comment that interpreters often require less memory, but confirms the competition for resources with the application.

Another view on resource utilization is the data storage for an application. Meijler et al. [6] point out that the interpretive approach often leads to a less optimal data schema. The schema might depend on the model and thus can change at run-time, therefore, the schema has to be flexible enough. This requirement often conflicts with optimizations that might be achievable otherwise.

4.2 ISO: Compatibility

The category *Compatibility* contains characteristics that express the quality of co-existence and operability of the system.

Co-existence - Only two papers contain evidence for a preference between interpretation or generation based on this characteristic. Gaouar et al. [28] share their experiences on making dynamic user interfaces and point out how the interpretive approach enabled them to use platform native elements. A different side is shown in Jörges [33]: the late binding that interpretation offers makes it possible to re-use the same application instance for different tenants.

Interoperability - Interpreters have access to the dynamic context of the application at run-time. Fabry et al. [8], Ousterhout [35], and Varró et al. [44] state this as a preference for interpreters, because it allows them to communicate with the application in a way that is not possible by generators.

4.3 ISO: Security

Security describes the quality in terms of integrity, authentication, and confidentiality. The literature only contained evidence for the characteristic confidentiality.

Confidentiality - Tanković [18] and Tanković et al. [41] describe the models used in a MDEE as intellectual property. The interpretive approach exposes

the model to the application, making it more vulnerable for exposure. In the generative approach the models do not need to be shipped which makes that approach more secure.

4.4 ISO: Maintainability

Maintainability is an important aspect in the quality of software products. Characteristics in this category that were mentioned by literature comment on the testability, modifiability, analysability, and modularity of the platform.

Modularity - Most literature favors interpreters over generation when looking at the modularity characteristic. Inostroza and Van Der Storm [31] and Consel and Marlet [24] propose solutions for modularization within interpreters. Cleenewerck [23] is the only one who argues that generators are more preferred than interpreters when it comes to modularization.

Analysability - An important aspect in MDEEs is the analysis of the resulting application. It should conform to the model and the defined semantics, which is not an easy task. When a generative approach is used, the model is translated in a separate language, without losing the semantics of the model. Proving that translation to be correct is hard, according to Guana and Stroulia [10]. According to Jörges [33], the interpreter can play the role of a reference implementation, used to document the semantics of the model. This improves the analysability of the platform.

Debugging is partly analyzing the run-time behavior of an application. According to Voelter [19] and Voelter and Visser [45] this process is easier in a generative approach, because the generated application can be debugged as if it were a normal application.

Modifiability - Many papers, Cook et al. [25] and Díaz et al. [2] among others, claim that interpreters are easier to write. We conclude that easier to write software is also easier to modify. Cordy [26] describes the process of a compiler as being heavy-weight, making it harder to modify. Cleenewerck [23] and Voelter and Visser [45] argue that generators give more freedom to developers, giving them room for better solutions.

Testability - The literature was far from conclusive on the testability of both approaches. On the one hand, interpreters can be embedded in test frameworks, this makes them easier to test. Generators on the other hand add indirection in the testing, because they are a function from model to code. Asserting the correctness of the output becomes fragile when just looking at the written code, the easiest way is to determine the correctness by running the code. Voelter [19] and Voelter and Visser [45] prefer generation when it comes to debugging, because the model translation can be left out of the testing.

4.5 ISO: Portability

Portability covers the characteristics adaptability and installability.

Adaptability - The separation between generation environment and application environment makes the generative approach preferred according to Meijler et al. [6], Batouta et al. [7], and Voelter [19]. The two environments can be evolved at a different pace when adaptation needed, which makes it more flexible. In an interpretive approach the whole interpreter needs to be rewritten and although this might be easy, it is more work. However, Tanković [18], Tanković et al. [41], and Gregg and Ertl [29] state that porting an interpreter to a new platform is no problem when platform independent technologies (such as programming languages and environments that run on multiple platforms) are used. This matches the results from Sect. 3, where three SPOs stated portability as the rationale for the interpretive approach.

Installability - The two separated environments in the generative approach not only have a clear advantage for adaptability, it is also an advantage with respect to installability. Meijler et al. [6], Cook et al. [25], Batouta et al. [7], and Voelter [19] prefer code generation because it can target any platform, it does not constrain the target application. The initial installation is, however, not all that is important, when the MDEE is updated, re-installations are needed too. The interpretive approach makes re-installations less frequent, because in many cases only the model needs to be updated. This advantage is pointed out by Tanković [18] and Mernik et al. [17].

4.6 Utilizing the Preferences

The results of the literature study as presented in Table 3 can be used by SPOs to design their execution approach. But before SPOs can use these results, they have to prioritize the quality characteristics, i.e., they have to determine which characteristics are most important for them. When priorities are assigned, the preference for either the generative or the interpretive approach can be calculated by the following formulas:

$$P_{generative} = \sum_{i=1}^{12} P_i G_i \quad \text{and} \quad P_{interpretive} = \sum_{i=1}^{12} P_i I_i$$

The formulas summarize over all twelve characteristics i , and applies the priority (P_i) on the corresponding preference (from Table 3) for both the generative (G_i) and the interpretive (I_i) approach. All priorities add up to a total of 1, and because for every characteristic i G_i I_i add up to 100%, $P_{generative}$ and $P_{interpretive}$ add up to 100%. The outcome shows for a certain set of priorities what the preference for either the generative or interpretive approach is.

How the priorities are determined is not prescribed, however, in the case study described in the next Section we will show two possibilities. The first option is by informally giving a weight to every characteristic, dividing 100% among the different characteristics. By doing this informally, the SPO takes the risk of calculating a preference with inaccurate data. Therefore, we also show a second option to prioritize the characteristics: the analytic hierarchy process

(AHP) method described by Saaty [47]. Falessi et al. [48] shows that the AHP method is helpful in protecting against two difficulties that are relevant for this study. The first is a too coarse grained indication of the solution. When the priorities are determined informally it becomes easy to overlook certain characteristics. The second difficulty is that there are many quality attributes that need to be prioritized, and many attributes have small and subtle differences. The AHP method helps by prioritizing in a pairwise manner, the priorities are only determined relative to other characteristics.

5 Case Study

We conducted a case study by observing the design of a MDEE at a Dutch SPO, AFAS Software. The NEXT version of AFAS' ERP software is completely model-driven, cloud-based and tailored for a particular enterprise, based on an ontological model of that enterprise. The ontological enterprise model (OEM, see Schunselaar et al. [49]) will be expressive enough to fully describe the real-world enterprise of virtually any business. The platform initially used a generative approach, generating many lines of C# and JavaScript. However, during the course of 2016 a shift was put into motion towards a hybrid form with more parts being interpreted at run-time. We took part in the discussions surrounding this shift and observed the team while they designed and implemented parts of the MDEE.

We already explained that the context of the MDEE influences the design of the execution approach. This can be seen if we approach the architecture as a set of design decisions as described by Jansen and Bosch [12] and van der Ven et al. [13]. These decisions are made during the software development life cycle. Every requirement is satisfied by first creating one or more solutions, from which the SPO selects the best fitting alternative. This is done by assessing the solutions, for instance in terms of quality, cost, and feasibility. After a solution is selected, the preferred solution is incorporated in the existing architecture. This process is continuous and will be repeated for every new requirement that needs to be satisfied.

The complete architecture of a MDEE is too large to present in this paper, therefore, we present the most important and guiding requirements and decisions. These are presented in two distinct phases, to illustrate two different utilizations of the results from Table 3. The requirements and decisions that form the architecture and are input for the prioritization are summarized in Table 4.

The initial requirement that guided the design of the MDEE is the envisioned target audience for the modeling language (**R1**). By choosing laymen as the target audience, it becomes possible for non-technical business users to model their own ERP solution. This requirement is driven by years of experience in the development of an ERP solution, and the knowledge that is accumulated in those years. The resulting design decision is that the modeling language should be a model with a high level of abstraction, an ontological enterprise model (OEM) (**D1**). This model abstracts from the many details that are needed for

Table 4. Summary of the requirements and decisions from the design of the MDEE.

Requirements
R1 Target audience for the modeling language are laymen
R2 Users do not manage or maintain the MDEE themselves
R3 Cost effectiveness of the MDEE is important
R4 Use a technology that the developers are familiar with
R5 The MDEE should handle the load from the existing customer base
R6 End users can change the model without intervention
Decisions
D1 Develop an ontological enterprise model
D2 Use a SaaS delivery model
D3 Use multi-tenancy to gain resource sharing
D4 The MDEE should run on the .NET runtime
D5 Deploy the MDEE as a distributed application
D6 Use a hybrid execution approach

creating software, those details are added by the platform (the generator or interpreter) when the model is transformed. A second requirement is that the hosting and management of the MDEE is done by the SPO (**R2**). Delivering the MDEE through a Software-as-a-Service (SaaS) model is the second design decision (**D2**) that satisfies requirement **R2**. A third important requirement is cost effectiveness of the MDEE (**R3**), and multi-tenancy is one of the ways of achieving that as stated by Kabbedijk et al. [50]. The decision for a variant of multi-tenancy forms the last important decision (**D3**) of this initial phase.

After the design of the initial architecture, that solved among many other requirements **R1**, **R2**, and **R3**, the execution approach is designed. At the time of this design, the literature study as presented in Sect. 4 was not yet done. After discussion with the team, we concluded and verified that in hindsight four quality characteristics were especially important for this phase of the development. *Run-time time behavior* and *resource utilization* followed from the decision for SaaS (**D2**) and multi-tenancy (**D3**). *Testability* and *analysability* were important for AFAS to secure the quality of the new MDEE. With the data from Table 3 and the priorities that we assigned in hindsight allow us to calculate the preference for an approach. The possible calculation is shown as an illustration. The first two characteristics (*resource utilization* and *run-time time behavior*) are assigned a priority (or weight) of 35%, the other 30% is split between the other two characteristics (*testability* and *analysability*). The resulting preferences can then be calculated by combining the priorities of the characteristics with their weights (expressed in percentages, summing up to a total of 100%). We apply formulas $P_{generative}$ and $P_{interpretive}$ on the percentages from Table 3 and the priorities, resulting in the following calculations:

$$P_{generative} = 0.35 * 0.88 + 0.35 * 0.875 + 0.15 * 0.55.5 + 0.15 * 0.20 = 0.729$$

$$P_{interpretive} = 0.35 * 0.12 + 0.35 * 0.125 + 0.15 * 0.44.5 + 0.15 * 0.80 = 0.271$$

The outcome of the calculation matches the decision that AFAS made: their initial execution approach was the generative approach. This initial phase of requirements, decision making, and design of the architecture can be summarized in three statements.

- **R1** leads to **D1**
- **R2** in the context of **D1** leads to **D2**
- **R3** in the context of **D1** and **D2** leads to **D3**

As the design of the MDEE advanced new requirements needed to be realized. First of all the technology that is used to develop the MDEE was selected. The requirement was that a technology should be used that is familiar to the development team (**R4**). This fourth requirement led to the decision for the .NET runtime (**D4**) as the technology to develop the platform on. The next requirement formulated expected load requirements: AFAS has a large existing customer base that needs to be transferred to this new platform. There is an expected load known from the existing customer base that needs to be handled (**R5**). As a result of this requirement, the decision was made to design and deploy the application as a distributed system (**D5**).

Table 5. Summary of the priorities of quality characteristics determined by applying AHP as described by Saaty [47]. Columns *Generative* and *Interpretive* show the preferences for code generation and model interpretation from Table 3. The final preferences are calculated with the formulas $P_{generative}$ and $P_{interpretive}$.

	Priority	Generative	Interpretive
Run-time time behavior	0.059	0.88	0.12
Build-time time behavior	0.278	0.00	1.00
Resource utilization	0.098	0.875	0.125
Co-existence	0.045	0.00	1.00
Interoperability	0.012	0.00	1.00
Confidentiality	0.012	1.00	0.00
Modularity	0.062	0.20	0.80
Analysability	0.023	0.20	0.80
Modifiability	0.150	0.15	0.85
Testability	0.085	0.555	0.445
Adaptability	0.155	0.30	0.70
Installability	0.021	0.50	0.50
Preference		0.293	0.707

The sixth requirement reopened the design of the model execution approach. Therefore, the team decided to backtrack on the earlier decision for the generative approach. AFAS envisioned that customers are able to customize the model without intervention from AFAS (**R6**). This requirement leads to other requirements, such as the expected turn around time between model changes and application updates. Based on requirement **R6** and the decisions **D1-D5** the quality characteristics were prioritized. Characteristics *build-time time behavior*, *adaptability*, and *modifiability* became more important. This time the prioritization was done by applying the AHP method: all the characteristics were pair-wise compared and ranked according to the method described by Saaty [47]. The results are shown in Table 5, combined with the preferences from Table 3. The final outcome preferred interpretation over generation with 71%.

The team decided to implement a simplification approach: the OEM is simplified into a simpler model by the generator. This way the team was able to satisfy the build-time time requirements, without sacrificing performance. Because the MDEE itself already grew quite large, the team decided to also switch to a mix-and-match approach. The simplification approach was first implemented in a specific component: the messages that are past between the different parts of the distributed system.

An architecture consists of many decisions, both large and small, both important and non-essential. Our case study only shows the five most important requirements. In the next Section we will reflect on the case study and derive a proposed decision support framework for the design of a model execution approach.

6 Case Study Reflection

In Sect. 5 we observed a SPO during the design of a MDEE. We have shown how design decisions from the architecture determine the priorities of the quality characteristics. The existing architecture of the MDEE and the design decisions that are present together form the context of the model execution approach. It shows that, just as with any component in a larger system, the design of an execution approach does not stand on its own, but needs to be embedded in the overall architecture. Some design decisions might constrain the execution approach, other design decisions might even mitigate the problems that an execution approach give. As an example we look at build-time time behavior, a requirement that was described in the previous Section. From Table 3 we learn that the interpretive approach is preferred when a specific build-time time behavior is required. However, when the MDEE will be built using a programming language and platform that uses interpretation, such as JavaScript, the decrease in build times with a generative approach might be mitigated. An interpreted language does not need a separate compile step that needs to be executed by the generator, and that reduces the build time. This shows that the design decisions that are already present influence the execution approach.

We have distilled three areas from the decisions described in Sect. 5 that steered the design of the model execution approach. The decisions described in Sect. 5, and summarized in Table 4 are used to illustrate the areas.

6.1 The Metamodel

The metamodel and its features and requirements have an influence on the most fitting model execution approach. This is illustrated by decision **D1: OEM** and requirement **R6: Customize the model**.

A model with a high-level of abstraction (such as **D1: OEM**) will require a more complex model execution, because the distance in terms of abstraction between a programming language and the model is larger. With an interpretive approach, the application will require more resources to perform this model execution. This influences the run-time behavior of the model execution approach, and thus the application itself.

On the other hand, requirement **R6: Customize the model** increases the priority of the build-time time behavior characteristic. This leads to a preference for run-time interpretation, because that approach is preferred if build-time time behavior is important.

6.2 The Architecture

The chosen architecture for the application forms a second area of influence on the most fitting model execution approach. A multi-tenant, distributed application (as defined by **D3: Multi-tenancy** and **D5: Distributed application**) can result in conflicting requirements for the most fitting model execution approach.

On the one hand, multi-tenancy prefers interpretation, because it allows the sharing of a single application instance for multiple tenants (see characteristic co-existence in Sect. 4). This maximizes the resource sharing, and enables fast unloading and loading of changes, which decreases the build times. On the other hand, a distributed application might not benefit from interpretation, because every process has to do the interpretation. Figure 2 shows that the interpretation process is part of the application, and is thus duplicated when the application is separated in multiple components and processes. This adds of course resource utilization to the platform.

The decision for a distributed application (**D5: Distributed application**), makes it possible to design a hybrid model execution approach. A distributed application consists of different (distributed) components that can use their own execution approach, shown in Sect. 5 where only the messages were re-designed.

6.3 The Platform

Although Kelly and Tolvanen [5] make no distinction between the architecture, framework, the operating system, or the runtime environment, we see a different influence from the operating system or runtime environment. As decision **D4: .NET platform** illustrates, the lack of support for dynamic software

updating requires a different model execution approach to satisfy the requested build-time behavior. This matches the approaches of Meijler et al. [6] with their customized Java class loader and Czarnecki and Eisenecker [4] using the extension object pattern.

The SaaS delivery model (**D2: SaaS delivery model**) removes most of the problems around *installability* and *co-existence*: the platform is controlled by the SPO.

6.4 The Decision Support Framework

From the observations we see three distinct areas that influence the model execution approach. The metamodel and its features and requirements lead to decisions that influence the execution approach. The architecture and the platform can both constrain the execution approach as well as mitigate challenges. Determining the priorities for the quality characteristics can be a difficult task.

The design of the best fitting model execution approach for a MDEE is not different from other parts of the MDEE; it is not possible without knowledge of the context. The description of the design process that we gave in Sect. 5 is generic for the software development life cycle. We propose, based on the observations made during the case study, a tailored version of the process for the design of a model execution approach (shown in Fig. 3). It shows that the current architecture is input for the prioritization of the quality characteristics. The priorities can then be used to assess the possible execution approaches. How the priorities are determined is not prescribed by the framework, however, we have shown two possible methods to determine them: an informal method and the AHP method.

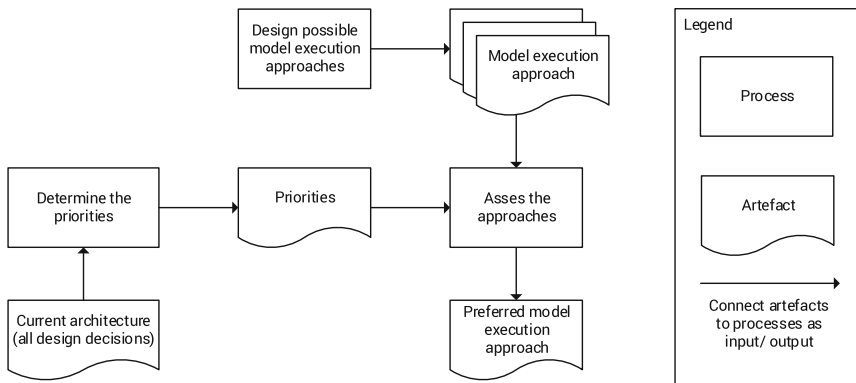


Fig. 3. The process of selecting a best fitting model execution approach. The process starts with the design of possible execution approaches. The current architecture is the input for the prioritization of quality characteristics. The priorities can be used in assessing possible execution approaches.

The framework offers guidance for SPOs in the design of their model execution approach. By formalizing their architecture in a set of design decisions, and by prioritizing the quality characteristics, SPOs can calculate the preference for either the generative or the interpretive approach. This can then in turn be used to design a fitting hybrid model execution approach.

7 Discussion

The validity of our research is threatened by several factors. The internal validity of our study is threatened because the correlation between quality characteristics on the one hand and the execution approach on the other hand are not straightforward. The claims in the reviewed literature, however, do show a convergence towards each other. Some characteristics lack a significant number of references, making them volatile. However, we regard the claims that are made not as controversial, but in line with existing research. The data that we found in literature consists of anecdotal argumentation, based on the experience of the authors. The claims that were made, were not validated and not supported with empirical evidence. To create a more trustworthy decision support framework, the data presented in Table 3 should be validated by empirical research. Experiments or large case studies should provide more quantitative data on the fulfillment of the different quality characteristics.

The construct validity of our case study is threatened by the fact that one of the authors is involved in the object of the study, resulting in a possible bias in our observations. However, the observations were made during a period of several months in which the model execution was actively designed. Our observations were reviewed and commented on by other team members involved. The descriptions of the observations, and the described requirements and decisions were correctly described according to these comments.

The external validity of our research is threatened because our case study is done at a single company. The observations, however, were done over an extensive period of time, and the results were discussed with the team. We argue that the conclusions and observations are in line with existing literature. The decision support framework, however, should be further strengthened by additional case studies.

8 Conclusion

We present two contributions to the research on MDD, and in particular the development of MDEEs. The survey in Sect. 3 illustrates that there is a lack of guidance and knowledge for SPOs. Although the SPOs show that indeed many forms of model execution approaches are used, they do not have an explicit rationale for their design.

In Sect. 4 we studied and summarized existing literature to correlate quality characteristics with model execution approach. Although this knowledge was

already available, it was scattered over many papers. Our study makes the experience and knowledge of many authors available to MDD researchers and practitioners. We summarized the results in Table 3, which can be used as a reference in the design of a fitting model execution approach. In Sect. 5 we demonstrate how these results can be used as input for the decision making in selecting alternatives.

The second contribution that we present is the decision support framework as presented in Sect. 6. With this framework, SPOs have a structured process for the design of the model execution approach. By making these design decisions explicit, and by adding the results from Table 3 as input to the decision making process, SPOs can design the best fitting execution approach. The influence of the context of the MDEE as shown in Sect. 6, and the interplay between existing design decisions and the model execution approaches is made explicit and can lead to better designs.

Although we are not able to relieve SPOs from the hard work of designing a model-driven engineering environment, we argue that our research brings them closer to the best fitting design. By making existing knowledge and experience accessible, the solutions in the decision making process can be assessed with more confidence. In Sect. 3 we show that many SPOs already use a hybrid form of model execution, but do not have a strong rationale. However, our research also uncovers the need for more empirical research to support SPOs in the design and development of MDEEs. Table 3 is primarily based on anecdotes, and often not backed by real evidence. Experiments and case studies should be conducted to strengthen the evidence used in our decision support framework. The framework itself is created by observing a single SPO designing a model execution approach, and it should be evaluated by applying it at other SPOs.

Many questions in the design of software can be answered with “it depends”, leaving the questioner puzzled as to what he should do. We present how the context of the MDEE influences the design of a model execution approach for MDEEs. Existing design decisions determine the priorities of quality characteristics, which in turn steer the design of the model execution approach. We also show how SPOs can utilize the knowledge presented in this paper to allow them to steer their design process towards the most fitting model execution approach.

Acknowledgements. This research was supported by the NWO AMUSE project (628.006.001): a collaboration between Vrije Universiteit Amsterdam, Utrecht University, and AFAS Software in the Netherlands. The NEXT Platform is developed and maintained by AFAS Software. Further more, the authors like to thank Jurgen Vinju, Tijs van der Storm, and their colleagues for their feedback and knowledge early on in the writing process. Finally we thank the team at AFAS Software for their opinions, feedback, and reviews.

References

1. Overeem, M., Jansen, S.: An exploration of the ‘It’ in ‘It Depends’: generative versus interpretive model-driven development. In: 5th International Conference on Model-Driven Engineering and Software Development, MODELSWARD (2017)

2. Díaz, V.G., Valdez, E.R.N., Espada, J.P., Bustelo, B.C.P.G., Lovelle, J.M.C., Marín, C.E.M.: A brief introduction to model-driven engineering. *Tecnura* **18**, 127–142 (2014)
3. Brown, A.W.: *An Introduction to Model Driven Architecture*. The Rational Edge, pp. 1–16 (2004)
4. Czarnecki, K., Eisenecker, U.W.: *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional, Boston (2000)
5. Kelly, S., Tolvanen, J.P.: *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley, Hoboken (2008)
6. Meijler, T.D., Nyttun, J.P., Prinz, A., Wortmann, H.: Supporting fine-grained generative model-driven evolution. *Softw. Syst. Model.* **9**(3), 403–424 (2010)
7. Batouta, Z.I., Dehbi, R., Talea, M., Hajoui, O.: Multi-criteria analysis and advanced comparative study between automatic generation approaches in software engineering. *J. Theor. Appl. Inf. Technol.* **81**, 609–620 (2015)
8. Fabry, J., Dinkelaker, T., Noye, J., Tanter, E.: A taxonomy of domain-specific aspect languages. *ACM Comput. Surv.* **47**, 1–44 (2015)
9. Zhu, L., Aurum, A., Gorton, I., Jeffery, R.: Tradeoff and sensitivity analysis in software architecture evaluation using analytic hierarchy process. *Softw. Qual. J.* **13**(4), 357–375 (2005)
10. Guana, V., Stroulia, E.: How do developers solve software-engineering tasks on model-based code generators? An empirical study design. In: *First International Workshop on Human Factors in Modeling* (2015)
11. Capilla, R., Rey, U., Carlos, J., Dueñas, J.C., Madrid, U.P.D.: The decision view's role in software architecture practice. *IEEE Softw.* **26**(2), 36–43 (2009)
12. Jansen, A., Bosch, J.: Software architecture as a set of architectural design decisions. In: *5th Working IEEE/IFIP Conference on Software Architecture (WICSA 2005)*, pp. 109–120 (2005)
13. van der Ven, J.S., Jansen, A.G.J., Nijhuis, J.A.G., Bosch, J.: Design decisions: the bridge between rationale and architecture. In: Dutoit, A.H., McCall, R., Mistrík, I., Paech, B. (eds.) *Rationale Management in Software Engineering*, pp. 329–348. Springer, Heidelberg (2006). https://doi.org/10.1007/978-3-540-30998-7_16
14. Svahnberg, M., Wohlin, C., Lundberg, L., Mattsson, M.: A quality-driven decision-support method for identifying software architecture candidates. *Int. J. Softw. Eng. Knowl. Eng.* **13**, 547–573 (2003)
15. Wohlin, C.: Guidelines for snowballing in systematic literature studies and a replication in software engineering. In: *18th International Conference on Evaluation and Assessment in Software Engineering (EASE 2014)*, pp. 1–10 (2014)
16. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: an annotated bibliography. *ACM SIGPLAN Not.* **35**, 26–36 (2000)
17. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. *ACM Comput. Surv.* **37**, 316–344 (2005)
18. Tanković, N.: *Model driven development approaches: comparison and opportunities*. Technical report (2011)
19. Voelter, M.: Best practices for DSLs and model-driven software development. *J. Object Technol.* **8**, 79–102 (2009)
20. ISO: *ISO/IEC 25010:2011 Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models*. Standard, International Organization for Standardization, Geneva, CH (2011)
21. Stahl, T., Völter, M., Bettin, J., Haase, A., Helsen, S.: *Model-Driven Software Development: Technology, Engineering, Management* (2006)

22. Brady, E.C., Hammond, K.: Scrapping your inefficient engine. *ACM SIGPLAN Not.* **45**, 297 (2010)
23. Cleenewerck, T.: Modularizing language constructs: a reflective approach. Ph.D. thesis (2007)
24. Consel, C., Marlet, R.: Architecturing software using a methodology for language development. *Princ. Declar. Program.* **1490**, 170–194 (1998)
25. Cook, W.R., Delaware, B., Finsterbusch, T., Ibrahim, A., Wiedermann, B.: Model transformation by partial evaluation of model interpreters. Technical report (2008)
26. Cordy, J.R.: TXL - a language for programming language tools and applications. In: *Proceedings of the ACM 4th International Workshop on Language Descriptions, Tools and Applications*, pp. 1–27 (2004)
27. Ertl, M.A., Gregg, D.: The structure and performance of efficient interpreters. *J. Instr.-Level Parallelism* **5**, 1–25 (2003)
28. Gaouar, L., Benamar, A., Bendimerad, F.T.: Model driven approaches to cross platform mobile development. In: *Proceedings of the International Conference on Intelligent Information Processing, Security and Advanced Communication*, pp. 19:1–19:15 (2015)
29. Gregg, D., Ertl, M.A.: A language and tool for generating efficient virtual machine interpreters. In: Lengauer, C., Batory, D., Consel, C., Odersky, M. (eds.) *Domain-Specific Program Generation*. LNCS, vol. 3016, pp. 196–215. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-25935-0_12
30. Hinkel, G., Denninger, O., Krach, S., Groenda, H.: Experiences with model-driven engineering in neurorobotics. In: Wařowski, A., Lönn, H. (eds.) *ECMFA 2016*. LNCS, vol. 9764, pp. 217–228. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-42061-5_14
31. Inostroza, P., Van Der Storm, T.: Modular interpreters for the masses implicit context propagation using object algebras. *ACM SIGPLAN Not.* **51**(3), 171–180 (2015)
32. Jones, N.D., Gomard, C.K., Sestoft, P.: *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International (1993)
33. Jörges, S.: *Construction and Evolution of Code Generators: A Model-Driven and Service-Oriented Approach*, vol. 7747. Springer, Heidelberg (2013). <https://doi.org/10.1007/978-3-642-36127-2>
34. Klint, P.: Interpretation techniques. *Softw.: Pract. Exp.* **11**, 963–973 (1981)
35. Ousterhout, J.K.: Scripting: higher-level programming for the 21st century. *Computer* **31**, 23–30 (1998)
36. Pessoa, L., Fernandes, P., Castro, T., Alves, V., Rodrigues, G.N., Carvalho, H.: Building reliable and maintainable dynamic software product lines: an investigation in the body sensor network domain. *Inf. Softw. Technol.* **86**, 54–70 (2017)
37. Riehle, D., Fraleigh, S., Bucka-Lassen, D., Omorogbe, N.: The architecture of a UML virtual machine. In: *International Conference on Object Oriented Programming Systems Languages and Applications (OOSPLA)*, pp. 327–341 (2001)
38. Romer, T.H., Lee, D., Voelker, G.M., Wolman, A., Wong, W.A., Baer, J.L., Ber-shad, B.N., Levy, H.M.: The structure and performance of interpreters. *ACM SIGPLAN Not.* **31**, 150–159 (1996)
39. Schramm, A., Preußner, A., Heinrich, M., Vogel, L.: Rapid UI development for enterprise applications: combining manual and model-driven techniques. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) *MODELS 2010*. LNCS, vol. 6394, pp. 271–285. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16145-2_19

40. Sundharam, S.M., Altmeyer, S., Navet, N.: Model interpretation for an AUTOSAR compliant engine control function. In: 7th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS) (2016)
41. Tanković, N., Vukotić, D., Žagar, M.: Rethinking model driven development: analysis and opportunities. In: Proceedings of the ITI 2012 34th International Conference on Information Technology Interfaces (ITI), pp. 505–510 (2012)
42. Thibault, S.A., Marlet, R., Consel, C.: Domain-specific languages: from design to implementation application to video device drivers generation. *IEEE Trans. Softw. Eng.* **25**, 363–377 (1999)
43. Thibault, S., Consel, C.: A framework for application generator design. *ACM SIGSOFT Softw. Eng. Notes* **22**, 131–135 (1997)
44. Varró, G., Anjorin, A., Schürr, A.: Unification of compiled and interpreter-based pattern matching techniques. In: Vallecillo, A., Tolvanen, J.-P., Kindler, E., Störrle, H., Kolovos, D. (eds.) *ECMFA 2012*. LNCS, vol. 7349, pp. 368–383. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31491-9_28
45. Voelter, M., Visser, E.: Product line engineering using domain-specific languages. In: 15th International Software Product Line Conference, pp. 70–79 (2011)
46. Zhu, M.: Model-driven game development addressing architectural diversity and game engine-integration. Ph.D. thesis (2014)
47. Saaty, T.: How to make a decision: the analytic hierarchy process. *Eur. J. Oper. Res.* **48**, 9–26 (1990)
48. Falessi, D., Cantone, G., Kazman, R., Kruchten, P.: Decision-making techniques for software architecture design. *ACM Comput. Surv.* **43**, 1–28 (2011)
49. Schunselaar, D.M.M., Gulden, J., Schuur, H.V.D., Reijers, H.A.: A systematic evaluation of enterprise modelling approaches on their applicability to automatically generate software. In: 18th IEEE Conference on Business Informatics, pp. 290–299 (2016)
50. Kabbedijk, J., Bezemer, C.P., Jansen, S., Zaidman, A.: Defining multi-tenancy: a systematic mapping study on the academic and the industrial perspective. *J. Syst. Softw.* **100**, 139–148 (2015)



Applying Integrated Domain-Specific Modeling for Multi-concerns Development of Complex Systems

Reinhard Pröll, Adrian Rumpold^(✉), and Bernhard Bauer

Institute for Software and Systems Engineering,
University of Augsburg, Augsburg, Germany
{reinhard.proell, adrian.rumpold, bauer}@informatik.uni-augsburg.de

Abstract. Current systems engineering efforts are increasingly driven by trade-offs and limitations imposed by multiple factors: Growing product complexity as well as stricter regulatory requirements in domains such as automotive or aviation necessitate advanced design and development methods. At the core of these influencing factors lies a consideration of competing non-functional concerns, such as safety and reliability, performance, and the fulfillment of quality requirements. In an attempt to cope with these aspects, incremental evolution of model-based engineering practice has produced heterogeneous tool environments without proper integration and exchange of design artifacts. In order to overcome these shortcomings of current engineering practice, we propose a holistic, model-based architecture and analysis framework for seamless design, analysis, and evolution of integrated system models. We describe how heterogeneous domain-specific modeling languages can be embedded into a common general-purpose model in order to facilitate the integration between previously disjoint design artifacts. A case study demonstrates the suitability of this methodology for the design of a safety-critical embedded system, a hypothetical gas heating, with respect to reliability engineering and further quality assurance activities.

Keywords: Domain-specific modeling · Model transformation
Model-based analysis · Model-based testing

1 Introduction

A clear trend towards increasing complexity is visible in modern embedded systems, both with respect to hardware and software. This development is fueled by a variety of factors, with one major driver being the advent of stricter regulatory guidelines in diverse domains such as automotive (with the ISO 26262 standard), aviation (ARP4754A and DO-178C/DO-254), and industrial automation (IEC 61508, IEC 61511, among others).

Two major strategies have emerged that attempt to let system designers cope with this rise in product complexity:

More stringent engineering methodologies, notably model-based techniques, are becoming essential for the design of complex systems. However, most existing model-based methods place disproportionate focus on functional requirements, mostly disregarding non-functional and quality aspects, such as reliability, safety, and security.

At the same time, tooling vendors have provided a sizable amount of products for the analysis and management of non-functional engineering concerns (compare Chap. 4.1.2 of [1]). No clear strategy exists as to how these heterogeneous tools can be integrated in a seamless workflow, in order to make their information base available throughout the entire product life cycle.

Due to these shortcomings caused by partial adoption of model-based techniques and inconsistent tooling environments, establishing traceability and consequent change management have emerged as two main challenges in systems engineering. The importance of these concerns can be seen clearly in the context of safety-critical systems: Here, regulatory standards and norms necessitate careful management of development processes and artifacts with respect to consistent traceability throughout the product life cycle. Non-compliance with these requirements may pose a significant financial risk (in the form of late changes required to attain safety certification) as well as a liability hazard for the manufacturer.

A similar argument holds for quality assurance activities during the development of such complex systems. Here, focus lies on a high degree of test coverage – some safety standards even mandate specific coverage requirements (e.g. the aviation norm DO-178C). The resulting need for careful manual review and management of traceability and consistency leads to sub-optimal process efficiency and ultimately a potential negative impact on product quality.

Problem Statement

Despite the advantages that stem from the use of state-of-the-art model-based engineering practice, a tighter integration between techniques and tools for functional and quality aspects is needed in order to conquer the difficulties of ever-increasing product complexity.

Some effort has been made towards artifact exchange between model-based engineering tools, e.g. through standardized interchange formats like XMI. However, the vision of truly seamless tool integration remains a fundamental challenge. The resulting need for manual process steps can delay quality-related design activities and consequently reduce overall product quality. As a result, quality defects discovered late in the development process drive costs and pose a hazard to timely product release (see [2]).

A consistent seamless design methodology is crucial when considering process artifacts such as documentation required for certification of safety-critical systems. It is immediately evident that consistency between the actual product and its supporting artifacts is of crucial importance. However, although common modeling tools allow for generation of technical documentation from system

models, the generation of more complex textual artifacts exceeds their limited capabilities.

In order to overcome the identified weaknesses we propose an approach which aims for a tight integration of all system modeling artifacts and a shift towards (semi-)automated integrated architecture analyses.

Based on an extensible set of domain-specific modeling languages, which make up a solid foundation for a more suitable description of quality aspects, we aim for a co-evolution of functional and quality architectures of the system under development. These modeling languages cover the domains of common non-functional requirements for embedded systems, for instance safety, reliability, and system integration. Further, we describe a reliable mechanism for quality assurance of systems developed using such heterogeneous modeling languages. Our approach aims to reuse existing design methodologies, as long as they generate artifacts that adhere to a formalized metamodel.

The model-level integration of multiple domain-specific aspects additionally enables developers to generate purpose-specific data from the system model, which offers the necessary flexibility for the development of complex embedded systems.

We foresee that this integrated modeling approach will lead to increased product quality and can thus support the development of safety-critical and similarly regulated systems.

Outline

This article is an extended and revised version of our earlier conference paper [3].

As a starting point, Sect. 2 introduces the modeling concepts underlying our approach and describes their application in analysis scenarios within our proposed framework. Starting with general-purpose modeling languages, which are actively maintained by the system engineer, we describe a set of essential domain-specific views on the system and their embedding into the general-purpose language. Based on this definition of embedded domain-specific languages, we propose a model-based analysis framework in Sect. 3, providing some insight into its technical background and implementation. In Sect. 4, we demonstrate the feasibility of our approach using a realistic use case. There, we perform some exemplary design and analysis steps utilizing the previously introduced framework. Section 5 discusses related work regarding the integration of heterogeneous modeling tools, domain-specific modeling, and model-based analysis. Section 6 summarizes the key results of this paper and briefly outlines future applications extending our research.

2 A Domain-Aware Modeling Approach for Embedded System Engineering

To overcome the challenges identified in the introduction, we have developed a concept designed to integrate legacy development and modeling techniques

with a new kind of domain-aware modeling approach and analysis framework. Based on the information embedded in an integrated system model, purpose-specific artifacts (e.g. certification- or test-related documentation), which had to be maintained manually before, can now be generated automatically. In order to switch between these representations and generate documents, we make use of model-to-model (M2M) and model-to-text (M2T), As a special case, we consider *x-to-code* (X2C) transformations, where *X* may stand for *text* (T2C) or *model* (M2C).

The high-level concepts and their relationships are illustrated in Fig. 1 and will be elaborated in the following sections.

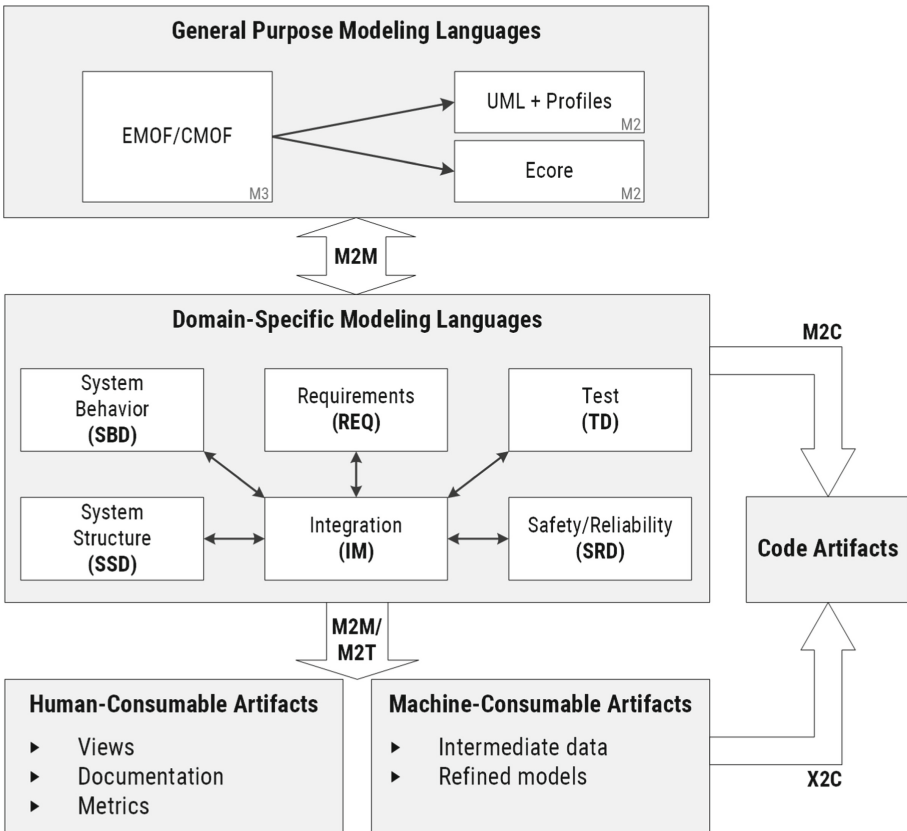


Fig. 1. Conceptual overview of the domain-aware integrated system modeling approach.

2.1 General Purpose Modeling Languages

Following our goal of easy application and seamless integration into state-of-the-art development processes, we have decided to embed all relevant data for the development process within a *General Purpose Modeling Language* (GPML), such as UML or Ecore.

Using such modeling languages improves the applicability of our presented approach: GPMLs are widely accepted as state of the art, with many practitioners being familiar with their proper use. This familiarity allows for easier and faster adoption of new approaches based on general-purpose modeling languages. On the other hand, the general applicability of GPMLs has created a huge variety of available CASE tools for creating and viewing models. This rich tool environment can be reused within our newly proposed methodology, rather than developing yet another immature modeling tool.

In our scenario, these general-purpose languages serve a two-fold purpose: First, they provide a common modeling basis for all domain-specific models, as described in the following section. Second, the GPML can itself be used to cover certain subsets of the domain-specific modeling disciplines, if their expressive power is sufficient for a specific use case. We will see an example for this simplified domain modeling in the case study in Sect. 4, where UML component diagrams and state machines are used to describe parts of the system architecture. Similarly, an extended version of the native UML activity chart is used for modeling of functional test models.

Our approach does not prescribe a certain GPML to be used for modeling the integrated system model. The only necessary requirement is the possibility to enhance the general-purpose language with metamodel extensions. In the case of UML this is achieved by defining profiles that leverage the stereotype mechanism. Similarly, we can extend the expressive capabilities of modeling languages which are themselves specified as UML profiles, for example SysML. Within the widely popular Eclipse Modeling Framework (EMF), metamodel extensions can be easily defined due to the reflexivity of the Ecore modeling language, which itself is its own meta-model.

2.2 Domain-Specific Modeling Languages

In order to accurately describe domain-specific aspects of the system under development, we embed them into the GPML mentioned above as *Domain-Specific Modeling Languages* (DSML). Our approach allows for any number of DSMLs to be used in conjunction with a general-purpose modeling tool to obtain an *integrated system model* (ISM) or *Omni model*.

These DSMLs preserve the separation of concerns, but enable developers to link information across domains in order to build up a holistic view of the system under development (SUD) and facilitate analyses based on domain-specific information. By using M2M transformations between the GPML and DSML representations, the distinct components of the ISM are in sync throughout the development process, yielding the best of both worlds.

In this section we briefly introduce some common domains pertinent to development of embedded systems and their focus. This lays the conceptual foundation for the following sections that will provide additional detail and demonstrate the application of these concepts.

The Requirements Domain (RQD) is related to the foremost engineering tasks of every modern software development process. As a result of these tasks, a set of requirements is extracted, which describes the desired system from a functional as well as a quality perspective.

Depending on the role of requirement specifications in the development process, an appropriate way of serialization must be chosen. In the early days of requirements engineering, Roman [4] pointed out its importance and already identified the need for knowledge integration.

In order to further make use of the generated set of requirements, a certain DSML needs to be specified. Natural language requirements with additional structuring capabilities as well as fully machine-processable requirement models are thinkable. For example, a ReqIF-based DSML (see [5]) can be used with most common CASE tools.

More advanced efforts propose ontologies as a suitable way of specifying requirements (e. g. [6]), which are also compatible with the basic concepts underlying our framework.

Being able to reference specific requirements in a model or parts of them, enables developers to use this semi-formal specification of the system for cross-domain traceability, thus extending the information base. For example, the availability of requirements information for a test engineer may result in a more transparent and effective test of the system under development. We give an example for this beneficial interaction in our case study in Sect. 4.

The System Structure Domain (SSD) contains the structural model of the system under development and reflects the architectural decomposition of the solution.

Our approach allows for a high degree of freedom regarding the actual implementation of the SSD model. For simple projects, the underlying GPML (see Sect. 2.1) itself may be sufficiently expressive to model the system structure without any domain-specific additions. For more complex systems, a modeling language with more powerful abstractions, such as SysML, can be integrated to describe the structural domain more adequately.

It should be noted that the SSD model may also be derived from a prior system description in case of a brown-field project. Here, it is feasible to use either existing architecture models as a basis for the newly defined integrated system model, or to reverse engineer a system description from its code artifacts.

The System Behavior Domain (SBD) contains models that describe the functional behavior of the system under development. As described above for the SSD, a range of modeling languages can be used to implement the behavioral

domain within our approach. Natural choices are the behavior diagrams found in the Unified Modeling Language or its SysML extension.

If further usage includes the simulation or any kind of abstract interpretation of the behavioral parts of the system, an executable variant of UML, like *Foundational UML* (fUML, [7]), may provide suitable types of model artifacts.

However, different domain-specific modeling languages might be more familiar to designers of certain embedded systems; one example is the Function Block Diagram (FBD) notation for programmable logic controllers defined in the IEC 61131-3 and IEC 61499 international standards.

Given a suitable technology integration bridge (e.g. OSLC or ModelBus), it is conceivable to integrate behavioral models from widely used simulation tools like Simulink or Stateflow, as an intermediate step during re-engineering of legacy systems.

The Test Domain (TD) reflects the information specific to a tester's viewpoint on the system under development.

On the one hand, it is used to simply formalize artifacts related to traditional quality assurance activities, such as test plans, test cases, and test execution reports. Depending on the expressiveness of the modeling languages used for the system description in the SSD and SBD, the test domain can sometimes be seen as an extension of these domains.

On the other hand, the benefit of a separate test domain can only fully be appreciated within a strict Model-Based Testing (MBT) approach. The main difference in this scenario is the purpose of the model artifacts: The artifacts related to a traditional testing process often represent the intermediate results of mostly manual, atomic steps. MBT in contrast, attempts to reduce the number of manual steps and the amount implicit knowledge in testing, thereby raising the efficiency, correctness, and reusability of artifacts. In this case it is conceivable to embed model languages specific to the testing domain, such as the OMG-maintained UML Testing Profile (UTP), which provides modeling facilities for test behavior description as well as quality assurance management activities. The UTP-affiliated Testing and Test Control Language (TTCN-3) may complete the palette of DSMLs of this domain in order to improve testing.

These considerations demonstrate only one possible solution and choice of DSMLs for the test domain – many others are conceivable, depending on the concrete use case. A possibly more intuitive solution is given by GPML-based test behavior specification via UML activity charts representing a set of test cases, i.e. a test suite. Once again, hybrids of the solutions mentioned above pose viable solutions for this domain and again encourage the use of our overall integrative approach.

Depending on whether test cases are generated or implemented manually, data specified or generated by other domain specific models, e.g. safety considerations, may guide this process. Further information on this topic can be found in Sect. 2.2.

The Safety and Reliability Domain (SRD) covers the modeling and analysis of system reliability. Such analyses are invaluable and often mandated by regulations to demonstrate the system's expected failure behavior and obtain measures of reliability and availability, for example for safety-critical systems.

In order to quantify the reliability of a system, a thorough analysis of potential hazards and their associated risks is required. These hazard analyses require profound domain knowledge and experience and are therefore frequently performed as team efforts. Despite the interactive nature of these activities, their results can be formalized as a hazard model that describes identified hazards and the risks as well as possible faults and failures that can cause these hazardous events.

A major task in the design of safety-critical systems is the classification of hazards based on their associated risk. Risks that are deemed intolerable, either by societal or regulatory standards, have to be mitigated by deliberate risk reduction measures. Based on the necessary level of risk reduction, levels of safety integrity and associated safety requirements can be allocated to protective system components (*safety functions* in the terminology of the functional safety norm IEC 61508). This SIL allocation process requires the quantitative analysis of failure occurrence likelihoods.

Traditionally, quantitative reliability models are maintained in separate tool environments, decoupled from the actual system model. This disjoint setup can lead to inconsistencies in reliability models and decisions made based on them, unless proper care is taken during ongoing development of the system. However, many traditional reliability approaches can easily be adapted for use in model-based environments. For example, the widely used Fault Tree Analysis (FTA) technique defines a set of graphical elements to analyze failure causes in a system [8], and proves a suitable candidate for a domain-specific modeling language with a familiar graphical representation.

By embedding the reliability and hazard analysis models into the integrated system model, our approach allows to easily maintain full traceability between these models and their associated system model counterparts in the SSD and SBD. Moreover, change impact analyses can be easily performed based on this traceability information, whenever a modification to any part of the system model is made.

In the context of model-based systems engineering, it makes sense to move beyond the traditional FTA technique and incorporate a component-based extension, such as the Component Fault Trees as proposed in [9]. This hierarchical structuring of reliability information creates synergies with the end-to-end traceability provided by our modeling approach.

The Integration Model Domain (IMD), as illustrated in Fig. 1, embodies the central mechanism to establish domain-specific model linking, mapping of artifacts, and cross-domain data accessibility.

In order to achieve this ambitious goal, its high-level structure represents an abstract, hierarchical breakdown of the instantiated system in a component-like fashion. Based on this abstract structure, cross-domain linking, represented

as bidirectional connectors in the model, on the one hand enables developers to make use of a solid and consistent traceability mechanism applicable throughout all development phases. On the other hand, the IM provides additional information to e.g. improve test related activities, previously out of scope. This holds for various combinations of domain-specific model data. Note that the IMD does not duplicate any information that has already been modeled in one of its connected domains.

Beside the linking and description of model interfaces, the IM holds analysis results generated by any kind of analysis executed by our proposed framework (see Sect. 3 for a description of the framework). These results may represent the basis for ongoing processing steps, e.g. the scoping of a certain test model part, based on a set of criteria to be met.

In addition to the functionality of the IM presented above, it also plays the role of a early phase design artifact, reflecting an abstract decomposition of the proposed system functionality. For this reason, the IM may undergo constant change until it is connected to a concrete instantiation of SSD and SBD models and subsequently linked with other participating domain models.

Other Domains. The above modeling domains cover a wide range of engineering artifacts relevant during the design and construction of embedded and/or safety-critical systems. However, our modeling approach does not prescribe a fixed set of domain-specific modeling languages or domains and can easily be extended and tailored to a particular specific modeling use case.

The set of modeling domains presented above are especially relevant to the design of embedded systems. However, our framework may also take into account aspects of business and other applications. To this end, we envision domains addressing security and privacy considerations (e.g. to model information flows), timing models, description of data persistence, as well as usability models.

The next logical step based on such integrated domains is a holistic multi-concern consideration and a tightly coupled derivation of architecture optimization guidelines. These are beyond the scope of our current work and thus remain open as future topics.

2.3 Purpose-Specific Data

While the domain-specific models described above are derived from the GPML data through model-to-model transformations, our approach also covers the generation of purpose-specific data artifacts through model-to-text and model-to-model transformations. In contrast to the bidirectional transformation between GPML and DSML artifacts, the transformation into purpose-specific data (PSD) is unidirectional. This limitation is by design, since the GPML/DSML model should be regarded as the true information source, from which derived artifacts can be regenerated automatically.

Previous considerations only took into account the varying focus of domain-specific (human) developers. In order to address the increasing amount of generative (i.e. machine-based processing) steps, another distinction is chosen for

this information base: PSD artifacts fall into either two categories; *Human-Consumable Artifacts* and *Machine-Consumable Artifacts*, as shown at the bottom of Fig. 1.

Human-Consumable Artifacts

As the name suggests, this kind of data centers on processing of information by humans. One can imagine a variety of scenarios where a tailored subset of the modeled information is desirable:

Views, a concept from software architecture, can be found again in our methodology. Since a system specified across several domain-specific models is not easily understood by non-technical stakeholders, a processed and condensed excerpt of the integrated system model is preferable. These views focus on specific aspects of the entire system and facilitate a better understanding and clearer communication.

Besides this more dynamic use case, which requires tool support, we also propose another variant of human-consumable artifacts:

Documentation, and in particular its automated generation, is an important factor in tightly integrated system engineering processes and plays a crucial role in quality-driven architecture.

The holistic nature of our proposed integrated system modeling approach facilitates document generation on a high abstraction level. For example, a common documentation requirement in safety-critical systems calls for seamless forward and backward traceability from system requirements down to the implementation level, and its proper documentation. Since the Omni model contains all necessary architectural elements and their relationships, generating such documentation consistently and in an easily navigable format (for example as hyperlinked HTML documents) is an effortless automated task. Using hypertext formats elegantly solves the traditional problem of limited traceability of these documents and makes them easily navigable.

The availability of usable, consistent, and up-to-date textual artifacts can help to reduce cost of safety certification by supporting high quality and early review of certification-related documents. Additionally, the same model-based document generation approach can be used to capture the results domain-specific analyses of the system that cover individual stakeholders' interests, leading to the next category of human-consumable artifacts:

Metrics are the concept of choice during analysis of certain Key Performance Indicators (KPI) of the system. From a project management perspective, we envision this approach to be useful for specification of (among others) test and requirement coverage metrics as an indicator of overall project progress.

A wholly different application scenario for metrics in the context of human-consumable artifacts is their use as a decision guidance in development processes: For example, a safety engineer may propose a change to the system model based

on the evaluation of a certain set of metrics. A similar use case is the improvement of test-related model artifacts based on a metric, reflecting the insights of a multi-concern consideration of the current application.

Machine-Consumable Artifacts

Our framework may also be used to export highly-specific data for further computation by external tools from the integrated system model. In contrast to the previous use case of human-consumable artifacts, this data is stored in a format optimized for machine processing. However, it also is subject to the limitation of unidirectionality, meaning that machine-consumable artifacts may not be part of a round-trip engineering approach. Such functionality is provided by the model-based analysis framework described in the following section.

We can distinguish different kinds of export formats, according to their intended application.

Intermediate Data, whose main purpose is to easily adapt to other tooling or the integration of libraries used for dedicated problem solving. For example, a processing step might provide input to an external optimization engine for the comparison of different architecture alternatives in the form of such intermediate data.

The term *intermediate* emphasizes the ephemeral nature of this kind of artifacts, representing a temporary result of a deterministic computation step based on the permanent information in the Omni model.

Refined Models, represent a special application of the previously mentioned *Intermediate Data*, where a domain-specific model is taken as a computational basis.

This model is either reduced to a limited scope, or enriched with supplementary information from another domain. For example, early applications of model-based testing did not separate the application model and the test model and instead only used the application model to generate test cases. Domain-aware approaches on the other hand favor the use of additional data in order to interface with existing tools to harness beneficial synergies.

While out of the immediate scope of our research, it should be noted that the final integrated system model is a suitable basis for generation of *source code*, as indicated by transformation steps on the right side of Fig. 1. The integrated nature of the Omni model as well as purpose-specific data allows the code generation engine to make more educated decisions about the context of the source code to be synthesized. A possible scenario could be the automated application of defensive programming techniques in generated code, e.g. assertion of pre- and post-conditions or calculation of checksums, based on component contracts or safety requirements from the integrated system model.

3 A Model-Based Architecture and Analysis Framework

Based on the modeling approaches introduced in the previous section, we have developed a reference technology platform geared towards the domain-aware modeling of safety-critical systems and their quality attributes. In addition to the domain-specific metamodels, the prototype includes a framework for definition of model-based architecture analyses, introduced below in Sect. 3.3.

3.1 Technical Foundations and Tooling

As shown in Fig. 2, the analysis framework consists of three major components:

Enterprise Architect. The commercially available Enterprise Architect (EA) is a general-purpose modeling tool, providing the full range of UML modeling capabilities to the system designer. Domain-specific metamodels are integrated via EA’s *Model-Driven Technologies* (MDG) feature

Model Repository. A relational database system is used for persistent storage of the model repository and allows for external access to the system model without the need for tight coupling with EA.

Architecture and Analysis Framework. The actual architecture and analysis framework, which offers model analysis services via a web service interface. Section 3.2 below describes the concrete execution model within the analysis framework.

For details on the various technologies involved and their interactions, see Sect. 3 of [3].

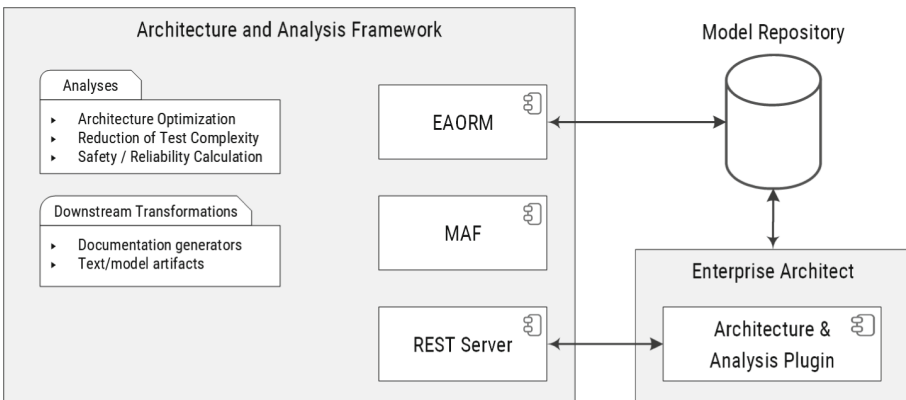


Fig. 2. Technical overview of the reference technology platform architecture.

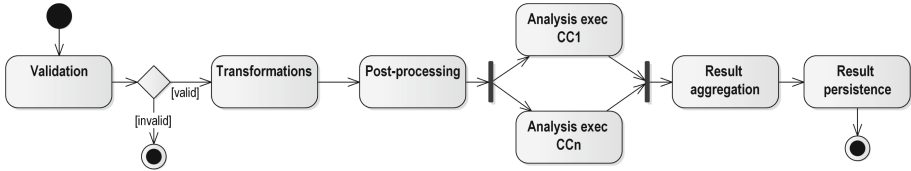


Fig. 3. Execution workflow within the architecture and analysis framework.

3.2 Analysis Execution Workflow

In order to execute an analysis request by the user, the Architecture and Analysis Framework passes through multiple execution phases. This section will give an overview of the necessary processing steps, as summarized in the activity diagram in Fig. 3.

Validation. Before any further processing takes place, the framework validates the analysis configuration supplied through the web service interface. This configuration adheres to a custom textual DSL and determines the types of analyses to be executed, their input model elements, as well as any additional parameters required to run the analysis.

The validation is carried out on the dependency graph between the analyses requested as part of the configuration. In order to qualify as a valid configuration, this graph must be acyclic and be closed under the transitive dependency relation. The second criterion assures that for each analysis, all its (transitive) prerequisites are also part of the configuration. If the configuration is found to violate these soundness assumptions, the execution engine aborts processing of the request and reports an error.

The actual execution order for all configured analysis is then calculated as a topological sorting of the dependency graph. As a subsequent optimization, the execution graph can be decomposed into its connected components to allow for parallel execution of multiple analyses: By definition, no dependencies exist between two analyses in different connected components, hence they are eligible to be processed by the framework simultaneously.

Figure 4 shows an example for such an execution graph from the case study described in detail in Sect. 4. Transformations are shown as parallelograms, rectangles represent the requested analyses together with their position in the calculated execution ordering. Dashed arrows indicate a dependency on transformation outputs, while solid arrows denote the prerequisite relation between analyses. For readability reasons, transitive prerequisite arrows are omitted from the graph.

Transformation Execution. The Enterprise Architect input UML model is transformed into the corresponding domain-specific representations using a set of model-to-model transformations for each modeling domain. We have chosen the

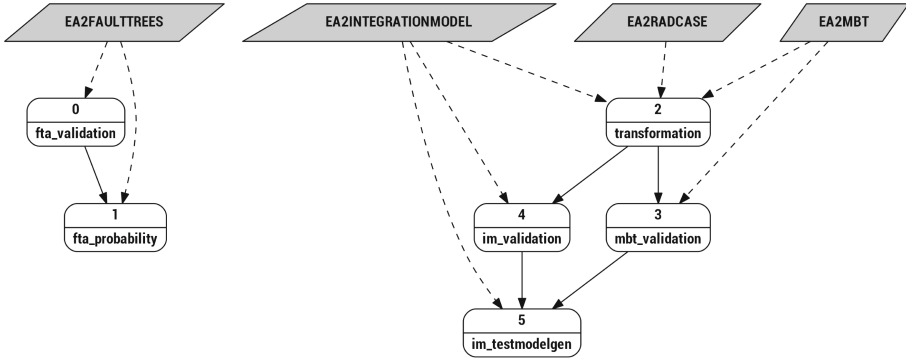


Fig. 4. Execution graph for the case study example (see Sect. 4).

QVT Operational language (QVTo, see [10,11] for details) as the M2M transformation language for our prototype. Each domain-specific model can be obtained from the integrated system model by applying its associated transformation, mapping the extended general purpose modeling language (see Sect. 2.1) onto the domain-specific modeling languages (Sect. 2.2).

In order to simplify integration with the Eclipse EMF-based QVTo engine used for implementation of the transformation phase and model-based analyses, all domain-specific metamodels have been described using the Ecore metamodeling facilities.

Post-processing. Optionally, an analysis may define arbitrary post-processing steps to be executed after the M2M transformation phase. Since the post-processing is implemented as regular application code, it can be used for additional calculations beyond the expressive capabilities of the transformation language. Examples include the handling of Java enumeration types and transformation of non-primitive value types into proper objects.

Analysis Execution. After inputs in the form of M2M transformation outputs are available, the execution engine iterates the pre-calculated execution order and runs all analyses specified by the configuration. Additional input parameters from the analysis configuration will be forwarded to the respective analysis.

For analyses that require functionality for data-flow based processing of a model, the execution phase can also delegate to the Model Analysis Framework (MAF in Fig. 2). This framework allows the use of data-flow techniques originally researched in compiler construction for iterative, model-based analysis of design artifacts (see [12] for details).

Analyses report their execution status back to the framework, and may create or modify arbitrary model elements to represent the results of their calculation.

Result Aggregation and Persistence. The execution engine tracks all modifications (object creation/deletion, attribute modification) to model elements performed in the analysis execution phase, as mentioned above. We have specified a compact domain-specific language for the description of reverse transformations of domain-specific models, while maintaining the integrity constraints of the original EA general-purpose model.

Note that only the general-purpose model is persisted in the model repository to remedy the problem of consistency across the transformation steps.

3.3 Processing of Integrated Model Data

Based on the execution mechanism previously described, we have developed a range of model analyses that can be applied to the integrated system model and its embedded domain-specific models. Consequently, one specific processing step is possibly made up out of multiple analyses, chained together. Most of the use cases mentioned in Sect. 2 use this mechanism as a technical basis for intermediate computations.

Conceptually, we have identified three major classes of model analyses that can be distinguished by their responsibilities as well as the type of input and output models:

Validation Analyses consume one or multiple input domain-specific models, but do not generate any new model elements as their output. Rather, a validation analysis verifies the syntactic and semantic well-formedness of its input models. In case this validation fails, the analysis produces a report of the identified violations and returns it as a separate result to the client.

Therefore, the purpose of validation analyses is the assurance of model integrity and quality. They are feasible candidates for tighter integration with the modeling tools used, and can be executed continuously without user interaction to provide rapid feedback about the state and quality of the model.

Note that the existence of this class of analyses is a testament to the state of metamodel extensibility in current general-purpose modeling tools. This shortcoming has previously been identified as the primary driver for so-called *descriptive stereotypes* [13]. If GPML tools provided first-class support for *restrictive stereotypes* or even full *restrictive metamodel extensions* instead, the syntactic and semantic constraints for a DSML could be directly validated as part of the metamodel extension.

Calculation Analyses consume one or more input domain-specific models and calculate additional attributes for existing model elements, but do not add new elements.

These analyses can be seen as the formalization of a function application to their input models. Examples for this class of analysis are numerous, e.g. the automated update of probability information in reliability models, risk classification, or the analysis of timing bounds in behavioral models.

In our analysis framework, calculation analyses are an obvious application point for the Model Analysis Framework (see above), since its feature set is well suited to the iterative nature of function evaluation on complex models.

Generative Analyses both consume and produce model elements in one or more domain-specific metamodels. As such, they are similar to model-to-model transformations. However, they serve a broader purpose, and hence should be considered separately.

Generative analyses offer a consistent interface for the programmatic modification of the integrated system model as part of the execution workflow described in Sect. 3.2 above. As opposed to ephemeral M2M transformations, their results are stored persistently.

Possible uses of this class of analyses are very broad: One possible example is the support of the system designer through wizard-type functionality, for example to generate skeleton reliability models from an existing structural model of a system. A different application scenario is the automated creation of a test model and test cases from the abstract description of system structure and functionality in the integration model.

While some generative analyses produce results that are intended for use inside the modeling loop centered on the integrated system model, the results of other analyses targets consumption outside the context of the analysis framework. This class of analyses is referred to as *downstream transformations* in Fig. 2 and corresponds to the concept of purpose-specific data introduced previously in Sect. 2.3.

The generation of source code from the integrated system model is a prime example for a downstream transformation. While the resulting code artifacts can still be regarded as a form of model, they are not persisted within the model repository and their main purpose lies outside the analysis framework.

Another important member of this class are model-to-text transformations in the form of document generators. They can make the creation of textual artifacts transparent to the client and encapsulate the actual invocation of the underlying M2T transformation engine.

As stated before, the three types of analyses can also be combined in order to handle more complex tasks. The processing of test model artifacts with the goal of reducing the final level of test complexity, for example composes an analysis for cross-domain calculation with a second analysis for generation of more specific test model artifacts. Again, this represents a common case of data pre-processing for further external use, subsumed under the category of *downstream transformations*.

4 Case Study: Design and Evaluation of a Gas Heating System

In the following section we will demonstrate the use of our domain-specific modeling approach to the reliability analysis of a gas heating system. Further, we

take a closer look on related testing activities, which in turn benefit from the integrated model basis.

Gas boilers are commonly found in residential buildings to provide central heating by combustion of natural gas in a burner. A common extension to such heating systems is a reservoir to buffer a suitable amount of hot water. In case of a malfunction of the system, personal injury might arise. Therefore, the design of such a system must encompass an evaluation of the safety risks and include appropriate protection systems to reduce potential risks to an acceptable level.

The model artifacts shown in this paper represent a simplified version of a standard heating system to limit complexity to a manageable level. However, they nicely illustrate the application of our integrated modeling approach, its suitability for the development of safety-critical systems, and the improved efficacy of related quality assurance mechanisms.

4.1 System Structure and Behavior

Since our point of view on the system architecture is on a very abstract level, a plain UML component diagram is sufficiently expressive to describe the system structural domain for this case study. Figure 5 shows the main components of our exemplary gas heating system. Such a coarse-grained model can be derived in early design stages, as soon as the operational context of the system has been determined.

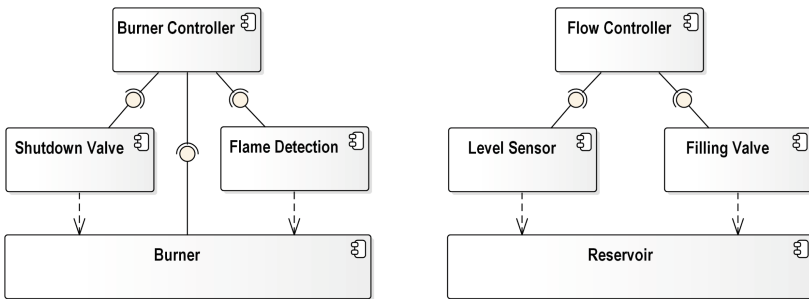


Fig. 5. Architecture of the gas heating system with integrated water heating circuit

We use UML state machines as well as other behavioral UML diagrams to model the internal functionalities of the elementary building blocks of the presented system. As an example, Fig. 6 describes the main operating states of the burner controller, which can be either operational or shut down in case a malfunction of the flame supervision mechanism has been detected.

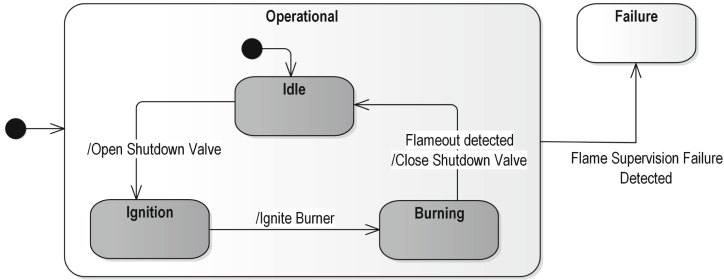


Fig. 6. Behavior model of the burner control logic.

4.2 Reliability Model

An important early step during development of a safety-critical system is the assessment of potential hazards and risks associated with the system under development (see Sect. 7.4 of [14] for details). This hazard and risk assessment, performed by a team of domain experts, can be documented inside the integrated system model.

As an illustrative example, we have chosen to analyze a potentially hazardous failure of the heating system, namely the presence of uncombusted gas in the burner chamber following a flameout. This situation can lead to rupture of the heating vessel due to over-pressurization as well as rapid deflagration or explosion of the uncombusted gas in the presence of an igniting spark. This hazard is assumed to occur with an intolerably high likelihood, which prompts the addition of a flame detection mechanism and an automatic safety shutdown valve as safety functions to the heating system.

The presence of a hot water reservoir in the heating system introduces an additional, unrelated hazard: If the filling valve malfunctions and becomes stuck in the open position, the reservoir might spill, posing the risk of severe scalding for anybody in its immediate vicinity.

A multitude of established engineering techniques exist for assessing the risk associated with a hazardous event and establishing the necessary risk reduction for an acceptable level of safety in the form of safety integrity levels. For this example, we have selected the risk graph technique described in appendix E of the IEC 61508-5 norm [15]. As shown in Fig. 7, the results of a qualitative assessment of each hazard are embedded inside the hazard analysis model as *RiskGraphSpecification* instances.

The introduction of these new system components demands for another iteration of the hazard and risk assessment, to ensure an acceptable safety level.

4.3 Requirements Model

Given the initial qualitative hazard and risk assessment, the analysis framework is able to automatically identify the necessary risk reduction for each hazard

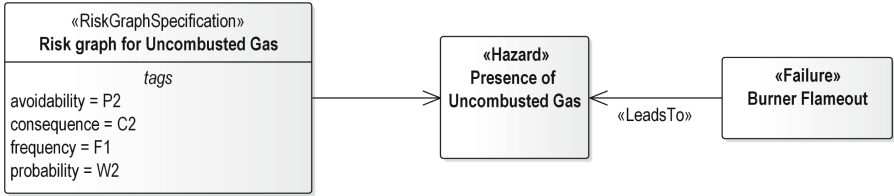


Fig. 7. Excerpt of hazard analysis and risk assessment model for the burner.

and generate appropriate safety function and safety integrity requirements (see Sects. 7.5 and 7.6 of [14] for the regulatory background). The system engineer can subsequently allocate these requirements to appropriate safety functions.

Figure 8 shows a part of the safety requirements model for the previously discussed risk of uncombusted gas as well as the risk of a spillover of the respective reservoir. A safety function with a specified safety integrity level has been introduced to mitigate these risks, and is allocated to the related system components described earlier via the integration model. Both of the mentioned system components are implicitly tied together via the more abstract system component, the heating itself.

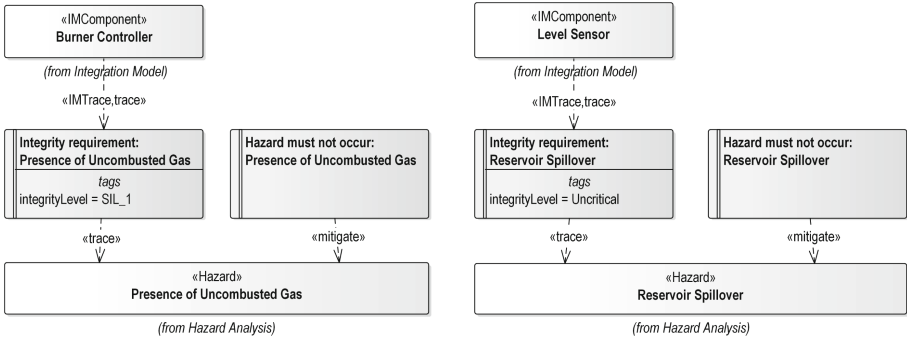


Fig. 8. Safety requirements model.

Note that beside this domain-specific model of safety requirements, a complete integrated system model of the gas heating would also contain all functional requirements that govern the regular operation of the system.

4.4 Integration Model

The integration model for our use case ties together the system structure and behavioral domain with all additional domain-specific models like the reliability or related test models. Additionally this artifact forms a hierarchy of abstract

components with the entire system under development at its root. Furthermore, the IM reflects the allocation of abstract functionality, e.g. the logic of the burner controller, to components and contains traceability information into the concrete behavioral model. In our example, the integration model associates the state machine for the burner control (see Fig. 6) with the abstract control logic functionality.

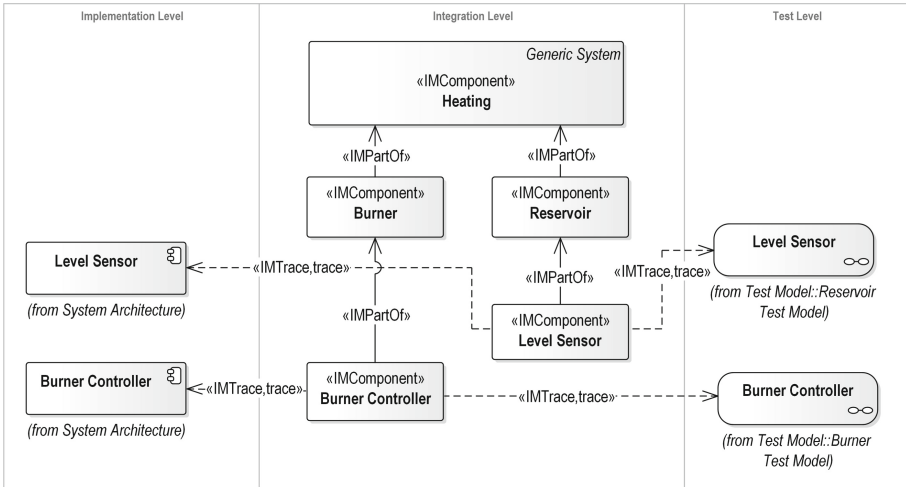


Fig. 9. Excerpt from Integration Model with links to SSD, SBD and TD.

We can see this contribution of the integration model to seamless model traceability in Fig. 10. The diagram emphasizes the trace relationship between the non-functional domains of the integrated system model for the heating system, in particular the requirements and hazard analysis domains. This traceability information is preserved by the model-based analysis framework and downstream transformations, especially during generation of source code. Therefore, based on this information, accompanying documentation can be generated that serves as evidence in safety certification of the burner control system.

4.5 Test Model

Beside the use case of generating comprehensive documentation for certification purposes, the results of such a safety and reliability analysis may also be utilized for test complexity reduction purposes. One might imagine a scenario, where a management decision cuts down on the amount of time available for test-related activities. However, certification requires that safety-critical software components need to be tested extensively, achieving a certain degree of test coverage. It is evident that these two conflicting restrictions cannot be met by simultaneously, necessitating a trade-off. To overcome this problem, the tester

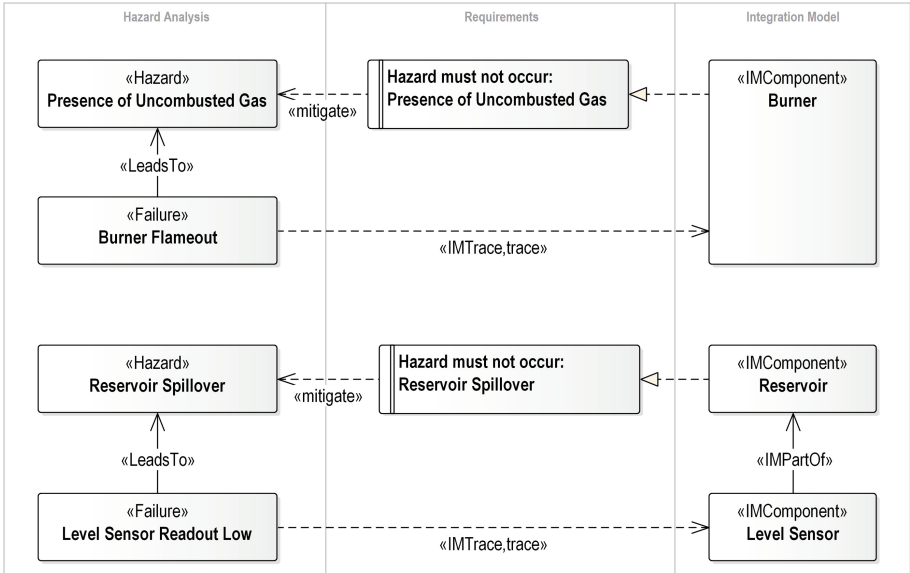


Fig. 10. Excerpt from Integration Model with trace links to reliability and requirements domains.

makes use of the central model artifact, the integration model, specifically its aspect capabilities described earlier in this paper. This mechanism enables the tester to disregard certain parts of the holistic test model, thereby producing a reduced model for further test case generation. An integrated view on Figs. 8 and 9 illustrates the path of information flow across the domain-specific models. Applying the previously outlined scenario of focusing tests on safety-relevant system parts, for example might drop the test model for the *Level Sensor*, since its safety impact has been marked as *uncritical* (see Fig. 8).

A combination of these flexible aspect configurations allows us to scope highly specific excerpts of the system model and thus focus on dedicated test cases with high impact on overall system quality.

5 Related Work

Our work relates to previous research in three related, but separate fields: Firstly, our approach provides a means of integrating various engineering disciplines into a coherent tool environment. Each of these disciplines brings with it its own set of domain-specific engineering artifacts and modeling languages. Finally, our implementation of an architecture analysis framework based on an integrated system model relates to prior work in the field of model-based analysis.

The following sections give a short overview of the relevant literature in these three fields, as they relate to our current research.

5.1 Modeling Tool Integration

In his seminal work, Wassermann [16] describes an approach for integration of heterogeneous tools in a software engineering tool chain. He describes an integrated software engineering framework based on three cardinal dimensions of interoperability – presentation, data, and platform integration.

The EU-funded iFEST project (Industrial Framework for Embedded Systems Tools¹) was aimed at developing an integrated framework for embedded systems, addressing both software and hardware concerns. The iFEST approach specifies a tool integration framework that leverages the OSLC specification to allow data exchange between heterogeneous modeling tools. Since it is focused exclusively on the aspect of tool integration, this approach does not address the field of model-based analyses of the integrated system model.

5.2 Domain-Specific Modeling

Zschaler et al. [17] propose a generalization of DSLs to domain-specific modeling languages, in order to capture common concepts found in families of related DSLs and facilitate automation.

Similarly, de Lara et al. [18] describe an approach for domain-specific multi-level metamodeling languages, allowing for the definition of deep language hierarchies. Their approach contains a set of reusable metamodel transformations for management of multi-level metamodeling languages and describes approaches for code generation in such a setting.

The use of UML as a graphical visualization language for domain-specific modeling languages is proposed by Graaf and van Deursen [19]. Their work proposes model-to-model transformations as a means of deriving a visual representation from a domain-specific model. Conceptually, these transformations can be regarded as an embedding of the DSML into a generic-purpose modeling language, specifically into UML.

As stated by Dias et al. [20], this also holds for the testing domain. Their seminal survey showed that the majority of MBT approaches makes use of UML behavioral modeling capabilities, sometimes extended by certain domain-specific data. One of their conclusive remarks mentions the active use of UML-like languages due to the wide distribution of basic skills in this area.

5.3 Model-Based Analysis

Papadopoulos and McDerimid [21] introduce HiP-HOPS (Hierarchically Performed Hazard Origin and Propagation Studies), a methodology for model-based hierarchical reliability analysis of component-based systems. Based on the architecture of the system under analysis and certain failure annotations, HiP-HOPS allows for bottom-up generation of Fault Trees and so-called interface-focused FMEA results for a system. Under the HiP-HOPS methodology, components are

¹ <http://www.artemis-ifest.eu/>.

enriched with additional model information about their failure behavior. Various classes of interface failures are defined that can be used to describe the black-box failure model of a system on a component level.

This approach has subsequently been extended to accommodate aspects of automatic architecture optimization. Papadopoulos et al. [22] further describe a conceptual approach for the automatic allocation of safety integrity levels to components of safety-critical systems. This work focuses on the automotive domain and uses the EAST-ADL2 modeling language for architecture description. Similarly, the authors propose a more generic architecture optimization technique based on the HiP-HOPS methodology and the use of genetic algorithms [23].

