Ferhat Khendek
Maria Toeroe
Abdelouahed Gherbi
Rick Reed (Eds.)

LNCS 7916

SDL 2013: Model-Driven Dependability Engineering

16th International SDL Forum
Montreal, Canada, June 2013
Proceedings

Springer

Lecture Notes in Computer Science 7916

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

Editorial Board

Ferhat Khendek Maria Toeroe
Abdelouahed Gherbi Rick Reed (Eds.)

SDL 2013: Model-Driven Dependability Engineering

16th International SDL Forum
Montreal, Canada, June 26-28, 2013
Proceedings

Springer

Volume Editors

Ferhat Khendek
Concordia University, Montreal,QC, Canada
E-mail: ferhat.khendek@concordia.ca

Maria Toeroe
Ericsson Inc., Montreal, QC, Canada
E-mail: maria.toeroe@ericsson.com

Abdelouahed Gherbi
École de Technologie Supérieure, Montreal, QC, Canada
E-mail: abdelouahed.gherbi@etsmtl.ca

Rick Reed
Telecommunications Software Engineering, Windermere, UK
E-mail: rickreed@tseng.co.uk

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-38910-8 e-ISBN 978-3-642-38911-5
DOI 10.1007/978-3-642-38911-5
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2013939785

CR Subject Classification (1998): D.2, C.2, D.3, F.3, C.3, K.6, D.2.4

LNCS Sublibrary: SL 5 – Computer Communication Networks and Telecommunications

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

Printed on acid-free paper

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

Preface

The System Design Languages Forum (SDL Forum), held every two years, is an international conference that provides an open arena for participants from academia and industry to present and discuss recent innovations, trends, experiences, and concerns in the field of system design languages and modeling technologies. Originally focusing on the Specification and Description Language — standardized and further developed by the International Telecommunications Union (ITU) over a period of more than 30 years — the SDL Forum has broadened its topics in the course of time.

The SDL Forum conference series is run by the SDL Forum Society, a nonprofit organization founded in 1995 by language users and tool providers to promote the ITU Specification and Description Language and related system design languages, including, for instance, Message Sequence Charts (MSC), ASN.1, TTCN, URN, UML, and SysML.

The local coorganizers of the 16$^{\text{th}}$ edition of the SDL Forum (SDL 2013) were Concordia University, Ericsson, and École de Technologie Supérieure. A special focus of SDL 2013 was on model-driven dependability engineering, which aims at developing dependable systems following the model-driven paradigm. The reason for setting this focus is that we have come to depend heavily on software systems in virtually every sector of human activity, including telecommunications, aerospace, automotive, process automation, and this trend is further increasing. These software systems are increasingly complex because of ever-growing demands for functionalities, features and improved user experience. The specification, design, validation, configuration, deployment and maintenance of such systems are accordingly complex tasks, to which the dependability requirements add yet another dimension. The dependability of software systems, which is a multi-attribute quality that includes reliability, availability and security, needs to be taken into account in the development process so that they meet the target requirements.

This volume contains the papers presented at SDL 2013. Sixteen high-quality papers were selected from 30 submissions. Each paper was peer reviewed by at least three Program Committee members and discussed during the online Program Committee meeting. The selected papers cover a wide spectrum of topics related to system design languages, ranging from the System Design Language usage and evolution to model transformations, and were grouped into six technical sessions as reflected in this volume. The first session is devoted

to verification and testing using ITU-T languages. The papers in the second session tackle different issues related to dependability engineering, while papers in the third session propose model driven analysis approaches for safety properties or dependability. Domain-Specific Languages are proposed in the fourth session papers, followed by a set of papers on model transformation, before concluding with contributions on the ITU-T Specification and Description Language and its evolution.

The SDL Forum has been made possible by the dedicated work and contributions of many people and organizations. We thank the authors of submitted papers, the 48 members of the Program Committee, and the members of the SDL Forum Society Board. We thank the Communications Services, Conference Services and Instructional and Information Technology Services of Concordia University for their support. The submission and review process was run with easychair.org, we therefore thank the people behind the EasyChair conference system. We thank the sponsors of the SDL 2013: Concordia University, Ericsson, École Technologie Supérieure, SYTACOM, and the University of Kaiserslautern.

April 2013

Ferhat Khendek
Maria Toeroe
Abdelouahed Gherbi
Rick Reed

Organization

We dedicate these proceedings to Rick Reed, to acknowledge his long-standing and continued contributions to the Specification and Description Language and related ITU languages, the SDL Forum Society, and the SDL Forum and SAM Workshop series. Thank you Rick!

Ferhat Khendek
Maria Toeroe
Abdelouahed Gherbi
Reinhard Gotzhein (Chairman of SDL Forum Society)

SDL Forum Society

The SDL Forum Society is a not-for-profit organization that, in addition to running the System Design Languages Forum (SDL Forum) conference series of events (once every two years), also:

- Runs the System Analysis and Modelling (SAM) workshop series, every two years between SDL Forum years
- Is a body recognized by ITU-T as co-developing System Design Languages in the Z.100 series (Specification and Description Language), Z.120 series (Message Sequence Chart), Z.150 series (User Requirements Notation) and other language standards
- Promotes the ITU-T System Design Languages

For more information on the SDL Forum Society, see http://www.sdl-forum.org.

Organizing Committee

Chairs

Abdelouahed Gherbi	École de Technologie Supérieure, Canada
Ferhat Khendek	Concordia University, Canada and Secretary SDL Forum Society
Maria Toeroe	Ericsson, Canada

Members

Reinhard Gotzhein	Chairman SDL Forum Society
Martin von Löwis	Treasurer SDL Forum Society
Rick Reed	Non-voting member of SDL Forum Society Board

Program Committee

Conference Chairs

Abdelouahed Gherbi	École de Technologie Supérieure, Canada
Ferhat Khendek	Concordia University, Canada & Secretary SDL Forum Society
Maria Toeroe	Ericsson, Canada
Reinhard Gotzhein	Chairman SDL Forum Society

Members

Daniel Amyot	University of Ottawa, Canada
Rolv Bræk	Norges Teknisk-Naturvitenskapelige Universitet, Norway
Reinhard Brocks	HTW Saarland, Germany
Jean-Michel Bruel	Université de Toulouse, France
Laurent Doldi	Aeroconseil, France
Joachim Fischer	Humboldt-Universität zu Berlin, Germany
Pau Fonseca i Casas	Universitat Politècnica de Catalunya, Spain
John Fryer	SAForum and OpenSAF, USA
Emmanuel Gaudin	PragmaDev, France
Abdelouahed Gherbi	École de Technologie Supérieure, Canada
Reinhard Gotzhein	Technische Universität Kaiserslautern, Germany
Jens Grabowski	Georg-August-Universität Göttingen, Germany
Peter Graubmann	Siemens, Germany
Øystein Haugen	SINTEF, Norway
Loïc Hélouët	INRIA Rennes, France
Peter Herrmann	Norges Teknisk-Naturvitenskapelige Universitet, Norway
Dieter Hogrefe	Georg-August-Universität Göttingen, Germany
Ferhat Khendek	Concordia University, Canada
Tae-Hyong Kim	Kumoh National Institute of Technology, Korea
Finn Kristoffensen	Cinderella, Denmark
Yves Le Traon	Luxembourg University, Luxembourg
Anna Medve	Pannon Egyetem, Hungary
Pedro Merino Gómez	Universidad de Málaga, Spain
Birger Møller-Pedersen	Universitetet i Oslo, Norway
Os Monkewich	Sympatico, Canada
Gunter Mussbacher	Carleton University, Canada
Najm, Elie	ENST, France
Ileana Ober	Institut de Recherche en Informatique de Toulouse, France

Iulian Ober	Institut de Recherche en Informatique de Toulouse, France
Andras Pataricza	Budapest University of Technology and Economics, Hungary
Dave Penkler	HP, France
Dorina Petriu	Carlton University, Canada
Javier Poncela González	Universidad de Málaga, Spain
Andreas Prinz	Universitetet i Agder, Norway
Rick Reed	Telecommunications Software Engineering, UK
Laurent Rioux	Thales, France
Manuel Rodríguez-Cayetano	Universidad de Valladolid, Spain
Richard Sanders	SINTEF, Norway
Amardeo Sarma	NEC Europe, Germany
Bran Selic	Malina Software, Canada
Edel Sherratt	Aberwrystwyth University,
Francis Tam	City University of Hong Kong, Hong Kong
Peter Tröger	Potsdam University, Germany
Toeroe, Maria	Ericsson, Canada
Martin von Löwis	Beuth-Hochschule für Technik Berlin, Germany
Thomas Weigert	Missouri University, USA

Table of Contents

Model Transformation

Specification and Description Language and Evolution

Data Flow Testing in TTCN-3 with a Relational Database Schema

Gusztáv Adamis[1], Antal Wu-Hen-Chang[1], Gábor Árpád Németh[1], Levente Erős[2], and Gábor Kovács[2]

[1] Ericsson Hungary
Irinyi J. u. 4-20, H-1117, Budapest, Hungary
{gusztav.adamis,antal.wu-hen-chang,gabor.arpad.nemeth}@ericsson.com
[2] Department of Telecommunications and Media Informatics
Budapest University of Technology and Economics
Magyar tudósok körútja 2, H-1117, Budapest, Hungary
{eros,kovacsg}@tmit.bme.hu

Abstract. In script based testing traditionally test data and test environment parameters are stored either in the test script document or in a separate file. In some test execution systems, test data items are loaded to a database during the initialization, however that set of data remains static during the execution. In this paper, we propose a TTCN-3 based approach that stores all test case related data, even constants, local variables and parameterized message templates of a test case in a relational database. Data types and data type instances are all mapped into SQL schemas. When executing a test case, appropriate test templates are fetched or even generated on-the-fly for the subsequent test step, which results in a higher data flow coverage. The course of test events are logged in the database that makes reproduction possible.

Keywords: TTCN-3, test data, data flow, relational database.

1 Introduction

Web portal development frameworks such as Django [1], Ruby on Rails [2] and many others have an incorporated script based test environment. As web portals usually have a database back-end as data source in their production environment, test data to be used in the scripts defined in plain text files must be loaded into the test database before the test execution. Test data definition files contain finite sets of records of the data types used within the portal, and it is possible to execute the same test case with different test data records. Once loaded, these records can be accessed, fetched and modified during the execution of a test case, however the changes made to them cannot be traced back, as they are restored to their initial values upon the initialization of the next test case. Other limitations of these frameworks are that: the test data items must be given manually and therefore have fixed initial values; and there is no obvious way to assign a random value to a field of a record, which is transparent for the current test case.

F. Khendek et al. (Eds.): SDL 2013, LNCS 7916, pp. 1–18, 2013.

The UML and the UML Testing Profile are proposed in [3] for the data-driven testing of object-oriented models. These together provide the means for defining values in the test specification resulting in reusable and maintainable tests. Several testing frameworks exist already with the ability to separate data. SoapUI [4] is a general purpose testing framework that allows the user to use data sources with many formats, such as XML, Excel configuration files or databases. During test execution, test data items are read from the data source independently from the test steps defined in a separate document. MBUnit [5] is a .NET unit testing framework that also supports the separation of the data flow. Beyond being able to use hard-coded test data values, it is capable of getting test data from external sources such as XML spreadsheets or CSV files. EasyTest [6] is another new open source testing framework that allows for writing data-driven tests in JUnit. The Fitnesse framework [7] is primarily for testing requirements, and it already supports data-driven testing.

In TTCN-3, test data is not separated from the control flow of test cases as much as in the case of web development frameworks. Test data items appear within the TTCN-3 test cases themselves. Implementation and environment parameters are loaded from configuration files, record type and template definitions are vital parts of the TTCN-3 module, and the values used by local variables and constants are hardcoded into the test case part of the module.

In this paper, we introduce a framework for testing telecommunication protocols on the pattern of web development test environments, we put all the test data into a relational database that we access from test case bodies during the test execution. The rationale behind this approach is that many higher layer protocols are stateless or have only a very few control states, but at the same time they use complex protocol data units. When conformance testing such protocols, the very same message sequences are used over and over again, but with different test data. This paper investigates how the test data can be stored in a database, and if values for complex data structures can be generated on-the-fly based on user defined records of that database. The record type definitions available for the TTCN-3 test case are mapped to an SQL metamodel, and parameterized templates are records in those tables. We consider that all interactions of the test case take place by means of using parameterized templates, and fetch the parameters before the send and receive operations with an external function. Parameterized templates allow a higher level of flexibility compared to templates with hardcoded values, because parameters not yet accessed can be chosen at random at each execution providing a better data flow coverage than the single hardcoded value.

The cost of this flexibility with the parameterized templates is that we must provide a way to trace back the template value used in a test step and loose the portability of test cases. Therefore to enable persistent logging, we define the formal graph model of a test case including alt statements, branches, loops and local variables and constants that we map to an SQL metamodel as well. This allows the on-the-fly tracking of changes of values of local variables, so that in subsequent test steps the proper value can be used in templates.