A complementary approach can be found in the EU-funded MBAT project (Combined Model-based Analysis and Testing of Embedded Systems²). This project aimed to provide a methodology and technology platform for specification of system analysis and V&V activities in the context of embedded system engineering. The central element of the proposed methodology is the so-called *A&T model* (short for [static] analysis and [model-based] testing), highlighting the focus of the approach to the quality-assurance domain.

An alternative path to overcome the constantly rising complexity of testing by utilizing information from other domains was proposed by Gebizli et al. [24]. The use of risk ratings of system components in combination with MBT in order to iteratively refine test models showed promising results. The resulting test suite boasts better fault detection capabilities in contrast to traditional MBT approaches, whereas the amount of time for testing was reduced.

6 Conclusions

We have proposed a valuable approach for integrated system modeling and model-based architecture analysis.

Our work introduces a solution to the challenge of integrating both system modeling and quality-related artifacts in the design and implementation of embedded systems. The resulting *integrated system model* or *Omni model* establishes explicit traceability between domain-specific modeling artifacts and enables consistent change management and change impact analyses.

This domain-centered view of systems engineering incorporates the fundamental challenge of multi-concerns design by unifying previously disjoint modeling domains.

Based on this holistic, model-based view on the system under development, complex model analyses can be performed to validate, process, or enhance the integrated system model. Analyses may also generate textual or model content for further processing outside our proposed methodology, for example as generated source code or management reports.

We have developed a reference technology platform that combines our proposed integrated system modeling approach with a model-based analysis framework. The suitability of this prototype is demonstrated through a case study,

² <http://www.mbat-artemis.eu/>.

which illustrates the use of the framework to model the reliability and testing aspects of a residential gas heating burner, a simple safety-critical embedded system.

We envision a variety of possible application fields for our approach as a basis for future research work: The seamless availability of information across domain boundaries makes the integrated system model open for use in automated systems engineering processes. For example, suitable analyses could be developed to support the (semi-)automated optimization of certain architecture aspects under consideration of reliability and safety aspects. In order to better support these optimization heuristics, the system model can be enhanced with stricter, machine-comprehensible formalization, such as component safety contracts.

Overall, we predict that the consequent application of model-based design methodologies will help to cope with the current challenges of systems engineering and help to create safe and maintainable products.

Acknowledgements. The research in this paper was funded by the German Federal Ministry for Economic Affairs and Energy under the Central Innovation Program for SMEs (ZIM), grant numbers KF 2751303LT4 and 16KN044120.

References

1. Sommerville, I.: Software Engineering, 9th edn. Pearson Education, New York (2011)
2. Boehm, B.W.: Software Engineering. Technical report, TRW Systems and Energy Group (1976)
3. Rumpold, A., Pröll, R., Bauer, B.: A domain-aware framework for integrated model-based system analysis and design. In: Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development (MODEL-SWARD), pp. 157–168. SCITEPRESS (2017)
4. Roman, G.C.: A taxonomy of current issues in requirements engineering. *Computer* **18**, 14–23 (1985)
5. Requirements Interchange Format (ReqIF), Version 1.2. Specification, Object Management Group (OMG), Needham (2016)
6. Siegemund, K., Thomas, E.J., Zhao, Y., Pan, J., Assmann, U.: Towards ontology-driven requirements engineering. In: Workshop Semantic Web Enabled Software Engineering at 10th International Semantic Web Conference (ISWC), Bonn (2011)
7. Semantics of a Foundational Subset for Executable UML Models, Version 1.2.1. Specification, Object Management Group (OMG), Needham (2016)
8. Vesely, W.E., Goldberg, F.F., Roberts, N.H., Haasl, D.F.: Fault tree handbook. Technical report, DTIC Document (1981)
9. Kaiser, B., Liggesmeyer, P., Mäkel, O.: A new component concept for fault trees. In: Proceedings of the 8th Australian Workshop on Safety Critical Systems and Software, vol. 33, pp. 37–46. Australian Computer Society, Inc. (2003)
10. Kurtev, I.: State of the art of QVT: a model transformation language standard. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) AGTIVE 2007. LNCS, vol. 5088, pp. 377–393. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89020-1_26

11. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.3. Specification, Object Management Group (OMG), Needham (2016)
12. Saad, C., Bauer, B.: Data-flow based model analysis and its applications. In: Moreira, A., Schätz, B., Gray, J., Vallecillo, A., Clarke, P. (eds.) MODELS 2013. LNCS, vol. 8107, pp. 707–723. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41533-3_43
13. Schleicher, A., Westfechtel, B.: Beyond stereotyping: metamodeling approaches for the UML. In: Proceedings of the 34th Annual Hawaii International Conference on System Sciences, 10 p. IEEE (2001)
14. Functional safety of electrical/electronic/programmable electronic safety-related systems - Part 1: General requirements. Standard, International Electrotechnical Commission, Geneva (2010)
15. Functional safety of electrical/electronic/programmable electronic safety-related systems - Part 5: Examples of methods for the determination of safety integrity levels. Standard, International Electrotechnical Commission, Geneva (2010)
16. Wasserman, A.I.: Tool integration in software engineering environments. In: Long, F. (ed.) Software Engineering Environments. LNCS, vol. 467, pp. 137–149. Springer, Heidelberg (1990). https://doi.org/10.1007/3-540-53452-0_38
17. Zschaler, S., Kolovos, D.S., Drivalos, N., Paige, R.F., Rashid, A.: Domain-specific metamodelling languages for software language engineering. In: van den Brand, M., Gašević, D., Gray, J. (eds.) SLE 2009. LNCS, vol. 5969, pp. 334–353. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12107-4_23
18. de Lara, J., Guerra, E., Cuadrado, J.S.: Model-driven engineering with domain-specific meta-modelling languages. *Softw. Syst. Model.* **14**, 429–459 (2015)
19. Graaf, B., van Deursen, A.: Visualisation of domain-specific modelling languages using UML. In: 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS 2007), pp. 586–595. IEEE (2007)
20. Dias Neto, A.C., Subramanyan, R., Vieira, M., Travassos, G.H.: A survey on model-based testing approaches: a systematic review. In: Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies, pp. 31–36. ACM (2007)
21. Papadopoulos, Y., McDermid, J.A.: Hierarchically performed hazard origin and propagation studies. In: Felici, M., Kanoun, K. (eds.) SAFECOMP 1999. LNCS, vol. 1698, pp. 139–152. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48249-0_13
22. Papadopoulos, Y., et al.: Automatic allocation of safety integrity levels. In: Proceedings of the 1st Workshop on Critical Automotive Applications: Robustness & Safety, pp. 7–10. ACM (2010)
23. Papadopoulos, Y., et al.: Engineering failure analysis and design optimisation with HiP-HOPS. *Eng. Fail. Anal.* **18**, 590–608 (2011)
24. Gebizli, C.S., Metin, D., Sozer, H.: Combining model-based and risk-based testing for effective test case generation. In: 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 1–4. IEEE (2015)



A Domain-Specific Modeling Approach for Testing Environment Emulation

Jian Liu¹(✉), John Grundy², Mohamed Abdelrazek²,
and Iman Avazpour²

¹ Swinburne University of Technology, Hawthorn, VIC 3122, Australia
jianliu@swin.edu.au

² Deakin University, Burwood, VIC 3125, Australia
{j.grundy, mohamed.abdelrazek,
iman.avazpour}@deakin.edu.au

Abstract. Software integration testing is a critical step in the software development lifecycle, as modern software systems often need to interact with many other distributed and heterogeneous systems. However, conducting integration testing is a challenging task because application production environments are generally neither suitable nor available to enable testing services. Additionally, replicating such environments for integration testing is usually very costly. Testing environment emulation is an emerging technique for creating integration testing environments with executable models of server side production-like behaviors. Aiming to achieve high development productivity and ease of use for business users, we propose a novel domain-specific modeling approach for testing environment emulation. Our approach is based on model-driven engineering, and abstracts software service interfaces, or endpoints, into different request message processing layers. Each of these layers represents a modeling problem domain. To model endpoints, we develop a suite of domain-specific visual languages for modeling these interface layers. To build a testing environment, we have created a supporting toolset to transform endpoint models to executable forms automatically. We provide a set of example scenarios to demonstrate the capabilities of our approach. We have also conducted a user study that demonstrates the acceptance of our approach by IT professionals and business users.

Keywords: Model-driven engineering
Domain-specific visual modeling language · Software integration testing
Testing environment emulation

1 Introduction

1.1 Software Integration Testing

Emerging computing strategies, such as cloud computing and social networking, represent an ongoing shift from monolithic applications to highly distributed, heterogeneous and shared computing environments [1]. Most software systems need to interact with other systems to provide composite services to their clients or end users.

Thus, the performance of a software system is no longer determined only by its own internal components, but is also subject to its increasingly complicated interactions with external systems in its operational environment. This means that for effective testing of a software system, testing interconnections (static communication aspects) and interoperability (dynamic communication aspects) with other systems that it communicates with in its realistic production environment is critical.

System Integration Testing (SIT) is a testing process that exercises a software system's behaviors when interacting with other inter-connected systems. It tests the interactions between different systems and verifies the proper execution of the system in its deployment environment [2]. To test the interactions of a System Under Test (SUT) with the systems (that we call "endpoints") in an enterprise environment, the testing environment must provide a test-bed, that encompasses all services of the endpoints the SUT will invoke in the environment.

Endpoints deployed to a testing environment have some unique characteristics. First, a SUT often interacts with many different types of applications in its environment. Therefore, it is desirable that each endpoint development cycle should be short and the development approach should have high development productivity. Second, SIT is normally conducted by testing engineers or business users. Most of them have rich business domain knowledge, but may lack programming skills. They prefer to model endpoints using problem domain concepts, rather than code them using a textual language. Last, endpoints, as server-side applications to provide testing services to their SUTs, do not necessarily provide accurate results under all circumstances. Therefore, we may simplify some internal implementations, in return for quick development.

1.2 Testing Environment Emulation

Testing Environment Emulation (TEE) is an emerging technique to develop SIT environments for SUTs that interact with many external systems. The main idea is to model the interactive behaviors of each system in a environment and replace the systems by instances of the corresponding models in the emulation environment [3]. The goal is to make the emulated testing environment rich enough to "fool" SUTs into behaving as though they are talking to the real external systems. Other components that sit underneath or in the background are ignored from the emulated environment perspective whenever possible. Particularly, an emulated endpoint is a simplified version of a real system with three assumptions:

- As an endpoint is used to provide test-bed for SUT integration testing, only the external behaviors of the endpoint application are considered and its internal implementations will be ignored;
- An endpoint is specifically developed for the integration testing of a SUT. Therefore, a subset of the endpoint application operations invoked by the SUT are provided;
- Serving as a defect detection tool for system debugging, an endpoint should be able to capture all SUT interface defects, together with their types, origins and other information.

The key benefits from using TEE include:

- It provides a production-like test-bed for provisioning of testing functionality to SUTs in a much more cost-effective way than application replication;
- Development of such a testing environment could be quick and easy, as some internal logic implementations and auxiliary modules are ignored;
- The test-bed is easily configured and monitored for performing Quality-of-Service (QoS) aspects testing, such as simulating different numbers of instances of a same endpoint type for performance test;
- Software interface defects can be captured and the defect cause information can be reported.

1.3 A Domain-Specific Approach to Testing Environment Emulation

Domain-Specific Modeling (DSM) achieves high development productivity and ease of use by focusing on a narrowed problem domain, so that specific high-level abstraction modeling languages and supporting toolsets can be created. To develop our DSM approach to emulate testing environment, we proposed a new software interfaces description framework, where software interfaces are abstracted into logically separated signature, protocol and behavior layers. We use modular development approach to model endpoints, and each module represents one interface layer.

Our DSM approach consists of an endpoint modeling environment and a runtime environment to provide testing services to SUTs (see Fig. 1). The modeling environment includes a suite of domain-specific Visual Modeling Languages for Testing environment emulation (TeeVML) to model endpoints in interface layers. The runtime environment is hosted in Axis2 SOAP engine [4], and is generated automatically by

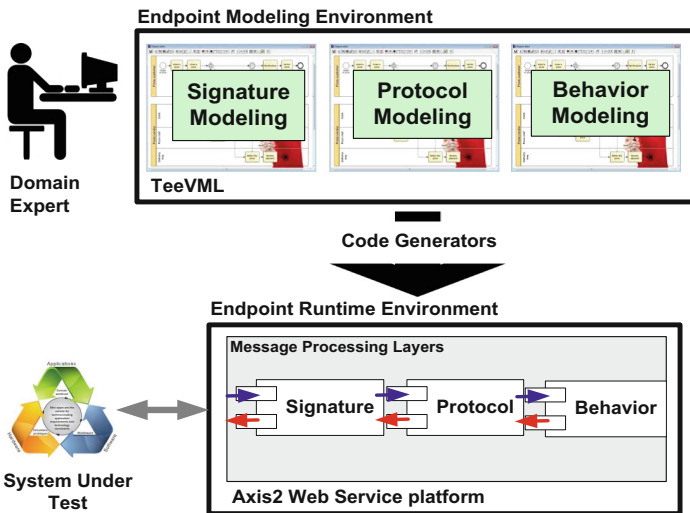


Fig. 1. Endpoint modeling and runtime environment.

transforming the endpoint signature model. Testing service is enabled through Web Service provided by Tomcat Servlet Container [5].

This research paper is an extended version of our MODELSWARD 2017 paper [6], and we add the following contents in addition to enriching the covered topics: (1) a brief discussion on SIT characteristics and the benefits from using a model-driven approach to develop endpoints; (2) a description of our visual language notation design for achieving high usability; and (3) an introduction to our Domain-Specific Visual Languages (DSVLs) for modeling endpoints.

This rest of the paper is organized as follows: Sect. 2 motivates our research by a case study, followed by an introduction to our DSM approach in Sect. 3. In Sect. 4, we briefly discuss our visual symbol design and introduce our TeeVML. We show how an endpoint is modeled and then describe the steps to convert endpoint models into testing runtime environment in Sect. 5. In Sect. 6, we evaluate our approach and discuss the key findings from the results of a technical comparison and a user survey. This is followed by a review of related work in Sect. 7. Finally, we conclude this paper and identify some key future work in Sect. 8.

2 Motivation

We select a typical business case of a company integrating its legacy system with a public cloud application and use this case to describe the potential interactions between an endpoint and its SUT. The company currently has an in-house Enterprise Resource Planning (ERP) system (such as PeopleSoft Finance [7]) to support its daily operations. For the purpose of streamlining its sales process and improving operational efficiency, the company plans to introduce a public cloud Customer Relationship Management (CRM) service (such as [salesforce.com](https://www.salesforce.com) [8]) as its frontend application. From operation and data security considerations, all company data will be kept in-house in the ERP. Therefore, the CRM application must interact with the ERP system intensively for accessing persistent data and processing business logics.

The activity sequence diagram in Fig. 2 illustrates a typical sales process flow among users, the CRM application and the ERP system. Users access the CRM application for handling their client Purchase Order (PO). For every user request, the CRM must invoke a corresponding ERP operation using Remote Procedure Call (RPC) communication style [9]. Our main interest is on the interactions between the client CRM and the server ERP as described below.

Whenever the ERP receives a *logon* request from the CRM, it transits from idle state to home state and an interactive session starts. The next valid operation is *porequest*, followed by *inventorycheck*. The returned value of *inventorycheck* will determine whether supplier chain related steps will be executed. If the purchase item has enough stock for the PO, the process flow will jump over the supplier purchasing steps and directly go to *paymentrequest*. Otherwise, we should go through the purchase steps (#4, #5, #6 and #7) to buy the missing quantity of the item. *supplierpoapproval* and *approvalnotification* are iteration operations, informing all approvers one-by-one to give their approvals. If all required approvals for the supplier PO are obtained, the rest

of purchasing steps will be executed in the order as in Fig. 2. Otherwise, the sales process will be aborted without success.

To ensure the interconnectivity and mutual interoperability between the ERP and CRM, SIT must be carried out before putting the CRM in production. For this study, we treat the ERP as the endpoint that we need to develop, and the CRM as the SUT. Just as any other software development tools, users' primary concerns about our endpoint modeling approach will be: What can it do for their service emulation modeling and generation? Will it improve endpoint development productivity? How easily can it be used? From these assumptions, we defined three key research questions for guiding the development of our approach:

RQ1 – *Can we emulate a functioning integration testing environment capable of capturing all interface defects of an existing or a non-existing system under test from an abstract service model?*

RQ2 – *Would our model-based approach improve testing environment development productivity, compared to using third-generation languages (e.g. Java) to implement endpoints?*

RQ3 – *Can we develop a user centric approach, easy to learn and use to specify testing endpoints by domain experts?*

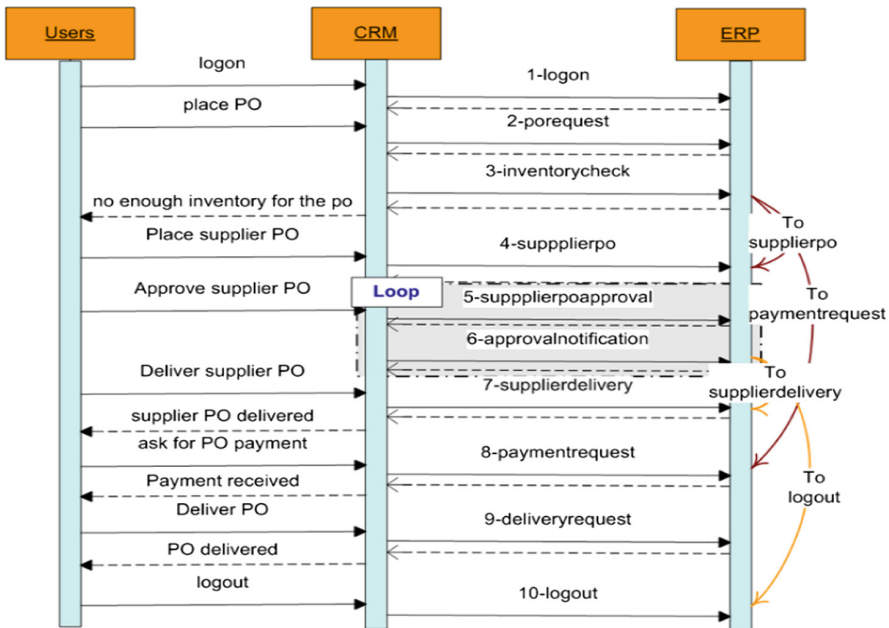


Fig. 2. The example ERP and CRM interactions process flow diagram [6].

3 Our Approach

To identify software interfaces common concepts and determine their relationships, we conducted our TEE domain analysis by investigating three applications interacting with their clients. These applications were the ERP system introduced in the motivation section, a LDAP server [10], and a core banking system [11]. These applications represent a variety of application domains in a typical enterprise environment. From this domain analysis, we proposed a layered software interfaces description framework for TEE, and defined interface defect types to be captured by endpoints. Consecutively, we then designed our Domain-Specific Languages (DSLs) to model endpoints in interface layers.

3.1 Software Interfaces Description Framework

There are three reasons for having a software interfaces description framework. First, we need to abstract software interfaces into different interactive aspects, so corresponding DSLs can be developed with a clearly defined problem domain boundary. Second, we can adopt a modular development architecture to model endpoints in layers. We may also be able to model a few versions of an endpoint type for different SUTs. Third, some of these interface modules may be shared among endpoints, if they have the exact same functionality.

Our framework abstracts software interfaces into three logically separated layers:

- *Signature* – following RPC communication style specification, this layer specifies the requests and responses of endpoint operations, their parameters and properties;
- *Protocol* – this layer defines the validity of a temporal sequence of endpoint operations, which can depend on either endpoint states (static protocol behavior) or runtime constraint conditions (dynamic protocol behavior), or both;
- *Behavior* – this layer abstractly describes endpoint internal operation request process and response generation, and the returned values in response messages are used to capture dynamic protocol defects.

A SUT operation request is processed by an endpoint step-by-step from signature, protocol, and down to behavior layers. Whenever an error occurs, the request processing will be terminated. Signature and protocol layers act as message pre-processors for checking the correctness of operation request syntax and temporal sequence, before handing the request over to behavior layer for generating a suitable response message.

3.2 Service Request Defects

To develop DSLs for endpoint layers, we must know all the defect types first. Table 1 lists and describes the possible interface defects types that a SUT request may have. The SUT request defects can be grouped into static and dynamic categories, depending on whether they will always cause interactive failures or under certain runtime conditions only. Normally, a software application has an interface specification to specify its provided operations and their parameters. A client SUT must send its requests to the application in accordance with the interface specification. Otherwise, interface fault

will occur due to a static interface defect. On the other hand, a dynamic defect happens under certain business scenarios. An example is the validity of the next request after *inventorycheck*, which is subject to the inventory result returned by the *inventorycheck* operation. In general, static defects can be found by code review against interface specification and SIT; while dynamic defects can only be captured by SIT.

We do not list any behavior defects in Table 1. This is because a SUT's obligation is to send correct requests to an endpoint and the way these requests are to be processed is defined internally by the endpoint. The reason why we still model the endpoint behavior layer is that the validity of alternative next operation requests may depend on what values are returned in the response message it has received based on a previous request.

Table 1. Service request defect types [6].

Type	Description
<i>Signature</i>	
Sig1	An operation request is not an operation provided by endpoint
Sig2	The parameters in an operation request are not matched with the parameters of the corresponding operation provided by endpoint, in terms of parameters' name, data type and order in the operation request
Sig3	One or more operation request mandatory parameter(s) is (are) missing
Sig4	One or more parameters in an operation request is (are) beyond the defined value range of the corresponding endpoint operation
<i>Protocol</i>	
Pro1	An operation request is invalid for the current endpoint state
Pro2	An operation request is invalid for the current endpoint state, as one or more parameter(s) violate(s) the defined constraint condition(s)
Pro3	An operation request is invalid for the current endpoint state, as one or more returned value(s) from a previous operation request violate(s) the defined constraint condition(s)
Pro4	An operation request is invalid, due to endpoint state transition driven by some internal event, such as time out
Pro5	An operation request is invalid, as endpoint is in processing a synchronous operation request
Pro6	An operation request is invalid, as one such request for an unsafe operation (i.e. not an idempotent operation that will produce the same results if executed once or multiple times) has been received by endpoint

3.3 Endpoint Metamodeling

A Metamodel defines all concepts and their relationships within a specific application domain. The key semantics and constraints associated with these domain concepts are also specified. The main inputs to our software interface metamodels are the software interface description framework and the software interface defects listed in Table 1.

Signature Metamodel

Endpoint signature layer models operations provided by endpoint and their parameters. Each parameter has some static properties, such as name, data type, order and mandatoriness. Some parameters with integer, float or date data type may also have upper and lower limits in dynamic nature.

Web Service Description Language (WSDL) specification describes the public interface exposed by a web operation, including what an operation does, where it resides, and how to invoke it [12]. Figure 3 illustrates the signature metamodel that our signature DSL is based on. The metamodel adopts a three-level architecture design. The top-level DSL (see Fig. 3a) uses WSDL 1.1 specification as its metamodel. It specifies the relationships among a root *definition* entity and other 5 entities: *service*, *port*, *binding*, *porttype* and *operation*. The middle-level Operation DSL (see Fig. 3b) is to define request and/or response message(s) in an endpoint operation. The operation communication pattern determines whether it contains a request message only, a response message only or both request and response messages. The bottom-level DSL (see Fig. 3c) is based on W3C XML Schema 1.1 for defining complex elements in a message.

The signature defects Sig1 to Sig3 in Table 1 can be detected by the Axis2 SOAP engine itself, transformed from signature model. To specify the upper and lower limits of a number or a date element (refer to Sig4 defect in Table 1), we add two properties to element type to detect any invalid request parameters beyond defined value limits.

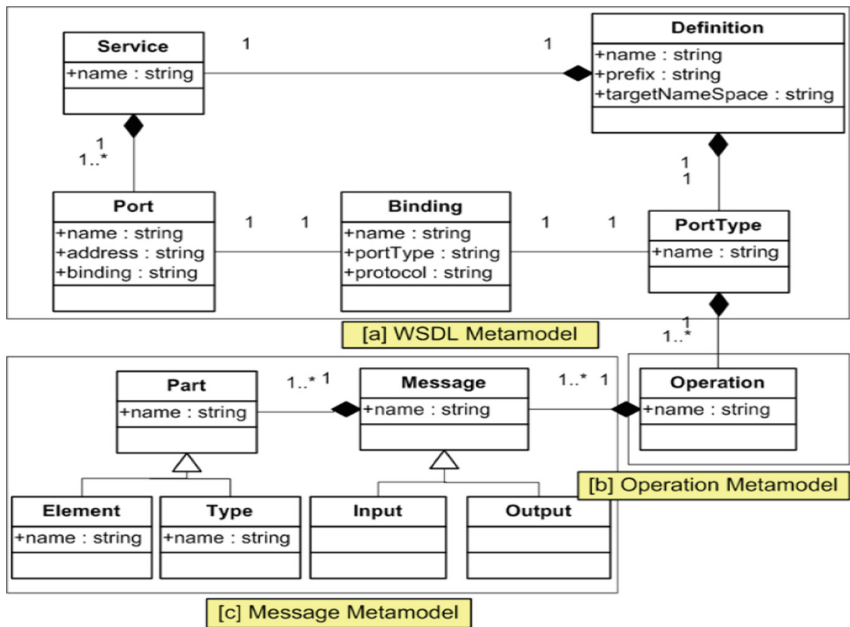


Fig. 3. Endpoint signature metamodel [6].

Protocol Metamodel

To capture both static and dynamic protocol defects, we designed an Extended Finite State Machine (EFSM) to enrich endpoint protocol modeling capability (refer to Fig. 4). Our EFSM adds one entity type and two entity properties to a standard operation driven endpoint state transition Finite State Machine (FSM) (the items we added are marked yellow in Fig. 4). Entity type is the *InternalEvent* which is used to define state transitions triggered by time event. One of the entity properties is the *StateTransitionConstraint* of the transition entity, and it is for specifying either static or dynamic constraints on state transition function. Another one is the *StateTimeProperty* of the state entity, which allows users to simulate synchronous and unsafe operations. As endpoint protocol modeling is relatively simpler than other two interface layers, we use a flat view presentation structure.

All protocol defects listed in Table 1 can be detected by an endpoint, modeled using the EFSM model: (1) Pro1 – the operation-driven state transition FSM; (2) Pro2 and Pro3 – the *StateTransitionConstraint* property; (3) Pro4 – the *InternalEvent* entity; and (4) Pro5 and Pro6 – the *StateTimeProperty*. Protocol modeling is only applied to stateful applications. This is because an endpoint uses its current state to validate the next coming operation. If an endpoint is a stateless application, its protocol modeling will be skipped, as all requests to the endpoint must necessarily contain the required information.

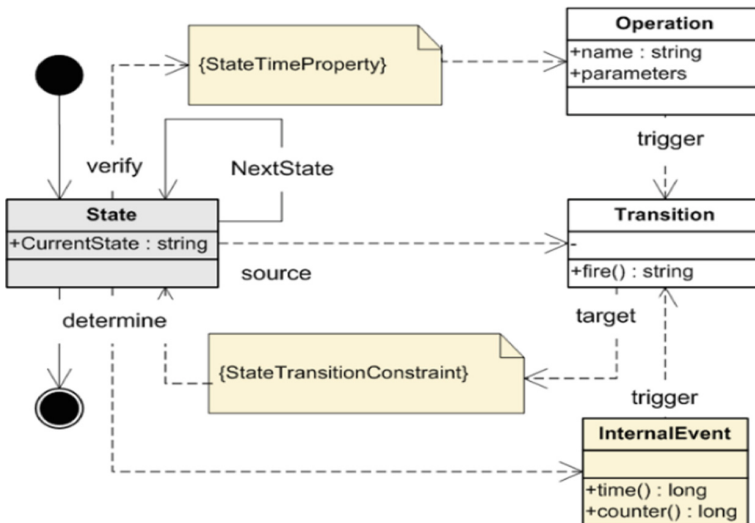


Fig. 4. Endpoint protocol metamodel [6] (Color figure online)

Behavior Metamodel

Our behavior metamodel is based on the Data Flow Diagram (DFD) programming paradigm [13]. We chose this metaphor as it allows for complex specification of behavior models but has also been shown to be understandable by a wide range of

software stakeholders. DFD programming execution model is represented by a directed graph; nodes of the graph are data processing units, and directed arcs between the nodes represent data dependencies. A node starts to process and convert the data whenever it has the minimum required parameters available on its input connector. The node then places its execution results onto its output connector for the next nodes in the chain.

To handle complicated business logic, we designed our behavior DSL using a hierarchical tree structure. The benefits of using the hierarchical structure are two-fold: First, we can reuse some of the nodes, if they perform exactly the same task but are located at different components. Second, it can help us manage diagram complexity problem. On the top level of node tree structure, discrete service nodes are used to represent the operations provided by an endpoint. At the bottom level, each node consists of some primitive programming constructs for performing operations on data and flow controls for directing execution sequence.

4 Our Domain-Specific Languages

A DSL realizes the concepts and their relationships defined in its domain metamodel by mapping them to corresponding programming constructs. A Visual Programming Language (VPL) lets users create programs by manipulating programming constructs graphically rather than by specifying them textually. Although there is no fundamental difference in expressivity, visual languages are generally easier to learn and use than textual languages [14]. To achieve ease of use for domain experts, we designed our TeeVML as a VPL.

4.1 Visual Symbol Design

Visual symbols have a profound effect on the usability and effectiveness of a visual language. Visual symbols are human thought representations for facilitating communications and problem solving among individuals. To be most effective in doing this, they need to be optimized for processing by human mind. For this reason to evaluate the “goodness” of a visual symbol, Larkin et al. defined the term *cognitive effectiveness* as “the speed, ease, and accuracy with which a representation can be processed by the human mind” [15]. The *cognitive effectiveness* determines the ability of visual symbols to communicate among a wide range of software stakeholders.

To establish a scientific foundation for visual symbols’ design, Moody proposed the Physics of Notations (PoN) and defined a set of principles to evaluate, compare, and construct visual symbols by using a synthesis approach based on theory and empirical evidence about the *cognitive effectiveness* of visual symbols [16]. Some of these principles are related to a visual language as a whole, such as Complexity Management, Cognitive Integration and Graphic Economy. While others focus on individual visual symbol’s properties, such as Semiotic Clarity, Visual Expressiveness and Perceptual Discriminability.

To maximize the *cognitive effectiveness* of our DSVLs, we applied Moody’s PoN principles to design our visual symbols. Among these principles, we put more emphasis

on those that are subject to DSVL's characteristics. When multiple entities are to be used, the Perceptual Discriminability principle will be our primary design consideration. This principle is assessed by the visual distance between symbols, measured by the number of visual variables on which they differ and the size of these differences. In contrast, there is no meaning to consider visual distance, if a DSVL contains only one entity. We would focus on the Semantic Transparency principle instead.

In addition to the *cognitive effectiveness* of visual symbols, there are also some other factors to be considered when designing DSVLs for this research such as reusability. To maximize the reusability, we should make models simple enough to be reused or easily assembled with others as a reusable component. This is the main reason why we have designed some single entity sub-DSVLs.

In the following sections, we introduce the designs of some TeeVML visual constructs. Interested readers can refer to our earlier publications [17, 18] for all the details.

4.2 Signature DSVL

Our Signature DSVL consists of three sub DSVLs: WSDL, Operation and Message. Table 2 describes the visual constructs for the five entities used in WSDL sub-DSVL. To provide sufficient visual distance for making them easily distinguishable, we used shape as the primary visual variable, supplemented with color and textual annotation.

4.3 Protocol DSVL

Protocol DSVL consists of three state entities for representing endpoint idle, home and working states and four transition relationships for managing endpoint state transitions. Table 3 provides our Protocol DSVL design details of the working state and relationships visual constructs. To distinguish between the relationship visual constructs, their visual variables include shapes at both ends, color, line type and textual annotation.



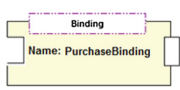
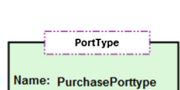
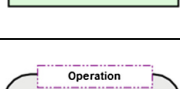
4.4 Behavior DSVL

Our Behavior DSVL has 9 visual constructs used for describing different types of tasks: (1) Service Node and Node for processing an operation or a task, (2) Arc and Entrance/Exit bars for directing dataflow, (3) Variable and Variable Array for holding intermediate results, (4) Conditional Operator and Loop for controlling process flows, and (5) Evaluator for performing arithmetic operations. To provide a general feeling of how Behavior DSVL is designed, we describe some of these constructs below.

Service Node

A service node specifies the process of an operation request and generates response. It imports the request and response parameters from the endpoint signature model. Figure 5a shows the visual construct of a service node. Its main design consideration is to display all request and response parameters for helping users to model the operation behavior. To manage behavior model view complexity, service nodes can be collapsed to hide parameters and reduce their symbol size (see Fig. 5b).

Table 2. WSDL sub-DSVL visual constructs.

Construct	Visual Symbol	Description	Property
Service		A set of system operations that are exposed to the Web-based protocols.	<i>Name:</i> A Service instance name.
Port		Address or connection point to a Web Service. It is typically represented by a simple HTTP URL string.	<i>Name:</i> A Port instance name. <i>Address:</i> The network address at which the Service is offered.
Binding		Binding entity specifies interface, SOAP binding style and transport protocol.	<i>Name:</i> A Binding instance name. <i>Type:</i> To identify the kind of binding details contained in a Binding entity instance.
PortType		PortType entity defines a Web Service, operations that can be performed, and the messages that are used to perform the operation.	<i>Name:</i> A PortType instance name. <i>Extends:</i> A lists of PortType entities that this PortType derives from.
Operation		A Web Service action and the way a message is encoded. An operation is like a method or function call in a traditional programming language.	<i>Name:</i> Operation instance name. <i>Pattern:</i> A template for the exchange of one or more messages.



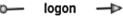
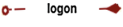
Entrance and Exit Bars

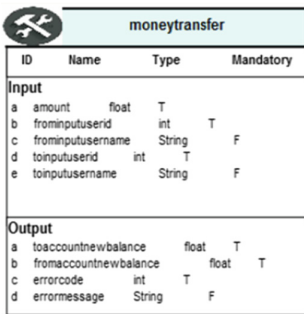
Entrance and exit bars (see Fig. 6) specify the input and output parameters and define where execution starts and ends within a service node or a node. The entrance bar has one “out” port underneath, and the exit bar has a normal “in” and an exceptional “in” ports on it. The parameters for both bars can be displayed or hidden by users, depending on whether users need to know these parameters.

Evaluator

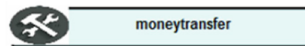
Evaluator (see Fig. 7) is use to perform arithmetic operations. The evaluator visual construct has three lines for defining a formula. The first line specifies a result variable to be assigned after the evaluator’s execution. The second line lists all variables to be used by the evaluator, separated by commas. The last line is the arithmetic formula with parameters in as “P” array. The order of the “P” array elements follows the sequence of the parameters in the second line.

Table 3. Protocol DSVL visual constructs.

Con-struct	Visual Sym-bol	Description	Property
Working State		It presents an endpoint state, which normally uses operation as its default name.	<i>Name</i> : State entity instance name; <i>Synchronous Operation</i> : Is the state operation in synchronous mode? <i>Processing Time</i> : Simulated operation processing time in seconds. <i>Safe Operation</i> : Is the state operation safe? <i>Transmission Time</i> : Simulated operation request transmission time in seconds.
Timeout Relationship		It links a “from” state to a “to” state for representing endpoint state change, if no valid operation request is received within a defined timeout period.	<i>Time</i> : The time in seconds for an automatic state transition.
Transition Relationship		It links a “from” state to a “to” state for representing a state transition.	<i>OperationName</i> : The operation triggers the state transition.
Constraint Transition Relationship		It links a “from” state to a “to” state for representing a state transition; The transition is subject to a constraint condition, defined by its dialog box (see Figure 4.7).	<i>Trigger Operation</i> : The operation triggers the state transition; <i>Operation Name1 + Field Name1</i> : To defines the first state transition condition; <i>Condition Operator</i> : It is used to compare the two conditions; <i>Operation Name2 + Field Name2</i> : To defines the second state transition condition.



[a] A service node definition



[b] A collapsed service node

Fig. 5. Behavior DSVL service node’s visual construct.

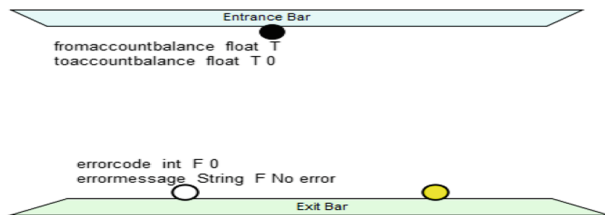


Fig. 6. Behavior DSVL entrance and exit bars' visual constructs.

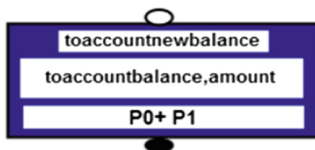


Fig. 7. Behavior DSVL evaluator's visual construct.

4.5 Code Generators and a Domain Framework

To make an endpoint executable to provide testing services to its SUT, endpoint models must be transformed to executable code by code generators. This code must then be integrated into a domain framework for providing infrastructure support. Ideally, the transformed code must be complete and in production quality, and should not require manual rewriting, inspection or additions.

Code Generators

A code generator accesses models, extracts information from them, and transforms the models into output in a specific form. This process is guided by the concepts, semantics and rules of the modeling language. Our TeeVML includes five code generators to transform endpoint signature, protocol and behavior models to corresponding codes:

- *WSDL* (signature) – To transform an endpoint signature model to WSDL 1.1 specification in XML format;
- *SQL Script* (signature) – To navigate through all operations and search for the parameters of the types of “int”, “float” or “date”. If such parameters are found, their “Minimum” and “Maximum” properties will be stored in an operation parameter table for verifying these parameters' ranges;
- *Groovy Code Generator* (protocol) – To access and navigate the entire protocol model for extracting endpoint protocol information. The extracted data are stored in a state transition table for validating the operation request and determining state transition;
- *Java* (protocol) – To generate Java code to query protocol information from the state transition table for each operation request and validate the operation request by several “if-else” statements;
- *Java* (behavior) – To define the interdependences among nodes and primitive visual constructs and specify internal implementations within the primitive constructs.

A Domain Framework

A domain framework normally serves for four purposes [19]: (1) to remove duplications from generated code, (2) to provide an interface for code generators, (3) to integrate with existing code; and (4) to hide target environment and execution platform. In addition, our domain framework also plays another important role – to provide a network infrastructure for facilitating message exchange between endpoints and SUTs.

As it is not our research focus, we have not developed our own but used Axis2 SOAP engine [4] instead. Axis2 brings some benefits to our endpoint modeling approach including: (1) Axis2 facilitates Design by Contract (DbC) programming style [20], and the implementations on both the endpoint and SUT sides are bound to a service contract defined by signature WSDL file; (2) Axis2 allows users to modify its SOAP message header by adding some QoS attributes to simulate a variety of business scenarios; and (3) Axis2 is a popular open-source tool, many IT professionals familiar with it.

To implement DbC programming, Axis2 generates linkage codes for both service provider and service client from a signature definition WSDL file. The service provider linkage code takes the form of a service specific implementation skeleton, along with a message receiver class that implements the *org.apache.axis2.engine.MessageReceiver* interface. The service client linkage code is in the form of a stub class, which always extends the Axis2 *org.apache.axis2.client.Stub* class. Both the service provider skeleton class and client stub class are generated by the *wsdl2java* tool.

5 Case Study

We use the motivating example ERP system as a case study to demonstrate how endpoint can be modeled by using our TeeVML in interface layers. We also describe the steps to convert the endpoint models to executable forms and integrate them to the domain framework in a target environment.

5.1 Signature Modeling

Signature modeling starts from specifying endpoint level properties. Then, WSDL sub-DSVL is used to instantiate the five WSDL entities by providing their names and relevant information. They are linked together by using either a composition or an association relationship. All the entities have just one instance, except for the operation. The number of the operation instances depends on the services provided by the endpoint.

We use the operation *paymentrequest* as an example to show how an operation can be modeled. The operation is instantiated by assigning the operation name as *paymentrequest* and pattern as “in-out”. Then, Operation sub-DSVL is used to specify the *paymentrequest_request* and *paymentrequest_response* messages in the operation. The request message label is “in”, and response message label is “out”.

Message sub-DSVL is used to define message elements. The request message contains only one element *pono*, and it is defined as integer and mandatory. Since a valid *pono* is a five-digit integer, the element’s minimum field is specified as 10000 and

maximum field as 99999. The response message consists of three elements: *amount*, *errorcode* and *errormessage*. They are placed in the message in alphabetic order. The *amount* is a float data type, *errorcode* integer and *errormessage* string.

Figure 8 shows the ERP system endpoint signature model. It contains the top-level WSDL model (refer to Fig. 8a, we only show five operations for a better view representation), the middle-level *paymentrequest* operation model (Fig. 8b), and the bottom-level request and response message models (Fig. 8c).

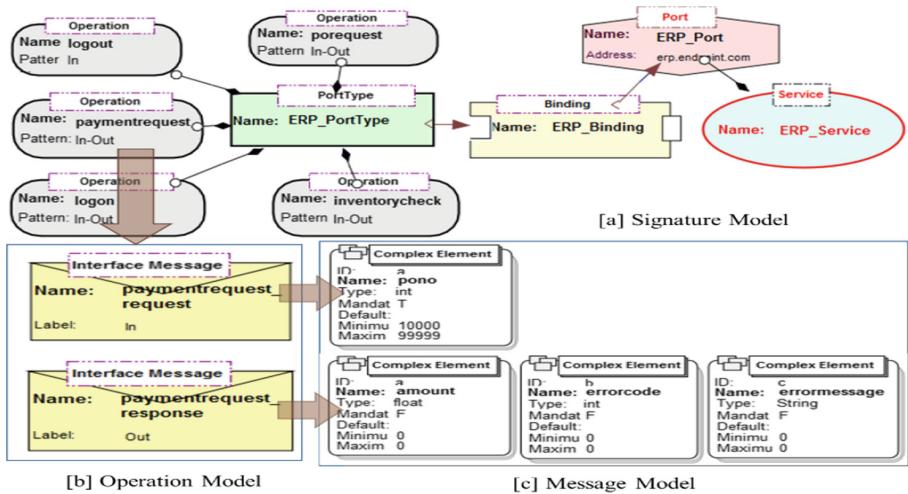


Fig. 8. The example endpoint signature model [6].

5.2 Protocol Modeling

Figure 9 shows the endpoint protocol model, where a purchase process progresses in clockwise direction. To demonstrate how to model the endpoint protocol layer, we select three typical protocol behaviors of interactive session management, constraint state transition and transition iteration. They are marked as A, B and C in the diagram, respectively.

A – Session management: Endpoint protocol modeling starts from specifying an interactive session by using a logon transition relationship from Idle state to Home state. To terminate the session, a logout transition relationship is used in the opposite direction. The session can also be terminated by a timeout event, which is specified by using a timeout relationship linking a “from” state to a “to” state.

B – Constraint transition relationship: When the endpoint is at *inventorycheck* state, there are alternative flows either to *supplierpo* or *paymentrequest* and the choice of the flows depends on whether the purchase item stock can meet the PO requirement. To specify this type of state transitions, the constraint transition relationship is used and it links the *inventorycheck* state to the *supplierpo* state. The constraint condition is specified using the relationship dialog box by comparing the quantity parameter of

porequest request with the inventory parameter of *inventorycheck* response. If the former is greater than the latter, the state transition will happen. Similarly, we specify another constraint transition from the *inventorycheck* state to the *paymentrequest* state, and the constraint condition is the item stock greater than or equal to the PO quantity.

C – Transition iteration: A loop relationship is used to specify that all the operations between the “from” state and “to” state of the loop relationship will be executed repeatedly. The approval process of a supplier PO is an iteration, which includes an *approvalnotification* and a *supplierpoapproval* operations. The approval process starts from the immediate manager of the purchaser until the manager with authority for the PO amount.

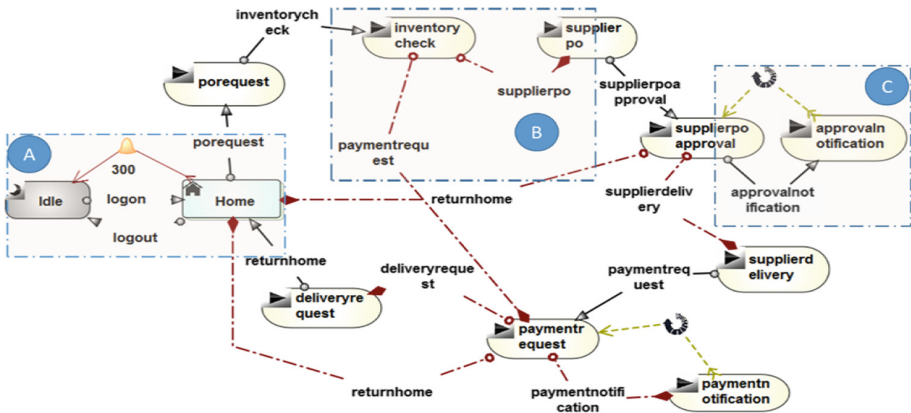


Fig. 9. The example endpoint protocol model [6].

5.3 Behavior Modeling

An endpoint behavior model consists of unrelated service nodes for all provided operations, and we use one operation *paymentrequest* as example to explain how endpoint behavior is modeled. Figure 10a shows the *paymentrequest* service node, which consists of two sub nodes: *poinformationretrieve* to retrieve the PO information from tables and *poamountcalculation* to work out the total PO amount. These two nodes are placed between an entrance and exit bars.

We select the *poinformationretrieve* node to show how Behaviour DSVL primitive visual constructs are used to implement business logics. Figure 10b illustrates the data query operations and dataflows within the *poinformationretrieve* node. The node has one input “pono”, and generates four outputs: “quantity”, “unitprice”, “discount” and “FatalError”. The node contains three data query operations: (1) to retrieve PO “category”, “item”, “quantity” and “client” from PurchaseOrderTable by the “pono”; (2) to retrieve “unitprice” from ProductTable by the “category” and “item”; and (3) to retrieve “discount” from ClientTable by the “client”. If searching records are found, searching results will be placed on the normal output port (black circle) of data store operator.

Otherwise, an “FatalError” variable will be assigned by following the exceptional output port (yellow circle).

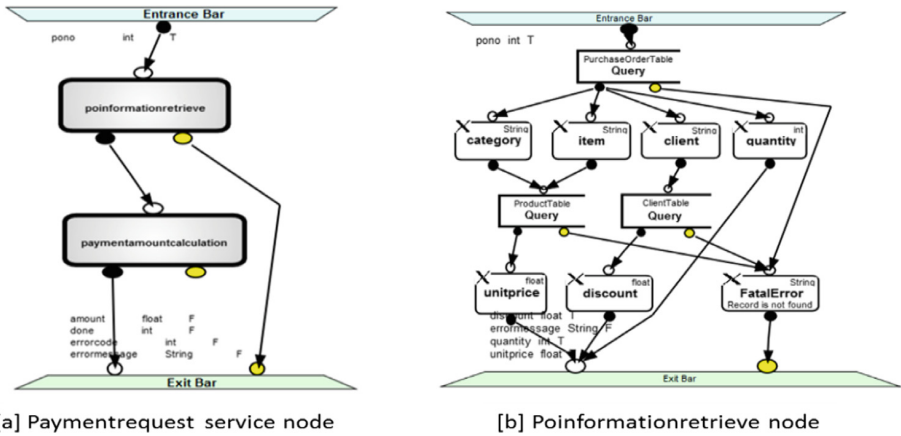


Fig. 10. The example endpoint Paymentrequest operation behavior model. (Color figure online)

5.4 Testing Environment Generation

Our approach provides a very simple and easy way to generate operational endpoints from their models. There are three tasks including: (1) to create two Java project folders for hosting server side and client side codes; (2) to transform models to codes by code generators and copy them to the server project folder; and (3) to run our supporting toolset for packaging Tomcat service and providing testing service to SUTs. To automate endpoint generation process, we have created an Apache Ant build file.

Figure 11 illustrates a deployment view on how an endpoint provides integration testing service to a SUT. The left-hand side is the emulated endpoint hosted in a Tomcat application server, its protocol and behavior classes are integrated into the Axis2 Skeleton class for performing SUT operation requests validation. The grey areas at the bottom of both sides are the Axis2 Web Service engine for encoding and decoding SOAP messages exchanged between the endpoint and the SUT. A SUT is located on the right-hand side at the top, communicating with the Axis2 Stub class through an API class. The SUT invokes the endpoint service through accessing Tomcat Axis2 service URL using SOAP over HTTP application protocol. To capture and see the exchanged messages, we use TCPMon tool [21] to act as an intermediary between the SUT and endpoint. TCPMon accepts connection from the SUT on one port and forwards the incoming traffic to the endpoint running on another port.

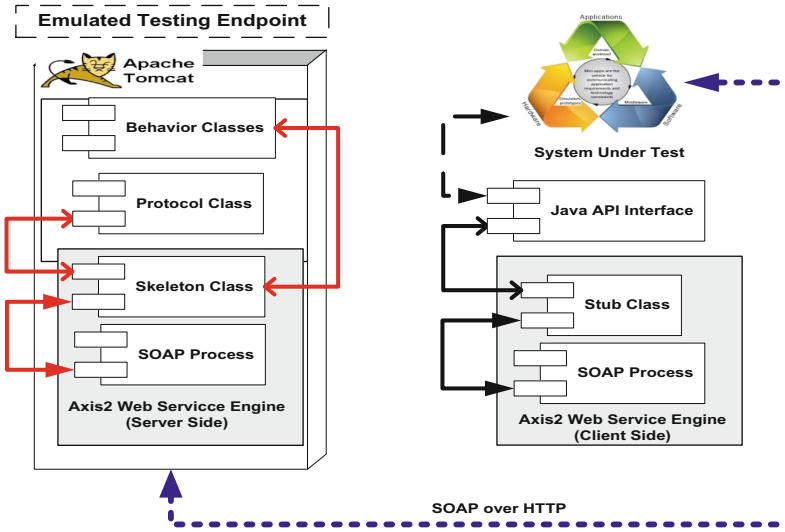


Fig. 11. The deployment view of an endpoint and its SUT.

6 Evaluation

To assess how well the issues related to the three research questions have been addressed by our approach, we have developed three evaluation criteria:

- *Testing functionality* (addressing RQ1) – the approach should be able to develop a wide variety of endpoints, which could be used to capture all the interface defects of a new or an existing system under test;
- *Development productivity* (addressing RQ2) – the approach should have high endpoint development productivity with less development effort and time;
- *Ease of use* (addressing RQ3) – the approach should be easy to learn and adapt by non-technical background users.

These criteria were first assessed by a technical comparison of our approach with the currently available endpoint emulation approaches. This comparison motivated our new DSM approach to address the shortcomings of the existing approaches. After our approach was ready to use, we also conducted a user survey to evaluate the extent to which our approach was accepted by software testing experts and developers.

6.1 Technical Comparison

Currently, there are two types of TEE approaches to develop integration testing environments: specification-based by manual coding (also called “manual coding”) and interactive trace data record-and-replay (also called “interactive tracing”). The manual coding approaches are used by IT professionals to develop simplified versions of applications with external behavior manually [22, 23]. They perform this using available knowledge of the underlying message syntax, interaction protocol and

behavior. The interactive tracing approaches create endpoint models from recorded request-response pairs between the endpoint system and an earlier version of a SUT automatically [24, 25]. Each endpoint’s simulated response is generated by finding a closely matched request in the recorded trace database.

To compare these two types of approaches with our new TeeVML, we use the three defined evaluation criteria and look into what and how some key techniques these approaches adopt to meet these criteria. Table 4 presents the comparison. From the development productivity and ease of use point of view, the interactive tracing approaches are the highest, as endpoints are created automatically. However, these approaches have two key shortcomings in terms of the testing functionality. One is their usability, which is subject to the availability of interactive tracing data. Another one is that they cannot report defect type and cause information. In contrast, the manual coding approaches and TeeVML need to develop endpoints by IT professionals. As TeeVML uses higher level abstraction models than code to express design intent, it achieves better endpoint development productivity and ease of use.

Table 4. The comparison of TEE approaches’ techniques.

Manual coding	Interactive tracing	TeeVML
<i>Testing functionality</i>		
The key motivation of these approaches is to provide performance testing by emulating large number of endpoints. To achieve this objective, these approaches adopt a light-weight architecture design and some testing features are deliberately neglected. Dynamic protocol behavior cannot be modeled, as state transition is triggered only by an operation. Unless great effort is made, behavior layer modeling will be limited	To provide integration testing, these approaches search for the right request matching on data byte level without any knowledge about upper-level message syntax. They can only tell whether a test is passed or failed, but cannot provide any defect information. These approaches are not usable for testing a new application, as its trace data are not available	Endpoints modeled by TeeVML provide integration testing functionality from signature, protocol and behavior abstraction layers. The signature layer model supports all RPC communication styles; the protocol layer can model both static and dynamic protocol behaviors; and the behavior layer uses a hierarchical structure dataflow programming for modeling complicated logic implementations
<i>Development productivity</i>		
The approaches adopt a modular architecture design, where an endpoint type dependent message engine module is separated from an endpoint type independent network and a system configuration	Endpoint is created by recording the interactive tracing data between the endpoint application and an earlier version of the SUT application. These approaches do not need any endpoint development	An endpoint is modeled by layers, and layer models are transformed to executable code. The key solution to productivity improvement is to maximize components reusability. We have adopted multi-level design for

(continued)

Table 4. (continued)

Manual coding	Interactive tracing	TeeVML
modules. However, as the message engine is coded manually, significant amount of development effort is needed for each new endpoint type	work, but some effort on trace data recording	Signature DSL and node hierarchical structure for Behavior DSL
<i>Ease of use</i>		
To develop an endpoint, developers must have both business domain knowledge and programming skills	Neither business domain knowledge nor programming skills are required. But, users must be trained to use the tool	Developers must have business domain knowledge, and some modeling skill is preferred

6.2 User Survey

User surveys incorporate a list of questions to extract specific data from a particular group of people. They provide a comprehensive mechanism for collecting information to describe, compare and explain knowledge, attitudes and behaviors of survey participants [26]. Survey results are used to improve products' quality and functionality by guiding and correcting the design, development and refinement.

Experiment Setup

We conducted our user survey in two phases. In the first phase, we extracted testing experts' opinions on what testing features they valued in endpoints and what functionality TeeVML should provide to develop endpoints. We introduced our TeeVML and endpoint testing functionality to the participants by using a PowerPoint presentation, then interviewed them and answered their queries. Sixteen testing experts were invited to participate in the survey, and most of them (94%) had more than one year solid testing experience and were knowledgeable about SIT. In the second phase, we assessed TeeVML's usability by collecting software developers' experience with the tool to work on an assigned task. We wanted them to compare TeeVML with a third-generation language they were familiar with, as the way the manual coding approaches do. Total of 19 software developers and IT research students took part in the survey. Most of them (95%) had IT background and (63%) were familiar with software modeling.

All the participants were asked to fill an online questionnaire, after finishing their user survey. The questionnaires include 5-point Likert Scale (5 to 1 representing strongly agree to strongly disagree), single-choice, multiple-choice, and open ended questions. For the 5-point Likert Scale questions, in favour responses encompass the answers of either 5 or 4 for a positive question, and 1 or 2 for a negative question. We counted the number of in favour responses to measure the degree of acceptance to a question statement. There were total 58 questions for both Phase One and Two, and we

only select some of them for this paper results presentation. The full result reports can be accessed at: <https://sites.google.com/site/teevmlapsec/>.

Table 5. Selected questions and responses from Phase One survey report.

[a] Likert scale questions						
No	Statement	Frequency				
		5	4	3	2	1
Q8	In your opinion, an emulated testing environment is useful for an application inter-connectivity and inter-operability test	8	6	0	1	1
Q17	It is useful for an emulated testing environment to provide signature testing functionality to its system under test	7	7	1	1	0
Q21	It is useful for an emulated testing environment to provide interactive protocol testing functionality to its system under test	12	4	0	0	0
Q25	It is useful for an emulated testing environment to provide interactive behavior testing functionality to its system under test	6	8	1	1	0
[b] Multiple choice questions						
No	Question statement and choices	Frequency				
Q13	What are the main motivations for you to use emulated testing environment?					
	<i>Cost saving on application software and hardware investment</i>	14				
	<i>Effort saving on application installation and maintenance</i>	10				
	<i>Lack of application knowledge</i>	5				
	<i>Early detection of interface defects</i>	15				
Q14	What are your main concerns, which could prevent you from using emulated testing environment?					
	<i>Extra development effort on testing endpoints</i>	6				
	<i>Learning a new technology</i>	6				
	<i>Inadequate testing functionality</i>	7				
	<i>Emulation accuracy</i>	7				
	<i>Result reliability</i>	12				

Survey Results Analysis – Phase One

We select a few typical questions in Table 5 to analyze them from two different angles: One is about participants' acceptance of an endpoint as a whole and by each interface layer from functionality point of view. Another is to find out the possible reasons why participants would consider using (or not using) our endpoints in their future projects.

Q8 reflects the overall usefulness of endpoints for conducting SIT. The responses to this question are quite positive with 14 out of 16 (87.5%) participants in favour. This is a good indication of the participants' acceptance of endpoints modeled by TeeVML.

To further investigate each interface layer, Q17, Q21 and Q25 are used to get participants' opinion on the usefulness of modeling signature, protocol, behavior layers, respectively. We can see that the protocol layer (Q21) received in favour responses from all participants. We believe one of the main reasons why all participants wanted to have protocol testing is that most applications do not have a well-documented protocol specification. Therefore, protocol related defects can only be found by SIT. On the other hand, the signature layer (Q17) had slightly less in favour response rate compared to the protocol layer. The signature correctness is a must for a client to access operations provided by a server. However, a few participants might have thought that endpoint signature could be easily coded and verified against product interface specification, hence actual testing would be unnecessary.

Q13 is a multiple-choice question, and lists four reasons why users want to use endpoints. Responses to Q13 indicate that the top reason for using endpoints was early detection of interface errors. In current practice, SIT is normally conducted during the later stages of software development lifecycle. This is partly because integration testing environment is not available before then. If a rapid and cheap solution for testing environment deployment was available, developers might have preferred to conduct at least part of SIT earlier. Q14 indicates that most participants' concerns were on the reliability of endpoint testing results. The main reason could be that software developers are used to using real applications for their SIT. However, an endpoint is actually a simplified version of its real application. Often, some implementation aspects of the application are neglected and treated as useless for SIT. This might have some impacts on SUT testing results. Our survey results indicate the importance of conducting endpoint functionality design before modeling it.

Survey Results Analysis – Phase Two

Giving that the participants have used our tool to model an endpoint operation, we want them to provide their opinions on whether the tool is ease of use and how much endpoint development productivity can be improved. The former uses the 10 questions from System Usability Scale (SUS), and the latter is based on two questions of the actual time spending on the task and a subjective comparison with a third-generation language.

SUS is a simple, 10 5-point Likert Scale questions (see Table 6) to give a global view of the subjective assessment of a product's usability [27]. SUS yields a single number by adding up the score contributions from each question and multiplying by 2.5 to represent a composite measurement of the overall usability of the system being studied. By a statistic study over a large number of products, the overall SUS mean score was 68 [28]. By this survey, TeeVML overall SUS score is calculated as 78.3 out of 100 points, which is equal to 83% from a percentile ranks for raw SUS scores table [28]. From another angle, our SUS score falls between "Good" and "Excellence" in the adjective ranges of the "Acceptability" scoring system proposed by Bangor et al. from a study on numerous products [29].

Table 6. System Usability Scale questions [27].

No	Statement
Q12	You would like to use the tool in your future project
Q13	You found the tool unnecessarily complex
Q14	You found the tool was easy to use
Q15	You would need support to be able to use the tool
Q16	You found the various features of the tool were well integrated
Q17	You found there was too much inconsistency in the tool
Q18	You would image that most people would learn to use the tool very quickly
Q19	You found the tool very cumbersome to use
Q20	You felt very confident using the tool
Q21	You needed to learn a lot of things before you could get going with the tool

Table 7 represents survey results for TeeVML's productivity to model endpoints. For Q9, 79% participants could finish their task within 30 min, which is a typical endpoint operation modeling. Based on this result, we can generalize that it is possible to model a relatively complex endpoint with more than 10 operations within a day through using our TeeVML. From Q22 we can see that more than half of respondents (57.8%) agreed that using TeeVML would reduce "50%–80%" or "80%+" of the time duration they use for endpoint development. No participant voted "Almost the same". As the results from these two questions, we can conclude that most participants agree that our TeeVML could increase endpoint development productivity significantly comparing with traditional manual coding approaches.

Table 7. Phase Two survey results for development productivity.

No	Statement	Frequency
Q9	How long did it take you to complete the task?	
	10–15 min	1
	16–20 min	4
	21–25 min	7
	26–30 min	3
	30+ min	4
Q22	In your opinion, comparing to a third generation language (e.g. Java) you are familiar with, how much would a typical endpoint development effort be reduced by using the tool?	
	<i>Almost the same</i>	0
	10–25%	2
	26–50%	6
	51–80%	9
	81%+	2

7 Related Work

Over the years, many approaches have been proposed to develop testing environments. Hardware virtualization tools, such as VMWare [30] and VirtualBox [31], provide management and control over virtual testing servers and they are capable in hosting many endpoint systems in one or a few machines. However, some applications need to be run in shared memory space and they cannot be virtualized. Method stubs [32] and mock objects [33] are programming approaches to mimic testing applications. Their key advantages are low cost and quick deployment. But these approaches abstract away from communication complexities which may significantly impact on the results encountered in the real deployment.

To address both static and dynamic issues related to software components interactions, Han first proposed a rich interface definition framework with logically separated layers [34]: signature, constraints, configurations, and a quality aspect across the three layers. Han's framework defines how to select and reuse a software component, not just based on static component signature, but also on other runtime aspects as well. From a service viewpoint, Beugnard et al. defined a four-level software component contract template with increasingly negotiable properties along with the levels [35]. Our approach on the other hand, focuses on how requests are to be processed in a layered manner and interface defects are captured by endpoints.

For protocol modeling, some researchers used a FSM [36, 37] or a formal protocol specification [38, 39] to validate operations sequence for different endpoint states. However, Wehrheim et al. argued that the use of operation name alone might not be sufficient to trigger a state transition for a realistic endpoint [40]. To deal with the so-called incomplete protocol specification, an EFSM-based protocol modelling calculus were proposed for specifying operation parameters and return values as runtime constraints. Although, various notions for protocol specification have been suggested, there are still some issues to be solved. One is the lack of concrete implementation solutions to capture endpoint runtime aspects. Another one is the textual form they used for writing state transition rules, and this will make protocol modeling difficult.

Software interactive behaviors can be modeled either externally or internally. Software behavioral interface specification [41] and programming from specification [42] are the external approaches, they model interactive behaviors by defining pre/post conditions to bind servers and their clients. As internal approaches, Business Process Model and Notation (BPMN) [43] and DataFlow Programming (DFP) [13] provide graphical notations for specifying internal data processes and flow controls. In general, external approaches and BPMN require extensive modeling and programming work. While, DFP languages are ease of use with user-friendly interface. But, they are less expressive and only suitable for a specific domain. In contrast to these approaches, our behavior DSVL is ease of use by dragging-and-dropping visual symbols. To handle complicated business logics, hierarchical nodes tree structure is adopted.

UML is a widely used general purpose modeling language, focusing on the definition of system static and dynamic behaviors [44]. Specifically related to our work, UML provides: (1) a testing profile to provide a generic extension mechanism for the automation of test generation processes [45], (2) state charts to simulate finite-state

automaton [46], and (3) activity diagrams to graphically represent workflows of stepwise activities and actions [47]. Two main problems with using UML to define new modeling languages [48] are that it is usually hard to remove parts of UML that are not relevant or need to be restricted in a specialized language and all the diagram types have restrictions based on the UML semantics.

8 Conclusion and Future Work

Aiming to achieve high development productivity and ease of use for domain experts, we have proposed a DSM approach for testing environment emulation. Our approach is based on a new software interfaces description framework to abstract an endpoint into three message processing layers, and a suite of DSVLs have been developed for modeling these layers. Using this layered modeling, our approach supports partial endpoint development, where an endpoint may have only one or two of these layers to meet SUT testing requirements. For a SUT without dynamic interactive aspects, the endpoint behavior layer could be ignored.

A fully functional endpoint should also be able to test SUT's QoS aspects. These QoS aspects may include security, performance, robustness, etc. For example, applications may put extra security constraints on the validity of operation requests. Some of the constraints are role-based, so that some operations are accessible only to a certain group of users. Other constraints are security policy related, such as restriction on available time or specific pattern required for some operation parameters. Object-oriented programming has higher expressive power than imperative and procedural programming by supporting inheritance, polymorphism, encapsulation, etc. Making our Behavior DSVL object-oriented can simplify behavior modeling, increase development productivity and output accuracy, and have a better diagrammatic view of behavior model. Furthermore, to reduce modeling overhead in effort and time, some special purpose utility nodes should be provided with Behavior DSVL for common modeling features. These and others could be part of our future work.

Acknowledgements. Support for this work from an Australian Postgraduate Award for the first author and partial support for this work from ARC Discovery Project DP140102185 is greatly acknowledged.

References

1. Accenture: Accenture Technology Vision 2015 (2015)
2. Dustin, E.: Effective Software Testing: 50 Ways to Improve Your Software Testing. Addison-Wesley Longman Publishing Co., Boston (2002)
3. Liu, J., Grundy, J., Avazpour, I., Abdelrazek, M.: TeeVML: tool support for semi-automatic integration testing environment emulation. Presented at the IEEE/ACM International Conference on Automated Software Engineering, Singapore (2016)
4. Jayasinghe, D.: Quickstart Apache Axis2. Packt Publishing Ltd, Birmingham (2008)
5. Vukotic, A., Goodwill, J.: Apache Tomcat 7. Apress, New York City (2011)

6. Liu, J., Grundy, J., Abdelrazek, M., Avazpour, I.: Testing environment emulation - a model-based approach. Presented at the 5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD), Porto, Portugal (2017)
7. Yadav, R.: Oracle PeopleSoft Enterprise Financial Management 9.1 Implementation: An Exhaustive Resource for PeopleSoft Financials Application Practitioners to Understand Core Concepts, Configurations, and Business Processes. Packt Pub., Birmingham (2011)
8. Wong, T.: *Salesforce.com For Dummies*, 4th edn. Wiley, Hoboken (2010)
9. Thurlow, R.: *RPC: Remote Procedure Call Protocol Specification Version 2*. Edited by The Internet Engineering Task Force (2009)
10. IETF: *Lightweight Directory Access Protocol (LDAP) v3*. Edited by The Internet Engineering Task Force (2006)
11. Dorsten, G.J.V., Spruit, A., Barendsen, A.: *Core Banking Systems Survey (2008)*. https://www.nl.capgemini.com/resource-file-access/resource/pdf/Core_Banking_Systems_Survey_2008_0.pdf
12. Christensen, E., Curbera, F., Meredith, G.: *Web Services Description Language (WSDL) 1.1*. W3C. Note 15 (2001). www.w3.org/TR/wsdl2001
13. Sousa, T.B.: *Dataflow programming concept, languages and applications*. In: *Doctoral Symposium on Informatics Engineering (2012)*
14. Boshernitsan, M., Downes, M.: *Visual programming languages: a survey*. *Computer Science (2004)*
15. Larkin, J.H., Simon, H.A.: Why a diagram is (sometimes) worth ten thousand words. *Cognit. Sci.* **11**, 65–100 (1987)
16. Moody, D.L.: The “Physics” of notations: towards a scientific basis for constructing visual notations in software engineering. *IEEE Trans. Softw. Eng.* **35**, 756–779 (2009)
17. Liu, J., Grundy, J., Avazpour, I., Abdelrazek, M.: *A domain-specific visual modeling language for testing environment emulation*. Presented at the IEEE Symposium on Visual Languages and Human-Centric Computing, Cambridge, UK (2016)
18. Liu, J.: *Model-driven endpoint development for testing environment emulation*. Ph.D. thesis, Swinburne University of Technology, Melbourne, Australia (2017)
19. Kelly, S., Tolvanen, J.P.: *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley, Hoboken (2008)
20. Dai, G., Bai, X., Wang, Y., Dai, F.: *Contract-based testing for web services*. In: *31st Annual International Computer Software and Applications Conference, COMPSAC (2007)*
21. Apache TCPMon (2013). <https://ws.apache.org/tcpmon/index.html>
22. Hine, C., Schneider, J.-G., Han, J., Versteeg, S.: *Scalable emulation of enterprise systems*. In: *Software Engineering Conference, Australian (2009)*
23. Yu, J., Han, J., Schneider, J.-G., Hine, C., Versteeg, S.: *A virtual deployment testing environment for enterprise software systems*. Presented at the Proceedings of the 8th International ACM SIGSOFT Conference on Quality of Software Architectures, Italy (2012)
24. Du, M., Schneider, J.-G., Hine, C., Grundy, J., Versteeg, S.: *Generating service models by trace subsequence substitution*. Presented at the Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures, Canada (2013)
25. Giudice, D.L.: *The Forrester Wave™: Service Virtualization And Testing Solutions (2014)*
26. Pfleeger, S.L., Kitchenham, B.A.: *Principles of survey research: part 1: turning lemons into lemonade*. *SIGSOFT Softw. Eng. Notes* **26**, 16–18 (2001)
27. Brooke, J.: *SUS-a quick and dirty usability scale*. In: *Usability evaluation in industry (1996)*
28. Sauro, J., Lewis, J.R.: *Quantifying the User Experience: Practical Statistics for User Research*. Morgan Kaufmann Publishers Inc., Burlington (2012)
29. Bangor, A., Kortum, P.T., Miller, J.T.: *An empirical evaluation of the system usability scale*. *Int. J. Hum.-Comput. Interact.* **24**, 574–594 (2008)

30. Sugeran, J., Venkitachalam, G., Lim, B.-H.: Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. Presented at the Proceedings of the General Track: USENIX Annual Technical Conference (2001)
31. Watson, J.: VirtualBox: bits and bytes masquerading as machines. *Linux J.* (2008)
32. Gibbons, P.B.: A stub generator for multilanguage RPC in heterogeneous environments. *IEEE Trans. Softw. Eng.* **13**, 77–87 (1987)
33. Freeman, S., Mackinnon, T., Pryce, N., Walnes, J.: Mock roles, objects. Presented at the Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, Canada (2004)
34. Han, J.: Rich Interface Specification for Software Components. Peninsula School of Computing and Information Technology Monash University, McMahons Road Frankston, Australia (2000)
35. Beugnard, A., Jézéquel, J.-M., Plouzeau, N., Watkins, D.: Making components contract aware. *Computer* **32**, 38–45 (1999)
36. De Alfaro, L., Henzinger, T.A.: Interface automata. In: ACM SIGSOFT Software Engineering Notes (2001)
37. Endriss, U., Maudet, N., Sadri, F., Toni, F.: Protocol conformance for logic-based agents. In: *IJCAI* (2003)
38. Plasil, F., Visnovsky, S., Besta, M.: Bounding component behavior via protocols. In: *Technology of Object-Oriented Languages and Systems, TOOLS 30 Proceedings* (1999)
39. Jin, Y., Han, J.: Specifying interaction constraints of software components for better understandability and interoperability. In: Franch, X., Port, D. (eds.) *ICCBSS 2005. LNCS*, vol. 3412, pp. 54–64. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30587-3_16
40. Wehrheim, H., Reussner, R.H.: Towards more realistic component protocol modelling with finite state machines. *UNU-IIST* (2006)
41. Hatcliff, J., Leavens, G.T., Leino, K.R.M., Muller, P., Parkinson, M.: Behavioral interface specification languages. *ACM Comput. Surv.* **44**, 16 (2012)
42. Morgan, C.: *Programming from SPECIFICATIONS*. Prentice-Hall Inc., Upper Saddle River (1990)
43. von Rosing, M., White, S., Cummins, F., de Man, H.: *Business Process Model and Notation —BPMN* (2015)
44. Jacobson, I., Booch, G., Rumbaugh, J., Rumbaugh, J., Booch, G.: *The unified software development process*. Addison-wesley Reading, Boston (1999)
45. Schieferdecker, I., Dai, Z.R., Grabowski, J., Rennoch, A.: The UML 2.0 testing profile and its relation to TTCN-3. In: Hogrefe, D., Wiles, A. (eds.) *TestCom 2003. LNCS*, vol. 2644, pp. 79–94. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44830-6_7
46. Zhang, S.J., Liu, Y.: An automatic approach to model checking UML state machines. In: *Fourth International Conference on Secure Software Integration and Reliability Improvement Companion (SSIRI-C)* (2010)
47. Dumas, M., ter Hofstede, A.H.M.: UML activity diagrams as a workflow specification language. In: Gogolla, M., Kobryn, C. (eds.) *UML 2001. LNCS*, vol. 2185, pp. 76–90. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45441-1_7
48. Abouzahra, A., Bézivin, J., Del Fabro, M.D., Jouault, F.: A practical approach to bridging domain specific languages with UML profiles. In: *Proceedings of the Best Practices for Model Driven Software Development at OOPSLA* (2005)



A Framework for UML-Based Component-Based Design and Code Generation for Reactive Systems

Van Cam Pham, Ansgar Radermacher^(✉), Sébastien Gérard, and Shuai Li

CEA, LIST, Laboratory of Model Driven Engineering for Embedded Systems,
P.C. 174, Gif-sur-Yvette 91191, France

{vancam.pham,ansgar.radermacher,sebastien.gerard,shuai.li}@cea.fr

Abstract. One way to design complex systems is to use an event-driven architecture. Model Driven Engineering (MDE) promotes the use of different abstract concepts, among which are the UML state machine, composite structure elements and their associated visualizations, as a powerful means to design such an architecture. MDE seeks to increase software productivity by automatically generating executable code from state machines and composite structures. To this end, a code generation approach in MDE should support all model elements used at the design level. However, existing code generation approaches and tools are still limited, especially when considering concurrency, event types, and pseudo states such as history and junction. Furthermore, in the literature, the combination of component-based design and UML state machines is supported by only a few existing approaches. This paper explores this combination and provides code generation patterns and framework-based tooling support for the combination and complete and efficient code generation from UML state machines. We extend a well-known state machine code generation pattern with concurrency support. In order to verify the semantics of generated code, we execute code generated by the proposed framework with a set of state-machine examples that are part of a test-suite described in the recent OMG standard Precise Semantics Of State Machine. The traced execution results comply with the standard and are a good hint that the execution is semantically correct. Regarding code efficiency, the code generated by our approach supports multi-thread-based concurrency, and the size of the generated code is smaller compared to considered approaches. Moreover, we demonstrate the practicality, feasibility, and scalability of the proposed approach with two case studies.

Keywords: UML state machine · Code generation
Semantics-conformance · Efficiency · Events · C++
Composite structures · Component-based design · Flow ports
Service ports · Components

1 Introduction

The UML state machine (USM) and composite structure (diagram) elements [1] and their associated visualizations are an efficient means to describe the behavior of event-driven architecture [2, 3]. In Model-Driven Engineering (MDE) [4], the system architecture can be represented as models by using the UML (model) elements (e.g. states and transitions of UML state machine, and ports and connectors for component-based modeling). Existing tools and approaches automatically translate models into executable code. However, despite many advantages of MDE and the UML elements, they are not widely adopted as a recent survey revealed [5]. This is partially due to poor support for code generation [6].

On one hand, the usefulness of these model elements, especially the UML state machine elements, is being enriched by OMG by providing more concepts and their precise semantics such as pseudo states and composite state machines. On the other hand, existing code generation tools and approaches have some issues regarding completeness, semantics and efficiency of generated code. Existing approaches either support a subset of UML state machine modeling concepts or handle composite state machines by flattening into simple ones with a combinatorial explosion of states, and excessive generated code [7]. Furthermore, in the literature, the processes of generating code from UML state machines and composite structures are often found separately from each other. The combination of these elements for component-based reactive system modeling are not clearly explored. In particular, the following issues are identified.

Completeness: Existing tools and approaches mainly focus on the sequential aspect while the concurrency of state machines is limitedly supported. Pseudo states are not rigorously supported by existing tools such as Rhapsody [8]. Designers are then restricted to a subset of UML state machine concepts during design.

Efficiency: Code generated from tools such as Rhapsody [3] and FXU [9] depends on the libraries provided by the tool vendor. Event processing speed and executable size of generated code are not optimized [10].

Semantics: The semantics of UML state machine is defined by a recent OMG-standardized: Precise Semantics of state machine (PSSM) [11]. This standard is not (yet) taken into account for validating the runtime execution semantics of generated code.

Combination of State Machine and Component-Based Design: The composite structure elements such as ports and connectors are used for component-based design. A state machine can model the behavior of a component. However, how the state machine-related elements such as events are related to parts and ports for modeling a complete software system is supported by only a few existing approaches [2].

Given the above issues, the objectives of this article are to (1) present a novel code generation pattern for UML state machine elements and composite structure elements; (2) to explore the collaboration between the UML state machine

and composite structure diagrams for modeling reactive systems; and (3) to provide a tooling support into the Papyrus modeling tool. The latter offers efficient code generated from UML state machines with full concepts to reduce the modeling-implementation gap. The proposed pattern for state machines extends IF-ELSE constructions with our support for concurrency. Runtime execution of generated code is experimented with the PSSM test suite.

To sum up, the contributions of this paper are: (1) an approach and tooling support for code generation from UML composite structure elements and state machines with full features; (2) an empirical study on the semantic-conformance and efficiency of generated code; (3) evaluation of the practicality and usability of UML state machine and composite structure elements for modeling a Traffic Light Controller (TLC) simulation case study; and (4) a real case study using code generation for an embedded system - Lego Car factory.

We assume that readers of this paper have knowledge about UML composite structure, UML state machine elements, and their basic execution semantics.

Compared to the previous version of the paper presented in the 5th International Conference on Model-Driven Engineering and Software Development [12], this present paper contains two completely new sections (Sects. 6 and 9), which discuss modeling and code generation for composite structure elements and a Lego Car case study, respectively. A new subsection (Subsect. 2.2) discusses the use of composite structures and state machines for component-based design of reactive systems is also added. Furthermore, Sects. 1 and 10 are entirely modified for added contributions.

The remaining of this paper is organized as follows: Sect. 2 describes the modeling of applications using UML state machines and composite structure elements and their combination. Section 3 mentions the code generation features of our extension of Papyrus. Section 4 shows thread-based concurrency. Based on this latter, the code generation patterns for state machines and composite structures are proposed in Sects. 5 and 6, respectively. The implementation and empirical evaluation are reported in Sect. 7. The applications of our extension to the TLC and Lego Car factory case study are presented in Sects. 8 and 9, respectively. Section 10 discusses related work. The conclusion and future work are presented in Sect. 11.

2 UML State Machine, Composite Structure and Events

This section presents overview of using UML state machines and composite structures, especially ports and connectors, for modeling and designing reactive software applications.

2.1 UML State Machine and Events

A state machine is used for describing the behavior of either a class or a component in component-based design using the composite structure elements. In the following, we commonly use the term *class* for either class or component.

The state machine accepts external and internal events. In UML, there are four event types: *CallEvent*, *SignalEvent*, *TimeEvent*, and *ChangeEvent*. A call event, which is associated with an operation/method, is emitted if the operation is invoked. Call events are processed synchronously meaning that it runs within the thread of the operation caller. Other events are asynchronously processed meaning that these events received by the class are stored in an event queue which is maintained by the class at runtime for later processing. A simple usage of call events is in user interaction (UI) applications, in which users click on a button. The click then emits an event and calls user code for synchronously processing the event to respond to the users.

A signal event is associated with a UML signal and is emitted if the class/component receives an instance of the signal type. The signal instance can be sent from either environment code or other classes to the reception class/component via one of the ports of the reception component (see Subsect. 2.2 for more details).

A time event specifies the time of occurrence relative to a starting time. The latter is defined as the time when a state with an outgoing transition triggered by the time event is entered. The time event is emitted if this accepting state remains active longer than the relative time of occurrence. The transition is then triggered once the event is emitted. In other words, the source state of a transition triggered by a time event will remain active for a maximal amount of time specified by the time event. A change event has a boolean expression and is fired if the expression's value changes from false to true. Note that unlike call and signal events, time and change events are automatically fired inside the class.

Deferred Events: A state can specify to one or more deferred events. If an event specified as deferred by a state, it will be not processed while the state remains active. The deference of events is used to postpone the processing of some low-priority events while the state machine is in a certain state.

2.2 Combination of State Machines and Ports and Connectors

The UML composite structure diagram provides notations for modeling component-based software. Each component can own ports, which can communicate with other ports via connectors. A connector connects two ports. It represents a link between components. We show how ports, connectors and state machine elements can be combined.

As previously discussed, call and signal events are external, meaning that instances of these events are sent from other components. The state machine of the component, that receives the events, then processes them. Specifically, ports with interfaces are combined with call events and flow ports are combined with signal events. We explain the combination in the following.

Service Port and Call Event: In UML, a port can have required and/or provided interfaces. We call this port a service port, which requests and/or provides services as invocations of the operations of the interfaces. A service port in our

approach can have only one required and/or a provided interface. For example, Fig. 1(a) shows a very simple system consisting two component parts *compA* and *compB*. *compA* has a p_A port with *IExample* as its required interface, which is provided by the p_B port and implemented by *compB*. There is a connector between the two ports. The *compA* component can request services of *compB* via the p_A port. Assuming that the behavior of the *compB* is described by a state machine *CompBMachine* as in Fig. 1(b) and one of the transitions of the state machine is triggered by a call event *CE_Add*. If the latter is, for example, associated with an *add* operation of *IExample* and implemented by *compB*, a call to *add* via p_A emits an instance of the call event, which might drive the state machine to change its active state. For example, the active state can be changed from *Idle* to *Working*.

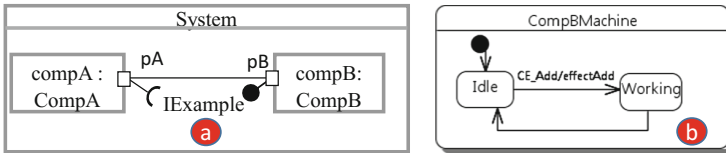


Fig. 1. Example illustrating combination of a service ports and a call event.

Flow Port and Signal Event: UML standard ports only provide method call-based interactions between components. Some UML extensions such as MARTE [13] define *flow ports*. The latter enable message-driven and data flow oriented communication between components, where messages exchanged between ports represent data items. The direction of the data flow of a flow port can be *in/out/inout*. The messages are modeled in terms of UML signals.

To describe our combination of flow ports and signal events, we reuse the example as in Fig. 1 with some modifications: p_A and p_B become *out* and *in* flow ports with *SigEx* as their signal, respectively; the event triggering the transition from *Idle* to *Working* is a signal event *SE_SigEx*. An instance of the *SigEx* signal can be sent from *compA* through its p_A port to p_B of *compB*. Upon the reception of the signal instance, an instance of the *SE_SigEx* event is then emitted and saved in an event queue managed by *compB* for asynchronous processing. The *CompBMachine* state machine then asynchronously processes the event and might change its active state from *Idle* to *Working*.

Our approach and tool support all of these events and the combination of state machines with composite structure elements to model event-driven reactive applications.

3 Features

Our approach and its associated tool have some features compared to other tools as follows:

Completeness: Our tool supports all state machine vertexes and transitions including all pseudo states and transition kinds such as external, local, and internal. Therefore, the tool improves flexibility of using UML state machines to express architecture behavior.

Event Support: Our tool favors the use of four UML event types and a event deference mechanism, which are able to express synchronous and asynchronous behaviors and exchange data between components/classes.

UML State Machine Conformance: We have experimented code generated by our tool with a test suite for state machines. The test suite is defined by a recent specification under standardization of the OMG that formalizes the Precise Semantics of UML state machine (PSSM). The latter defines a test suite consisting of 66 test cases for validating the conformance of runtime execution of code generated from UML state machines. Traced execution results of 62/66 test cases comply with the standard and are, therefore, a good hint that the execution is semantically correct. For the moment, our tool cannot deal with transitions from an *entry point* to an *exit point*. We believe that these transitions are not used in reality. This is because the contradictory semantics of *entry points* and *exit points* (see [12] for more information).

State Machine Configuration: Asynchronously processed events including signal events, change events, and time events are stored in an event queue. A signal event can transfer data (message). We allow for the configuration of the event queue size and the maximal size of signals. The configuration is not specified by the UML specification because this latter wants to be abstract. We allow to determine these values through a specific profile. This configuration might not be needed in dynamic memory allocation. The latter, however, is not recommended in embedded systems.

Efficiency: We conducted experiments with two benchmarks (see Sect. 7 for detailed information). The results show that code generated by our tool is efficient and can be used to develop resource-constrained embedded software. Specifically, event processing is fast and the size of executable files compiled from generated code is small.

Event API: Generated code in our tool provides APIs for environment code to invoke operations or send data signals to reactive classes via their ports. The invocations and sending will automatically fire events for state machines to process.

Concurrency: Concurrency aspects in state machines including doActivity of states, orthogonal regions, event detection, and event queue management are handled by the execution of multiple threads. Currently, we use POSIX threads for concurrency.

Portability: Currently, our tool generates C++ (prior to C++11) code. The generated code can run on POSIX systems such as Ubuntu without installing any additional libraries to be able to compile and execute the code. Our code

generation pattern and tool can be extended to generate code in more recent C++ versions (e.g. C++11 and C++14) with C++ native threads or in other programming languages such as Java, which supports thread-like and mutex-like mechanisms for multi-thread synchronization.

Combination of Composite Structure Elements and State Machine Elements: As previously discussed in Subsect. 2.2, this paper explores the use of UML composite structures and state machines for component-based design for reactive systems. This design semantics is automatically translated to code. We provide APIs for interactions between components, ports, and state machines. For the example of service ports shown in Fig. 1, we generate an attribute, namely *pA_req*, typed by the *IExample* interface, for the *pA* port. The code automatically binds this attribute to the implementation in *CompB*, in which every call of the *add* method will fire an event to the state machine.

In the next sections, we describe how our approach provides these features.

4 Concurrency

This section describes our design of concurrency aspects of state machines in generated code at runtime.

4.1 Thread-Based Design

The concurrency of state machines is designed based upon multiple threads including permanent and spontaneous threads. Permanent threads are created once and live as long as the state machine is running. On the other hand, spontaneous threads are spawned and active for a while. Each permanent thread is initialized upon the initialization of the state machine. The execution of permanent threads follows the paradigm “wait-execute-wait”. In the latter, a thread **waits** for a signal to **execute** its associated method and goes back to the **wait** point if it receives a stop signal or its associated method completes. Each of the following actions is associated with a permanent thread:

- *doActivity* of each state if has any.
- Sleep function associated with a time event which counts ticks and emits the event once completes.
- Change detection function associated with a change event which observes a variable or a boolean expression and pushes an event to the queue if a change occurs.
- State machine main thread, which reads events from the event queue, and sends start and stop signals to other permanent¹.

¹ Some approaches call the method of this thread as a super loop reading events from the queue.

Spontaneous threads are spawned by a parent thread. During execution, these threads are destroyed once the associated methods complete. The spontaneous threads follow a paradigm in which the spawning parent thread must wait until its children complete their associated methods. The following cases are associated with spontaneous threads:

- A thread is created for each effect of transitions outgoing from a *fork* or incoming to a *join*.
- Entering a concurrent state, after the entry action of the state, a thread is created for each orthogonal region.
- Exiting a concurrent state, before the exit action of the state, a thread is created for each region to exit the corresponding active sub-state.

4.2 Thread Communication

Each permanent thread is associated with a mutex for synchronization in the multi-thread-based generated code. The mutex must be locked before the method associated with the thread is executed.

Run-to-Completion: The processing of events must follow the run-to-completion semantics of UML state machines. This semantics requires that the state machine must finish the processing of each event before processing the next event. If all events are asynchronous, the main thread processes events by simply reading one-by-one from the event queue. However, because call events are synchronously processed, the processing of synchronous and asynchronous events can violate the run-to-completion semantics. To avoid it, a main mutex is associated with the main thread to protect the run-to-completion semantics. Each event processing must lock the main mutex before executing the actual processing. In generated code, lock and unlock are implemented using signals and conditions in POSIX [14].

5 Code Generation Pattern for UML State Machines

This section describes our code generation for states, regions, events, transitions, and pseudo states.

5.1 State

A common state type *IState* is created. This is particularly different from existing approaches, which represent states either as separate classes [15–17] or as enumerations [18]. The state type contains two attributes called *actives*, to preserve the hierarchy of composite states, and *previousActives* to refer to current and previous active sub-states in case of the presence of history pseudo states. Each UML state is translated into an instance of **IState** and a state identifier is assigned to each state. During initialization of the state machine, each instance initializes its attributes to a default value that represents an inactive state.

In the following sections, we consider C++ as a specific generated language. The discussion of other object-oriented languages is similar since these share the same concepts.

Listing 1.1 shows the state type and its instances with *STATE_MAX* as the number of states. *STATE_MAX* is calculated for each state machine². The state actions such as entry/exit/doActivity are translated to corresponding methods that contain action codes. For example, the state action *entry* in the listing implements the entry actions of all of the states.

State *doActivities* of active states, as specified by UML, are run concurrently. Each *doActivity* is then run within a permanent thread and a mutex is created for controlling the wait-execute-wait paradigm. Listing 1.2 shows a code segment for *doActivity* threads. The method *doActivityThread* takes as input a state identifier to use and call the appropriate *doActivity* of the active state. The method does nothing but stays in a waiting point if the state corresponding to the input parameter state identifier is inactive (line 5). If the state becomes active, a start signal is sent to this thread method to start the execution of *doActivity*. The generated code typically uses the common paradigm in POSIX threads [14].

Listing 1.1. IState interface in C++.

```

1 typedef struct IState {
2     int previousActives [2];
3     int actives [2];
4 } IState;
5 class C {
6 private:
7     IState states [STATE_MAX];
8 public:
9     void entry(StateId id) {
10        switch(id) {
11            case S0.ID:
12                //action code for each state
13                break;
14            //code for other state actions
15        }
16    }
17 }

```

Listing 1.2. Example code for doActivity.

```

1 while (true) {
2     pthread_mutex_lock(&mutex[stateId])
3     ;
4     while (!isStarts[stateId]) {
5         //await start signal
6         pthread_cond_wait(
7             &cond,&mutex[stateId]);
8     doActivity(stateId);
9     isStarts[stateId]=false; //reset
10    flag
11    pthread_mutex_unlock(
12        &mutex[stateId]);
13    if (!isStops[stateId]) {
14        if (stateId==S0.ID
15            |...) { //atomic
16            states
17            pushCompletionEvent(stateId);
18        }
19    }
20 }

```

5.2 Region

Our approach treats a variety of ways how transitions can enter and exit a region. For each region, a method for entering and exiting is generated. The entering method controls how a region *r* is entered from an outside transition, the exiting method executes exit actions of sub-states from innermost to outermost.

A region can be entered in two different ways: (1) **entering by default**: the transition ends at the border of a composite state; and (2) **cross transition**: the transition ends at a direct or indirect sub-vertex of a composite state. The two entering ways execute the entry action of the containing composite state after the transition effect. The subsequent executions are usually different for

² To avoid runtime memory allocation, *STATE_MAX* is for each state machine, rather than for all state machines, which will waste resource small state machines.

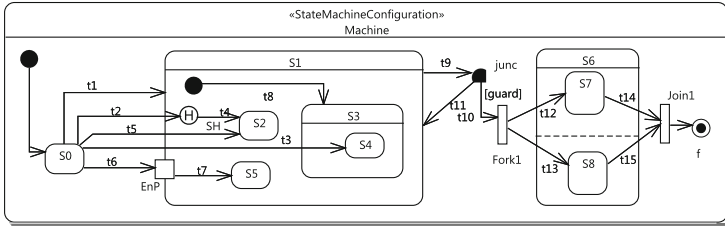


Fig. 2. Example illustrating different ways entering a composite state [12].

each way. In order to illustrate this, let us use the example in Fig. 2 with *S1* as target composite state. *t1* corresponds to way (1) while *t2*, *t5*, *t6* corresponds to way 2.

The entering method associated with the region of *S1* has a parameter *enter_mode* that indicates how the entering should be executed. The parameter *enter_mode* takes values depending on the number of transitions targeting the composite state. The details of how these modes are implemented in specific languages are not discussed here. Listing 1.3 shows the generated C++ code.

Listing 1.3. Example code generated for the region of *S1*.

```

void S1Region1Enter(int enter_mode){
2   if (enter_mode == DEFAULT) {
      states[S1.ID].actives[0] = S3.ID;
4     entry(S3.ID); sendStartSignal(S3.ID);
      S3Region1Enter(DEFAULT);
6   } else if (enter_mode == S2.MODE) {
      //..
8   } if (enter_mode == SH.MODE) {
      StateIDenum his;
10    if (states[S1.ID].previousActives[0]!=STATE.MAX){
        his=states[S1.ID].previousActives[0];
12    } else {
        his = S2.ID;
14    }
      states[S1.ID].actives[0] = his;
16    entry(his); sendStartSignal(his);
      if (S3.ID == his) {
18      S3Region1Enter(S3.REGION1.DEFAULT);
    }
20  } else if (enter_mode == S4.MODE) {
      states[S1.ID].actives[0] = S3.ID;
22    entry(S3.ID); sendStartSignal(S3.ID);
      S3Region1Enter(S4.MODE);
24  } else if (enter_mode == ENP.MODE) {...}
}
    
```

By default, the active sub-state of a region is assigned after the completion of the execution of any effect associated with the initial transition. Therefore, *S3* is set as active sub-state of *S1*. Entering at (*S2*) implies that the state *S2* becomes the active sub-state of *S1*. In case of an indirect sub-state (*S4*), the entry action of *S3* is executed before *S4* is set as the active-sub state of *S3* and the execution of the entry action of *S4*. It is worth noting that after the execution of each

entry action, a start signal is sent to activate the waiting thread associated with *doActivity* of the corresponding states.

The code generation for a transition from a vertex to a sub-vertex of the composite state is not as simple as that of two states (the transition from the *S0* state to the *SH* shallow history state is a particular case). This is detailed in the next subsections.

The method generated for exiting a region is simpler than that of entering because it basically executes the exit actions of all active sub-states from innermost, specified by the current active sub-state, to outermost.

5.3 Event

In our approach, one method is generated for each event similar to the approach in [15]. Each event is also associated with an identifier. The event list of a state machine contains explicitly defined events and a special event called *completion event*. A completion event is fired when either the execution of the *doActivity* of a simple/atomic state completes or all regions of a composite state have reached a final state. For each event type, the pattern is realized as follows:

CallEvent: When its associated operation is called, the event processing waits and locks the main mutex protecting the run-to-completion semantics as previously mentioned, and executes the event processing (see Subsect. 4.2).

SignalEvent: In the generated code, an API *push* is created. The invocation of this method (from the environment) implies that an instance of the signal associated with the event is created and written into the event queue.

TimeEvent: A thread associated with the event is created and initialized. The thread method starts sleeping for a specified duration, once it receives a signal which is sent after the execution of the entry of an accepting state. When the relative time expires, the event is emitted and written to the event queue if the state is still active.

ChangeEvent: Similarly to time events, a thread is initialized and its method waits for a re-evaluation signal. The method checks whether the value of the boolean expression of the event changes from false to true. If so, the event is sent to the event queue. The expression is composed of attributes of the class owning the state machine. The starting signal is sent if one of the expression's constituents (attributes of the class) changes. We track changes of the attributes' values by using setters of the attributes. For example, for an expression $x + y > 10$, x and y are constituents. The setters (*setX* and *setY*) are automatically generated. These methods do not only affect the value of x and y but also send the starting signal to the thread.

As previously presented, all asynchronous incoming events are stored in a runtime priority queue, in which each event type has a priority. The completion event has always the highest priority, the priorities of other events are equal by default. Event type, priority, identifier, the associated state *stateId* of completion events, and signal data are specified in an internal structure. The associated

state is responsible to specify which atomic/simple state completes its *doActivity* execution or the composite state whose sub-states have reached final states.

5.4 Transitions and Pseudo States

Each event triggers a list of transitions. We suppose $T_{trig}(e)$ is the transition list triggered by the event e , and $S_{trig}(e)$ is a depth-ordered (from innermost to outermost) set of the source states of the transitions in $T_{trig}(e)$.

Algorithm 1 describes how to generate the body of an event method. It first finds the innermost active states which are able to react e by orderly looping over $S_{trig}(e)$. This is to ensure that, in case of multiple transitions triggered by the event, the generated code for the transitions outgoing from innermost states will be executed. For each transition from an innermost state, code for active states and deferred events, guard checking, and transition code segments are generated by *GEN_CHECK*, *GEN_GUARD*(t) and *GEN_TRANS*, respectively. If the identifier of e is equal to one of the deferred event list of the corresponding state (not shown in this paper), *GEN_CHECK* generates code, which checks whether should be deferred and - if yes - pushes the event to a deferred event queue managed by the runtime main thread. The latter also pushes the deferred events back to the main queue once one of the pending events is processed and the active state is changed.

Algorithm 1. Code generation for events.

Require: Event e

Ensure: Code generation process for event method

```

1: procedure EVENTGENPROCESS( $e$ )
2:   for  $\forall s \in S_{trig}(e)$  do
3:      $T_s = \{t \in T_{trig}(e) | src(t) = s\}$ 
4:     for  $\forall t \in T_s$  do
5:       GEN_CHECK( $s, t, e$ )
6:       GEN_GUARD( $t$ )
7:       GEN_TRANS( $s, t, tgt(t)$ )

```

For a transition t , *GEN_CHECK* can generate single or multiple active state checking code. The latter occurs if the target of the transition is the pseudo state join because the transitions incoming to a *join* are fired if and only if all of their source states are active. The detailed discussion on these is not presented due to space limitation.

Listing 1.4. Example code generated for completion events triggering transitions $t14$ and $t15$.

```

1  if(event.stateId==S6_ID || event.stateId==S7_ID){
2      if(states[S6_ID].actives[0]==S7_ID &&
3          states[S6_ID].actives[1]==S8_ID) {
4          thread_r1=FORK(S6Region1Exit);
5          thread_r2=FORK(S6Region2Exit);
6          JOIN(thread_r1); JOIN(thread_r2);
7          sendStopSignal(S6_ID); exit_S6();
8          thread_t14=FORK(effect(t14));
9          thread_t15=FORK(effect(t15));
10         JOIN(thread_t14); JOIN(thread_t15);
11         effect_t16();
12         activeStateID=STATE.MAX; //inactive
13     }
14 }

```

Algorithm 2. Code generation for transition.

Require: A source v_s , a target vertex v_t and a transition t

Ensure: Code generation for transition

```

1: procedure GEN_TRANS( $v_s, v_t, t$ )
2:   Find  $s_{ex}$  and  $s_{en}$  as vertices in the same region and directly or indirectly containing/being  $v_s$  and  $v_t$ .
3:   Generate IF-ELSE statements for junctions
4:   if  $s_{ex}$  is a state then
5:     for  $r \in$  regions of  $s_{ex}$  do
6:       FORK(RegionExit( $r$ )) //exiting region threads
7:       Generate JOIN for threads created above
8:       Generate sendStopSignal to  $s_{ex}$ 
9:       exit( $s_{ex}$ ) //exit the state
10:  if  $v_t$  is a pseudo state join then
11:    for  $in \in$  incoming transitions of  $v_t$  do
12:      FORK(effect( $in$ )) //transition effect threads
13:      Generate JOIN for threads created above
14:    else
15:      effect( $t$ ) //execute transition effect
16:  if  $s_{en}$  is a state then
17:    entry( $s_{en}$ ) //state entry
18:    Generate sendStartSignal to  $s_{en}$ 
19:  if  $s_{en}$  is a composite state then
20:    for  $r \in$  regions of  $s_{en}$  do
21:      FORK(RegionEnter( $r$ ))//enter region threads
22:      Generate JOIN for threads created above
23:    else
24:      Generate for pseudo states by patterns

```

Listing 1.4, lines 2–3 show a portion of the code with multiple checking generated for the completion event processing method. The transitions $t14$ and $t15$ incoming to $Join1$ are executed if $S6$ and $S7$ are active. In addition, the code portion checks the state associated with the current completion event emitted upon the completion of either $S6$'s or $S7$'s *doActivity*. In lines 4–6, the code concurrently exits the sub-states of $S6$ by using *FORK* and *JOIN*, which are respectively used to spawn and wait for a thread, for the region methods associated with $S6$'s orthogonal regions, which actually exit $S7$ and $S8$. Then, *exit*($S6$)

is executed before the concurrency of transition effects $t14$ and $t15$ is taken into account.

Listing 1.5. Example code generated for *Fork1* and *junc*.

```

1  if(activeRootState==S1_ID) {
2    junc = 0; //transition t9 of junc
3    if (guard) {junc = 1;}
4    //Exit substates of S1 and S1
5    effect(t9);
6    if(junc==0) {
7      effect(t11);
8    } else {
9      effect(t10)
10   }
11   FORK(effect(t12)); FORK(effect(t3));
12   //JOIN...=>concurrent execution
13   //Enter state S6, S7 and S8
14 }

```

GEN_TRANS is able to generate code for transitions between two vertexes. Algorithm 2 shows how it works. The generated code is contained by the code for checking the deferral events, active states, and guards.

Firstly, Algorithm 2 looks for the s_{ex} and s_{en} vertexes, that are contained in the same region. And s_{ex} and s_{en} also contain the source and target vertexes of the transition t , respectively. For example, s_{ex} and s_{en} in case of the $t3$ transition are $S0$ and $S1$ contained by the top region. If the transition t is part of a compound transition (we use the algorithm presented in [19,20] to compute compound transitions), which involves some *junctions*, IF-ELSE statements for junctions are generated first (as PSSM says *junction* is evaluated before any action). The composite state is exited by calling the associated exiting region methods (*FORK* and *JOIN* for orthogonal regions) in lines 4–9 and followed by the generated code of transition effects (lines 10–15). If the parent state s_{en} of the target vertex v_t is a state (composite state), the associated entry is executed (lines 16–18). Entering region methods are then called once the above code completes its execution (lines 19–24). If the target v_t of the transition t is a pseudo state, the generation pattern corresponding to the pseudo-state types is called. These patterns are as follows:

- **Join:** Use *GEN_TRANS* for v 's outgoing transition (Listing 1.4, lines 4–6).
- **Fork:** Use *FORK* and *JOIN* for each of outgoing transitions of v (see Listing 1.5, lines 11–12).
- **Choice:** For each outgoing transition, an *IF – ELSE* is generated for its guard together with code generated by *GEN_TRANS*.
- **Junction:** As a static version *choice*, a *junction* is transformed into an attribute $junc_{attr}$ and evaluated before any action executed in compound transitions (see Listing 1.5, lines 2–3 and 6–10). The value of $junc_{attr}$ is then used to choose the appropriate transition at the place of *junction*.
- **Shallow History:** The identifiers of states to be exited are kept in *previousActives* of *IState*. Restoring the active states using the history is exemplified as in Listing 1.3. The entering method is executed as default mode at the first time the composite state is entered (lines 9–19). *previousActives*

is updated with the active state identifier before exiting the region containing the history.

- **Deep History:** The save and restoration of active states are done at all state hierarchy levels from the composite state containing the deep history down to atomic states. Updating *previousActives* is committed before exiting the region, which is directly or indirectly contained by a parent state, in which a *deep history* is present.
- **Entry Point:** If an *entry point* has no outgoing transition, the composite state is entered by default. Otherwise said, *GEN_TRANS* is called to generate code for each outgoing transition. If the entry point has multiple outgoing transitions, the code generation process is similar to that of a *fork*, except that the outgoing transitions must be activated after the entry action (if any) of the containing state completes.
- **Exit Point:** The code for each transition outgoing from an *exit point* is generated by using *GEN_TRANS*. If the *exit point* has multiple incoming transitions from orthogonal regions, it is generated similarly to a *join* to multiple-check the source states of these incomings, except transitions incoming from the exit point are activated before the exiting of the containing state of the exit point.
- **Terminate:** The code executes the exit action of the innermost active state, the effect of the transition and destroys the state machine object.

Note that, the procedure in Algorithm 2 only applies to external transitions. Due to space limitations, the detail of generating local and internal transitions is not discussed here but the only difference is that the composite state, that contains the local or internal transitions is not exited.

Non-deterministic Transitions: It is possible multiple transitions from the same source vertex (state or pseudo state) are triggered by the same event. In this case, only one of the enabled transitions should be activated. Consider the example in Fig. 2 and assume that the transitions $t1$, $t2$, and $t5$ with *guard1*, *guard2*, and *guard5* as their respective expression guard, can be triggered by the same event e_{S0} . If $S0$ is active and an instance of e_{S0} is received by the state machine, non-determinism occurs if at least two of the three guard expressions become true. We assume the values of *guard1* and *guard5* are true at runtime in this case. Only $t1$ or $t5$ should be activated and the next active configuration of the state machine is different from each activation. $S3$ is active if $t1$ is activated and $S2$ is active otherwise. For transition selection in this situation, we propose three options to deal with the non-deterministic transition selection, as follows:

- Priority by creation: Among the enabled transitions, the transition created (by modelers) first is chosen to be activated. If $t1$ is created before $t5$, it is selected for activation.
- Random selection: A transition is randomly selected to be activated.
- User configuration: A UML profile is created and allows users/modelers to explicitly specify which transition has higher priority. The transition has the highest priority in the enabled set is selected.

In the next section, we present how code can be generated from UML composite structures.

6 Code Generation for Composite Structure

This section presents our code generation for composite structure elements, especially for ports and connectors, and their combination with UML state machine elements as previously discussed.

Service Port and Call Event: For each service port p of the component class, structural members of the corresponding classes are generated as follows:

- If p has a required interface $IReq$, an attribute, namely *name of p + “_req”*, typed by $IReq$ and a *setter* method for setting the attribute are generated.
- If p has a provided interface $IProvide$, an attribute, namely *name of p + “_provide”*, typed by $IProvide$ and a *getter* method for returning the interface implementation are generated.
- If p has a required interface $IReq$ and a provided interface $IProvide$, two attributes and methods are generated for required and for provided.

The created attributes corresponding to the required ports allow user-code embedded as blocks of text within the model, to call methods/operations provided by its component from the required ports.

Listing 1.6. Example code for service ports and call events.

```

2  class CompA {
3  public:
4      IExample* pA_req;
5      void connect_PA(IExample* ref) {
6          pA_req = ref;
7      }
8  }
9  class CompB:public IExample{
10 public:
11     IExample* pB_provide;
12     IExample* get_PB() {
13         pB_provide=this;
14         return this;
15     }
16     int add(int a, int b) {
17         processCE_Add(a,b);
18         return a+ b;
19     }
20     void processCE_Add(int& a, int& b)
21     {
22         //code for event processing
23     }
24 }
25 class System {
26 public:
27     CompA compA;
28     CompB compB;
29     void createConnections() {
30         compA.connect_PA(compB.get_PB());
31     }
32 }

```

Listing 1.7. Example code for flow ports and signal events.

```

2  class CompA {
3  public:
4      IPush<SigEx>* pA_out;
5      void connect_PA(IPush<SigEx>* ref){
6          pA_in = ref;
7      }
8  }
9  class CompB:public IPush<SigExample>{
10 public:
11     IPush<SigEx>* pB_in;
12     IPush<SigEx>* get_PB() {
13         pB_in=this;
14         return this;
15     }
16     void push(SigEx& sig){
17         //create a signal event
18         //put the event to the queue
19     }
20     void processSE_SigEx(SigEx& sig){
21         //code for event processing
22     }
23 }
24 class System {
25 public:
26     CompA compA;
27     CompB compB;
28     void createConnections() {
29         compA.connect_PA(compB.get_PB());
30     }
31 }

```

Listing 1.6 shows the code generated from the example in Fig. 1. Attributes and methods in the code are created for the p_A and p_B ports in the model. The *CompB* class implements the *IExample* interface, which has the *add* method, provided by its p_B port. The implementation of *add* calls the *processCE_Add* method for processing the *CE_Add* event before it executes the user-code (line 17). By this way, any invocation of *add* of *CompB* follows the execution semantics as described in Sect. 2.

The connector between the p_A and p_B ports is transformed into a statement. The latter uses the *getters* and *setters* generated above to refer required attributes (pA_req) to appropriate provided attributes ($pB_provide$) (see line 28 in Listing 1.6).

Flow Port and Signal Event: The code generation for flow ports in combination with signal events is much similar to that of service ports and call events. Required and provided service ports are replaced by *out* and *in* flow ports, respectively. An attribute generated for a flow port is typed by an $IPush<S>$ interface, in which S is the signal type of the flow port. The $IPush<S>$ interface has a *push* method for sending signal instances in the implementation.

Listing 1.7 shows a code portion for using flow ports in the example in Sect. 2.2. Attributes are generated for the p_A and p_B flow ports at lines 3 and 10. The $IPush<SigEx>$ interface is implemented by *CompB*, which creates a signal event instance and puts it to the event queue of *CompB*. This event instance will be asynchronously processed (lines 19–21) by the state machine code of *CompB* generated using the pattern in Sect. 5.

In the next section, we present our empirical study to evaluate the proposed approach.

7 Empirical Study

The pattern is implemented in Papyrus Designer [21], which is an extension of the UML modeling tool Papyrus [22]. Papyrus Designer supports component-based modeling and code generation. The behavior of a component in Papyrus Designer is described by using UML state machines. The tool allows to use some time notions from the MARTE profile to specify time events. C++ code is generated and runs within POSIX systems such as Ubuntu, in which pthreads are used for implementing threads for concurrency. This section reports our experiments with Papyrus Designer on the semantic-conformance and efficiency of generated code.

7.1 Semantic Conformance of Runtime Execution

This section presents our results found during experiments with our tool to answer the following research question.

Research question 1: *Is the runtime execution of code generated from USMs by our tool semantic-conformant to PSSM?*

To evaluate the semantic conformance of runtime execution of generated code, we use a set of examples provided by Moka [23], which is a model execution engine offering PSSM (and also part of the Papyrus modeler). Figure 3 shows our method. The latter consists of the following steps:

Step 1. For a **State machine** from the Moka example set, we use our code generation tool to generate code.

- Step 2.** We simulate the execution of the **State machine** using Moka and then extract the sequence **Trace 1** of observed traces including executed actions.
- Step 3.** The sequence (**Traces 2**) is obtained through the runtime execution of the code generated in Step 1.
- Step 5.** *Trace 1* and *Trace 2* are compared. The code is semantic-conformant if **Traces 1** and **Traces 2** are the same [24].

The PSSM test suite consists of 66 test cases for different state machine element types. The results are promising: our tool passes 62/66 tests including: behavior (5/6), choice (3/3), deferred events (6/6), entering (5/5), exiting (4/5), entry(5/5), exit (3/3), event (9/9), final state (1/1), fork (2/2), join (2/2), transition (11/14), terminate (3/3), others (2/2). In fact, our tool fails with some tests containing transitions (1) from an *entry point* to an *exit point* or (2) from an entry point/exit point to itself. This is, as our observation, rarely used in practice because of the contradictory semantics of *entry points* and *exit points* as previously discussed.

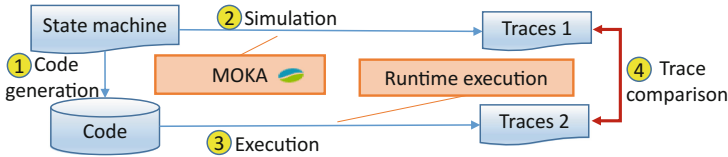


Fig. 3. Semantic conformance evaluation methodology [12].

The results of this evaluation are not enough to prove that our pattern and tooling support preserve the UML state machine execution properties but are a good hint that runtime execution of generated code is semantically correct (for all but the case identified above).

This evaluation methodology has the limitation that it is dependent on PSSM. Currently, for event support, PSSM only specifies signal events. History pseudo states are not supported. Thus, our evaluation result is limited to the current specification of PSSM.

Threats to Validity: Operation behaviors in PSSM are defined by activities while our prototype requires fine-grained behavior as blocks of code embedded into models. Therefore, we manually re-create these tests and convert activities into programming language code. A potential threat is that the conversion might change the semantics of the model.

7.2 Benchmarks

In this section, we present the results obtained through the experiments on efficiency aspects of generated code to answer the following question.

Research question 2: Runtime performance and memory usage are undoubtedly critical in real-time and embedded systems. Particularly, in event-driven systems, the performance is measured by event processing speed. Are the performance and memory usage of code generated by our tool comparable to existing approaches?

Two state machine examples are obtained by the preferred benchmark used by the Boost C++ libraries [25,26]. One simple example only consists of atomic states and the other both atomic and composite states. We compared our tool with tools such as Sinelabore (which generates efficient code for Magic Draw [27], Enterprise Architect [28]), Quantum Modeling (QM) [29] (which generates code for event-driven active object frameworks [30]), Boost Statechart [31], Meta State Machine (MSM) [32], C++ 14 MSM-Lite [26], and functional programming like-EUML [33].

We used a Ubuntu virtual machine 64 bit hosted by a Windows 7 machine. For each tool, we created two applications corresponding to the two examples, generated C++ code and compiled it in two modes: normal (N), by default GCC compiler; and optimal (O) with GCC optimization options -O2 -s. 11 millions of events are generated and processed by the simple example and more than 4 millions for the composite example. Processing time is measured for each case.

Performance. Figure 4 shows the event processing performance of the approaches for the two benchmarks. In the normal compilation mode (postfix N), Boost Statechart, MSM, MSMLite, EUML are quite slow and not displayed in the box-plot.

In both of the simple and composite benchmarks, in optimization mode (postfix O) MSMLite and our tool run faster than the others in the scope of the experiment. The figure also shows that the optimization of GCC is significant. In normal mode only the performance of Sinelabore, QM, and our tool is acceptable. The event processing speed of MSM, MSM_Lite and EUML is too slow without GCC optimizations.

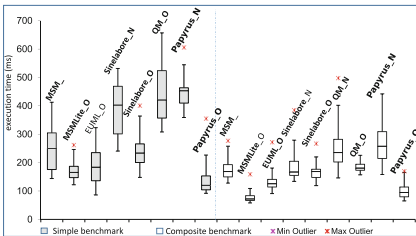


Fig. 4. Event processing speed for the benchmarks [12].

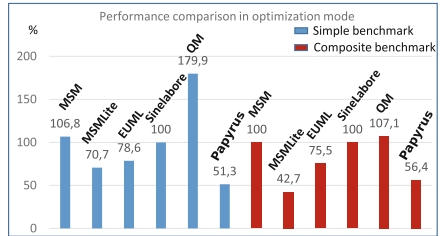


Fig. 5. Event processing performance in optimization mode [12]. (Color figure online)

Memory Usage. Table 1 shows the executable size for the examples compiled in two modes. Without optimization, Sinelabore generates the smallest executable

size while our approach takes the second place. In GCC optimization mode, MSMLite, Sinelabore and our approach require less static memory than the others.

Let’s look closer at the event processing performance in optimization mode in terms of time medians. Figure 5 shows the figures of the two benchmarks, relative to the performance of Sinelabore (normalized to 100%). For the simple (blue) benchmark, our approach (51.3%) is the fastest. For the composite (red) benchmark, with the support of C++14, the performance in MSMLite (42.7%) is the fastest and ours is the second.

For runtime memory consumption, we use the Valgrind Massif profiler [34,35] to measure memory usage. Table 2 shows the memory consumption measurements including stack and heap usage for the composite example. Compared to others, code generated by our approach requires a slight overhead with regard to runtime memory usage (0.35 KB). This is predictable since the major part of the overhead is used for C++ multi-threading using POSIX Threads and resource control using POSIX Mutex and Condition. However, the overhead is small and acceptable (0.35 KB).

Table 1. Executable size in KB.

Test	MSM		MSM-Lite		EUML		Sinelabore		QM		Our tool	
	N	O	N	O	N	O	N	O	N	O	N	O
Simple	414,6	22,9	107,3	10,6	2339	67,9	16,5	10,6	22,6	16,6	21,5	10,6
Composite	837,4	31,1	159,2	10,9	4304,8	92,5	16,6	10,6	23,4	21,5	21,6	10,6

Table 2. Runtime memory consumption in KB. Columns from left to right are SC, MSM, MSM-Lite, EUML, Sinelabore, QM, and Our tool, respectively.

SC	MSM	MSM-Lite	EUML	Sinelabore	QM	Papyrus
76.03	75.5	75.8	75.5	75.8	75.7	76.38

8 Traffic Light Controller Simulation

In order to assess the feasibility of using UML state machines and events for the design of an event-driven architecture, we applied our tool to a simplified Traffic Light Controller (TLC) system. This case-study originally appeared in [36] and its design is available in [37].

The TLC system controls an intersection of a busy highway and a little-used farm-way as shown in Fig. 6. Detectors are placed along the farm-way to raise the signal *C* as long as a vehicle is waiting to cross the highway. The highway lights remain green as long as no vehicle is detected on the farm-way. If a vehicle is detected, the highway lights should change from yellow to red, allowing the

farm-way lights to become green. In addition, the farm-way lights never stay green longer than a set interval to allow the traffic to flow along the highway. If there is no vehicle, or the timeout is expired, the farm-way lights change from green to yellow to red, allowing the highway lights to return to green. Even if vehicles are waiting to cross the highway, the highway should remain green for a set interval.

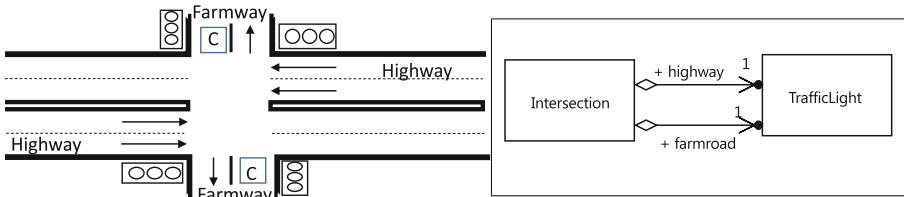


Fig. 6. Traffic Light Controller (left) and its class diagram (right) [12].

The object-oriented class diagram³ follows the design in Yasmine [37], a C++11 state machine framework. The diagram is shown in Fig. 6 (right). Each class has a behavior described by a state machine. The state machines of *Intersection* and *TrafficLight* are shown in Fig. 7 (left and right, respectively). Except for *FarmwayOpen*, all of the states of *IntersectionStateMachine* are composite. The details of *SwitchingHighwayToFarmroad* and *SwitchingFarmroadToHighway* are actually shown on the Yasmine website [37].

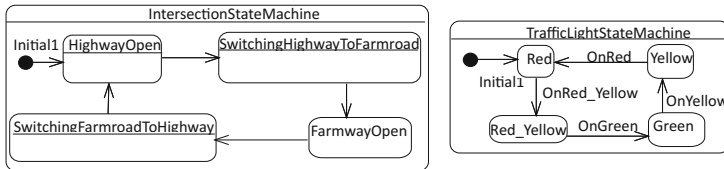


Fig. 7. State machines for describing the behavior of Intersection (left) and TrafficLight (right) [12].

The conditions for switching from the state *HighwayOpen* to *SwitchingHighwayToFarmroad* are: (1) a minimum time for the highway open is elapsed; and (2) the sensors emit a signal.

In terms of UML events, two alternative designs can be specified by using time events and change events. These alternatives are shown in Fig. 8(a) and (b) respectively. The first design in Fig. 8(a) uses a time event. The time event triggers the transition from *WaitingForHighwayMinimum* to *MinimumTimeElapsed*.

³ Component-based design can be used, but the purpose of this case study is to show the usability and practicality of UML state machines and events.

The first design also uses a signal event deferred by the *WaitingForHighwayMinimum* state. When *HighwayOpen* becomes active, its active sub-state remains *WaitingForHighwayMinimum* for as long as the minimum time (e.g. the latter is 5 s in this case.) If a signal C is fired from the detector, a signal event *DetectorOn* is sent to the state machine. The event is, however, not immediately processed but delayed until the active sub-state becomes *MinimumTimeElapsed* in case the time event is fired. The signal event is then processed to finish the execution of *HighwayOpen* and activate the farm-way.

To switch from *WaitForPreconditions* to a final state, the second design uses a change event instead of deferred events. Two flags *timeFlag* and *detectFlag* are used. The *WaitForPreconditions* state has two internal transitions. One is triggered by a signal event associated with the signal C and calls a transition effect to update *detectFlag* to true. The other one is triggered by a time event that sets *timeFlag* to true. Once two flags *timeFlag* and *detectFlag* are set to true, the expression associated with the change event updates from false to true. The periodic evaluation time is configured as 10 ms.

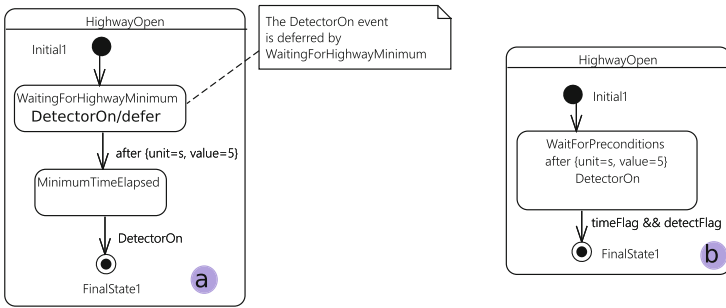


Fig. 8. Alternative state machine designs for the *HighwayOpen* state [12].

For the simulation of TLC, we reuse the detector class developed in [37] to automatically generate *DetectorOn/DetectorOff* signals.

Using UML events (change events and time events) and deferred events does not only provide designers with more options to specify their system, but it also simplifies the system behaviors. Indeed, the alternative designs reduce the number of states. For example, the numbers of sub-states of *HighwayOpen* with the use of deferred events and change events are two and one, respectively, while Yasmine requires three states. However, because of its specialized semantics, deferred events may make the design more difficult to understand.

9 Lego Car Factory

This section presents the application of our approach for modeling and generating code for a real case study. The objective of this application is to evaluate

the feasibility and scalability of the proposed approach. The case study is an embedded software for LEGO. The LEGO car factory consists of small LEGO cars used for simulating an industrial process [38]. It is chosen for the evaluation because it is a real world embedded system with enough complexity and it is developed within our lab for demonstrating the Papyrus capabilities.

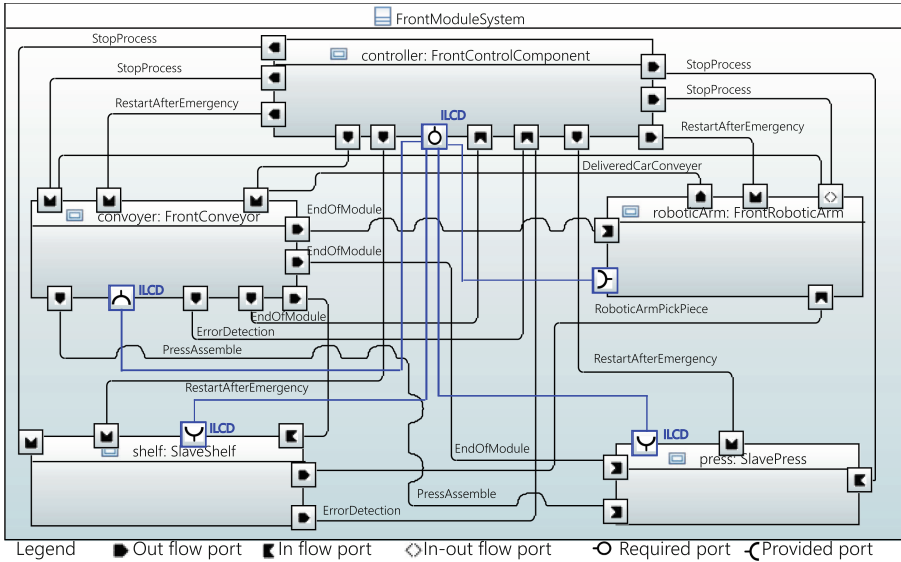


Fig. 9. Composite structure diagram of the front module for flow ports.

A LEGO car is composed of four modules: chassis, front, back, and roof. The communication between these modules is based on Bluetooth and master-slave like. In the latter, the chassis acts as master while the other modules act as slaves. Each slave module consists of five components: bluetooth communication controller, conveyor, robotic arm, press, and shelf. The behavior of each component is described by a UML state machine.

We use flow ports to exchange data items/signals between the components within a module. API invocations from a component to other components are designed by using service ports. Figure 9 shows the composite structure diagram for the *front* module without showing detailed structures of each of its components. For simplification, only flow ports are shown in the figure. The names of the signals/messages exchanged between flow ports of the components are annotated to the connectors between the flow ports as shown in Fig. 9. The three flow port types are used. For example, the controller can send the *StopProcess* signal to the other four components through its ports. The robotic arm component can send/receive *StopProcess* instances to the conveyor/from the controller

through its bidirectional flow port respectively. Note that the processing of signals incoming to a component via its ports is realized by the component's state machine.

Table 3. Lego car factory metrics.

Module	Lines of code	Binary size (KB)
Chassis	11193	264
Front	12246	268
Roof	12232	268
Back	12245	256

The code for the four models were generated and successfully compiled with an ARM C++ cross compiler. Table 3 shows the lines of code and binary size of the modules. Around 50000 lines of code were totally generated. This assesses the feasibility and scalability of the proposed approach.

10 Related Work

Code generation from state machines and composite structures has received a lot of attention in automated software development. This section mentions some existing code generation patterns and how our approach differs from them. A systematic review of code generation proposals for state machines is presented in [39].

Switch/if is the most intuitive technique for implementing a “flat” state machine. It either uses a scalar variable [18] and a method for each event, or two variables as the active state and the incoming event used as the discriminators of an outer switch statement to select between states and an inner one/if statement, respectively. The state table approach [40] uses one dimension for representing states and the other one for all possible events. These approaches require a transformation from hierarchical to flat state machines. However, these approaches are hardly applied to state machines containing pseudo states such as deep history or join/fork while taking concurrent states into account.

The object-oriented state pattern [16,40] transforms a state into a class and an event into a method. Events are processed by delegating from the class containing the state machine to its sub-state classes. Separation of states in classes makes the code more readable and maintainable. Unfortunately, this technique only supports flat state machines. This pattern is extended in [15] to support hierarchical state machines. Recently, a double-dispatch (DD) pattern presented in [17] extends [15] to support maintainability by representing states and events as classes, and transitions as methods. However, as the results show in [17], these patterns require much memory because of an explosion of the number of classes and the use of dynamic memory allocation, which is not preferred in

embedded systems. It is worth noting that none of these approaches provides implementation for all of state machine pseudo states as well as events.

Tools such as [3, 41] apply different patterns to generate code. However, as mentioned in Sect. 1, true concurrency, some pseudo-states, and UML events are not supported. FXU [9] is the most complete tool but generated code is heavily dependent on their own library and C# is generated.

Umple [7] is a textual UML programming language, which supports code generation for different languages such as C++ and Java from state machines. However, Umple does not support pseudo states such as fork, join, junction, and deep history, and local transitions. Furthermore, only call events and time events are specified in Umple. The ThingML [42] modeling language for embedded and distributed systems relies on non-UML state machines and connectors and supports code generation from these elements to various programming languages.

PapyrusRT [2] is the most similar to our approach, which uses UML composites structures and state machines for modeling, designing, and code generation for embedded systems. The difference between the two approaches is that our approach conforms to the UML specification and generated code does not rely on an additional runtime library while PapyrusRT conforms to the UML Real-Time Profile (UML-RT) [43]. This latter addresses modeling concepts suitable for modeling complex real-time systems [44]. Code generated by PapyrusRT relies on a runtime library - the PapyrusRT runtime. Furthermore, PapyrusRT does not support concurrent states, shallow history pseudo states, and transitions from a vertex at the outside of a composite state to one of its sub-vertexes, e.g. the $t5$ transition from $S0$ to $S2$ in Fig. 2.

Our approach for state machine code generation combines the classical switch/if pattern, to produce small footprint, and the pattern in [15], to preserve state hierarchy. Furthermore, we define a pattern to transform all USM concepts including states, pseudo states, transitions, and events. Therefore, users are flexible to create their USMs conforming to UML without restrictions.

11 Conclusion

We presented an approach for the design and code generation for component-based reactive systems, by providing a complete, efficient, and UML-compliant code generation approach from UML state machines along with a code generation pattern for UML composite structures. We extended the IF-ELSE/SWITCH patterns and used a hierarchical structure for implementing state machines. Furthermore, code generation for composite structures provides a means for developing reactive systems that combine component-based design and state machines.

The approach is integrated into the Papyrus modeling tool. Multiple experiments were conducted to evaluate our approach with respect to different aspects, including the semantic-conformance and efficiency of generated code. Sixty-two of sixty-six PSSM test cases are passed the conformance test. The results are a good hint that our tool preserves the UML state machine semantics during code generation. For efficiency, the benchmarks defined by the Boost library are

used to compare code generated by our tool to other approaches. The results show that code generated by our approach features fast event processing speed and small executable size. A Traffic Light Controller case study was used in the experiments to show the usability and practicality of UML state machines. To demonstrate the scalability and feasibility of the whole approach for component-based design of reactive systems, we used our approach to develop the Lego Car Factory software case study. This latter was successfully developed.

One of the limitations of the proposed approach is that, code produced for state machines by our tool consumes slightly more memory than that of the others at runtime. In future work, we will fix this issue by making the multi-thread part of generated code more concise. Furthermore, we will adapt the patterns to supporting code generation from UML state machines to Java and C++11 with thread support.

References

1. OMG Available Specification without Change Bars. OMG Unified Modeling Language (OMG UML), pp. 1–212 (2007)
2. Posse, E.: PapyrusRT: modelling and code generation (invited presentation). In: Proceedings of the International Workshop on Open Source Software for Model Driven Engineering Co-located with ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS 2015), Ottawa, Canada, 29 September 2015, pp. 54–63 (2015)
3. IBM: IBM Rhapsody (2016). Accessed 4 July 2016
4. Mussbacher, G., et al.: The relevance of model-driven engineering thirty years from now. In: Dingel, J., Schulte, W., Ramos, I., Abrahão, S., Insfran, E. (eds.) MODELS 2014. LNCS, vol. 8767, pp. 183–200. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11653-2_12
5. Whittle, J., Hutchinson, J., Rouncefield, M.: Model-driven engineering practices in industry: social, organizational and managerial factors that lead to success or failure. *Sci. Comput. Program.* **89**, 144–161 (2014)
6. Forward, A., Lethbridge, T.C., Badreddin, O.: Perceptions of software modeling: a survey of software practitioners. In: 5th Workshop from Code Centric to Model Centric: Evaluating the Effectiveness of MDD (C2M: EEMDD). Citeseer (2010). <http://www.esi.es/modelplex/c2m/papers.php>
7. Badreddin, O., Lethbridge, T.C., Forward, A., Elaasar, M., Aljamaan, H., Garzon, M.A.: Enhanced code generation from UML composite state machines. In: 2014 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD), pp. 235–245. IEEE (2014)
8. IBM: IBM Rhapsody and UML Differences (2016). <http://www-01.ibm.com/support/docview.wss?uid=swg27040251>. Accessed 4 July 2016
9. Pilitowski, R., Derezińska, A.: Code Generation and Execution Framework for UML 2.0 Classes and State Machines. In: Sobh, T. (ed.) Innovations and Advanced Techniques in Computer and Information Sciences and Engineering, pp. 421–427. Springer, Dordrecht (2007). https://doi.org/10.1007/978-1-4020-6268-1_75
10. Charfi, A., Mraidha, C., Boulet, P.: An optimized compilation of UML state machines. In: 2012 IEEE 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, pp. 172–179 (2012)

11. OMG: Precise Semantics of UML State Machines (PSSM) Revised Submission (2016). Revised Submission, ad/16-11-01
12. Pham, V.C., Radermacher, A., Gérard, S., Li, S.: Complete code generation from UML state machine. In: Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2017, Porto, Portugal, 19–21 February 2017, pp. 208–219 (2017)
13. OMG: A UML Profile for MARTE: Modeling and Analysis of Real- Time Embedded Systems, version 1.1 formal/2011-06-02 (2011)
14. Butenhof, D.R.: Programming with POSIX Threads. Addison-Wesley Professional, Boston (1997)
15. Niaz, I.A., Tanaka, J., et al.: Mapping UML statecharts to Java code. In: IASTED Conference on Software Engineering, pp. 111–116 (2004)
16. Shalyto, A., Shamgunov, N.: State machine design pattern. In: Proceedings of the 4th International Conference on .NET Technologies (2006)
17. Spinke, V.: An object-oriented implementation of concurrent and hierarchical state machines. *Inf. Softw. Technol.* **55**, 1726–1740 (2013)
18. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide, vol. 3 (1998)
19. Balsler, M., Bäumlner, S., Knapp, A., Reif, W., Thums, A.: Interactive verification of UML state machines. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 434–448. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30482-1_36
20. Knapp, A.: Semantics of UML State Machines (2004)
21. LISE: Papyrus Software Designer. https://wiki.eclipse.org/Papyrus_Software_Designer
22. Gérard, S., Dumoulin, C., Tessier, P., Selic, B.: 19 papyrus: a UML2 tool for domain-specific language modeling. In: Giese, H., Karsai, G., Lee, E., Rumpe, B., Schätz, B. (eds.) MBEERTS 2007. LNCS, vol. 6100, pp. 361–368. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16277-0_19
23. Papyrus: Moka Model Execution (2016). <https://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution>. Accessed 1 Nov 2016
24. Blech, J.O., Glesner, S.: Formal verification of Java code generation from UML models. In: Proceedings of the 3rd International Fujaba Days, pp. 49–56 (2005)
25. Boost Library: Boost C++ (2016). <http://www.boost.org/>. Accessed 4 July 2016
26. Jusiak, K.: State Machine Benchmark (2016). <https://github.com/boost-experimental>. Accessed 20 Oct 2016
27. Magic, N.: Magic Draw (2016). <https://www.nomagic.com/products/magicdraw.html>. Accessed 14 Mar 2016
28. SparxSystems: Enterprise Architect (2016). <http://www.sparxsystems.com/products/ea/>. Accessed 14 Mar 2016
29. Quantum Leaps: Quantum Modeling (2016). <http://www.state-machine.com/qm/>. Accessed 14 May 2016
30. Lavender, R.G., Schmidt, D.C.: Active object. *Context*, pp. 1–12 (1996)
31. Boost Library: The Boost Statechart Library (2016). Accessed 4 July 2016
32. Boost Library: Meta State Machine (2016). http://www.boost.org/doc/libs/1_59_0_b1/libs/msm/doc/HTML/index.html. Accessed 4 July 2016
33. Boost Library: State Machine Benchmark (2016). http://www.boost.org/doc/libs/1_61_0/libs/msm/doc/HTML/ch03s04.html
34. Valgrind: Valgrind Massif (2016). <http://valgrind.org/docs/manual/ms-manual.html>. Accessed 20 Nov 2016

35. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: ACM SIGPLAN Notices, vol. 42, pp. 89–100. ACM (2007)
36. Katz, R.H., Borriello, G.: Contemporary Logic Design (2005)
37. Yasmine: The classic farmroad example (2016). <http://yasmine.seadex.de/yasmine.html>. Accessed 20 Nov 2016
38. LEGO Car Factory. <http://robotics.benedettelli.com/lego-car-factory/>. Accessed 22 Mar 2017
39. Domínguez, E., Pérez, B., Rubio, A.L., Zapata, M.A.: A systematic review of code generation proposals from state machine specifications. *Inf. Softw. Technol.* **54**(10), 1045–1066 (2012)
40. Douglass, B.P.: Real-Time UML: Developing Efficient Objects for Embedded Systems (1999)
41. SparxSystems: Enterprise Architect (2016). <http://www.sparxsystems.eu/start/home/>. Accessed 20 Nov 2016
42. Harrand, N., Fleurey, F., Morin, B., Husa, K.E.: ThingML: a language and code generation framework for heterogeneous targets. In: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, pp. 125–135. ACM (2016)
43. Selic, B., Gullekson, G., Ward, P.T.: Real-Time Object-Oriented Modeling, vol. 2 (1994)
44. Cheng, S.W., Garlan, D.: Mapping Architectural Concepts to UML-RT (2001)



Automatic UI Generation for Aggregated Linked Data Applications by Using Sharable Application Ontologies

Michael Hitz¹(✉), Thomas Kessel¹, and Dennis Pfisterer²

¹ Cooperative State University Baden-Wuerttemberg, Stuttgart, Germany
michael.hitz@hitznet.de,
thomas.kessel@dhbw-stuttgart.de

² Institute of Telematics, University of Lübeck, Lübeck, Germany
pfisterer@itm.uni-luebeck.de

Abstract. The ongoing digitalisation efforts of businesses are a driving force to expose processes as services to third parties to enable the integration into third-party applications (e.g., booking of a trip or requesting the quote for a complex product). To standardize processes and related data, increasingly semantic web technologies are used. This leads to a shared conceptualization of the business domains and results in a *linked data service ecosystem* for domain-specific services, allowing third parties to aggregate services to novel applications - even across different domains. Using semantic web technologies enables the standardized communication on machine level. But the integration of the user into the overall process is still a manual task. The aggregation of services to complex applications is mostly done at the service level. The User Interfaces (UI) for collecting input data for the processes are usually still hand-crafted for different user groups and environments.

Our claim is, that given a linked data service ecosystem, the UI for a business process can be modelled once and be automatically generated for the integration into different contexts. The models can be combined to automatically build complex UIs for combined linked data applications – thus, supporting the aggregation of applications on the user interface level. This paper presents an ontology-based, model-driven approach for modelling UIs for the automatic generation of dialog-based applications, providing output understood by associated linked data services. In addition, the paper shows that the approach is suited to combine UI models as components to build aggregated linked data service UIs.

Keywords: User interface ontologies · Model driven user interfaces
Linked data application modelling · Application aggregation

1 Introduction

The digitalisation of business processes and the increasing need to expose business functionality to be used in different contexts led to a strong adoption of service oriented concepts for enterprise information systems. Companies offer their services (e.g., as web services) that are driven by user input data and invoke processes like ordering

goods or services. This opens new business opportunities for third parties that aggregate services to novel applications. Examples are *Uber* (*developer.uber.com*) or *Amazon Marketplaces* (*developer.amazonservices.com*) who expose their offerings through proprietary APIs.

Emerging business models such as *Distributed Market Spaces* in an *Internet of Everything* (IoE) context (e.g., [18]) go beyond that proprietary solution. They use a generic, non-proprietary data format by incorporating semantic web/linked data approaches (i.e., ontologies) to describe the semantics of the expected input data. This creates a shared conceptualization of the domain and thus allows multiple suppliers to participate in a transaction (e.g., providing information for the comparison of offerings). The participants can rely on the same input data, based on strictly defined semantics leading to a unified view on the processes; and thus, create a *linked data services ecosystem*. The industry begins adopting these principles by defining reliable data interchange semantics for different domains (e.g., the BiPRO initiative, www.bipro.net that standardizes business processes and data for the insurance domain).

Although there exists **a clear concept on the technical level for machines** to work on and communicate with semantically specified data (i.e., linked data technologies such as RDF/OWL) there is still a **lack of approaches for the integration of the human user**. Non-trivial user interfaces (UIs) are needed to collect user input for the business processes while supporting a platform-specific user experience (e.g., web frontends, mobile apps or rich client desktop apps). To provide novel applications, different variants of user interfaces are required, that aggregate multiple services from different domains within a single UI (e.g., a travel booking application, combining services for flight-, transportation-, hotel-, and event bookings). Currently, these UI variants are mostly developed manually for each application context.

Given the above-mentioned *environment of a linked data ecosystem as a prerequisite*, the assumption of this paper is that UIs for dialog based linked data applications can be **(1) automatically generated serving linked data services** and can be **(2) reused and shared in different contexts** (e.g., portals or rich client applications). Moreover, they can **(3) be aggregated to build combined, functionally augmented applications**.

The prerequisite for this approach is that linked data services provide a definition of the semantics of the expected input data (e.g., by using ontologies). Hence, UIs providing the expected input data can be used as a frontend for these services. In addition, if UIs are modelled in a technology-agnostic way, they can be shared, reused and even aggregated.

To automatically generate UIs for that purpose, the following major requirements have to be met: **(1) an abstract UI description** is needed to generate UIs for different technical platforms. The description needs to be **(2) modelled in a non-proprietary, standardized way** to make it sharable and reusable. Finally, **(3) the UI needs to produce output data that complies to the expected input** of targeted business processes to be used in conjunction with linked data services.

Although there already exist approaches for model-driven UI generation (cf. related work, Sect. 9), there is - to the best of our knowledge - no widely accepted approach or UI modelling technique that meets all of the aforementioned requirements (cf., [16]). Traditional approaches (e.g., user interface description languages - UIDL) rely on

proprietary UI technology focused models. They are strong in producing technological variants of UIs. The downside in their applicability in a linked data context is the proprietary, UI-focused nature of the modelled artefacts, which impedes their use in different contexts [4]. In addition, the mapping of input to target data - if possible at all - encompasses the creation of many related artefacts and thus is very complex (e.g. *UWE* [13]). Current research on ontology-based generation of UIs mainly focuses on providing general editors for arbitrary ontologies. The resulting generic interfaces are technical in nature and not suitable for presentation to a customer - as they do not focus on user experience.

The paper presents a novel approach for the automatic UI generation of linked data applications that bridges the gap between the traditional and ontological approaches. It contributes to the field of automatic UI generation applied to linked data concepts.

The content is structured as followed: First, the problem is demonstrated in more detail along with an illustrative example used throughout the paper. Section 3 outlines the proposed solution. Sections 4 and 5 provide details for the proposed Application Ontology. Section 6 outlines the process for derivation of UIs and resulting instance data. This is followed by an application of the results to build aggregated applications in Sect. 7 and the current state of evaluation of the concept. The paper closes with related work and a conclusion pointing out future work.

Remark: The paper is based on a contribution for the MODELSWARD 2017 Conference [10]. It extends the original publication by focussing on the aspect of the aggregation of applications.

2 Problem, Motivational Example and Requirements

To generate high-quality UIs for dialog applications, a data-model alone (e.g., a model of the input data for the underlying business process) is not sufficient. For example, UIs usually group information in a meaningful way, which is different from the target data structure of a consuming service. Most UIs include dynamic behaviour to guide the user through the data gathering process in an intuitive way (e.g., prefilling related information as a city name given a zip code or showing/hiding information based on provided data). The information needed to build these aspects are usually not part of a pure data-model [8] and thus need to be modelled separately.

Example: Consider a (simplified) process collecting quotes for a *flight booking*. A customer wants to request a quote from a target business service and thus needs to specify data about the intended flight. He supplies information about dates, number of tickets, return/open-yaw-flight and customer-related information (e.g., name, billing address etc.)

An excerpt of the possible request data (*target data*) based on a user's input is shown in Listing 1. It is intentionally simplified and represented in RDF/Turtle notation as instance of an (assumed) flightbooking ontology. It contains information about the flight (*3 tickets from Hamburg to Stuttgart*), return flight (*to Hamburg*) as an open-yaw-flight (*from Munich*) along with data about the customer.

Figure 1 shows possible UI variants for the user input dialogs: (a) a desktop application for an agent and (b) a mobile application for end customers. The information is

structured in **meaningful succession of groups and questions** (e.g., *Basic travel data*, *Flight Information* and *Your information*) and might have **hierarchical relations** (e.g., *Address* data being part of *Your information*). The questions are presented in a reasonable order, using **type-related input** controls allowing an intuitive user interaction.

Listing 1. Instance data for a flight request.

```
@prefix : <http://mimesis.solutions/bookers/flightbooking/individuals#> .
@prefix owl: ... ,rdf: ..., xml:..., xsd, ..., rdfs:...
@prefix fbo: <http://mimesis.solutions/bookers/flightbooking/v1#> .
@prefix foaf: <http://xmlns.com/foaf/0.1#> .
@base <http://mimesis.solutions/bookers/flightbooking/individuals> .
<http://mimesis.solutions/bookers/flightbooking/individuals
  rdf:type owl:Ontology .

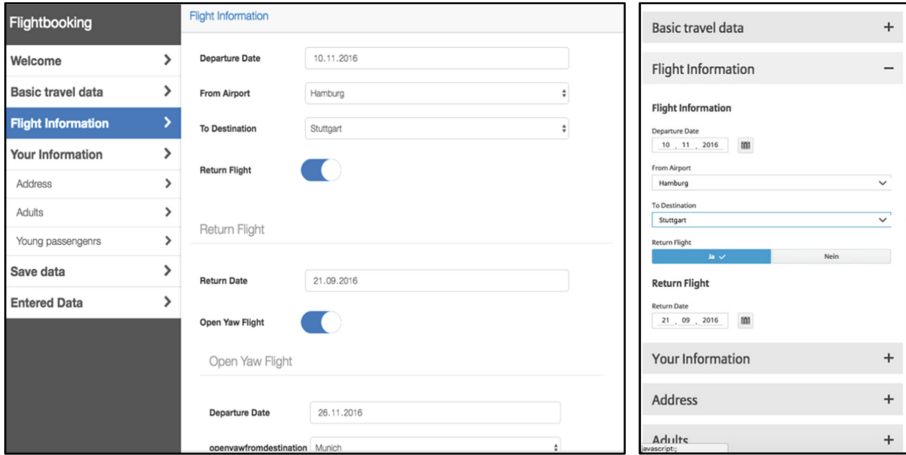
:flightbookingrequest_i1474371413428
  rdf:type <fbo:FlightBookingRequest> , owl:NamedIndividual ;
  <fbo:childtickets> "1"^^<xmles:integer> ;
  <fbo:adul tickets> "2"^^<xmles:integer> ;
  <fbo:customerinfo> :customerinfo_i1474371413428 ;
  <fbo:flight> :flight_i1474371413428 .

:flight_i1474371413428 rdf:type <Flight> ,owl:NamedIndividual ;
  <fbo:returndate> "2016-09-20T22:00:00.000Z"^^<xmles:date> ;
  <fbo:startdate> "2016-11-09T23:00:00.000Z"^^<xmles:date> ;
  <fbo:fromdestination> "HAM"^^<xmles:string> ;
  <fbo:todestination> "STR"^^<xmles:string> ;
  <fbo:openyawstartdate> "2016-11-25T23:00:00.000Z"^^<xmles:date> ;
  <fbo:openyawtodestination> "HAM"^^<xmles:string> ;
  <fbo:openyawfromdestination> "MUC"^^<xmles:string> .

:customerinfo_i1474371413428 rdf:type <Customerinfo> , ... ;
  <foaf:givenName> "Max"^^<xmles:string> ;
  <foaf:familyName> "Mustermann"^^<xmles:string> ;
  <foaf:gender> "male"^^<xmles:string> ;
  <foaf:email> "max.mustermann@onemail.com"^^<xmles:string> ;
  <fbo:billingaddress> :billingaddress_i1474371413428 .

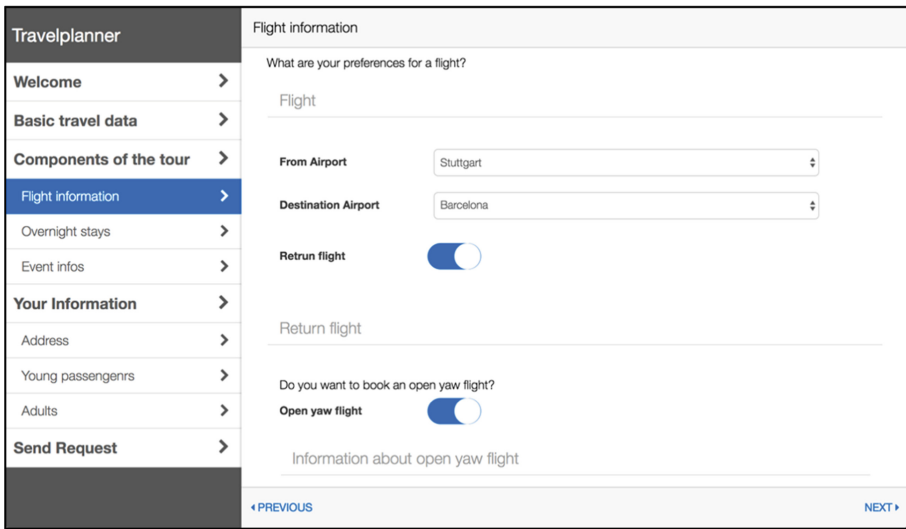
:billingaddress_i1474371413428 rdf:type <fbo:BillingAddress> , ... ;
  <foaf:buildingNo> "178"^^<xmles:string> ;
  <foaf:zip> "70178"^^<xmles:string> ;
  <foaf:street> "Reinsburgstraße"^^<xmles:string> ;
  <foaf:city> "Stuttgart"^^<xmles:string> ;
  <foaf:country> "germany"^^<xmles:string> .
```

In addition to the *structural aspects*, the UI needs to offer *behaviour* for a satisfying user experience: input needs to be **validated** and errors shown (e.g., if the *return date* is before the *departure date*), data might be prefilled as **reaction to previous input** (e.g., restricting *destination airports* that are in served by an already selected *departure airport*) and information needs to be **shown/hidden based on previous selections** (e.g., hiding *return flight related information* if the user deselects this option).



a) Desktop version (agent)

b) Mobile version (customer)



c) Aggregated UI for booking a trip (agent)

Fig. 1. Possible UIs for the flight booking application sample.

Figure 1, (b) shows a *variant* of the UI for mobile devices. The structure remains the same but is rendered for a different target device. In addition it does not offer the possibility to book an open-yaw-flight to reduce the complexity of the application.

Figure 1, (c) shows an **aggregated application** that is intended for the **booking of a trip**. It incorporates a version of the aforementioned *flight booking* as a component, but augments the overall functionality by adding components for *booking an overnight stay* and for *events* during the trip. ■

The examples show the *non-trivial* nature of UIs, which is not inferable from simple data models: additional structural and behavioural information is needed (e.g., sequence and rules for showing additional questions when input changes). Furthermore, the structure presented to the user for input differs from the structure of the actual request required by the backend service.

The goal of the presented approach is to provide a sharable way for describing UIs in a technology-agnostic manner, suited for the automatic derivation of UIs for different contexts. Thus, the following requirements need to be considered:

- **Req. 1:** A UI description is required, addressing the complexity of non-trivial UIs. It needs to contain all information about the data to be gathered and for the automatic generation of UI variants for different technologies and platforms.
- **Req. 2:** Information has to be provided, allowing the mapping of entered data to instances of the target ontology required by consuming services.
- **Req. 3:** To achieve sharable UI descriptions, a non-proprietary description is required that contains a minimum set of artefacts to be shared.
- **Req. 4:** A process for (a) building final UIs and (b) for inferring instance data from user input that can be processed by linked data driven backend services.
- **Req. 5:** The targeted UI description needs to be suitable for aggregation into complex UIs. It needs to be self-contained (i.e., complete regarding generation of a runnable UI) to be easy to share.

3 Proposed Solution

To meet the requirements, we propose a *single, declarative, data-centric application description*. It incorporates the required information (1) to derive non-trivial UIs and (2) for the mapping of input data to target ontology instances (cf. **Req. 1**, **Req. 2**). To be applicable to multiple contexts, a *UI technology-agnostic model* is used, which is based on the data to be processed by the application. For the content of the model the approach relies on our previous work on data-centric UI description models proposed in [8]. This approach is applied to ontological concepts (Sect. 4.2) and extended to contain additional data, required for the mapping of input data onto target instances (Sect. 5).

To meet requirement **Req. 3**, we rely on RDF/OWL [11] ontologies. RDF/OWL is used, as it is a well understood, widely adapted technology, already applied to different contexts and for which tooling is available (e.g. reasoners, APIs). The result is a sharable **Application Ontology** (AO) containing the required information needed as input for the automatic generation (cf. **Req. 4**).

Figure 2 shows the use of the proposed Application Ontology targeting **Req. 5** – i.e. to build aggregated applications within a linked data environment (Sect. 7). The central elements are the *Target Ontologies* (TO) and the corresponding *Application Ontologies* (AO) as shared elements between multiple client applications and domain specific backend services. The TOs define the semantics of possible input data for a business process. The AOs define variants of the user data to be gathered as outlined above.

An *Application Frontend* references multiple shared AOs and generates UIs for each of them according to the required technology base. The UI components can be

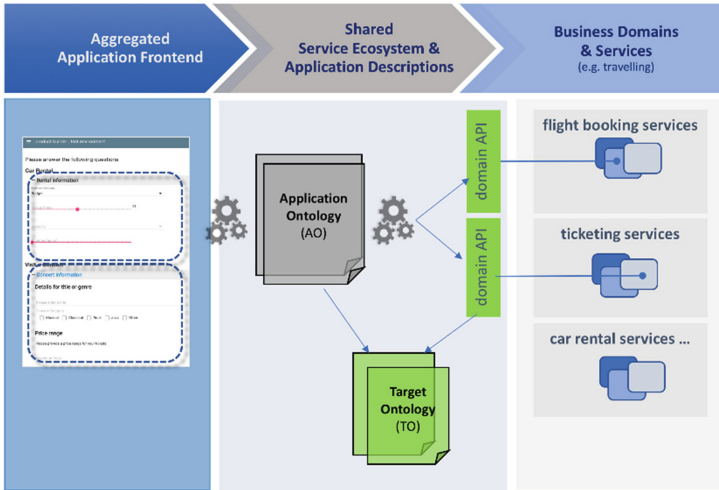


Fig. 2. UI aggregation of linked data applications.

aggregated within the UI to present a seamless combination presented to the user for input. Since each of the components produces output conforming to a certain TO, the outcome of each component can be sent to concrete linked data service instances as input to execute the requested functionality (i.e. trigger the business processes associated with the TO).

The approach has benefits over existing approaches for the automated generation of UIs for aggregated linked data applications:

- It uses a sharable artefact that allows generating UIs, that can be integrated into generic linked data applications
- The generated UIs are able to produce output for arbitrary linked data services by incorporating mapping rules for arbitrary target ontologies
- It uses a single, self-contained artefact to be easily shared, containing all structural and behavioural information to generate executable frontends
- The resulting UI components can be aggregated to build novel applications

The following sections focus on the details of the approach: first the information required for UI derivation and its ontological description is outlined, followed by the enhancement of the model towards information needed to derive a target instance from user input. Next we outline the processes for UI derivation and target ontology instance generation. Finally we focus on the use of the model to build aggregated linked data applications.

4 Application Ontologies for Automatic UI Derivation

The proposed *Application Ontology* is partially based on results of previous work of the *mimesis project* [8]. The approach uses a model of the data processed by the application as foundation, which is enhanced by additional information regarding its

semantics. The following sections summarize this information and show its extension to an ontological application description.

4.1 Information Requirements for Automatic UI Generation

To derive the information required for generating UIs, a set of interaction patterns was identified by analysing existing, frequently used ‘real-life’ applications along with a review of related work [8]. Next, the data necessary to build UIs for these patterns was extracted. Table 1 summarizes the required information along with its usage within a UI derivation process:

- **Type related and Structural Information** (I_1 – I_4) is needed to describe data elements (i.e., types and type restrictions like *ranges* or *allowed values*), their structure (i.e., *grouping* and *hierarchical correlation*), and a meaningful *temporal sequence* of the questions to gather the data.
- **Behavioural Information** (I_5 – I_7) is needed to model dynamic aspects of the UI at runtime. This includes conditions about the *existence/activation* of *elements/groups* bound to the content of other data elements within the model, the indication for complex *validations*, *operations* triggered on changes of the input data (*reactions*) or triggered by the user (*actions*).

Table 1. Information needs and usage for UIs [8].

Ref	Information need	Usage for UI derivation
Type related & structural information		
(I_1)	Type information for a data element or group (based on XMLSchema)	Selection of suitable input control based on type restrictions (e.g. presets and value ranges); provision of type-related validations
(I_2)	Hierarchical grouping of elements	Grouping of questions into display units; dependencies and hierarchical inclusion of groups; derivation of suitable navigation structures (sequential, tree,...)
(I_3)	Temporal succession of data- or group elements	Display order of groups and input controls
(I_4)	Semantic cohesion of elements	Arrangement of controls (e.g. proximity of a <i>zip code</i> and <i>city</i>); identification of possible breakpoints for pagination
Behavioural information		
(I_5)	Existence and activation conditions for data and group elements	Show/hide or de-/activate groups and questions, triggered on change of already entered data
(I_6)	Validation operations	Trigger (complex) validations operations usually related to already entered data
(I_7)	Actions and reactions	Trigger operations on change of already entered data (reaction) or initiated by the user (action)

Based on the processed data, a meta model can be created that incorporates the identified information and serves as a foundation to develop data descriptions for interview applications. Figure 3 shows this meta model as UML diagram.

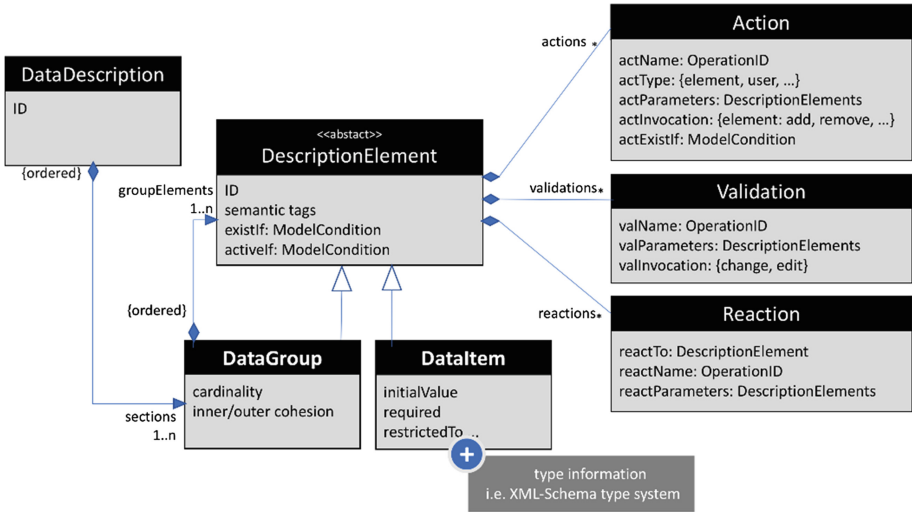


Fig. 3. Meta-model in UML notation [8].

Table 2. Facets for Datagroups and Dataitems [10].

Facet	Description	Contents
DescriptionElement		
<i>name*</i>	Unique name as identifier for the element	[a-zA-Z0-9]+
<i>type</i>	Type of the group or data item	s. below
<i>existsIf</i>	Condition for the existence of the group or element. if it evaluates to true, the data is relevant and presented	Boolean expression. Referencing model items
<i>activeIf</i>	Condition for the editability of the group or element. if it evaluates to true, the data is editable, else just displayed	Boolean expression. Referencing model items
DataGroup		
<i>type</i>	Type of the group	
<i>cardinality</i>	Possible cardinality of the group. Defines, how often the group might be repeated. (e.g., used to express, that a person might have multiple addresses)	*: no limit <n>: fixed value <n> .. <m>: range

(continued)

Table 2. (continued)

Facet	Description	Contents
<i>DataItem</i>		
<i>type</i>	Type of the data item simple datatypes: semantics according XML-Schema custom datatypes e.g. domain or context specific. implies additional behavior (e.g. country specific validation for a zip code)	simple datatype: text, number, boolean, date, float custom datatype: email, zipcode, phone, licenseplate
+ <i>restrictions</i>	Additional type specific constraints XMLSchema (e.g. min/maxInclusive)	Additional facets for datatypes
<i>restrictedTo</i>	Restriction of possible vaules	Value ranges, e.g. dog cat mouse
+ <i>multiple</i>	Allows multiple values to be selected	True, false
<i>required</i>	Indicates that the data is not optional	True, false
<i>initialValue</i>	Initial value of the content	Depending on type and restrictions

A data description (*DataDescription*) consists of a succession of data groups (*DataGroup*) that might contain an ordered list of further groups or data elements (*DataItem*). This constellation allows to model the requested **structural information** regarding cohesion, (hierarchical) grouping and temporal sequence of the elements (I_2 , I_3 , I_4). Groups and data items are detailed by attributes/facets. E.g., **type information** (I_1) and **existential and activation conditions** (I_5) can be specified for each description element in the model. Further facets are used to specify the element more precisely in terms of data related aspects, i.e., *type restrictions* that are usually part of a type system like XML-Schema (I_1). Table 2 summarizes the semantics of the facets for *DataGroups*. and *DataItems*. Additionally, each description element might have associated **validation-, reaction- and action operations** (I_6 , I_7), which are complemented by further facets like name of the operation, triggering events, and model elements required for the execution of the operation (cf., [8]).