The paper is organized as follows. Section 2 defines the architecture of the framework presented. In Sect. 3, we show the graph model of the test case we use in the database to represent the control flow of test cases. Section 4 gives the SQL metamodel for storing record types and parameterized templates, and describes the SQL metamodel we store this test case representation in. How a test case is transformed and executed within this framework is shown in Sect. 5, where we show an example where our method can be useful: a flow control counter overflow check. Finally, Sect. 6 summarizes the paper.

2 A Framework for Storing TTCN-3 Test Data in a Relational Database

The fundamental idea behind this framework is to completely separate test data from the test cases, so that each test case rewritten according to the concepts presented in this paper can be executed with arbitrary number of possibly random test data items without a change. Therefore, we remove all occurrences of values, variable and template accesses from original test case and replace them with external TTCN-3 function calls that implement the fetching and storing of these data items in a database.

When storing test data separately, a database has several benefits over using a simple configuration file. Meta-information such as the type of a field or of a variable can be encoded in the database schema. The complex data structures, or even protocol data units (PDUs), are constructed based on the user defined data, are used in an interaction. These structures or PDUs can be stored in the database and reused when repeating the test. In a long sequence of interactions, the user defined initial values may change, the database helps with tracking these changes, thus the database can store snapshots of the variable vector and associate that with a state of the interaction. Another advantage is that the database manager understands the natural ordering of primitive types such as integer, and for instance boundary values with regard to a table constraint can be very easily fetched.

Our method consists of two phases. First, the input TTCN-3 documents are parsed, and graph models are extracted from them: one that represents the data types available for the test cases, and for each test case a graph that represent the control flow. These pieces of information are then loaded into the database. The second phase is the test execution itself, where the evaluation of expressions and the parameterization of templates are performed within the database.

Figure 1 shows the process of the initialization phase. The input of the process is a set of TTCN-3 documents. These documents are parsed with a customized TTCN-3 parser, and a test data model and test case models are extracted, which are later on loaded into the metadata tables of the database. If data constraints have been defined on the datatypes and the database server supports constraint specification, the schema is updated. We use a custom mapping for that, not a nowadays standard way of object relational mapping (ORM). Transformations are performed on the input TTCN-3 documents that preserve the test case behavior, for each data type definition a set of parameterized template definitions

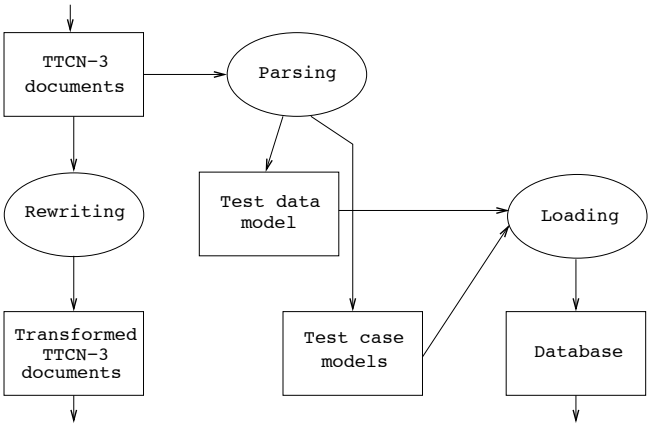


Fig. 1. The initialization of test execution

are generated that are used in `send` and `receive` operations. All variable accesses are replaced with calls to external functions. The outputs of this phase are the transformed TTCN-3 documents and the information stored about the test data and test cases in the database.

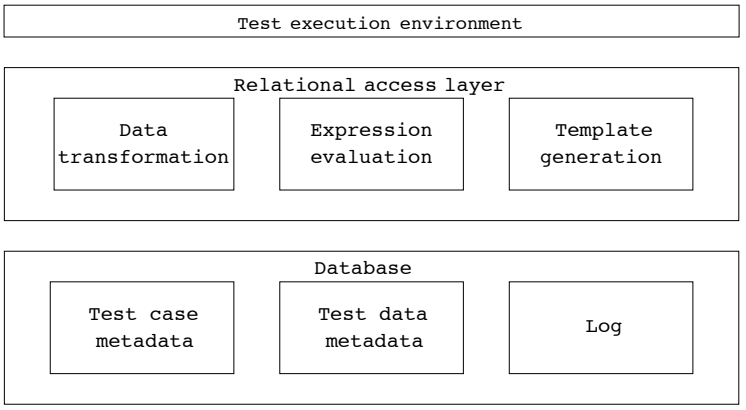


Fig. 2. The layered architecture of the framework, and the main function groups used in test execution

Figure 2 shows the architecture of our framework. When executing a test case, the test execution environment [8] accesses the database by calling the functions of the relational access layer implemented with external TTCN-3 functions. The relational access layer has three main function groups. In the transformed test case, a value can be accessed in three different ways. Data transformation that implements the relational mapping transparently maps database records to native TTCN-3 values and vice versa. When a value is given as an expression with

several variable identifiers, the expression evaluation module resolves the variables, transforms the operator in the expression to SQL syntax, and retrieves the value by executing an SQL query built on the pattern of the expression. The third method of value generation is the use of parameterized templates, which are built on-the-fly with Gecse's [9] method. If a field of a template has not yet been defined in previous interactions of the test case, a value is generated possibly at random and stored in the database for future re-use. The database layer is responsible for storing the data type, template and test case models, and logs the current and previous snapshots of test states of the execution. This latter logging allows the identification of freely definable variables and template fields. The database layer is completely hidden from the test engineers, who write the TTCN-3 test cases. They have to follow a well-defined set of conventions, but are not required to have specific database or SQL knowledge.

In the following, we introduce the data type and test case metamodels we create and map to the database after the parsing of the original TTCN-3 documents.

3 TTCN-3 Data and Test Case Representation

In this section, we define two graphs we use to represent TTCN-3 type and template definitions and TTCN-3 test case definitions. These models can automatically be generated from a TTCN-3 document with a parser and a code generator.

3.1 TTCN-3 Data Representation

In this paper, we use the concept of the type structure graph defined by Gecse's [9]. We model data types with a forest of directed graphs, however our focus is limited to TTCN-3 primitive and record types and TTCN-3 `record of`, `set of`, `set` and `union` types are left out of consideration. We denote the collection of data types with $D = (N, E, L)$ such that $E : N \times L \times N$, where a node $n \in N$ represents a data type, and an outgoing edge $e = (n_i, l, n_j)$, $e \in E$, $n_i, n_j \in N$, $l \in \mathbb{N}$ denotes that the data type n_i has field of the type represented by the target node n_j, and the $l \in L$ label defines the field name and an ordering constraint on the edges. Note that D is not necessarily a tree, the data type definitions may include recursive associations that result in strongly connected cliques.

The set of walks returned by the function $\mathsf{walk} : N \to W$ called subgraph templates in Gecse's work is an unbounded set of finite trees generated by traversing recursively all outgoing edges at each visited node starting from a node $n \in N$ until any node $n_i \in N$, $N_{primitive} \subseteq N$ representing a primitive type is reached. A walk $w \in W$, $W = \mathsf{walk}(n)$, $n \in N$ corresponds to a TTCN-3 parameterized template of type N. A data type instance or a template for walk $w \in W$ is generated by recursively applying the function $\mathsf{val} : W \to \mathcal{D}$ to all subtrees of W, where $\mathcal{D}$ is the domain of W rooted in type N. The function $\mathsf{type} : W \to N$ returns the type of a template, and the function $\mathsf{type} : V \to N$ gives the type of a variable.

```
1  type record M {
2   A a,
3   C c,
4   octetstring d
5  }
6
7  type record A{
8   octetstring a
9  }
10
11 type record C{
12  integer t,
13  integer s,
14  integer r
15 }
16
17 template C t_c (integer v_t
       , integer v_s, integer
       v_r) {
18   t := v_t,
19   s := v_s,
20   r := v_r
21 }
```

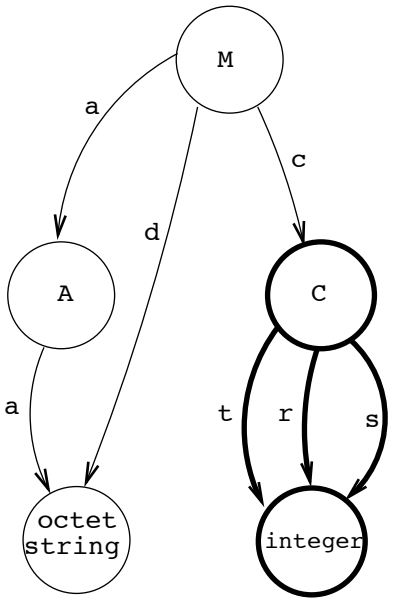


Fig. 3. Graph model of the data types. On the left a TTCN-3 code snippet with some type definitions can be seen, on the right the relationship graph between those types and a sub-template graph with bold lines are shown

Figure 3 shows a sample TTCN-3 type definition code snippet on the left and its graph representation after parsing on the right. In this example, five data types are used, three of them are defined here and two of them are primitive types, so $N = \{M, A, C, \mathsf{integer}, \mathsf{octetstring}\}$. The edge $e = (A, a, \mathsf{octetstring}) \in E$ for instance shows that type A has a field named a of type octetstring. Bold lines represent a walk $t_c \in W$, which is the only available walk from node C, this corresponds to the parameterized template named t_c. This t_c has a subgraph template rooted in node C. Note that the graph model is not necessarily acyclic.

3.2 TTCN-3 Test Case Representation

Mathematically, TTCN-3 test cases can be unfolded into a tree structure that is not necessarily finite. A common model in the literature for representing such trees is a rooted, edge labelled, directed graph, or more specifically a test transition system as defined in [10].

We, in this paper, use a different approach to represent the control flow of test cases. Our model of a test case $TC = (S, P, V, I, O, T)$ is a rooted edge and node labelled directed graph with two types of nodes, which is closely related

to the EFSM model used in [11]. The not necessarily finite set S represents the set of end nodes (test states) of a test step. The other type of node set is the set P that represents the set of branching nodes between test step nodes that correspond to TTCN-3 `if..else if..else` and `select..case` constructs. The set T represents the set of edges of the graph. The sets V, I and O represent the sets of variable identifiers, input template names and output template names respectively. Each $v \in V$ is associated with a type $n \in N$ and a value $\mathsf{val}(\mathsf{walk}(n))$.

Labeling functions assign labels to the nodes and edges of the graph. Each $s_l \in S$ node with the out-degree $deg_{out}(s_l) = 0$ is labelled with the function $\mathcal{L}_s : S \rightarrow \{\mathsf{pass}, \mathsf{fail}, \mathsf{inconc}\}$, which corresponds to the TTCN-3 `setverdict` statement. Each $p \in P$ is labelled with the function $\mathcal{L}_p : P \rightarrow \mathsf{expr}(V)$, where function expr is a variable expression on a subset of V; this corresponds to an `if` condition. Input labels make it possible to assign values to elements of V: $\mathcal{L}_i : I \rightarrow V$, which is mapped from a `receive` statement. By means of output labels it is possible to assign values represented by the $v_i \in V, 0 \leq i \leq |V|$ to an $o \in O$: $\mathcal{L}_o : O \rightarrow \times_k V_k$ on the analogy of a `send` statement. Action labels represent a transformation of values of V: $\mathcal{L}_a : expr(V) \rightarrow V$, these can be generated based on the assignments in the TTCN-3 test case source. Four mutually disjoint subsets of edges that correspond to TTCN-3 statement blocks are distinguished based on the types of nodes they connect: $T = T_{ss} \cup T_{sp} \cup T_{pp} \cup T_{ps}$. Each of these edges is labelled with a set of labeling functions defined in this section. The labeling function for an edge connecting two test states is defined as follows: $\mathcal{L}_{ss} : T_{ss} \rightarrow (S \times V) \times \mathcal{L}_o \times \mathcal{P}(\mathcal{L}_i \cup \mathcal{L}_a) \times (S \times V)$, where $\mathcal{P}$ is the set of all possible ordered subsets operator. The labeling function for an edge connecting a test state node and a predicate node is $\mathcal{L}_{sp} : T_{sp} \rightarrow (S \times V) \times \mathcal{L}_o \times \mathcal{P}(\mathcal{L}_i \cup \mathcal{L}_a) \times (P \times V)$. The labeling function for and edge connecting two predicate nodes is $\mathcal{L}_{pp} : T_{pp} \rightarrow (P \times V) \times \mathsf{expr}(V) \times \mathcal{P}(\mathcal{L}_i \cup \mathcal{L}_a) \times (P \times V)$ The labeling function for an edge that connect a predicate node and a test state node is $\mathcal{L}_{ps} : T_{ps} \rightarrow (P \times V) \times \mathsf{expr}(V) \times \mathcal{P}(\mathcal{L}_i \cup \mathcal{L}_a) \times (S \times V)$.