The model meets the requirements regarding UI descriptions (**Req. 1**) as it contains all structural and behavioural information to derive non-trivial UIs within a single description.

5 Using Ontologies as Representation

To get a sharable representation, the meta-model is applied to RDF/OWL. The objective is to map the information requirements (I_1 – I_7) listed in Sect. 4.1 towards RDF/OWL and hence develop a sharable **Application Ontology**. This is done by projecting the elements contained in the meta-model onto RDF/OWL elements.

The general intention of ontologies is to describe entities, relationships, contained data elements and additional facts. Hence, expressing most of the structural information with RDF/OWL is straightforward: *DataGroups* can be modelled as *owl:Classes* and their hierarchical relations as *owl:ObjectProperties*. *DataItems* are defined as *owl:Data-typeProperties*.

Listing 2 shows a fragment of the Application Ontology for the flight booking example introduced in Sect. 2 as illustration of the mapping. The *Classes* section (Listing 2, ❶) declares the *DataGroups* (e.g., *Flightbooking*, *Flightinfo*, *CustomerInfo*) as part of the application ontology (i.e., <http://...bookers/flight/v1#>). Examples for relations appear in the *Object Properties* section (Listing 2, ❷ - e.g., *Flightinfo* as an object property of *Flightbooking* with range *flightinfo*). Contained *DataItems* appear in the *Data Properties* section (Listing 2, ❸) with information to which class they belong to, along with *basic* type information (e.g., exemplary data associated with a *Flight* and *ReturnFlight*). Using these basic RDF/OWL concepts, the structural information of I_2 and I_4 and partially I_1 are covered.

Not all of the identified information can be expressed with standard RDF/OWL means. Ontologies in general do not contain information such as the *sequence of data* (I_3), *existential conditions* (I_5) or *functional aspects* (I_6 , I_7). To the best of our knowledge, RDF/OWL does neither include a concept for the description of operations nor for declaratively modelling conditions/references based on instance data.

To express this information, we use the **OWL annotation concept** as applied in Khushraj and Lassila [12] and Gaulke and Ziegler [7] to produce a **profiled ontology**. This allows incorporating the information *declaratively*. This leads to an ontology, that is (1) still covered by basic RDF/OWL (and thus can be used for standard reasoning) yet (2) exposes the additional information for reasoners (e.g., UI generators) that understand the specific profile.

Table 3 lists the used annotations of the proposed profile along with their mapping to the information needs. As an example, Listing 2 ❹ shows annotations for *type*, *sequence*, *existence* and *reactions* applied to elements of the sample ontology.

Listing 2. Application Ontology (excerpt) in OWL/Turtle notation.

<pre>@prefix mdt: <http://mimesis/datatypes#>. @prefix owl: <http://www.w3.org/2002/07/owl#>. ... @prefix ma: <http://mimesis/annotations/v1> @prefix sa: <http://mimesis/linkedata/v1> @base <http://mimesis/bookers/flight/v1> . 1 Classes : :Flightbooking rdf:type owl:Class . :Basictraveldata rdf:type owl:Class . :Flightinfo rdf:type owl:Class . :Customerinfo rdf:type owl:Class . :Persons rdf:type owl:Class . :Flight rdf:type owl:Class . :Returnflight rdf:type owl:Class . :Openjawflightinfo rdf:type owl:Class . :Customer rdf:type owl:Class . :Address rdf:type owl:Class 2 Object Properties: :Flightbooking.basictraveldata rdf:type owl:ObjectProp...; rdfs:range :Basictraveldata ; rdfs:domain :Flightbooking . :Flightbooking.flightinfo rdf:type owl:ObjectProperty ; rdfs:domain :Flightbooking ; rdfs:range :Flightinfo .</pre>	<pre>:Flightbooking.customerinfo rdf:type owl:ObjectProp... ; rdfs:range :Customerinfo ; rdfs:domain :Flightbooking . :Basictraveldata.persons rdf:type owl:ObjectProperty ; rdfs:domain :Basictraveldata ; rdfs:range :Persons . :Flightinfo.flight rdf:type owl:ObjectProperty ; rdfs:range :Flight ; rdfs:domain :Flightinfo . :Flightinfo.returnflight rdf:type owl:ObjectProperty ; rdfs:domain :Flightinfo ; rdfs:range :Returnflight . :Returnflight.openjawflightinfo rdf:type owl:ObjectP...; rdfs:range :Openjawflightinfo ; rdfs:domain:Return...</pre> <p>3 Data Properties</p> <pre>:Flight.fromdestination rdf:type owl:DatatypeProperty ; rdfs:domain :Flight ; rdfs:range xsd:string . :Flight.todestination rdf:type owl:DatatypeP...; rdfs:domain :Flight ; rdfs:range xsd:string . :Flight.startdate rdf:type owl:DatatypeProperty ; rdfs:domain :Flight ; rdfs:range xsd:date . :Flight.returnflight rdf:type owl: DatatypeProp...; rdfs:domain :Flight ; rdfs:range xsd:boolean . :Returnflight.returndate rdf:type owl:Datat...; rdfs:domain :Returnflight ; rdfs:range xsd.date .</pre>
<p>4 Profile Annotations</p> <pre>:Flightinfo.flight ma:sequence "1" ; :Flight.startdate ma:sequence "1" ; ma:type "date" ; :Flightinfo.returnflight ma:existsif "(returnflight == true)"; ma:sequence "2" . :Flight.returnflight ma:sequence "4" ; ma:type "boolean" ; ma:initialValue "true" . :Returnflight.returndate ma:sequence "1" ; ma:type "date" ;</pre>	<pre>:Flight.fromdestination ma:sequence "2" ; ma:restrictedTo "flightbooking .getDepartureAirports()" ; ma:type "text" ; :Flight.todestination ma:restrictedTo " " ; ma:sequence "3" ; ma:type "text" ; ma:activef "fromdestination.length0" ; ma:reactions "fromdestination:flightbooking .changeDestinations(fromdestination, \$destination)" . :Returnflight.openjawflightinfo ma:existsif "(openjawflight == true)" ; ma:sequence "3" .</pre>

The result is an ontological description of the UI-specific aspects of the application. It permits a single artefact incorporating all information contained in the meta model of Sect. 4.1 needed to derive non-trivial UIs (cf. Sects. 6 and 7). As it contains information about relations to model elements, it even allows consistency verification of the modelled UI. The mapping to RDF/OWL leads to an *ontological description* for dialog-based application UIs. It is sharable and thus meets requirements **Req. 1** and **Req. 3**. The resulting UI is able to collect user input and provide it for further processing (i.e., as an instance of the AO).

Yet, the approach has some *limitations regarding its universality*. Using a *profiled ontology* with proprietary annotations requires a reasoner that is aware of the profile. The additional information is not interpretable by generic reasoners.

Table 3. Additional annotations [10].

Annotation	Content	
Type related & structural information		
<i>:sequence</i>	Number - position of the element in the flow of questions	<i>I₃</i>
<i>:type</i>	Typeinformation for a group or element	<i>I₁</i>
<i>:<constraint></i>	typerelated constraints - > XMLSchema, e.g. :restrictedTo, :initialValue, :max, :min	<i>I₁</i>
Behavioural information		
<i>:existIf</i>	Conditional expression References data within the hierarchy using path expressions at runtime for an instance	<i>I₅</i>
<i>:activeIf</i>	Conditional expression References data within the hierarchy using path expressions at runtime for an instance	<i>I₅</i>
<i>:validations</i>	Definition of validation, reaction and action operations	<i>I₆</i>
<i>:reactTo</i>	Validations syntax: <trigger>:<operation> (<parameter>*)	<i>I₇</i>
<i>:actions</i>	Reactions syntax: <element>:<operation> (<parameters>*) Action syntax: <type>:<trigger>:<operation> (<parameter>*)	

6 Modelling the Relation to Target Ontologies

By this time, the model does not contain information about how to map user input to instances to the Target Ontology required by a linked data service. This section shows, how the data entered in the UI can be prepared for further processing.

When the user enters data, he actually builds an instance of the AO (*Application Ontology instance*, AOI). To derive a *Target Ontology instance* (TOI), a transformation from an AOI to a TOI is required. The main task is to map data elements of the AOI to the required structure for the TOI.

AOI groups and data elements are related to elements of the TOI - although they might appear in a different structure. *DataGroups* are related to objects in the target ontology. *DataItems* correspond to data properties of object instances in the TO. Figure 4 shows this for the flight booking example. It shows that the *flightbooking* instance of the AOI is associated with the *:flightbookingrequest* of the TOI - as is the *flight* to the *:flight* instance. Flight information as the *startdate*, *from-* and *todestination* need to be mapped as data properties of the *:flight* instance. The *returndate* and additional open-yaw-flight information maps into the *:flight* instance, despite being part of a different *DataGroup* of the AOI (an example for structural differences between AO and TO).

Figure 4 shows as well the information needed, on the TOI side: to represent an object instance, its type needs to be known (e.g., *rdf:type Flight* for the *:flight* object instance). For a data property, its property name, type and the instance value is needed (cf. Listing 1, Sect. 2).

To meet requirement **Req. 2**, the mapping information needs to be integrated into the AO. Again we apply the **OWL annotation concept** to express this information, as in Sect. 4 for additional data semantics. Hence a profile for expressing the linked data context is added for the AO.

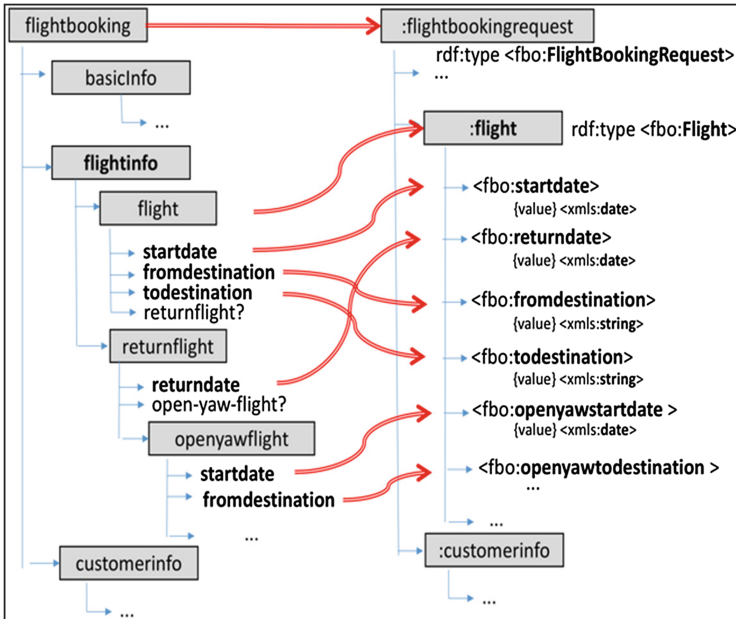


Fig. 4. Mapping AOI data to TOI [10].

The profile annotations used for that purpose are summarized in Table 4. For each *DataGroup* in the AO, that corresponds to an object instance in the TOI, an instance name (e.g. *:flight*) and the type needs to be specified. If the object is associated with another object (e.g. *:flight* as part of *:flightbookingrequest*, Fig. 4), information about the parent instance and the propertyname within that object needs to be supplied. For a *DataItem*, its type and propertyname is needed (e.g. *<fbo:startdate>* with type *<xmls:date>* for the departedate of the flight, cf. Figure 4) along with the instance, the dataproperty is associated with (e.g. *startdate* as part of *:flight*). Listing 3 shows the annotations for the flight example.

Given an AOI and the AO, the TOI can now be generated by traversing the AOI tree nodes: If passing a *GroupItem* node with annotated TOI information, an RDF triple for an instance is created exploiting the *DataGroup* annotations (cf. Table 4, *:swIndividual*, *:swClass*). If there is a relation to another instance, an *ObjectProperty* triple is

generated to reflect the relation (:swForIndividual, :swProperty). If passing a *DataItem node*, a RDF triple for a *DataProperty* is created, using (a) the type, name and relation annotations and (b) the instance data entered by the user for the corresponding field in the AOI.

This approach allows the automatic generation of a suitable TOI from an AOI based on information contained in the AO and thus meets **Req. 2** and **Req. 3**.

Our approach uses a simplified method for mapping AO instance data to the TOI, allowing only a unidirectional mapping of the data onto the TOI. This restricts the contained data to the use case we focus on, but does not allow mapping back from TOI to an AOI (for example, this could be used to preset data). There exists research on bidirectional tree transformations (e.g. [6]), which can be applied to extend the proposed solution in future work. Additionally, since a profiled ontology is used, the restrictions discussed in Sect. 4 also apply here.

Table 4. Linked data profile annotations.

Annotation	Description
DataGroup/ObjectProperty annotations	
:swIndividual	Name of the instance to be generated
:swClass	Object type/class of the group in target ontology
:swForIndividual*	Name of the instance, this item is associated with
:swProperty*	Object property name, this item has in the associated instance
* = for nested object properties only	
DataItem/DataProperty annotations	
:swType	Type of the data property in the target ontology
:swForIndividual	Name of the instance this data item is associated with
:swProperty	Data property name, this item has in the associated instance

Listing 3. Linked Data Annotations.

```

:Flightbooking.flightinfo
sa:swClass "fbo:FlightBookingRequest" ;
sa:swIndividual "flightbookingrequest" .

:Flightinfo.flight
sa:swClass "fbo:Flight" ;
sa:swProperty "fbo:flight" ;
sa:swIndividual "flight" ;
sa:swForIndividual "flightbookingrequest" .

:Flight.startdate
sa:swProperty "fbo:startdate" ;
sa:swForIndividual "flight" ;
sa:swType "xmls:date" .

:Flight.fromdestination
sa:swProperty "fbo:fromdestination" ;
sa:swForIndividual "flight" ;
sa:swType "xmls:string" .

...

:Returnflight.returndate
sa:swProperty "fbo:returndate" ;
sa:swForIndividual "flight" ;
sa:swType "xmls:date" .

:Openjawflightinfo.openyawfromdestination
sa:swProperty "fbo:openyawfromdestination" ;
sa:swForIndividual "flight" ;
sa:swType "xmls:string" .

:Openjawflightinfo.departuredate
sa:swProperty "fbo:openyawstartdate" ;
sa:swForIndividual "flight" ;
sa:swType "xmls:date" .
    
```

7 Generating the UI- and Target Ontology Instances

As outlined in Sect. 3, two transformations are needed to bring an application to life: (1) a *User Interface Transformation* to generate the final UI to be presented to the user, and (2) a *Target Instance Transformation* building a Target Ontology instance when user input is ready. Figure 5 shows the transformation steps needed for the overall solution. To generate a UI based on the AO, the approach presented by *mimesis* [8] is used. It is based on the concepts of the CAMELEON framework introduced by Calvary et al. [3].

Table 1 in Sect. 4.1 showed the information contained in the AO and how it is used for the derivation of UIs. Figure 5 (c.f., *User Interface Transformation*) outlines the steps for this transformation. The process starts with an instance of the *data-centric core model*, which is built from the information contained in the AO. The core model describes the processed data of the application according to the structure and properties presented in Sect. 4.

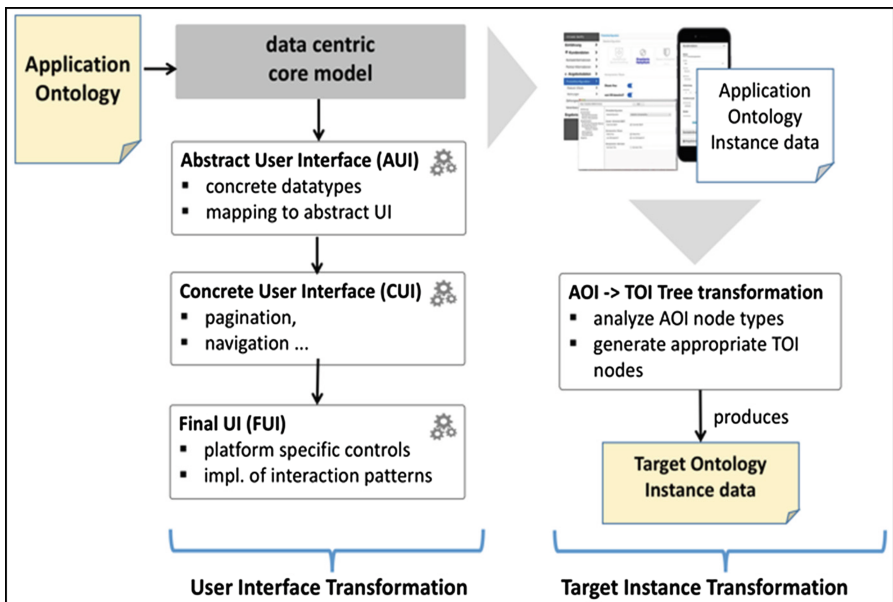


Fig. 5. Derivation process [10].

- **Step 1:** the core model is transformed to an *abstract UI* (AUI) using information about the *context of use* to concretize the information contained in the model. This step is crucial to generate usable UIs from a solely data-centric model that intentionally omits technical details. This includes enrichment with labels and texts depending on the language context, the mapping of data types to concrete types of the AUI (e.g., to map a German zip code to a text field restricted to 5 digits) and

abstract UI input elements to be used. The information here is derived from I_1 , I_2 , I_3 and I_4 (cf. Table 1)

- **Step 2:** derives a *concrete UI* from the AUI description by incorporating the *device context* for which the UI is intended. It maps fields to pages by using information about device restrictions and cohesion information contained in the model. The latter indicates how a flow of questions may be split up and positioned on pages for different device categories. The information needed here is derived from I_2 and I_4 .
- **Step 3:** Depending on the *technological context* the final UI is derived by generating now concrete UI Widgets for the abstract controls of the AUI and by implementing the functional aspects for the specific platform. This exploits the behavioural information contained in the basic application model. The information needed here is derived from I_1 , I_5 , I_6 and I_7 .

These steps transform the model to a final UI, which is presented to the user on a specific platform. After the user finishes input, an AOI containing the input data is available. That needs to be mapped to an instance of the TO as outlined in Sect. 5. This results in the last, deferred step of the process (Fig. 5., *Target Instance Transformation*).

- **Step 4:** Traversal of the instance data tree within the AOI and generation of a TOI based on the instance mapping annotations contained in the AO. The resulting data object can be consumed by a linked data service following the target ontology.

8 Aggregation of Sharable Application Ontologies: A Novel Paradigm for Composite Applications

The presented approach enables the description and automatic generation of a UI for a single business process in a sharable way. It also offers the opportunity for a novel kind of composite applications, that integrate services on the UI level.

8.1 Aggregation Scenario

In Fig. 6 a **generic solution scenario** is shown for the use of the proposed Application Ontology to build aggregated frontends in a linked data environment (introduced in Sect. 3). The central elements are the *Target Ontologies* (TO) and the corresponding *Application Ontologies* (AO) - both sharable as reusable artefacts. The TOs define the semantics of the input data for a business processes and thus are the connecting link between an application and its backend. The AOs define variants of the user data to be gathered as input for a specific business process and contain the information to produce output conforming the Target Ontologies.

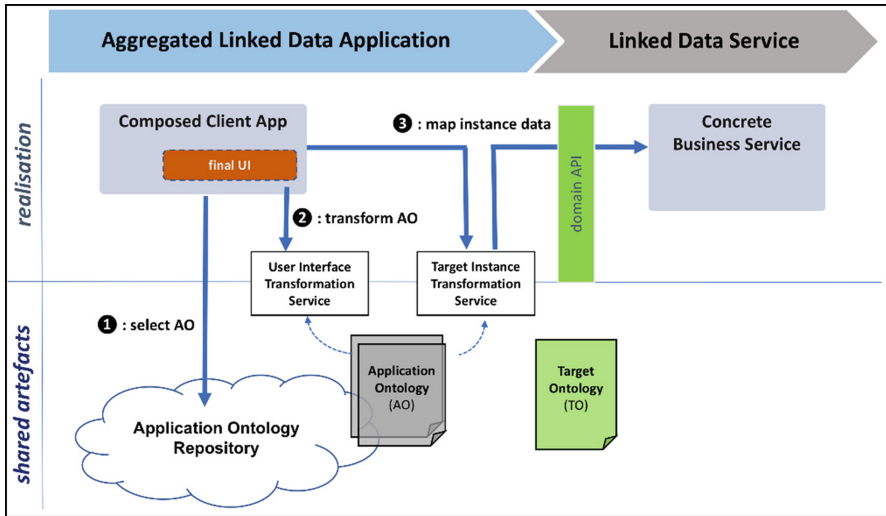


Fig. 6. Generic Solution Scenario architecture.

Figure 6 shows the **process for aggregating multiple business processes** into one common UI based on the AO and the TO. In the depicted scenario, a *Composed Client Application* selects multiple AOs **1** from an arbitrary source (e.g. a local repository or public resources on the internet).

For each of the selected AOs, the UI is generated **2**. To achieve this, a generic *User Interface Transformation* is used, that derives the UI based on the UI related information contained in the AO (structure, dynamics, etc.) following *steps 1 to 4* outlined in Sect. 6.

The resulting UI-components are then aggregated into a final composed UI of the client application. The combination is presented to the user for input.

When the user finished entering information, the collected data for each AO component (i.e., an *instance of the AO*) is sent to a *Target Instance Transformation*, which exploits the mapping-related information available in the AOs to generate an instance of the corresponding TO **3** (outlined in Sect. 6, step 4).

The result is a collection of Target Ontology instances. Since the TOIs comply to the expected data of the associated linked data services in the backend, they can be used as input for the business process. The last step of the process would be to select a concrete linked data service as target and to submit the TOI for further processing.

8.2 A Novel Application Integration Paradigm

The above approach has an impact on the way applications are built. It results in a novel way to integrate application components as **it may completely decouple the UI from the targeted backends – at design time and at runtime**. This is possible by exploiting the benefits of semantic web and linked data approaches. Both, the Application- and the Target Ontologies are **shared knowledge** in that scenario. They have clearly defined semantics accessible and interpretable by everyone.

Benefits at Design Time

Since data is semantically specified by ontologies as common knowledge, providers of services and application frontends can be freely exchanged. Both can provide reusable artefacts based on the common vocabulary:

- **Service providers** can offer their domain service by implementing a shared and commonly standardized API (i.e. the Target Ontology) by clearly indicating what data is expected from a consumer of the service.
- **Application UI providers** can specify application variants (i.e. Application Ontologies) relying on the Target Ontologies. Since common linked data technologies are used for the descriptions, they can be shared and reused by anyone.
- **Aggregators** can select from an arbitrary collection of Application Ontologies to aggregate functionality and generate the frontend they need by using generic Transformations.

Using the approach, *Service Providers* (e.g. businesses or other organisations) are enabled to participate in an open linked data ecosystem. They are not responsible for providing UIs for their services. Anyone can provide UI variants as an *Application UI provider* and expose them to be used by anyone (e.g. through local, public or community based repositories providing AOs for certain domains). Finally, *Aggregators* (e.g. businesses, third party businesses or even end users) can freely choose from the shared AOs and are even free to choose the service providers, that process the business functionality in the end.

Benefits at Run Time

Even runtime components might be shared within this ontology-driven scenario. *User Interface Transformation* and the *Target Instance Transformation* mostly rely on information available within the shared Application- and Target Ontology – which is common knowledge. Hence, even transformation services might be supplied by anyone, that perform the transformation to a specific UI based on the common vocabulary:

- **Application Ontology Repository Providers** might offer services to manage and offer for example user group focused, reviewed and quality tested AOs for reuse.
- **User Interface Transformation Providers** might offer generators for different technologies, platforms or user groups (e.g. outdoor workforce, disabled-persons) relying on the informations in the Application Ontology.
- **Aggregators** can decide which Transformation Provider to choose to derive a specific UI for the targeted user group/technology.

Application Ontology Repository Providers (e.g. companies, organisations, communities) may play the role of distributors for AOs. They can offer AOs for selection based on specific criteria (e.g., community-, quality- or domain related) and hence ease the selection process for Aggregators. *User Interface Transformation Providers* (e.g. businesses, organisations, communities) are enabled to contribute to UI generation for the linked data ecosystem. They can provide UI transformations for e.g. very specific situations and contexts reusable by everyone. *Aggregators* are enabled reuse a variety and to provide a multitude of – literally generic - UIs.

This novel approach leads to a high degree of freedom in service aggregation on UI level and enables the development of business model concepts that combine services from different domains – like the aforementioned Distributed Marketspaces (e.g., [18]).

9 Validation

This section focuses on the validation of the stated objectives to show, that (1) ontologies can be used to describe application UIs in a non-proprietary way, which (2) can be used to produce output conforming a target ontology. It shows as well, that Application Ontologies provide (3) a sharable, reusable representation, which (4) can be combined to generate aggregated UIs for arbitrary linked data applications.

The validation was carried out in association with a major German insurance company from which we got data for the evaluation and which already uses parts of our implementation results in production environments (i.e., to generate UIs of *electronic risk acceptance check* applications for different products on customer and agent portals).

For the analysis phase and evaluation of the implementation, the company provided a set of typical ‘real-life’ dialog-based applications. From this set, relevant applications were selected that cover the interaction patterns identified during analysis and to demonstrate the usefulness of the automated process and the proposed Application Ontology.

To allow a deeper investigation, a DOI is provided below¹. It points to sample resources for the *flightbooking application* used throughout this paper. It presents a working example of application variants and the complete application models.

Basic Setting and Preparation

As basic design for the evaluation, we chose the architecture and building blocks outlined in Sect. 7 (cf. Fig. 6) and implemented the required components for that setting to be used in further validation steps.

First, the *User Interface Transformation* component was implemented as outlined in Sect. 6, which resulted in a *UI Transformation Service* (exposed as a web service). The implementation is based on available components from previous work [8]. We reused the transformation and – for a comparative evaluation – an import module for a proprietary application DSL (Domain Specific Language) that was developed as part of the previous work. The *UI Transformation Service* transforms a data-centric core model to a final UI for different platforms. It focuses on web-based dialog applications (using HTML, JavaScript, CSS) for different device categories (mobile, desktop).

As a second step, an import module was implemented, reading the proposed Application Ontology and converting it into the core data model of the *Transformation Service*. Third, the *Target Instance Transformation* (cf. Figure 6) was implemented as a web service. It consumes an AO and instance data as input, producing a TOI based on the contained data as outlined in Sect. 5.

¹ <https://doi.org/10.13140/RG.2.2.24909.23520>.

Applicability of Ontologies

To validate the applicability of the ontological approach, a *comparative evaluation* was chosen based on the implementation of the *UI Transformation Service*. Figure 7 shows the basic setting for the evaluation. The goal is to demonstrate that the proposed ontology has the same expressive power as the *mimesis* DSL, which was already evaluated in previous work. To achieve this, the same applications were modelled using (1) the *mimesis* DSL and (2) the Application Ontology. Both were transformed to the core model of the transformation service and the generated output was compared.

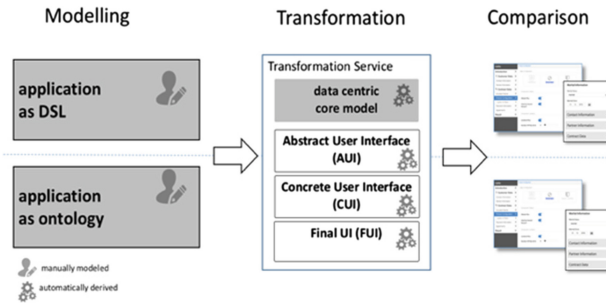


Fig. 7. Basic setting for comparison [10].

Results: The results show that both kinds of descriptions can be mapped to the same core model and bear the same expressive power. The implementation shows that the proposed approach for using Application Ontologies to describe UIs leads to the same results as the solution using the proprietary *mimesis* DSL. While not being a formal proof, the results indicate that the data-centric approach may be applied to ontological descriptions of dialog applications. The evaluation showed as well, that the DSL (as proprietary approach) was much easier to use and less error prone than manually building AOs from scratch. But since the expressive power of both approaches is the same, it is possible to use the DSL for modelling and automatically transform the model into the proposed AO – preserving the benefits of both approaches.

Aggregation of Sharable and Reusable Application Descriptions

For the suitability of the proposed ontology as sharable, reusable application descriptions for aggregated linked data applications, we applied the approach to a concept for *Distributed Market Spaces* working with generic UIs for the specification of complex product requests. This concept is already published in [9] and summarized here. The objective is to show that Application Ontologies can be (1) shared and used to generically build aggregated UIs and (2) can produce linked data requests – in this case to build a complex product request from user input.

Figure 8 shows the basic architecture of the demonstrator. As generic user frontend a *Complex Product Builder* (CPB) application was implemented, that lets users search and select arbitrary *Application Ontologies* (AO) as proposed in this paper (Fig. 8, ①). These were drawn from a shared UI description repository containing AOs for different

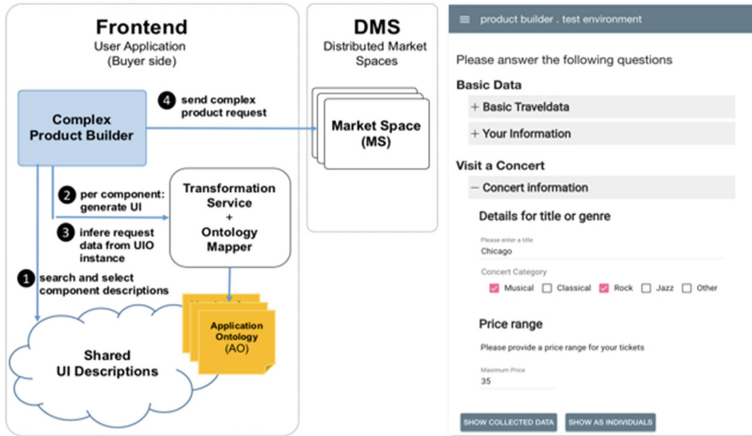


Fig. 8. Generic UIs for complex product requests [10].

product components (e.g., booking a concert ticket or a flight). The user-selected AOs are sent to the *Transformation Service* (Fig. 8, ②), which returns generated UIs for each AO. These were aggregated to a final UI (Fig. 8, right). Since the UIs are generated from the elements contained in the AO, the user input relates to the corresponding ontology elements. This allows building *an instance model* for each presented AO containing the input data of the user using an *Ontology Mapper* (Fig. 8, ③). The result is a set of ontology instances on which a reasoner can build a *complex product request*, which is sent to the *Market Space* for further processing (i.e. generating a quote for the requested product components).

Results: Despite the demonstrator is still a proof-of-concept it shows that sharing application descriptions is possible. In future work, AOs could be assembled from arbitrary sources (e.g., topic-related repositories for *insurance*, *travel planning*, etc.) and hence UIs for arbitrary domains might be generated. In addition, it shows that target ontology instances can be derived from user input data using the mapping information contained in the AO.

Although the approach lead to satisfying results and is easy to implement, we observed a drawback regarding the user friendliness in modelling the mapping for bigger AOs: hence the approach is focused on the AO, the information of the TOI is scattered all over the AO model and thus hard to grasp and maintain. Future work might focus on better tool support for this task or advanced mapping concepts.

10 Related Work

The research on the automatic generation of UIs covers many contributions during the last years that are based on model-driven concepts.

User Interface Description Languages (UIDL) focus mainly on the description of concrete UIs in a technology independent way. Examples are *JavaFX* (Fedortsova [5]),

UIML (Abrams et al. [1]), *UsiXML* (Limbourg [14]) and *XForms (W3C)*. The basic idea is to model dialogs and forms by using technology independent descriptions of in-/output controls and relations between elements (e.g. visibility) within a concrete UI. **Task-/conversation-based approaches** describe applications by dialog flows which are derived from task models – e.g. *CAP3* (Van den Bergh et al. [21]), *MARIA* (Paterno et al. [17]) and conversation-based approaches e.g. (Popp et al. [19]). They focus on a concrete model of the dialog flows. To generate an application frontend, the steps in a dialog flow are associated with technology independent UI descriptions displayed to the user. **Data-centric approaches** can be found in *JANUS* (Balzert et al. [2]) and *Mecano* (Puerta et al. [20]) which use a **domain model** as starting point for the derivation of UIs. While *JANUS* was designed to only provide CRUD-like interfaces for applications that work on a persisted domain model that does not support much dynamics in the UI, *Mecano* adds these aspects to its description.

Existing **Ontology-based approaches** generally rely on the concepts of the mentioned approaches and use ontologies to represent the information about concrete UIs. For instance, in analogy of UIDL approaches, Liu et al. [15] propose an ontology driven framework to describe UIs based on concepts stored in a knowledge base. Khushraj and Lassila [12] uses web service descriptions to derive UI descriptions based on a UI ontology, adding UI related information to the concept descriptions (profile). In analogy with task-based approaches, Gaulke and Ziegler [7] use a profiled domain model enriched with UI related data to describe a UI and associate it with an ontology driven task model.

Dissociation: The major requirements for the proposed model stated in in Sect. 2, were to use a sharable representation that can be reused in different contexts. It has to be used for UI derivation, must contain information for the mapping of user input to target data and needs to be a self-contained description, suitable for aggregation on the UI-level. (*Req. 1* to *Req. 5*).

Although the models of the aforementioned approaches usually are suitable to derive technical variants for specific environments (e.g. web applications), they do not contain enough semantical information, that could be used for deriving UI variants for different contexts of use (e.g. platforms, user groups). The UIs are manually modelled using a occasionally large number of artefacts which usually are proprietary and UI-specific. In addition, most approaches do not offer concepts for declaratively specifying the mapping to associated backend services. This impedes the reuse as shared, self-contained artefacts and their generic aggregation to more complex applications

The solution proposed in this paper is based on the application's processed data and enriches its model by additional semantics. This leads to a **single, central description for the application that serves as a knowledge base for the automatic derivation of functional UI variants**. The data-centric approach allows the reuse of the models in different contexts and - by using a non-proprietary representation for the model - the sharing and integration into different environments. Since the descriptions contain both the information to derive UIs and the mapping to standardized linked data services, it is possible to automatically generate fully functional applications for existing linked data service ecosystems.

11 Conclusion and Future Work

In this paper an ontology-based, model-driven approach for modelling and automatic generation of UIs for linked data services is presented. It allows a technology-agnostic, sharable and reusable description of UIs for arbitrary domain-specific services relying on linked data principles. The paper presents an approach for the reuse of such descriptions, allowing to build novel applications by aggregation of services on the UI level.

The novelty of this approach lies in the complete decoupling of the UI- and the service provider – taking benefit of semantic web and linked data approaches. It is a clear shift away from solely describing specific UIs towards a description of combined application components - which is a major differentiator to existing approaches.

The approach is based on an ontological model of the processed application data, enhanced by type-related, structural and behavioural information needed to generate non-trivial UIs. It contains additional information on how input data maps to input data of targeted linked data services. The model represents a self-contained, ontological description of the application data for a business process. The contained information is used to generate UIs and to transform user data to input data of the targeted service. Since the model is self-contained, it can be combined with other services.

In the course of the paper, we first identified the information needed to generate non-trivial UIs and presented a meta-model, that contains the information generating the application frontend. From this, an ontological model for applications was derived by applying RDF/OWL to get a non-proprietary, sharable representation. The information needed to map input data to target service data was identified and used to enhance the model. Next, the process for UI- and target data transformation was outlined, showing the steps to derive a final UI and the generation of the requested target data based on the user input. An approach for the aggregation is outlined, that reuses and combines the aforementioned artefacts as components for integration on the UI level. Finally, the evaluation is presented which provides an implementation of the generation process for UIs from an Application Ontology and a demonstrator for the aggregation approach.

The results of the evaluation indicate the applicability of the proposed Application Ontology for generating UIs for linked data applications: the evaluation demonstrated, that (1) non-trivial UIs can be automatically generated based on a (2) sharable Application description, which (3) may be aggregated to build novel applications with augmented functionality. As the demonstrator shows, it is possible to automatically generate aggregated UIs that may span different domains. The final UI is able to produce an output that can be processed by related linked data services using the information contained in instances of the proposed Application Ontology.

Future work: Currently the approach is intentionally limited to dialog based applications, as being a very important and frequently used application type in enterprise information systems. Since a limited set of applications was used for analysis, we do not claim completeness of the identified interaction patterns. The practical use of the approach might bring forth additional interaction patterns, extending the basic information set in future. Regarding the concept for aggregated applications, our research

does not cover all aspects yet. Currently, application descriptions are considered to be separate, self-contained parts - ignoring possible dependencies between them on the data level. Furthermore, it does not consider a wider user-related context that components and their UIs may share and react to. Future work might focus on these interaction aspects on the data level, which might add additional dependency and context information to the proposed Application Ontologies.

References

1. Abrams, M., et al.: UIML: an appliance-independent XML user interface language. In: WWW 1999 Proceedings of the Eighth International Conference on World Wide Web, pp. 1695–1708 (1999)
2. Balzert, H., Hofmann, F., Kruschinski, V.: The JANUS application development environment - generating more than the user interface. In: Computer Aided Design of User Interfaces, vol. 96, pp. 183–206 (1996)
3. Calvary, G., et al.: The CAMELEON Reference Framework, Components, vol. 60 (2002). <http://giove.isti.cnr.it/projects/cameleon.html>
4. Coutaz, J.: User interface plasticity: model driven engineering to the limit! In: EICS 2010 Proceedings of the 2nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems, pp. 1–8 (2010)
5. Fedortsova, I., Brown, G.: JavaFX Mastering FXML, Release 8. JavaFX Documentation (2014). <http://docs.oracle.com/javase/8/javafx/fxml-tutorial/preface.htm>
6. Foster, J., Greenwald, M., Moore, J.: Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. *ACM Sigplan* **3**, 1–64 (2005)
7. Gaulke, W., Ziegler, J.: Using profiled ontologies to leverage model driven user interface generation. In: Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems – EICS 2015, pp. 254–259 (2015)
8. Hitz, M.: mimesis: Ein datenzentrierter Ansatz zur Modellierung von Varianten für Interview-Anwendungen. In: Nissen, V., et al. (eds.) Proceedings - Multikonferenz Wirtschaftsinformatik (MKWI) 2016. pp. 1155–1165 (2016)
9. Hitz, M., Radonjic-Simic, M., Reichwald, J., Pfisterer, D.: Generic UIs for requesting complex products within distributed market spaces in the internet of everything. In: Buccafurri, F., Holzinger, A., Kieseberg, P., Tjoa, A.M., Weippl, E. (eds.) CD-ARES 2016. LNCS, vol. 9817, pp. 29–44. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45507-5_3
10. Hitz, M., Kessel, T. & Pfisterer, D., 2017. Towards Sharable Application Ontologies for the Automatic Generation of UIs for Dialog based Linked Data Applications. In Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2017), pages 65–77, 2017
11. Hitzler, P. et al.: OWL 2 Web Ontology Language Primer. W3.org (2009). <http://www.w3.org/TR/2009/REC-owl2-primer-20091027/>
12. Khushraj, D., Lassila, O.: Ontological approach to generating personalized user interfaces for web services. In: The Semantic Web–ISWC 2005, pp. 916–927 (2005)
13. Kraus, A., Knapp, A., Koch, N.: Model-Driven Generation of Web Applications in UWE. In: Proceedings of 3rd International Workshop on Model-Driven Web Engineering (MDWE 2007). CEUR-WS, p. 261 (2003)

14. Limbourg, Q.: USIXML: a user interface description language supporting multiple levels of independence. In: Matera, M., Comai, S., (eds.) ICWE Workshops. Rinton Press, pp. 325–338 (2004)
15. Liu, B., Chen, H., He, W.: Deriving user interface from ontologies: a model-based approach. In: Proceedings - International Conference on Tools with Artificial Intelligence, ICTAI 2005, pp. 254–259 (2005)
16. Meixner, G., Paternò, F., Vanderdonckt, J.: Past, present, and future of model-based user interface development. *i-com*. **10**(3), 2–11 (2011)
17. Paterno, F., Santoro, C., Spano, L.D.: MARIA: a universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environment. *ACM Trans. Comput.-Hum. Inter.* **16**(4), 19 (2009)
18. Pfisterer, D., Radonjic-Simic, M., Reichwald, J.: Business model design and architecture for the internet of everything. *J. Sens. Actuator Netw.* **5**(2), 7 (2016)
19. Popp, R., et al.: Automatic generation of the behavior of a user interface from a high-level discourse model. In: Proceedings of the 42nd Annual Hawaii International Conference on System Sciences, HICSS (2009)
20. Puerta, A.R., Eriksson, H., Gennari, J.H., Musen, M.A.: Beyond data models for automated user interface generation. In: Proceedings British HCI 1994 (1994)
21. Van den Bergh, J., Luyten, K., Coninx, K.: CAP3: context-sensitive abstract user interface specification. In: Proceedings of the 3rd ACM SIGCHI Symposium on Engineering Interactive Computing Systems – EICS 2011, pp. 31–40 (2011)



Surveying Co-evolution in Modeling Ecosystems

Jürgen Ettlstorfer¹, Elisabeth Kapsammer¹, Wieland Schwinger¹,
and Johannes Schönböck²(✉)

¹ Johannes Kepler University, Linz, Austria

{juergen.ettlstorfer, elisabeth.kapsammer, wieland.schwinger}@cis.jku.at

² Upper Austrian University of Applied Sciences, Hagenberg, Austria
johannes.schoenboeck@fh-hagenberg.at

Abstract. Metamodels, defining the determinant concepts of a domain, constitute the core components in Model-Driven Engineering. Together with their depending artifacts, e.g., models and transformations, they form modeling ecosystems. To be operable, it is essential for a modeling ecosystem to be in a valid state with respect to the various interdependencies between the metamodel and its depending artifacts as well as among the depending artifacts. Consequently, in case of metamodel evolution, caused by, e.g., changing requirements, the depending artifacts have to be co-evolved accordingly to keep the system in a valid state. With respect to modeling ecosystems, special effort has to be laid to a consistent co-evolution across the different kinds of artifacts and their relationships. Although several approaches for the co-evolution of depending artifacts have been proposed, there was no special focus on an ecosystem-wide perspective of co-evolution, yet. Therefore, this paper focuses on co-evolution in modeling ecosystems by discussing the various components of a modeling ecosystem and their relationships, depicting the respective co-evolution process, proposing an evaluation framework for co-evolution, and applying this framework to current approaches. Based on this evaluation we derive lessons learned and present future research directions.

Keywords: Model-driven engineering · Evolution · Co-evolution
Modeling ecosystem

1 Introduction

Software systems have become more and more complex with a rising number of features, execution environments and platform dependencies [30, 32]. In order to cope with the ever growing complexity of software systems, the development thereof has to be conducted on a steadily increasing level of abstraction. A promising paradigm which raises the level of abstraction has been proposed by the introduction of Model-Driven Engineering (MDE) [37]. MDE promotes a shift from the “everything is an object” paradigm to an “everything is a model”

paradigm [2], putting models as the central artifacts during the complete software development life-cycle.

As the cornerstones in MDE, *metamodels* define the basic concepts and relationships in between for a certain domain. Their importance is further underlined by the fact that a diversity of different kinds of artifacts depends on a metamodel, e.g., models, *instantiating* the concepts of the metamodel [2], transformations, being comparable to compilers in high-level programming languages, operating on models that *refer* to concepts of the metamodel, [8,39], concrete syntaxes, defining how modeling concepts are rendered visually [1] or textually expressed [4], and even tools, building upon the metamodel [40]. All these artifacts together build a so-called *modeling ecosystem*, i.e., “a metamodel-centered environment whose entities are traditionally subject to distinct evolutionary pressures but cannot have independent life-cycles” [10]. In a more general perspective, a *software ecosystem* “consists of the set of software solutions that enable, support, and automate the activities and transactions by the actors in the associated social or business ecosystem and the organizations that provide these solutions” [3]. The different kinds of artifacts in a modeling ecosystem have *different kinds of relationships and dependencies* to the metamodel and among each other. In this context, it is of utmost importance that all artifacts are in a valid state, i.e., they have to satisfy the given relationships and dependencies, both, with respect to the metamodel as well as to each other in a consistent way, in order to preserve validity of the whole modeling ecosystem.

As a matter of course, metamodels are not static. In the course of evolution, e.g., caused by changing requirements, *metamodels might change*. For illustrating *metamodel evolution*, a small, but indeed realistic example is shown in Fig. 1. As one can see, class `Type` has been extracted from class `Attribute` and corresponds to attribute `Attribute.type` of `MetamodelV0`, resulting in the classes `Attribute` and `Type` of `MetamodelV1`, being connected by the new reference `Attribute.type`. As a consequence, referring to attribute `Attribute.type` is no longer consistent with `MetamodelV1`, necessitating all depending artifacts, e.g., all models that instantiate `MetamodelV0`, in the ecosystem to be *migrated* to be again valid, i.e., they have to *co-evolve* (cf. Fig. 1). Due to their different nature and since the distinct kinds of artifacts existing in an ecosystem may hold different kinds of relationships, each kind of artifact has to be treated specifically. Consequently, migration is not limited to one kind of artifact, only, but in fact has to be performed for all kinds of depending artifacts and their respective instances, entailing the risk of introducing divergence between the various migrations leading to *inconsistencies* [26]. Discovering and resolving already introduced inconsistencies is a tedious task, since they might *spread* across all artifacts in the ecosystem. Thus, comprehensive tool support is indispensable.

Although several approaches for the co-evolution of modeling artifacts have been proposed, e.g., [20,35] for models and [27,29] for transformations, they differ substantially from each other regarding their capabilities with respect to modeling ecosystems. This is since there are several aspects in co-evolution that are specific in the context of an ecosystem-wide perspective, in contrast to an

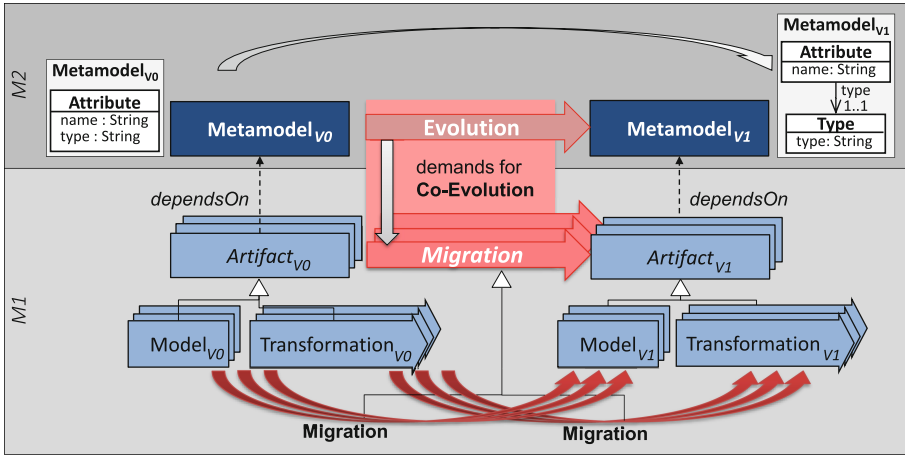


Fig. 1. Evolution and co-evolution in a modeling ecosystem.

isolated view on a single kind of artifact, only. Thus, an in-depth investigation of existing co-evolution approaches is needed to discover their adequacy in the context of a modeling ecosystem.

Therefore, in this paper four major contributions are provided: (i) the relationships between artifacts in modeling ecosystems are presented, (ii) a respective co-evolution process is depicted, which further builds the basis for (iii) an evaluation framework for the comparison of co-evolution approaches with special respect to modeling ecosystems. Finally, (iv) the evaluation framework is applied to state-of-the-art co-evolution approaches and lessons learned are drawn from the evaluation which outline current drawbacks and possible future research lines. With respect to evolving and depending artifacts, this paper focuses on metamodels as evolving artifacts and models as well as transformations as depending artifacts, respectively. This is due to the fact that most of the related research as well as existing approaches focus on such a scenario. Nevertheless, the findings of this paper as well as the general parts of the evaluation framework are also applicable to other modeling artifacts (e.g., concrete textual or graphical syntax). The work presented in this paper bases on and extends our previous work discussed in [11].

The paper is organized as follows: The next section explores modeling ecosystems and the relationships between artifacts in these ecosystems, while in Sect. 3 the according co-evolution process is depicted and the evaluation framework is presented as well as applied to current approaches. In Sect. 4 we discuss lessons learned, while Sect. 5 presents related work. Finally, Sect. 6 concludes the paper by summarizing and presenting future research directions.

2 Evolving Modeling Ecosystem

In the following, we discuss artifacts in a modeling ecosystem and specially focus on the relationships between the involved artifacts.

2.1 Modeling Ecosystem

In the course of MDE, a model is tightly interwoven with a diversity of different artifacts that may depend on each other, thereby building a modeling ecosystem [10]. With respect to evolution of modeling ecosystems, *metamodels* take an outstanding position among the artifacts. This is since the metamodel can be seen as a central artifact, defining the determinant concepts of a domain, which are utilized in other artifacts, that therefore *depend on* the metamodel. The metamodel may be expressed in terms of a *meta-metamodel* (e.g., MOF or Ecore), consequently, a metamodel itself needs to conform to a meta-metamodel as shown in Fig. 2. As further depicted, we differentiate the artifacts in metamodels and their depending artifacts, being again specialized into different kinds of artifacts, like models, transformations, concrete syntax, tools, and other artifacts (cf. Fig. 2).

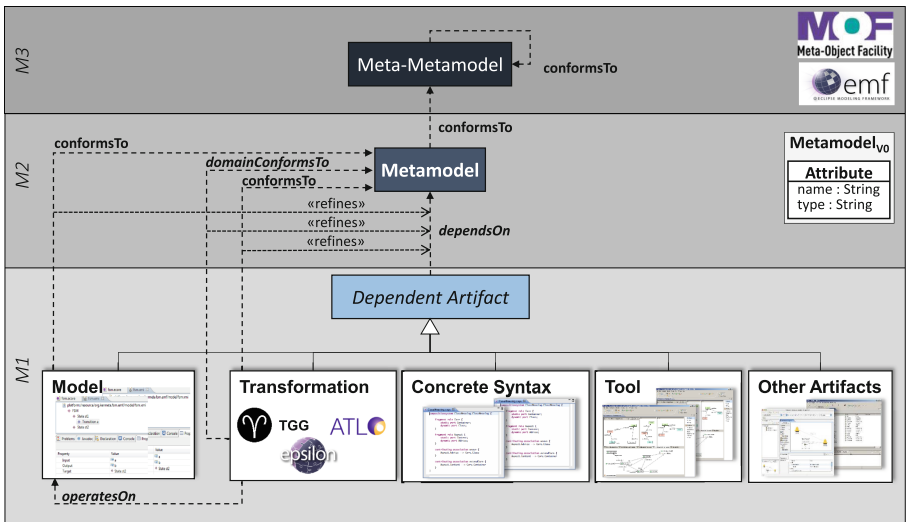


Fig. 2. Artifacts and relationships in a modeling ecosystem.

The general *dependsOn* relationship between a metamodel and its depending artifacts may be refined for each kind of artifact within the modeling ecosystem. For example, in case of models it is required that they *conform to* their respective metamodel. In this respect, this conformance relationship is a refinement of the more general *dependsOn* relationship. Regarding transformations,

they need to *conform to* their transformation metamodel defining the syntactic constraints of transformation definitions, but additionally, have dependencies on the *source domain* and *target domain* employed in the transformation definition. Furthermore, other depending artifacts, like concrete syntax specifications and tools assisting the modeler in diverse tasks, might exhibit specific dependencies on the metamodel.

Just like any other piece of software, artifacts in a modeling ecosystem are subject to constant *evolution*, e.g., due to changing requirements [19]. Each artifact might undergo changes which might impact depending artifacts. As one might see, an evolution of a *model metamodel*, i.e., *ModelMM*, impacts all conforming models, i.e., instances of this ModelMM, as well as all transformations, if the source or target domain conforms to a metamodel under evolution. Additionally, since most transformation languages themselves rely on metamodels, changes to these *transformation metamodels*, i.e., *TrafoMM*, might impact the conformance of all depending transformations. In this context, we do not explicitly differ between graph transformations, bi-directional transformations, or plain text-based transformations, since the conceptual work presented in this paper is applicable to all of them. In general, the kind of dependency determines the impact of a metamodel change on the depending artifact, thus, the impacts are specific for each kind of depending artifact.

To keep the modeling ecosystem in a consistent state it is vital that all artifacts hold valid relationships (i) to their respective metamodels as well as (ii) among each other. Due to their dependencies the co-evolution of artifacts may influence each other, e.g., when changes are applied on one artifact in the ecosystem it might induce an inconsistency in the ecosystem, necessitating a corresponding migration of the affected depending artifacts. The inter-dependencies between the artifacts demand for an ecosystem-wide perspective and, thus, an ecosystem-wide co-evolution across all artifacts to re-establish the consistency and preserve the operability of the modeling ecosystem is needed.

2.2 Relationships in a Modeling Ecosystem

In order to understand the inter-dependencies between the different kinds of artifacts responsible for impacts of evolution, in the following, the relationships in a modeling ecosystem are identified and discussed, based on studied literature (e.g., [2,9,29,36]) as well as our own experiences and findings. Therefore, the most generic relationship is presented first, while refinements for models and transformations are subsequently discussed.

- **dependsOn.** Since a metamodel defines the abstract syntax all artifacts in the modeling ecosystem *depend on* a metamodel in general. However, this kind of dependency has to be specialized for each kind of artifact, since the *dependsOn* relationship does not impose any specific constraints on the validity of this relationship between the metamodel and other depending artifacts. This kind of relationship might not have a formalized representation, but can relate

arbitrary artifacts to the metamodel in general, as for example GMF models¹ [9]. Artifacts connected via this relationship are affected by the metamodel definition in an arbitrary way. Consequently, the *dependsOn* relationship is the most generic dependency relationship between artifacts in the ecosystem and is *refined* by other dependency relationships.

- **conformsTo**. The most prominent refinement of the *dependsOn* relationship in a modeling ecosystem holds between a model and its according metamodel. More strictly, a model *conforms* to a metamodel, if only concepts that are defined in the metamodel are used, according to the rules and constraints specified in the metamodel [38], e.g., multiplicity constraints. Furthermore, metamodels themselves have to *conform* to their meta-metamodels, e.g., MOF [33] or its open-source implementation Ecore as basis of the Eclipse Modeling Framework². Moreover and as already highlighted in [2], model transformations themselves can be seen as models that *conform* to their respective transformation metamodel, e.g., the ATL metamodel [22]. Thus, an evolution of a metamodel has impact on the *conformsTo* relationship of all its depending artifacts, which in response have to co-evolve to re-establish a broken *conformsTo* relationship. Additionally, artifacts taking part in this kind of relationship typically strongly drive the definition of the metamodel.

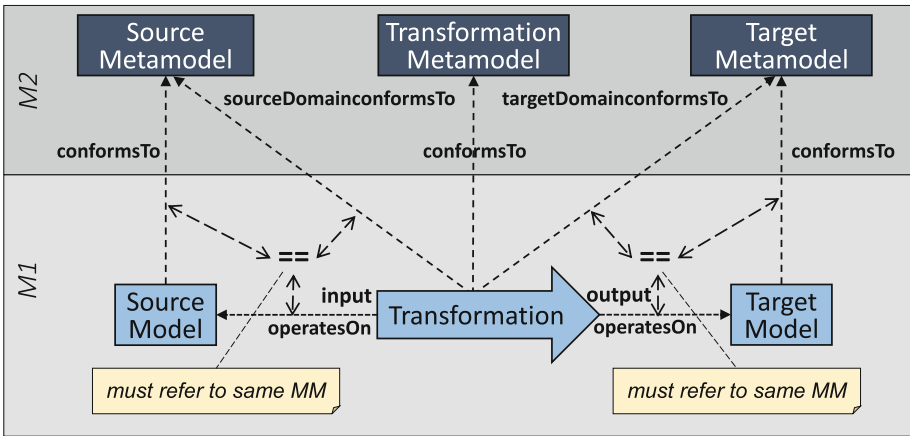


Fig. 3. Detailed relationships for transformations.

- **sourceDomainConformsTo** and **targetDomainConformsTo**. Playing a vital role in MDE, transformations, comparable to compilers in high-level programming languages [23], take source models conforming to a source metamodel as input and generate target models conforming to a target metamodel as output (cf. Fig. 3), thus, specializing *domainConformsTo* in (cf. Fig. 2). A

¹ <https://eclipse.org/modeling/gmp/>.

² <https://eclipse.org/modeling/emf/>.

transformation itself has two different kinds of relationships to the source and target metamodel, respectively. First, the *sourceDomainConformsTo* relationship states that in the source domain of the transformation definition concepts from the source metamodel are permitted, only. In contrast, the *targetDomainConformsTo* relationships only holds, if in the target domain of the transformation definition concepts from the target metamodel are used, only. In this context, the domain conformances can be seen as specifications the input and output has to conform to.

Although in [29] the term *domainConformsTo* for the relationship between the source metamodel and the source domain of a transformation has been introduced (and analogously the term *coDomainConformsTo* for the target representatives), we stick to the terms proposed in this paper – *sourceDomainConformsTo* and *targetDomainConformsTo* – being more intuitive and precise, since they clearly explicate the respective source or target roles of the involved metamodels.

Regarding an evolution of a metamodel involved in a transformation specification as source or target domain, its impact is determined by both, the role of the metamodel as source or target and the actual models the transformation operates on, since impacts might not only reveal at specification time, but also during execution.

- **operatesOn.** Having model transformations as part of the modeling ecosystem, they are eventually executed on models, thus, they *operate on* them, i.e., a transformation is executed processing a specific model. An important aspect for this relationship is, that it has to be ensured that the metamodel of the source model is the same metamodel as for the *sourceDomainConformsTo* relationship of the transformation, i.e., the transformation operates on a model which is conform to the source domain of the transformation. An evolution of a metamodel affects the *operatesOn* relationship, since the models as well as the transformations depending on the metamodel are affected, having a transitive effect on the *operatesOn* relationship. Thus, by co-evolving the depending artifacts, the *operatesOn* relationship is re-established as a result thereof. It is of utmost importance, however, that the co-evolution of models and transformations is performed consistently, i.e., following the same strategy, to successfully re-establish this relationship.

In summary, one might see that modeling ecosystems not only comprise different kinds of artifacts, but also different kinds of relationships between the artifacts and the metamodel as well as between the artifacts themselves, which are affected by an evolution of the metamodel. Consequently, an ecosystem-wide perspective on evolution and co-evolution is necessary to keep the system in a valid state and to maintain the operability of the entire modeling ecosystem.

3 Aspects of Co-evolution

In the following, aspects of co-evolution with respect to modeling ecosystems are discussed. Therefore, we present (i) the overall co-evolution process (cf. Sect. 3.1)

which is then used to (ii) propose a set of criteria for a corresponding evaluation framework (cf. Sect. 3.2). This evaluation framework and its criteria serve (iii) for an evaluation of existing approaches addressing co-evolution presented in Sect. 3.3.

3.1 Co-evolution Process

In the context of modeling ecosystems, as shown in Fig. 4, the process of co-evolution spans over four phases. Supposing the modeling ecosystem is in a consistent state V_0 , in line with [17], evolution of the metamodel triggers the following phases: (i) *change detection*, (ii) *propagation specification and execution*, (iii) *impact analysis*, and optionally (iv) *validation* (not depicted in Fig. 4).

Thereby, *change detection*, essentially determining the succeeding phases, addresses the identification of actual changes applied during metamodel evolution. It has to be noted, that this phase operates on the metamodel, only, whereas subsequent phases have to be performed on each kind of artifact. Most often, specific tools such as EMF Refactor³ or EMF Compare⁴ may be applied in order to conduct the evolution or to detect changes to the metamodel.

Based on the results of the change detection phase, the actual *specification* of *propagation semantics* takes place. Often specific co-evolution languages are provided that ease this kind of specification. The specification should take place on a level, which is independent of the kind of artifact. Based on this specification, however, artifact dependent co-evolution scripts should be derived in order to make use of existing engines to actually *execute* the *propagation* semantics to accomplish co-evolution.

In between the specification and the execution of the propagation semantics, the *impact analysis* determines, which impacts the previously specified propagation semantics has on diverse kinds of depending artifacts. In this respect, tracelinks between original and evolved versions of the metamodel as well as meta data on the affected depending artifacts may serve as a basis in order to estimate the potential problems and costs, e.g., how many elements are affected by a change or how many manual changes are required after propagation to ensure a consistent co-evolution. These results should be presented to the user to provide a solid basis for deciding, if the change is worth the effort and, consequently, the propagation semantics should be executed or even the respective change to the metamodel should be set back.

Finally, an optional *validation* of the migrated artifacts may follow, checking whether the modeling ecosystem is again in a consistent state in its migrated version V_1 .

3.2 Evaluation Framework

In the following, specific aspects of co-evolution in the context of modeling ecosystems are discussed based on the co-evolution process presented before.

³ <https://www.eclipse.org/emf-refactor/>.

⁴ <http://www.eclipse.org/emf/compare>.

In form of an evaluation framework these aspects will serve as criteria for evaluating and comparing existing co-evolution approaches (cf. Sect. 3.3). The criteria have been derived top-down from the modeling ecosystem co-evolution process (cf. Fig. 4) as well as in bottom-up manner by considering related criteria that have been discussed in literature like [21, 34]. The evaluation framework may easily be extended for other kinds of artifacts. In the scope of this paper, however, it is elaborated in detail for models as well as transformations.

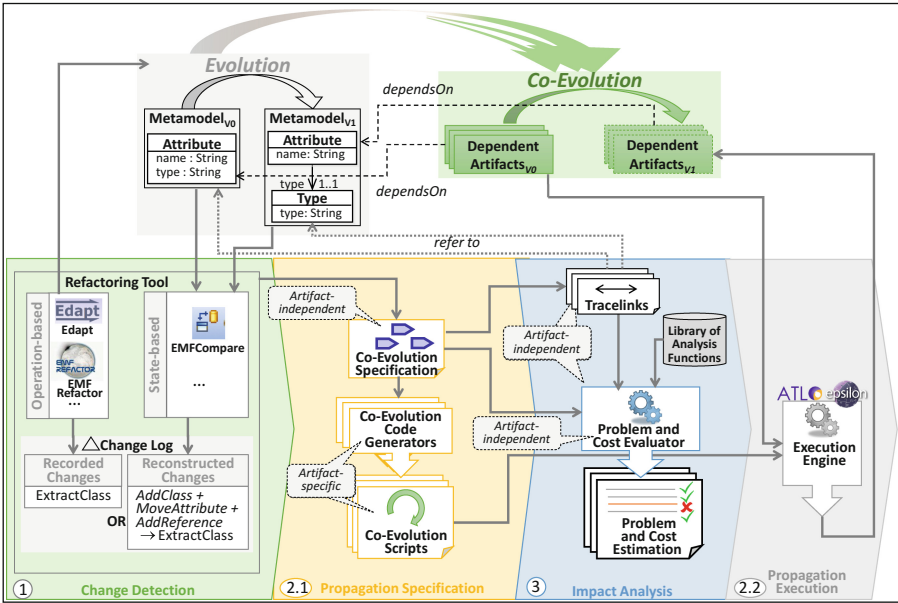


Fig. 4. Co-evolution process of a modeling ecosystem.

General Criteria. At the beginning, the first set of criteria covers general characteristics of co-evolution approaches with respect to modeling ecosystems.

- **Evolving Artifact.** This criterion investigates which artifacts are considered for evolution by a particular approach. Specifically, it indicates which meta-models are considered, acting as metamodel for either models, i.e., *ModelMM*, or transformations, i.e., *TrafoMM*. In case of a *ModelMM*, it identifies further, whether it acts as *source metamodel* or *target metamodel* of a transformation. Finally, also support for *other* evolving artifacts is investigated.
- **Depending Artifact.** As already discussed in Sect. 2.2 the kind of dependency is specific for each kind of artifact. Depending artifacts that are explicitly focused in this evaluation comprise *models* and *transformations* as well as *other* artifacts, that are affected by changes, e.g., concrete textual or graphical syntax.

- **Automatic Monitoring.** In order to inform the user about changes, it would be advantageous for maintaining or re-establishing consistency by automatically monitoring the modeling ecosystem. In this respect this criteria gives indication whether or not an approach monitors any artifact in the ecosystem to alert the user in case of changes. co
- **Transactions.** This criterion tests whether co-evolution is considered as transaction, ensuring that artifacts are either completely migrated to their new version or set back to their initial state [16]. If supported it is indicated whether transactions may span over different *ranges* of artifacts and *phases* of the co-evolution process. Regarding the range, it is indicated whether a transaction spans over *one artifact* (i.e., ensuring that all references to the evolved metamodel within one artifact are considered), *all instances of one kind of artifact* (i.e., ensuring that all instances of a particular kind of dependent artifact are considered in their entirety), or *all instances of all kinds of artifacts* (i.e., ensuring that all instances of all kinds of dependent artifact are considered in their entirety). Considering the phases of the co-evolution process, it is investigated, to which extent the detailed phases of *change application*, *change detection*, *impact analysis*, and *change propagation* are considered within transactional support.
- **Version Management.** Since evolution yields various versions of artifacts (either as evolution branches or due to intermediate co-evolution states), this criterion identifies if and to which extent handling of different versions in the modeling ecosystem is explicitly supported. In addition to common features of version management systems, e.g., going back to a previous version of a modelling ecosystem, it should be possible to identify which artifacts depend on which version of the metamodel(s) and other artifacts, respectively.

Change Detection. The second set of criteria covers the change detection phase of the co-evolution process (cf. Fig. 4) and is employed to determine in detail how and which changes can be detected.

- **Kind of Detection.** This criterion identifies on which basis changes may be detected, either state-based or operation-based [24]. *State-based* means that changes are identified by comparing the original version and the evolved one in order to infer a set of changes applied during evolution. Alternatively, changes can be detected by recording the actually applied changes, which is termed *operation-based*. However, this scenario requires specific tool support during evolution, i.e. an editor that either provides predefined evolution operations or that tracks changes. Finally, there is the option that the changes to be considered for evolution are specified *manually* and thus explicitly stated. Finally, *hybrid* forms combining the different change detection options are possible as well.
- **Granularity of Change.** This criterion explicates the level of granularity at which changes are detected. Specifically, it is indicated whether *atomic* or *composite* changes may be detected [5]. Atomic change detection recognizes changes on the level of adding, updating, or deleting elements but does

not respect eventual inter-connections between changes, which would provide more semantics for an adequate co-evolution strategy. In contrast, *composite* change detection explicitly considers semantically connected sequences of changes, e.g., refactorings like introducing a base class or vertical partitioning (as applied in our evolution scenario in Fig. 4). It has to be noted that approaches employing a state-based identification of changes would need to infer composite changes based on differences of individual metamodel elements in the compared versions while operation-based approaches typically provide specific predefined operations for such more coarse-grained changes.

- **Target of Change.** The criterion target of change identifies whether changes are identified with respect to syntactical changes on the evolving artifact (i.e., *syntax*) or whether the *semantics* of the metamodel element is considered for the change as well. The latter would be the case if changes to the interpretation of what is expressed with a metamodel element are considered. For example changing the unit in which a value of an attribute is expressed does not have an effect on the syntax and, thus, the syntactical conformance of depending artifacts is not violated. Nevertheless, model as well as transformation co-evolution could be necessary. This is since the values of the particular model element, however, would have to be changed and transformations could be affected, e.g., in case that attribute values are interpreted and treated incorrectly, when processing the migrated models.

Propagation Specification and Execution. The third set of criteria explicates how, depending on the identified changes from the change detection phase, an according propagation semantics is specified and eventually propagated to the depending artifacts.

- **Specification Language.** In order to enable the user to specify the actual propagation semantics either (i) a specific co-evolution DSL may be provided or (ii) standard transformation or programming languages may be employed. On the one hand, a DSL reduces the specification effort but, on the other hand, probably limits the possibilities for co-evolution to a restricted set of predefined operators.
- **Propagation Strategy.** This criterion explicates if built-in, predefined strategies for metamodel changes are provided to perform co-evolution. If such strategies are provided it is further investigated if only one fixed strategy is provided or if *alternative* propagation strategies are provided. As discussed in [38] and shown in Fig. 5 more than one alternative might exist to re-establish consistency for depending artifacts. For instance, the vertical partitioning presented in our example might be resolved for models either by creating a **Type** object for every element or by creating a **Type** object only for distinct values of the corresponding attribute **Attribute.type**. Besides providing a number of alternatives, it may also be possible to allow the user to *customize* (e.g., adaptation, parametrization or overwriting) provided propagation strategies. Finally, since it is hardly possible to consider all possible scenarios upfront, a comprehensive co-evolution tool should allow

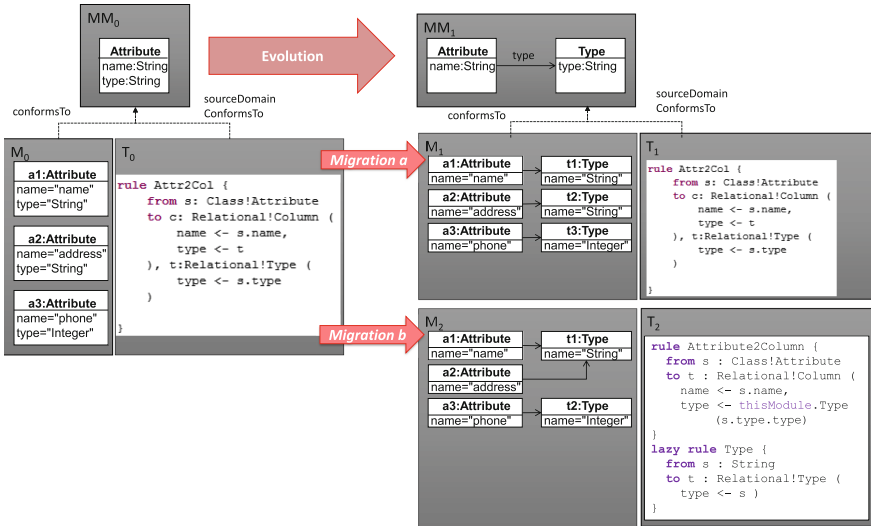


Fig. 5. Migration alternatives.

the user to incorporate new migration strategies independently of predefined ones (i.e., *user-definable*).

- **Automation.** This criterion evaluates the grade of automation provided for co-evolution with respect to propagation. Thereby, propagation can be either performed *automatic*, i.e., operating without user intervention or *semi-automatic*, i.e., incorporating user intervention during propagation. In this respect, semi-automatic approaches often only generate some stubs for co-evolution but require the user to complete the specification.
- **Consistency.** This criterion identifies to which extent the approach considers the modeling ecosystem’s consistency during propagation. Without an ecosystem-wide perspective the focus in a co-evolution process is on a single kind of depending artifact, only. However, if we consider that the modeling ecosystem comprises various kinds of depending artifacts with potentially multiply instances, the co-evolution process gets extensive. Please note, that although change propagation is performed on each depending artifact individually, it has to be in accordance across all different kinds of artifacts in order to allow for a comprehensive co-evolution (cf. Fig. 6). In this respect approaches may either ensure *intra-artifact* or *inter-artifact* consistency. Intra-artifact means that a correct co-evolution is ensured for all instances of a certain kind of artifact, only, e.g., it ensures that all models are co-evolved in the same manner. However, potential relationships to transformations are not considered and thus a different co-evolution strategy might be employed for transformations leading to an inconsistent state within the modeling ecosystem. In contrast, inter-artifact consistency means that the same propagation semantics should be ensured for all kinds of depending artifacts within a mod-

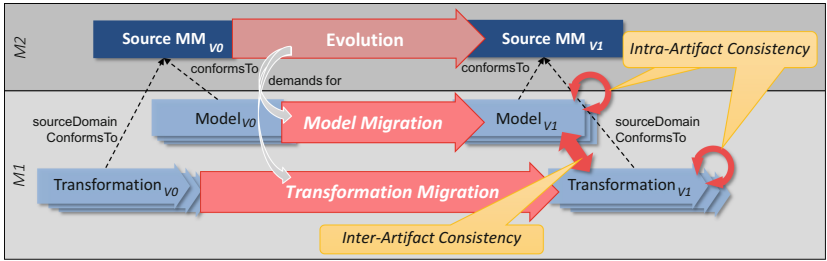


Fig. 6. Consistency in co-evolution in context of a modeling ecosystem.

eling ecosystem, e.g., by deriving artifact specific co-evolution scripts on the basis of an artifact independent co-evolution DSL, as envisioned in Fig. 4.

Impact Analysis. This set of criteria determines if and at which level of detail impacts are identified.

- **Explicated Analysis.** This criterion identifies to which extent the impact analysis is explicated with respect to the different kinds of dependency relationships. It has to be noted that this criterion is dependent on the kind of artifact supported by the approach, since each kind yields different specific dependency relationships. Thus this criterion identifies explicated analysis for *conforms-to* in case of models, *source-domain-conforms-to* and *target-domain-conforms-to* in case of transformations, and finally, on *other* relationships for other kinds of depending artifacts in the ecosystem.
- **Level.** This criterion identifies at which level the impact is analyzed. Thereby, the impact analysis can be either performed on *type level*, i.e., an analysis of potential impacts on models or transformations of metamodel changes, or on *instance level*, i.e., revealing the impacts on the actual instances in the ecosystem that are affected.
- **Granularity.** Impacts of metamodel changes on the artifacts may be detected on different level of granularity which again is dependent on the kind of depending artifact supported. In case of models this criterion spans from *feature level*, i.e., fine-grained, over *class level* to *package level*, i.e., the more coarse-grained level. In case of transformations, granularity spans over the levels of *bindings*, *rules*, and *modules*. Furthermore, if the transformation employs OCL for querying model elements, impacts may even be detected on *OCL level*.
- **Impact Severeness Scoring.** This criterion identifies whether the impact analysis yields information on the severeness of the changes' impact on the dependent artifacts. Such impact severeness scoring would allow users to distinguish changes causing costly migration effects on the modeling ecosystem (e.g., a large number of affected depending artifacts) from changes having

little or no effect on migrating the depending artifacts (e.g., so-called non-breaking changes). This would allow the user's reflecting on the changes to the metamodel, possibly reverting changes and turning to other changes implying less severe implications on the depending artifacts.

- **Processing Impacts.** This criterion identifies whether, in case of executable artifacts like transformations, impacts that only occur at processing time (i.e., running the transformation logic) are considered. An example of such an impact is changing a feature from mandatory to optional. This might cause transformations that assume the existence of this feature to operate incorrectly, if the value for this feature is not set, or even to break at run-time similarly to null-pointer exceptions in programming languages.

Validation. Related to the final and optional phase of validation, this set of criteria is employed to determine if and to which extent the performed change propagation is validated to ensure a consistent ecosystem.

- **Type of Validation.** This criterion explicates on the type of validation offered after the propagation is performed. Thereby, the propagation might be validated *syntactically*, e.g., for models conformance may be validated with EMF, while for transformations the syntactical correctness may be validated with the compiler specific for the transformation language. Additionally, the *semantical* correctness may be validated, e.g., with regression testing [12].
- **Range of Validation.** This criterion gives indication whether validation ranges over *one instance of an artifact*, *all instances of one kind of artifact*, or *all instances of all kinds of artifact*.

Based on the presented evaluation framework in the next section contemporary co-evolution approaches will be evaluated.

3.3 Evaluation

To give an overview on the current state of the art with respect to the evolution and co-evolution of a modeling ecosystem, in the following eleven contemporary approaches have been selected and evaluated according to the evaluation framework presented in the previous section. Approaches selected for this evaluation either directly support co-evolution or provide a language or framework for defining and performing co-evolution on models, transformations, or other depending artifacts. Reviewing the literature dealing with model-driven engineering topics in terms of proceedings of conferences and workshops as well as journals, we identified eleven approaches that provide for such support. Work that describes application of proposed approaches, only, however, has not been included in this selection. Investigating these approaches yields the results as summarized in Table 1 and discussed in the following.

Table 1. Evaluation of co-evolution approaches (based on [11]).

			Herrmannsdoerfer et al., 2009	Rose et al., 2010a	Gruschko, 2007	Cicchetti et al., 2009	Garcés et al., 2009	Wachsmuth, 2007	Garcés et al., 2013	García et al., 2013	Kruse, 2011	Di Ruscio et al., 2012	Kusel et al., 2015	
General Criteria	Evolving Artifact	Model Metamodel	✓	✓	✓	✓	✓	✓	✗	✗	✗	✓	✓	
		Transformation Metamodel	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	
		Source Metamodel	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
		Target Metamodel	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	✓	✓	✓	✓	✗	✗
		Other	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	Depending Artifact	Models	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✓	✓
		Transformations	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓
		Other	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗
	Automatic Monitoring		✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	Transactions	Range	One Artifact	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
			One Kind of Artifact	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
			All Kinds of Artifacts	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
		Phases	Change Application	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Change Detection			✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Impact Analysis			✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Change Propagation	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗		
Version management		✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	
Change Detection	Kind of Detection	State-Based	-	-	✓	✓	✓	-	✓	✓	-	-	-	
		Operation-Based	✓	-	-	-	-	✓	-	-	✓	-	✓	
		Manual	-	✓	-	-	-	-	-	-	-	✓	-	
		Hybrid	-	-	-	-	-	-	-	-	-	-	-	
	Granularity of Change	Atomic	✓	n.a.	✓	✓	✓	✓	✓	✓	✓	✓	n.a.	✓
Composite		✓	n.a.	✗	✓	✓	✓	✓	✓	✓	✓	n.a.	✓	
Syntax		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Target of Change	Semantics	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	
	DSL	✓	✓	-	-	-	-	-	-	-	✓	✓	-	
Propagation	Specification Language	Other	-	-	✓	✓	✓	✓	✓	✓	✓	-	-	
		Predefined	✗	n.a.	✗	✗	✗	✗	✗	✗	✗	✗	n.a.	✓
	Strategies	Alternatives	✓	n.a.	✗	✓	✓	✓	✓	✓	✓	✓	n.a.	✓
		Customizable	✓	n.a.	✗	✓	✓	✓	✓	✓	✓	✓	n.a.	✓
	Automation	User-Definable	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓
		Automatic	✓	✓	✓	✓	✓	✓	✓	-	-	-	✓	✓
	Consistence	Semi-Automatic	-	-	-	-	-	-	✓	✓	✓	-	-	
		Intra-Artifact	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
Impact Analysis	Explicated Analysis	Inter-Artifact	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	
		ConformsTo	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	
		SourceDomainConformsTo	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	
		TargetDomainConformsTo	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	
	Level	Other	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	
		Type-Level	✓	✗	✓	✓	✗	✗	✓	✓	✓	✓	✓	
	Granularity	Instance-Level	Instance-Level	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
			Models	Feature-Level	✓	✗	✓	✓	✗	✗	n.a.	n.a.	n.a.	✗
			Class-Level	✓	✗	✓	✓	✗	✗	n.a.	n.a.	n.a.	✗	
		Trans-formations	Package-Level	✓	✗	✓	✓	✗	✗	n.a.	n.a.	n.a.	✗	
OCL-Level			n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	✗	✓	✓	✗	✓	
Rule-Level			n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	✗	✓	✓	✗	✓	
Module-Level			n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	✗	✓	✓	✗	✓	
Other	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	✗			
Impact Severness Scoring		✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗		
Processing Impacts		✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗		
Validation	Type of Validation	Syntactic	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	
		Semantic	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	
	Range of Validation	One Artifact	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	
		One Kind of Artifact	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	
All Kinds of Artifacts	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓		

Legend: ✓ yes ✗ no ~ partially n.a. not applicable

General Criteria. Out of the evaluated eleven approaches, most approaches thereby focus on a specific kind of depending artifact, only. Six approaches, namely [6, 13, 17, 20, 35, 41], aim at model co-evolution whereas three approaches, [14, 15, 25], focus solely at co-evolution of transformations. Two of the investigated approaches [10, 26], however, take up a more ecosystem-wide perspective by supporting, both, models as well as transformations. Thereby, [10] serves as a framework for co-evolution of an even wider range of modeling artifacts, e.g., GMF models.

Of all five approaches dealing with transformation co-evolution, three approaches [10, 14, 15] consider the evolution of, both, source and target meta-models. The approach of [25] also supports co-evolution of transformations in reaction to source and target metamodel evolution, but in case of copy transformations, a specific form of a model transformation, only. Evolution of the transformation metamodel or other modeling artifacts is not considered in the surveyed approaches. A sole exception therefrom is [10], since they provide a framework instead of a concrete approach.

The automatic monitoring of the evolution of the respective supported artifacts is not supported by any of the approaches evaluated. Likewise none of the approaches supports transactions or version management.

Change Detection. Regarding change detection, five approaches rely on a state-based changed detection ([6, 13–15, 17]), whereas four approaches employ operation-based change detection ([20, 25, 26, 41]). Only two approaches do not provide any means for automatic change detection and base on manual change specifications ([10, 35]).

Almost all approaches support, both, atomic as well as composite changes. The only exception is [17], which does not support composite changes. Interestingly, [26], instead of dealing with composite changes by predefining a fixed set, considers composite changes as pure composition from atomic changes, thus being open to allowing for new, self-defined composites. Since [10, 35] rely on manual detection, only, this criterion is not applicable for those approaches.

As expected, all of the examined approaches focus on syntax as target of changes. Semantic changes are currently not considered.

Propagation. Concerning the specification of changes, only four approaches rely on a dedicated co-evolution DSL. The other approaches employ standard transformation or programming languages, aggravating the specification of the co-evolution semantics.

Regarding the actual propagation of changes, all but two approaches [10, 35] support a predefined migration strategy. Only [26], however, allows for the specification of alternative propagation semantics and, thus, enables users to specify their intent more precisely. All approaches except [17] allow for customization of the migration process. Only five out of eleven approaches, however, allow for user-definable propagation of changes. Only the approach of [26] provides dedi-

cated means for easing the specification within the DSL. The other approaches require dealing with the low level details of the co-evolution approach.

Eight approaches allow for an automatic migration process [6, 10, 13, 15, 17, 26, 35, 41], while the other three target transformation co-evolution in a semi-automatic way [14, 15, 25], requiring user intervention during the co-evolution process. The approach presented in [14] bases on a so-called mapping model which needs to be further refined by the user. In contrast, in [15] only code stubs are generated which have to be completed during the co-evolution process by the user. A similar approach is followed in [25]. It has to be noted, that despite inducing a specification overhead for users, these approaches allow them to influence the co-evolution process according to their desired intent.

Concerning consistency, only two approaches [14, 26] ensure intra-artifact consistency. Notably, only one approach also provides inter-artifact consistency [26]. Inter-artifact consistency is achieved by providing a single DSL for evolving depending artifacts. Based on this DSL, artifact-specific co-evolution scripts are automatically derived, which execute the specified semantics and propagate the changes to a specific depending artifact, e.g., an ATL HOT is used to co-evolve transformations defined using ATL.

Impact Analysis. Considering impact analysis, two approaches provide an explicated analysis of changes with respect to the conforms-to relationship between models and metamodels [17, 26], two approaches consider the source-domain-conforms-to relationship [15, 26], and a single approach regards the target-domain-conforms-to relationship [15].

Impact analysis is performed on type level by seven approaches [7, 14, 15, 17, 20, 25, 26], while impact analysis on instance level is supported by none of the examined approaches. Results of the impact analysis, however, can not be directly employed for user intervention by any of the examined approaches. Approaches that support impact analysis on models provide analysis on feature, class, and package level [6, 17, 20, 26]. In case of transformations, the level of bindings, rules, and modules is supported by three approaches [15, 25, 26], whereas only two approach additionally detect impacts on OCL level [15, 26]. None of the examined approaches, however, is capable of detecting run-time effects before the actual execution of artifacts, e.g., by running a transformation.

Validation. Up to now, validation of the co-evolution process is just sparsely supported. Only the approach presented in [26] supports validation of migrated artifacts, both, syntactically and semantically. In this context, the approach is able to validate all instances of all supported kinds of artifacts, i.e., models and transformations [26]. For the validation the PaMoMo language is used [18].

The evaluation of these eleven contemporary co-evolution approaches yields some lessons learned that are presented in the next section.

4 Lessons Learned

After having evaluated existing co-evolution approaches with the presented evaluation framework, in the following, lessons learned drawn from the evaluation are discussed.

Need for Ecosystem-Wide Perspective. As one might see from the evaluation in Sect. 3.3, most current co-evolution approaches tackle either models or transformations as depending artifacts, but miss an ecosystem-wide perspective, spanning over several kinds of modeling artifacts and supporting an overall co-evolution process. Consequently, a systematic and consistent co-evolution is hindered due to the fact, that diverse tools or approaches have to be employed to co-evolve a complete modeling ecosystem. This may lead to an inter-artifact inconsistency, i.e., different kinds of artifacts are co-evolved differently, thus, the operability of the modeling ecosystem may be broken.

Support for Alternative Propagation Strategies Required. There may be different ways to propagate metamodel changes to depending artifacts. Therefore, co-evolution approaches should provide alternative propagation strategies as well as means for exploration and selection of appropriate ones, fitting to the intention of the concrete metamodel change. Considering an evolution scenario consisting of more than one metamodel change, this need becomes even more important, since propagation alternatives of different metamodel changes have to be combined in a reasonable way, preventing senseless or even contradictory combinations.

Increased Effort in Evolution May Reduce Effort in Co-evolution. As shown in Fig. 7, a limited set of changes reduces the specification effort during evolution. However, intra- and inter-artifact inconsistencies may be introduced. Thus, in order to resolve these inconsistencies, increased effort is needed during actual co-evolution process. This is even more aggravated if several dependent artifacts and numerous instances of a specific artifact are considered. In contrast, a more feature-rich change set might increase the effort for specifying the evolution. This overhead, however, is overhauled by a limited effort in the co-evolution phase since no inconsistencies need to be resolved.

Incorporation of Semantical Changes Needed. As discussed earlier, changes in a modeling ecosystem might be of semantical nature, thus, not affecting the syntax of a metamodel. As an example, a modeler might intrinsically assign the unit centimeter to a “height” attribute of a person. Thus, a change of centimeter to millimeter affects all depending artifacts, since the actual values in the models or potential calculations in transformations have to be adapted. A change of unit, however, can not be explicitly expressed in current co-evolution approaches, thus, hindering an automated co-evolution of depending artifacts. Consequently, means to express semantical changes are needed.

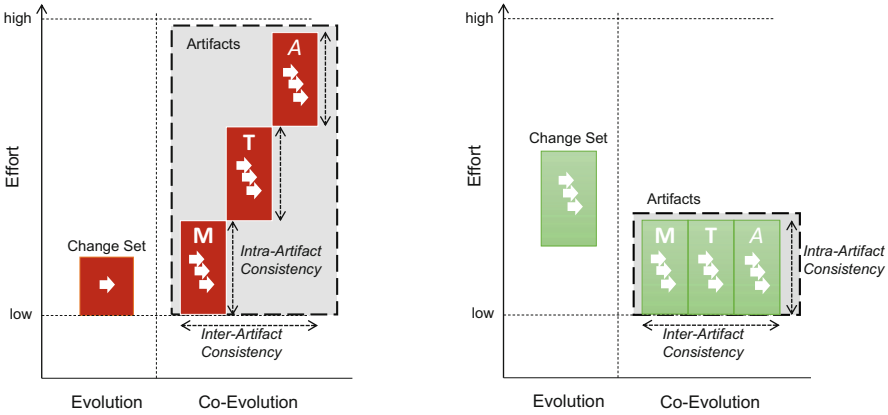


Fig. 7. Comparison of efforts for co-evolution.

Transactional Execution Beneficial for Consistency. Since transactions are currently not supported in any of the surveyed approaches, errors that occur during co-evolution have to be identified and corrected manually. In order to mitigate this burden, transactions spanning over all phases of the co-evolution process including the actual propagation of changes enable consistency of the ecosystem, since either all artifacts are co-evolved or the whole ecosystem is set back to its initial state.

Enabling Run-Time Effects Detection by Employing Code Analysis Techniques. Changes on the metamodels might impact the ecosystem not only at design-time, but also at run-time, e.g., when a transformation queries the value of an element which is no longer set. However, by employing static code analysis techniques [28] to detect errors, e.g., potential null-pointer references, before executing the transformation, run-time effects can be detected and corrected in an earlier stage, i.e., before execution. Consequently, a more sophisticated impact analysis that reveals potential run-time effects helps the evolution designer in deciding if one or more changes should be applied on the metamodel.

5 Related Work

This section discusses related work that focuses on comparing and evaluating co-evolution approaches in the area of MDE.

In a survey published in [21], the authors utilize a comprehensive criteria catalog to compare several approaches for co-evolution of models after evolution of their metamodels. In comparison to our contribution in this work, the related survey in [21] considers models as depending artifacts, only. Recent work discusses state-of-the-art techniques and approaches for co-evolution in MDE [34], thereby outlining future research challenges. In contrast to the contributions of

our paper, however, they neither pursue a concrete focus on modeling ecosystems as a whole nor provide a comparison of concrete co-evolution approaches. [35] conducts a comparison between co-evolution of models and transformations in response to evolution of metamodels. The authors discuss the differences between the migration of models and the migration of transformations and highlight the demand for a more uniform, i.e., consistent, co-evolution of both kinds of artifacts. Within that work, however, no concrete approaches are compared in contrast to the contribution of this paper. In [31] the authors propose a framework for the evolution of metamodels and co-evolution of various kinds of depending artifacts, including models and transformations. A taxonomy for metamodel evolution is proposed, which is applicable to existing approaches. An evaluation of co-evolution approaches, however, as given herein, is not part of their work.

Summarizing, this work presents a unique contribution advocating an ecosystem-wide perspective regarding co-evolution and the consideration of different kinds of depending artifacts, thereby particularly focusing on models and transformations.

6 Conclusion and Future Work

In this work, we focused on co-evolution in modeling ecosystems, emphasizing that evolution of metamodels requires an ecosystem-wide perspective with respect to co-evolution of depending artifacts. The ultimate goal in this respect is to re-establish consistency not only intra-artifact-wide, but also across different kinds of depending artifacts to keep the whole system in a valid state and, thus, ensure an operable modeling ecosystem. To substantiate this concern, we discussed artifacts as well as relationships among them, typical for ecosystems in MDE, comprising metamodels in different roles as well as different kinds of depending artifacts, like models and transformations. Further, we proposed a respective co-evolution process and an evaluation framework for co-evolution in the context of modeling ecosystems, which was built upon the process. We then utilized the evaluation framework to evaluate and compare current co-evolution approaches. Thereof, we derived lessons learned which, finally, lead us to future research directions.

As the findings of our evaluation demonstrate, future work may focus on enhancements to co-evolution approaches with respect to an ecosystem-wide perspective. This would include, e.g., the support of more than one kind of depending artifact, the provision of alternative propagation strategies, the possibility to shift propagation semantics to changes applied during evolution, as well as the incorporation of transaction mechanisms to cope with problems during the overall co-evolution process and to enable the user to set back to a previous state in case of, e.g., not acceptable costs detected during impact analysis. Further important issues would be a support for detection and incorporation of semantic changes as well as strengthening impact analysis by employing static code analysis techniques.

References

1. Baar, T.: Correctly defined concrete syntax for visual modeling languages. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) *MODELS 2006*. LNCS, vol. 4199, pp. 111–125. Springer, Heidelberg (2006). https://doi.org/10.1007/11880240_9
2. Bézivin, J.: On the unification power of models. *SoSym* **4**(2), 171–188 (2005)
3. Bosch, J.: From software product lines to software ecosystems. In: *Proceedings of the 13th International Software Product Line Conference, SPLC 2009*, pp. 111–119. Carnegie Mellon University, Pittsburgh, PA, USA (2009). <http://dl.acm.org/citation.cfm?id=1753235.1753251>
4. Brambilla, M., Cabot, J., Wimmer, M.: *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers, Los Altos (2012)
5. Brosch, P., et al.: An example is worth a thousand words: composite operation modeling by-example. In: Schürr, A., Selic, B. (eds.) *MODELS 2009*. LNCS, vol. 5795, pp. 271–285. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04425-0_20
6. Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A.: Automating co-evolution in model-driven engineering. In: *Proceedings of the EDOC 2008*, pp. 222–231 (2008)
7. Cicchetti, A., Di Ruscio, D., Pierantonio, A.: Managing dependent changes in coupled evolution. In: Paige, R.F. (ed.) *ICMT 2009*. LNCS, vol. 5563, pp. 35–51. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02408-5_4
8. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. *IBM Syst. J.* **45**(3), 621–645 (2006). <https://doi.org/10.1147/sj.453.0621>
9. Di Ruscio, D., Iovino, L., Pierantonio, A.: What is needed for managing co-evolution in MDE? In: *Proceedings of the International Workshop on Model Comparison in Practice*, pp. 30–38. ACM (2011)
10. Di Ruscio, D., Iovino, L., Pierantonio, A.: Evolutionary togetherness: how to manage coupled evolution in metamodeling ecosystems. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) *ICGT 2012*. LNCS, vol. 7562, pp. 20–37. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33654-6_2
11. Etlzstorfer, J., Kapsammer, E., Schwinger, W.: On the evolution of modeling ecosystems: an evaluation of co-evolution approaches. In: *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development, MODELSWARD, vol. 1*, pp. 90–99. SCITEPRESS (2017)
12. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Reading (1999)
13. Garcés, K., Jouault, F., Cointe, P., Bézivin, J.: Managing model adaptation by precise detection of metamodel changes. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) *ECMDA-FA 2009*. LNCS, vol. 5562, pp. 34–49. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02674-4_4
14. Garcés, K., Vara, J.M., Jouault, F., Marcos, E.: Adapting transformations to meta-model changes via external transformation composition. *SoSym* **13**, 1–18 (2013)
15. García, J., Diaz, O., Azanza, M.: Model transformation co-evolution: a semi-automatic approach. In: Czarnecki, K., Hedin, G. (eds.) *SLE 2012*. LNCS, vol. 7745, pp. 144–163. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36089-3_9
16. Gray, J., Reuter, A.: *Transaction Processing: Concepts and Techniques*. Elsevier, Amsterdam (1992)

17. Gruschko, B., Kolovos, D., Paige, R.: Towards synchronizing models with evolving metamodels. In: Proceedings of the International Workshop on Model-Driven Software Evolution (2007)
18. Guerra, E., de Lara, J., Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W.: Automated verification of model transformations based on visual contracts. *J. Autom. Softw. Eng.* **20**(1), 5–46 (2012)
19. Hebig, R., Giese, H., Stallmann, F., Seibel, A.: On the complex nature of MDE evolution. In: Moreira, A., Schätz, B., Gray, J., Vallecillo, A., Clarke, P. (eds.) *MODELS 2013*. LNCS, vol. 8107, pp. 436–453. Springer, Heidelberg (2013). <https://doi.org/10.1007/978-3-642-41533-3-27>
20. Herrmannsdoerfer, M., Benz, S., Juergens, E.: COPE - automating coupled evolution of metamodels and models. In: Drossopoulou, S. (ed.) *ECOOP 2009*. LNCS, vol. 5653, pp. 52–76. Springer, Heidelberg (2009). <https://doi.org/10.1007/978-3-642-03013-0-4>
21. Herrmannsdörfer, M., Wachsmuth, G.: Coupled evolution of software metamodels and models. In: Mens, T., Serebrenik, A., Cleve, A. (eds.) *Evolving Software Systems*, pp. 33–63. Springer, Heidelberg (2014). <https://doi.org/10.1007/978-3-642-45398-4-2>
22. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: a model transformation tool. *Sci. Comput. Program.* **72**(1–2), 31–39 (2008)
23. Kappel, G., Langer, P., Retschitzegger, W., Schwinger, W., Wimmer, M.: Model transformation by-example: a survey of the first wave. In: Düsterhöft, A., Klettke, M., Schewe, K.-D. (eds.) *Conceptual Modelling and Its Theoretical Foundations*. LNCS, vol. 7260, pp. 197–215. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28279-9_15
24. Koegel, M., Herrmannsdoerfer, M., Li, Y., Helming, J., David, J.: Comparing state- and operation-based change tracking on models. In: 2010 14th IEEE International Enterprise Distributed Object Computing Conference (EDOC), pp. 163–172. IEEE (2010)
25. Kruse, S.: On the use of operators for the co-evolution of metamodels and transformations. In: *International Workshop on Models and Evolution @ MODELS (2011)*
26. Kusel, A., Etlzstorfer, J., Kapsammer, E., Retschitzegger, W., Schwinger, W., Schönböck, J.: Consistent co-evolution of models and transformations. In: *Proceedings of Models*, pp. 116–125 (2015)
27. Levendovszky, T., Balasubramanian, D., Narayanan, A., Karsai, G.: A novel approach to semi-automated evolution of DSML model transformation. In: van den Brand, M., Gašević, D., Gray, J. (eds.) *SLE 2009*. LNCS, vol. 5969, pp. 23–41. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12107-4_4
28. Louridas, P.: Static code analysis. *IEEE Softw.* **23**(4), 58–61 (2006)
29. Méndez, D., Etien, A., Muller, A., Casallas, R.: Towards transformation migration after metamodel evolution. In: *Proceedings of International Workshop on Models and Evolution @ MODELS (2010)*
30. Mens, T.: On the complexity of software systems. *Computer* **45**(8), 79–81 (2012)
31. Meyers, B., Vangheluwe, H.: A framework for evolution of modelling languages. *Sci. Comput. Program.* **76**(12), 1223–1246 (2011)
32. Myers, C.R.: Software systems as complex networks: structure, function, and evolvability of software collaboration graphs. *Phys. Rev. E* **68**(4) (2003)
33. Object Management Group: Meta Object Facility (MOF) 2 Core Specification (2011). www.omg.org/spec/MOF/2.4.1

34. Paige, R., Matragkas, N., Rose, L.: Evolving models in model-driven engineering: state-of-the-art and future challenges. *J. Syst. Softw.* **111**, 272–280 (2016). <http://www.sciencedirect.com/science/article/pii/S0164121215001909>
35. Rose, L., Kolovos, D., Paige, R., Polack, F.: Model migration with epsilon flock. In: *Proceedings of ICMT*, pp. 184–198 (2010)
36. Rose, L., Etien, A., Méndez, D., Kolovos, D., Paige, R., Polack, F.: Comparing model-metamodel and transformation-metamodel co-evolution. In: *Proceedings of Models and Evolution Workshop* (2010)
37. Schmidt, D.: Guest editor’s introduction: model-driven engineering. *Computer* **39**(2), 25–31 (2006)
38. Schönböck, J., Kusel, A., Ettlstorfer, J., Kapsammer, E., Schwinger, W., Wimmer, M., Wischenbart, M.: CARE - a constraint-based approach for re-establishing conformance-relationships. In: *Proceedings of APCCM* (2014)
39. Sendall, S., Kozaczynski, W.: Model transformation: the heart and soul of model-driven software development. *IEEE Softw.* **20**(5), 42–45 (2003)
40. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: *EMF: Eclipse Modeling Framework*. Pearson Education, London (2009)
41. Wachsmuth, G.: Metamodel adaptation and model co-adaptation. In: Ernst, E. (ed.) *ECOOP 2007*. LNCS, vol. 4609, pp. 600–624. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73589-2_28



Functional Decomposition for Software Architecture Evolution

David Faitelson¹, Robert Heinrich²(✉), and Shmuel Tyszberowicz³

¹ Afeka Tel-Aviv Academic College of Engineering, Tel Aviv, Israel

² Karlsruhe Institute of Technology, Karlsruhe, Germany
robert.heinrich@kit.edu

³ The Academic College Tel Aviv Yaffo, Tel Aviv, Israel

Abstract. Software systems need to evolve continuously in order to avoid becoming less useful. However, repeated changes in the software may impede the inner quality of the system. Modularity is considered an important aspect of a good inner quality, and the functional decomposition is an approach that enables to achieve good modularity. Nevertheless, existing approaches for functional decomposition ignore implementation efforts, and this may cause a situation where the required changes are too costly to implement. In this paper we describe an approach to employ functional decomposition for software architecture evolution considering also the implementation efforts.

1 Introduction

Software systems must evolve over time, otherwise they progressively become less useful [1]. Once a system is released, it continuously changes, e.g. due to emerging user requirements (perfective changes), bug fixes (corrective changes), platform alterations (adaptive changes), or correction of latent faults before they become operational faults (preventive changes). Consequently, the system drifts away from its initial architecture due to the evolutionary changes.

Knowledge about the software architecture is essential for predicting maintenance efforts. Developers must produce software that can be changed without the risk of degrading its architecture [2]. However, ensuring that the system satisfies its requirements while keeping the integrity of its architecture (i.e. keeping the requirements and the architecture synchronized) is a challenging task [3].

An important aspect of a good software architecture is its modularity. A system may initially consist of highly cohesive subsystems with low coupling between them. However, as more functionality is added to the system, their coupling tends to increase and their cohesion decreases. Thus the system becomes less understandable for developers, resulting in decreased quality and a system that is more difficult to maintain.

In a previous work [4] we have described a technique for systematically decomposing a system into subsystems. Each time the system needs to change we can use this technique to find an ideal functional decomposition. However, if

we ignore the implementation efforts, it might be that the implementation cost will be too high. Therefore, we must assess implications of the suggested decompositions and find a compromise that balances both the functional modularity of the system and the implementation effort. This, however, is one of the major challenges in software evolution [5].

There are two views on software evolution that need to be considered: the what and why perspectives versus the how perspective [5]. The former perspectives study the nature of the software evolution phenomenon, the latter studies pragmatic aspects, i.e. technology, methods, and tools that provide the means to control software evolution.

In this paper we describe an approach to handle evolving systems, while maintaining good modularity and keeping track of the efforts of implementing a change. This is achieved by using the KAMP approach to architecture-based change impact analysis, that provides insight into how much the architecture will be impacted in order to handle changing requirements [6].

Within this work we use a parking lot management system as a running example. This example helps us to demonstrate our ideas about using a formal model of system decomposition that takes relations as atomic units of design and how those ideas support the change impact propagation approach as used in KAMP. The parking lot has a set of parking spaces, a camera that detects license plate numbers, and an entrance gate that may open (or close) to allow cars to enter the parking lot. Cars leave the parking lot through a separate one way exit gate. In addition, it maintains a registry of authorized cars—only authorized cars may enter the parking lot.

Our contribution consists of a technique that combines functional decomposition analysis with architectural change impact analysis. This combination ensures a good balance of functional modularity concerns with implementation efforts. We illustrate the technique by showing how, given a set of new functional requirements, we can select a good subsystem decomposition while staying within the budget allocated for implementing the new requirements.

This paper significantly extends the work reported in [7]. We have added a discussion on the relational model of software systems that is used as our means of system specification, elaborating the explanation on those Alloy elements that are relevant to our approach. We expand previous explanations and provide the rationale of our approach regarding modeling the system decomposition and the visualization process. A detailed description of how to translate a subsystem decomposition diagram into a component diagram that is suitable for KAMP approach and an elaborated explanation of the KAMP approach to change impact analysis have been added.

In the following section we present a formal model of software systems. This model takes relations as atomic units of design. Next, in Sect. 3, we illustrate our functional decomposition approach to produce the initial structure of the parking lot management system and explain how we visualize the subsystem decomposition. Then, in Sect. 4, we explain how we transform the decomposition notation into a notation that the KAMP approach can understand. In

Sect. 5 we describe the KAMP approach. In Sect. 6 we describe a hypothetical set of changes to the system, the new functional decomposition that results from these changes, and the KAMP analysis of their cost. This section is the heart of the paper. It shows how we can combine KAMP functional decomposition to select a reasonable functional decomposition that can be implemented within the given budget. Section 7 describes work related to functional decomposition and architectural-based change propagation. We conclude with a summary in Sect. 8.

2 A Relational Model of Software Systems

The relational state based model is a familiar approach to the specification of software systems (see, e.g., [8–11]). Indeed, it is the underlying theory behind relational databases [12]. In essence, a relational model consists of:

- a finite collection of sets of atomic entities (basic sets in Z [13], atoms in Alloy [11],
- a finite set of state variables, each being a relation between the basic sets,
- an invariant predicate that constrains the sets and relational variables to fit the requirements of the system that we model, and
- a finite set of operation-specifications, each a predicate that defines the effect of an operation by describing the relation of the state variables before the operation begins to their value after it ends.

Since we use Alloy to specify the systems, we now briefly explain the main concepts of Alloy and the modeling style that we use. Full details can be found in [11].

The Alloy language describes constraints over relational variables that range over relations between sets of atoms. The constraints are first order predicates on the variables. These concepts are presented in a syntax similar to that of popular object oriented languages, thus making Alloy more friendly to software engineers. For instance, the (partial) Alloy model of the parking lot management system in Fig. 1 looks similar to a class model with two classes: The class *Car* with the fields *atGate*, *inside*, *authorized*, and *parked*, and the class *Space* with the *available* and *occupiedBy* fields. This model actually defines two sets of atoms (*Car* and *Space*) and six relational variables. Each relational variable of type *T* defined in a sig *S* is a relation between the set *S* and the set *T*. For example, *atGate* is a relation between *Car* and *State*, specifying in which states a car has approached the gate. Inside a signature *S* the notation $v : X \rightarrow Y$ means that *v* is a ternary relation between the sets *S*, *X*, and *Y*. For instance, the variable *parked* is a ternary relation between *Car*, *Space*, and *State*—*parked* : *Car* → *Space* → *State*. Its tuples describe in which states a car has parked at a particular space.

Alloy has no predefined notion of a software system, hence we have to adopt a style and a set of conventions for writing such models. This style is explained in detail in [11, Chap. 6]. In this style, each system variable that represents a

```

sig Car {
  atGate : set State,
  inside : set State,
  authorized : set State,
  parked : Space lone->State
}

sig Space {
  available : set State,
  occupiedBy : Car lone -> State
}

pred ParkingLot_Enter[s,s':State,c:Car]
{
  c in atGate.s
  c in authorized.s

  no atGate.s'
  inside.s' = inside.s + c

  authorized.s' = authorized.s
  parked.s' = parked.s
  occupiedBy.s' = occupiedBy.s
  available.s' = available.s
}

```

Fig. 1. An Alloy fragment of the parking lot system model. It shows the definitions of two signatures (Car and Space), six relational variables (atGate, inside, authorized, parked, available, and occupiedBy) and one system operation, ParkingLot_Enter. Note that in Alloy, predicates on separate lines are implicitly conjoined.

relation of arity n is modeled as a relational variable with an arity of $n + 1$, where the extra dimension represents the system's state at a particular moment in time, in which the variable held this particular value. The signature *State* represents the state of the system at particular moments in time. For instance, given a particular state s which represents a particular moment in time, the expression $atGate.s$ (that is, the relational composition of the relation $atGate$ with the singleton set that contains s) gives the set of cars that are waiting in front of the gate at time s . Similarly, the expression $parked.s$ is a binary relation that describes which cars are parked at which spaces at that moment of time. To illustrate, the following relations

```

atGate.s10 = c0
inside.s10 = c1, c2
parked.s10 = (c1,p1), (c2,p3)
authorized.s10 = c1, c2, c3, c4

```

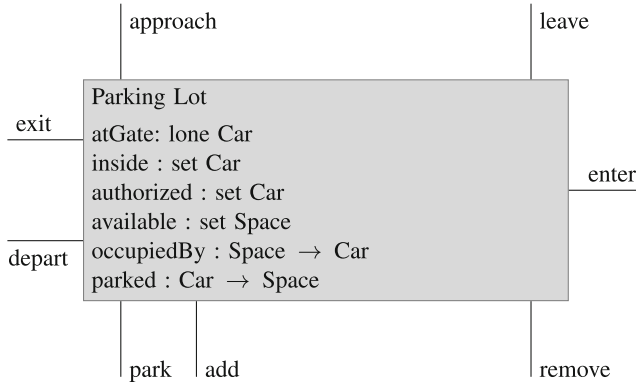



Fig. 2. A subsystem diagram that summarizes the entire parking lot system. The system is displayed as a box with its name at the top and the list of state variables inside the box. System operations appear as line segments emanating from the box, each labeled with the operation name. We have removed the *State* component from the relations to avoid cluttering the diagram. There is no need to write it here, as we show only state variables inside the box and we know that each state variable is a function of state. (Taken from [7]).

describe a moment in time (s_{10}) in which car c_0 waits in front of the gate, cars c_1 and c_2 are inside the parking lot, car c_1 is parked at location p_1 , and car c_2 is parked at location p_3 . We can also see that the cars c_1 , c_2 , c_3 , and c_4 are authorized to enter the parking lot.

In general, given a system with n state variables v_1, \dots, v_n , the state of the system at a particular moment s is defined as the values of all the state variables at moment s , that is as $v_1.s, \dots, v_n.s$.

Facts are predicates that define constraints on the basic sets and the relational variables of the model. These constraints may represent domain properties that are either facts of our universe (for example only one car may be in front of the gate at any particular moment in time) or assumptions that we consider to hold in the environment of our system (e.g., that a car occupies only one parking space). Some facts can be expressed more economically, as constraints directly in the definition of a variable. For instance, the keyword **lone** in the definition of *parked* indicates that in any particular state a car may park at most in one parking space.

The **pred** definitions of Alloy are a general purpose tool for specifying parameterized predicates. We use them to model the system’s invariant and operations. For example, the *ParkingLot_Enter* predicate models the operation that allows a car to enter the parking lot. The states s and s' represent the system state just before the operation begins (s) and immediately after it finishes (s'). We can see that the operation is enabled only when the car is at the gate and is authorized to enter. The meaning of this predicate is that when car c is authorized to enter and in front of the gate, then after it enters it is inside, no car is currently in front of the gate, and nothing else changes. All other system operations are described in a similar way.

Table 1. A summary of the operations provided by the parking lot management system [7].

Operation	Description
approach	A car is detected by the entry sensor
leave	A car at the gate drives away
enter	A car enters the parking lot
exit	A car exits the parking lot
park	A car parks at a parking space
depart	A car departs from its parking space
add	Authorize a car to enter the lot
remove	Unauthorize a car

As a system model can be long, we summarize it in a *subsystem diagram*. A subsystem diagram represents the system as a box with its state variables inside, and its operations as short line segments protruding from the edges of the box. Figure 2 is a subsystem diagram that summarizes the operations that the parking lot management system model supports. When we have a single component we do not need a diagram, a simple table would do just as well. However, in the general case this diagram is very useful to describe the structure of subsystem models, as it clearly shows which parts of the state space belong to which subsystems, and how the subsystem operations are used to support the system operations. The complete model of the parking lot management system can be found in the website [14]. A summary of the system operations used in our example is shown in Table 1.

Note that there are two important differences between the way the state variables appear in the model and in the diagram. First, in the diagram they appear as relations without using the **sig** keyword (remember that a field $x: T$ in signature A is a relation $x: A \rightarrow T$). Second, as each state variable has a final State component we omit it from the diagram. Thus, for example, the field `authorized: set State of sig Car` (i.e., the `authorized: Car \rightarrow State` relation) becomes, after removing `State`, the variable `free: set Car` that represents the set of cars that are authorized to enter the parking lot at a particular state. As we will see in the next section, the subsystem diagram is very useful for describing the structure of subsystem models.

Now we can rephrase the problem of decomposition a system in precise terms: given a system model that consists of a state space, an invariant, and a finite set of operations, how can we partition it into subsystems—each with its own state space, invariant, and operations—such that when these subsystems communicate with each other, the result refines [15] the original system?

3 Modeling the Subsystem Decomposition

We model a system decomposition as a syntactical partition of the state space; i.e., the subsystems partition the set of system state variables. The operations of each subsystem may access only the subsystem’s variables. Whereas each subsystem operation is a predicate on the entire state space, it refers only to the variables of the subsystem—leaving the other variables unspecified. This facilitates composition of subsystems by conjoining their operations (note that in Alloy predicates on separate lines are implicitly conjoined).

3.1 System Decomposition

As we have argued in a previous work [16], the traditional object-oriented approach that assumes that classes are atomic design units leads to fundamental difficulties, and it is better instead to consider individual relations (associations and attributes) as atomic units, and allocate them to the components according to how they are used by the system operations. For example, the parking lot management system has a list of authorized cars, and also keeps track of where cars are parked. Because these are two separate areas of functionality, it is reasonable to partition the system into two components: one for managing the authorization registry and the other to keep track of the location of parked cars. The concept of a car is of course essential to this application, and therefore we define a class to model it. But in which component should we put this class? In the component that is responsible for managing the list of authorized cars or in the component that keeps track of which cars park in which places? Or should we create a special component to hold the class?

It is better to break the conceptual classes into individual attributes and associations (considering them as relational state variables), and allocate them to the authorization registry and to the parked cars subsystems. As we have shown in [16], this approach also facilitates selection of ‘good’ decompositions (meaning they have low coupling and high cohesion). In our approach, we visualize the relationships between the system operations and the state variables that they read and write in such a way that we can recognize clusters of dense relationships that are weakly connected to other clusters. Each such cluster is a good candidate for a component.

In the rest of this section we illustrate how we use our approach to partition the parking lot management system into subsystems. In the previous section we have seen a relational model of this system. We now focus on revealing the structure of the major subsystems in our parking lot management system example.

3.2 Visualizing Subsystem Decompositions

A good decomposition partitions a system into loosely coupled, yet cohesive subsystems. Our approach to finding such a decomposition is to visualize the relationships between the system operations and the state variables that they use, in such a way that we can recognize clusters of dense relationships that are

weakly connected to other clusters. Each such cluster is a good candidate for a subsystem for two reasons:

1. the amount of information it shares with the rest of the system is small, thus it is protected from changes in the rest of the system and vice versa, and
2. its internal relationships are relatively dense which in most cases indicates a cohesive unit of functionality.

Table 2. An operation/relation table for the parking lot system. Each column represents one state variable and each row represents one operation. If the operation in the i -th row reads(writes) the state variable in the j -th column, the table's (i, j) entry will contain r (w). For example, the *add* operation writes only to the *authorized* variable and does not read any other variable. (Taken from [7]).

Operation	State variable (relation)			
	inside	atGate	authorized	parked
approach	r	w		
leave		w		
add			w	
remove	r		w	
enter	w	w	r	
exit	w			r
park	r			w
depart				w

To create a visualization, we record in an *operation/relation table* the relationships between the system operations and the state variables that they read and write. The information for this table is taken from the functional specification of the system. For each system operation we note which relational state variables it reads and writes. Table 2 provides the operation/relation dependencies in the parking lot system. An operation reads a variable if the operation's predicate includes only references to the variable at the current system state. An operation writes to a variable if the variable is referenced in the next system state. For example, the operation *ParkingLot.Enter* only reads the variable *authorized* as it uses this variable to check that the input car c is authorized to enter, but it does not change its value (in fact it insists that it remains the same). The operation changes the variable *inside* as it insists that c will be added to *inside* in the next system state.

We now use the table to build an undirected bipartite graph whose vertices are the system's state variables and operations. (Building the graph could be easily automated.) An edge connects operation p to variable v if and only if p uses v (either reads or writes to v). In addition, we assign a weight to each edge, depending on the nature of the connection. A read connection has the lowest weight (currently 1) and a write connection has the highest weight

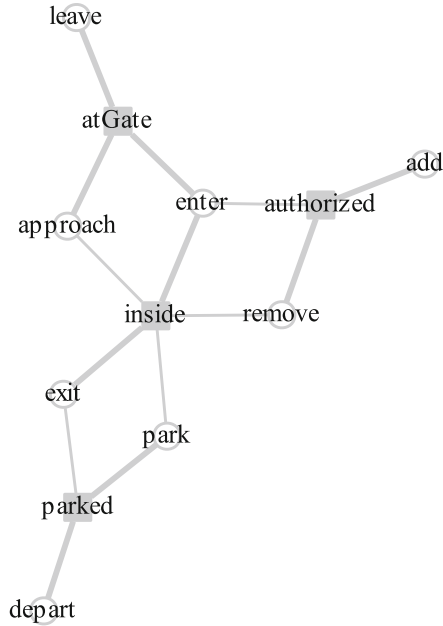


Fig. 3. An operation/relation dependency diagram of the parking lot system. Thin edges represent read relationships; thick edges represent write relationships. Circles represent operations; squares represent state variables.

(currently 2). This choice tends to cluster together data with operations that change the data, thus preferring read interfaces between clusters. A write interface has a stronger coupling than a read interface because it actively engages both subsystems whereas a read interface affects only the reader. Finally, we use NEATO [17]—a spring model based drawing algorithm [18]—to visualize the graph. The algorithm draws undirected graphs such that nodes that are close to each other in graph theoretic space (i.e. shortest path) are drawn closer to each other (see Fig. 3). In NEATO, the weight of the edge, affects the force of the spring that connects the edge’s nodes. A higher weight causes the nodes to be placed closer to each other. This visualization can now be used in three ways:

1. to suggest low dependency partitions,
2. to evaluate partitions that are dictated by non-functional constraints, and
3. to explore changes to the system model that reduce the dependencies between areas that we consider as good subsystem candidates.

Figure 4 shows a partition based on the visualized graph, and it illustrates the first usage. We have decomposed the parking lot management system into three subsystems: *Parking* keeps track of which cars park in which parking space, *Authorized* is responsible for managing the authorized list, and *Gate* manages the entry of cars to the parking lot and their exit from the parking lot. The other two are illustrated in the following sections.

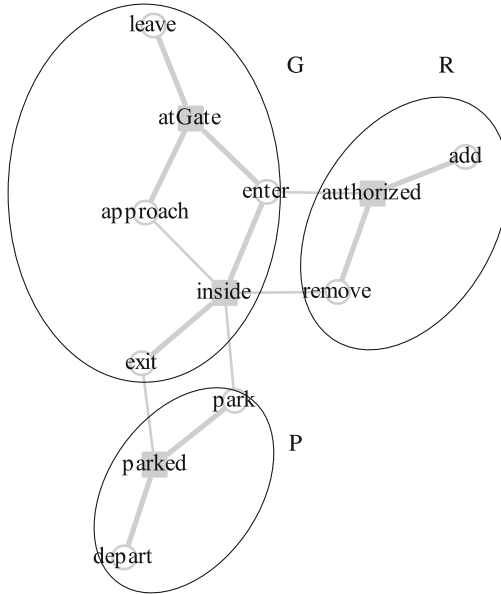


Fig. 4. An operation/relation dependency diagram of the parking lot system with a suggested partition. Each subsystem candidate is enclosed in an ellipse. The edges that cross the partitions (when they exist) are few and weak. This partition corresponds to the subsystem decomposition we have described earlier: one subsystem (marked as P) manages the information concerning which car parks in which space, one (R) manages the authorized list, and one (G) manages the entry and exit to and from the parking lot. (Taken from [7]).

Note that we have developed our own notation for subsystem diagrams, as we believe that diagrams such as UML component diagrams are not well suited for illustrating our notion of subsystem decomposition. It is possible to draw the subsystems using UML's component diagrams, but then we have to abuse the diagram's semantics to show which subsystem operations collaborate to support a system operation. This is because in order to connect the subsystem operations to the system operation we must use a delegation connector. But the standard UML semantics of delegation connectors is that of forwarding [19]. The information is forwarded from the system's boundary to the subsystems. In most cases, however, this is not how the operations are used in our decomposition. We would also have to further abuse the diagram's notation to show the state variables of each subsystem. UML components do not have a notion of relational state variables. While there are benefits to a standard and popular notation, we think that in this case it is better to use a different notation, one that perfectly suits our purpose, and not to risk the confusion that may arise from abusing the syntax and semantics of existing notations.

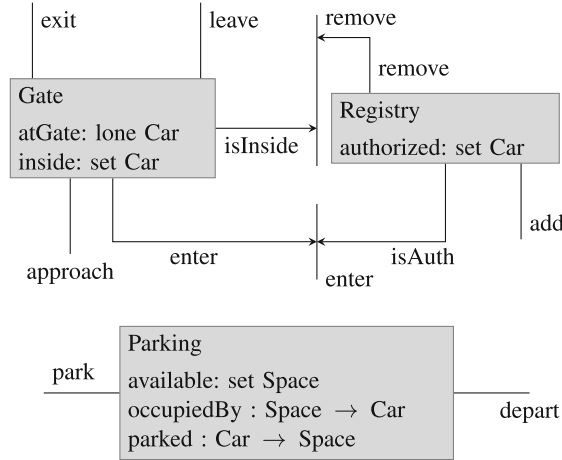


Fig. 5. A subsystem diagram of the parking lot management system. Each subsystem box contains a subset of the system’s state variables and operations. All the system operations must appear in the subsystem diagram. When a system operation is supported by a single subsystem, we draw a line on the border of the subsystem and label it with the operation’s name. When several subsystems collaborate to support a system operation, we connect the operations of each subsystem to the system operation’s line segment. Any subsystem operation whose purpose is to support a system operation must be connected to the system operation that it supports. For example, there are arrows going from the gate *enter* and registry *isAuth* operations to the system *enter* operation because *enter* requires the cooperation of the gate and the registry subsystems. (Taken from [7]).

4 Mapping Decomposition to Components

Our purpose is to apply the KAMP (Karlsruhe Architectural Maintainability Prediction Method) approach [6] to the decompositions we have found in the previous section, to assess the effects of change requests during the evolution of the system. However, KAMP requires a UML component model but our diagrams have a different structure. We must therefore transform them into compatible UML component diagrams before we can apply KAMP.

The presentation of the system when decomposing it into subsystems (Fig. 5) is different from that of a UML component model (Fig. 6), yet they are semantically identical. When decomposing a system into subsystems, we have an implicit *system* component that contains the subsystems. If we will draw a UML component diagram of the decomposition, we will see that the subsystems never use the requires symbol, as they are entirely decoupled from each other. Whatever information a subsystem requires, it is supplied by the system level component, which provides the interface to the outer world. KAMP, on the other hand, needs an explicit representation of the system component (a.k.a. manager or controller).

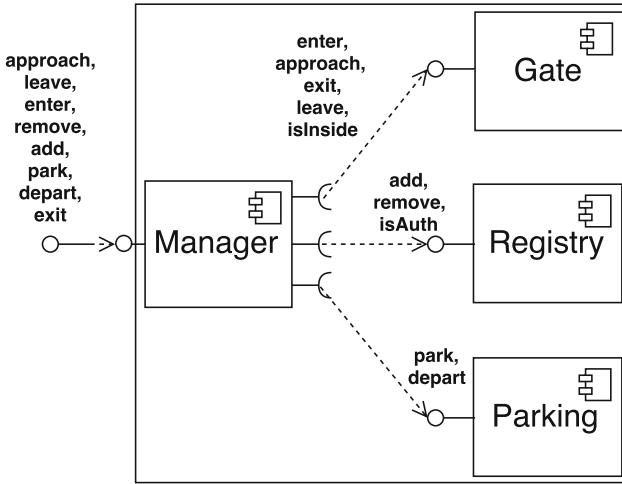


Fig. 6. Component model of the initial parking lot system. (Taken from [7]).

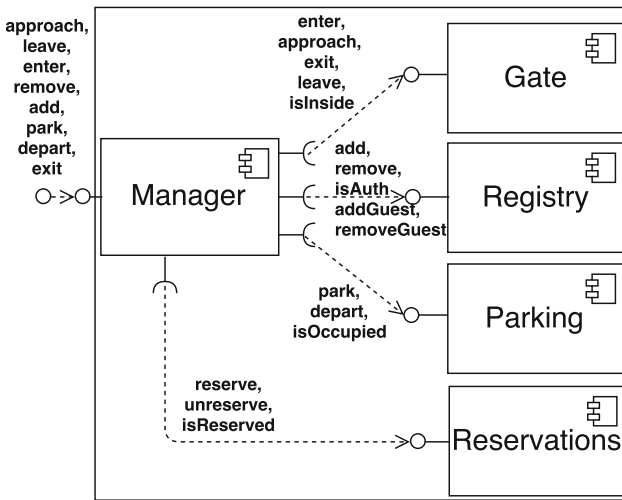


Fig. 7. Component model of the evolved parking lot management system. (Taken from [7]).

Taking the subsystem diagram, we can create a component diagram which serves as input to KAMP. Every operation that is provided to the system (Fig. 2) is mapped to a manager component, which in turn delegates the operation to the appropriate subsystems (Fig. 5). For example, the enter operation is delegated to the manager. The manager delegates it to the Registry subsystem which checks whether the car is authorized to enter the parking lot, and when it is—the manager tells the gate to open.

To translate a subsystem decomposition diagram into a component diagram that is suitable for KAMP, we have to perform the following steps:

1. Create a system component with a provides interface for all system operations.
2. In the system component, create a component for each subsystem with a provides interface for all subsystems operations.
3. In the system component, create a manager component with a provides interface for all system operations and a requires interface for all the interfaces provided by the subsystems.
4. Connect each subsystem operation to the appropriate requires interface of the manager component.
5. Use a delegation connector to connect each system operation to the corresponding manager operation.

This process is illustrated at the subsystem diagram in Fig. 5 and the corresponding component diagram in Fig. 6.

5 Component-Based Change Impact Analysis

The KAMP approach [6] aims at supporting software architects assessing the effects of change requests on technical and organizational work areas during software evolution. KAMP supports modeling the initial software architecture, named the *base architecture*, and the architecture after a certain change request has been implemented in the model, the *target architecture*. Examples of such modifications may be adding new features for guest visitors and reserved parking spaces in the parking lot management system. Then, the KAMP tooling calculates the differences between the base and the target architecture models, analyses the propagation of changes in the change request analysis phase, and generates a maintenance task list as depicted in Fig. 8. This figure reflects the structural propagation of changes as well as corresponding maintenance tasks such as test case development and execution, build and deployment configuration updates. KAMP basically comprises three contributions:

- meta-models to describe system parts and their dependencies based on an established software architecture description language [20],
- a procedure to automatically identify system parts to be changed for a given change request, and
- a procedure to automatically derive required change tasks, to simplify the identification of a change effort and thus the maintainability estimation.

KAMP has been applied in the past for analyzing change propagation in architecture models for solving performance bottlenecks, e.g. by replacing a database [21] or by splitting an interface [22]. Furthermore, there are extensions to KAMP for change impact analysis in business processes [23] and automated production systems [24]. In this paper we use KAMP to identify change tasks required to transfer an initial software architecture into an architecture restructured according to the decomposition approach described in Sect. 3.1.

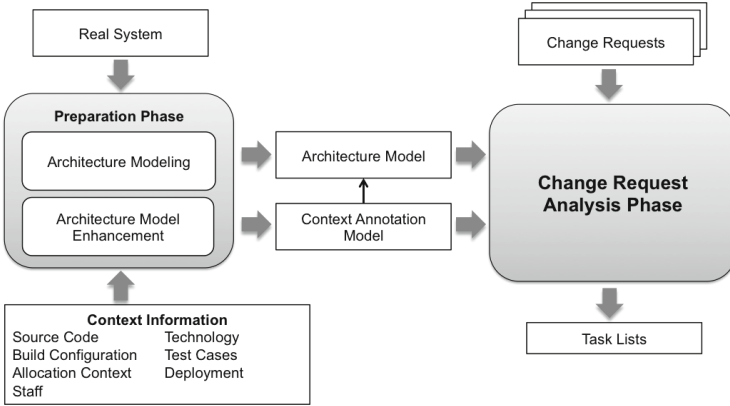


Fig. 8. Overview of the KAMP change impact analysis approach.

6 Impact Analysis Example

After the parking lot management system has been working for a while, the customer asked to extend its functionality with two major features. The first feature is the ability to reserve parking spaces in advance, and the second one is to support occasional, visiting, guests. Guests must ask for an entrance permit for a specific date. The budget provided for these changes is limited to 17 Man/Months.

We have added these two features to the original parking lot model. The system diagram of the updated parking lot management system is shown in Fig. 9. Then we have updated the operation/relation table, from which we rebuilt

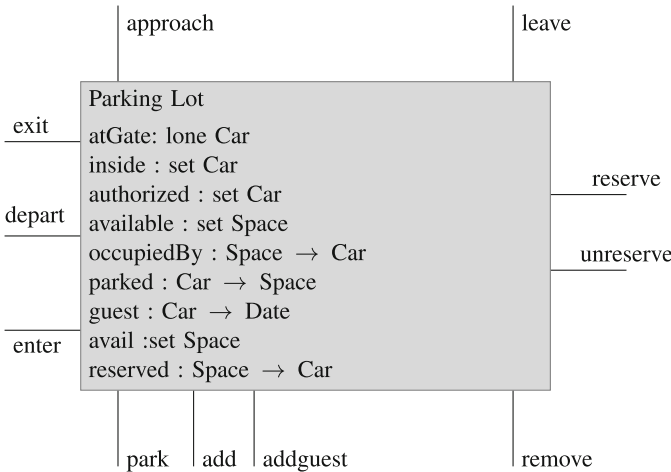


Fig. 9. A subsystem diagram that summarizes the entire updated parking lot system. We have added operations for reserving parking spaces, and state variables that record guests and reserved parking spaces.

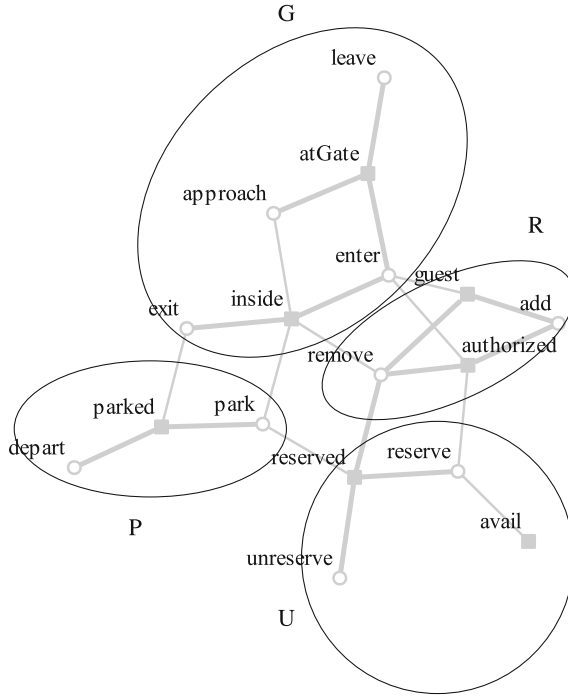


Fig. 10. The dependency diagram of the parking lot after it was extended with two features: guest visitors and reserved parking spaces. We also provide the partition that we have selected. We can see that the old structure was preserved, but that subsystem R has additional functionality to support the management of guests, and in order to to manage the reservations, a new subsystem was introduced. (Taken from [7]).

the bipartite graph which then was visualized. Figure 10 shows a partition based on the visualized graph.

After performing the decomposition, the system has changed as follows. We have added a new subsystem (*Reservations*) to manage the reserved parking spaces, and we have added to the registry subsystem a new variable (*guest*) to keep track of guests. The new variable keeps track of the dates on which guest cars may enter the parking lot. The *Gate* and *Parking* subsystems were not affected by these changes. Figure 11 presents the subsystem diagram of the evolved system after adding the two new features.

The application of KAMP comprises three phases, *preparation phase*, *analysis phase* and *interpretation phase*. In the preparation phase, an architecture model is created by using a meta-modeled architecture description languages [20]. In our running example, this model represents the initial parking lot system (base architecture), as depicted in Fig. 6. Each component in the base architecture is annotated with several test cases, a build script, and deployment information. Furthermore, another architecture model (target architecture) is created

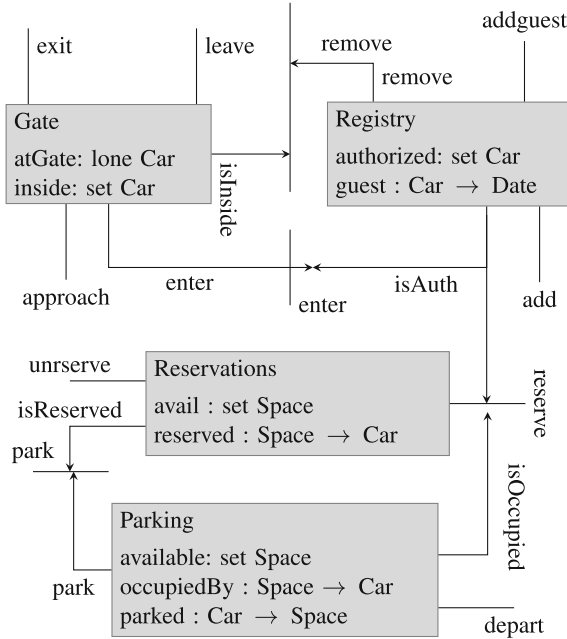


Fig. 11. Subsystem diagram of the evolved system. The registry subsystem has an additional variable (`guest`) that records which guest cars may enter at which dates. In addition there is a new subsystem that manages parking space reservations. Reserving a car requires cooperation with the registry and parking subsystems. (Taken from [7]).

to reflect the parking lot system after modification, as depicted in Fig. 7. This architecture model reflects the restructuring, if necessary, according to our approach.

In the analysis phase, KAMP automatically calculates the expected structural changes and their propagation, while transferring the base architecture into the target architecture. First the delta between the initial architectural model and the evolved architectural model is determined automatically by a model diff. Each of the deltas results in a change request to the system which is the starting point of the change propagation in KAMP.

In Step 1, changes are propagated through the system along the interfaces between components. Details on the change propagation of KAMP are described in [6]. The result of the change propagation is a task list of detailed maintenance tasks for each change request. See the middle column in Tables 3 and 4.

In Step 2, annotations to the components—i.e., test cases, build scripts, and deployment information—are applied to extend the task list for additional maintenance tasks. For the new component Reservations three test cases must be added, one for each of the methods `reserve()`, `unreserve()`, and `isReserved()`. Moreover, a build script and the deployment of the new component must be specified. The task list is extended by the corresponding tasks.

Table 3. List of maintenance tasks produced by the application of KAMP for the new reservation change request, and their associated (hypothetical) cost. (Taken from [7]).

Step	Maintenance task	Cost (Man/Month)
1	add Component Reservation	4
	add Provided Interface of Reservations	1
	add Required Interface of Manager	1
	modify Component Manager	2
	modify Provided Interface of Manager	1
	modify Provided Interface of System	1
2	add test cases for <i>reserve()</i> , <i>unreserve()</i> , and <i>isReserved()</i>	1
	Total cost	11

Table 4. List of maintenance tasks produced by the application of KAMP for the guest parking change request, and their associated (hypothetical) cost. (Taken from [7]).

Step	Maintenance task	Cost (Man/Month)
1	modify Provided Interface of Registry	1
	modify Component Registry	2
	modify Required Interface of Manager	1
	modify Component Manager	2
	modify Provided Interface of Manager	1
	modify provided Interface of System	1
2	modify test cases for <i>park()</i> , <i>reserve()</i> , and <i>depart()</i>	1
	Total cost	9

The test cases for the methods of the Parking component must be modified as the Parking component in the evolved systems requires the Reservations component. Furthermore, the Registry component in the evolved system provides functionality for guest parking. Thus, existing test cases may be modified, and new test cases must be added. The task list is extended by tasks for adding and modifying the test cases.

In the interpretation phase, change efforts are estimated by software developers based on the task list identified by KAMP. In addition, we provide hypothetical cost estimates for each task. These estimates are not a part of KAMP, however, they are important for determining if we can implement the changes within the given budget. We have determined the costs by considering the effort it takes to implement not only the functionality but also the user interface, testing, optimization etc. The estimates we provide are only used to illustrate the

approach, one may come up with different estimates, in which case the decisions may be different but the approach remains the same.

The maintenance tasks and costs for the change request of a new reservation functionality appear in Table 3. The maintenance tasks and costs for the change request of guest parking appear in Table 4. Unfortunately, the maintainability analysis reveals that the cost of decomposing the system according to the functional analysis is too high. This result demonstrates a typical situation in the engineering of complex systems. We cannot use a greedy approach to solve the problem because optimizing one aspect (functional decomposition) without regard for others (cost of implementation) may result in a bad solution, in this case it will cost too much to implement. Instead we must find a compromise that results in a good overall solution. That is, a solution that improves the system and can be implemented within the given budget. One reasonable compromise is to merge the reservation and registry subsystems. The result can be seen in Fig. 12 that presents the subsystem diagram of the alternative decomposition, grouping together reservations with the registry, and in Fig. 13, showing the revised component model. Note the difference to the prior figures is the introduction of the guest attribute.

Whereas merging the two subsystems reduces the cohesion of the result, it also reduces the cost to an acceptable level. We replace the 4 Man/Month units of work that went into the development of a new reservations component with

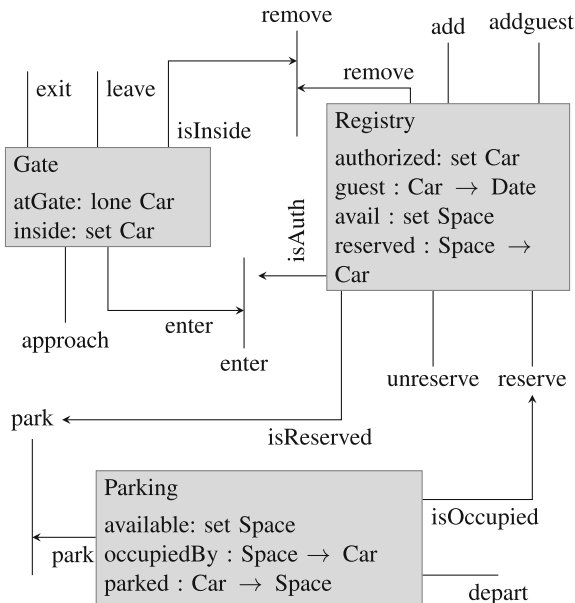


Fig. 12. Subsystem diagram of the alternative decomposition that groups reservations with the registry.

1 Man/Month of work for adding the guest functionality to the existing registry subsystem. The rest of the work is not changed. As a result the total cost is reduced to 17 Man/Months, exactly as allowed by the budget.

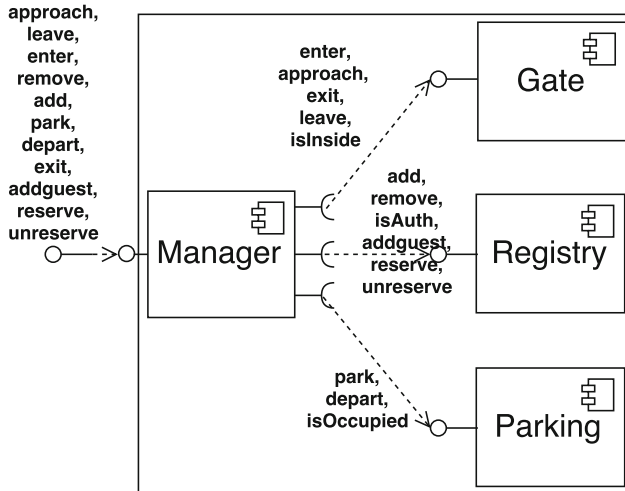


Fig. 13. A revised component model of the evolved parking lot management system.

7 Related Work

Vanya et al. [25] suggest to assess the current decomposition by considering its past evolution, searching for components that often changed together. This approach is useful for assessing the current state of the system; however, unlike our work, it cannot be used to evaluate the impact of future changes.

Streekmann [26] introduces an approach to restructuring software architectures, meant to support engineers with modernizing existing legacy systems. There are two important differences between this work and our work. First, it expects the user to manually supply dependency weights between the original system elements. Second, it takes the target decomposition as a given goal. Thus this work is more relevant for the actual process of implementing the transformation whereas our work is more relevant for initially exploring the space of possible transformations.

We divide the rest of this section into work that is related to functional decomposition and into work that is related to Architecture-based change impact propagation.

Functional Decomposition

Parnas [27] argued that modules should hide parts of the system that may change in the future, thus protecting the rest of the system from the effects of these changes. We consider information hiding to be orthogonal to our work, as in essence it argues for implementing systems using high level abstractions, while we consider the problem of decomposing a system model (i.e., already at a high level of abstraction) into coherent subsystems at the same level of abstraction. Both techniques are examples of the principle of separation of concerns. But while information hiding separates the concerns of purpose from implementation, we separate different functional concerns into separate subsystems.

Event-B [28] supports a notion of process refinement, where an abstract atomic event is refined to a sequence of events at a more concrete level. It is possible to reason about subsystems in Event-B using events, but we show that they are not necessary, as simple propositional logic is enough.

UML component diagrams describe what services one subsystem requires from another, but they do not show how several subsystems collaborate to implement the system. In contrast, our subsystem diagram shows which subsystems (and which subsystem operations) collaborate to implement every system operation. In addition, our subsystem diagram can be formally derived from a formal model and thus serve as a guiding and organizing map of the model.

There exists a large body of research on computing coupling and cohesion metrics for object oriented systems [29–34]. These works focus on assessing the quality of existing decompositions, by measuring properties of the code. As a result they cannot be used at the early design phase.

Subsystem decomposition is similar in many ways to component identification. Researchers have suggested several approaches for identifying components (see [35] for a survey). These approaches are essentially elaborations of the heuristic mentioned in the introduction. They define a metric for measuring the similarity between use cases [36] (or classes [37]) and use clustering algorithms to collect groups of similar use cases (or classes) into components. When a class participates in several use cases, a conflict resolution algorithm decides which component is allocated the class (no work considers the possibility of partitioning the class itself between the components). And none of these works discuss the problems that we have raised. Indeed it is not even clear what is meant by allocating a class to a component. Does it mean that the class is entirely hidden inside the component? if so, what happens to the components whose use cases (or objects) refer to instances of this class? and what happens when a use case requires the cooperation of several components (for example, sitting a guest at a table requires the cooperation of the checkin and reservation subsystems)? None of these works consider these problems. In fact, we could not compare these approaches with ours, because they either lack examples entirely (e.g. [36]) or use examples that lack key details such as the actual uses cases (e.g. [37–39]).

The work most relevant to our approach is [40]. It describes a clustering technique for software requirements based on how they reference common *attributes*, where an attribute is any descriptive property or object associated

with a requirement. The requirements and the attributes are written in a table, then the requirements are clustered based on their similarity with respect to the attributes that they use. The result is displayed as a dendrogram—a tree whose leaves are the requirements and the degree of similarity is higher the deeper the nodes are in the tree. The requirements are then partitioned into subsystems by selecting subtrees of the dendrogram. Compared with our approach, only the functional requirements are clustered, there is no rule to cluster the attributes. This is problematic when attributes are shared between requirements from different components. Second, the technique does not consider relationships between the attributes. Next, it offers no way to check correctness. Finally, because the dependencies between the suggested subsystems are not visible, it is more difficult to explore different alternatives.

Architecture-Based Change Propagation

Work related to change effort identification and maintainability estimation can be put into four categories sketched in the following. Detailed description is given in [6].

- (i) Task-based Project Planning such as Hierarchical Task Analysis (HTA) [41] for decomposing a high-level task into a hierarchy of subtasks or the Comprehensive Cost Model (COCOMO) II [42] for estimating costs during requirements phase and architectural design phase by applying the abstract measure of applications points (i.e. function points) based on an informal requirements description. Yet, if used at all, these techniques only apply coarse-grained architectural artifacts which make accurate predictions difficult.
- (ii) Architecture-based Project Planning such as Architecture-Centered Software Project Planning (ACSPP) [43] deem software architecture as an artefact in project planning while combining top-down and bottom-up effort estimation techniques. Existing approaches on architecture-based project planning, however, do not support estimating change efforts based on a given architecture and do not allow for automated change impact analysis and derivation of change activities [6].
- (iii) Architecture-based Software Evolution such as Garlan et al. [44] propose a pattern-based approach to assist in expressing architectural evolution and for reasoning about the correctness and quality of evolution paths. As discussed before, work on architecture-based software evolution does not support change effort estimation and impact analysis.
- (iv) Scenario-based Architecture Analysis such as Software Architecture Analysis Method (SAAM) [45] evaluates software architectures regarding modifiability by using an informal architecture description (mainly the structural view). Architecture-Centric Project Management (ACPM) [43] takes software architecture as the central artifact for planning and management activities. For architecture-based cost estimation, the architecture is applied to decompose planned software changes into various tasks to realize the changes.

8 Conclusion

Software systems need to evolve continuously to avoid becoming less useful. In this paper we have presented an approach that employs functional decomposition for software architecture evolution, considering also non-functional constraints—namely implementation efforts.

We have used a formal technique to model and reason about software subsystem decomposition. A diagram notation is used to show how the system state variables and operations are partitioned into subsystems, and how the subsystems collaborate to implement the system operations. The visualization technique that we have presented can be applied to informal models (such as UML class diagrams), provided that those models can be described in relational terms.

We have used a systematic functional decomposition to achieve good modularity in terms of low coupling and high cohesion, thus obtaining good inner quality. However, when deciding on a system architecture, it is not enough to consider only the functional requirements of the system. Using the KAMP approach we can refer also to implementation efforts, avoiding situations where the required changes are too costly to implement. A system decomposition can be detected automatically, yet we can change the model of the system in order to reduce coupling when the decomposition is forced by non-functional constraints.

We have avoided calculating explicit metrics for the quality of decompositions. Deciding on a decomposition must involve integrating much more information and knowledge about the system, information that currently does not exist in the diagrams. We also plan to extend our theory to support concurrent execution of subsystems.

Acknowledgement. This work has been partially supported by GIF (grant No. 1131-9.6/2011) and the DFG (German Research Foundation) under the Priority Programme SPP1593.

References

1. Lehman, M.M.: On understanding laws, evolution, and conservation in the large-program life cycle. *J. Syst. Softw.* **1**, 213–221 (1980)
2. Williams, B.J., Carver, J.C.: Characterizing software architecture changes: a systematic review. *Inf. Softw. Technol.* **52**, 31–51 (2010)
3. Yskout, K., Scandariato, R., Joosen, W.: Change patterns - co-evolving requirements and architecture. *Softw. Syst. Model.* **13**, 625–648 (2014)
4. Faitelson, D., Tyszberowicz, S.: Improving design decomposition (extended version). *Form. Asp. Comput.* **29**, 601–627 (2017)
5. Breivold, H.P., Crnkovic, I., Larsson, M.: A systematic review of software architecture evolution research. *Inf. Softw. Technol.* **54**, 16–40 (2012)
6. Rostami, K., Stammel, J., Heinrich, R., Reussner, R.: Architecture-based assessment and planning of change requests. In: 11th International ACM SIGSOFT Conference on Quality of Software Architectures, QoSA, pp. 21–30 (2015)

7. Faitelson, D., Heinrich, R., Tyszberowicz, S.: Supporting software architecture evolution by functional decomposition. In: Pires, L.F., Hammoudi, S., Selic, B. (eds.) 5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD), Porto, Portugal, pp. 435–442 (2017)
8. Woodcock, J., Davies, J.: Using Z: Specification, Refinement, and Proof. Prentice-Hall, Upper Saddle River (1996)
9. Abrial, J.: The B-book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (2005)
10. Abrial, J., Hallerstede, S.: Refinement, decomposition, and instantiation of discrete models: application to Event-B. *Fundam. Inform.* **77**, 1–28 (2007)
11. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, Cambridge (2012)
12. Codd, E.: The Relational Model for Database Management. Addison-Wesley Longman Publishing, Boston (1990)
13. Spivey, J.M.: Z Notation - A Reference Manual, 2nd edn. Prentice Hall, Upper Saddle River (1992)
14. Subsystem decomposition. <http://goo.gl/m5gnW3>. Accessed Apr 2018
15. Morgan, C.: Programming from Specifications. Prentice-Hall, Inc., Upper Saddle River (1990)
16. Faitelson, D., Tyszberowicz, S.: Improving design decomposition. In: Li, X., Liu, Z., Yi, W. (eds.) SETTA 2015. LNCS, vol. 9409, pp. 185–200. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25942-0_12
17. North, S.C.: Drawing graphs with NEATO. NEATO User's Manual (2004)
18. Kamada, T., Kawai, S.: An algorithm for drawing general undirected graphs. *Inf. Process. Lett.* **31**, 7–15 (1989)
19. OMG: UML superstructure specification, v2.4.1. Technical report, OMG (2011)
20. Reussner, R.H., Becker, S., Happe, J., Heinrich, R., Koziolok, A., Koziolok, H., Kramer, M., Krogmann, K.: Modeling and Simulating Software Architectures: The Palladio Approach. MIT Press, Cambridge (2016)
21. Heinrich, R., Rostami, K., Stammel, J., Knapp, T., Reussner, R.: Architecture-based analysis of changes in information system evolution. In: 17th Workshop Software-Reengineering & Evolution, SWT-Trends, vol. 34 (2015)
22. Heger, C., Heinrich, R.: Deriving work plans for solving performance and scalability problems. In: Horváth, A., Wolter, K. (eds.) EPEW 2014. LNCS, vol. 8721, pp. 104–118. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10885-8_8
23. Rostami, K., Heinrich, R., Busch, A., Reussner, R.: Architecture-based change impact analysis in information systems and business processes. In: 2017 IEEE International Conference on Software Architecture (ICSA), pp. 179–188 (2017)
24. Vogel-Heuser, B., Heinrich, R., Cha, S., Rostami, K., Ocker, F., Koch, S., Reussner, R., Ziegler, S.: Maintenance effort estimation with kamp4aps for cross-disciplinary automated production systems - a collaborative approach. In: 20th IFAC World Congress, Toulouse, France (2017)
25. Vanya, A., Klusener, S., Premraj, R., van Vliet, H.: Supporting software architects to improve their software system's decomposition - lessons learned. *J. Softw.: Evol. Process* **25**, 219–232 (2013)
26. Streekmann, N.: Clustering-Based Support for Software Architecture Restructuring. Software Engineering Research. Vieweg+Teubner Verlag, Wiesbaden (2011). <https://doi.org/10.1007/978-3-8348-8675-0>
27. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. In: Broy, M., Denert, E. (eds.) Software Pioneers, pp. 1–6. Springer, Heidelberg (2002). https://doi.org/10.1007/978-3-642-59412-0_26

28. Abrial, J.: *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, Cambridge (2010)
29. e Abreu, F.B., Goulão, M.: Coupling and cohesion as modularization drivers: are we being over-persuaded? In: 5th European Conference on Software Maintenance and Reengineering, CSMR, pp. 47–57 (2001)
30. Al-Dallal, J.: Measuring the discriminative power of object-oriented class cohesion metrics. *Trans. Softw. Eng.* **37**, 788–804 (2011)
31. Moser, M., Misis, V.B.: Measuring class coupling and cohesion: a formal metamodel approach. In: 4th Asia-Pacific Software Engineering and International Computer Science Conference, APSEC, pp. 31–40 (1997)
32. Mayer, T., Hall, T.: Measuring OO systems: a critical analysis of the MOOD metrics. In: TOOLS, pp. 108–117 (1999)
33. Darcy, D.P., Kemerer, C.F.: Software complexity: toward a unified theory of coupling and cohesion. In: Friday Workshops, Information Systems Research Center, Carlson School of Management, University of Minnesota (2002)
34. Hitz, M., Montazeri, B.: Measuring coupling and cohesion in object-oriented systems. In: International Symposium on Applied Corporate Computing (ISACC), pp. 1–10 (1995)
35. Birkmeier, D., Overhage, S.: On component identification approaches – classification, state of the art, and comparison. In: Lewis, G.A., Poernomo, I., Hofmeister, C. (eds.) CBSE 2009. LNCS, vol. 5582, pp. 1–18. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02414-6_1
36. Kim, S.D., Chang, S.H.: A systematic method to identify software components. In: 11th Asia-Pacific Software Engineering Conference (APSEC), pp. 538–545 (2004)
37. Lee, J.K., Jung, S.J., Kim, S.D., Jang, W.H., Ham, D.H.: Component identification method with coupling and cohesion. In: APSEC, pp. 79–86 (2001)
38. Jang, Y.-J., Kim, E.-Y., Lee, K.-W.: Object-oriented component identification method using the affinity analysis technique. In: Konstantas, D., Léonard, M., Pigneur, Y., Patel, S. (eds.) OOIS 2003. LNCS, vol. 2817, pp. 317–321. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45242-3_33
39. Fan-Chao, M., Den-Chen, Z., Xiao-Fei, X.: Business component identification of enterprise information system: a hierarchical clustering method. In: IEEE International Conference on e-Business Engineering, ICEBE, pp. 473–480 (2005)
40. Lung, C.H., Xu, X., Zaman, M.: Software architecture decomposition using attributes. *Softw. Eng. Knowl. Eng.* **17**, 599–613 (2007)
41. Kirwan, B., Ainsworth, L.: *A Guide To Task Analysis: The Task Analysis Working Group*. Taylor & Francis, Abingdon (2003)
42. Boehm, B.W., et al.: *Software Cost Estimation with Cocomo II with Cdrom*. Prentice Hall, Upper Saddle River (2000)
43. Paulish, D.J.: *Architecture-Centric Software Project Management: A Practical Guide*. AW, Boston (2002)
44. Garlan, D., et al.: Evolution styles: foundations and tool support for software architecture evolution. In: *Software Architecture, WICSA/ECSA*, pp. 131–140. IEEE (2009)



Model-Driven Approach to Handle Evolutions of OLAP Requirements and Data Source Model

Said Taktak¹(✉), Jamel Feki², Abdulrahman Altalhi³,
and Gilles Zurfluh⁴

¹ FSEGS Faculty, Miracl Laboratory, University of Sfax, Sfax, Tunisia
said.taktak@fsegs.rnu.tn

² Faculty of Computing and IT, University of Jeddah,
Jeddah, Saudi Arabia
jfeiki@uj.edu.sa

³ Faculty of Computing and IT, King Abdelaziz University,
Jeddah, Saudi Arabia
ahaltalhi@kau.edu.sa

⁴ IRIT Laboratory, University of Toulouse 1 Capitole, Toulouse, France
gilles.zurfluh@ut-capitole.fr

Abstract. Data Warehouse (DW) evolution is becoming a critical research topic for several organizations mainly because their analytical data change permanently and rapidly due to changes in the data source and decision-makers' requirements. This paper presents an MDA-compliant (Model Driven Architecture) approach and a software tool for propagating automatically the evolutions of the data source model and OLAP (On-Line Analytical Processing) requirements towards the multidimensional DW model. More accurately, we propose a DWE (Data Warehouse Evolution) framework. Being MDA compliant, we perform this DW evolution through Model-To-Model transformation rules we have defined as QVT (Query/View/Transformation) along with M2T (Model-To-Text) transformations realized using *Acceleo* templates. Thus, the evolution operations (Create table, Add column...) are firstly modeled, secondly transformed into multidimensional evolution operations (Create dimension, hierarchy...), and then are used with *Acceleo* templates for generating the DW alteration code.

Keywords: Data warehouse · Evolution modeling · Data source model
OLAP requirements · MDA · M2M · M2T

1 Introduction

Nowadays the DW is a powerful technology for strengthening the decision-making process within organizations. It gathers synthesis information from internal and/or external operational data sources.

DW modeling has been considered, for more than one decade, as a real challenging research topic for which several approaches were proposed. Three major categories of

approaches for designing a DW schema (i.e., data-model) are well known in the literature: Top-down [1], bottom-up [2, 3], and mixed [4] approaches.

All these DW design approaches rely on a rigid assumption; they consider that the conceptual model of the DW is *time-invariant*. However, in practice, this assumption is unrealistic most of the time, and therefore restricts the evolution of the real world. In fact, the DW model may evolve due to internal and/or external factors (e.g., business processes changes, organization environment evolution). Furthermore, it is difficult to fix definitively the DW model at the design phase; in fact, for sustainability issues, the DW model should often undergo changes after its implementation. These changes are due to two main reasons namely (a) Evolution of the analytical needs: changes in these needs might require extending the DW model (e.g. adding new axes or subjects of analysis), and (b) Evolution of the data source model (DS) due to the evolution of the business processes (e.g., adding/removing conceptual entities). To the best of our knowledge, we can claim that the problem of changes in the DW model needs more research investigations and appropriate software features. Indeed, all evolution strategies proposed in the DW literature are at a single level of modeling: schemas before and after changes conform to the same meta-model. In the DW domain, the evolution of schemas expressed in different models has not yet received its full share of the investigation.

To alleviate this problem, we propose in this paper an MDA (Model Driven Architecture) approach [14] that automates the propagation of the evolution of the DS model and the evolution of decision-makers' requirements (OLAP-requirements in the remaining of the paper) towards its associated DW model. In this context, we suggest an approach based on a classification of the evolution scenarios and a set of transformation rules to identifying the evolution operations to apply to the DW model.

This paper is organized as follows. Section 2 provides a review of works dealing with the DW evolution problem. Section 3 describes our MDA-based approach for the propagation of OLAP-requirements and DS evolution towards the multidimensional DW model. Section 4 discusses the effect of the evolution of the DS model on the DW. Section 5 introduces our classification of evolutions of OLAP-requirements; it develops algorithms to derive the appropriate changes that should apply on the DW model. Section 6 describes the implementation of the DW evolution through MDA transformations at two levels: Model-To-Model (M2M) and Model-To-Text (M2T). Finally, Sect. 7 concludes the paper and enumerates its perspectives.

2 Related Works

The DW evolution problem is considered from two main viewpoints: (a) *Evolution of data source model*, and (b) *Evolution of business requirements of decision-makers*. Hereafter, we review the approaches for each trends.

2.1 Approaches Based on DS Evolution Model

Organizations' business processes evolve over time due to the modification of existing processes or the emergence of new ones that may create new real world objects.

Naturally, these evolutions affect the data-model of the DS (i.e., information system) that feeds the DW with data. In turn, the DW cannot be immunized against the evolutions of its DS; consequently this evolution deserves to be studied in order to semi-(automatically) propagate towards the DW data-model and the ETL (Extract Transformed and Load) process. This evolution problem has been addressed from different perspectives; we classify the related works into three main categories: (i) *Evolution of the DW multidimensional model*, (ii) *Maintenance of materialized views*, and (iii) *Adaptation of the ETL process*.

Works addressing views maintenance consider the DW as a set of materialized views directly built on, and loaded from, the DS. In this category of approaches, any change in the DS data-model requires views maintenance efforts. As a practical extension, [5, 6] proposed approaches for a dynamic adaptation of materialized views in response to the evolution of the DS/DW. These approaches maintain not only the schema views but also their content; they mainly attempt to avoid recalculating views after DS changes by deriving a new schema view from the old one. More details on views maintenance in multidimensional context are available in [7].

Other research works adapt the ETL process when the DS data-model evolve. Among these works, the authors in [8, 9] provide a mechanism for adapting the ETL tasks to the changes occurred in the DS data-model. However, this study was restricted to the ETL process without tackling the impact of the DS evolution on the DW model components (Dimensions, facts, hierarchies...).

To lighten these shortcomings, the authors in [10] have defined a formal model for a multi-version DW. They presented a set of evolution operations that affect the DW schema and content. These authors have distinguished two types of DW versions: *real version* and *alternative version*. The DW real version reflects the changes in the real world environment of the organization whereas the DW alternative version simulates the change process; to do so, “What-If” analysis strategy was adopted. Furthermore, the authors have developed the MVDW (Multi-Version Data Warehouse) prototype for the DW maintenance and versions management. A major drawback of this contribution is the manual identification of the DW evolution operations; this identification requires high expertise of the DW administrator and, therefore, is out of reach of end-users.

2.2 Approaches Based on Business Requirement Evolution

Let us note that in mixed DW design approaches [4], the design of the DW relies, from the one hand, on the DS model and, from the other hand, on the OLAP-requirements. Obviously, OLAP-requirements could not be static in time; therefore, the DW-design driven by user requirements may become obsolete and no longer comply the new requirements. To overcome this issue, it is necessary to consider the new analytical requirements and then adapt the DW to encompass them. Among the research works of this category, the authors in [11] suggested an approach for the customization of analyses based on “If-Then” rules model; this model allows users to integrate their own knowledge in order to enlarge the panoply of analysis on the DW by changing the DW schema. The suggested evolution operations affect only two components of the DW: dimensions and hierarchies. The authors have developed a prototype called *WEDriK* (Warehouse Evolution Driven by Knowledge) based on a set of DW evolution

algorithms to create new analytical axes. The analytical requirements introduced by each user are processed and transformed into DW evolution operations. Nevertheless, the authors assume that the DW users are skilled enough to express properly their requirements. Moreover, the supported changes are simple: they do not cover all cases that decision-makers may ask for.