Figure 4 shows a TTCN-3 code snippet with a part of a test case definition on the left, and its graph representation after parsing on the right. Timeout events are represented with the name of the timer, which is T, and the parameterless `Q.receive` is denoted with an `*` input label. The test purpose of this test case is to check the overflow of a counter. It counts to the maximum value of a domain, for instance the size of a sliding window in the flow control mechanism of a protocol, then it checks if the overflow is handled correctly. The graph model has five nodes in the set S represented with circles in the figure, and three of them are labelled with the possible verdicts. There is one node $p \in P$ represented with a rectangle in the figure, and its label is the `if` condition: $s < w_s$. Edges are assigned with a set of labels. For instance the edge $e_3 \in T_{ss}$ from the unlabeled top node to the $\mathsf{inconc} \in S$ node is labelled with the name of the template sent ($t\_req \in O$) together with its parameters ($s \in V$), a timer action, and an input event ($T \in I$). Edge $e_4 \in T_{ps}$ from the $p \in P$ node to the unlabeled node at the top is labelled with a valid value for $\mathsf{expr}(p)$ that is in this example the boolean value `true`.

```
1  testcase tc_t (integer w_s)
       runs on C_CT{
2   map(mtc:Q,system:Q);
3   /* ... */
4   Q.send(t_req(s));T.start;
5   alt{
6     []Q.receive(t_resp(s,s+1)){
7      s := s+1;
8      if (s < w_s) {
9       Q.send(t_req(s));T.start;
10      repeat;
11     }
12    }
13    []Q.receive {
14     setverdict(fail)}
15    []T.timeout {
16     setverdict(inconc)}
17  }
18  Q.send(t_req(w_s));T.start;
19  alt{
20    []Q.receive(t_resp(w_s,0)){
21     setverdict(pass)}
22    []Q.receive {
23     setverdict(fail)}
24    []T.timeout {
25     setverdict(inconc)}
26  }
27 }
```

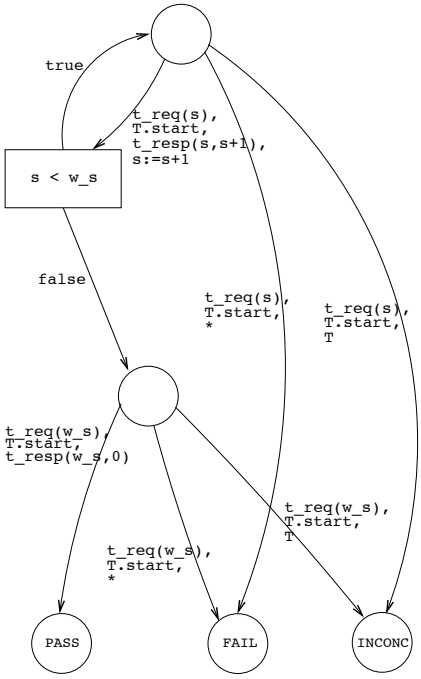


Fig. 4. Graph model of the test case. On the left a TTCN-3 code snippet with a part of a test case definition is shown, on the right its graph representation is shown

4 Mapping TTCN-3 Documents to a Relational Schema

Mapping data structures of different programming languages to database tables has been studied for more than two decades. Several frameworks have been proposed and developed, and some of these have made their way to the everyday practice. A widely used framework that implements the persistence of Java and .NET objects is for instance Hibernate [12][13], while Django [1] provides a framework for Python, and Ruby on Rails [2] gives a programming interface for Ruby.

The main idea of these solutions is that with a little piece of meta-information it is possible to make a one-to-one correspondence between the data types and database tables, data type attributes and database table attributes, and data type instances and database records. Association and aggregation relations

between data types and collection types are mapped by means of foreign keys in tables.

Former frameworks such as [14] intended just to hide the database back-end from the application. Our approach follows this track, because of the metadata necessary to enable the efficient tracking of changes of variable values and to support logging.

4.1 Mapping TTCN-3 Type Definitions to a Relational Schema

Several approaches can be defined to map a TTCN-3 type definition to a relational schema. Though TTCN-3 is not an object-oriented language, object-relational mapping should be considered: mapping record types to tables, record fields to table attributes, and values to the records of that table. In this case, the mapping from TTCN-3 to the database is very easy, however the reverse direction is far less straightforward as type definitions are available at runtime, and the meta information encoded in the schema may not be enough to reconstruct the value.

We use a mapping that preserves all meta-information. It maps data flow elements of a TTCN-3 document to a relational schema of six tables derived from the test data graph model. Figure 5 shows the entity-relationship diagram of the database schema we use for storing a TTCN-3 type definitions, and below we list these tables and their attributes within parentheses, where the attributes that define the primary key of the table are underlined.

- Datatype(name, type)
- Compositefield(parent, fieldname, typeof)
- Template(name, typeof)
- Templatevalue(template, field, value)
- Typeinstance(name, typeof)
- Typeinstancevalue(variable, field, simplevalue, parent)

The Datatype table stores each data type declared in the testcase. Its name attribute is the name of the data type, and the type attribute can take the following values: primitive, record. These values correspond to the primitive data types and the record type, respectively.

Each node of the data type graph of Fig. 3 is stored as a record in the Datatype table. The attributes of the record are the name of the data type, and primitive if the given data type has no child nodes in the type graph, otherwise the attributes are the type name and record.

The Compositefield table is responsible for storing the type hierarchy, more precisely, the types and field names of record type fields. Each record of this table corresponds to a field of a record type, that is, an edge in the graph of Fig. 3. The parent attribute stores the name of the type of the given field is declared in. The name attribute stores the name of the field, and the typeof attribute stores the type of the field.

The Template table holds information about the parameterized templates defined in the test source. Each of its records corresponds to a template, a bold

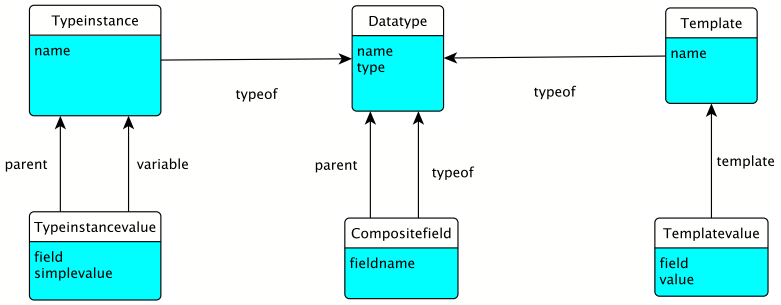


Fig. 5. Entity-relationship diagram for the metamodel of test data and test case used in the database

node in Fig. 3 with with no incoming bold edges. The name attribute stores the name of the parameterized template, and the typeof attribute, which is a foreign key to the Datatype table, stores the data type of the template.

The Templatevalue table stores parameterized templates as defined in the test code. Each of its records corresponds to a template field, to a bold edge in Fig. 3. The template attribute, which is a foreign key to the Template table, stores the name of the template containing the given field. The field attribute stores the name of the field, and the value attribute stores the value of that field.

The Typeinstance table is similar to the Template table. It stores the names and types of data type instances including variables and generated data type instances in its name and typeof fields, respectively.

The Typeinstancevalue table stores the field values of record type instances or the value of primitive type instances. Its attribute named parent stores the name of the parent data type instance of the given field. The field attribute stores the name of the field. If the parent instance is of a primitive type, the simplevalue attribute stores the value of the parent instance, and in that case the field attribute is an empty string, and the variable attribute is null. On the other hand, if the parent type is a record type, then the simplevalue attribute is null, and the variable attribute stores a reference to the data type instance storing the value of the field. The parent relationship is used in the next section when assigning this value to a variable.

As an example of the mapping described in this section, type `C` and template `t_c` are mapped to the following records of tables Datatype, Compositefield, Template and Templatevalue. In the example below, the Datatype table has two records that correspond to two of the types used, `C` and `integer`.

name	type
C	record
integer	primitive

The Compositefield table details the record type `C`, which has three fields, all of the type `integer`, named `t`, `s` and `r` that appear as records of the table.

parent	field	typeof
C	t	integer
C	s	integer
C	r	integer

As a template represents a value of a type available in the Datatype table, the Template table associates the template name and the type name, in this example `C` and `t_c`.

name	typeof
t_c	C

Template `t_c` is an instance of type `C`, hence it has tree fields. The Templatevalue table assigns the value of the variable identified with `v_t` to field `t` of template `t_c`, and similarly initializes the two other fields as well.

template	field	value
t_c	t	v_t
t_c	s	v_s
t_c	r	v_r

4.2 Random Template Generation with SQL

Gecse's method assumes that the relational logic should make it possible to generate random values of a type to be used when generating a template. We use the data structure shown in Fig. 3, that is used as the basis for the template generation. In the following, we show how these random values can be obtained using SQL queries. The generator algorithm is implemented in the Template Generator component of the relational access layer.

In the following queries, there are references to vector elements. These elements are not included in the queries in their presented forms, but are substituted with the values stored in the referenced vector element.

1. The first step of the random template or value generation is the collection of available data types, which can be obtained using the following SQL query:

   ```
   SELECT name, type FROM datatype
   ```

   The result of the query is stored in vector types in the memory.
2. Next, we iterate through types, and generate a number of random names for each type later to be used as identifiers. The names generated for the i^{th} element of types are stored in the vector at the i^{th} position of the vector names. Each type-name pair, in this case the i^{th} type and its j^{th} name, is

stored in the database using the following pseudo-SQL statement:

```
INSERT INTO typeinstance (name, typeof)
  VALUES (names[i][j],types[i])
```

3. Then, all these newly generated type instances are queried:

```
SELECT d.name typename, t.name instance, d.type
  FROM datatype d, typeinstance t
  WHERE d.name=t.typeof AND
   t.name NOT IN (SELECT parent FROM typeinstancevalue)
```

   The result of the above query is the set of all newly generated instance names. Since the Typeinstancevalue table can store some variable values before starting the generation, in order to get the list of newly generated instances, the records belonging to values have to be filtered from the list of all type instances. This is done by not including those instance names in the list, which are already included in the Typeinstancevalue table (the newly generated instances are not yet included in the Typeinstancevalue table so this selection condition is appropriate). The result of this query as a list of type name, instance name, type triplets is stored in vector typesnames in the memory.
4. The next step is to generate the values of each newly generated template name. This is done by iterating through typesnames and generating random values for each instance.
   If the current element of vector typesnames is of a primitive type, then a random value rand is generated according to the type of the instance, and inserted into the database by the following statement:

```
INSERT INTO typeinstancevalue
  (parent, field, simplevalue, variable)
  VALUES (typesnames[i].instance,'',rand,null)
```

   If the current element of this vector is of a record type, then its fields are queried by the following pseudo-SQL query in the i^{th} cycle:

```
SELECT field, typeof FROM compositefield
  WHERE parent=typesnames[i].typename
```

   The result of this query is a list of all the fields that the current type instance has according to its type, and it is stored in vector in the i^{th} position of the vector fields.
5. In the next step, we iterate through this vector, that is, the elements of i^{th} element of fields, and one randomly chosen, previously generated instance name is assigned to each field of the current instance. This is done by the pseudo-SQL query below, and the j^{th} random value is stored at the j^{th} position of the value candidate vector of the i^{th} field, which is an element at

the i^{th} position in the vector value:

```
SELECT name FROM typeinstance
  WHERE typeof=fields[i][j].typeof ORDER BY RANDOM() LIMIT 1
```

The above query selects the list of all instances generated for the type of the current field, which are then ordered randomly (`ORDER BY RANDOM()`), and only the first element of the so-created result is selected (`LIMIT 1`).

6. Finally, the result of this query is stored in the Typeinstancevalue table using the following pseudo-SQL statement:

```
INSERT INTO typeinstancevalue
  (parent, field, simplevalue, variable) VALUES
  (typesnames[i].instance,fields[i][j].field,null,value[i][j])
```

Thus, according to the semantics of the Typeinstancevalue table, the parent attribute gets the name of the instance the field value whose has just been selected. The field attribute gets the name of the current field. The simplevalue attribute is null, since the parent instance is a record. Finally, the name attribute gets the newly selected value.
According to the earlier described semantics of the Typeinstancevalue table, in the case of a type instance of a primitive type, the parent attribute gets the name of the parent type instance, the field attribute is an empty string, the simplevalue attribute is the value generated randomly according to the type of the instance and the variable attribute is null.

4.3 Mapping a TTCN-3 Test Case to a Relational Schema

The model of each test case is stored in a relational schema as well to enable the tracking and logging of test case execution, to be able to track the changes of variable values made by assignment, receipt of messages, and tackle the effect of predicates to the control flow.