To overcome this problem, in [12] the authors studied the evolution of complex hierarchies (multiple alternative hierarchies, dependent and independent parallel hierarchies). They defined constraint-based evolution operations to ensure data integrity and schema consistency of the new DW model. Operations and constraints are defined in ULD (Uni-Level Description language) and MDD (Multilevel Dictionary Definition). This study is an extension of the work in [12] where the authors presented a conceptual requirement-oriented framework called *DWEVOLVE* for DW evolution. It analyzes the changes in the requirements specified by stakeholders as well as developers, and then incorporates them into the DW by performing appropriate additions, deletions and updates. Nevertheless, the authors do not suggest mechanism for automatic inference of evolution operations from OLAP-requirements. In fact, this task is borne entirely by the DW administrator.

In the same context, in [13] authors have also investigated the problem of business requirements evolution. They defined a formalism for modeling the new analytical needs and proposed a semi-automatic approach to adjust and create a new version for the DW model. However, the evolution operations supported by this solution are simple and lack accuracy. For instance, when adding an attribute, the proposed algorithm is able to identify just the dimension to change but cannot find the role of the new attribute in the dimension, i.e., whether it creates a hierarchy or inserts a level into an existing hierarchy... How to find the role of the new component is really a hard task left to a skilled user.

2.3 Discussion

In the related works section we have focused on two complementary categories of evolutions in DW systems, namely evolution of the DS model and evolution of OLAP needs. We have identified three deficiencies concerning (i) complementarity, (ii) complexity of the evolutions, and (iii) automatic propagation of changes from the DS toward the DW.

First, concerning the complementarity, to the best of our knowledge, no solution has combined the DS evolution with business requirements evolution so far. Indeed, contributions have addressed these two evolutions separately.

Secondly, few works were interested in studying the DS evolution effect on the multidimensional model. Moreover, most of these works provide solutions touching a few isolated aspects and treating simple evolution cases (i.e., Dimension evolution, Fact evolution, ETL evolution).

Thirdly, automatic propagation was not a main concern in these works, and where addressed, it was carried out according to traditional modeling and programming approaches.

Finally, from the technological viewpoint, we note that all proposed solutions were realized in a conventional software engineering context; therefore, implementations are

platform-dependent. Obviously, using the MDA approach allows benefiting from its multiple advantages.

The objective of this paper is to propose a Data Warehouse Evolution framework (*DWE*) as a complete solution covering the DS evolution and the OLAP-requirements evolution. Our proposal is MDA compliant, it promotes the automatic propagation of changes occurred in the DS model along with business requirements changes towards the multidimensional DW. Relying *DWE* on the MDA technology is really a challenging proof. In fact, MDA facilitates realizing our proposed approach, which inherits benefits from this technology (i.e. platform-independent, reduced efforts, and improved quality of results). We define one for the OLAP needs and one model for the DS evolution. In the remaining of this paper, we present our approach that addresses the DW model evolution problem.

3 Overview of the Proposed Approach

Our MDA-based approach aims to automate the propagation of the changes raised by decision-makers (as new needs) and DS model (as new evolution operations) towards the DW multidimensional model. Figure 1 depicts our approach where the evolution of the DW model is due either to an evolution of its DS model (Fig. 1, panel A), or to an evolution of OLAP needs (panel B). To do so, we define an appropriate evolution model for the new OLAP needs; this enables us reusing our on-hand DW evolution model [15]: we keep the same M2T transformation rules for code generation.

Our approach relies on three evolution models: (i) *DS Evolution Model* (DSEM), (ii) *DW Evolution Model* (DWEV), and (iii) *Requirements Evolution Model* (REM). In addition, it applies M2M and M2T transformations:

- DSEM: This model describes all evolution operations that may affect the relational DS elements (table, column...).
- DWEM: It describes all operations that may affect the multidimensional structures (dimensions, facts...). These operations should be derived from the DSEM model.

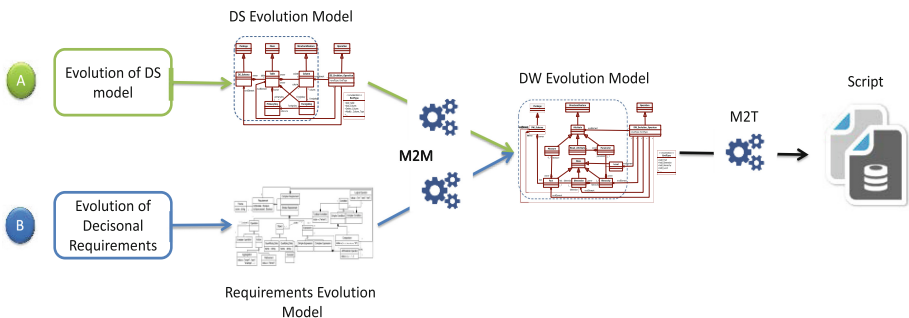


Fig. 1. Overview of our MDA-based DW evolution approach.

- REM: This model describes the new needs of decision-makers in terms of subject and axes of analysis. It also allows defining knowledge introduced by the user (e.g. rules, formulas). This model will be transformed into a DWEM model.
- M2M transformation: It generates the DWEM model from REM model. It relies on automatic mapping between these two models. M2M transformation rules are implemented in QVT (Query-View-Transformation), and use a set of meta-models stored upstream as *Ecore* files.
- M2T transformation: It generates the code that performs the DW model alteration; the generated code results from the DWEM previously generated by applying a set of transformation rules we have formalized in *MOF2Text*. M2T process takes as input the physical model (PSM) along with the DW evolution models; it produces SQL script file(s) for creating or modifying the DW model. We have defined *Acceleo* templates for transforming DWEM operations into an executable script. This transformation process is valid as well for processing the DS evolution as for processing the needs evolution. In fact, this reuse is feasible because these two transformations start from the same DWEM evolution model.

4 Evolution Inferred by the DS Model

The DW schema may evolve over time due to the evolution of its DS data-model. Naturally, the evolution frequency is domain-dependent. As an illustration, in the banking domain the DS of a DW changes every 2–4 weeks on average, also the DS of a telecommunication company is less stable since its schema changes every 7–13 days on average [16].

Two crucial questions arise when the DS evolve: (1) What are the changes to apply to the DW model (i.e., adding a dimension, fact, level of analysis...), and (2) How to perform these changes efficiently and quickly; rapidity is an imperative factor for some decisional systems as argued before. A trivial solution rebuilds the DW from the new DS data-model starting from scratch; but this is a poor approach because the DW reconstruction is a heavy and complex task requiring a lot of time and effort, and is therefore costly. Furthermore, rebuild from scratch cannot be envisaged especially in frequently changing domains. In order to address this evolution issue, we have proposed a model-driven approach for propagating changes from the relational DS towards its DW in an almost-automatic way, thus avoiding the need for the full reconstruction of the DW model (and later its full-reloading process). To do so, we have proposed an MDA-based architecture [14] for propagating the evolution operations occurred on the DS model towards the DW data-model. We have identified sets of evolution operations on the DS and their transformation rules. These operations concern tables, columns, keys...; their execution is not systematic (a precondition should be satisfied). Table 1 lists the evolution operations that could affect the relational DS, and gives for each one the corresponding set of plausible evolution operations we may apply on the DW. For example, in line 1 when we “Add new table” to the DW the effect of this evolution operation may create a “New Fact”, “New Dimension”, “New Hierarchy” and/or “New Level” within an existing hierarchy.

Table 1. DS evolution operations and their corresponding evolution operations on the DW.

DS evolution operation	Plausible DW evolution operation	
Add new table	New fact New dimension	New hierarchy New level
Add new column	New measure New hierarchy New level	New parameter New weak attribute*
Modify column type	Modify weak attribute type* Modify parameter type	Modify parameter type Modify measure type
Drop table	Delete fact Delete dimension	Delete level Delete parameter
Remove column	Delete measure Delete level	Delete weak attribute* Delete hierarchy
Add new constraint	New fact New dimension	New hierarchy New level
Drop constraint	Delete fact Delete dimension	Delete level Delete parameter
Split table	New fact New dimension Delete measure Delete level	New hierarchy New level Delete weak attribute* Delete hierarchy

*Less significant operation

In order to define the transformation rules we adopt the following notation:

- DS: a third normal form relational DS schema
- t : a relational table belonging to DS
- $t.pk$: the set of primary key columns of table t
- $t.Cols$: the set of non-primary key columns of t ($t.pk \cap t.Cols = \emptyset$)
- $t_i \rightarrow t_j$: table t_i references table t_j via a foreign key belonging to t_i
- DW: a multidimensional data warehouse schema loadable from tables in DS
- f : a fact table belonging to DW
- d : a dimension belonging to DW
- $d_i.h_j$: a hierarchy h_j of dimension d_i
- $d_i.h_j.l_k$: a level l_k belonging to $d_i.h_j$
- $d_i.d_j.l_k.p$: a parameter at level $d_i.h_j.l_k$
- $d_i.h_j.l_k.p.W$: a possibly empty set of weak attributes associated with parameter $d_i.h_j.l_k.p$
- $f.M$: the set of measures of fact f
- $f.D$: the set of dimensions of fact f
- Load (t, d): A Boolean function returning True if table t loads dimension d .

In this section, we limit ourselves to detail two transformation rules:

- Transforming a table into a dimension,
- Transforming a table into a fact,

In our running example (Fig. 2), let us create the table *RETAIL_OUTLET* (*Id_Ro, Ro_name, Ro_zone...*) that is referenced by the DS table *SALE* that feeds the fact *F_SALE*. Applying rule *T2D*, the new table creates a new dimension called *D_RETAIL_OUTLET* linked to the *F_SALE* fact.

4.2 Transforming a Table into a Fact

The creation of table t_{new} in the DS using *Add_table*(t_{new}) may create a new fact f_{new} in the DW by calling the *Add_fact*(f_{new}) operation. This evolution is realized by applying rule *T2F* hereafter.

Rule T2F: Table-To-Fact

Input:

- t_{new} : table added to the DS via *Add_Table* (t : Table)
- DS, DW

Condition:

- $t_{new} \rightarrow DS.t_1 \dots, DS.t_k$ with $k \geq 2$ -- t_{new} references k tables in the DS
- Numeric (t_{new}) $\neq \emptyset$ -- t_{new} has numeric attributes
- Load ($DS.t_i, DW.d_j$) with $1 \leq i \leq k$ -- each table t_i referenced by t_{new} loads a dim d_j

Processing: /*Create a new fact f_{new} */

- Find D' , the set of all dimensions loaded from tables referenced by t_{new}
- $f_{new}.D := D'$ -- link f_{new} with dimensions in D'
- $f_{new}.M :=$ Numeric ($T_{new}.Cols$) -- numeric columns of t_{new} become Measures in f_{new}
- $f_{new}.name :=$ "F_" + $t_{new}.name$ -- name of the new fact.

Output:

- f_{new} : new fact added to DS via the *Add_Fact* (f : Fact) operation.

Continuing with our example, we create the table *SCORE_PROD* (*#Id_Prod, #Id_Cust, ScoreNumeric...*) that references two tables *PRODUCT* and *CUSTOMER* in the DS. These tables feed respectively the two dimensions *D_PRODUCT* and *D_CUSTOMER*. Applying rule *T2F*, the new table creates a fact called *F_SCORE_PROD* with *Score* as a measure, related to *D_CUSTOMER* and *D_PRODUCT* dimensions.

5 Evolution Implied by the Decision Makers Needs

The evolution of the OLAP-requirements leads to several cases of evolution on the DW model. We group these evolution cases into three classes namely: *Evolution by derivation*, *Evolution by reorganization*, and *Evolution by extension*. More details about this classification are available in [17]. We clarify these classes and we textually explain the transformation rules that generate the modifications operations to apply on the DW model when the OLAP-requirements evolve.

- *Statico*: Nothing to change if the current DW model meets a new requirement.
- *Reorganization*: Applies when the necessary elements (i.e., measure or attribute) for the new requirement already exist in the DW model but their current roles are not

adequate. We change the role of such elements by creating new links between some elements of the DW model. This reorganization mainly affects temporal and spatial dimensions.

- *Derivation*: If an element is vital for a new requirement but is not in the DW model, therefore, we check if it is derivable from an existing DW element; the derivation uses knowledge introduced by the DW administrator as rules or formulae. Otherwise, if the vital element is derivable from the DS, then we extend the DW model with the derived element.
- *Extension*: This alternative is the most delicate. In fact, when the DW model cannot satisfy the new requirement, either by derivation or by re-formulation, we have to identify which element from the DS we should add to the DW and define its role, and then we expand the DW model with the new element.

In order to decide which alternative of evolution - from above -to apply to the DW model, we develop the Main algorithm (Algorithm 1).

Note that we use these alternatives independently or combined. In the following, we detail each one and specify the evolution operations to perform it. To do so, we use the notation below:

- *Req*: a new requirement
- *A*: set of attributes describing *Req*; *A* divides into two subsets $A = A_{quant} \cup A_{qual}$
- A_{qual} : all qualitative attributes of *Req*
- A_{quant} : all quantitative attributes of *Req*.
- *DW*: set of elements of the DW multidimensional model (i.e., schema)
- *DS*: set of elements of the DS model.

The *Main* algorithm depicts the principle of defining the evolution strategy. It calls three algorithms *Reorganize* (Algorithm 2), *Derive* (Algorithm 3) and *Extend* (Algorithm 4).

Algorithm 1: Main.

```

Input:
  Req, DW, DS
Begin:
1.  if DW_answer(Req) then
2.      Null // No changes to do on the DW model
3.  else if  $A \subseteq DW$  then
4.      Reorganize() // see Reorganize algorithm
5.      else
6.          for each  $a \in A$ 
7.              if  $a \notin DW$  and (Rule(a) or Formula(a)) then
8.                  Derive() // see Derive algorithm
9.              else if  $a \notin DW$  and  $a \in DS$  then
10.                 Extend() // see Extend algorithm
11.             end if
12.          end for
13.      end if
14.  end if
End.

```

$DW_answer(Req)$ is a Boolean function that returns True if the DW model meets the new requirement (Req): *Statico* alternative, and False otherwise.

$Rule(a)$ is a Boolean function True if attribute a is defined through a rule, and False otherwise.

$Formula(a)$ is a Boolean function True if attribute a is defined through a formula, and False otherwise.

5.1 Reorganization

The reorganization process (see Algorithm 2) begins with the identification of the DW elements (fact, dimensions) for the new requirement. It calls two functions $Find_Fact$ and $Find_Dimension$; these functions return respectively the fact containing quantitative attributes A_{quant} , and dimensions containing qualitative attributes A_{qual} . The fact f_{new} will be enriched with the set of measures A_{quant} attributes. Dimensions containing A_{qual} attributes are refined using the $Refine$ function before their link to the new fact. This function prunes hierarchies by eliminating unnecessary attributes for the new requirement.

Algorithm 2: Reorganize.

```

Input:
 $A_{quant}, A_{qual}$ 
Begin:
1.  $f = Find\_Fact(A_{quant})$ 
2.  $D = Find\_Dimensions(A_{qual})$ 
3. if  $f == \emptyset$  Then
4.    $f_{new.M} = A_{quant}$ 
5. else
6.    $f_{new} = f$ 
7. end if
8. for each  $d \in D$ 
9.    $d_{new} = Refine(d)$ 
10.   $d_{new}.f = f_{new}$ 
11.  Add_Dimension( $d_{new}$ )
12. end for
13. Add_Fact( $f_{new}$ )
End.

```

5.2 Derivation

The *Derive* algorithm describes the derivation process; it takes as input the attribute to derive as well as the knowledge given by the DW administrator as rules or formulae. We treat differently qualitative and quantitative attributes of this class.

If the derived attribute a_d is quantitative, and if there is, a fact f related to the dimension that contains the qualitative attributes of the new requirement, then we add a_d to f as new measure m_{new} . Otherwise, we create a new fact f_{new} for the derived attribute a_d .

If a_d is a qualitative attribute, it necessarily belongs to a dimension where its position generally depends on the a_{source} attribute in the rules. If a_{source} belongs to a terminal level l_t then a_{new} becomes a terminal level l_{new} in the same hierarchy as l_t . Otherwise, we create a new hierarchy h_{new} that contains level l_s and all its predecessor levels. l_{new} adds to the new hierarchy as a terminal level.

Algorithm 3: Derive.

```

Input:
ad: a derived attribute
asource: an attribute of DS model used within a rule or formula
Aquant, Aqual.
Begin:
1. if ad ∈ Aquant and formula(ad) then
2.   f = Find_Fact(Aqual) //find the fact linked to dimensions containing Aqual
3.   if f = ∅ then
4.     fnew.M = fnew.M ∪ ad // define the measure of the new fact
5.     fnew.D = Find_Dimensions(Aqual) //find dimensions containing Aqual
6.     Add_Fact (fnew)
7.   else
8.     mnew = ad ; mnew.fact = f
9.     Add_Measure (mnew)
10.  end if
11. else if ad ∈ Aqual and Rule(ad) then
12.   return level containing asource
13.   if Terminal_Level(ls) then
14.     lnew.h = ls.h //hierarchy of level lnew is ls hierarchy
15.     lnew.p = ad // parameter of lnew is the derived attribute ad
16.     lnew.pred = ls // the predecessor level of lnew is ls
17.   else
18.     hnew.d = ls.h.d //dimension of hnew is the dimension of ls
19.     //the levels of hnew are all ls predecessor levels
20.     Add_hierarchy (hnew)
21.     lnew.p = ad ; lnew.pred = ls ; lnew.h = hnew
22.   end if
23.   Add_Level(lnew)
24. end if
25. End.

```

5.3 Extension

The *Extend* algorithm states the principle of the extension, which enriches the DW model with elements extracted from the DS to satisfy the new OLAP-requirement. We assume that a semi-automatic association between attributes of the new requirement and the DS attributes is provided; this treatment could use a semantic resource or a dictionary of the DS attributes. The role of each element depends on the type (quantitative or qualitative) of its associated attribute and its membership table in the DS.

Algorithm 4: Extend.

```

Input:
DW, DS,
 $a_e$  : attribute to retrieve from the data source
Begin:
1.  $t = Find\_Table(a_e)$  // returns the table that contains  $a_e$ 
2. if  $a_e \in A_{qual}$  then
3. returns the level which is loaded from  $t$ 
4. if  $l == null$  then
5.    $t' = ref(t)$  // returns the table which references table  $t$ 
6.    $t'' = IsRef(t)$  // returns the table which is referenced by  $t$ 
7.   if  $t'$  not null and  $t''$  not null then
8.      $l' = Load\_Level(t')$ 
9.      $l'' = Load\_Level(t'')$ 
10.    if  $l''.pred == l'$  then
11.       $l_{new}.h = l'.h$  ;  $l_{new}.pred = l'$  ;  $l_{new}.succ = l''$ 
12.      Add_Level( $l_{new}$ ) // add level  $l_{new}$ 
13.    end if
14.  else if  $t'$  is not null then
15.     $l' = Load\_Level(t')$ 
16.    if Terminal_Level( $l'$ ) then // add terminal level
17.       $l_{new}.p = a_e$  ;  $l_{new}.pred = l'$  ;  $l_{new}.h = l'.h$ 
18.    else // add hierarchy and a new level
19.       $h_{new}.d = l_s.h.d$ 
20.       $h_{new}.L = l_l..l'$  // the level in the new hierarchy
21.      Add_Hierarchy( $h_{new}$ )
22.       $l_{new}.p = a_e$  ;  $l_{new}.pred = l'$  ;  $l_{new}.h = h_{new}$ 
23.      Add_Level( $l_{new}$ )
24.    end if
25.  end if
26.  else
27.     $a_{new}.p = l.p$ 
28.    Add_Attribute( $a_{new}$ )
29.  end if
30.  else
31.     $f = Load\_Fact(t)$ 
32.    if  $f$  not null then
33.       $m_{new} = a_e$  ;  $m_{new}.fact = f$ 
34.      Add_Measure( $m_{new}$ ) // add measure  $m_{new}$  to the fact I
35.    else
36.       $f_{new}.M = a_e$  ; Add_Fact( $f_{new}$ ) // add fact I
37.    end if
38.  end if
End.

```

If the attribute to extract a_e (a_e belongs to a table t) is qualitative, four situations arise to define the role of a_e in the multidimensional model:

- If table t feeds a level l then it becomes a low attribute by applying the *Add_Attribute* evolution operation.

- If t feeds no levels, and if t is referenced by a table t' which feeds a terminal level l' , then a_e becomes an attribute for a new terminal level l_{new} by applying the *Add_Level* evolution operation.
- If t does not feed any level, and if t is a table referenced by t' and refers to a table t'' - t' and t'' respectively feed the two successive levels l' and l'' - a_e can then feed a hierarchical level inserted between the two levels l' and l'' .
- If t does not feed any level and if t is referenced by table t' which feeds a non-terminal level l' then a_e creates a new hierarchy h_{new} by calling the *Add_Hierarchy* evolution operation. h_{new} contains the level l' and all its predecessor levels in the hierarchy of l' . Then, we create a new terminal level l_{new} for the new hierarchy h_{new} .

When the extracted attribute a_e is quantitative, if t (table of a_e) feeds a fact f , then a_e becomes a measure of f . Otherwise, we create a new fact with the new measure a_e .

6 Implementation

To validate our approach, we have developed a DWE (Data Warehouse Evolution) software prototype under the EMF (Eclipse Modeling Framework) platform that is a complete environment for MDA. Figure 3 shows the DWE overall architecture that offers two evolution features: (i) Evolution of the DW model as a result of changes in its DS model; (ii) Evolution of the DW model to meet new OLAP-requirements.

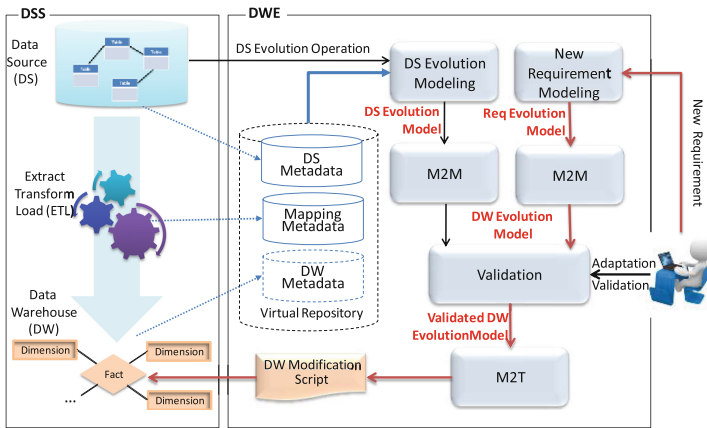


Fig. 3. Architecture of the DWE prototype [17].

The DW evolution process starts with the modelling of the new requirements or changes occurred in the DS model; it aims to generate respectively the requirement evolution model (REM) and the DS evolution model (DSEM). The next step is the M2M that transforms the REM and the DSEM into DWEM. Once the DWEM is generated, thereafter the new DW model displays graphically; this enables the DW

administrator to observe and study the effects (i.e., computed changes) of the DW-evolutions operations. At this stage, the DW administrator can accept the suggested changes or even adapt them. Finally, the M2T process transforms the DWEM into script for DW alteration. In what follows, we detail these steps.

6.1 Modeling of Evolution

We use UML (Unified Modeling Language) class diagrams to define the evolution models DSEM, REM and DWEM. The static property list in the classes define the models of the DS, Requirements and the DW whereas the operations define the changes that may affect each of these structures. Next, we detail these three evolution models.

DS Evolution Model (DSEM)

The DSEM model is the basic model for deducing the DWEM model. It defines the relational DS schema (tables, constraints...) through class properties as well as the evolution operations (add table, add column...).

The DSEM conforms to its Meta-Model in Fig. 4. The latter has two components: (i) The DS Meta-Model (enclosed within the dashed area) stores the DS schema; and (ii) The Meta-Model of the DS Schema Evolution Operations that stores the DS schema evolution operations.

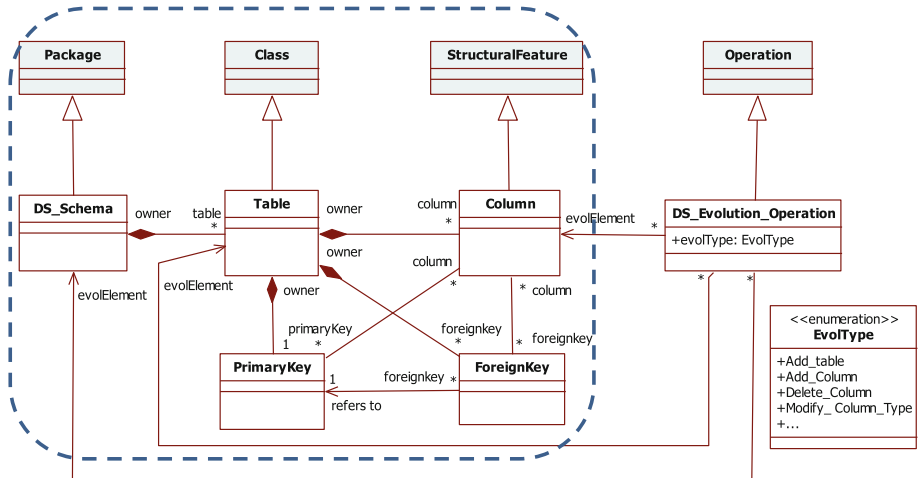


Fig. 4. DS evolution meta-model.

Modeling Decision-makers Requirements

This step takes as input the new requirements expressed as queries, rules or formulae and returns a Requirements Evolution Model (REM) compliant to the Meta-Model in [13] depicted in Fig. 5. A new requirement has *quantitative* and *qualitative* attributes, arithmetic operations (i.e., formula) and logical expressions (i.e., rules).

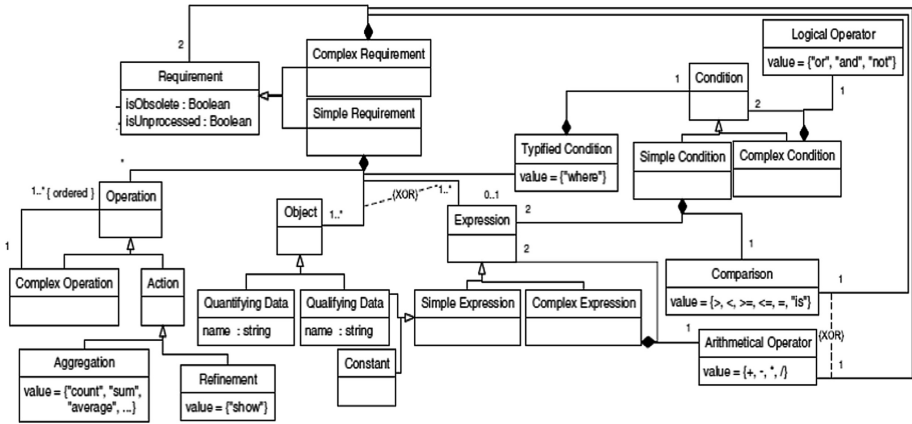


Fig. 5. Requirements evolution meta-model [13].

DW Evolution Model

The DW Evolution Meta-Model has two components (cf. Fig. 6): (i) The DW Meta-Model (dashed area) that stores the DW Schema, and (ii) The Meta-Model of the DW Schema Evolution Operations that stores the DW Schema Evolution Operations. This latter will be deduced automatically from the DSEM using transformation rules.

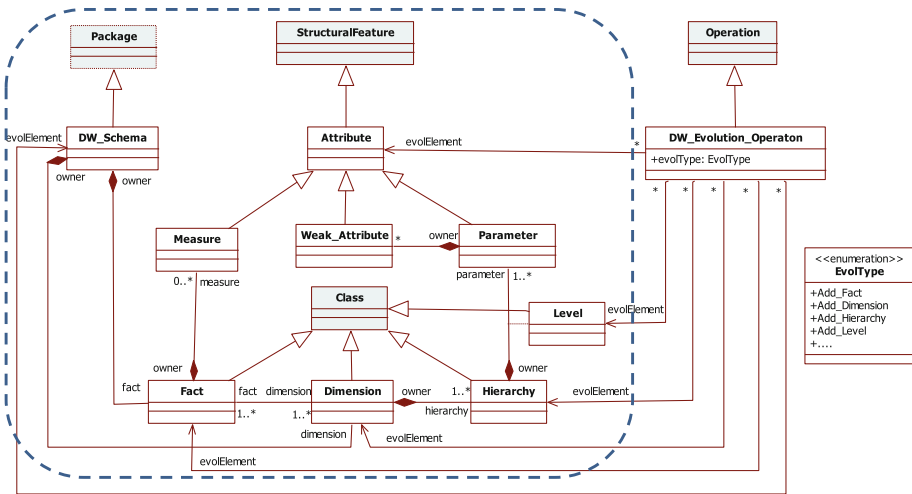


Fig. 6. DW evolution meta-model.

6.2 M2M Transformations in QVT: REM to DWEM

The first aim of our approach is to determine the evolution operations to apply on the DW model after the appearance of new analytical needs. Figure 7 lists transformations potentially applicable to the DW according to the evolution strategies.

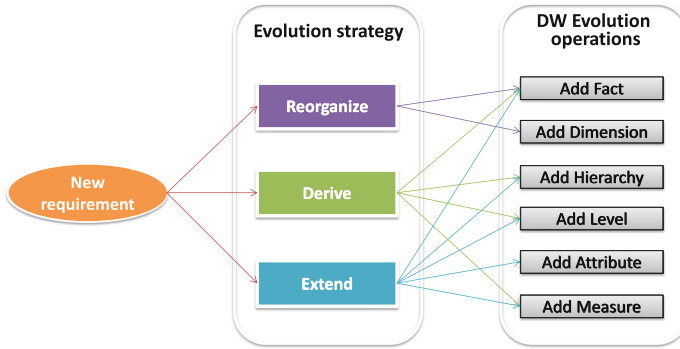


Fig. 7. DW evolution operations for new requirements [17].

Due to space limitation, we define the rules that transform a new requirement into the *Add_Fact* evolution operation that adds a fact into the DW.

Each new requirement, defined using an appropriate model, is converted automatically into a set of evolution operations on the target model (DW evolution model).

The relation *Main* is the entry point of the transformation process; it has elements of the two following models (cf. Fig. 8):

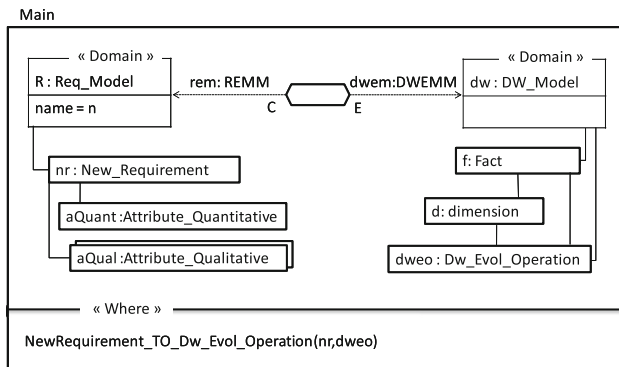


Fig. 8. Graphical representation of the QVT relation *Main* [17].

- « *rem* » model conform to *REMM* (Requirement Evolution Meta-Model),
- « *dwem* » model conform to *DWEMM* (DW Evolution Meta-Model).

The *Domain* element of the « rem » model is marked with « C » (*Checkonly*); this means when a transformation occurs in this direction (i.e. the direction of a Checkonly domain) it simply checks if there is a valid match in the relevant model that satisfies the relationship. The domain of the « dwem » model is marked with « E » (*Enforce*); this means when a transformation occurs in this direction (i.e. the direction of the model of an enforced domain) if the checking fails then the target model « dwem » is modified to satisfy this relation. The left side of this relation describes the elements of the source model « rem », which transforms into elements of the target model « dwem ». More specifically, a new requirement from the left « nr: *New_Requirement* » transforms into evolution operation(s) for the DW « dweo: *Dw_Evol_Operation* » by invoking the relation *New_Requirement_TO_Dw_Evolution_Operation* (nr, dweo) specified in the *where* clause. Consequently, the following relations executes:

- *New_Requirement_TO_AddDimension*,
- *New_Requirement_TO_AddLevel*,
- *New_Requirement_TO_AddFact*,
- *New_Requirement_TO_AddMeasure*,
- *New_Requirement_TO_AddParameter*, and
- *New_Requirement_TO_AddAttribute*.

Let us focus on the *New_Requirement_TO_Add_Fact* relation. Figure 9 describes the relation that transforms a new requirement « nr » into the DW evolution operation *Add_Fact*.

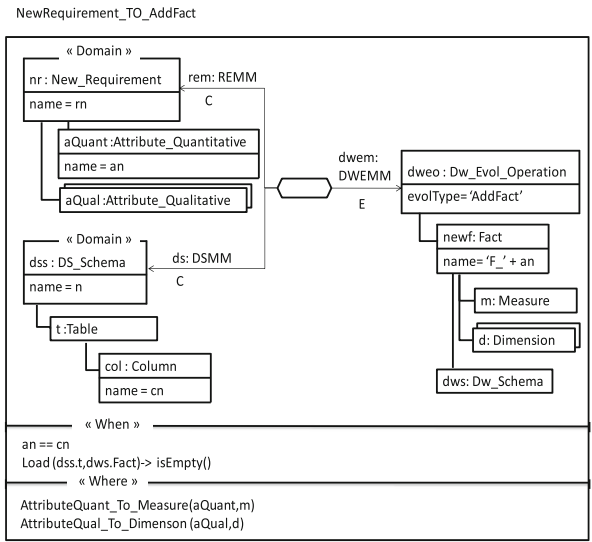


Fig. 9. QVT relation *NewRequirement_TO_AddFact* [17].

Since we are treating the DW evolution problem according to the extension strategy, we have elements from the DS model (« Domain: Ds_Schema ») in the $New_Requirement_TO_Add_Fact$ relation. Truthfully, a quantitative attribute a_{Quant} (in a new requirement nr) that belongs to a table t of the DS model « dss » may create a new fact $newf$ in the DW model « dws » if table t does not load any fact of the « dws ». Then, the a_{Quant} attribute feeds a measure of the new fact $newf$ via the relation $AttributeQuant_to_Measure(a_{Quant},m)$. The dimensions of $newf$ will be deduced from the qualitative attributes present in the new nr requirement using the relation $AttributeQual_To_Dimension(a_{Qual},d)$.

6.3 M2M Transformations in QVT: DSEM to DWEM

Here, we define our QVT rules to transform the DS evolution model into a DW evolution model. Figure 10 depicts how the evolution operations performed on the DS model will be transformed into evolution operations on the DW model. Among these relations, we have selected to detail $AddTable_TO_AddDimension$ and $AddTable_TO_AddFact$.

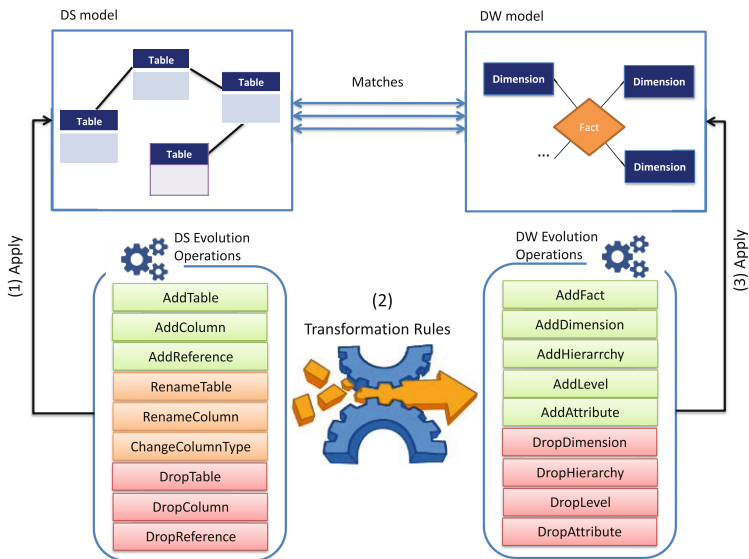


Fig. 10. Principle of transforming DS-evolution operations into DW-evolution operations.

Relation $AddTable_TO_AddDimension$.

Note that in multidimensional modeling, each fact f is associated with a finite set of n ($n > 1$) dimensions; each dimension is an analysis axes of the measures in f . Dimensions are loaded from the DS tables directly or indirectly related to the table that feeds f [18, 19]. Hence, if a new table $newt$ is added to the DS and is referenced by a table that feeds f , then $newt$ transforms into a dimension for f .

This *AddTable* evolution operation is achieved through the QVT relation *AddTable_TO_AddDimension* in Fig. 11 that transforms the operation « *AddTable* » on the DS data-model into the operation « *AddDimension* » on the DW data-model. The *When* clause specifies the condition to check for executing this relation. It means if the new table noted « *newt* » is referenced by a table noted « *refl* » that feeds a fact « *f* » through the relation « *Load(refl, f)* » then « *newt* » will be transformed into a new dimension « *newd* » via the relation « *Table_TO_Dimension(newt, newd)* » specified in the *Where* clause.

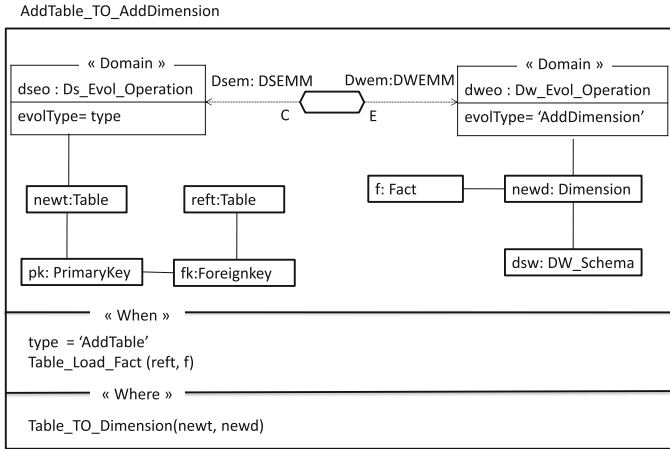


Fig. 11. QVT relation *AddTable_TO_AddDimension*.

Relation *AddTable_TO_AddFact*

In DW design approaches, an n-ary relationship having non-(prime and foreign key) numeric columns transforms into a fact [20, 21].

This heuristic helps us to decide whether a new table added to the DS will transform into a fact or not. Thus, the relation *AddTable_TO_AddFact* transforms the *AddTable* operation into *AddFact* operation on the DW. Figure 12 gives its formalization in QVT explained hereafter. If the new table « *newt* » refers to two tables « *ta* » and « *tb* » that feed two dimensions « *da* » and « *db* » respectively, and if « *newt* » has numeric columns then « *newt* » is likely to transform into fact via the relation « *Table_To_Fact (newt, newf)* ». Numeric columns in *newt* transform into measures through a relation called « *Column_To_Measure (c, m)* » not defined in this paper.

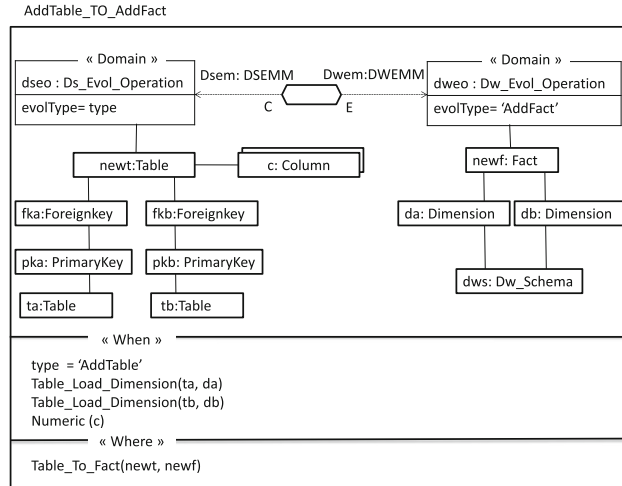


Fig. 12. Relation *AddTable_TO_AddFact* in QVT.

6.4 Validation and Adaptation Module

Once the DWEM is generated, thereafter the new DW model could be visualized graphically; this enables the DW Administrator (DWA) to follow/study the effects (i.e., suggested changes) of the DS-evolutions operations on the original DW model. Figure 13 shows the DWE graphical interface after adding the *Retail_Outlet* table to the DS model. At this stage, the DWA can validate these changes or adapt them according to the evolution requirements. Consequently, the DWEM is automatically modified and then the M2T process generates the code.

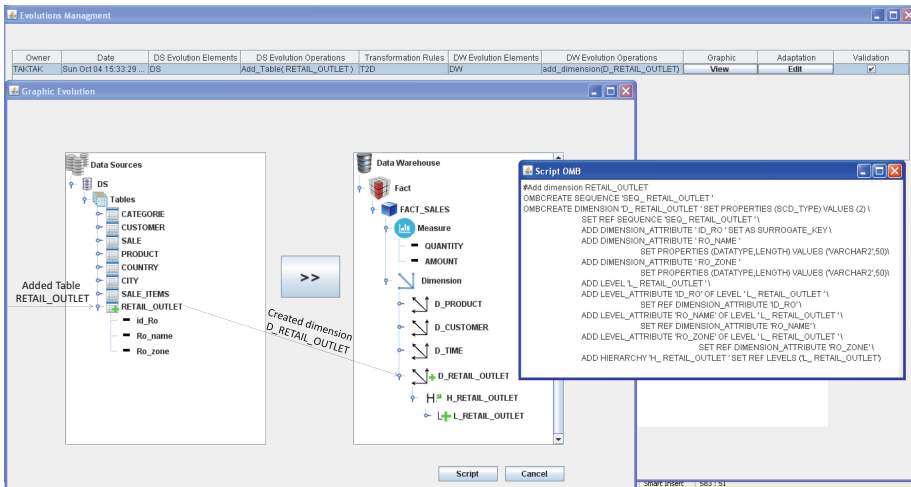


Fig. 13. Sample DWE interfaces (graphical and code).

6.5 Implementing M2t Transformations

We use *Acceleo* plugin that implements the *MOFM2T* standard of the OMG [22]. *Acceleo* provides tools for generating codes from models. This generation of code conforms to a template-based approach.

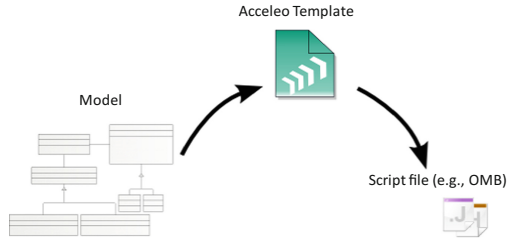


Fig. 14. *Acceleo* schema for the generation of OMB script.

A template is a text containing placeholders to fill with information extracted from the input model (Fig. 14). For our running example, the input model is the DW evolution model issued from the Requirement Evolution Model (REM) or DS Evolution Model (DSEM). For M2T transformations, we developed a PSM (Platform Specific Model) as an *Acceleo* template for generating the code [17] for the target platform *Oracle Warehouse Builder* (OWB). Our Template generates *OMB* (Oracle MetaBase) script that runs under *OMB-Plus* with *Oracle JDeveloper* or *OMB-Plus console*. The execution of this template generates the code to connect to OWB and propagates the changes to the DW data-model (Fig. 13).

7 Preliminary Results and Evaluation

Using the case study of Fig. 2, we have conducted a preliminary assessment by considering a significant set of DS and OLAP-requirements evolution scenarios leading to changes on the DW model, as the creation of new facts and dimensions. The achieved results are very promising. Hereafter, we present four evolution scenarios:

7.1 Evolution Scenarios of the DS

The creation of the DS-table *RETAIL_OUTLET* (*Id_Ro*, *Ro_name*, *Ro_zone*...) with a reference from the *SALES* DS-table to the *RETAIL_OUTLET* DS-table causes applying rule T2D that creates a dimension *D_RETAIL_OUTLET* linked to the *F_SALE* fact.

Adding the DS-table *SCORE_PROD* (*#Id_Prod*, *#Id_Cust*, *ScoreNumeric*...) referencing tables *PRODUCT* and *CUSTOMER* has caused applying rule T2F that creates the fact *F_SCORE_PROD* referring dimensions *D_CUSTOMER* and *D_PRODUCT* and having *Score* as a measure.

7.2 Evolution Scenarios Due to OLAP Requirements

Assume the decision-maker wants to analyze the Sales by *Category* (analysis parameter) of products. The *Category* is in the DS but not in the multidimensional model. To do so, he gives a rule indicating that the last digit of the product identifier (*Id_Prod*) codifies the *Category* of the product. Because of this evolution in requirement, a new parameter “*Category*” is created within a new hierarchy $Id_Prod \rightarrow Category$ for the *D_PRODUCT* dimension.

Suppose the decision-maker needs to analyze The Sales by product provider. The DW does not exist in the DW but the *Provider* table exists in the DS. The prototype creates a new parameter *Id_Prov* within a new hierarchy $Id_Prod \rightarrow Id_Prov$ for the *D_PRODUCT* dimension.

Actually, DWE offers the DW administrator the ability to graphically view the changes suggested on the DW model, adjust these changes, and automatically generate the DW alteration script. This allows a considerable gain in terms of quality and time. Further experiments are in progress to improve the quality of the propagations obtained, for example, the systematic addition of any weak attributes to be associated with a new inserted parameter.

8 Conclusion

In this paper, we have proposed a model-driven based approach in order to automate the propagation of the evolution of OLAP-requirements and the data source model towards its associated data warehouse. To do so we have defined three evolution models: DSEM (DS Evolution Model), REM (Requirement Evolution Model) and DWEM (DW Evolution Model). Furthermore, we have defined a set of transformation rules and formalized them in QVT (Query/View/Transformation) language; these rules implement the transformation process for the passage between these models; they support the propagation of changes due to changes occurred in the data source or to new OLAP-requirements.

In order to validate our approach, we have developed a software prototype called *DWE* (Data Warehouse Evolution). *DWE* is compliant to the Model Driven Approach. Moreover, we have presented the functional architecture of *DWE* based on two levels of transformations. The first is Model-to-Model (M2M) which transforms the DS and the requirements evolution data-models into a DW evolution data-model. The second transformation is Model-To-Text (M2T), which generates the script for the DW alteration using *Acceleo* templates that we have defined for generating OMB (Oracle MetaBase) code. The execution of this template allows log in to Oracle Warehouse Builder and executing the OMB scripts that alter the DW data-model.

Our *DWE* prototype differs from the literature solutions mainly because it provides (semi-)automatic propagation of evolutions applied to the OLAP-requirements and DS data-model towards the DW data-model. Indeed, *DWE* covers the whole cycle of the DW evolution starting from the identification of the DW evolutions and extends to code generation. Additionally, being MDA-based, *DWE* allows benefits offered by this

technology (i.e. independence of platforms, reduction of efforts, reuse of models, and improvement of the quality of result).

This work is currently opening up many perspectives. As a further step, we intend to study the effect of such evolutions on the ETL (Extract-Transform-Load) process. Obviously, the ETL process must evolve to consider the effects of the DS-DW changes on the existing loading procedures. We are also planning a case study for efficiency measurement and performance evaluation of the transformation rules.

References

1. Kimball, R., Ross, M.: *The Data Warehouse Toolkit*, 2nd edn. Wiley, New York (2002)
2. Golfarelli, M., Rizzi, S., Vrdoljak, B.: Data warehouse design from XML sources. In: *Proceedings of ACM International Workshop on Data Warehousing and OLAP (DOLAP 2001)*, Atlanta, GA, USA, pp. 40–47 (2001)
3. Rusu, L.I., Rahayu, W., Taniar, D.: A methodology for building XML DW. *Int. J. Data Warehous. Min.* **1**(2), 67–92 (2005)
4. Nabli, A., Soussi, A., Feki, J., Ben Abdallah, H., Gargouri, F.: Towards an automatic data warehouse and data mart design. In: *7th International Conference on Enterprise Information Systems (ICEIS 2005)*, Miami, USA, pp. 226–231 (2005)
5. Rundensteiner, E.A., Nica, A., Lee, A.J.: On preserving views in evolving environments. In: *The 4th International Workshop Knowledge Representation Meets Databases*, pp. 131–141 (1997)
6. Bellahsene, Z.: Schema evolution in data warehouses. *Knowl. Inf. Syst.* **4**(3), 283–304 (2002)
7. Thakur, G., Gosain, A.: A comprehensive analysis of materialized views in a data warehouse environment. *Int. J. Adv. Comput. Sci. Appl. (IJACSA)* **2**(5), 76–82 (2011)
8. Papastefanatos, G., Vassiliadis, P., Simitsis, A., Sellis, T., Vassiliou, Y.: Rule-based management of schema changes at ETL sources. In: *Grundspenkis, J., Kirikova, M., Manolopoulos, Y., Novickis, L. (eds.) ADBIS 2009. LNCS, vol. 5968*, pp. 55–62. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12082-4_8
9. El Akkaoui, Z., Zimanyi, E., Mazón, J.N., Trujillo, J.: A model-driven framework for ETL process development. In: *Proceedings of the ACM 14th International Workshop on Data Warehousing and OLAP (DOLAP 2011)*, New York, USA, pp. 45–52 (2011)
10. Wrembel, R., Bębel, B.: Metadata management in a multiversion data warehouse. In: *Spaccapietra, S., et al. (eds.) Journal on Data Semantics VIII. LNCS, vol. 4380*, pp. 118–157. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-70664-9_5
11. Favre, C., Bentayeb, F., Boussaid, O.: Dimension hierarchies updates in data warehouses: a user-driven approach. In: *9th International Conference on Enterprise Information Systems (ICEIS 2007)*, Madeira, Portugal, pp. 206–211 (2007)
12. Thakur, G., Gosain, A.: DWEVOLVE: a requirement based framework for DW evolution. *SIGSOFT Softw. Eng. Notes* **36**(6), 1–8 (2011)
13. Solodovnikova, D., Niedrite, L., Kozmina, N.: Handling evolving data warehouse requirements. In: *Morzy, T., Valduriez, P., Bellatreche, L. (eds.) ADBIS 2015. CCIS, vol. 539*, pp. 334–345. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23201-0_35
14. Object Management Group (OMG): *Model Driven Architecture (MDA)* (2004)
15. Taktak, S., Feki, J., Zurfluh, G.: Toward evolution models for data warehouses. In: *2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2014)*, Lisbon, Portugal, pp. 472–479 (2014)

16. Bellatreche, L., Wrembel, R.: Evolution and versioning in semantic data integration systems. *J. Data Semant.* **2**, 57–59 (2013)
17. Taktak S., Alshomrani S., Feki J., Zurfluh G.: The power of a model-driven approach to handle evolving data warehouse requirements. In: *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2017)*, pp. 169–181 (2017). ISBN 978-989-758-210-3
18. Hachaichi, Y., Feki, J., Ben-Abdallah, H.: Designing data marts from XML and relational data sources. In: *Design and Advanced Engineering Applications: Methods for Complex Construction. Advances in Data Warehousing and Mining Series*, pp. 55–80. IGI Global (2009). Bellatreche Edition
19. Taktak, S., Alshomrani, S., Feki, J., Zurfluh, G.: An MDA approach for the evolution of data warehouses. *Int. J. Decis. Support Syst. Technol. (IJDSST)* **7**(3), 65–89 (2015)
20. Golfarelli, M., Maio, D., Rizzi, S.: The dimensional fact model: a conceptual model for data warehouses. *Int. J. Coop. Inf. Syst.* **7**(2–3), 215–247 (1998)
21. Hachaichi, Y., Feki, J.: An automatic method for the design of multidimensional schemas from object oriented databases. *Int. J. Inf. Technol. Decis. Mak.* **12**(06), 1223–1259 (2013)
22. Object Management Group (OMG): MOF Model to Text Transformation Language, v1.0 (2008). <http://www.omg.org/spec/MOFM2T/1.0/>



Complex Event Processing for User-Centric Management of IoT Systems

Moussa Amrani¹, Fabian Gilson²(✉), and Vincent Englebert¹

¹ PRcISE Research Center, University of Namur, Namur, Belgium
{moussa.amrani,vincent.englebert}@unamur.be

² Computer Science and Software Engineering, University of Canterbury,
Christchurch, New Zealand
fabian.gilson@canterbury.ac.nz

Abstract. The amount of available connectible devices and Internet of Things (IoT) solutions is increasing as such equipments are becoming popular and widely available on the market. This growth in popularity goes together with a keen interest for *smart homes* where individuals deploy *ad hoc* solutions in their houses. However, the task to translate the users' needs into a concrete IoT infrastructure is not straightforward and often require to deal with proprietary APIs, complex interconnection protocols, and various technical details, so that the link to user requirements may be lost, hampering the validity of their interaction properties. In order to define and manipulate devices deployed in domestic environments, we propose *IoTDSL*, a Domain-Specific Language relying on a high-level rule-based language. Users in charge of the deployment of IoT infrastructures are able to describe and combine in a declarative manner structural configurations as well as event-based semantics for devices. Modellers are then freed from technical aspects, playing with high-level representations of devices. The events orchestration is transferred to a dedicated component where high-level rules are automatically translated into a Complex Event Processing (CEP) facility meant to evaluate and trigger runtime events. Additionally, simulation code can be generated to play with user-defined configurations.

1 Introduction

Facing the explosion of available connected devices, many vendors are jumping into the market, proposing a large spectrum of products ranging from connected devices to associated end-user services [16]. This results in a wide heterogeneity in software and hardware implementations, as well as an ever growing list of concerns and opportunities in terms of interoperability, data management, privacy and scalability [8].

As the Internet of Things (IoT) infiltrates many aspects of people's life through their cars, homes or business buildings, phones and so forth, a critical challenge is to provide end-users the possibility to benefit from the plethora of connected devices and configure them for their particular needs. Such configurations should address needs captured by user-defined workflows, or scenarios,

that are unlikely to stay unchanged for long. Besides, the infinite possibilities offered by even simple combinations of a small number of devices already brings a combinatorial explosion that cannot be captured by configurations that remain static. One possible path towards mastering this complexity is to put the configuration tools directly in the end users' hands, so that they become in charge of managing the workflows they are interested in. This calls for a radical raise in the abstraction level devices are manipulated through, since end users cannot keep up with the ever-evolving IoT market. However, they usually feels comfortable with basic devices functionalities: a temperature sensor is supposed to capture the temperature, an alarm should buzz in case of emergency, a smart vent or thermostat is supposed to switch on at some predefined conditions, no matter how they communicate with the home, and no matter which vendor manufactured them. Hiding the underlying complexity of protocol communication, data exchange and technical APIs for device manipulation is a key enabler of large adoption of smart devices, especially in the context of smart homes.

Model-Driven Engineering (MDE) has been recognised during the last decade as a software engineering technique dedicated to the design, management and evolution of computer languages enabling automatic generation of production code, diverse types of analysis and early verifications [17]. In particular, Domain-Specific (Modelling) Languages (DSLs) allow straight manipulation of domain concepts, thus allowing experts to directly deal with notions they are familiar with. Over the years, the MDE community put an effort to automate many aspects necessary to facilitate the daily use of such languages, particularly by fading away the distinction between textual and visual language syntaxes, and by bringing appropriate tools to developers to simply and quickly design their own, new DSLs.

Following this trend, we introduce `IoTDSL`, a prototype Domain-Specific Language (DSL) meant to allow end users drive the IoT devices installed in their homes. `IoTDSL` has two major concepts at its heart. First, it promotes *separation of concerns* by properly distinguishing the phases an IoT system is composed of, namely capturing the devices capabilities, then deploying them through the house in an interconnected network, and defining scenarios to drive the overall system to achieve end users' goals. Second, it relies on *events* to describe devices capabilities and combine them into rule-based definitions for scenarios [10,21]. Rule-based systems are widely used in a vast range of domains like finance [27], disaster monitoring [7], social threats discovery [3] and so forth. Rules are particularly suitable to express composition of events because of their declarative nature and their high-level of abstraction, thus in `IoTDSL`, user scenarios are expressed in a rule-based language that empowers reusability and automatic translation into a runnable Complex Event Processing (CEP)-based language [11].

Compared to [2], we present in this extension paper a full compilation scheme for `IoTDSL`: any instance of the language could then be translated into an executable framework, namely TRex [12], that can be embedded in a middleware that orchestrate devices intercommunication by centralising information

delivered by sensors and taking the appropriate decisions to activate reactions according to end users scenarios.

Outline. We start in Sect. 2 by presenting an archetypal scenario of a smart house to highlight the usefulness to bring end-users back in control of their own domestic IoT environment. We also extract crucial IoT challenges specific to the use of DSLs and MDE techniques to realise this vision. In Sect. 3, we introduce `IoTDSL`, our prototype DSL to specify and interconnect devices in an intuitive and general way and illustrate its benefits through use cases extracted from our smart house example. Then, in Sect. 4, we detail how we translate `IoTDSL` rules into a concrete CEP engine and how we generate simulation facilities meant to test and validate the IoT deployment. We discuss our approach and the remaining challenges to tackle in Sect. 5. We overview in Sect. 6 the use of DSLs for IoT, comparing existing approaches with ours and assessing them against the challenges we identified. Finally, we conclude in Sect. 7 and present the main lines of work ahead to transform our prototype in a fully functional DSL framework.

2 Motivation and Challenges

A major issue for IoT is the rapid growth of the offer, spanning from very simple devices (e.g., temperature, light or sound sensors) to more elaborate objects that interact with their environment (e.g. building security or multimedia solutions). For domestic usage however, it is likely that devices would have simpler capabilities, while responding to various scenarios that are specific to their inhabitants. Therefore, taming the complexity of smart homes requires handling *interactions* between devices, rather than their own, specific capabilities.

Even in this context, simple devices could be manipulated in a plethora of scenarios: for example, a sensor that reports temperature values might be paired with a heating system that regulates rooms temperature and keeps them in comfortable ranges, where in other configurations, it might detect fire situations.

At technical level, driving such devices for realising end-users scenarios is hampered by, among others, the complexity of vendors and their proprietary data formats, the plethora of APIs that are quickly evolving, and the large variety of communication protocols. This overburden the work of IoT technicians when dealing with end user requirements. Besides the need for more standardization in IoT specifications, there is also a crucial need for abstract definition of the semantics of IoT configurations [24].

Our proposal consists of two facets. First, we clearly separate the responsibilities of *technicians* who deal with technical details relative to a specific solution deployed in a smart home, and *end users*, who control the devices according to their evolving needs. Second, we offer end users a way to interact with their home at an abstract level: far from knowing how each device works, users manipulate them through abstract events describing their interactions. To this end, we propose a Domain-Specific Language (DSL) to describe on the one hand, smart home configurations and on the other hand, events' intercommunications.

In this section, we show a typical smart home installation with affordable and simple devices. We then outline the main components of a DSL that captures IoT systems, and from such archetypal configuration, we list the main challenges such a DSL for IoT should address to effectively provide a viable solution.

2.1 Typical IoT Scenarios

Figure 1 describes a typical configuration at Alice's smart home, the fictional character we use in our case study. Alice asked her technician to deploy light sensors and bulbs to lighten the rooms, motion and door detectors to detect human activity, an alarm used in case of emergency and a toggle button placed in the balcony for security purposes.

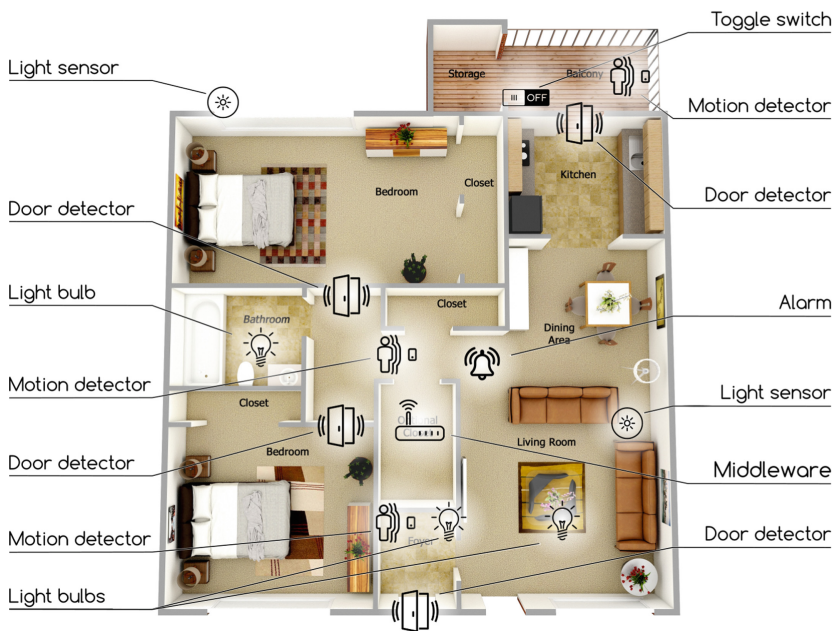


Fig. 1. Hypothetical Alice's *smart-home* configuration of IoT devices.

Alice is interested in simple scenarios for her comfort and her little boy's security: she wants the entrance lights to automatically switch on to welcome her when she arrives home. Also since her boy often plays in the balcony, or sometimes wakes up at night and walks through the apartment, she needs to ensure he does not fall or injure himself. Those scenarios may be met with her current equipment as we will illustrate in Sect. 3.3 with a set of rule-based specifications.

2.2 Challenges

Many challenges arise directly from the previous hypotheses, in order to provide a feasible, tractable and realistic IoT solution. Many contributions already investigated the various challenges IoT systems pose, but we revisit the literature in order to extract those directly relevant to the definition of DSLs for the IoT.

Capability Discovery. Providing the ability to drive interconnected devices assumes the capacity of automatically discovering devices' capabilities in a standardised and uniform way [8]. Similar processes exist for other technologies, like USB devices plugged into computers that automatically expose their natures and capabilities. Classifying those capabilities should be useful to build an ontology of normalised functions that could result in powerful APIs to manipulate devices. A DSL for IoT would then directly benefit from this kind of APIs to expose dynamically connected devices without any preliminary configuration step.

Reusability. Knowledge exchange and reusability of devices' definitions and interaction specifications are essential prerequisites to the adoption of a DSL for the IoT. It is not uncommon to reuse existing scenarios that involve a set of devices in different configurations. Those partial IoT structures with their event orchestrations should be *externalisable*, despite the large amount of standards, APIs or hardware [19].

Complex Event Processing (CEP). Letting end users deal with devices through their low-level capability interfaces could lead to confusion and stiff complexity for defining usage scenarios [18]. Rather, providing a way of reifying low-level device computations into high-level events could help end users leverage the complexity of devices networks and pave the way to manipulate them freely and transparently [12]. Since CEP consists of deriving meaningful conclusions from a stream of events occurring within a system and responding to them as quickly as possible, it provides a solution to extract meaningful events from low-level computations. However, for a solution to be complete and useful, the reverse operation should be addressed: high-level actions should be adequately translated into low-level actuations and interactions to link the high-level events manipulated by a DSL to the actual hardware infrastructure of devices.

Protocol Interoperability. A smart-home solution with heterogeneous devices would often integrate elements from various providers, thus communicating through disparate protocols. In order to make them interact efficiently without forcing end users to stick with one vendor, a powerful DSL should provide ways for interoperability over multiple communication protocols, without requiring end-users to understand the protocols' intricacies, versions and technical restrictions [14]. An adequate DSL for IoT should completely hide and automate this aspects, and would possibly rely on multi-protocol solutions like OpenRemote (<http://www.openremote.com>) or EnOcean (<https://www.enocean.com>), to name a few of them.

Scalability. As the number of application domains increases, the amount of connected devices is expected to rise exponentially. When updating existing

IoT configurations, current solutions may not collapse when adding more elements [22]. Furthermore, a DSL must provide a way to absorb scalability problems, hiding as much as possible purely technical constraints regarding increases in size and complexity of operating configurations.

Data Management. Analogously to scalability issues, the massive increase in connected devices will produce more and more data to be processed, stored and, for some of them, post processed [16]. More data means seemingly more storage capabilities and the required space to handle such flow of information will be at its highest ever. Furthermore, the multiplication of available (sensors) sources is creating a whole new world of data processing and mining possibilities, but also a profusion of divergent concrete data types that sooner or later must be mapped to equivalent concepts. Although the kind of DSL we are targeting comes earlier in this general scheme, it would eventually benefit from the knowledge acquired by mining the collected data to drive and guide users through their daily scenarios.

Non-functional Properties. A powerful DSL should encompass typical non-functional properties of device networks to ensure long-life and secure realisation of scenarios. *Performance* is crucial, and depends both on the devices capabilities but also on the quality of the communication network. *Resource availability*, both in terms of computation and memory capability, but also in terms of energy, is another crucial bottleneck for the adoption of DSLs as a solution for defining scenarios. The generated code from the DSL should not overload the devices with repetitive communications or unnecessary computations that would drain the device's battery. *Security* is yet another concern with respect to two aspects. First, sensitive data could be exposed through the communication network, endangering users privacy. Second, some functionalities could be locked and only accessible to authorised users [30].

This challenges will be revisited in Sect. 5 in the light of our proposal, to discuss which and to what extent our DSL offers potential solutions.

2.3 Components for an IoT Language

We argue that a good way of capturing the many variations of scenarios relying on a specific IoT system deployed at home would consist in offering end users, i.e. home inhabitants like Alice, and technicians in charge of configuring such systems and effectively deploying them, a DSL that provides at least the following components:

Device Description. We need facility to make a precise inventory of the devices used in a specific deployment as well as the high-level capabilities of these devices, described in terms that are immediately understandable by end-users, as opposed to conveying technical details about how those devices precisely operate;

Network Description. A way to capture where each device is located and how it is possible to communicate with it, in order to receive or send data to it;

Dynamics. A way to describe the interactions wished by end-users, *i.e.* how to leverage the functionalities of the devices to effectively realise one or several scenarios that are convenient for the end-users.

Those components are obviously not sufficient to obtain a fully-fledged solution that becomes adaptable to any situation, but they still represent necessary steps to provide end-users the capacity to manipulate a collection of devices without relying on specific technologies. Defining such DSLs should encompass a series of facilities dedicated to hide hardware- and protocol-related constraints, and high-level models of devices should somehow be easily *transformable* and *traceable* into concrete infrastructures with simulation and verification possibilities.

3 IoTDSL

Based on the challenges identified in Sect. 2, we now introduce `IoTDSL`, our DSL devoted to facilitate the high-level manipulation of IoT systems. At the heart of `IoTDSL` are two governing principles. First, we promote a clean separation of concerns for all aspects the DSL has to handle, by specifying one sublanguage for each concern. We believe this approach to be scalable, and to support independent evolutions of each concern without impacting the other aspects, since those aspects are composed through well-defined interfaces. Second, our DSL relies on events, a natural paradigm for specifying various models of interactions that is widely used in embedded and critical systems, and where a clear separation between the system and its environment is performed, further empowering the separation of concerns. Despite its early stage of development, `IoTDSL` shows its ability to capture the definition of small-scale IoT systems appropriately.

Building a well-calibrated DSL is known to be difficult and error-prone. It usually requires a broad expertise of the domain under consideration before a consensus emerges on the domain's key concepts and how to effectively represent them. Fortunately, MDE technologies operated substantial breakthrough over the past decade, allowing language designers to define their own DSL structures and user interfaces more easily. Adopting such a trend, we have built an early prototype for our DSL under GeMoC [5, <http://gemoc.org>], a MDE framework that supports both visual and textual representations as concrete syntaxes and maintains a full synchronisation between them. Since we are at early development stage, only a textual syntax is currently available to modellers, but other syntaxes, even graphical ones, can be smoothly added thanks to *GeMoc*.

To illustrate our proposal, we show how `IoTDSL` is built by describing each sublanguage, following the DSL components identified in Sect. 2.3, and illustrate them by providing the full implementation for Alice's apartment as depicted in Sect. 2.1.

3.1 Type Definition

The first task is to provide a description of which capabilities each device possess, how each device may provide information about the environment through

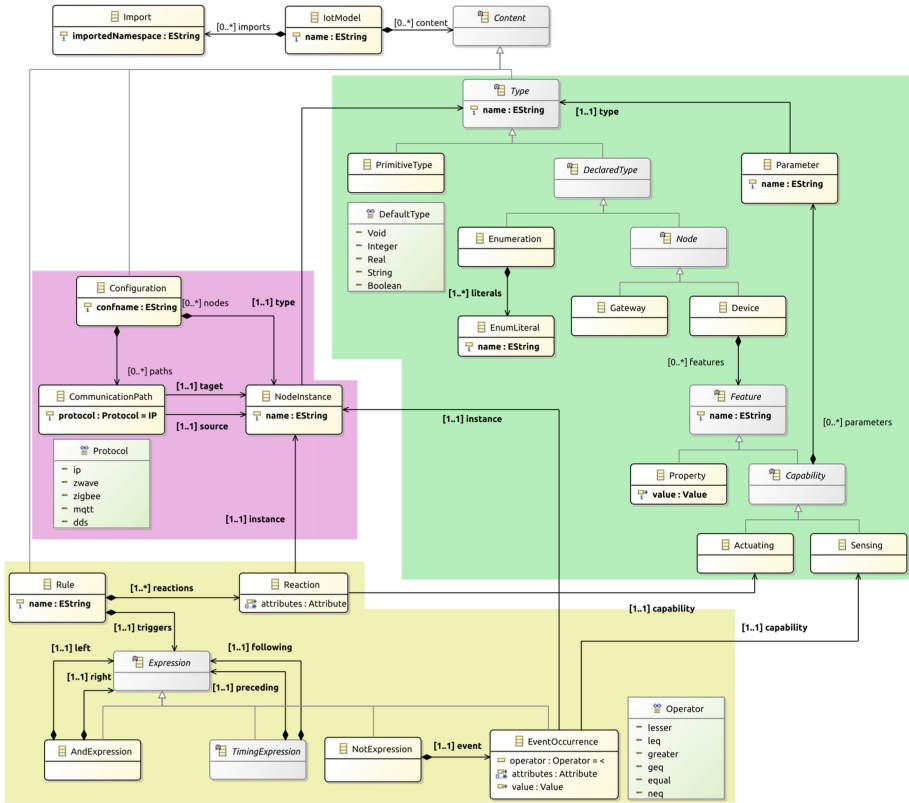


Fig. 2. Metamodel of IoTDSL, separated in three concerns: *Type Definition* captures devices’ capabilities (top-right green part), *Network Configuration* details how device instances are connected to each others (middle-left purple part), *Business Rules* defines the functionalities expected from the IoT installation (bottom yellow part). (Color figure online)

a *sensing* operation, and how it could react and influence it through *actuations*. Our framework currently requires that an advanced user extract relevant information regarding devices’ exposed features, but it is flexible enough to accommodate automation in the future, so that such pieces of information could be automatically extracted from pre-existing devices databases (either from a knowledge database the IoT system is connected to, or from a library of *off-the-shelf devices*).

The concepts dedicated to type definition are shown in Fig. 2 (top-right part in green background). This part is similar to the notion of Classifier in MOF-like languages: a *Type* is either a *PrimitiveType*, or a user-defined *DeclaredType*. We distinguish between general *Gateways*, which centralise information and processing, and *Nodes* deployed in the environment and communicating with *Gateways*, and which possess capabilities to interact with the environment. A *Capability* is

basically a parametrised event that drives the node to either capture data from the environment, act on it, or perform both. This abstract view of a “thing” allows us to manipulate any device at a high level of abstraction, exhibiting a clean and uniform interface for end-users based on device capabilities. Since Nodes are Types themselves, they may be referenced as parameters for the purpose of dynamic discovery across devices.

Listing 1 illustrates how the devices in Fig. 1 are declared in `IoTDSL`. Each device is introduced by the keyword `device`, possesses a name and lists capabilities that correspond to reporting events (`sensing`) or operating over the environment (`actuating`).

Table 1. Type declarations in `IoTDSL`: capabilities as high-level events.

1	<code>gateway</code> Middleware	12	<code>device</code> LightSensor {
2	<code>device</code> DoorDetector {	13	<code>sensing</code> light()
3	<code>sensing</code> opened()	14	}
4	<code>sensing</code> closed()	15	<code>device</code> LightBulb {
5	}	16	<code>actuating</code> on()
6	<code>device</code> MotionDetector {	17	<code>actuating</code> off()
7	<code>sensing</code> moving()	18	}
8	}	19	<code>device</code> Alarm {
9	<code>device</code> ToggleSwitch {	20	<code>actuating</code> sound()
10	<code>sensing</code> toggled()	21	}
11	}		

Any IoT system should declare a special device, introduced with the keyword `gateway`, that centralises data from all devices connected to it, as we will show in Sect. 3.2. This device will be responsible of the event orchestration and will host the CEP engine that embeds the implementation of the business rules. Also note that the above model is the *user-defined* part of `IoTDSL`. In the background, abstract events attached to all devices will need to be mapped to concrete low-level APIs events using a dedicated mapping language that is out of the scope of this paper.

3.2 Network Configuration

The configuration constructs of `IoTDSL` are specified in the middle-left purple part of Fig. 2. Since we use an architecture centralised around gateways, a network Configuration is a graph-like structure where vertices are Gateways and NodeInstances (so that instances may communicate with each others), while edges represent CommunicationPaths (or channels). Such paths define, among others, one or more protocols used to interact. We actually rely on existing platforms, such as OpenRemote (<http://www.openremote.org>) or SmartThings (<https://www.smartthings.com>) to handle the intricate details of the protocols since such details are, from an end-user point of view, technical aspects rather than essential matters of the configuration itself. By knowing which protocols are used between each pair of devices, we can automatically perform data conversion in

the proper format required by the protocols: most of those protocols are already implemented in *General-Purpose Programming Languages* (GPLs), like Java or C.

Listing 2 shows an instantiation as well as the connection that conforms to the types given in Listing 1 and the configuration presented in Fig. 1.

Table 2. Network configuration in IoTDSL for our smart house.

1	configuration SmartHouse {		
2	node middle : Middleware	17	from alarm to middle via IP
3	node alarm : Alarm	18	from frontDoor to middle via IP
4	node toggle : ToggleSwitch	19	from parentDoor to middle via IP
5	node frontDoor : DoorDetector	20	from childDoor to middle via IP
6	node parentDoor : DoorDetector	21	from balconyDoor to middle via IP
7	node childDoor : DoorDetector	22	from outLight to middle via IP
8	node balconyDoor : DoorDetector	23	from livingLight to middle via IP
9	node outLight : LightSensor	24	from livingBulb to middle via IP
10	node livingLight : LightSensor	25	from bathroomBulb to middle via IP
11	node livingBulb : LightBulb	26	from foyerBulb to middle via IP
12	node bathroomBulb : LightBulb	27	from balconyMotion to middle via IP
13	node foyerBulb : LightBulb	28	from foyerMotion to middle via IP
14	node balconyMotion : MotionDetector	29	from hallMotion to middle via IP
15	node foyerMotion : MotionDetector	30	}
16	node hallMotion : MotionDetector		

A specific device is considered as an instance of a defined type such that particular devices with the same set of capabilities may be distinguished via identifiable unique references. Communications are purely declarative and only mention the protocol type (introduced by the **via** keyword). In our example, we simply decided to use an IP protocol for all bindings. Note that a similar mapping process that the one described at the end of Sect. 3.1 is required to reify abstract connections between NodeInstances to physical ports and protocols, but again, these mapping statements are outside of the scope of this paper.

3.3 Business Rules

Business rules are the core of the manipulation of IoT systems and compose the third part of IoTDSL as detailed in the bottom yellow part of Fig. 2. This last sub-language relies on an event-based framework that allows to specify a set of Rules expressing the many functionalities an end-user wants to achieve in his/her concrete configuration.

An IoTDSL Business Rule is identified by the keyword **rule** followed by an unique identifier, and a body of the form «**when trigger do reaction**». Rules' **triggers** are cyclically evaluated against the surrounding environment and specify the conditions under which the corresponding **reactions** have to be performed to realise the end-users' scenarios. A **reaction** defines actuations on the IoT system to send or require data of identified devices, or issues events that are internally used to synchronise rules.

Our approach is currently purely middleware-oriented: all rules are evaluated inside a single gateway that supposedly possesses enough processing power. We leave as future work the exploration of parallelisation techniques to support multiple gateways that communicate appropriately, or the possibility to decentralise parts of the computation into nodes with sufficient processing and power resources to optimise resource consumption and lighten communication exchanges.

We now illustrate how the scenarios Alice is concerned about (cf. Sect. 2.1) can be translated into business rules in `IoTDSL` with the devices' definitions detailed in Listings 1 and 2.

Switching Entrance Lights On when Coming In. When Alice gets home (and thus opens the front door), she wants the lights to be automatically switched on in the foyer and in the living room.

```

1 rule SwitchLightsWhenEntering:
2   when (foyerMotion.moving after frontDoor.opened) do {
3     foyerBulb.on
4     livingBulb.on
5   }
```

Listing 1.1. Rule to switch on the lights at home incoming.

This rule introduces what we call *facilitators*, i.e. keywords that define an unspecified time window in which a sequence of events should be observed. This time window is system-specific and needs to be defined independently in configuration files independent of descriptions in `IoTDSL`. In this case, the `foyerMotion` should detect movement *nearly after* the `frontDoor` detects an opening.

Note that reactions are defined as a sequence that does not matter: the order in which the `foyerBulb` and the `livingBulb` switch on largely depends on the platform capacities, i.e. they can be actuated synchronously or sequentially (in which case, no guarantee is given that the definition order will be respected). At the abstraction level `IoTDSL` operates, it is irrelevant since the end user wishes to see both switched on at some point, without having to consider low-level details that would enforce such behaviour.

Illuminate bathroom when children wake up at night. When Alice's little boy wakes up at night, she would like to have the light in the bathroom to be switched on to prevent him from falling or injuring himself. Analogously, she wants the light to be switched off when he gets back to sleep afterwards.

```

1 rule SwitchBathroomLightOnAtNight:
2   when (not livingLight.light and
3     (hallMotion.moving after childDoor.opened)) do {
4     bathroomBulb.on
5   }
6
7 rule SwitchBathroomLightOffAtNight:
8   when (not hallMotion.moving within 3 min from childDoor.closed) do {
9     bathroomBulb.off
10  }
```

Listing 1.2. Rules to switch on/off lights in the corridor at night.

The rule `SwitchBathroomLightOnAtNight` introduces a new keyword `not`, which represents the *absence* of a certain event type, here `livingLight.light`. This is different than simply observing some events occurring. Note also that the second part of the rule trigger uses parenthesis to relate the facilitator `after` to the closest event `childDoor.open`, instead of spanning on the whole condition.

The rule `SwitchBathroomLightOffAtNight` presents a combination of negation with an explicit time window with the construct `within ... from`: it indicates that no event of type `movement` from the hall motion sensor should occur in a three-minute time window after observing the `closed` event from the boy's door, in order to trigger the rule. `IoTDSL` defines several useful time units to cope with simpler definitions (seconds, minutes, hours, or a combination of the three).

Report Unsupervised Children on Balcony. Alice considers that it is a critical situation if a child enters into the balcony without her knowledge, because of fall risks. To avoid that, she placed a switch button high enough that only an adult could press when accompanying a child outside. If the button is not pressed within 3s after someone enters the balcony, an alarm should sound.

```

1 rule AlarmWhenChildOnBalcony:
2   when (not toggle.toggled within 5 sec from
3     (balconyMotion.moving after balconyDoor.opened)) do {
4     alarm.sound
5   }
```

Listing 1.3. Rules to sound the alarm in case of an unsupervised child on the balcony.

This last rule states that once the balcony door has been opened and movements are detected on the balcony, the alarm should sound unless the toggle button is pressed in a five-second time window. This rule is similar to `SwitchBathroomLightOffAtNight`, except that the baseline of the time window is here a composite event using a facilitator: once an opening followed by movements on the balcony is observed, we expect the toggle button to be pressed.

To summarise, an end-user uses the Business Rules sublanguage to specify the scenarios of interest in the form of `when (trigger)do {actuactions}`: the `trigger` condition specifies the event (non-) occurrence pattern under which the `actuactions` are performed, by using common boolean connectors as well as time windows to observe delayed events; whereas the `actuactions` are undeterministically performed independently to their definition order.

From a qualitative point of view, adopting a rule-based language presents the advantage of mimicking the cognitive process of establishing a scenario, which should ease the adoption of `IoTDSL`. However, we are conscious that this requires a further examination and actual validation with end-users that are not aware of the underlying DSL mechanisms, but we believe that presenting a visual representation for rules and powerful analysis of rule activation could ease the adoption process and facilitate scenario definitions.

4 General Architecture and Code Generation

Fundamentally, IoTDSL describes a real-time reactive system: information is regularly reported to the middleware where it is processed in order to react on the environment. However, in the context of IoT systems, the environment cannot be controlled, but it is perceived through the many deployed devices, that allow at the same time to react on it. The key task is then to process events quickly enough to ensure appropriate reactions, although it is not critical to react in a precise timeframe. In this section, we first describe the general architecture of our tool as well as the technical choices for processing events in an IoT system, then provide a compilation schema to translate IoTDSL business rules into Tesla rules, the entry language of TRex [12], the CEP engine we have chosen in our architecture.

4.1 General Architecture

Our proposal relies on a middleware that embeds a CEP engine for handling event processing, as depicted in Fig. 3. Our tool offers a simulation mode, where devices are simulated as software components mimicking their actual execution, thus allowing to test IoT scenarios without physical devices.

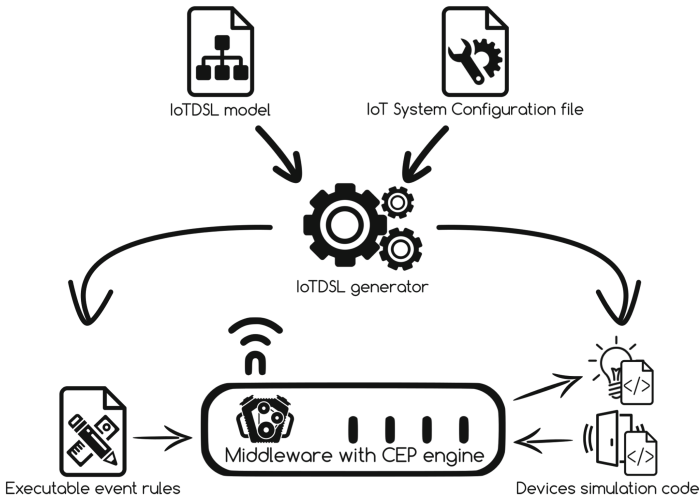


Fig. 3. General architecture of IoTDSL framework.

The central element is the automatic code generator process: it produces executable code from IoTDSL models and configuration files that define platform-specific details on the execution timing of devices (that the technicians set once and for all), and, for the purpose of our prototype, simulation code used to emulate the behaviour of the devices. The executable code is deployed into the

CEP engine running on a middleware, which reuses in simulation mode the code for communicating with the (abstract) devices.

We are currently investigating how business rules could be broken down into smaller clusters that might be deployed directly into devices, assuming they offer sufficient battery and computation power. When addressing non-functional properties of devices, we envision the possibility of expressing technical specifications relative to the energy consumption and computation capabilities in a separate file that `IoTDSL` would take into consideration to identify distributable rules.

4.2 TRex as CEP Engine

To enable efficient event processing from distributed connected things, we rely on TRex, a powerful and highly optimised CEP engine developed by Cugola and Margara [12]. TRex relies on Tesla [11], an entry language that is expressive enough to address most of the necessary patterns for capturing complex events definitions. As a consequence, the expression language used for `IoTDSL` TRIGGERS is directly inspired from Tesla. This allows us to offer end-users the expressibility they need, while at the same time simplifying the translation of `IoTDSL` rules into Tesla rules.

TRex offers a queueing mechanism to overcome bursts of incoming events: when deploying the IoT system on site, it becomes possible to customise the queue size, thus balancing between event loss and treatment latency.

TRex is conveniently organised as a Client/Server architecture in a «*publish / subscribe*» way, and relies on the Tesla language to define the necessary components: event *notifications* (or *occurrences*, or simply events); *subscriptions* and *rules*. TRex, as the CEP engine, permanently receives event notifications from sources, and redistribute them to subscribers. An event notification is produced by a source and sent to the CEP system, and is typed by an event type that possesses typed attributes: for example, `Temp@100(location = "Living", value = 25.0)` defines a notification of type `Temp` with two attributes and a timestamp (noted after the `@`) that records the moment in time the notification is produced. A subscription by sinks (or, event consumers) is submitted to the CEP system in order to receive notifications that things happened: for example, `Subscribe(Temp, location = "Living" and value > 20)` indicates a subscription to `Temp` notification that matches the filtering condition on its location and value. Tesla rules define complex events from simpler ones, which may emanate from actual sources, or be complex events defined by other rules, thus leading to a hierarchy of events.

A rule has the following form:

```
define CE(attr1 : Type1, ..., attrN : TypeN)
from Pattern
where attr1 := f1, ..., attrN := fN
```

Intuitively, a rule defines a complex event `CE` together with its signature (i.e. an ordered list of typed attributes) that issues `CE` notifications

whenever the `Pattern` is matched, assigning values to CE attributes from the functions defined in the `where` clause (that may depend on elements of the `Pattern`). Valid patterns include event *occurrences* that filter attribute values (e.g., `from Temp.val > 20`), event *compositions* that combine events together with boolean operators or time windows (e.g., `from Rain and Temp.val > 20 within 5 min from Smoke`, indicating that a temperature reading above 20°C should occur within five minutes from a `Smoke` notification, while raining); event *negation* (e.g., `from not Rain between Temp and Smoke`, indicating that it should not rain between an elevated temperature reading and a smoke notification). Tesla proposes more powerful patterns like aggregation and iterations, but we will not use them in our examples. Note that the server is interactive so that clients can (un-)subscribe while the engine is running and rules may be added or deleted at runtime without affecting the overall infrastructure.

From the perspective of `IoTDSL`, `TRex` offers several benefits as a CEP engine. `TRex` is powerful enough to handle typical IoT scenarios like the one described in Sect. 2, thanks to the expressive power of Tesla. It adopts a decentralised architecture that directly reflects our design choices, and supports distributed processing of events to reduce the cost of communication and to optimise resource usage. It is developed in C, so it is even suitable for *small form factor* middlewares. Finally, on top of an API written in C, Java libraries have been developed on which we rely to generate devices' simulation code.

4.3 Compiling `IoTDSL` Rules

We now describe how we obtain the final code deployed into the gateway using `TRex`, and illustrate our compilation scheme on the Business Rules of Sect. 3.3.

In `IoTDSL`, a Business Rule has the following general form: `rule R: when (trigger)do reaction`. The scheme for producing Tesla code relies on a three-step process:

1. For traceability purposes, we map each rule name `R` to the resulting Tesla rule(s), to be able to trace results directly from `IoTDSL` rules in future analysis.
2. The second step requires a preliminary task: since Tesla does not handle instances (in the form of our dot-like notation for events), we need to add a predefined attribute `E(_inst : Instance)` to each event `E` used in `IoTDSL` Business Rules, and link the type `Instance` to simple strings; unicity of instance names is ensured by `IoTDSL` while checking Network Configurations. The rest of the second step depends on the nature of the `reaction`:
 - (a) If it is not a composite, i.e. the rule consists of only one actuation of the form `inst.actuation(<param1, ..., paramN>)`, we translate it in a simple rule of the form

```
define actuation(param1, ..., paramN)
from transformedTrigger
where param1 := f1, ..., paramN := fN
```

where the actuation parameters are computed in the `where` line, and the `trigger` condition is transformed into `transformedTrigger` with the process in Step 3.

- (b) If the `reaction` is composite, i.e. it consists of several actions `a1, ..., aN`, we issue $n+1$ Tesla rules: one rule for each actuation `aI`, and one additional rule to bind things together.

```

define R(Rparam1, ..., RparamM)
from transformedTrigger
where Rparam1 := g1, ..., RparamM := gn
...
define aI(AIparam1, ..., AIparamN)
from R(Rparam1, ..., RparamM)
where AIparam1 := f1, ..., AIparamN := fn
...

```

When the event pattern for rule `R` is detected, the first rule issues the complex event `R`, which is immediately produced by the CEP engine to trigger the subsequent rules. This way, all actuations are processed at the same time, leaving the platform dealing with the actuations. Note that in the additional Tesla rule `R`, no `_inst` parameter appears (as it is not needed), but all parameters necessary to the n actuations are part of event `R`'s signature, so that functions `f1, ..., fN` rematches the parameters of each actuation (`aIparamK`) correctly from `R`'s parameters.

3. So there remains to compute the `transformedTrigger` appearing in Tesla rules, which depends on whether facilitators (like `after` or `before`) have been used. In the absence of facilitators, the transformation is straightforward since the same expressions are natively available in Tesla. Otherwise, we rely on an external configuration file that describe the expected latency delays specific to devices and their communication paths, to translate such triggers into appropriate time windows. Producing this file is not the responsibility of end-users, since it rather belongs to knowledge pertaining to IoT system installation and deployment.

Let us now apply this compilation scheme to the three Business Rules described in Sect. 3.3. Rule `SwitchBathroomLightOffAtNight` is the simplest one: it contains only one actuation, and its trigger has a regular Tesla time window expression. Applying our compilation scheme results in the following Tesla rule:

```

define Off(_inst : Instance)
from not Movement(_inst = hallMotion) within 3 min from Closed(_inst = childDoor)
where _inst = bathroomBulb

```

Note that all event names are capitalised to cope with Tesla entry language, and that the `from` clause only binds the `_inst` attributes to their respective instances in the source `IoTDSL` rule.

Rule `SwitchLightsWhenEntering` is a good illustration of `IoTDSL` rules with multiple actuations. Applying the compilation scheme results in three rules, one for each actuation, and an additional one that glues things together.

```

define SwitchLightsWhenEntering
from Moving(_inst = foyerMotion) within 10 sec from Opened(_inst = frontDoor)

```

```

define On(_inst : Instance)
from SwitchLightsWhenEntering
where _inst = foyerBulb

define On(_inst : Instance)
from SwitchLightsWhenEntering
where _inst = livingBulb

```

The additional rule defining the `SwitchLightsWhenEntering` event does not define an `_inst` attribute, and converts the event facilitator `after` into a time window from a predefined value (*i.e.* 10s in this case). The two other rules originate from the actuators that have the particularity to activate the same event on two different devices, namely `foyerBulb` and `livingBulb`: this results in the same rule with different bindings for `_inst`.

Rule `SwitchBathroomLightOnAtNight` is the trickiest one as it combines negation outside a time window. Applying the compilation scheme results in only one rule, since there is only one actuation, but the trigger condition is more complicated than the first rule. Since Tesla does not support negation operators outside of time windows, we need to integrate it inside one. Intuitively, the starting point of this scenario is the opening of the door room: at this point, the bathroom light should be switched on if movements are detected shortly after and there is currently no light in the living room. Such trigger patterns are detected and refactored as follows:

```

define On(_inst : Instance)
from (Light(_inst = livingLight) within 2 sec from Opened(_inst = childDoor))
and
(Moving(_inst = hallMotion) within 10 sec from Opened(_inst = childDoor))
where _inst = bathroomBulb

```

5 Discussion and Remaining Challenges

The `IoTDSL` framework has been designed to empower non-experts with facilities regarding the requirements of *smart home* IoT solutions. Coupled to IoT devices and network specifications, the framework offers a rule-based language compilable into a concrete CEP infrastructure in charge of event orchestration. `IoTDSL` users are then able to describe their own configurations and needs in terms of conditional events and reactions to the manifestation of such conditions.

Tracing back to the features and challenges identified in Sect. 2.2, `IoTDSL` currently covers most of these aspects. We created a description language that captures devices capabilities at a high-level of abstraction, describing them as entities that produce and consume (typed) events, relieving end users from acquiring the low-level knowledge to manipulate them. Based on these specifications, users become able to represent smart home configurations easily by describing links between devices and declaring which protocols are used for communication from a set of predefined and widely used protocols. Device interactions are simply expressed as conditions triggering other events that may react on the environment, inducing physical actions on the real world. These three sublanguages defined in `IoTDSL` cover the language components identified as necessary for any DSL dedicated to IoT systems.

Based on the literature, we also identified seven important challenges an IoT modelling solution should tackle, most of them being considered in `IoTDSL`. Devices are seen through their high-level capabilities, which basically correspond to an ontological device description: it describes the *interface* of devices in a general way, facilitating *Capability Discovery* as it becomes available. By separating device descriptions from how they are connected to each others, `IoTDSL` empowers *Reusability* of devices through different IoT systems. Furthermore, since partial configurations may be defined and imported in `IoTDSL` models, partial definitions and behavioural specifications may also be shared between various installation.

We notably rely on a powerful CEP engine to handle event orchestrations and deliver physical actions through the system. Our framework processes abstract business rules and transforms them automatically into executable code and produces sample code for simulation purposes. The *scalability* issue is almost exclusively concentrated in the device intercommunication and the rule processing, making the CEP engine dealing with scalability issues. TRex, the engine we have chosen, already offers parallelisation mechanisms that we will leverage to divide monolithic solutions into smaller entities that would collaborate, assuming an IoT system becomes too large to handle within a single engine instance, so that part of the business rules might be deployed on distinct parts of the system to minimise the middleware workload. In turn, these could generate even more events through the network and result in a congestion with data transmissions. An appropriate tradeoff needs to be found between having more *one-to-one* communications or grouping more logics inside a particular node.

There is however three challenges that we scarcely target with `IoTDSL`, even if they are partially encompassed in some way, mostly because they represent orthogonal aspects or concerns that comes after the domain targeted by our language. `IoTDSL` targets the manipulation of device *interactions*, but already provides, through CEP, a limited paradigm for *Data Manipulation* that scales. However, as usual, a dedicated DSL could be more relevant since the operations required for such operations are somehow different. An interesting discussion would then to identify the interface needed to exchange information between data retrieved from device interactions and data processed offline with higher processing capabilities. For now, `IoTDSL` hides the intricacies of *protocol communication interoperability* by implementing simple connectors to each protocol we handle, and reusing existing infrastructures for protocols that we do not handle natively. However, it could be interesting to propose a generic infrastructure for protocol communication by separating the transport layer from the message representation. This is a specific expertise domain on its own that we will later tackle. For `IoTDSL` to handle *non-functional properties*, we first need to have precise description of the devices hardware properties, which is an active research domain on its own. From that, we could integrate information that would guide the automatic code generation process to specialise the code to either decentralise parts of the processing activities, ensure better performance or integrate best practices for security.

6 Related Work

A series of overviews have been recently conducted on several aspects of IoT. In [1, 31], the authors reviewed the applications, protocols and technologies used in the distinct IoT layers, while [14, 29] focused on architectural aspects and [30, 32] reviewed security ones. Most of these contributions identify a number of challenges crossing the application domain of a DSL for IoT, from which we identified the most relevant ones to our contribution in Sect. 2 and to which we confront our framework in Sect. 5.

Capturing variations of a domain with explicit constructs close to the domain concepts resides at the essence of DSLs. In that regard, many DSLs were proposed for various purposes in the IoT stack. CHARIOT [25] addresses Cyber-Physical Systems by providing a component model that clearly distinguishes between communication and computation, while ensuring resilience features in highly reconfigurable systems. In [6] is presented a DSL aimed at facilitating the deployment of applications, based on a component model of the environment used to locate the architecture nodes where business logic can be leveraged. ALPH [23] is a DSL for ubiquitous healthcare that focuses on three concerns: mobility, by helping users to manage frequent devices disconnections; context-awareness to adapt application behaviour to environmental changes; and infrastructure, for managing the heterogeneity of communication protocols. Midgar [13] offers a visual interface to support end-users in controlling interconnected devices and generate the glue application making these devices interoperate. In [26], the authors present a visual DSL for capturing the features and intercommunications of devices distributed in various application domains spanning from smart homes to patient monitoring. These contributions target different application domains at different abstraction levels, but possess every key features we identified in Sect. 2 in a more or less explicit way. Since `IoTDSL` targets end-users with no prior knowledge in programming, we contrast with these contributions by offering a more intuitive, declarative style for expressing the system's dynamics through semantics rules that are compilable into a runnable CEP engine.

ThingML [15] is the closest contribution to our DSL: it uses a similar device description with messages and communication ports attached to devices, but describes the dynamics of devices and systems through state machines, which appear to be more obscure for end-users. However, the conceptual drawbacks are similar in both paradigms: state machines need to be deterministic on their transitions, while rules have to avoid multiple concurrent firing to avoid executing several rules at the same time.

Other approaches, *e.g.* [4, 9], relying on the *Event Condition Action* (ECA) paradigm, share a similar view for IoT devices orchestration through CEP, though not having the same expressiveness for devices' definition as we propose, especially with time frames and event compositions. In [28], the authors add *pre-* and *post-conditions* to ECA rules but they still do not address time frames constraints too.

All previous contributions take advantages of MDE technologies and tools. More general MDE framework like GeMoC [5] or ThingML allow to specialise the

description of interconnected devices, for example to describe Arduino systems specifically in ArduinoML [20]. On contrary, `IoTDSL` framework concentrates on generating executable rules from user-defined requirements.

7 Conclusion and Future Work

The Internet of Things promotes the usage of various interconnected devices that promise to help end users achieve more automation in daily life recognisable scenarios. As such, a smart home could automatically close doors and switch off lights when the inhabitants leave, or facilitate life routines by assisting in tasks like preparing coffee just in time. The flip side of the coin resides in the ever growing spectrum of connected devices proposed by vendors that spot the market as a good opportunity to make profit, without ensuring a minimal interoperability between their products and those available from other merchants. As a result, the promise seems far from happening in the near future without powerful solutions to catalogue connected devices, to make them exchange relevant data and act in a disciplined way. Furthermore, without bringing end users at the heart of their own story and providing them tools to define, drive and adapt their own scenarios, vendors will always keep a grasp at the IoT market.

In this paper, we explore a first step towards achieving this large challenge by proposing a prototype that aims at raising end users as main actors of how smart home devices interact for their own needs. Aware of the many challenges surrounding IoT systems including reusability, interoperability, scalability and non-functional properties, we designed our solution as an evolving and decentralised tool that allows end users to specify their own scenarios based on so-called rule-based definitions. Our prototype takes the form of a Domain-Specific Language (DSL) associated with a code generator that produces executable code designed to run on a Complex Event Processing (CEP) engine, as well as emulation code dedicated to simulate the whole system before effectively deploying it with concrete devices.

Our prototype `IoTDSL` clearly separates three necessary aspects when describing solutions for IoT systems. First, it captures devices capabilities as high-level events that are meaningful to end users, thus hiding the intricacies of low-level manipulation of DSLs into our platform. Second, it describes device interconnections in a declarative language with predefined communication protocols, leaving the burden of translating data in the appropriate format and transferring them to technicians familiar with those technological details (and who only need to provide links once per protocol). Third, users specify their own scenarios through rules that observe events produced in the environment and trigger reactions when relevant conditions are met. For that purpose, we rely on TRex [12], a powerful, decentralised CEP engine and we automatically generate the necessary code transparently. Our solution takes advantages of Model-Driven Engineering (MDE) to design a DSL that is simple enough while capturing the relevant concepts appropriately and making it flexible enough to rebuild prototypes as the language evolves. In particular, our prototype currently relies upon

a textual syntax, but we plan to design a more intuitive visual syntax for end users.

Despite the promising results we experimented while using `IoTDSL` on small examples with our industrial partners, we acknowledge that many challenges remain. First, reconciling high-level descriptions with low-level devices' APIs, as well as ensuring proper configuration of protocols from declarative intentions, necessitates *glue code* that is not trivial. Fortunately, many initiative already exist, *e.g.* we could rely on platforms like OpenRemote (<http://www.openremote.com>), SmartThings (<https://www.smartthings.com>) or EnOcean (<https://www.enocean.com>) that abstract away several widely used protocols under the same API. Here again, the use of dedicated DSLs could help designing robust and automatic solutions for these technical challenges. Second, our prototype is still at its early stage of development and many improvement directions remain. We must assess the relevance of our DSL against various users and bigger cases to come up with a solution that can be widely adopted. Also, the aforementioned technical challenges need an appropriate answer to widen the automation capabilities of our solution, making it more relevant and usable. Deepening the understanding of each field (low-level devices APIs and communication protocols) would help implementing interesting, reusable solutions. Finally, at the long run, integrating specific properties of IoT systems to guide the code generation while ensuring non-functional properties is necessary in such distributed and vulnerable IoT systems.

References

1. Al-Fuqaha, A., Guizani, M., Mohammadi, M., Aledhari, M., Ayyash, M.: Internet of things: a survey on enabling technologies, protocols, and applications. *IEEE Commun. Surv. Tutor.* 17(4), 2347–2376 (2015, Fourthquarter)
2. Amrani, M., Gilson, F., Debieche, A., Englebort, V.: Towards user-centric DSLs to manage IoT systems. In: *International Conference on Model-Driven Engineering and Software Development (Modelsward)* (2017)
3. Baran, M., Ligęza, A.: Rule-based knowledge management in social threat monitor. In: Dziech, A., Czyżewski, A. (eds.) *MCSS 2013. CCIS*, vol. 368, pp. 1–12. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38559-9_1
4. Bhandari, S.R., Bergmann, N.W.: An internet-of-things system architecture based on services and events. In: *2013 IEEE Eighth International Conference on Intelligent Sensors, Sensor Networks and Information Processing*, pp. 339–344, April 2013
5. Bousse, E., Degueule, T., Vojtisek, D., Mayerhofer, T., Deantoni, J., Combemale, B.: Execution framework of the GEMOC studio (tool demo). In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2016*, pp. 84–89. ACM, New York (2016). <http://doi.acm.org/10.1145/2997364.2997384>
6. Brandtzæg, E., Mohagheghi, P., Mosser, S.: Towards a domain-specific language to deploy applications in the clouds. In: *Third International Conference on Cloud Computing, GRIDs, and Virtualization*, pp. 213–218 (2012)

7. Broda, K., Clark, K., Miller, R., Russo, A.: SAGE: a logical agent-based environment monitoring and control system. In: Tscheligi, M., de Ruyter, B., Markopoulos, P., Wichert, R., Mirlacher, T., Meschterjakov, A., Reitberger, W. (eds.) *AmI 2009*. LNCS, vol. 5859, pp. 112–117. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-05408-2_14
8. Chaqfeh, M.A., Mohamed, N.: Challenges in middleware solutions for the internet of things. In: 2012 International Conference on Collaboration Technologies and Systems (CTS), pp. 21–26, May 2012
9. Cheng, B., Zhu, D., Zhao, S., Chen, J.: Situation-aware IoT service coordination using the event-driven SOA paradigm. *IEEE Trans. Netw. Serv. Manag.* **13**(2), 349–361 (2016)
10. Cristea, V., Pop, F., Dobre, C., Costan, A.: Distributed architectures for event-based systems. In: Helmer, S., Poulouvasilis, A., Xhafa, F. (eds.) *Reasoning in Event-Based Distributed Systems*, vol. 347, pp. 11–45. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19724-6_2
11. Cugola, G., Margara, A.: Tesla: a formally defined event specification language. In: *Proceedings of the 4th Conference on Distributed and Event-Based Systems* (2010)
12. Cugola, G., Margara, A.: Complex event processing with T-REX. *J. Syst. Softw.* **85**(8), 1709–1728 (2012). <https://doi.org/10.1016/j.jss.2012.03.056>
13. García, C.G., G-Bustelo, B.C.P., Espada, J.P., Cueva-Fernandez, G.: Midgar: generation of heterogeneous objects interconnecting applications. A domain specific language proposal for internet of things scenarios. *Comput. Netw.* **64**, 143–158 (2014). <http://www.sciencedirect.com/science/article/pii/S1389128614000528>
14. Gubbi, J., Buyya, R., Marusic, S., Palaniswami, M.: Internet of Things (IoT): a vision, architectural elements, and future directions. *Future Gener. Comput. Syst.* **29**(7), 1645–1660 (2013). <https://doi.org/10.1016/j.future.2013.01.010>
15. Harrand, N., Fleurey, F., Morin, B., Husa, K.E.: ThingML: a language and code generation framework for heterogeneous targets. In: *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, MODELS 2016*, pp. 125–135. ACM, New York (2016). <http://doi.acm.org/10.1145/2976767.2976812>
16. Lee, I., Lee, K.: The Internet of Things (IoT): applications, investments, and challenges for enterprises. *Bus. Horiz.* **58**(4), 431–440 (2015). <http://www.sciencedirect.com/science/article/pii/S0007681315000373>
17. Lúcio, L., Amrani, M., Dingel, J., Lambers, L., Salay, R., Selim, G., Syriani, E., Wimmer, M.: Model transformation intents and their properties. *J. Softw. Syst. (SoSyM)* **15**(3), 647–684 (2014)
18. Ma, M., Wang, P., Chu, C.H.: Data management for internet of things: challenges, approaches and opportunities. In: 2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing, pp. 1144–1151, August 2013
19. Ma, M., Wang, P., Chu, C.H.: Ontology-based semantic modeling and evaluation for internet of things applications. In: 2014 IEEE International Conference on Internet of Things (iThings), and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom), pp. 24–30, September 2014

20. Mosser, S., Collet, P., Blay-Fornarino, M.: Exploiting the internet of things to teach domain-specific languages and modeling: the ArduinoML project. In: Demuth, B., Stikkolorum, D.R. (eds.) Proceedings of the MODELS Educators Symposium co-located with the ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014), Valencia, Spain, 29 September 2014, CEUR Workshop Proceedings, vol. 1346, pp. 45–54. CEUR-WS.org (2014). http://ceur-ws.org/Vol-1346/edusymp2014_paper_3.pdf
21. Mühl, G., Fiege, L., Pietzuch, P.: Distributed Event-Based Systems. Springer, Secaucus (2006). <https://doi.org/10.1007/3-540-32653-7>
22. Mukhopadhyay, S.C., Suryadevara, N.K.: Internet of Things: Challenges and Opportunities. In: Mukhopadhyay, S.C. (ed.) Internet of Things. SSMI, vol. 9, pp. 1–17. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-04223-7_1
23. Munnely, J., Clarke, S.: A domain-specific language for ubiquitous healthcare. In: 2008 Third International Conference on Pervasive Computing and Applications, vol. 2, pp. 757–762, October 2008
24. Park, H., Kim, H., Joo, H., Song, J.: Recent advancements in the Internet-of-Things related standards: a oneM2M perspective. *ICT Express* **2**(3), 126–129 (2016). <http://www.sciencedirect.com/science/article/pii/S2405959516300911>. Special Issue on ICT Convergence in the Internet of Things (IoT)
25. Pradhan, S.M., Dubey, A., Gokhale, A., Lehofer, M.: Chariot: a domain specific language for extensible cyber-physical systems. In: Proceedings of the Workshop on Domain-Specific Modeling, DSM 2015, pp. 9–16. ACM, New York (2015). <http://doi.acm.org/10.1145/2846696.2846708>
26. Salihbegovic, A., Eterovic, T., Kaljic, E., Ribic, S.: Design of a domain specific language and IDE for internet of things applications. In: 2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), pp. 996–1001, May 2015
27. Schultz-Møller, N.P., Migliavacca, M., Pietzuch, P.: Distributed complex event processing with query rewriting. In: Proceedings of the Third ACM International Conference on Distributed Event-Based Systems, DEBS 2009, pp. 4:1–4:12. ACM, New York (2009). <http://doi.acm.org/10.1145/1619258.1619264>
28. Shimokura, M., Nakanishi, S., Ohta, T.: Home network service programs described in a rule-based language. In: International Conference on Software Engineering Advances (ICSEA 2007), pp. 62–62, August 2007
29. Singh, D., Tripathi, G., Jara, A.J.: A survey of internet-of-things: future vision, architecture, challenges and services. In: 2014 IEEE World Forum on Internet of Things (WF-IoT), pp. 287–292, March 2014
30. Tan, L., Wang, N.: Future internet: the Internet of Things. In: 3rd International Conference on Advanced Computer Theory and Engineering(ICACTE), vol. 5 (2010)
31. Xu, L.D., He, W., Li, S.: Internet of Things in industries: a survey. *IEEE Trans. Ind. Inform.* **10**(4), 2233–2243 (2014)
32. Xu, T., Wendt, J.B., Potkonjak, M.: Security of IoT systems: design challenges and opportunities. In: 2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 417–423, November 2014



Efficient Distributed Execution of Multi-component Scenario-Based Models

Shlomi Steinberg¹, Joel Greenyer², Daniel Gritzner², David Harel¹, Guy Katz³,
and Assaf Marron¹(✉)

¹ The Weizmann Institute of Science, Rehovot, Israel
assaf.marron@weizmann.ac.il

² Leibniz Universität Hannover, Hannover, Germany

³ Stanford University, Stanford, USA

Abstract. In scenario-based programming (SBP), the semantics, which enables direct execution of these intuitive specifications, calls, among others, for synchronizing concurrent scenarios prior to every event-selection decision. Doing so even when the running scenarios are distributed across multiple physical system components, may degrade system performance or robustness. In this paper we describe a technique for automated distribution of an otherwise-centralized specification, such that much of the synchronization requirement may be relaxed. The technique calls for replicating the entire scenario-based executable specification in each of the components, locally transforming it in a component-specific manner, and reducing the synchronization requirements to very specific and well-defined points during execution. Our evaluation of the technique shows promising results. Given that relaxed synchronization can lead to what appears as different runs in different components we discuss various criteria for what would constitute acceptable differences, or divergence, in the parallel, distributed runs of almost-identical copies of a single specification.

This paper incorporates and substantially extends the material of the paper published in MODLESWARD'17 *Distributing Scenario-Based Models: A Replicate-and-Project Approach* by the same authors [37].

Keywords: Software engineering · Scenario-based modeling
Concurrency · Distributed systems

1 Introduction

With modern reactive systems becoming both pervasive and highly complex, modeling them is becoming increasingly difficult. Modelers are forced to spend ever-larger amounts of time and effort in order to reconcile two goals: (1) accurately describe complex real-world systems and phenomena; and (2) do so using models that are simple, comprehensible and intuitive to humans. These two goals are often conflicting: it is difficult to describe the properties of such systems accurately, while at the same time avoiding clutter, which makes it harder for humans to comprehend the resulting models.

Over the recent two decades, an approach termed *Scenario-Based Modeling* [6] has emerged as an attempt to tackle these difficulties. The idea at its core is to model systems in a way that is more intuitive and understandable to humans—by defining *scenarios* that describe desirable or undesirable system behavior—and then to automatically combine these scenarios in a way that produces a cohesive, global model. Appropriate scenario-based approaches and tools have executable semantics, thus helping to streamline the deployment of scenario-based models in the real world.

A scenario-based approach has been claimed to be more intuitive for humans to understand (see, e.g., [11]). It allows the modeler to specify different but possibly interrelated behavioral aspects as separate scenarios, reducing the inherent complexities of the modeling process. However, by default and as explained later, a scenario-based execution requires that all scenarios synchronize at every step, for the purpose of joint event selection. When executing scenario-based specifications in a distributed architecture, inter-scenario synchronization induces inter-component synchronization, which may be undesirable in real-world systems, where communication is often costly, slow, or unreliable. This difficulty constitutes a serious barrier when considering the use of scenario-based modeling in a real-world distributed setting.

We seek to address this problem by proposing an automated technique for the transformation of classical, highly synchronous scenario-based models into equivalent models with a greatly reduced level of synchronization. The basis of our approach is a rather straightforward *replicate-and-project* (R&P) technique but with some subtle facets: we *replicate* the full set of scenarios in all the distributed components, but *project* them in a component-specific fashion, so that each component is made responsible only for the actions that fall within its the local scope. Other, external actions, are assumed to be performed by projections running on other components.

The scenarios then progress asynchronously, each selecting and triggering events almost completely at its own pace. In order to make the replicated-and-projected scenarios behave the same as their non-distributed version, the distributed components broadcast the local actions they perform to all other components. At times a situation arises that forces some of the distributed components to mutually agree on the next action to perform. This might happen either due to an exclusive choice among multiple enabled actions (i.e., events), or due to communication latency that might result in different orders of broadcast actions as observed by different components. In these cases, the affected scenarios indeed synchronize and reach a joint decision. An important part of the work in this paper is dedicated to classifying these cases, understanding when they arise, automatically detecting their occurrence in a program, and proposing practical approaches for resolving them.

This process is handled automatically by our distribution algorithm and infrastructure, and, as we discuss and demonstrate later, it aims to generate a distributed model that has as few synchronization points as possible.

The motivation behind the approach is to retain the modeler’s ability to use classical scenario-based modeling, with its associated advantages, but to be able to then transform the model into a version that is more amenable to distribution and deployment in the real world. We prove that, under certain restrictions, our proposed transformation preserves the behavior of the original model. This gives rise to a methodology for developing distributed scenario-based models, where one models a distributed system as if it were centralized, and the model is then automatically adjusted to more accurately simulate (or even run in) its final setting.

Automatic distribution of general models (i.e., not just scenario-based ones) or synthesizing distributed models from specifications have been long-standing goals of the software modeling and engineering community. Specifically, distributed synthesis is known to be undecidable in general [36]. We contribute to this effort by studying the problem in the context of scenario-based modeling, and leveraging some of the paradigm’s properties of naturalness and relative simplicity. However, difficulties nevertheless arise. We classify and describe them, and explain how they can still be addressed. Our experimental results indicate that the technique holds much potential for becoming practical.

The rest of the paper is organized as follows. In Sect. 2 we provide a brief introduction to the scenario-based approach. In Sect. 3 we present an example of a distributed execution of a scenario-based specification for a light show to be performed by light-equipped drones. This general description is used in the rationale and explanation of various technical details throughout the paper, and a variant of the example is used in the technical evaluation. In Sect. 4 we discuss variations, which may or may not be allowed when transforming a fully synchronized execution into one where some synchronization requirements are relaxed and certain actions may occur in a different order. In Sect. 5 we describe the replicate-and-project technique for automatically generating an executable distributed scenario-based model from a non-distributed one. In this section we also prove the correctness of this transformation according to the criteria set in Sect. 4. Section 6 describes how the approach can be applied when different components in the model operate on different time scales. An example implementation and its evaluation appear in Sect. 7. In Sect. 8, we discuss related work that has been carried out on automatic distribution, both in the general setting and in the context of scenario-based modeling, Sect. 9 contains a discussion of our ongoing and planned future work. We conclude in Sect. 10.

2 Background: Scenario-Based Specifications

Scenario-based specifications were introduced with the *Live Sequence Charts* (*LSC*) formalism [6, 25]. The approach, aimed at developing executable models of reactive systems, shifts the focus from describing individual objects of the system into describing individual behaviors thereof. The basic building block in this approach is the *scenario*: an artifact that describes a single behavior of the system, possibly involving multiple different components thereof. Scenarios can

describe desirable behaviors of the system or undesirable ones, and their combinations. A set of user-defined scenarios is then interwoven into one cohesive, potentially complex, system behavior.

Several facets of scenario-based modeling have been discussed and handled in different ways: scenarios can be represented graphically, as in the original LSC approach, or textually, embedded within conventional programming languages [13, 27]; scenario-based models can be executed by naïve *play-out* [26], by smart play-out with lookahead [23] or via controller synthesis (see, e.g., [13, 29]). The modeling process can be augmented by a variety of automated verification, synthesis and repair tools [16, 21]. Research has shown that the basic principles at the core of the approach, shared by all flavors, are *naturalness* and *incrementality*—in the sense that scenario-based modeling is easy to learn and understand, and that it facilitates the incremental development of complex models [1, 11]. These properties stem from the fact that modeling is done in a way similar to the way humans explain complex phenomena to each other, detailing the various steps and behaviors one at a time.

For the remainder of the paper, we focus on a particularly simple variant of scenario-based modeling, called *behavioral programming (BP)* [27]. Despite its simplicity, BP has been successfully used in developing medium scale projects [18, 20], and is also known to be particularly amenable to automatic analysis tools [22]. These properties render BP a good candidate for demonstrating our approach. The rest of this section is dedicated to demonstrating and formally defining BP.

In BP, a model is a set of scenarios, termed also *behavior threads*, or *b-threads*, and an execution is a sequence of points, in which all the scenarios synchronize. At every behavioral-synchronization point (abbreviated *bSync*) each scenario pauses and declares events that it *requests* and events that it *blocks*. Intuitively, these two sets encode desirable system behaviors (requested events) and undesirable ones (blocked events). Scenarios can also declare events that they passively *wait-for*—stating that they wish to be notified if and when these events occur. The scenarios do not communicate their event declarations directly to each other; rather, all event declarations are collected by an execution infrastructure common to all b-threads, termed the *event selection mechanism (ESM)*, after its main function. Then, at every synchronization point during execution, the ESM selects (*triggers*) an event that is requested by some scenario and not blocked by any scenario. The ESM notifies every scenario that requested or is waiting for the triggered event about this selection. The b-threads can then update their internal states, and proceed to their next synchronization point. When all affected b-threads synchronize again (with each other and with the b-threads that were not affected) the ESM repeats the event selection process. In BP, this notification of all affected b-threads is the essence of event triggering. Any additional action that the designer wishes to associate with an event (e.g., opening a water tap, turning car’s steering wheel, or flashing a light) is to be carried out by the individual b-threads, using arbitrary method calls, as they transition from one synchronization point to another (by contrast, in the LSC language, the trigger-

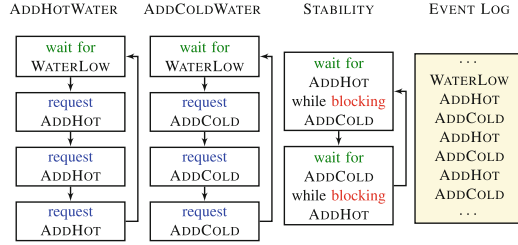


Fig. 1. Incrementally modeling a controller for the water level in a tub. The tub has hot and cold water sources, and either may be turned on in order to increase/reduce the water temperature. Each scenario is given as a transition system, where the nodes represent synchronization points. The scenario ADDHOTWATER repeatedly waits for WATERLOW events and requests three times the event ADDHOT. Scenario ADDCOLDWATER performs a similar action with the event ADDCOLD, capturing a separate requirement, which was introduced when adding three water quantities for every sensor reading proved to be insufficient. When a model with scenarios ADDHOTWATER and ADDCOLDWATER is executed, the three ADDHOT events and three ADDCOLD events may be triggered in any order. When a new requirement is introduced, to the effect that the water temperature be kept stable, the scenario STABILITY is added, enforcing the interleaving of ADDHOT and ADDCOLD events by using event blocking. The execution trace of the resulting model is depicted in the event log.

ing of an event also drives the invocation of a corresponding method provided by the application). Figure 1 (borrowed from [20]) demonstrates a simple behavioral model.

Formally, BP’s semantics are defined as follows. A scenario, also referred to as a *behavior thread* (abbreviated *b-thread*), is defined as a tuple

$$BT = \langle Q, q_0, \delta, R, B \rangle$$

and with respect to a global set of events Σ . The components of the tuple are: a set of states Q representing synchronization points; an initial state $q_0 \in Q$; a deterministic transition function $\delta : Q \times \Sigma \rightarrow Q$ that specifies how the thread changes states in response to the triggering of events; and, two labeling functions, $R : Q \rightarrow \mathcal{P}(\Sigma)$ and $B : Q \rightarrow \mathcal{P}(\Sigma)$, which specify the events that the thread requests (R) and blocks (B) in a given synchronization point.

A *behavioral model* M is defined as a collection of b-threads

$$M = \{BT^1, \dots, BT^n\},$$

all of them with respect to the same event set Σ . Denoting the individual b-threads as

$$BT^i = \langle Q^i, q_0^i, \delta^i, R^i, B^i \rangle,$$

an execution of model M starts at the initial state $\langle q_0^1, \dots, q_0^n \rangle$. Then, at every state $\langle q^1, \dots, q^n \rangle$, the model progresses to the next state $\langle \bar{q}^1, \dots, \bar{q}^n \rangle$ by:

1. selecting an event $e \in \Sigma$ that is enabled, i.e. requested by at least one b-thread and blocked by none:

$$e \in \left(\bigcup_{i=1}^n R^i(q^i) \right) \setminus \left(\bigcup_{i=1}^n B^i(q^i) \right)$$

2. triggering event e and advancing the individual b-threads according to their transition systems:

$$\forall i, \quad \bar{q}^i = \delta^i(q^i, e)$$

For reactive systems, executions are usually considered to be infinite, although BP can also be used to model systems with finite executions.

The BP definitions above are generic, making it easier to reason about behavioral models. However, for practical purposes, the BP modeling principles have been integrated into a variety of high-level languages such as Java, C++, Erlang and Javascript (see the BP website at <http://www.b-prog.org/>). These frameworks allow engineers to integrate reactive scenarios into their favorite programming or modeling environments. Further, the same principles that underly BP play a significant role in several popular modeling frameworks, such as *publish-subscribe* architectures [8] and *supervisory control* [35].

We conclude the section by defining the global state (the *cut*) of a behavioral model that is being executed:

Definition 1. *Given a behavioral model $M = \{BT^1, \dots, BT^n\}$, the program cut $r \in Q^1 \times \dots \times Q^n$ is defined to be the current model state: $r = \langle q^1, \dots, q^n \rangle$ where q^i is the current state of b-thread BT^i .*

3 A Running Example

In many complex multi-participant operations, the participants, be they mechanical entities or people, have to carry out actions in turns, one participant after the other. A typical example is the all-way-stop traffic intersection (a.k.a. four-way stop). When there are queues in each of the intersecting roads, the cars cross the intersection one at a time, in a round-robin fashion, each coming from the front of the next queue. Another example is an audience in a packed stadium ‘doing the wave’, where groups of people stand up briefly and then sit down, in sequential order. These behaviors are very easily described using scenario-based specifications, where the most basic behavior can be described with a single scenario showing all the relevant entities performing their required actions in turn. (Of course, there are also other kinds of scenarios; e.g., for passing a all-way-stop intersection when you are the only car, or for the starting or the ending of a stadium wave by an audience.)

The example that we will use both to illustrate our general considerations and as the subject of our detailed analysis, is a simple drone-based light show (see elaborate shows by Disney in www.youtube.com/watch?v=gYr-PO9meHY, and

by Intel in www.youtube.com/watch?v=teQwViKMnxw). In our case, a set of drones form a circle and flash their respective lights in successive turns, creating the appearance of a point of light moving in a circle. More details are provided in Sect. 7. The basic, single-cycle example is then expanded into repeating the cycle, stopping the cycle and then restarting it with a different, arbitrarily-selected drone, and having multiple concurrent cycles where each drone is equipped with multiple lights, perhaps of different colors.

In considering this example one may also think of analogies to human behavior: replacing the programmer or designer with a show director, the drones with people, perhaps children, who play roles in the show, and the computerized scenarios or programs (as well as the underlying SBP infrastructure), with the instructions given by the director to the participants about what they should do, and when. The autonomous starting of a new cycle at arbitrarily-selected drones may also be considered as reacting to an uncontrolled environment event, e.g., when the show-director decides on their own and unpredictably, at run time, which drone will be the first in the next cycle, and then signals it to do so.

4 Distributing a Centralized SBP Execution: Success Criteria

4.1 Success Criteria

In order to assess the properties of a distributed execution of a specification that was originally written with centralized-execution semantics assumptions, we first discuss (a) formalization assumptions, namely: which physical properties of the distributed environment will be reflected in the formal solution and which will be abstracted away, and (b) criteria for what constitutes a correct, desirable, or perhaps just acceptable, distributed execution.

In a centralized system the concept of a run is well defined and intuitive as the sequence of system states and generated events. In a distributed environment, especially one that includes replication, this very definition is no longer without question. E.g., is the distributed run the collection of local runs as executed and observed in each component? Or perhaps it is a sequence of only the triggered events, without state transitions, ordered according to the occurrence of events in the real world, e.g., according to a time order as defined by fully synchronized component-specific clocks? Or should yet another definition be applied?

Once defined, what would be the desired properties of such a distributed run? Clearly, our goal is that it be materially different from a run where all distributed components fully synchronize before every event, and that some independent local progress will be allowed in each component. So the question we are facing is this: how much should the execution of the various components be allowed to vary from each other?

In subsequent sections we offer a particular set of principles for the sought-for solution, and a distribution mechanism that satisfies them. But beforehand we first discuss a broader list of candidate principles from which the above

were chosen. Some of these can be formally defined and then examined both by model-checking and by run-time monitoring. Each of the candidate principles is accompanied by two examples - one demonstrating its desirability, and one showing that acceptable distributed executions exist that do not satisfy this principle, and hence it cannot be required of all distributed executions.

Constant Composite-State Consistency. This principle assumes that all scenarios are replicated in all components (as proposed in the present paper), and requires that the all components go through exactly the same orchestrated state transitions, and hence observe the same runs (even if not exactly at the same time). The replicated run is also a legal run of a centralized or fully synchronized execution. This could be desired and applicable, for example, in an application that has only pre-programmed actuation (as in the most basic drone light show example). Clearly this could not be demanded in a case of a reactive application with distributed input sensors, where two environment events can occur in two distinct components, one after the other, but with a time difference that is smaller than the inter-component communication delay, and where the sensing component has to acknowledge the event receipt even before all other components learned about it. As a result, the runs of the two sensing components will be different—each having a state where its own sensing scenario has sensed the event and changed its state, but none of the others have done so.

Always Eventually Reaching Composite-State Consistency. Under this principle, the entire specification is replicated as before, but components' runs are allowed to diverge as long as there is at least one composite state that each component reaches infinitely often. In other words, the components may diverge, as long as sooner or later they maintain the same view of reality (not necessarily at the same time). An example of such an application can be seen in a distributed application of industrial robots performing manufacturing tasks in parallel on a large piece of sheet metal, where the order of events across robots is not critical as long as all components are occasionally synchronized and are at the same state (e.g., when releasing the finished piece of sheet metal and moving on to the next one). This however will not satisfy what is needed for a highly orchestrated robotic collaboration (and not even for the basic drone light show).

Distinguishing *What* and *How* Scenarios. This principle views specifications as being divided into scenarios that specify the criteria for success of the system's operation, i.e., what the system should accomplish, and scenarios that specify how the system should accomplish these goals. In classical programming the *what* scenarios appear in requirements documents and test plans, and the *how* instructions constitute the application. Research in areas such as automated program synthesis attempts to automatically generate the details of the *how* from the specification of the *what*. Here we propose to specify and retain both sets of scenarios, but require that only the *what* scenarios must be complied with in the distributed run, while the *how* scenarios can be violated in the distribution process. In the light-show example, the show director wants to achieve the appearance of cycles, or perhaps even just the appearance of pretty patterns. He

or she may not care if certain drones flash their lights out of order, especially if they are in close proximity, the successive inter-drone flash delays are short, and the duration of each flash is much longer than this inter-drone delay; a drone that misses its cue may also be allowed to avoid flashing altogether in a given cycle; a drone whose battery runs out may leave the show altogether; and, neighbors of a failing drone may change their behavior as well. Thus, the divergence of runs among various components may be unbounded, while the show goes on successfully.

An example of when this approach cannot be applied can be seen in the following: a show director and an engineer created an elaborate show whose specification contains many scenarios. For testing purposes the show was implemented on a single computing component with multiple physical lights. The show is elaborate and its specification gradually evolved to have many scenarios. The director has now left, and the engineer has been tasked with distributing the implementation to the separate drones. As far as the engineer is concerned, the entire specification is the *what*, everything that was done in the centralized execution should be done in the same way in the distributed version—he or she does not know which of properties were considered essential by the director, and which can be compromised.

Language Equivalence. Under this principle we do not care about run variation among the components. Instead, we only look at the sequence of events produced (triggered) by the system, in all components, as ordered in the real world, according to some global time stamp. In this case, we do not require that a particular run of the distributed environment, defined in this manner, be equivalent to a particular run of the single-component system, but assume that there is some nondeterminism in both implementations, and simply require that the two languages, each containing *all runs* of the implementation, be equal. Thus the nondeterminism implied by the underspecification that is already built into the original requirements will be exploited by the variation imposed by the non-synchronized, sometimes-delayed, distributed execution. This can indeed be viewed as a variant of the previous principle, where the existence (in the centralized execution) of synchronization points where more than one event is enabled, is taken to be an explicit specification that selecting any of them would be acceptable. (We assume that the event selection strategy is random, and that the application was verified with all possible combinations of event selection.)

4.2 Semantic Consideration

It goes without saying that a success criteria to be added to the above is that the execution should comply with the basic BP semantics, in that, e.g., only requested events are triggered, and events that are blocked are not to be triggered. The solution that we propose in the coming sections satisfies this basic requirement with one exception: *There is no reliance on cross-component blocking of already-enabled events.* Clearly, when an event is triggered, two b-threads may change their states, where one will start requesting a particular event e_1 ,

and another will start blocking that same e_1 . The effect of such blocking is immediate. This semantics is generally preserved in the solution we will describe, e.g., when these two b-threads change states together, in response to a single event, within a single component. However, we introduce an assumption that relaxes this requirement in that it allows event blocking to not take hold in the following case: An event e_0 causes a b-thread BT_1 to change states and start requesting event e_1 . An event e_2 is then triggered, and causes b-thread BT_2 to change state and start blocking e_1 . If after event e_2 occurred in one component, but before this event reaches a component requesting e_1 , e_1 is already triggered in that component, we do not consider it a violation of the specification or of the BP semantics. Another way to look at this relaxation of the semantics is that it assumes that the application does not rely on the ability of one component to force the blocking of already-enabled, not-previously blocked events in other components, in time, before they are triggered.

For illustration, consider the following example: a robot-driven car is approaching an intersection, and in order to avoid collisions it must communicate with other cars. However, if the communication happens just before entering the intersection, any delay or missed messages could result in an accident.

In order to avoid this kind of issue, programs designed for distribution should employ design patterns and methods that take a realistic communication delay into account. E.g., checking for other cars early, while approaching the intersection, rather than, say, relying on scenarios to block all events of cars entering the intersection following the occurrence of an event reporting that one car already entered that intersection. We feel that this is a valid assumption in designing distributed systems and does not contradict or make redundant the advantages of BP.

This assumption, formalized in Sect. 5, can thus be seen as a restriction on how the application should be coded, or on features that must be added to the application if not already written in this manner.

4.3 Additional Considerations

As distributed implementations introduce new risks, additional responsibilities have to be imposed both on the distribution mechanism and on the application scenarios themselves.

Robustness. There is a desire to minimize the probability of error and of failure. First, we would like the scenarios governing the behavior to be as simple as possible. Second, ‘the show must go on’ even if one of the participants made a mistake or missed their cue. For the latter, specific scenarios can probably be added. In the light show example, we could add “when a drone observes that a predecessor drone has failed or is delayed, it should nevertheless continue the cycle.” **Efficiency.** Often, the joint operation should also be required to be efficient. Consider for example the case when many bricks have to be moved from point A to point B over a narrow passage. A group of robots may be arranged in a row—passing bricks from one to the next, rather than each one traveling the entire distance. The scenarios should be designed so that inter-scenario

synchronization and coordination is minimized, or decreased, and both scenario progression and the physical motion of bricks occur in parallel, asynchronously. Such measures of efficiency are evaluated in the example in Sect. 7

5 Distribution via Replicate-and-Project

The execution of a classical BP model, as described in Sect. 2, is highly synchronized and centralized by nature: at every step along the execution, the ESM gathers the sets of requested and blocked events from each individual b-thread, selects an enabled event (i.e., requested by some b-thread but blocked by none), and broadcasts it back to the b-threads. While this underlies some of the benefits of BP [27], it also results in limited scalability and distributability. Excessive synchronization tends to add unnecessary complexity, impact performance, and create inter-component dependencies that reduce robustness. For example, having a scenario wait for an event that is supposed to be requested by a scenario running on a separate, failed component might result in deadlock. Furthermore, synchronization forces b-threads to execute in lockstep, which can be undesirable if they are to model phenomena that occur at different timescales.

In this section we propose a *distribution process* that, given a centralized (undistributed) behavioral model, generates a distributed one: It creates multiple *component* models—subsets of the original, centralized behavioral model—each a behavioral program, designed to be run on a separate machine. Run simultaneously, these behavioral component models (or simply, component models) mimic the behavior of the original system, but require much less synchronization. Below we elaborate on the abstract concepts and formal definitions of the proposed process.

Each of the component models produced by our distribution process is a behavioral model in its own right, intended to be responsible for a certain subset of the events of the original model, which are uniquely owned and *controlled* by it—meaning that no other component can request or block them. The behavioral component models are intended to be executed in an asynchronous manner in a distributed system, resulting in a natural, robust and simple extension of the scenario-based paradigm.

The main difficulty in this approach is to ensure that the distributed components behave in the same way as the original model although they are not synchronized at every step. In mitigating this difficulty, the crux of our distribution process is the replication of the entire set of original scenarios in each of the distributed components, granting the components the ability to follow what other components are doing, but avoiding synchronization whenever possible. First, there is no central, coordinating ESM. Every component runs a separate, local, ESM, which by default, performs local event selection without synchronizing with other components. However, at every synchronization point where multiple components have to agree on the particular event to select, the ESMs of these components do synchronize.

The communication between components is asynchronous, and they notify each other about chosen events as they progress through the scenarios. Keeping

track of each scenario state is simply a matter of listening to incoming broadcasts and updating the current state. This asynchrony is a cornerstone of the process, allowing us to generate true concurrent distributed models.

The classical problem of multicasting or broadcasting a message efficiently in a distributed network is well studied (for example, the authors of [33] present an approach for minimum-energy-broadcasts in distributed networks with limited resources and unknown topology). However it is beyond the scope of this paper. For simplicity we assume that the cost of those broadcasts and bookkeeping is small. Note that even in systems with a large number of components and scenarios, a component often needs to keep track of only a small subset of the other components; for example, an autonomous car considers other cars only when they are in its immediate vicinity, and does not have to keep track of all the vehicles in the world. Still, this dynamic registering and unregistering of components is also beyond the scope of this paper and is left for future work.

In the remainder of the section we formalize these notions and the distribution process itself.

5.1 Defining Event Components

Let M denote a behavioral model over event set Σ . An *event component* E is a subset of the global event set, $E \subseteq \Sigma$. Intuitively, each subset E reflects (or is implicitly defined by) a physical component of the distributed system and its responsibility in terms of physical capabilities and/or environment interfaces, i.e., sensors and actuators, that this component has. An event $e \in E$ is said to be a *local event* of E ; otherwise, if $e \notin E$ then e is *external* to E .

A collection of event components $\{E_1, \dots, E_k\}$ is an *event separation* of Σ if $\bigcup_{i=1}^k E_i = \Sigma$. An event separation is *strict* if it also forms a partition of Σ :

$$\forall i, j, \quad 1 \leq i \neq j \leq k \implies E_i \cap E_j = \emptyset.$$

In the remainder of the paper we will only deal with strict event separations and assume that they are provided by the user. Automated ways of generating an event separation are discussed in Sect. 8.

5.2 Creating Behavioral Component Models by Replication and Projection

Given a behavioral model $M = \{BT^1, \dots, BT^n\}$ over event set Σ and a strict event separation $\{E_1, \dots, E_k\}$, each event component E gives rise to a *behavioral component model* C , in the following way. C is the behavioral model $C = \{BT_E^1, \dots, BT_E^n\}$, obtained by *projecting* each of the original b-threads onto event component E . The projection operation, denoted as $C = \text{project}(M, E)$, transforms each of the original b-threads as follows. If $BT^i = \langle Q^i, q_0^i, \delta^i, R^i, B^i \rangle$ then

$$BT_E^i = \langle Q^i, q_0^i, \delta^i, R_E^i, B_E^i \rangle$$

is defined as follows: The state set Q^i , the initial state q_0^i and, most importantly, the transition function δ^i which specifies how events cause state transitions, are replicated by the projection process without change. The original labeling functions R^i and B^i , namely the sets of requested and blocked events in each state, are projected onto the respective R_E^i and B_E^i according to the rules:

$$\begin{aligned} R_E^i(q) &= R^i(q) \cap E \\ B_E^i(q) &= B^i(q) \cap E \end{aligned}$$

That is, the projected b-threads are modified to request and block only events that are in E ; but because δ^i is unchanged they continue to respond in the same way to the triggering of all events, including those not in E . Consequently, where an external event is requested in a b-thread, it is modified to only be waited-for.

Now, given a (strict) event separation $\{E_1, \dots, E_k\}$, our distribution process entails projecting the model M onto each of the event components, producing a set of component models $\{C_1, \dots, C_k\}$ such that

$$\forall i \ 1 \leq i \leq k, \quad C_i = \text{project}(M, E_i)$$

By treating each component C_i as a separate behavioral model that performs event selection and scenario advancement (i.e., state transition) locally, the components can be run independently and in a distributed manner. This is, however, qualified by the fact that, in order to keep the execution consistent between components, at certain points two or more components need to synchronize with each other. This is discussed in detail in the next subsection.

The following useful corollary is a direct conclusion that arises from the definition of the distribution process, when applied in the context of strict event separations.

Corollary 1. *An event $e \in \Sigma$ can be selected by at most one component.*

Proof. $\{E_1, \dots, E_k\}$ is a strict event separation, hence there is only one value of i such that $e \in E_i$. Only C_i can request e , since, by the definition, in all other components C_j , $j \neq i$, the requests for e are replaced by waiting for it. Therefore only C_i can select e . \square

5.3 Distributed Execution of Replicated-and-Projected Component Models

As discussed in Sect. 4, despite their parallel asynchronous execution, it is our goal that component-model execution be consistent with each other and with that of the original model. Since in the specification more than one event may be requested at a given state, occasionally these distributed runs need to be synchronized. In this subsection and the next we describe the mechanics of parallel distributed execution of component models, and the specific synchronization constraints this execution is subjected to.

The R&P approach includes using in each component a modified BP execution infrastructure. The component's ESM is different from the one described in Sect. 2, in that it broadcasts to other components its local independent decisions, it processes similar messages received from other components, and, when required, it synchronizes with other components to make a joint decision.

Specifically, the following rules govern each component's ESM and the distributed execution.

1. Each component has an *event queue*, to the end of which the component's ESM can push (i.e., add) events, and from the front of which it can pop (i.e., remove and process) events.
2. When a b-thread enters a new state, the execution infrastructure determines whether or not it is an *inter-component decision point* (ICDP), i.e., whether or not it should induce synchronization with certain other components (ICDPs are defined in the next subsection).
3. When a component's ESM receives an event that was broadcast by another component, the event is pushed to the end of the component's event queue.
4. When a component enters a new state (either initially, or following re-synchronization of all b-threads following the triggering of an event that affected at least one b-thread), the ESM does the following:
 - (a) If the component's event queue has at least one event, the ESM pops the first event from the queue, and triggers it (i.e., notifies affected b-threads, who then change states and re-synchronize).
 - (b) If the queue is empty, then
 - i. If one of the b-threads is in an ICDP, the ESM waits for the components specified in the ICDP to reach the corresponding ICDP and/or confirms that they are already at that point (note that no component goes past an ICDP without synchronizing with the others). If two ICDPs are in effect concurrently, they are handled, separately, in arbitrary order. Hence, all the components involved in an inter-component decision consider the same sets of requested and blocked events. The components then synchronize and mutually agree upon the triggered event. This event is then broadcast to all components (including the ones involved in the decision itself). Note that the chosen event may or may not be one of those that induced the need for inter-component decision. In the latter case, the b-threads that induced the ICDP will not react to the chosen event, and the component will be at an ICDP at the next synchronization point as well.
 - ii. If there is an event that is in the local requested event set and not in the local blocked event set (for the current composite, synchronized state of all b-threads in the specification as modified locally under R&P), the ESM triggers that event, and broadcasts it to all other components.
 - iii. If the event queue is empty and there is no event that is locally enabled, the ESM waits for external events to arrive via broadcast from other components.

- iv. Otherwise, that is, if the event queue is empty and there is no event that is locally enabled, the ESM waits for external events to arrive via broadcast from other components.
- 5. When b-threads are notified of selected events they change their states according to their local state-transition function (which is identical in all components and is the same as in the original non-distributed specification).

We observe that deadlock-detection needs to be treated differently in the distributed case compared to the centralized case. According to the semantics given in Sect. 2, the system can detect a deadlock if the ESM determines at some point that all requested events are blocked, so that none can be selected. This, of course, holds only in the case of static scenarios, and where simulation of environment behavior is already included in the model. By contrast, in the distributed case this is no longer the case, as components begin to serve as each other's environment: If one of the local b-threads waits for an event that is external to the component, another component might broadcast that event. Thus, the component should just be stalled until such a broadcast arrives.

Definition 2. A distributed model produced from a behavioral model M , with respect to a strict event separation, $S = \{E_1, \dots, E_k\}$, denoted as $\mathcal{D}(M, S)$, is defined to be the set of projections of M along the components of the event separation:

$$\mathcal{D}(M, S) = \{\text{project}(M, E_1), \dots, \text{project}(M, E_k)\}.$$

Executing a distributed model means executing the component models (i.e., the projections) according to the operational semantics defined in this section.

5.4 Conditions for Inter-component Synchronization

The following definition is useful in identifying the points during the execution in which multiple components need to synchronize:

Definition 3. Given a component model $C_j = \text{project}(M, E_j)$, a b-thread BT^i and some state $q \in Q^i$. We say that BT^i is controlled by C_j at state q if one or more of E_j 's local events is requested or waited-for in q ; i.e., if $\exists e \in E_j$ such that $\delta^i(q, e) \neq q$ or $e \in R^i(q)$.

Definition 4. Given a component model $C_j = \text{project}(M, E_j)$, we call a state $q \in Q^i$ in a component's b-thread BT^i an inter-component decision point (ICDP) if and only if q is controlled by multiple components and $\exists e \in E_j$ such that $e \in R^i(q)$.

The R&P distribution process dynamically determines when a b-thread is in a state that is an inter-component decision point per Definition 4.

For example, assume that in the original specification for a four-wheel vehicle a single b-thread requests two events (e.g., `steerRight` and `steerLeft`), allowing the ESM to non-deterministically choose one, as would be the case if a 'random

walk' were desired. Then, in the distributed implementation, if the two events end up in a single physical component, this will not be an ICDP. But, if they are in separate components, coordination will be required, naturally, and this will be an ICDP. Consider also the case where these two events are requested by two separate b-threads. In a centralized implementation this will be valid, especially if each of the two b-threads also waits at this point for the other b-thread's event and stops requesting its own if it sees that the other's request is selected. Moreover, if the two events are in distinct components as before, then the requesting and waiting (in a single b-thread) would cause the corresponding state, which appears in the replicated b-thread in both components, to be marked as an ICDP, yielding the same sets of runs. Alternatively, if the events are indeed in physically independent components, as would be the case when `steerRight` is implemented by advancing (rolling forward) the left front wheel, (and `steerLeft`, respectively, by advancing the right wheel), then the developer has the option of removing the waiting-for-the-other-event from the `bSync` call in that state. In this case, these states will no longer be ICDPs, and one of the possible runs is that both requested events will be selected (one after the other), both front wheels will be advanced, and the vehicle will advance forward rather than turn. We note however, that here the specification and the set of runs has changed dramatically to accommodate, or take advantage of, some new capabilities of the distributed environment, and we no longer attempt to preserve the set of original runs.

It is important to note that the properties that induce the existence of an ICDP are properties of a single state of a single b-thread and not of the entire specification: the set of all b-threads may, at the same time (i.e., at a given synchronization point), request and/or wait for events controlled by multiple components, but if no single b-thread is controlled by two components, this will not force an inter-component decision. However, at any synchronization point in any component (which means synchronization of all b-threads in that component), if a single b-thread is in an ICDP, the ESM will synchronize the entire component with the other affected components.

When at an ICDP, the actual joint decision of multiple, already-synchronized ESMs, can be performed, e.g., via a distributed leader election protocol [10]. Once a specific ESM is selected as the leader, it chooses the next triggered event based on the local requested and blocked events in its current state.

Note that Definition 4 mandates that the requested-events set not be empty. This restriction reduces the scope of when an ICDP is called for. Consider for example a logger scenario that, oblivious to any synchronization implications, waits for all possible events in the system and writes the relevant data to a log file (without requesting any behavioral event). Without the requirement that at least one event be requested by the b-thread causing the ICDP, such a simple logger would cause the entire execution to synchronize at every event selection. However, this feature, which enables more asynchronous execution, has its price. E.g., two such simple logger scenarios running in two separate components may observe differently the order of a given sequence of events. The issue of seeing

different orders may be resolved either by programming the application such that it induces an ICDP only when it is called for explicitly by the requirements, or by order-enforcing infrastructure, such as the one as described below in Subsect. 5.5 and Assumption 1.

5.5 Equivalence to Centralized Executions

As described above, given a centralized behavioral model M over an event set Σ and a strict event separation $\{E_1, \dots, E_k\}$, our distribution process produces a set of component models $\{C_1, \dots, C_k\}$, whose execution then follows a very particular protocol. We would like to prove that, under certain assumptions, this distributed model behaves like the centralized model, i.e. the set of all possible executions of the distributed model is identical to those of the centralized model.

First, we present the following assumptions.

Assumption 1 (Strict and Total Event Ordering). Given $\mathcal{D}(M, S)$ (Definition 2), we assume that there exists a strict total ordering of all selected events, and this ordering is global and visible to all components (see Sect. 4). I.e., for any pair of events a, b selected by any one or two components, exactly one of the following is true:

- a happened before b , or
- b happened before a

and, all components observe these events in the same order.

Stating the above more formally, we assume that in each component model $C_i = \text{project}(M, E_i)$, the event queue described in the R&P execution semantics is subsumed by a virtual queue, termed VQueue and denoted \hat{Q}_i , with the following properties as well as communication and execution semantics. After an event e is selected by a component C , the event is pushed atomically and simultaneously onto all VQueues of all components (including the one where it was selected). Notice that the atomicity here regards all pushes onto all queues, and any event selection or other important behavioral processing action (including another collective push) occurs either before or after such a collective-push action of one selected event. Each component processes events by popping them, one at a time, from its VQueue, and announcing the event to the b-threads running in that component (which are, in fact, all the b-threads in the specification, as modified/projected locally by R&P). The b-threads then change states according to BP semantics and resynchronize locally. The next event selection at this component can occur at any time during this process as long as all events previously selected *by this component* (and pushed onto its VQueue and onto the VQueues of all the other components), have been popped from the local VQueue \hat{Q}_i and fully processed. However, the local VQueue does not need to be empty when the event selection occurs, i.e., it may contain events that were pushed onto it by other components, since the previous local event selection.

One may consider this assumption as limiting the class of applications covered by the formal argument to those where notification of events to components is

serialized by some virtual central controller, and, where each component waits for the arrival of all events that were triggered in any component after its own last event selection, before the next event selection. In Sects. 8 and 9 we discuss why these limitations are not of great concern and do not diminish from the power of R&P and of reduction of synchronization requirements.

Assumption 2 (No Reliance on Cross-component Blocking of Already-enabled Events). Let $\mathcal{D}(M, S)$ be a distributed model that is being executed. For a given component C_j , let \hat{Q}_i be the totally ordered set $\{e_1, e_2, \dots, e_m\}$, i.e., these are the pending events in its VQueue. Let $r = \langle q^1, \dots, q^n \rangle$ be the component's current program cut (Definition 1). The component's enabled events are:

$$E_r = \left(\bigcup_i R_j^i(q^i) \right) \setminus \left(\bigcup_i B_j^i(q^i) \right)$$

We assume that popping events from the queue does not remove elements from E_r , i.e.,

$\forall l \leq m$, Let $\hat{q}_l^i = \delta^i(\dots \delta^i(\delta^i(q^i, e_1), e_2) \dots, e_l)$ and

$$E_r \cap \left(\bigcup_i B_j^i(\hat{q}_l^i) \right) = \emptyset \quad (1)$$

This is in line with the discussion of not relying on cross-component blocking of already-enabled events in Subsect. 4.2. In other words, we assume blocking is done sufficiently in advance to avoid race conditions.

Lemma 1. *Under Assumptions 1 and 2, the set of all possible executions (the language) of M is identical to the set of all possible executions produced by the component models $\{C_1, \dots, C_k\}$ when run jointly in a distributed fashion.*

This lemma, which is the main proven result of this work, is of practical importance, as it implies that the proposed R&P distribution process will not cause the model to behave in unexpected ways. As discussed under the Language Equivalence principle in Subsect. 4.1, note that the lemma is about the collection of all runs, and *does not* claim that if the distributed and centralized models are run side-by-side, they will always produce the same run. The main reason is that in a cut where more than one event is enabled, we cannot guarantee that two side-by-side runs of the executable specification will make the same choice; and, this holds independently of whether either of them is centralized or distributed. Given the language equivalence result, one can study and analyze the centralized version of the model (which is far easier for humans to grasp and comprehend, and for tools to analyze) and the conclusions will apply to the distributed setting too. We will discuss some of the implications of this result in Sect. 9.

Note that for the lemma to hold we also implicitly assume that each enabled event has a positive probability of being selected. If the event selection is *unfair*, in the sense that it always selects certain events and not others in particular

situations, then the lemma will not hold. We do not consider this assumption to be a major constraint on the kind of applications supported by R&P.

Proof of Lemma 1. Assumptions 1 and 2 shape the rest of the proof. Components select events based on standard BP execution semantics applied to the replicated-and-projected specification. Those selected events are immediately pushed into all the event queues of all components. This operation is instantaneous and defines some global order among the selected events. We do not define when components pop, announce, and process events from their event queue, but simply assume that they do so at some point, and soon enough as to not violate Assumption 2.

Claim 1. In a distributed execution of $\mathcal{D}(M, S)$, if at any point in time all components empty their event queues \hat{Q}_i (processing the events), then the cuts of all component models are at the same state.

Proof. Given a component model $C_i = \text{project}(M, E_i)$ and its event queue \hat{Q}_i , let $\{e_{l_1}, e_{l_2}, \dots, e_{l_m}\}$ be all the events, in order, popped from the queue and processed by the component since the execution started. By Assumption 1, the indices $\{l_1, l_2, \dots\}$ are identical for all components. And, since we assume that selected events are pushed into all event queues instantly, once components empty their queue the total count of processed events is also the same for all components.

While it may be obvious that at any instance at most one event will be selected, in exactly one component (and all components will eventually see this event), when considering possible causes of divergence it is useful to notice that:

1. Given that components are, in general, not synchronized, their event selections are always strictly ordered. The event selection in one component is always before or after any event selection in any other component.
2. In a given cut in a given component, if (after R&P) multiple events are enabled, then:
 - (a) If these enabled events are controlled by this same component, then this is the only component in which they can be enabled. The one event that will be chosen by this component from this set will be visible identically to all components.
 - (b) If the enabled events are controlled by multiple components, then the cut meets the requirements for ICDP, and all the relevant components are also synchronized at the cut at hand; a single event will be chosen via an agreed-upon decision, made for all of them.

Therefore, all components process the same totally ordered set of events $\{e_{l_1}, e_{l_2}, \dots, e_{l_m}\}$.

Observe that in the execution of $\mathcal{D}(M, S)$ all components begin at the same initial program cut $\langle q_0^1, \dots, q_0^n \rangle$, and after m steps a projected b-thread BT_j^i in component C_j transitions to some state $\hat{q}_m^i = \delta^i(\dots \delta^i(\delta^i(q_0^i, e_{l_1}), e_{l_2}) \dots, e_{l_m})$. By definition of the projection process, the δ^i functions are identical across

components, and hence all projections of each thread proceed to the same state, $\forall i, r : \dot{q}_m^r = \dot{q}_m^i$. Therefore all component end up in the same cut. The claim follows. \square

Corollary 2. *Given a distributed model $\mathcal{D}(M, S)$, all the components process the same totally ordered set of events.*

Proof. Follows immediately from Claim 1. \square

Using Corollary 2 we can talk about the sequence of events processed by $\mathcal{D}(M, S)$, as all its components process the same sequence (albeit they might do so at different speeds).

We now define what the formal language generated by a behavioral model is, and prove that the languages of the distributed model and of the undistributed model are the same.

Recall that for an undistributed model M an *enabled event* at some program cut is an event that is requested by some b-thread and is not blocked by any of the b-threads. Recall also that under R&P all components run all b-threads but requesting and blocking of events take place only in components that control these events. We thus extend the *enabled event* term to a distributed system $\mathcal{D}(M, S)$ as follows:

Definition 5. *In a distributed model $\mathcal{D}(M, S)$, an enabled event is one that is requested by some b-thread of some component in which all b-threads are presently synchronized (i.e., a component that is at a cut), and, is not blocked by any b-thread in that component.*

Definition 6. *Let $\Delta(r, e)$ denote the program cut transition function, where r is a program cut and $e \in \Sigma$ is an event. Δ is fully defined by the b-threads state transition function δ^i as follows: for $r = \langle q^1, \dots, q^n \rangle$, $\Delta(r, e) = \langle \delta^i(q^1, e), \dots, \delta^i(q^n, e) \rangle$.*

Definition 7. *The language L of a behavioral model M denoted $L(M)$ is a set of words defined over the alphabet Σ . A word $w = e_1 e_2 \dots$ is in $L(M)$ if its letters constitute a legal run of M ; i.e., if we begin in the initial cut and apply Δ according to the sequence of events in w , the next event is always enabled in the current cut.*

The language of the distributed model $\mathcal{D}(M)$ is defined similarly. A word w is in $L(\mathcal{D}(M, S))$ if and only if there exists a run of $L(\mathcal{D}(M, S))$ where the components select the totally ordered set of events in w . (We assume that the environment is incorporated into the behavioral model as b-threads that non-deterministically request environment events.)

The equality between $L(M)$ and $L(\mathcal{D}(M, S))$ will follow from the following claim:

Claim 2. $L(\mathcal{D}(M, S)) \subseteq L(M)$

Proof. At any time during the execution of distributed model $\mathcal{D}(M, S)$, the enabled events are as determined by the cuts of those components that are presently in a cut (in fact, one can also conveniently assume that a cut transition in a component is always atomic in the sense that a component can only be observed when in a cut, yet not all components may be in the same cut at the same instance of time). As components cannot block external events, the set of enabled events at a given instance is the union of sets of enabled events of all components which are in a cut.

We will denote by E_m^M the set of enabled events of a centralized model after selecting $m \geq 0$ events. Likewise, we denote $E_m^{D_j}$ the set of enabled events of component C_j in the distributed model $\mathcal{D}(M, S)$ after m events have occurred. We do not specify the number of events $l \leq m$ that were actually popped and processed by the component. $E_m^D = \cup_j E_m^{D_j}$ is defined as the set of enabled events in the distributed model after m events were selected and each component C_j has processed $l_j \leq m$ events. By definition, the set of initial enabled events of M is

$$E_0^M = \left(\bigcup_i R^i(q_0^i) \right) \setminus \left(\bigcup_i B^i(q_0^i) \right) \tag{2}$$

and after m steps the set of enabled events of M is

$$E_m^M = \left(\bigcup_i R^i(q^i) \right) \setminus \left(\bigcup_i B^i(q^i) \right) \tag{3}$$

where $\langle q^1, \dots, q^n \rangle$ is M 's program cut after m events have occurred.

The set of initial enabled events in the distributed model $\mathcal{D}(M, S)$ is E_0^D . Clearly $E_0^D = E_0^M$.

Consider the distributed model $\mathcal{D}(M, S)$ after m steps, i.e., after selection and VQueue-ing of exactly m events as counted collectively in the entire model, and examine an arbitrary component C_j . The component has processed l events, where $l \leq m$, and has $m - l$ events in its VQueue. Specifically, it has processed the sequence of events defined by the totally ordered set $\{e_1, e_2, \dots, e_l\}$, and $\hat{Q}_j = \{e_{l+1}, \dots, e_m\}$.

Let $r = \langle q_j^1, \dots, q_j^n \rangle$ be the current program cut of C_j and let ξ_l be the set of enabled events of C_j after processing $l \leq m$ events.

By Assumption 1, if \hat{Q}_j contains an event selected by C_j then the component will not attempt to select another event, until processing that event, and therefore, effectively, $\xi_l = \emptyset$. Otherwise, at this stage the set of enabled events is defined by:

$$\xi_l = \left[\left(\bigcup_i R^i(q^i) \right) \setminus \left(\bigcup_i B^i(q^i) \right) \right] \cap C_j = \left(\bigcup_i R_j^i(q_j^i) \right) \setminus \left(\bigcup_i B_j^i(q_j^i) \right). \tag{4}$$

Whenever C_j processes any number of the $m - l$ events in \hat{Q}_j , no enabled events will be removed from ξ_l for the following reasons:

- No b-thread will change into a state where it blocks events in $E_l^{D_j}$. This is due to Assumption 2 which claims that the application does not rely on instantaneous blocking.
- No b-thread that requested an event will change into a state where it no longer requests this event as this would imply that this b-thread was simultaneously requesting a local event and waiting for an external one which would then require an ICDP. As discussed before, an event chosen by an inter-component decision is considered as selected by all participating components, and we had assumed that \hat{Q}_j contains no event selected by C_j .

Therefore

$$\forall 0 \leq l \leq m : \xi_l \subseteq \xi_m.$$

By definition, ξ_m is the set of enabled events in component C_j after processing all the m events from the VQueue, therefore, as the VQueue is empty, ξ_m is simply:

$$\begin{aligned} \xi_m &= \left[\left(\bigcup_i R^i(q^i) \right) \setminus \left(\bigcup_i B^i(q^i) \right) \right] \cap C_j \subseteq \\ &\quad \left(\bigcup_i R^i(q^i) \right) \setminus \left(\bigcup_i B^i(q^i) \right) = E_m^M \end{aligned}$$

By the way we defined $E_m^{D_j}$ the following holds: $E_m^{D_j} = \xi_l$ for some $l \leq m$, but as we saw $\forall 0 \leq l \leq m : \xi_l \subseteq E_m^M$, therefore $E_m^{D_j} \subseteq E_m^M$ and the following holds:

$$\forall m \geq 0 : E_m^D = \cup_j E_m^{D_j} \subseteq E_m^M.$$

Therefore $L(\mathcal{D}(M, S)) \subseteq L(M)$. □

Claim 3. The language of a behavioral model $L(M)$ is equal to the language of its distributed version $L(\mathcal{D}(M, S))$.

Proof. We need to show that $L(M) \subseteq L(\mathcal{D}(M, S))$. That is trivial: Assume that a run of $\mathcal{D}(M, S)$ always empties its VQueues instantly as soon as events are pushed. In this case the distributed model would behave identically to the centralized version. Ergo $L(M) \subseteq L(\mathcal{D}(M, S))$.

As $L(\mathcal{D}(M, S)) \subseteq L(M)$ by Claim 2 it immediately follows that

$$L(\mathcal{D}(M, S)) = L(M)$$

□

This concludes the proof of Lemma 1, which also implies that the distributed model *behaves correctly*, i.e., produces executions that are allowed under BP semantics.

6 Per-component Timescales

As explained earlier, in a centralized behavioral model, all b-threads must synchronize in order for the ESM to announce the selected event. The b-thread that takes the longest to reach its synchronization point (e.g., because it performs slow local calculations or writes to a file) forces the rest of the b-threads to wait until it synchronizes. This lockstep execution thus results in the slowest b-thread dictating the timescale for the whole system. This is a common issue in behavioral models that involve multiple scenarios operating on different timescales (see, e.g., [17]), and it also applies to our distributed variant of BP: for example, a slower component might experience delays before broadcasting events that a faster component depends on, forcing the latter to wait. Furthermore, external events can “pile up”, increasing the processing time of future event selections and delaying the selection of potentially crucial events.

In this section we discuss how to allow the generated components to operate efficiently on different timescales.

Previous work [17] has tackled this difficulty in a variety of ways. One approach in [17] introduced an *eager execution* mechanism for behavioral models. This technique lessened the severity of the problem by sometimes allowing the ESM to trigger an event even when some of the b-threads have not yet synchronized. Our distribution technique lends itself naturally to this kind of idea, because within a given component, we know that b-threads controlled by other components, which have not synchronized yet, cannot block local requested events. Thus, by applying a method similar to eager execution, the ESM does not have to wait for b-threads which wait only for external events (such b-threads may be in the original specification, or they may be the projected version of b-threads with event requests changed to waiting for events).

In our distributed setting, eager execution can be applied as follows. Given a behavioral model $M = \{BT^1, \dots, BT^n\}$ and its distributed component models $\{C_1, \dots, C_k\}$, let $q \in Q^i$ be a state in which b-thread BT^i is not controlled by component C_j . Observe BT_j^i , i.e., the copy of BT^i that is running in component C_j . Because BT_j^i is not controlled by C_j , it does not request or wait for any local events and must be waiting for an external event e controlled by some other component C_m . In other words, until such time as e is triggered by C_m , thread BT_j^i will not affect local event selection in component C_j . In such situations we propose to temporarily *detach* thread BT_j^i from its local ESM, effectively allowing event selection in component C_j without considering BT_j^i . This allows component C_j to operate in its own pace, while BT_j^i can be regarded as temporarily operating in the same time scale as C_m . Whenever e is finally triggered and BT_j^i reaches a new state \bar{q} in which it is controlled by C_j , it is reattached to the local ESM. This technique readily enables different components to simultaneously operate at different timescales.