Figure 6 shows the entity-relationship diagram of the database schema we use for storing a TTCN-3 test cases. The listing below shows the entities and their attributes within parentheses, and the primary key attributes are underlined, multiple underlined attributes denote a composite primary key.

- State(id, verdict)
- Transition(id, from, next, output, parameterlist, expression)
- Action(transition, serial, type, expression, input, value)

The State table stores information about states of the test case. For each state of the test case, one record is inserted into this table. The id attribute is the artificially generated identifier of the state as test states are unlabeled. The verdict is the verdict value that labels leaf states. For non-leaf states, the verdict attribute is null.

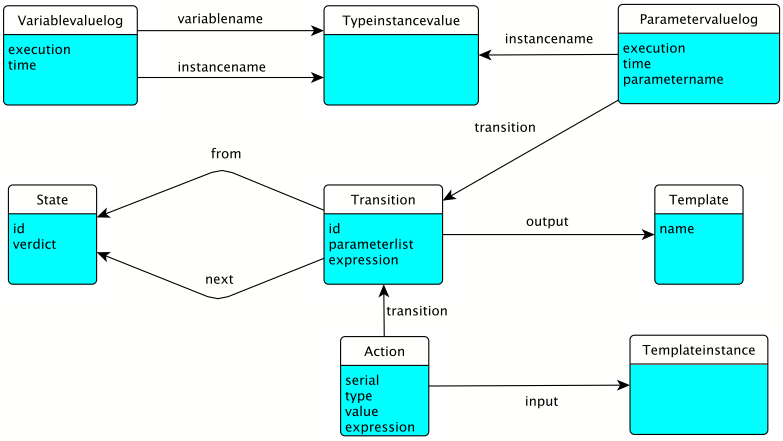


Fig. 6. Entity-relationship diagram for the metamodel of test data and test case used in the database

For each transition one record is inserted into the Transition table. The id attribute of a transition is its artificially generated identifier. The from and next attributes store the current and next test states of the corresponding transition, respectively, and these two attributes are foreign keys to the State table. The output attribute stores a reference to the template sent as output, and the parameterlist attribute stores the list of those variables, which are the parameters of that template. These two attributes are null in case of a predicate transition, i.e., a transition starting from a predicate denoted with a rectangular node in Fig. 4. For such transitions, the expression attribute stores the triggering expression, which is null for output triggered transitions originating in test states.

The Action table stores the input events and value assignments performed during state transitions, whose ordering is determined by the serial attribute. Its transition attribute refers to the transition, during which the action is performed, thus, it is a foreign key to the Transition table. The type attribute of the action can take two values: `assignment` and `input`. The former represents a value assignment, where an internal variable of the test case gets the value of an evaluated expression, while the latter represents a message input action. The expression attribute stores the expression that updates a variable in case of an assignment, while the value attribute, a foreign key to the Typeinstance table, stores the variable, to which the input value or the expression value is assigned in case of an input or assignment action, respectively. The input attribute that is a foreign key to the templatevalue table stores a reference to the input template. The following two functions define the mappings of action and input labels of the test case model to this relational schema.

Logging and making snapshots of test states are vital parts of our framework. For logging the test execution, two additional tables are defined that together can be used for tracing any test execution (see top of Fig. 6).

- Variablevaluelog(execution, time, variablename, instancename)
- Parametervaluelog(execution, time, transition, parametername, instancename)

The Variablevaluelog table stores the data type instances used as values of variables during test execution. Each record of this table belongs to one variable value assignment, where variable value assignment means selecting a value from the database and assigning it to a variable. The execution attribute of the Variablevaluelog table stores the identifier of the test execution, while the time attribute stores the timestamp of the assignment. The variablename attribute stores the name of the variable and the instancename stores the identifier of that randomly generated data type instance the value whose is assigned to the variable. The instancename and variablename attributes are both foreign keys to the Typeinstance table.

The Parametervaluelog table stores the values assigned to template parameters during test execution, each record of this table belongs to a parameter value assignment. The execution attribute of the table is the identifier of the test execution, during which the value assignment is performed. The time attribute is the timestamp of the assignment. The transition attribute stores that transition, during which the message with the value assigned parameter is sent. The parameternname attribute stores the name of the parameter, while instancename stores the identifier of the data type instance assigned to the parameter.

5 TTCN-3 Test Case Transformation and Execution

The role of test case transformation is to remove all data from the test case definitions, and to access all those pieces of data from the database through external functions:

- For each data type, a parameterized template definition is generated unless it already exists such that it has a parameter for each field of the type of the template. For type M in Fig. 3 the following template is constructed:

```
1 template M t_m (A v_a, C v_c, octetstring v_d) {
2   a := v_a,
3   c := v_c,
4   d := v_d
5 }
```

- Original template references are replaced with these parameterized template definitions. In the example we use in this paper, there is one parameterized template for each data type. Any template with a different parameter list is replaced with this most general template. For instance, if before the transformation a template definition for type `M` existed with a single parameter, it is replaced with the definition above.
- All variables, templates and formal parameters are redeclared as TTCN-3 `anytype`, however in the database these are still represented with their real type. The `anytype` is the only type in TTCN-3 that is able to represent all

possible types, therefore it can be used in the native layer to dynamically generate a value of the proper type. Note that this is removed from the examples for the sake of better readability. In the example of Fig. 4, the formal parameter variable `integer w_s` is transformed into `anytype w_s`, and its value is referred with `w_s.integer`.
- For each literal, a local constant is introduced in the test case body, that constant is initialized with that literal, and the value is saved by an external function to the database. In the example shown in Fig. 4, there is one literal: `1`. The local constant is introduced in `const integer c_ONE=1`, and it is stored with `var('c_ONE','integer','1')`.
- For each constant within the test case body, the value is loaded to the database, and the constant is initialized with an external function.
- For each expression, the expression is passed to the relational access layer as a charstring that returns with the evaluated value. For instance, `expr('i+1')` resolves the expression, replaces the `+` operator with its SQL correspondent, which is in this case `+`, and executes an SQL query that reads the value of the resolved variables and evaluates the expression. The expression in this case is translated to the following query:

```
SELECT simplevalue+1 FROM typeinstancevalue WHERE parent='i'
```

- An assignment is replaced with an external function call that pairs the name of the variable with the value of the right hand side expression. For instance, the `i := i + 1` of Fig. 4 is replaced with `set('i',expr('i+c_ONE'))`.
- A variable access such as in template parameters is replaced with a function call that returns with the proper value. For instance, template `t_c(c_ONE, r, s)` is substituted with `t_c(val('c_ONE'),val('r'), val('s'))`.
- After every `receive` operation, the template and its fields are stored in the database with a `set` external function call.

During test case execution, when a test case is initialized, the database, the log and snapshot are reset, variables are set to their initial values. Whenever a `send` or `receive` or `timeout` statement is activated or a database access layer function is called, some of the variables are accessed and database statements are executed. When a variable is accessed for the first time, its value is set for the rest of the test case. If it has not yet been defined, then we call that variable free, so it is possible to generate a value for that with Gecse's method described in section 4.2 possibly at random. If the variable has already been set, its value is changed based on the parameters of a `set` function call and its previous value.

The example in Fig. 4 is a TTCN-3 snippet from simplified version of an HDLC flow control. The test case checks if the incrementation of the received sequence number is correct after an overflow. As the value of variable `w_s` is removed from the test case, the same test case can be reused even for different protocol versions of this family.

The parameterization of a frame represented by template `t_m` to be sent still works identically to the original test case. However, as the value of the data field

`d` of type `M` is free in the whole test case, our framework makes it possible to generate its value on-the-fly, instead of using a hardcoded `'0'` value. The same statement holds for the template `v_a` at any point in the example, but the cycle variable `s` that represents the sent sequence number is bound after its first access.

At runtime, the data transformation module saves and fetches the `anytype` values from the memory to the database and vice versa. The easier direction, just like in the case of the object-relational mapping, is the saving of a memory object to the database. This requires only the parameterization and the execution of a few well-defined SQL statements. However, the reverse direction is more challenging. The value fetched from the database must be decoded into a variable of an arbitrary type known only at runtime, while decoders are available only for primitive TTCN-3 types at compile time. The name of that type is known at runtime, so the field of the `anytype` union to be set is known as well. If that type is a TTCN-3 record type, its fields are – recursively – initialized one-by-one in the native module. If that type is a primitive type, the value produced by the decoder is assigned to that field.

6 Conclusion

The framework presented in this paper increases the reusability of test case definitions at the cost of reducing the portability. With test data removed from the TTCN-3 documents, test parameterization takes place in a relational database back-end, and it is possible to test the behavior with different set of parameters. Our method makes the identification of template fields transparent for the executing test case possible, and can generate appropriate values for those fields.

This framework has been implemented as a prototype extension plugin for the TITAN TTCN-3 Test Executor [8]. The test case and test data models are extracted from the TTCN-3 code with a specialized parser. The execution framework that provides the on-the-fly database access consists of a large set of external functions implemented in C++. To make variable identification by variable name possible, we use metadata and C++ reflection mechanisms, and in the TTCN-3 documents all variables and templates are declared as `anytype`. This framework has been tested with SQLite 3 as database back-end.

Our experiments we made on Diameter Base Protocol [15] and two sample protocols show that the overhead of using a database is a little over 1 ms for each access on average. Though this value is small, this framework should not be used for performance testing.

References

1. Django: Django (2005-2013), `http://www.djangoproject.com/`
2. Hansson, D.H.: Ruby on Rails (2013), `http://rubyonrails.org/`
3. Baker, P., et al.: Data-Driven Testing. In: Model-Driven Testing, pp. 87–95. Springer (2008)
4. Soapui.org: Functional Tests – Data Driven Testing (2013), `http://www.soapui.org/Data-Driven-Testing/functional-tests.html`

5. Gallio: MbUnit (2012), http://gallio.org/wiki/doku.php?id=mbunit
6. EaseTech: easytest (2013), http://github.com/EaseTech
7. Fitnesse Test Manager: Fitnesse Resources (2010), http://fitnesse.testmanager.info/fitnesse/DataDrivenTesting.pdf
8. Szabo, J.Z., Csondes, T.: TITAN, TTCN-3 Test Execution Environment. Infocommunications Journal 57(1), 27–31 (2007), www.hiradastechnika.hu/data/upload/file/2007/2007_1a/HT_0701a-6.pdf
9. Gecse, R.: Towards automatic generation of a coherent TTCN-3 template framework. In: Núñez, M., Baker, P., Merayo, M.G. (eds.) TESTCOM 2009. LNCS, vol. 5826, pp. 223–228. Springer, Heidelberg (2009)
10. Tretmans, J.: Testing Concurrent Systems: A Formal Approach. In: Baeten, J.C.M., Mauw, S. (eds.) CONCUR 1999. LNCS, vol. 1664, pp. 46–65. Springer, Heidelberg (1999)
11. Wang, C.J., Liu, M.T.: Generating Test Cases for EFSM with Given Fault Models. In: Proceedings of the Twelfth Annual Joint Conference of the IEEE Computer and Communications Societies, Networking – Foundation for the Future, vol. 2, pp. 774–781. IEEE (1993)
12. O'Neil, E.J.: Object/relational Mapping 2008 – Gibernate and the Entity Data Model (EDM). In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, pp. 1351–1356. ACM (2008)
13. JBoss.org: Hibernate (2013), http://www.hibernate.org/
14. Alashqur, A., Thompson, C.: O-R gateway – A System for Connecting C++ Application Programs and Relational Databases. In: USENIX C++ Technical Conference, pp. 151–170. Usenix Association (1992), http://openlibrary.org/works/OL12608637W/USENIX_C_Technical_Conference
15. Fajardo, V., Arkko, J., Loughley, J., Zorn, G.: Diameter base protocol (2012), http://tools.ietf.org/html/rfc6733

Property Verification with MSC

Emmanuel Gaudin and Eric Brunel

PragmaDev, France
{emmanuel.gaudin,eric.brune}@pragmadev.com

Abstract. In the development process the very first phase focuses on the requirements. Most of the requirements are dynamic and describe how the system reacts to a set of stimuli. Not all the possible reactions are listed in the requirements but some mandatory reactions are described that can be seen as properties. Later in the development process is a real system or a representative model of the future system. At that point it is possible to gather execution traces of the real system. Based on the work of the European PRESTO project this paper describes the work that has been done to use the same kind of model in both cases and match one against the other.

Keywords: MSC, PSC, Sequence Diagram, Property verification, Trace, Artemis

1 PRESTO Presentation

The PRESTO project started on April the first 2011, and its duration is 36 months. It is co-funded by the European Commission under the ARTEMIS Joint Undertaking Programme. The ARTEMIS JU aims to achieve effective coordination and synergy of resources and funding from the industry, the Framework Programme, national R&D programmes and intergovernmental R&D schemes, thus contributing to strengthening Europe's future growth, competitiveness and sustainable development.

The partners involved in this project are Teletel (Greece), Thales Communications (France), Rapita Systems (UK), VTT (Finland), Softeam (France), Thales (Italy), MetaCase (Finland), INRIA (France), University of L'Aquila (Italy), Miltech Hellas (Greece), PragmaDev (France), Prismtech (UK), Sarokal Solutions (Finland).

PRESTO stands for imProvement of industrial Real time Embedded SysTems development prOcess, from a technical point of view the project aims at improving test-based embedded systems development and validation, while considering the constraints of industrial development processes. This project is based on the integration of test traces exploitation along with platform models and design space exploration techniques

The expected result of the project is to establish functional and performance analysis and platform optimisation at early stage of the design development.

F. Khendek et al. (Eds.): SDL 2013, LNCS 7916, pp. 19–35, 2013.

The approach of PRESTO is to model the software/hardware allocation, by the use of modelling frameworks, such as the UML profile for model-driven development of Real Time and Embedded Systems (MARTE). The analysis tools, among them timing analysis including Worst Case Execution Time (WCET) analysis, scheduling analysis and possibly more abstract system-level timing analysis techniques will receive as inputs on the one hand information from the performance modelling of the HW/SW-platform, and on the other hand behavioural information of the software design from tests results of the integration test execution.

In order to verify the functional and non-functional properties, two approaches have been taken into consideration:

1. Verification on the model
   Model checking was proposed in the 1980s independently by Clarke and Emerson [1] and by Quielle and Sifakis [2]. It aims at testing the correspondence between a logical formula against a mathematical structure, i.e., a model. Model checking is an important member of the family of formal methods, together with testing and deductive verification, all aimed at improving the reliability of systems.
   In recent years model checking has gained popularity due to its increasing use for software system verification even in industrial contexts [3,4]. However the application of model checking techniques is still prevented by the state explosion problem. As remarked by Gerald Holzmann in [5] no paper was published on reachability analysis techniques without a serious discussion of this problem. State explosion occurs either in systems composed of (not too) many interacting components, or in systems where data structures assume many different values. The number of global states easily becomes enormous and intractable.
2. Verification on the traces
   In that approach the system is considered as a black box and a set of typical and representative execution traces are gathered. The functional properties and non-functional properties are then verified on theses traces. The main interest with that approach is that the traces can come from a simulated model or from a real target. This will help to make sure the model is representative of the target.
   Anca Muscholl and Doron Peled have investigated in [6] the automatic verification (model checking) of MSCs, as well as the expressiveness of MSCs, in particular the ability to express communication protocols. Jindrich Babica has discussed Message Sequence Charts properties and checking algorithms in [7]. In [8] the Live Sequence Chart (LSC) language introduces the distinction between mandatory and possible on the level of the whole chart and for the elements messages, locations, and conditions in an MSC. Its primary objective is the application of LSCs in the context of formal system verification.

Of particular novelty in PRESTO is the exploitation of traces for the exclusion of over-pessimistic assumptions during timing analysis: instead of taking all possible inputs and states into account for a worst-case analysis, a set of relevant traces is analysed separately to reduce the set of possible inputs and states for each trace.

As a result the work presented here aims at using the MSC as a basis for expressing the properties, tracing, and verifying the properties on the traces.

2 ITU-T Message Sequence Charts

2.1 Scope

The purpose of the MSC (Message Sequence Chart) [9,10] is to provide a language for the specification and description of the communication behavior of system components and their environment by means of message interchange. Since in MSCs the communication behavior is presented in a very intuitive and transparent manner, particularly in the graphical representation, the MSC language is easy to learn, use and interpret. In connection with other languages it can be used to support methodologies for system specification, design, simulation, testing, and documentation.

2.2 Basic Concepts

Figure 1 illustrates an MSC.

Agent Instance. An agent instance starts with an agent instance head followed by an instance axis and ends with an instance tail or an instance stop as shown in the diagrams below.

Message Exchange. A message symbol is a simple arrow with its name and optional parameters attached to it. The arrow can be horizontal, or the arrow can go down to show the message arrived after a certain amount of time or after another event. A message cannot go up.

When the sender and the receiver are represented on the diagram, the arrow is connected to their instances. If the sender is missing it is replaced by a white circle (found message); if the receiver is missing it is replaced by a black circle (lost message). The name of the sender or the receiver can optionally be written next to the circle.

Timers. An agent instance can start a timer that will automatically send back a message when it times out. While the timer hasn't timed out, the instance can cancel it. Specific symbols are available for timer start, cancel and time out, always attached to the instance performing the action. Figure 2 shows timer elements.

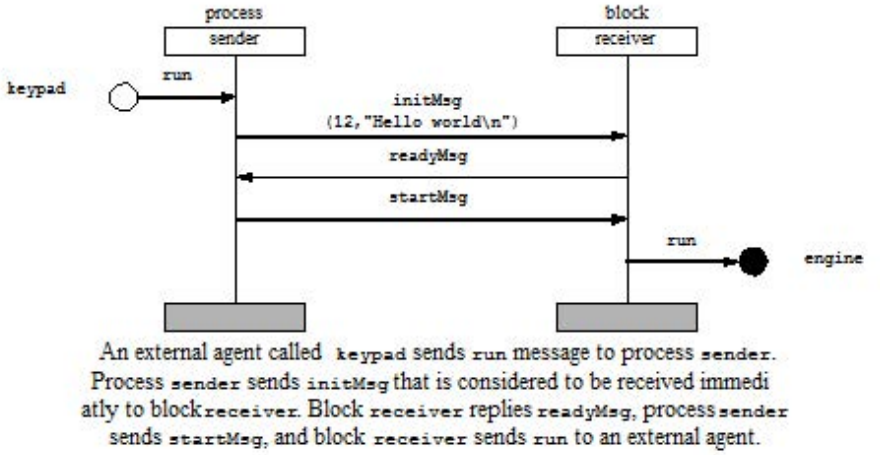


Fig. 1. Message Exchanges in MSC

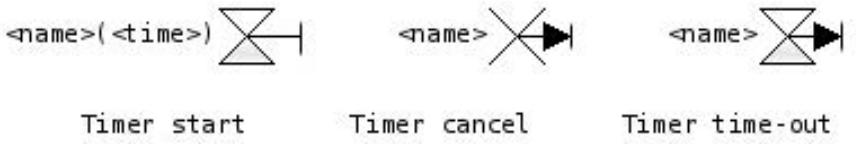


Fig. 2. Timers in MSC

Semaphore Extension. The SDL-RT MSC [9] also introduced the support for semaphore representation. In practice this is rarely used for requirements.

2.3 Inline Expressions

Special semantics can be added to MSC diagrams by the means of inline expressions. These can enclose one or several parts of the diagram and specify that:

- they are optional (opt);
- one or the other part can happen (alt);
- the part can be repeated (loop);
- the parts happens in parallel (par);
- the ordering within the part is not significant (seq).

An example of an alternative inline expression is given on the diagram in Fig. 3:

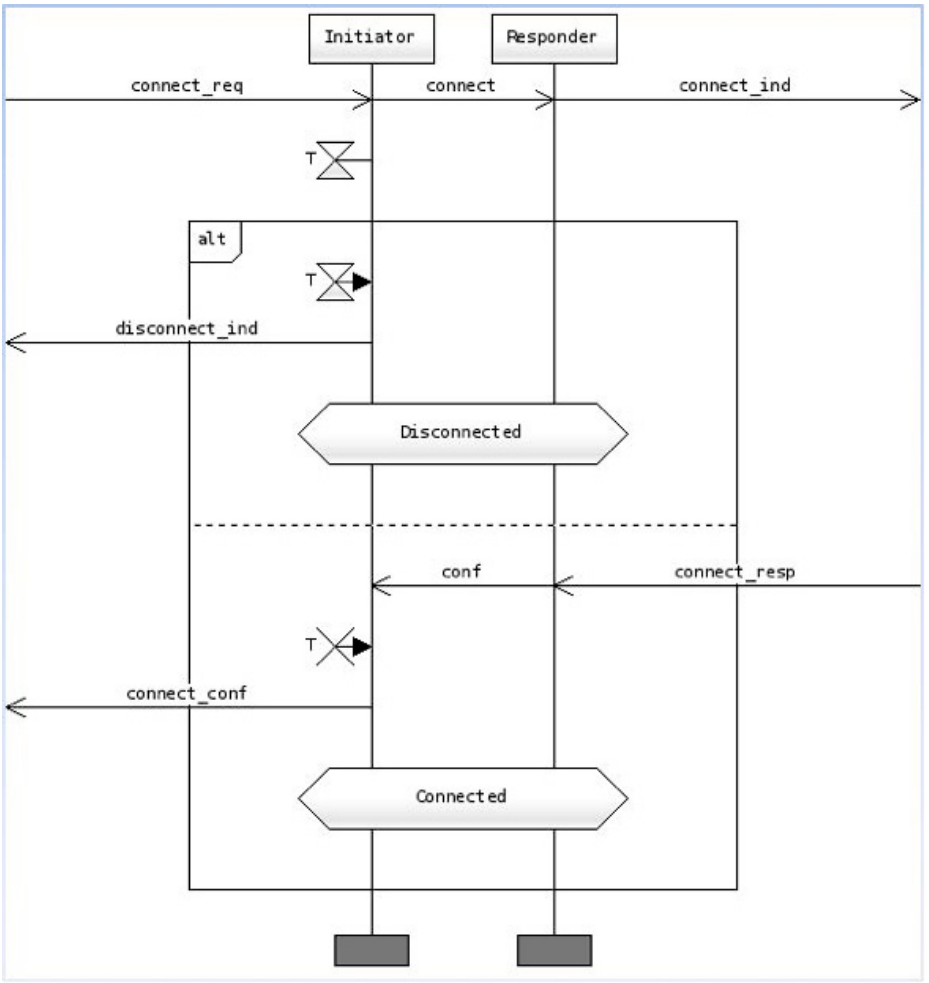


Fig. 3. Inline alternative expression in MSC

2.4 Time Constraints

It is possible to express a relative time constraint in the MSC diagram, specifying a constraint on the time between two events in the diagram. That would define a typical non-functional property of the system. See Fig. 4.

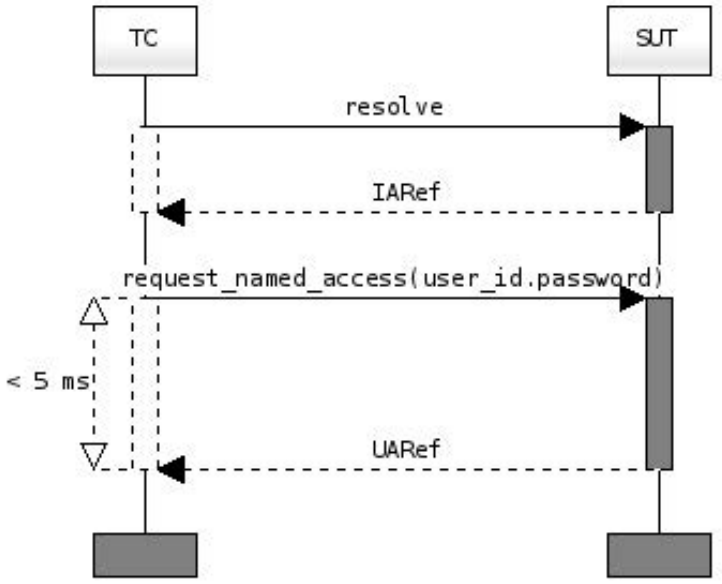


Fig. 4. Time constraint in MSC

3 Property Sequence Chart

Property Sequence Chart (PSC) is a simple but expressive formalism that has been proposed to facilitate the non trivial and error prone task of specifying temporal properties in a correct way and without expertise in temporal logic. PSC is a language that extends a subset of UML 2.0 Interaction Sequence Diagrams or the ITU-T Message Sequence Chart. Further details might be found in [9].

Within the PSC language, a property is seen as a relation on a set of exchanged system messages, with zero or more constraints. PSC may be used to describe both positive scenarios (i.e., the "desired" ones) and negative scenarios (i.e., the "unwanted" ones) for specifying interactions among the components of a system. For positive scenarios, PSC allows to specify both mandatory and provisional behaviours. In other words, it is possible to specify that the run of the system must or may continue to complete the described interaction.

Figure 5 shows the available symbols in PSC diagrams.

Instances are represented as in MSC diagrams. The parallel, alternative and loop operators are represented the same way as the par, alt and loop inline expressions in MSC diagrams respectively. The relative time constraint has the same representation and semantics as in MSCs.