To support eager execution within our distributed framework, the external event queue within each component model needs to be decoupled from the distributed ESM. Instead, each b-thread in the component receives its own external-event queue, and at each synchronization point pops all external events and selects them one at a time. The changes in the BP execution engine are summarized as follows:

- Each b-thread should flag itself as synchronized or unsynchronized with each bSync call, depending on the state.
- A separate event queue is created in each b-thread, thus allowing b-threads to process external events independently of the local ESM. A b-thread that arrives at a state first empties its event queue by repeatedly popping and selecting an event.
- External events received at a given component are injected into all the b-thread event queues by the component’s BP execution engine. B-threads that are already awaiting the local ESM are notified to handle the external events.

7 Example and Evaluation

We now describe in more detail the distributed application upon which we carried out our evaluation. Specifically, and as introduced in Sect. 3, we implement a drone-based light show as follows. Each of four drones has a green light and a red light. Initially, the drones “do the wave”, each flashing its green light briefly, in turn. This is implemented by the scenario in Algorithm 1. The scenario in Algorithm 2 shows the projection of the scenario in Algorithm 1 to Drone1.

```

i=0;
while true do
  bSync(R = {FlashGreen((0 + i)%4)});
  bSync(R = {FlashGreen((1 + i)%4)});
  bSync(R = {FlashGreen((2 + i)%4)});
  bSync(R = {FlashGreen((3 + i)%4)});
  nextEvent = bSync(R = {NW0, NW1, NW2, NW3});
  i = indexOfWave(nextEvent);
end

```

Algorithm 1: Pseudocode of a BP scenario demonstrating a simple undistributed wave example. For each bSync synchronization point, R is set requested events. The events NW0 through NW3 indicate a request the start a new wave at the corresponding component. These events are requested after each full cycle, and BP event selection then decides which component starts the new wave. The method *indexOfWave* translates an event NW*i* to the index *i*.

```

i=0;
while true do
  bSync(W = {FlashGreen((0 + i)%4)});
  bSync(R = {FlashGreen((1 + i)%4)});
  bSync(W = {FlashGreen((2 + i)%4)});
  bSync(W = {FlashGreen((3 + i)%4)});
  nextEvent = bSync(R = {NW1}, W = {NW0, NW2, NW3});
  i = indexOfWave(nextEvent);
end

```

Algorithm 2: Projection of the scenario of Algorithm 1 onto the component Drone1. Notice that requested events controlled by other components become waited-for (represented by the W sets).

Our example is a slightly richer scenario, coded as a behavioral program written in C++. The four drones (labeled Drone0 through Drone3) participate in “a green wave”, starting with Drone0. After the conclusion of two full cycles, the drones jointly decide which of the drones will start the next wave. The next wave will, again, last for two full cycles, and the entire process repeats five times. For now, the entire specification consists of a single scenario. In this implementation, the light-flashing events are labeled as FlashGreen0 through FlashGreen3, each representing the flashing of the light in the respective drone, in either a centralized or distributed implementation. The selection of the drone that will start the next wave is carried out by the scenarios requesting four “new wave” events, NW0 through NW3, and the BP event-selection mechanism arbitrarily selecting one of these events. We then associate each of the FlashGreen and the NW events with the corresponding component. In this simplified example the duration of the flashing of each light is implemented in a delay (sleep) of 250 msec in the b-thread that is about the request a FlashGreen event.

For simplicity, this implementation uses a *centralizer* component and does not implement a leader-election mechanism. The centralizer is an infrastructure component which is responsible for: (i) receiving notifications of events triggered in any behavior components, and broadcasting this information to all other components, and (ii) managing joint decisions, by receiving notices from any component ESM that wishes to synchronize, which include the sets of requested and blocked events, waiting for all other components to reach their corresponding state, selecting an event which is requested and not blocked, and notifying all components of the selection. Note that the centralizer serves only in simulations and studies of the approach, and that in real distributed implementations broadcasting can be performed by a variety of techniques (including the above), and joint decisions can be reached by classical distributed-processing solutions, such as leader election.

At this point it is important to distinguish between the concepts of classes and objects and the concept of components as used here. Events may be self-standing entities, or they may be associated with objects. In our example, each drone is a component, and objects may reside within a component, or may span

multiple component. Such objects can be, e.g., a drone controller, a drone light, a wave effect (which can have a beginning and end events, or a color property) or an entire light show. As can be seen in the example given in Algorithm 2, each component executes “the entire specification”, in this case, this one scenario. In the distributed implementation, when scenarios request or wait for FlashGreen events, they do not synchronize, but when they request the four new wave events, they all synchronize. This results in a partially synchronized execution, which mimics the centralized execution but does so with less inter-component synchronization.

We compare our target, partially synchronized execution of a specification created with the replicate-and-project implementation (R&P), with a fully synchronized distributed execution (abbr. FS), where each component executes the same specification, and they synchronize with every event selection. The decision in each component whether to actually turn on its own light following its respective FlashGreen event is left as a small implementation detail, i.e., the light-switch actuation method skips the operation if there is no direct connection with the device. Both implementations execute the same one-scenario specification, replicated over four components. The total number of events that occurred, all of which were broadcast to all components, is 44—the same for FS and for R&P (five repetitions of two four-event cycles, and four joint decisions). In the R&P however, only four of these required synchronization. The total execution time was the same in both cases, dominated by the duration of the light flashes, but if synchronization delay is artificially increased, total execution time is increased accordingly (e.g., a 100 msec delay purely due to synchronization, in addition to any ordinary communication delay, would add 400 msec to the duration of each cycle of this single wave).

We now extend our mini-light-show example with another wave of flashing lights. We add a scenario in which, starting with Drone2, each of the drones briefly flashes a red light, in its turn. This multi-cycle wave continues uninterrupted and with no change until the ten cycles of the green wave terminate. The delay (sleep) before requesting a FlashRed event is 1000 msec. When multiple events are requested e.g., both a FlashRed together with FlashGreen or NW, the ESM selects an event at random. The forty FlashGreen events in the ten-cycles determine the beginning and end of the run, and the number of FlashRed events selected during this time varies. Since we are presently more interested in understanding the underlying effects than in measuring improvements over a large number of runs, we suffice with this artificial example. To highlight these effects we show in Table 1 a comparison of the two cases when in both FS and R&P, 44 FlashGreen events were triggered.

The basic communication delay in these experiments is set to 50 msec, resulting in 100 msec delay for broadcasting an event occurrence via the centralizer.

Some interesting explanations and observations include:

- In FS, at every synchronization point, both a FlashRed event, and, either a FlashGreen or NW events are enabled. This is true regardless of sleep delays and number of components. Hence in such runs, on average, half of the events

will be FlashRed. By contrast in R&P, FlashRed is enabled in a component together with one of the other two events in a way that depends on lengths of sleep delays and on the number of components in the cycle, yielding, in our case fewer FlashRed events during the run.

- Common to all runs is a $40 * 250$ msec taken by the FlashGreen events, plus $4 * 100$ msec minimum number of joint decisions, plus about 3s of overhead (total of 13–14s).
- The 41 s duration of R&P is the result of adding to the above ~ 13 s $28 * 1000$ msec FlashRed events.
- The 67s duration of FS is the result of adding to the above 41s of R&P $17 * 1000$ msec of additional FlashRed events and $85 * 100$ msec communication delays due the additional synchronizations, all of which had to occur during the same ten cycles of the green wave.
- Even though the total number of events triggered in R&P is less than in FS, the per-second event rate is higher.
- In the worst case, the performance of a distributed system resulting from an R&P distribution process will be the same as when a replicated specification executes without local changes in all components, with full synchronization at every event selection.

Table 1. Comparing an execution of a fully synchronized (FS) implementation of a two-scenario specification in a four-component configuration, to an execution of the partially synchronized replicate-and-project implementation (R&P). See discussion in the Sect. 7 [37].

Measure	FS	R&P
Number of FlashGreen event notification broadcast	40	40
Number of FlashRed event notification broadcast*	45	28
Number of “new wave” event notification broadcast	4	4
Total number of events	89	72
Total number of Inter-component synchronizations	89	4
Run duration (in seconds)	67	41
Events per second	1.32	1.75

While the above examples illustrate and quantify the kind of savings resulting from reduced synchronization, we must note that the synchronization delay itself is sometimes not the main issue. For example, if we were to replace the FlashGreen event(s) in our design with, e.g., pairs of TurnGreenLightOn and TurnGreenLightOff events, all scenarios might have had enough time to synchronize with each other following the event TurnGreenLightOn, in parallel to waiting for the time ticks that would signal the end of the shining of the light. A relaxed synchronization approach, separating the scenarios of the two waves into separate modules within each component, would further streamline an otherwise

fully synchronized implementation. Nevertheless, the reduced inter-component synchronization still helps in simplifying the designs, and in enhancing system robustness. For example, consider recovering from loss of a drone, due to battery running out, while “the show must go on”. It is much easier for all drones to observe and react to delays in other drones’ behavior, when they are fully functional as opposed to waiting in a global synchronization point (even when the latter is enhanced with timeout facilities as in [18]).

8 Related Work and Comparison

Distributed system have been the subject of extensive research and studies; see, e.g., [3,32]. In general most approaches that aim to distribute a centralized system fall into one of the following classes:

1. The distribution process employs a kind of orthogonalization (or partitioning) process that decomposes the system into independent, orthogonal partitions that form the distributed system. This might be done with some user intervention and input or using a fully automatic process. The resulting executable partitions model parts of the system which can be ran, in parallel, as a distributed system without ever requiring to synchronize. Typical examples include the parallelization of an abstract computation, or the execution of a multi-agent system where an agent may wait for another agent’s messages, or may even coordinate a joint application decision, but they cannot in any way depend on synchronizing with each other their own internal computations and processes.
2. The executable partitions that form the distributed system are given in advance, and they do not map to logically-orthogonal parts of the specification. Instead, they are formed to satisfy other constraints (physical properties, performance, etc.) Unlike systems with orthogonal partitioning, some synchronization might be required to ensure the distributed system has largely the same function as the original one. A typical example would be a distributed database whose components are defined by physical machine capacities and boundaries. The component synchronize as part of their underlying computation, to ensure properties such as atomicity, consistency, isolation and durability (ACID).

Each class has a unique difficulty: Orthogonalization or synchronization. In terms of system design synchronizing distributed systems enjoy a larger degree of freedom in the way the distributed partitions can be chosen. Behavioral specifications generally do not expose orthogonal partitions that map to the physical parts or properties of the system, or at times, none at all. Performance-wise, the trade-off between an orthogonal and a non-orthogonal approach can be seen as the trade-off between distribution performance as opposed to execution performance.

Within the realm of behavioral programming, the research in [28] suggests an approach for orthogonal distribution, where the distributed system consists

of multiple, manually design, independent programs, termed *behavior nodes* (*b-nodes*), each with its own set of internal events. As this is an orthogonal approach, those b-nodes never need to synchronize with each other. Similar to our approach the b-nodes communicate by external events, however those events require manual translation to and from internal events. While in [28] the distributed system is generated by a manual partitioning of a model into multiple b-programs, [15] proposes a synchronizing approach for distributing BP models by manually partitioning the b-threads of a single b-program into modules, where each module runs its set of b-threads and synchronizes with other modules upon choosing events that might matter to other modules. The set of events that require synchronization as well as which modules each events needs to synchronize with is known a priori. The research in [15,28] contains examples of an orthogonal distribution approach and a synchronizing one, respectively, in behavioral programming. However, in both approaches the component structure is dynamic and implied by the specification, in contrast to the present paper where the component structure is dictated by the physical structure of the system and external events emerge naturally and automatically from internal events. Furthermore our approach supports more general designs, inter-component scenarios and fine-grained synchronizations when scenarios give rise to inter-component decisions.

A different framework for the distributed execution of scenarios is presented in [12]. The approach there is similar to the one in this paper in that the distributed components can each choose to execute events that they are responsible for, and selected events are broadcast to all other components. Further, a coordinator component in [12] forces the situation where, as in Assumption 1, all components observe a single event order. The main issues with this implementation relative to R&P are that it requires that individual scenarios are written to not have states where events of multiple components are enabled. By contrast, R&P automatically coordinates all components when reaching a state where a joint decision is required, and it allows components to advance asynchronously when possible, and in particular, after locally selecting an event. An advantage, though, of the implementation of always enforcing a common event order in [12] is that it avoids the risk of sensitivity to different event orders. While Lemma 1 relies on such enforcement for the proof, R&P in general allows also for applications that forego this requirement, and solve order-dependencies in application-specific means. However, we must note that the actual reliance in the implementation on a physical centralized coordinator for the entire distributed system carries many disadvantages both in performance and robustness. The introduction of single order assumption in the proof of Lemma 1, can be seen more as an abstraction—a requirement that is either guaranteed by some efficient and robust means or by application-specific properties.

A more general, automatic handling of event-order dependencies in R&P, and possible generation of additional ICDPs, is left for future research, e.g. using formal methods, as discussed in Sect. 4.

The research in [14] describes (though without an implementation) a mechanism for the distributed execution of scenarios with *dynamic role bindings*. There, synchronization is done only among relevant components, as determined dynamically.

There has also been work on synthesizing scarcely-synchronizing distributed controllers from scenario-based specifications [4]. Distributed finite automaton controllers can be synthesized from scenario specifications in a way that greatly reduces communication overhead compared to previous approaches, especially compared to the broadcasts of events as also suggested in this work. However, the synthesis procedure is computationally complex and does not scale well as specification and system size increase. In [9], the authors study a similar problem and present an approach for synthesizing executable implementations from specifications given in a distributed variant of LSC, termed *dLSC*.

Another work related to distribution of centralized scenario-based models (but outside of the realm of BP) is [34], which presents a synchronizing approach for distributing workflow specifications. This work exposes domain-specific knowledge in order to be able to generate automatically distributed partitions and synchronization semantics such that the resulting distributed system preserves the execution semantics of the original centralized version.

Outside the scope of scenario-based modeling [7] is an example of distribution of systems modeled using Petri Nets, specifically High Level Timed Petri Nets (HLTPN). This research uses the orthogonalization approach where the HLTPN is decomposed into subnets connected by *shared places*, nodes that are common to multiple subnets. Arcs are not allowed to cross subnet boundaries, ensuring that the decomposing is an orthogonal partitioning of the net. The shared places can be seen as global memory, shared between multiple subnets, used to control firing of transitions, however there is no synchronization between the subnets.

Further non-scenario-based research discusses the trade-off between performance optimization and communication minimization in parallel and distributed settings has been studied extensively. These two conflicting goals are discussed, e.g., in [5, 39]. In [38] the author suggests imposing certain limitations on the communication between the components, thus allowing for execution-time optimization to be performed during compilation.

It is interesting to note that distribution approaches that rely on scenario-based specifications typically exploit the execution semantics of the modeling language to generate synchronized distributed systems. Meanwhile, non-scenario-based approaches generally employ a form of orthogonalization, and usually rely on domain-specific knowledge or on high-level temporal specifications to facilitate the distribution. As discussed above, orthogonalization approaches are more rigid and are not always possible, feasible or applicable. To our knowledge there is scarcely any research that involves generating a non-orthogonal distributed system from a generic non-scenario-based specifications. It appears that despite (and perhaps due to) the simplicity of behavioral programming and of scenario-based specifications in general, it is generally amenable automated decomposition and distribution.

9 Discussion and Future Work

Previous research on scenario based programming has shown the great importance of formal methods and tools in ensuring that the resulting models, composed of many individual scenarios, perform as intended as a whole. Past efforts have yielded a large portfolio of tools for model checking [24], automatic repair [21,30] and compositional verification [16,31], and have even indicated that scenario-based programming may be more amenable to formal analysis than other modeling approaches [19,22].

Given the above, applying formal analysis in the distributed case seems even more vital, as distributed models are inherently more difficult for humans to comprehend than centralized ones. Fortunately, Lemma 1 enables us to immediately apply existing tools in our setting. Because the centralized and distributed models present the same behavior, it is possible to apply existing approaches to the centralized version and use them to draw conclusions regarding the distributed case.

Future research on new applications of formal methods to distributed implementations can also distill situations and “critical states” where special handling is needed. E.g., identify when there are special dependencies on observing a particular event order, and devise solutions that are automatic, reduce synchronization, and reduce the need for a total strict event order, and where the equivalence of the resulting distributed execution to the centralized one can be proven. For example, in the present implementation, an application that waits for two events from two different components in any order, and then transitions into the same final state, depends on guaranteed event order, and/or on two ICDPs, where, in fact, neither is required. This research will also include proofs for correctness of different distribution procedures—e.g., that in cases where the application does not depend on particular event order, a particular distribution method which does not guarantee Assumption 1 still works correctly. For example, cars arriving at an intersection, each detecting all other cars in their environment, do not have to rely on observing identical event orders. We wish to devise a distribution methods for scenario-based specifications for handling such situations, and prove their correctness, namely, that the cars executing these scenarios in a distributed manner indeed cross the intersection safely.

Nonetheless, in a distributed environment there are some hazards that do not appear in the fully-synchronized model, and may thus be overlooked by existing tools:

- **Inter-component Deadlock.** An inter-component deadlock occurs when a component C has no enabled local events that it can trigger, and is thus waiting for certain external event(s). However due to various reasons, these external events may never arrive. For example, the reason might be that another component is actually waiting for an event that C needs to trigger. Note that a situation where a component is waiting for events local to a crashed component is not an inter-component deadlock, but a soft deadlock, as restarting the failed component might resolve the issue.

- **External Event Queue Overflow.** When a component repeatedly takes longer to process external events than it takes the other components to trigger and broadcast these events, could result in exceeding the memory available for the external event queue. An example of this could be a logger component that takes too long to post its log entries to a remote location.
- **Latency.** Communication delays can cause poorly-designed systems to exhibit undesired behavior. As we discussed in Sect. 5.5, Lemma 1 does not hold when latency is too high, and so such errors cannot be detected by existing tools.

We are working on extending the presently available techniques to handle the issues listed above. For instance, in the latency case an improved model-checking algorithm might simulate a realistic latency for external event communication, depending on the communication method used (e.g., wired communications over a local network will have a much lower latency than a satellite connection). We are also exploring the use of quantitative approaches to formal verification to attempt and derive bounds on the maximal size a queue can reach, given certain constraints on the broadcast and processing times of system components.

In the context of inter-component deadlock, one approach for recovering from component failure or missed messages could be adding state information to the external events, permitting components that missed a transition to “fast-forward” to the correct state in a scenario. Another direction could involve having multiple instances of critical components, for redundancy.

As an additional future work direction, we would like to study approaches to choosing a strict event separation. While the components are usually derived manually from physical system requirements, at times it might be desired to delineate their boundaries automatically based on other criteria. One approach is to use clustering algorithms that take as input a function f that assigns, for every two events $e_1, e_2 \in \Sigma$ a correlation value $f(e_1, e_2) \in [-1, +1]$. The clustering algorithms then attempt to partition the events into a strict separation into k components (with k either known or unknown beforehand), such that two events are in the same component if their correlation is high and are in separate components if their correlation is low. While this problem is known to be NP-Complete, it can be approximated up to a log-factor [2].

10 Conclusion

The replicate-and-project approach transforms a centralized scenario-based specification so that it can be executed in a distributed configuration, by creating component-specific variations, based on each component’s capabilities. We have shown that the resulting distributed models behave similarly to the centralized model from which they originated. This important property allows us to carry out most of the modeling work, including testing and analysis, in the centralized setting, which is easier to model-check and reason about. The projected models retain the naturalness and incrementality traits of behavioral programming. In their avoidance of excessive synchronization, they improve robustness

and the ability to model systems with multiple time scales. In addition to the advantages of this approach in streamlining design and improving performance, it captures the more general fact that distributed operations that are robust and efficient often involve the sharing of knowledge between components, such that each of them knows at least some of the rules that control the behavior of the others - a concept whose applicability me go beyond scenario-based/behavioral programming.

Acknowledgements. This work is funded by grants from the German-Israeli Foundation for Scientific Research and Development (GIF) and from the Israel Science Foundation (ISF).

References

1. Alexandron, G., Armoni, M., Gordon, M., Harel, D.: Scenario-based programming: reducing the cognitive load, fostering abstract thinking. In: Proceedings of the 36th International Conference on Software Engineering (ICSE), pp. 311–320 (2014)
2. Bansal, N., Blum, A., Chawla, S.: Correlation clustering. *Mach. Learn.* **56**(1–3), 89–113 (2004)
3. Błażewicz, J., Ecker, K., Plateau, B., Trystram, D.: Handbook on Parallel and Distributed Processing. Springer, Heidelberg (2013). <https://doi.org/10.1007/978-3-662-04303-5>
4. Brenner, C., Greenyer, J., Schäfer, W.: On-the-fly synthesis of scarcely synchronizing distributed controllers from scenario-based specifications. In: Egyed, A., Schaefer, I. (eds.) FASE 2015. LNCS, vol. 9033, pp. 51–65. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46675-9_4
5. Cheng, Y., Robertazzi, T.: Distributed computation with communication delay (distributed intelligent sensor networks). *IEEE Trans. Aerosp. Electron. Syst.* **24**(6), 700–712 (1988)
6. Damm, W., Harel, D.: LSCs: breathing life into message sequence charts. *J. Formal Methods Syst. Des.* **19**(1), 45–80 (2001)
7. De La Puente, J.A., Alonso, A., León, G., Dueñas, J.C.: Distributed execution of specifications. *Real-Time Syst.* **5**(2), 213–234 (1993)
8. Eugster, P., Felber, P., Guerraoui, R., Kermarrec, A.: The many faces of publish/subscribe. *ACM Comput. Surv. (CSUR)* **35**(2), 114–131 (2003)
9. Fahland, D., Kantor, A.: Synthesizing decentralized components from a variant of live sequence charts. In: Proceedings of the 1st International Conference on Model-Driven Engineering and Software Development (MODELSWARD), pp. 25–38 (2013)
10. Ghosh, S., Gupta, A.: An exercise in fault-containment: self-stabilizing leader election. *Inf. Process. Lett.* **59**(5), 281–288 (1996)
11. Gordon, M., Marron, A., Meerbaum-Salant, O.: Spaghetti for the main course? Observations on the naturalness of scenario-based programming. In: Proceedings of the 17th Conference on Innovation and Technology in Computer Science Education (ITICSE), pp. 198–203 (2012)
12. Greenyer, J., Gritzner, D., Gutjahr, T., Duente, T., Dulle, S., Deppe, F.-D., Glade, N., Hilbich, M., Koenig, F., Luennemann, J., Prenner, N., Raetz, K., Schnelle, T., Singer, M., Tempelmeier, N., Voges, R.: Scenarios@run.time – distributed execution of specifications on IoT-connected robots. In: Proceedings of the 10th International Workshop on Models@Run.Time (MRT), pp. 71–80 (2015)

13. Greenyer, J., Gritzner, D., Katz, G., Marron, A.: Scenario-based modeling and synthesis for reactive systems with dynamic system structure in ScenarioTools. In: Proceedings of the 19th International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 16–32 (2016)
14. Greenyer, J., Gritzner, D., Katz, G., Marron, A., Glade, N., Gutjahr, T., König, F.: Distributed execution of scenario-based specifications of structurally dynamic cyber-physical systems. In: Proceedings 3rd International Conference on System-Integrated Intelligence: New Challenges for Product and Production Engineering (SYSINT), pp. 552–559 (2016)
15. Harel, D., Kantor, A., Katz, G.: Relaxing synchronization constraints in behavioral programs. In: McMillan, K., Middeldorp, A., Voronkov, A. (eds.) LPAR 2013. LNCS, vol. 8312, pp. 355–372. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-45221-5_25
16. Harel, D., Kantor, A., Katz, G., Marron, A., Mizrahi, L., Weiss, G.: On composing and proving the correctness of reactive behavior. In: Proceedings of the 13th International Conference on Embedded Software (EMSOFT), pp. 1–10 (2013)
17. Harel, D., Kantor, A., Katz, G., Marron, A., Weiss, G., Wiener, G.: Towards behavioral programming in distributed architectures. *Sci. Comput. Program.* **98**(2), 233–267 (2015)
18. Harel, D., Katz, G.: Scaling-up behavioral programming: steps from basic principles to application architectures. In: Proceedings of the 4th International Workshop on Programming based on Actors, Agents, and Decentralized Control (AGERE!), pp. 95–108 (2014)
19. Harel, D., Katz, G., Lampert, R., Marron, A., Weiss, G.: On the succinctness of idioms for concurrent programming. In: Proceedings of the 26th International Conference on Concurrency Theory (CONCUR), pp. 85–99 (2015)
20. Harel, D., Katz, G., Marelly, R., Marron, A.: An initial wise development environment for behavioral models. In: Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD), pp. 600–612 (2016)
21. Harel, D., Katz, G., Marron, A., Weiss, G.: Non-intrusive repair of reactive programs. In: Proceedings of the 17th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), pp. 3–12 (2012)
22. Harel, D., Katz, G., Marron, A., Weiss, G.: The effect of concurrent programming idioms on verification. In: Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD), pp. 363–369 (2015)
23. Harel, D., Kugler, H., Marelly, R., Pnueli, A.: Smart play-out of behavioral requirements. In: Aagaard, M.D., O’Leary, J.W. (eds.) FMCAD 2002. LNCS, vol. 2517, pp. 378–398. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-36126-X_23
24. Harel, D., Lampert, R., Marron, A., Weiss, G.: Model-checking behavioral programs. In: Proceedings of the 11th International Conference on Embedded Software (EMSOFT), pp. 279–288 (2011)
25. Harel, D., Marelly, R.: Come, Let’s Play: Scenario-Based Programming Using LSCs and the Play-Engine. Springer, Heidelberg (2003). <https://doi.org/10.1007/978-3-642-19029-2>
26. Harel, D., Marelly, R.: Specifying and executing behavioral requirements: the play in/play-out approach. *Softw. Syst. Model. (SoSyM)* **2**, 82–107 (2003)
27. Harel, D., Marron, A., Weiss, G.: Behavioral programming. *Commun. ACM* **55**(7), 90–100 (2012)

28. Harel, D., Marron, A., Weiss, G., Wiener, G.: Behavioral programming, decentralized control, and multiple time scales. In: Proceedings of the 1st SPLASH Workshop on Programming Systems, Languages, and Applications based on Agents, Actors, and Decentralized Control (AGERE!), pp. 171–182 (2011)
29. Harel, D., Segall, I.: Synthesis from live sequence chart specifications. *Comput. Syst. Sci.* (2011, to appear)
30. Katz, G.: On module-based abstraction and repair of behavioral programs. In: McMillan, K., Middeldorp, A., Voronkov, A. (eds.) LPAR 2013. LNCS, vol. 8312, pp. 518–535. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-45221-5_35
31. Katz, G., Barrett, C., Harel, D.: Theory-aided model checking of concurrent transition systems. In: Proceedings of the 15th International Conference on Formal Methods in Computer-Aided Design (FMCAD), pp. 81–88 (2015)
32. Liu, J., Ahmed, E., Shiraz, M., Gani, A., Buyya, R., Qureshi, A.: Application partitioning algorithms in mobile cloud computing: taxonomy, review and future directions. *J. Netw. Comput. Appl.* **48**, 99–117 (2015)
33. Miller, C., Poellabauer, C.: A decentralized approach to minimum-energy broadcasting in static ad hoc networks. In: Ruiz, P.M., Garcia-Luna-Aceves, J.J. (eds.) ADHOC-NOW 2009. LNCS, vol. 5793, pp. 298–311. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04383-3_22
34. Muth, P., Wodtke, D., Weissenfels, J., Dittrich, A.K., Weikum, G.: From centralized workflow specification to distributed workflow execution. *J. Intell. Inf. Syst.* **10**(2), 159–184 (1998)
35. Ramadge, P., Wonham, W.: Supervisory control of a class of discrete event processes. *SIAM J. Control Optim.* **25**(1), 206–230 (1987)
36. Ștefănescu, A., Esparza, J., Muscholl, A.: Synthesis of distributed algorithms using asynchronous automata. In: Amadio, R., Lugiez, D. (eds.) CONCUR 2003. LNCS, vol. 2761, pp. 27–41. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45187-7_2
37. Steinberg, S., Greenyer, J., Gritzner, D., Harel, D., Katz, G., Marron, A.: Distributing scenario-based models: a replicate-and-project approach. In: 5th International Conference on Model-Driven Engineering and Software Development (MODEL-SWARD) (2017)
38. van Gemund, A.: The importance of synchronization structure in parallel program optimization. In: Proceedings of the 11th International Conference on Supercomputing (ICS), pp. 164–171 (1997)
39. Yook, J., Tilbury, D., Soparkar, N.: Trading computation for bandwidth: reducing communication in distributed control systems using state estimators. *IEEE Trans. Control Syst. Technol.* **10**(4), 503–518 (2002)



Modelling the World of a Smart Room for Robotic Co-working

Uwe Aßmann^(✉), Christian Piechnick, Georg Püschel, Maria Piechnick,
Jan Falkenberg, and Sebastian Werner

Chair of Software Engineering, Fakultät Informatik, Technische Universität Dresden,
Dresden, Germany

{uwe.assmann, christian.piechnick, georg.puschel, maria.piechnick,
jan.falkenberg, sebastian.werner}@tu-dresden.de

<http://st.inf.tu-dresden.de/weir>

Abstract. Robots come out of the cage. Soon, it will be possible to interact with free-standing robots along an assembly line or in a manufacturing workshop (*robotic co-working*). New sensitive robot arms have appeared on the market [1] that slow down or stop when humans enter their context, which creates rich opportunities for collaboration between human and robots. But how to program them? This paper contributes an architectural design pattern to engineer software for robotic co-working with *world-oriented modelling (WOM)*. We argue that robotic co-working always has to take place in *smart rooms* tracking the movements of humans carefully, so that the robotic system can automatically adapt to their actions. Because robotic co-working should be *safe* for humans, robots, and their work items, the robots should enter safe states before harmful encounters happen. Based on the safety automata in the style of [1], we suggest to engineer software for the smart rooms of human-robotic co-working with an explicit *world model*, an automaton of the world's states, and a *software variant space*, a software variant family, which are related by a total activation mapping. This construction has the advantage that the world model is split off the software system to make its construction simpler, avoiding if-bloated code. Also, proofs about the entire smart system can be split into a proof about the world model and a proof obligation for the software variant space. Therefore, we claim that world-oriented modelling (WOM) simplifies the development of robotic co-working applications, leveraging the principle of separation of concerns for improved maintainability and quality assurance.

1 The Trend: Robots Come Out of the Cage

Since two years, sensitive robot arms can be bought which stop when humans touch them. In 2014, the KUKA LBR iiwa appeared, which has been derived from an arm of the DLR robotics institute [1]. While this robot arm has a fast mode—too fast to protect humans during an encounter—it also has a safe mode, in which it stops when its sensors feel a human touch. Similar sensitive

functionality is provided by the UR-10 from Universal Robots [2], the ABB YuMi [3], and the Baxter from Rethink Robotics [4]. These sensitive robot arms have been designed to collaborate with humans, i.e., to take over simple tasks of an assembly line or in a manufacturing workshop. In the following, we will call them *robotic co-workers*, no matter, whether they are mounted stationary or on a mobile platform.

Robotic co-workers are much more flexible than classic robotic assembly cells. They can easily be re-targeted to new tasks by teaching. To mount them in a workshop does not require a lot of time, and mounting can be done online, while the work is ongoing. When they need maintenance or repair, they can be replaced by humans so that such an incident does not stop the entire workshop. Thus, robotic co-workers can be added to current workshops quite flexibly.

Table 1. The costs of a robotic co-worker with a life-time of 7 years and 24/7 operation.

Investment (total)	100k€	50k€	20k€
Investment/hour (24 h, 7 days, 7 years)	1.66€	0.82€	0.41€
Maintenance/hour	0.11€	0.11€	0.11€
Energy/hour	0.30€	0.30€	0.30€
Total	2.07€	1.23€	0.82€

Robotic co-workers have, already now, a low cost per hour, as a simple calculation shows (Table 1). Industrial robots have a depreciation of 7 years and are capable to run 24 h, 7 days a week. At the moment, these robots cost between 20k€ and 100k€, but even in the latter case, investment costs are below 1.7€/hour [5]. (Of course, if the robot only runs for 12 h a day, the costs are doubled.) Robotic co-workers need energy, but these costs amount to only a few kilowatt hours per day. Also, they need maintenance, which diminishes the number of their active hours and adds the costs of a technician. However, their total operational costs per hour are still quite low, around 2€, and they sink further, because the investment costs sink. Thus, in only a small number of years, sensitive robotic arms will provide an affordable companion to humans in workshops of small and medium size companies. Unfortunately, programming them encounters several problems.

2 State of the Art

Before we discuss these problems, this section presents several related approaches introducing the terminology required for the WOM pattern, from the areas of adaptive systems and context-oriented programming.

2.1 Adaptive Systems

Dynamic Aspect-Oriented Programming (Dynamic AOP). Dynamic AOP languages can be used to implement adaptive systems. A dynamic aspect system is able to add or remove crosscutting code slices at run-time to an application. When such a dynamic aspect is changed, potentially all objects of the application are being adapted to the new situation. In this paper, we are mainly interested in the language ECaesarJ [6], which provides *component classes*, dynamic aspects, that can be added to and removed from a running application. Additionally, the language allows for the specification of objects as finite state machines (*state-machine objects*), which can describe the configuration of an application. These features can be used to realize the WOM pattern.

Dynamic Software Product Lines. A dynamic software product line (DSPL) [7] maintains a set of *variants* of a software system (e.g., a set of dynamic aspects), which can dynamically be switched on and off, but which are additionally related to a set of *hierarchical features* of a *feature model* specifying constraints for their variation. Thus, a DSPL is a simple adaptive system which can adapt to dynamically changing requirements represented by features. A *Featured Transition System (FTS)* annotates a finite automaton to a feature model of a DSPL, specifying the dynamics of the switching of the features and their related variants by a regular state space [8]. FTS allow for verifications of qualities of the software variants with model checking, i.e., reachability analysis on finite automata. The WOM pattern generalises FTS to automata over the states of the room in which the robotic co-working takes place.

Hybrid Automata. A hybrid automaton [9] couples a finite state automaton with a set of equations for a continuous model of a physical system. The automaton is discrete, the equations are continuous, thus the automaton is *hybrid*. Hybrid automata are useful to describe transitions of continuous behaviour of a physical system, based on external events or time conditions. Thus, hybrid automata can be regarded as simple adaptive systems with a finite number of continuous states. The approach we introduce in this paper is inspired by hybrid automata in so far that the finite automaton forms the skeleton of the robotic co-working application, but it does not insist on the coupling continuous equations, instead on the coupling to hand-programmed variants of a variant family.

Self-Adaptive Systems (SAS) and M@RT. Self-Adaptive Systems (SAS) have a metalevel architecture, which (self-)adapts the application to changes, for example, on changes of context [10]. A specific subclass of SAS are systems that maintain *models at runtime (M@RT)* [11] and use the M@RT to decide on adaptations. An example is the system Genie [12], a flood management system, in which the configuration of the software depends on a finite state transition system describing all architectural states of the flood management system. Changing a state in this state machine switches to another variant of the flood management system. Thus, the states denote global configuration states of the

overall system, not of its selected features, as in a FTS. WOM can be seen as an adaptive M@RT system maintaining a model of the smart room for robotic co-working at runtime. However, the architectural pattern of WOM does not need to be represented as a metalevel architecture, but can be integrated into the base level as in hybrid automata.

All these adaptation techniques should support well-defined *switching points*, run-time states, at which the code variants can safely be switched. To this end, all software objects of the application must be in a safe state so that this state can safely be transferred to the objects corresponding to new code variant. Then, the switching procedure works as follows: At run time of the application, whenever a new configuration state is entered, via the activation mapping the corresponding system variant is determined. As soon the switching point is reached, the old code variant is stopped and the run-time state of all objects is transferred to new objects conforming to the memory layout of the new code variant. Then, the newly selected code variant is (re-)started. This requirement of safe switching points holds for all kind of adaptive systems.

2.2 Programming with Contexts

Some programming paradigms adapt the application if the context of the application changes (context-aware programming). These languages usually provide contexts as first-class programming constructs.

Context-Oriented Programming (COP). COP [13] is a programming technique for dynamic adaptation of runtime behaviour in reaction to changes of (explicitly specified) contexts. Contexts are monitored on changes. A context change can switch software variants, so-called *layers*, specifically targeted to a particular context. Layers are composed at runtime, *stacked*, which makes them somewhat different from dynamic aspects. A *stackable layer* consists of a set of method wrappers for varying the behaviour of a set of methods of a *base layer*, and a *stacked layer* is executed before, after, around or instead of the base layer. Thus, COP provides important concepts for context-based program adaptation. The approach has been implemented on top of languages such as Lisp, Smalltalk, Python, Ruby, and Java; and these COP languages can be used to implement the WOM pattern.

[14] presents COP for wireless sensor networks (WSN) with an approach called ContextGroups. The idea is similar to Genie, but generalises it for any WSN. The contextual states can be grouped into *context groups*, similar to statecharts. Every context state can be associated to a dynamic aspect, and every nested context state to a nested dynamic aspect. Therefore, context groups can also be composed: They can be arranged as communicating automata able to model interacting contexts and their relationships. For example, several context groups can be composed to *parallel sections*, i.e., a set of communicating parallel nested context states. Due to these points, context groups are well suited for the WOM pattern.

Role-Oriented Programming (ROP). Another specific paradigm for context-adaptive systems is Role-Oriented Programming (ROP), a paradigm providing relationships and the roles of the objects taking part in the relationship [15]. A role is a dynamic service of an object relative to a relationship. If a relationship of an object changes, its roles are switched, too. Thus, roles adapt a set of related objects to changes, e.g., to changes of context. However, only few role-oriented languages do provide explicit contexts as first-class concepts [16]. In *ObjectTeams*, the set of roles belonging to one context is called a *team* [17]. In the *Compartment-Role Object Model (CROM)*, roles belong to a *compartment* [16]. Both notions are similar to contexts in COP, but in role-oriented programming, the unit of behaviour is not a method, but a role: A team or compartment groups all context-specific roles of all related objects, so that the behaviour associated to a context can be varied with larger granularity than in COP. The WOM pattern adds explicit world models to ROP, in order to enable the modelling of physical contexts of physical objects, and their causal connection to software contexts and software objects.

3 Programming Problems for Robotic Co-workers

Robotic co-working software must meet special quality requirements. Robots must be safe to use for humans at any time. Because any harmful collisions should be avoided, robotic co-working software should track humans, robots, and their work items, and adapt to their movements and actions. (We call these physical objects *moveables*.) Robotic co-working software should adapt to *context changes*: whenever a human enters the room or the neighborhood of a robotic co-worker, the robot has to slow down or stop (sensitive behaviour). On the contrary, when all humans have left the room, the robotic co-workers should run faster and complete their jobs without delay. It is easy to see that human-robotic co-working has to take place in *smart rooms* that track the actions of all moveables and automatically adapt to events [1]. In other words, human-robotic co-working requires *adaptive smart rooms*.

Knowing these requirements, S. Haddadin suggests in his seminal work on human-robotic co-working [1] that humans have to interact with robots following a protocol described by a statechart with four basic modes (Fig. 1):

- /**Autonomous**/. If no human is present in the room, the robot can run at full speed.
- /**Human-Friendly**/. If a human enters the co-working space, the robot has to work more carefully; at least, it has to slow down its speed.
- /**Collaborative**/. If a human approaches the robot for collaboration, e.g., for an interaction on work items, the robot must be particularly sensitive.
- /**Fault**/. Finally, if something fails, e.g., if power fails, the robot must enter fault mode and stop.

In the statechart, also several structured modes can be defined: *Collaborative* and *Human-Friendly* states form a super-state, the *Human-Aware* state; *Human-Aware* and *Autonomous* form the *Working* state.

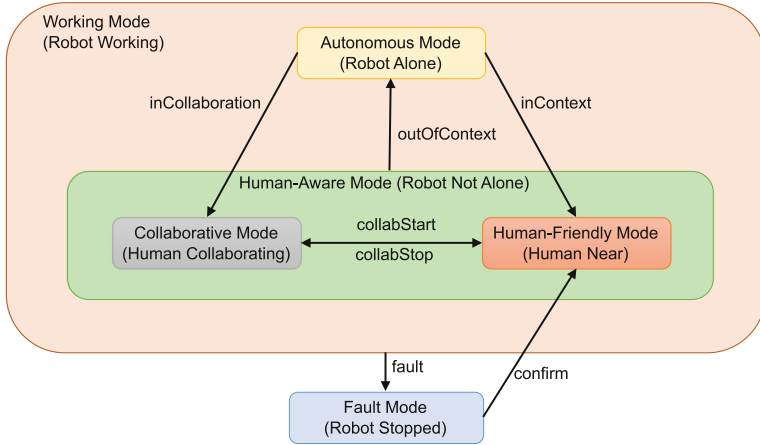


Fig. 1. Functional modes of the DLR co-worker [1] (transitions are simplified for readability). In brackets: context states in the context automaton of the smart room of robotic co-working.

These modes seem to be quite reasonable for all kinds of applications in human-robotic co-working; however, what do they exactly describe? We consider two interpretations. Firstly, because the smart room has to track the actions of all moveables, it can be argued that the Haddadin automaton describes the *states of their relationships* in the physical world, as well as their *transitions* (for this *world-centric interpretation* ignore the names in brackets in Fig. 1). Then, the modes of the automaton are the adaptation modes of the *smart room of robotic co-working* - the world in which humans and robots co-work, and this automaton forms a *world automaton* for robotic co-working applications (Sect. 5.3).

Secondly, many modes of the Haddadin automaton express the human-robot relations by describing the current *context* of the robot, e.g., the human-aware mode specifies that a human is found in the physical context of the robot (for this *context-centric interpretation*, use the names in brackets in Fig. 1). Therefore, a more specific interpretation of the world automaton is that all working mode states, the substates of *Working*, are *context states*, so that the entire automaton can be regarded as a *context automaton*. This interpretation of the automaton seems to be valid, in particular, if the smart room hosts a single robot, of which humans and work items are contexts (Sect. 5.1).

It seems also clear that the Haddadin automaton only specifies the minimal requirements of robotic co-working, in the sense that it must be refined and elaborated for use in specific applications. For instance, the automaton abstracts from the state and the flow of work items. In a realistic co-working application, these should be tracked in a precise manner (see Sect. 6). Also, the teaching of a robot will require a *teaching* mode, which must be entered, when new sub-activities for the autonomous mode should be learned.

3.1 Technical Requirements for Programming with the Haddadin Automaton

If any robotic co-working application has to respect a specific form of Haddadin's automaton for a safe smart room, the question arises, how the automaton can be encoded into the application. The modes of the Haddadin automaton, the states of the smart room, cross-cut the entire application. If the application is programmed in a classic general purpose programming language (GPL), and the automaton is encoded with classic implementation patterns, if-bloating results and verification is hampered.

Context Checks and If-Bloating. First, software development for robotic co-workers is hampered by ubiquitous tests on the state of the world (world-centric interpretation) or the contexts of the robot (context-centric interpretation). Usually, these context checks are intermingled with the application code and produce *if-bloating*, the proliferation of if- and case-statements. This phenomenon is well-known in robotic programming and stems from the fact that the classic implementation pattern for an automaton realises a complex case analyses with if-statements. Then, context checks are implemented as if-cascades, and for an automaton of n states, a complex case analysis of n^2 cases is required [18]. When the contexts are nested, see the *Human-Aware* mode in Fig. 1, a nested case analysis is required with nested if- or case statements. Alas, advanced programming constructs for complex case analysis, such as *decision tables*, are not available in mainstream GPL.

AOSD explains that if-bloating is *scattering* of the context checks over the code [19], because the concerns of world and application are not separated. Because scattering can be solved by aspects, it seems clear that the software architecture of a smart room should employ, in some form, dynamic aspect orientation.

Verification of Safety Features. Second, the safety requirements of a robotic co-worker must be *proven*, so that a vendor of a smart room for co-working can give guarantees on its quality and establish a liability for the software product. However, proving safety requirements about an adaptive smart room for human-robotic co-working turns out to be difficult, due to the following reasons.

1. **Hidden Context Checks.** In a GPL-based implementation, the context checks are hidden, so that a program verifier cannot easily distinguish them from other checks. Because the verifier needs to be cautious what to infer about the context-dependent behaviour of the application, it must make conservative assumptions about the control flow, which deteriorates its analysis results. This is the typical consequence of mingling two concerns, context and application.
2. **Size of Proofs.** Second, proofs of the complete code of a system are most often infeasible. Thus, the reactions of a robot to changes of its context should be proven on an appropriate *model*, which is simpler than the code, but which

captures the essential behaviour. This suggests to employ model-driven software development (MDSO) for robotic co-working software, in which models can be used for verification and later for the generation of the code.

Because of these problems for development and maintenance of smart room applications, the next section introduces an architectural pattern that separates the concerns of context and application.

4 The Architectural Pattern “WOM”

This section introduces the architectural pattern of World-Oriented Modelling (WOM). WOM maintains a model of the world, the smart room, at run-time, i.e., is a M@RT technique [11]. The robotic co-working software system (the application) is split into two interrelated dimensions:¹

1. A *world space* (or *world model*) describing the states of the world, including all states of the relationships of the physical objects, and thus, implicitly, their contexts (Fig. 2, left).
2. A *software variant space* consisting of software variants attributed with different quality features (Fig. 2, right).
3. A total *activation mapping* which maps a world state to a code variant. A transition between states in the world triggers a change of the variants in the software variant space (Fig. 2, middle, mapping expressed by colors).

It is assumed that the world model is simpler than the entire application, that it lends itself to analysis, and that it can be employed to generate an implementation. These assumptions are valid if the world model is an automaton, as the example of the Haddadin automaton suggests. The next assumption is that the mapping between world and software variant space must be *total* so that every state of the world is related to at least one corresponding software variant in the software variant space, and that the quality attributes of these software variants hold in the associated state of the world model. As discussed in Sect. 2, it is important that the programming paradigm of the software variant space enables the safe switching of code variants at switching points. Finally, if the activation mapping is *injective* (i.e., *bijective*), the state in the world solely inherits those features of the related variant in the variant space.

The WOM pattern offers two main advantages for the software engineering of smart rooms for robotic co-working.

Improving Comprehensibility by Avoiding If-Bloating. WOM architectures avoid if-bloating. In a WOM architecture, the tests on contexts and context changes are factored out of the software variant space (the code of the application) into

¹ These dimensions could also be called *aspects*, but that could imply that in the software variant space, dynamic AOP is used as a programming paradigm, which is not intended.

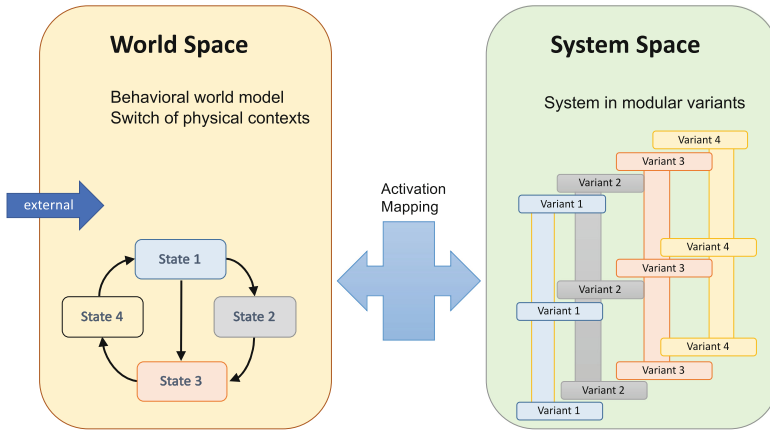


Fig. 2. The WOM approach separates world space and software variant space with variants fitting to the states of the world. The modular variants may overlap, e.g., share subcomponents, roles, or subobjects. (Color figure online)

the world space. Context checks are then executed by the implementation of the world model, while the variants of the application are switched by the run-time system of the software variant space. For the smart room of robotic co-working, this means that the application code is not polluted with context monitoring and checking.

Property 1. The architectural design pattern of World-Oriented Modelling avoids if-bloating in the software variant space, because tests on the world state are factored out of the software variant space into the implementation of the world model. Thus, increasing the number of states in the world model linearly leads to a linear growth of variants in the software variant space, and does not increase the number of context tests in the variants.

Splitting Safety Proofs. With WOM, safety proofs can be split into a reachability proof on the world model (reachability analysis) and a test on a safety feature for the system variants:

Reachability Analysis. For every state of the world, the associated quality features of the related software variants hold. If a sequence of state changes ends up in a state, it is guaranteed that the system variants related to the state are switched on, while all others variants have been switched off. If the activation mapping is injective, this set of qualities holding in a state is uniquely determined by the related variant; if it is non-injective, the qualities all related variants must be joined. Thus, a reachability analysis on the state-based world model can prove which system qualities hold in any state.

Proof Obligations for Qualities. Of course, the WOM approach generates, for every state of the world, a proof obligation that every variant has the attributed quality feature.

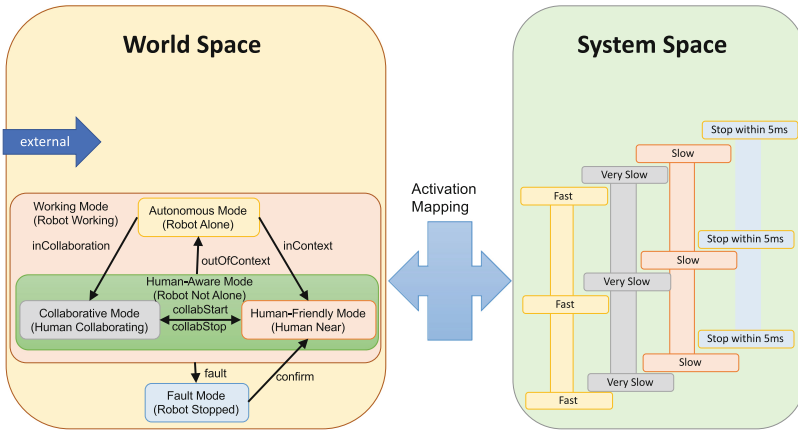


Fig. 3. The WOM approach applied to Haddadin’s automaton of human-robotic co-working. (Color figure online)

Consider the example in Fig. 3 in which the Haddadin automaton is coupled with a software variant space of four variants. The injective activation mapping, expressed by colours, denotes that the autonomous mode inherits the quality feature “Fast” from the yellow variant in the software variant space; the state “Human-friendly mode” inherits the quality feature “Slow”, the state “Collaborative mode” inherits the quality “Very Slow”, and in the fault mode, the system stops within 5 ms. The proof obligations for the variants must be shown by standard verification procedures:

1. The attributes “Fast”, “Slow” are soft requirements and can be shown by benchmarking.
2. For the variant related to the fault mode, it must be proven that the system will stop in 5 ms. The proof depends on the programming paradigm in the software variant space. In a real-time system, a proof can be given, for example, with Worst Case Execution Time Analysis (WCETA) on the code variant, delivering a threshold for the maximum delay to a switching point. In the classic case of a non-real time system, statistical data can be collected with benchmarking, and the mathematical proof is replaced by a quality assurance procedure, e.g., extensive regression testing.

However, with WOM, the proof of the proof obligation will be simpler, because the context checks are factored out from the software variant space to the world model.

Property 2. The architectural design pattern of World-Oriented Modelling allows for splitting the verification of quality features of an application into a reachability analysis on a state-based world model and a quality assurance procedure of the software variant space.

5 Flexibility of WOM

The WOM pattern does not fix a specific modelling language for the world space, nor a specific programming language for the software variant space. Many different modelling languages can be chosen for the world space, as well as many programming languages for the software variant space. To illustrate this flexibility, this section presents several examples from programming and modelling languages. First, we discuss the simpler scenario when humans interact with *one* robotic co-worker, and the all states of the smart room form *contexts* of the robot. In Subsect. 5.3, we discuss extensions to the more complex scenario when several robotic co-workers co-habitate a smart room and show that modelling approaches offer advantages for the world space.

5.1 The States of the World as Contexts of a COP Application

As mentioned above, the *context-centric* interpretation of the world space consists of interpreting the set of states as a set of contexts of *one* robot (*context automaton*). For this context-centric interpretation, languages are appropriate which provide the explicit modelling of contexts.

WOM and Context-Oriented Programming (COP). In Context-Oriented Programming (COP), contexts are statically defined (see Sect. 2), i.e., in an application, a fixed number of contexts exist. Thus, a COP program may represent the world model as a *finite* context automaton. The following listing shows a JCOP pseudocode for the world space of a robotic co-working smart room derived from Fig. 3. The defined software contexts of the robotic co-worker, *RobotAlone*, *HumanNear*, *HumanCollaborating* and *Fault*, correspond to the world states of Haddadin’s automaton, i.e., the physical contexts of the robot (Fig. 1, names in brackets). Their transitions are implicitly specified by checking the validity of entry predicates of the contexts.²

```

1  /* World space */
2  context RobotAlone { when(!inContext(human)) {
3      with(Fast);
4      without(Defensive, Slow, Stop);
5  }
6  context HumanNear { when(inContext(human)) {
7      with(Defensive);
8      without(Fast, Slow, Stop);
9  }
10 context HumanCollaborating {
11     when(inCollaboration(human)) {
12     with(Slow);

```

² In the following, for simplicity, the listing only specifies the atomic states and their transitions; transitions to complex states can easily be added.

```

13     without(Fast , Defensive , Stop); }
14 }
15 context Fault { when(faultOccured()) {
16     with(Stop);
17     without(Fast , Defensive , Slow); }
18 }
19 /* Software variant space */
20 layer Fast { /* fast behaviour */ }
21 layer Slow { /* slow behaviour */ }
22 layer Defensive { /* sensitive behaviour */ }
23 layer Stop { /* immediate halting */ }

```

The JCOP pseudocode defines the reactions of the robot with respect to the actions of the human being in its physical context. The *when* conditions define the conditions for activating a software context. Without a human in the work area (*when (!inContext())*), the robot is in *RobotAlone* context. When a human appears, the robot switches to *HumanNear* or *HumanCollaborating* context, depending on whether the human being is in the near neighbourhood of the robot (*when (inContext())*), or collaborating (*when (inCollaboration())*). When a fault occurs, e.g., power fails (*faultOccured()*), the context *Fault* is activated that should immediately halt the robot.

All contexts, once they are enabled, activate layers by *with* and deactivate other layers by *without* specifications (activation mapping). In our example, the four layers *Fast*, *Slow*, *Defensive*, and *Stop* form the software variant space of the WOM pattern, because they define the speed of the robot as a function of the active context. From the analysis of the *with* and *without* statements, it can be seen that only one layer is active for each context. Thus, the mapping between the world model and the software variant space is bijective.

Therefore, COP is a possible technique to implement the WOM pattern, if a fixed number of contexts is appropriate to describe the world. However, there are the following constraints that should be respected:

- Context model and layers should be separately specified. Because contexts and layers can be mixed in COP languages, this should be enforced by appropriate programming rules.
- Overall, the activation mapping must be *total*, i.e., in every context of the context automaton of the world space, at least one layer must be set active. If the activation mapping is not injective, i.e., activates several layers, these will be composed.

Thus, COP-based languages are useful to realise applications in robotic co-working, representing their context automata as contexts activating and deactivating layers.

WOM with Dynamic Aspect-Oriented Programming in ECaesarJ. The language ECaesarJ does not have the ability to specify contexts directly, but

provides objects realising finite state machines (*state-machine objects*) by which context automata can be represented. Additionally, the language allows for the specification of *component classes*, cross-cutting code slices used in a similar way as COP layers. Thus, the world of a robotic smart room can easily be specified as a state-machine object *SmartRoom*:

```

1  class SmartRoom { // World space
2  // Event definitions
3  event void inContext(); event void inCollaboration();
4  event void fault(); event void outOfContext();
5  event void confirm(); event void collabStop();
6  event void collabStart();
7  // Context automaton: context states and transitions
8  state initial RobotAlone =
9  (inContext() => Slow.Behaviour() -> HumanNear
10 | inCollaboration() =>
11   Defensive.Behaviour() -> HumanCollaboration);
12 state HumanNear = (collabStart() =>
13   Defensive.Behaviour() -> HumanCollaboration
14 | outOfContext() => Fast.Behaviour() -> RobotAlone);
15 state HumanCollaboration = (collabStop() =>
16   Slow.Behaviour() -> HumanCollaboration
17 | outOfContext() => Fast.Behaviour() -> RobotAlone);
18 state Fault = (confirm() =>
19   Slow.Behaviour() -> HumanNear);
20 // Software variant space: Layers as cclasses
21 cclass Slow { Behaviour() {..} }
22 cclass Defensive { Behaviour() {..} }
23 cclass Fast { Behaviour() {..} }
24 cclass Stop { Behaviour() {..} }
25 }
```

In ECAesarJ, state transitions between context states must be specified explicitly. For instance, Line 9 specifies that in state *RobotAlone*, when the event *inContext()* occurs, the state should change to *HumanNear*, and the *Slow* behaviour component should be switched on. For one state, several transitions can be specified, for instance, in state *RobotAlone*, alternatively, the event *inCollaboration()* triggers the activation of the *Defensive* behaviour.

With ECAesarJ's state-machine objects, open, dynamic world models can be specified, because the number of state-machine objects, their interactions and their relationship, as well as the number of layers and variants is not bound. With ECAesarJ, it is possible to realise context automata as communicating state-machine objects, as well as to build up the world space as a set of communicating context automata. Thus, ECAesarJ should also be a good language to implement robotic co-working applications.

5.2 World Spaces with Context-Oriented Modelling

In the last years, also context-oriented modelling techniques have appeared particularly suited for the specification of contexts, context switches, and context automata. Because these approaches can be employed for reachability analysis and verification of the world model, they are ideally suited to model world spaces in WOM. Essentially, these approaches couple a modelling approach for the world space with a programming approach for the software variant space, leading to an model-driven software development (MDS) approach.

ContextGroups for WSN. ContextGroups provide a model-driven approach [14]: Context statecharts can be translated to a textual notation, the language CONESC, in which the nested states of a context statechart can be associated with layers. Thus, the WOM pattern can be realised by specifying a context statechart for the world space, while coupling it to a software variant family in CONESC by a total activation mapping. This implies that, in contrast to COP, the world model is, by construction, separated from the software variant space.

Instead of automata or statecharts, also *Context Petri Nets* [20] can be employed, in which a control Petri net describes the available contexts and their transitions. Due to the nature of Petri nets, this approach can model parallel transitions of multiple contexts in the world. Like ContextGroups, Context Petri Nets can be used to describe world spaces and coupled to a variant family in the software variant space. When a context is activated, a basic Petri net describing the software variants is adapted.

Compartment-Role-Oriented Modelling (CROM) and SCROLL. Section 2.2 showed that role-oriented programming is also a valid approach for the context-adaptive software of embedded and cyber-physical systems, and hence, also for robotic co-working. In the following, we discuss an example in the language Compartment-Role Object Modelling (CROM) language with both *compartments* (*explicit contexts*) and *roles* as first-class language concepts [16]. As in ContextGroups, CROM can be coupled in an MDS process with a role-based programming language SCROLL, an extension of Scala [21]. The model-driven approach enables to couple nested contexts (in the world space) and role-oriented programming (in the software variant space).

```

1 // Nesting of contexts
2 compartment SmartRoom {
3   compartment Working {
4     compartment Autonomous {
5       role model: role Fast { .. } ..
6     };
7   compartment HumanAware {
8     compartment HumanFriendly {
9       role model: role Defensive { .. } ..

```

```

10     };
11     compartment Collaborative {
12         role model: role Slow { .. } ..
13     };
14 };
15 };
16 compartment Fault {
17     role model: role Stop { .. }
18 };
19 role group AdaptiveBehaviour
20     selecting 1..1 {Fast; Defensive; Slow; Stop}
21 }

```

When realising the WOM pattern in CROM, contexts in the world space are represented by compartments, and the variants of the software variant space are represented by roles, role groups, and role models (similar to layers) capturing the interaction of roles. Similar to ContextGroups, CROM provides aggregation of compartments, so that the nesting of the states of the Haddadin automaton can easily be specified. Like in Context Petri Nets, several compartments can be active at the same time. Because in CROM all roles belong to a specific compartment, the activation mapping between compartments and roles is total and injective (see the scope nesting in the Listing). Due to the strong connection of roles to compartments, it is not possible to specify *inactive* role models. Thus, in CROM, it is possible to see the transitions in the world model as transitions between compartments, switching on and off of compartments, and the switching of variants as the switching of the role models associated to the compartments.³ On the other hand, CROM supports wellformedness constraints over compartments and roles. Line 19, for example, specifies that only one role model (one layer) can be active at a time (injective activation mapping). Thus, in CROM, the activation mapping is simply expressed by the containment relationship of roles in role models and contexts, as well as further constraints over role-groups.

In CROM+SCROLL, compartments are objects, i.e., the number of compartments is not fixed. Thus, infinite world automata can be constructed, as in ECAesarJ. Because CROM+SCROLL offer a model-driven approach, world model and software variant space are cleanly separated, offering a good opportunity for reuse in both worlds.

5.3 WOM in General Smart Rooms

If a smart room for robotic co-working contains a single robot, the above context-oriented programming and modelling techniques can be used to realise the world

³ At the moment, however, CROM does not support the specification of compartment automata directly. However, from CROM specifications, code in the language SCROLL can be generated. This code can be extended with transition rules for switching compartments.

space of the WOM pattern. However, if several robots need to be coordinated, the states of the world should be interpreted as global configurations of the *smart room* capturing *all* contexts of *all* robots simultaneously. Therefore, the context automata of the singular robots should be *composed* to get a full world automaton (world-centric interpretation). This suggests that for WOM in general smart rooms, a MDS approach is more appropriate, in which world spaces are composable models. Additionally, in this interpretation, the interactions of the robots must be taken into account, for instance, when they collaborate, new synchronisations should be added to their context automata. This requires that other specification languages, e.g., Context Petri Nets, come into play that can represent the natural parallelism of the robots and their synchronisations more appropriately than finite automata. Thus, for a full scenario of robotic co-working, the language in which the world model of the smart room is specified should support flexible composition and synchronisation of state machines. This leads to the following observation:

Property 3. For robotic co-working applications with several robots, compositional modelling techniques for the world space offer advantages because the world model must be composed of the context automata of the singular robots.

Constructing smart rooms with WOM relies on three forms of composition: composition of context automata, composition of variants in the software variant space, and composition of the activation mappings. Thus for co-working with *several* robots in a smart room, MDS techniques with formal methods supporting compositionality seem to be adequate. The MDS techniques discussed above support composition of world models in different ways.

Programming Paradigms: differ in their support for compositionality:

1. While COP supports the composition of layers, it does not support the composition of contexts and context automata, although this does not seem to be impossible to integrate into a COP language.
2. ECaesarJ state-machine classes and objects can be composed by class composition and object composition operators, of which the language provides a rich set (mixin composition, virtual class composition, wrapper composition, and more). Thus, more complex world models can easily be programmed in ECaesarJ, and still, world models and software variant spaces can be kept disjoint and simple.

Modelling Paradigms: usually provide composition systems:

1. ContextGroups provides nesting of contexts for context composition, synchronisation and communication. Therefore, context statecharts support flexible composition of contexts and context automata.
2. Context Petri Nets provide the classic composition operators for petri nets, i.e., sequential or parallel composition, as well as synchronised composition. Petri nets are made for parallel modelling, also for modelling of parallel robots. The insertion of a synchronisation protocol or other forms of complex interaction between contexts of multiple robots can easily be specified.

3. CROM+SCROLL provides inheritance and nesting as composition operators on compartments. However, composition of context automata is not yet supported.

6 Case Study WEIR

A prototype case study employing WOM for safe co-working robotics is the WEIR system demonstrated in the KUKA innovation award at the Hanover fair 2016⁴. WEIR has been designed for the remote teaching of industrial and service robots by recording the movements of humans wearing of smart clothes and wearables. The WEIR system includes a smart jacket and a glove equipped with inertial motion units (IMUs) and bendable resistors. Those sensors measure the orientation of different body parts such as torso, upper and lower arm, the hand, and the direction of one or more fingers. The gathered data is compiled at run-time to a world model, which specifies the physical conditions and postures of a human worker. The resulting model is mapped to the inverse kinematics of a KUKA LBR iiwa robot arm, such that the latter can be controlled the human movements. Recording these movements, the robot can quickly be taught, with the purpose of prototyping complex automation tasks without the need of programming skills. Due to the capability to measure human postures, the WEIR system is also an example for building safe robotic co-working applications in an industrial set up.

6.1 Sensor- and Demonstration-Based Teaching

The initial idea of WEIR was to monitor human body postures by sensors and to map them to the kinematics of an industrial robot arm. Until recently, this problem was solved by techniques based on direct guidance of the actuator (*direct teaching*) or on *optical tracking* (*demonstration-based teaching*). The former principle is based on the sensitivity feature that modern lightweight robots provide. Sensitivity is understood as the capability to measure torque within an axis' motor and to react to a change in real time. For instance, the KUKA LBR iiwa comes with seven axes, which all contain quite precise torque sensors. The robot can be configured with safety settings specifying a torque threshold that leads to a safety stop, by activating the physical brakes within the axes. Besides this functionality, the torque sensors can be used for direct teaching: A human may touch the robot and move it to a new position. The navigated trajectories or positions can be recorded by reading the torque sensor values and, afterwards, be replayed. This direct teaching by dragging is the standard way of teaching the KUKA LBR iiwa.

Alternatively to direct teaching, optical tracking of human movements can be used for the recording of activities (*demonstration-based teaching*) [22, 23].

⁴ <http://st.inf.tu-dresden.de/weir>. A video can be found on YouTube: <https://www.youtube.com/watch?v=i4Dmzm1CHwM>.

This requires that the 3-D posture of a human is tracked by a set of stereo cameras, measured, and mapped to the robot's kinematics. However, optical tracking suffers from high cost for camera equipment and complex procedures for calibration and precise setup. Also, since humans should not be present in hazardous, dirty, poisonous, or heavy-weight environments, tracking-based teaching cannot be used for many operations in nuclear, coal, or steel industries.



Fig. 4. Hardware components of WEIR.

To overcome these problems, WEIR provides a new variant of demonstration-based teaching, *sensor-based teaching* with sensor-equipped wearables (Fig. 4). With the WEIR jacket and glove, human trainers need not be positioned in the physical neighbourhood of the robot, but can reside in a safe remote environment being connected to the actual robot via wireless network for remote control. The WEIR jacket and glove, equipped with several inertial measurement units (IMU) connected to each other via bluetooth, assemble and aggregate their data to measure the spatial position of the arm and the hand. The glove is equipped with bending stripes (see digit of left glove in Fig. 5) so that it can be monitored whether a hand is open or closed. The sensor data is aggregated with different micro-controllers in the jacket (see its pocket, Fig. 4) and then forwarded to a server, which also assembles the world model, computes the inverse kinematics, and communicates with the robot actuator.

Besides remote teaching, the WEIR method has several advantages. First, the sensor-based jacket costs less than 500€, and these costs are going to shrink a lot in the next years, because sensors and micro-controllers become rapidly cheaper. Second, setting up and calibrating the system works quite fast: WEIR expects the demonstrator to take a predefined posture and press a button, after which the system is calibrated and ready to operate. In consequence, the WEIR

system is much more flexible to employ in many situations than the optical approaches, for example, during a shift of workers in a manufacturing workshop. And finally, because the worker has to put on the wearable cloth explicitly, she knows precisely when she is tracked and when not. On the one hand, this gives a lot of self-assurance when collaborating with the robotic co-worker. On the other hand, if the actions of “entering the room via a door” and “putting on the wearable jacket” are combined as a complex event for starting the collaboration with the robot, it can easily be detected whether humans *not* wearing a jacket are in the room - with should lead to fault stop. Of course, it is always possible to *combine* the sensor-based jacket with image-based tracking, e.g., to provide better fault-tolerance.

6.2 Safe Robot Co-working in WEIR Smart Spaces

The WEIR case study organises the underlying smart room with the WOM architectural pattern. The world model is based on a set of automata derived from Haddadin’s suggestion, but extended with 3-D collaboration zones. The software variant space of the robot is programmed by a state-based workflow with variants.

In the context of robotic co-working, the 3-D models of all involved moveables form important ingredients of the world model, because they have to be set in spatial relation to each other to derive the state of their physical relationship. As a minimum, the 3-D body model of the human should be set into relation with the 3-D body model of the robot. Also important spots in the smart room, e.g., boxes or feeder belts, should be modelled as 3-D objects, because they provide the zones where robot and humans meet. For this purpose, WEIR employs defines *spatial zones of collaboration*, 3-D primitives such as spheres or cubes in the 3-D coordinate system. For the robot, a 3-D collaboration zone is defined, based on information of its physical posture, which is normalised in the room’s coordinate system. For the human and other moveables, their collaboration zone is computed with the help of in-door localisation within the same coordinate system. All collaboration zones together form the *smart room* in which robotic co-working can take place.

As a case study, we used WEIR’s co-working feature to realise a *pick-and-place process* with workflow adaptation (Fig. 5). This process can be employed as a quality-assurance process in production in which the robot automatically separates defect from intact parts. As an example for the 3-D smart room, compare the left side of Fig. 5 with its right side. In front of the KUKA LBR robot arm, four boxes are being placed, of which the two outer ones are *sources*, whereas the two inner ones are *sinks* of the pick-and-place task. In an alternating order, the robot picks objects from the sources and randomly places them in one of the sinks. On the right side, the smart room with its collaboration zones and moveables is shown. The four boxes form four *collaboration zones* (SRCA, SRCB, GOOD, BAD), which represent the virtual position of each physical box. They are included in a fifth one, called AWARENESS, which allows for detection of human presence. Additionally, the 3-D positions of the human arm are depicted:

Green dots represent moveables, such as the single body parts (ARM1, GLOVE, HAND, CENTER). For the gripper, two positions are tracked: One position indicates where robot is navigating to (GRIPPERTARGET), while the other tracks where it is currently located (GRIPPERREAL). From the 3-D model in Fig. 5, it can be inferred that the human arm does not grip into one of the boxes, i.e., has not entered a collaboration zone.



Fig. 5. WEIR collaboration zones for pick-and-place case study. (Color figure online)

To adapt the robot's behaviour, for each collaboration zone of the world model a Haddadin automaton is specified. Transitions of these state machines are triggered by either entry or exit events of a tracked moveable. For instance, the designer may define an state machine for a collaboration zone, which switches to a new state, if a human hand enters the physical space, and, subsequently, forces the robot actuator to react on this state change. During the execution of a transition of a collaboration zone's state machine, the robotic control system can be adapted with two types of adaptations:

Parametric Adaptation. A parameter [24] such as speed or maximum torque to control sensitivity may be set. In this case, the workflow stays the same.

Workflow Adaptation. Another possibility is to change the workflow of the robot to another variant, based on state conditions of the collaboration zone automata. The robot's workflow is also described by a state machine, whose states refer to a certain movement or posture. The state machines of the robot and the collaboration zones are linked. The transitions in the robot's workflow can be constrained by state conditions of a collaboration zone automaton. This means that the current state of the specific collaboration zone can be used as a condition in the workflow of the robot, for instance, to run another variant of the workflow. Thus, this approach allows for creating complex behaviours that vary according to the monitored physical relations of moveables.

For these adaptations, there are four requirements (left side of Fig. 5):

- /R0/. As soon as a human enters the robot's workspace, represented by the global collaboration zone AWARENESS, its automaton switches to a state, in which the robot will work with only 30% of its full speed. In this case, a parametric adaptation is applied (setting speed to 30%).
- /R1/. As soon as a human and robot work in the same box, the robot is expected to stop operation completely. Here, a parametric adaptation is applied (setting speed to 0).
- /R2/. When the human removes his hand, the operation is continues (indicated by the lighted ring around the actuator).
- /R3/. If a human operates in a source box, the robot is expected use the source box on the other side in order to avoid collisions (workflow adaptation).

The WEIR pick-and-place system illustrates how complex robotic co-working processes can be realised following the WOM pattern. The WEIR smart room has a WOM world model, which is composed from several collaboration zones whose behaviours follow the Haddadin automaton. The context checks are performed in the world model, not in the application code. The adaptation of the software variant space is controlled by the transitions of the world model's state space. The adaptation does not only change parameters, but adapts the robot's workflow. This makes the robot react to the presence of the human worker and enables interaction and collaboration.

7 Conclusion

We have argued in this paper that human-robotic co-working always has to take place in *adaptive smart rooms* in which the movements of humans have to be tracked intelligently so that the robotic system can automatically adapt to them. A particular contribution of the paper is an architectural design pattern, World-Oriented Modelling (WOM), which is useful for structuring applications in robotic co-working, because it separates the concerns of *world* and *application*. It models the states of the physical objects in the smart room in a separate world model and leaves the context checks to its run-time system. Thereby, it avoids if-bloating in the software variant space of the adaptive application. Additionally, WOM simplifies the required safety verification procedures to a reachability analysis on the world model and quality assurance methods for the software variant space. WOM generalises dynamic SPL and featured transition systems, combining their principle of separation of concerns with context-aware programming.

References

1. Haddadin, S., et al.: Towards the robotic co-worker. In: Pradalier, C., Siegart, R., Hirzinger, G. (eds.) *Robotics Research*. Springer Tracts in Advanced Robotics, vol. 70, pp. 261–282. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-19457-3_16
2. Pransky, J.: The Pransky interview: Dr Esben Ostergaard, inventor, co-founder and CTO of Universal Robots. *Ind. Robot* **42**, 93–97 (2015)
3. Kirschner, D., Velik, R., Yahyanejad, S., Brandstötter, M., Hofbaur, M.: YuMi, come and play with me! A collaborative robot for piecing together a tangram puzzle. In: Ronzhin, A., Rigoll, G., Meshcheryakov, R. (eds.) *ICR 2016*. LNCS (LNAI), vol. 9812, pp. 243–251. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-43955-6_29
4. Ju, Z., Yang, C., Li, Z., Cheng, L., Ma, H.: Teleoperation of humanoid Baxter robot using haptic feedback. In: *International Conference on Multisensor Fusion and Information Integration for Intelligent Systems (MFI)*, pp. 1–6. IEEE (2014)
5. Many: Discussion on the web platform reddit (2015)
6. Nunez, A., Gasiunas, V.: *ECAesarJ User’s Guide*. Technische Universität Darmstadt, Germany (2009)
7. Capilla, R., Bosch, J., Trinidad, P., Cortés, A.R., Hinchey, M.: An overview of dynamic software product line architectures and techniques: observations from research and industry. *J. Syst. Softw.* **91**, 3–23 (2014)
8. Classen, A., Cordy, M., Schobbens, P.Y., Heymans, P., Legay, A., Raskin, J.F.: Featured transition systems: foundations for verifying variability-intensive systems and their application to LTL model checking. *IEEE Trans. Softw. Eng.* **39**, 1069–1089 (2013)
9. Raskin, J.F.: An introduction to hybrid automata. In: Hristu-Varsakelis, D., Levine, W.S. (eds.) *Handbook of Networked and Embedded Control Systems*, pp. 491–518. Birkhäuser (2005)
10. Kramer, J., Magee, J.: Towards robust self-managed systems. *Prog. Inf.* **5**, 1–4 (2008)
11. Bencomo, N., France, R.B., Cheng, B.H.C., Aßmann, U. (eds.): *Models@run.time - Foundations, Applications, and Roadmaps*. LNCS, vol. 8378. Springer, Cham (2014). <https://doi.org/10.1007/978-3-319-08915-7>
12. Bencomo, N., Grace, P., Flores-Cortés, C.A., Hughes, D., Blair, G.S.: Genie: supporting the model driven development of reflective, component-based adaptive systems. In: Schäfer, W., Dwyer, M.B., Gruhn, V. (eds.) *30th International Conference on Software Engineering (ICSE 2008)*, Leipzig, Germany, 10–18 May 2008, pp. 811–814. ACM (2008)
13. Appeltauer, M., Hirschfeld, R., Lincke, J.: Declarative layer composition with the JCop programming language. *J. Object Technol.* **12**(4), 1–37 (2013)
14. Afanasyov, M., Mottola, L., Ghezzi, C.: Context-oriented programming for adaptive wireless sensor network software. In: *International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pp. 233–240. IEEE Computer Society (2014)
15. Steimann, F.: On the representation of roles in object-oriented and conceptual modelling. *Data Knowl. Eng.* **35**, 83–106 (2000)
16. Kühn, T., Leuthäuser, M., Götz, S., Seidl, C., Aßmann, U.: A metamodel family for role-based modeling and programming languages. In: Combemale, B., Pearce, D.J., Barais, O., Vinju, J.J. (eds.) *SLE 2014*. LNCS, vol. 8706, pp. 141–160. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11245-9_8

17. Herrmann, S.: A precise model for contextual roles: the programming language ObjectTeams/Java. *Appl. Ontol.* **2**, 181–207 (2007)
18. Moret, B.M.E.: Decision trees and diagrams. *ACM Comput. Surv.* **14**, 593–623 (1982)
19. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-oriented programming. In: Akşit, M., Matsuoka, S. (eds.) *ECOOP 1997*. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997). <https://doi.org/10.1007/BFb0053381>
20. Cardozo, N., González, S., Mens, K., Straeten, R.V.D., D'Hondt, T.: Modeling and analyzing self-adaptive systems with context Petri nets. In: *TASE*, pp. 191–198. IEEE Computer Society (2013)
21. Leuthäuser, M.: Pure embedding of evolving objects. In: *Ninth International Conference on Adaptive and Self-Adaptive Systems and Applications (ADAPTIVE)*, IARIA (2017)
22. Maycock, J., Steffen, J., Haschke, R., Ritter, H.: Robust tracking of human hand postures for robot teaching. In: *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 2947–2952. IEEE (2011)
23. Ude, A., Atkeson, C.G., Riley, M.: Programming full-body movements for humanoid robots by observation. *Rob. Auton. Syst.* **47**, 93–108 (2004)
24. Andersson, J., de Lemos, R., Malek, S., Weyns, D.: Modeling dimensions of self-adaptive software systems. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) *Software Engineering for Self-Adaptive Systems*. LNCS, vol. 5525, pp. 27–47. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02161-9_2

Author Index

- Abdelrazek, Mohamed 272
Altalhi, Abdulrahman 401
Amrani, Moussa 426
Apvrille, Ludovic 47
Aßmann, Uwe 484
Avazpour, Iman 272
- Barat, Souvik 94
Barn, Balbir 94
Bauer, Bernhard 247
Brogi, Antonio 1
- Clark, Tony 94
- da Silva, Alberto Rodrigues 23
Di Tommaso, Antonio 1
- Englebert, Vincent 426
Estivill-Castro, Vladimir 119
Etzlstorfer, Jürgen 354
- Faitelson, David 377
Falkenberg, Jan 484
Feki, Jamel 401
Fortuin, Sven 222
- Geismann, Johannes 72
Genius, Daniela 47
Gérard, Sébastien 300
Gilson, Fabian 426
Greenyer, Joel 449
Gritzner, Daniel 449
Grundy, John 272
- Harel, David 449
Heinrich, Robert 377
Hexel, René 119
Hitz, Michael 328
Holtmann, Jörg 173
Höttger, Robert 72
- Jansen, Slinger 222
- Kapsammer, Elisabeth 354
Katz, Guy 449
Kessel, Thomas 328
Koch, Thorsten 173
Krawczyk, Lukas 72
Kristensen, Lars Michael 198
Kulkarni, Vinay 94
- Lamo, Yngve 198
Li, Letitia W. 47
Li, Shuai 300
Lindemann, Timo 173
Liu, Jian 272
- Marron, Assaf 449
- Overeem, Michiel 222
- Pfisterer, Dennis 328
Pham, Van Cam 300
Piechnick, Christian 484
Piechnick, Maria 484
Pohlmann, Uwe 72
Pröll, Reinhard 247
Püschel, Georg 484
- Rabbi, Fazle 198
Radermacher, Ansgar 300
Reis, André 23
Rumpold, Adrian 247
- Schmelter, David 72
Schönböck, Johannes 354
Schwägerl, Felix 145
Schwinger, Wieland 354
Soldani, Jacopo 1
Steinberg, Shlomi 449
- Taktak, Said 401
Tyszberowicz, Shmuel 377
- Werner, Sebastian 484
Westfechtel, Bernhard 145
- Zurfluh, Gilles 401