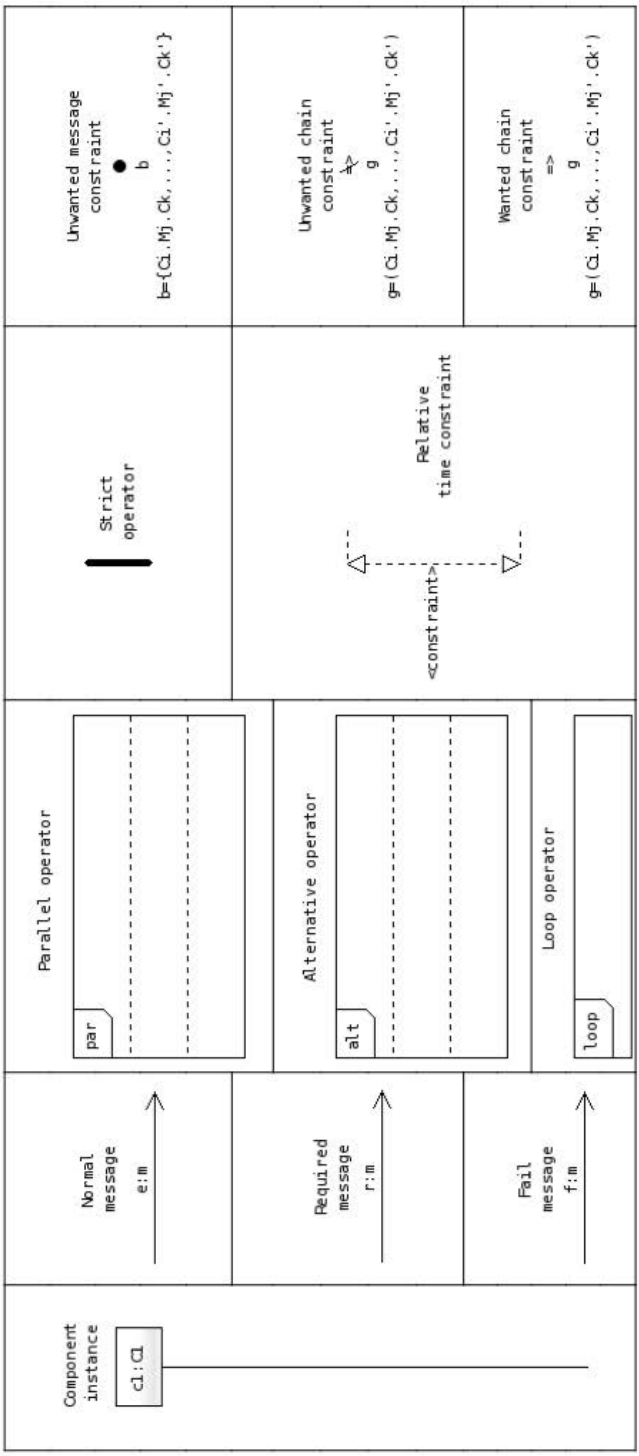


Fig. 5. PSC Graphical notation

Messages in PSCs have two representations:

- An arrow going from the sender to the receiver, just as in MSC diagrams;
- A textual representation, with the format "<sender instance name>.<message name>.<receiver instance name>". This representation is used in constraints, explained below.

Unlike messages in MSC diagrams, message arrows in PSC diagram can have three kinds:

- A regular message, identified by the prefix "e:" for the message text, is a precondition for what follows.
- A required message, identified by the prefix "r:" for the message text, is a message that must occur if the preconditions are met. Required messages must always appear after all regular messages.
- A fail message, identified by the prefix "f:" for the message text, is a message that must not occur if the preconditions are met. Fail messages must also always appear after all regular messages.

When describing a property, the default ordering is the loose ordering: anything can happen between a message specified in the PSC and the one following it. For cases where a strict ordering is necessary, i.e when a message in the PSC must be directly followed by the one following, the strict operator can be used, either on a message send or a receive. See Fig. 6.

The PSC diagrams also allows to set constraints on the messages. These constraints are shown as symbols at the beginning or end of message arrows with an associated text. These constraints can have three types:

1. An unwanted message constraint denotes a set of messages where none should happen before or after the message it is attached too, depending on whether it appears at the beginning or the end of the arrow.
2. An unwanted chain constraint denotes a sequence of messages that should not appear as a whole before or after the message it is attached to.
3. A wanted message constraint denotes a sequence of messages that must appear as a whole before or after the message it is attached to.

A simple example of a PSC diagram is shown in Fig. 7: According to the semantics described above, the property can be read as follows:

- If a message "login" is sent from UserInterface to ATM (normal message "e:login"),
- If a message "wReq" is sent from UserInterface to ATM after the login, without a "logout" message sent from UserInterface to ATM in between (normal message "e:wReq" with the unwanted message constraint "UserInterface. logout.ATM");
- Then a message "uDB" must be sent from ATM to BankDB, unless a message "logout" has been sent from UserInterface to ATM before (required message "r:uDB" with unwanted message constraint "UserInterface.logout.ATM").

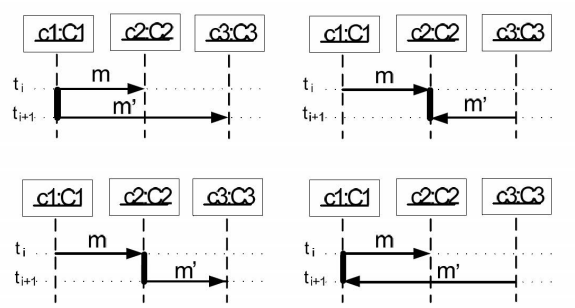


Fig. 6. PSC strict operator example where m' must immediately follow m

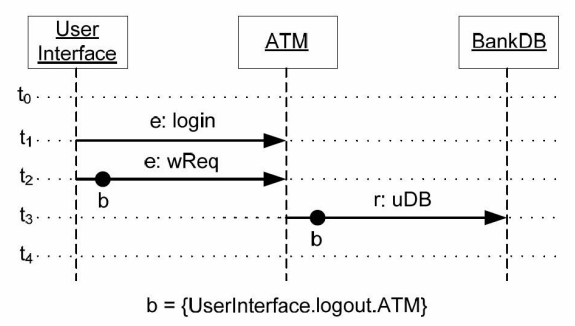


Fig. 7. Example of a property expressed in PSC

The textual notation of PSC together with denotational and operational semantics of the language can be found in [11,12].

4 Using the Same Representation

PSC representation is very close to the MSC representation. In the frame of the PRESTO project the idea is to use the same tool to express both the properties and view the execution traces.

The following sections describe the PRESTO enhancements to the new MSC tracer by PragmaDev. MSC tracer can be used by designers and testers of embedded systems to visualise the flow of control of a system. The benefit of the enhancements is to be able to express a property at the same level as the trace that will be generated by the execution or the simulation of a system. The new research enables the user of the software to write non-functional properties and functional properties at a high level using the well known MSC standard.

5 Functional Property Verification

5.1 Verification for Consistency between Properties, and Collected Traces

The goal of these techniques is to verify that the execution traces conforms to the identified properties. This verification activity checks the consistency between the running system (represented as the observed traces) and the system requirements. In this context, three levels of diagrams will be considered:

1. Requirements
2. Properties
3. Traces

Traces are real execution traces or simulated traces on which a set of requirements or a set of functional properties must be verified. A requirement is basically an expected behavior of a system. It may contain alternatives or loops.

In the example in Fig. 8 the expected scenario is either Stimulus1, Reaction1, EndOfScenario sequence, or the Stimulus1, Reaction2, EndOfScenario sequence.

The traces shown in Fig. 9 will therefore both verify the requirements.

This was a basic approach but the PRESTO project context showed that when it comes to a property, things are slightly different. A property will basically say that if a specific set of events occur, then the following event must or must

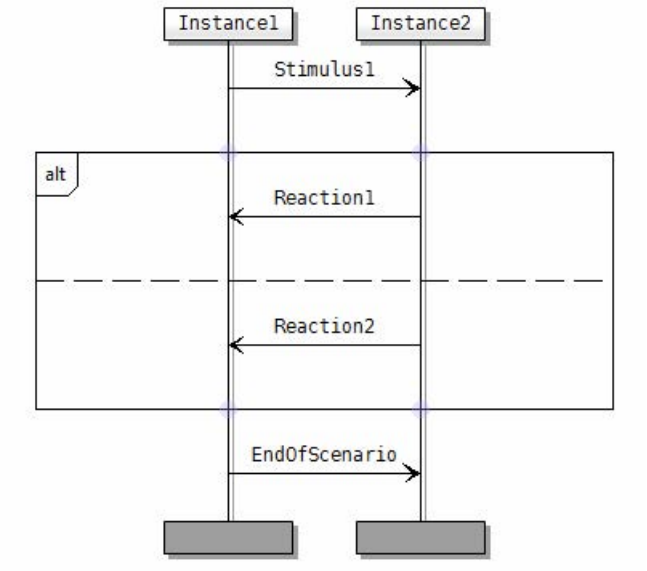


Fig. 8. A simple requirement

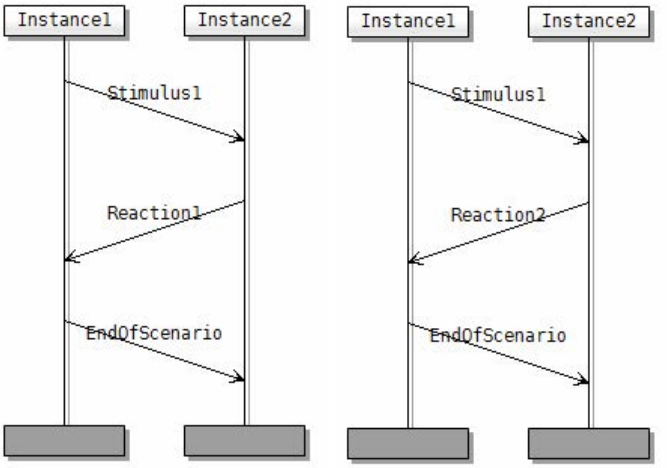


Fig. 9. Two simple traces

not occur. This is very close to what is already available in an MSC. It is just a question of marking the events as a condition, a required event, or a failed event. For that matter the PSC (Property Sequence Chart) complementary notation to the MSC has been adopted.

Therefore the events in the scenario will look like the ones in Fig. 10.

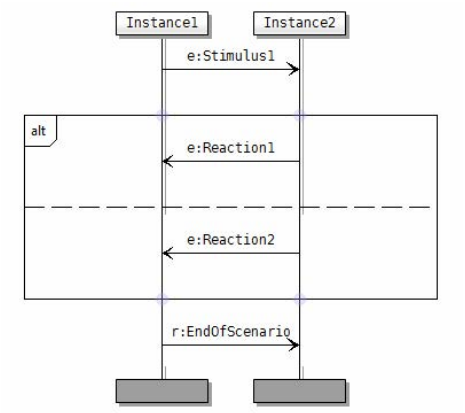


Fig. 10. A property using the PSC notation

That scenario means that if the sequence Stimulus1, Reaction1, or Stimulus1, Reaction2 occurs then the EndOfScenario must occur to verify the property.

This will have very little impact on the look and feel of an MSC but that changes the semantics of the diagram. Therefore three levels of checking have been considered:

- "Basic MSC diff" that makes a simple difference between two diagrams. At this level any logical or graphical difference is considered. A diagram containing an alternative with m1 at the top and m2 at the bottom, and a diagram with m2 at the top and m1 at the bottom are considered different.
- "Spec vs trace" that will handle alternatives and loops in the spec. The two diagrams in the example above would match in that configuration. It is typically intended to deal with a specification diagram without any property against a real execution trace.
- "Property match" that will verify a certain set of events will lead to a required set of events as described in a PSC.

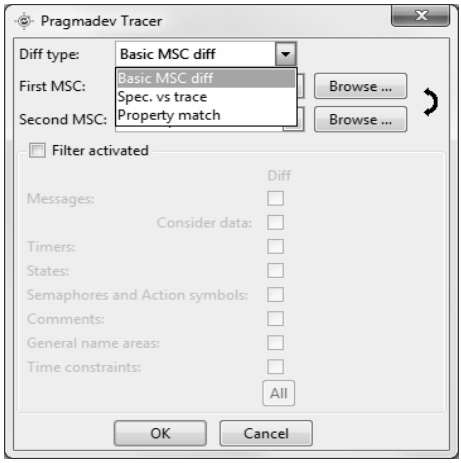


Fig. 11. The PragmaDev Tracer diff options

This has been implemented in the free PragmaDev Tracer prototype; the compare window allows to choose among the different options and appears as in Fig. 11.

Once the property and the trace have been selected, the tool checks the properties on the trace. As a result both diagrams are opened and a third window displays the result of the verification as shown in the example in Fig. 12.

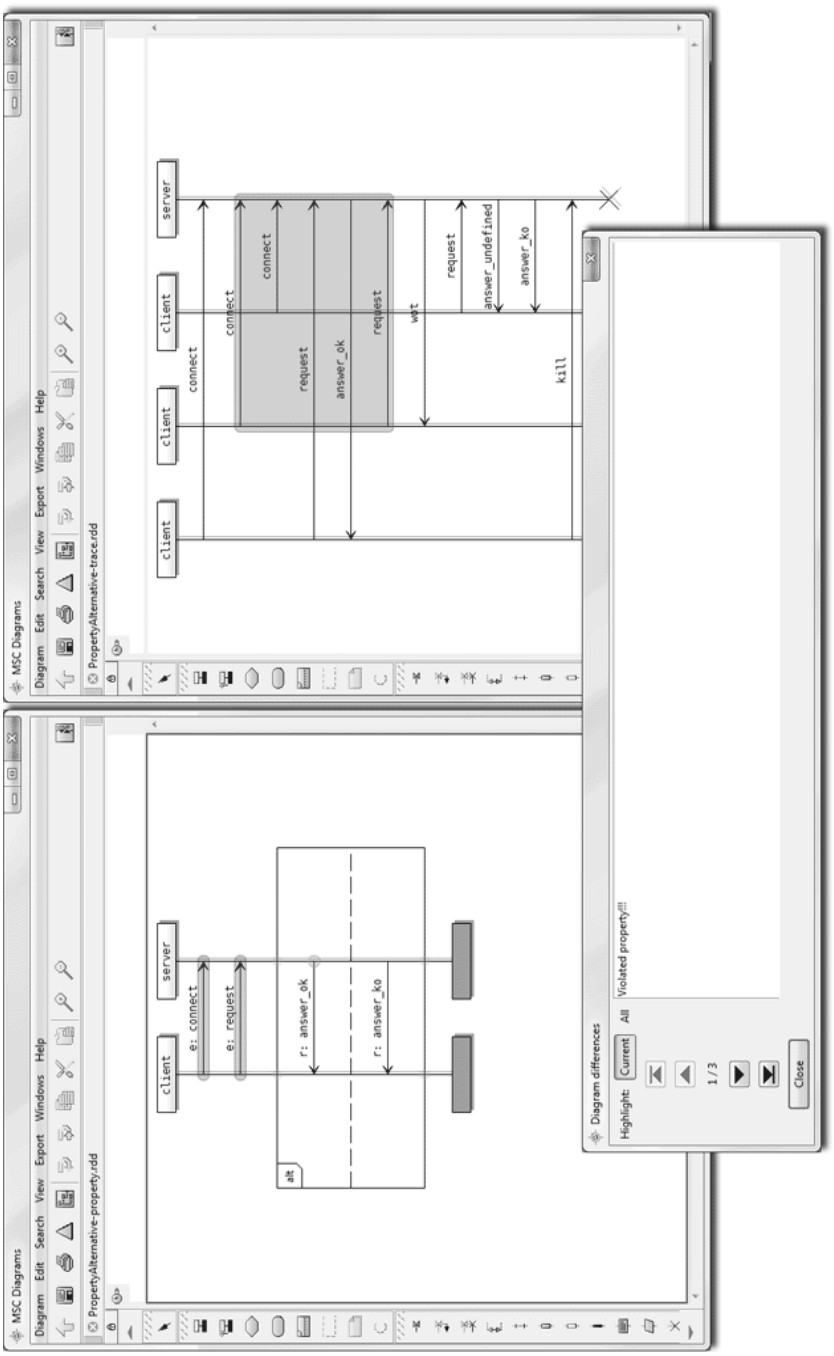


Fig. 12. The property is violated

6 Non-functional Properties

The basic idea is to write non-functional properties at a high level using international standard MSC. One of the main non-functional properties that can be expressed in an MSC is a time constraint in which a set of events must take place in a given amount of time. In the example in Fig. 13 the non-functional property states the exchange of messages between InstanceA and InstanceB must take place within 5 units of time.

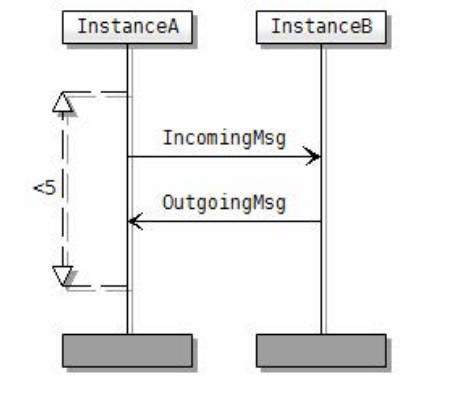


Fig. 13. MSC Message deadline

6.1 Traces

In the trace coming from a real target or from a simulated target, events come with timing information. In the example in Fig. 14 the IncomingMsg is sent at 100 and received at 102. The answer OutgoingMsg is sent at 104 and received at 106. The overall sequence is therefore done in 6 units of time. The new PragmaDev Tracer developed in the context of the PRESTO project can now compare the time constraint in the requirements with a real execution trace.

6.2 Property Verification

In our example in Fig. 15 the time constraint is not fulfilled in the execution trace. PragmaDev Tracer will show clearly the time constraint in the property diagram and the corresponding events in the trace for analysis.

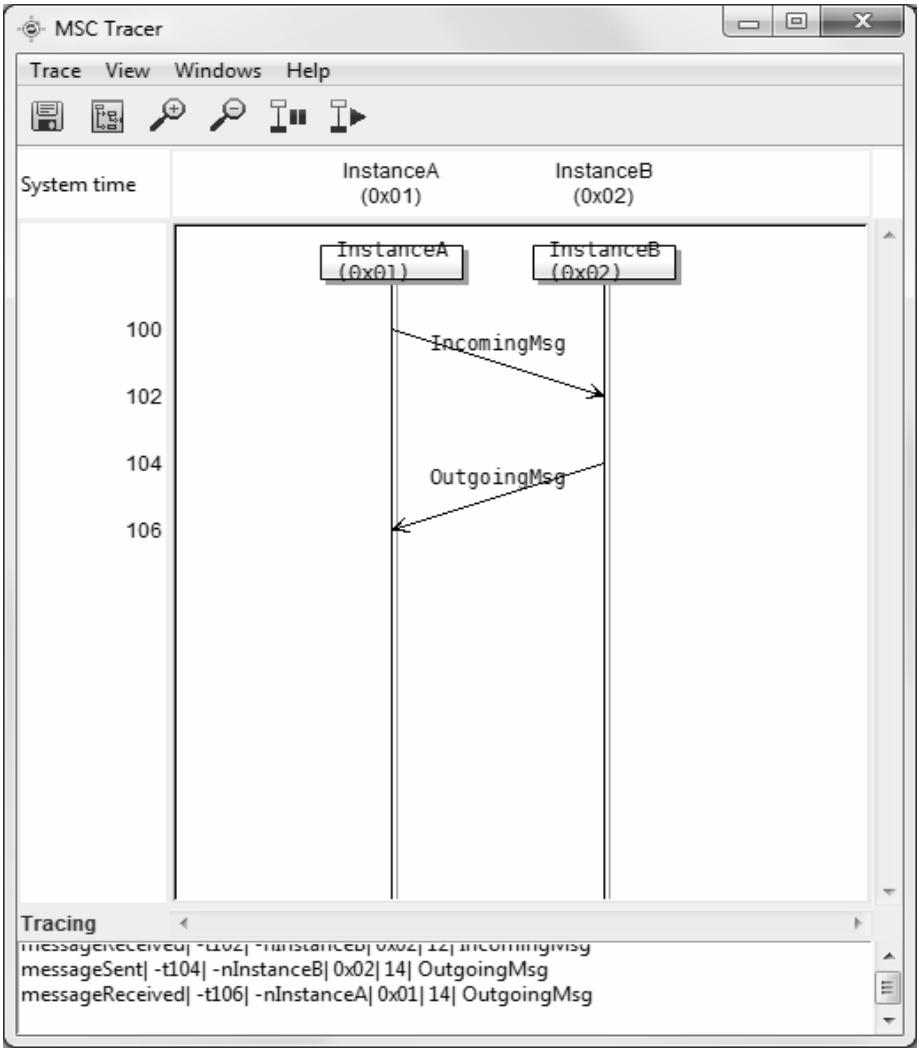


Fig. 14. Screenshot of a real execution trace

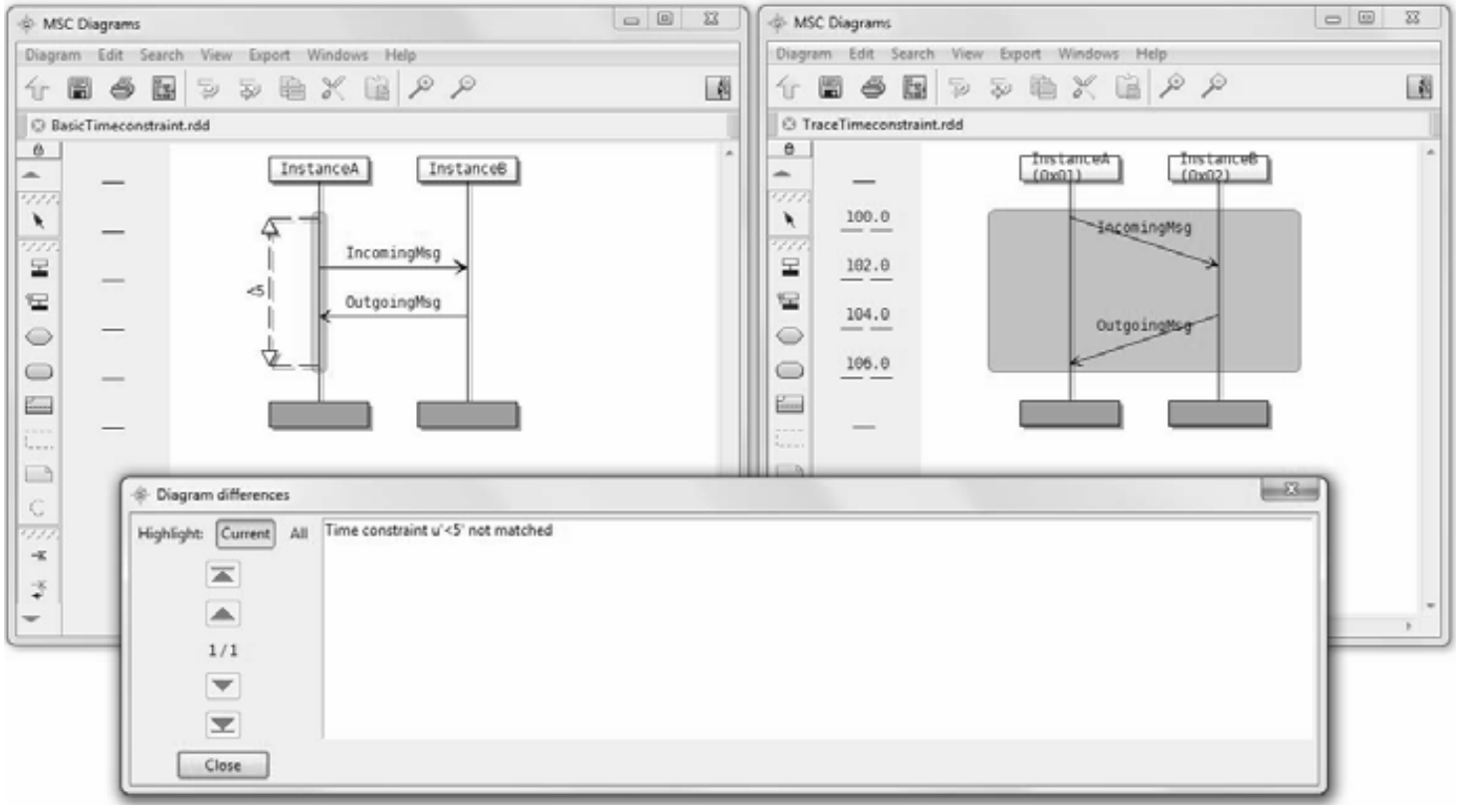


Fig. 15. The timing property is not fulfilled

7 Conclusion

The ITU-T MSC and the PSC are two very close notations that can be used to trace a system behavior and to express properties. The possibility to use both notations in the same tool that will eventually match the properties on real or simulated traces will definitely simplify the verification process.

Thales Italy, one of the partners of the PRESTO project is currently experimenting the new tracer on a real industrial use case. Based on this experiment the tracer that is free to download [13] is constantly evolving.

References

1. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982)
2. Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: Dezani-Ciancaglini, M., Montanari, U. (eds.) Programming 1982. LNCS, vol. 137, pp. 337–351. Springer, Heidelberg (1982)
3. Compare, D., Inverardi, P., Pelliccione, P., Sebastiani, A.: Integrating model-checking architectural analysis and validation in a real software life-cycle. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 114–132. Springer, Heidelberg (2003)
4. Holzmann, G.J.: The SPIN Model Checker – Primer and Reference Manual. Addison Wesley (2003)
5. Holzmann, G.J.: The logic of bugs. In: Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering, SIGSOFT 2002/FSE 2002, pp. 81–87. ACM (2002)
6. Muscholl, A., Peled, D.: Deciding Properties of Message Sequence Charts. In: Leue, S., Systä, T.J. (eds.) Scenarios. LNCS, vol. 3466, pp. 43–65. Springer, Heidelberg (2005)
7. Babica, J.: Message Sequence Charts properties and checking algorithms. Master Thesis at Masarykova Univerzita Fakulta Informatiky Brno (2009), http://scstudio.sourceforge.net/files/thesis_babica09.pdf
8. Brill, M., Damm, W., Klose, J., Westphal, B., Wittke, H.: Live Sequence Charts. In: Ehrig, H., Damm, W., Desel, J., Große-Rhode, M., Reif, W., Schnieder, E., Westkämper, E. (eds.) INT 2004. LNCS, vol. 3147, pp. 374–399. Springer, Heidelberg (2004)
9. Specification & Description Language - Real-Time (2006), http://www.sdl-rt.org/
10. Internation Telecommuication Union: Recommendation Z.120 (02/11) Message Sequence Chart (MSC), http://www.itu.int/rec/T-REC-Z.120
11. Autili, M., Inveradi, P., Pelliccione, P.: Graphical scenarios for specifying temporal properties: an automated approach. Automated Software Engineering 14(3), 293–340 (2007)
12. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in Property Specifications for Finite-State Verification. In: Proceedings of the 21st International Conference on Software Engineering, ICE 1999, pp. 411–420. IEEE Computer Society (1999)
13. PragmaDev Tracer, http://www.pragmadev.com/product/tracing.html

Towards the Generation of AMF Configurations from Use Case Maps Based Availability Requirements

Jameleddine Hassine[1] and Abdelwahab Hamou-Lhadj[2]

[1] Department of Information and Computer Science
King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia
jhassine@kfupm.edu.sa
[2] Electrical and Computer Engineering Department
Concordia University, Montréal, Canada
abdelw@ece.concordia.ca

Abstract. Dependability aspects, such as availability and security, are critical in the design and implementation of distributed real-time systems. As a result, it is becoming crucial to model and analyze dependability requirements at the early stages of system development life-cycle. The Service Availability Forum (SA Forum) has developed a set of standard API specifications to standardize high-availability platforms. Among these specifications, the Availability Management Framework (AMF) is the service responsible for managing the availability of the application services by handling application redundant components, dynamically shifting a workload of a faulty component to a healthy component. To manage service availability, AMF requires a configuration of the application it manages. This configuration consists of a logical view of the organization of the application's services and components. Recognizing the need to plan for availability aspects at the early stages of system development life-cycle, this paper proposes an approach to map high level availability requirements into AMF configurations. The early availability requirements are expressed in terms of the Use Case Maps (UCM) language, part of the ITU-T User Requirements Notation (URN) standard. Our approach allows for the early reasoning about availability aspects and promotes the portability and the reusability of the developed systems across different platforms.

1 Introduction

Several definitions of availability have been proposed [1,2,3,4,5,6]. According to IEEE [1], the availability of a system may be defined as the degree to which a system or a component is operational and accessible when required for use. The ITU-T recommendation E.800 [3] defines *availability*, as the ability of an item to be in a state to perform a required function at a given instant of time, or at any instant of time within a given time interval, assuming that the external resources, if required, are provided. Wang and Trivedi [4] define the availability

F. Khendek et al. (Eds.): SDL 2013, LNCS 7916, pp. 36–53, 2013.

as the probability of service provision upon request, assuming that the time required for satisfying each service request is short and negligeable.

Availability requirements can be very stringent as in highly available systems used in telecommunication services (a.k.a. 5 nines (99,999%)). Many proprietary approaches have been proposed to achieve high-availability. However, such solutions hinder the portability of applications from one platform to another. To address this issue, the Service Availability Forum (SA Forum) [7], a consortium of telecommunications and computing companies was created to define and standardize high availability solutions for systems and services. SA Forum [7] supports the delivery of highly available carrier-grade systems through the definition of standard interfaces for availability management [10], software management [8] and several other high availability middleware services [9]. SA Forum [7] has developed an Application Interface Specification (AIS), which includes the Availability Management Framework (AMF)[10]. AMF constitutes the core component of the middleware as it is the service responsible for managing the high availability of the services.

An AMF configuration describes an application in terms of logical entities representing services and service provider resources. The application software, managed by AMF, is described by the vendor in the Entity Types File (ETF) [8] in terms of entity prototypes that characterize the deployment options, constraints and limitations of the software. Many attempts to construct AMF configurations from user and vendor requirements, have been addressed in the literature [11,12,13,14,15]. Salehi et al.[11] have presented a model based approach for generating AMF configurations using UML profiles. The authors have defined a set of transformation rules, expressed in the ATLAS Transformation Language (ATL), to generate AMF configurations from UML model elements representing software entities and configuration requirements. Kanso et al. in [12] and [14] have adopted a code-centric approach. These authors have used a *Configuration Requirements* (CR) artifact to describe AMF middleware requirements for a given application (i.e., ETF types and configuration parameters such as the number of service units (SUs), component service instances (CSIs), etc., provided by a configuration designer), allowing for automatic generation of AMF configuration. In a closely related work Colombo et al. [13] have proposed an approach that aims at producing multiple sets of Configuration Requirements (CR) (resulting in multiple AMF configurations) from User Requirements (UR) and based on a selection mechanism of ETF types [8].

The Use Case Maps (UCM) language, part of the ITU-T User Requirements Notation (URN) standard [16], is a high-level visual scenario-based modeling language that has gained momentum in recent years within the software requirements community. Use Case Maps [16] can be used to capture and integrate functional requirements in terms of causal scenarios representing behavioral aspects at a high level of abstraction, and to provide the stakeholders with guidance and reasoning about the system-wide architecture and behavior. System non-functional aspects such as *availability* and *security* are often overlooked and underestimated during the initial system design. To address this

issue, the UCM language has been extended with availability information in [17] and [18]. These extensions cover the well-known availability tactics, introduced by Bass et al. [19].

Availability requirements modeling and analysis constitute the major motivation of this research. We focus on the need to express system availability aspects while assuring portability of applications. This paper serves the following purposes:

- It extends the UCM-based availability annotations introduced in [17] and [18] to accommodate Availability Management Framework (AMF) [10] concepts (e.g., *Service group*, *service unit*, etc.).
- It provides a mapping of the newly introduced UCM-based availability requirements to AMF (Availability Management Framework) [10] concrete APIs.
- It complements the approach introduced in [12]. The configuration requirements (CR) model can be extended and automatically derived from UCM specifications annotated with availability aspects.
- It extends our ongoing research towards the construction of a UCM-based framework for the description and analysis of availability aspects in the very early stages of system development life cycle.

The remainder of this paper is organized as follows. The next section introduces the Availability Management Framework (AMF). Section 3 presents our UCM-AMF configuration generation approach. Use Case Maps availability modeling is provided in Section 4 followed by a discussion in Section 5. An illustrative example is presented in Section 6 to demonstrate the applicability of our approach. Finally, conclusions and future work are outlined in Section 7.

2 The Availability Management Framework (AMF)

The role of AMF is to manage the availability of applications in a clustered environment (note that we use here the term AMF to refer to an implementation of the AMF standard, since AMF is just a specification). To do so, AMF needs a configuration of the components (service providers) and the services.

An AMF configuration consists of a number of logical entities, introduced in the AMF standard [10]. An example is shown in Fig. 1. In this figure, we can see that each node has two components grouped in a logical AMF entity called service units (SUs). The services are represented by service instances, also known as component service instances (CSIs). Multiple CSIs can be assigned to the same SU and are grouped in another of AMF entities called service instance (SI).

There are two additional AMF logical entities used for deployment purpose: the cluster and the node. The cluster consists of a collection of nodes under the control of AMF.

AMF supports five different redundancy models, namely, 2N, N+M, N Way, N Way Active, and No-Redundancy. These redundancy models vary in the level

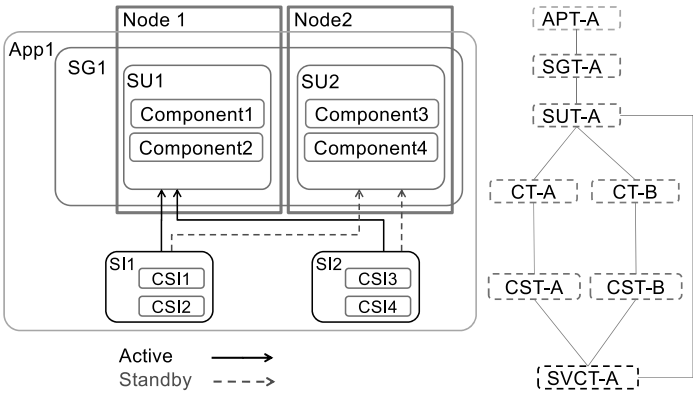


Fig. 1. An example of AMF configuration and its services. The left figure shows a configuration. The right figure shows type of the configuration entity types (taken from [11]).

of protection they provide. For example, in a 2N model (see Fig. 1) there is only one SU that is active for an SI and another SU that is used as a standby. In the N+M model, N SUs share the active assignments and M share the standbys. Like 2N models, N+M models allow at most one active and one standby assignment for each particular SI.

The set of SUs that follow the same redundancy model are grouped in AMF logical entities called service groups (SGs). The same configuration can have many SGs. For example, some components can be protected using a 2N redundancy model, whereas others (in different SUs) can be protected using an N+M model. Similarly, multiple SGs can be grouped to form an application. A good reference on AMF redundancy models can be found in [15].

When building a configuration, there are several decisions that need to be made. For example, it is important to put highly coupled components in the same SU. In case a failure happens, we can failover the whole SU to recover the service. In addition, there should be a way for AMF to know which CSIs to assign to a specific component depending on whether the component can provide the service or not. Many other similar decisions are needed to produce valid configurations. AMF types aim to do just that.

In AMF, every entity has a type except the cluster and the node. These AMF types are derived from AIS standard, known as the Entity Types File (ETF) [8], which is a file provided by the software vendor to describe the characteristics of the software system that runs under the control of AMF. ETF types should be thought of as power types (or meta types), that describe the possible ways a software system can be deployed on an AMF cluster. Once a configuration is built, only instances of some ETF types are used to construct AMF types. Table 1 describes the ETF types. Each row shows an ETF type, a description, and the AMF entity for which the type derives from that ETF type.

Table 1. ETF Types

ETF Type	Description	AMF Entity
Component Type (CT)	It describes the component version (used more particularly during upgrades), the component service types that the component of this type can provide, and the component capabilities (how many active and standby CSIs the component of this type can support).	Component
Component Service Type (CST)	It describes the service attributes (e.g., range of IP addresses the component that handles this service can provide).	Component Service Instance (CSI)
Service Unit Type (SUT)	It describes the service type that an SU can provide as well as the set of component types of the components that an SU of this type can contain.	Service Unit (SU)
Service Type (SVCT)	It describes the set of component service types from which its SIs can be built. The service type may limit the number of CSIs of a particular CS type that can exist in a service instance of the service type. It is also used to describe the type of services supported by an SU type.	Service Instance (SI)
Service Group Type (SGT)	It describes the service group. Typical attributes of SGT is the redundancy model (e.g., 2N, N+M, etc.). It also specifies the supported SUTs. In other words, an SG can contain and SU only if its SGT supports the SU's SUT.	Service Group
Application Type	Similar to SGT, an application type defines the SGTs types that are supported by the applications of this type.	Application

3 Extending the Use Case Maps Language with AMF Concepts

Figure 2 illustrates our approach for extending Use Case Maps [16] with AMF [10] concepts. Note that the configuration generation process is outside the scope of this paper. Many algorithms, including the work of Kanso et al. [12], and Salehi et al. [11], exist to generate automatically AMF configurations. Our focus is to model AMF concepts using the Use Case Maps language [16]. By doing so, an AMF configuration (at the conceptual level) will always be represented as a UCM map. Note, however, that the AMF standard defines an XML carrier to exchange configurations among tools. This XML representation of AMF configurations can also be generated from UCM (extended with AMF concepts).

The proposed availability extensions are added orthogonally to the UCM specification (functional model and binding architecture). These extensions are modeled using *Metadata* mechanism, which is a mechanism used to support the profiling of the language to a particular domain. *Metadata* are described as name-value pairs that can be used to tag any URN specification or its model elements, similar to stereotypes in UML. *Metadata* instances provide modelers with a way to attach user-defined named values to most elements found in a URN specification, hence providing an extensible semantics to URN. *Metadata* is supported by the jUCMNav tool [20], the most comprehensive URN [16] tool available to date.

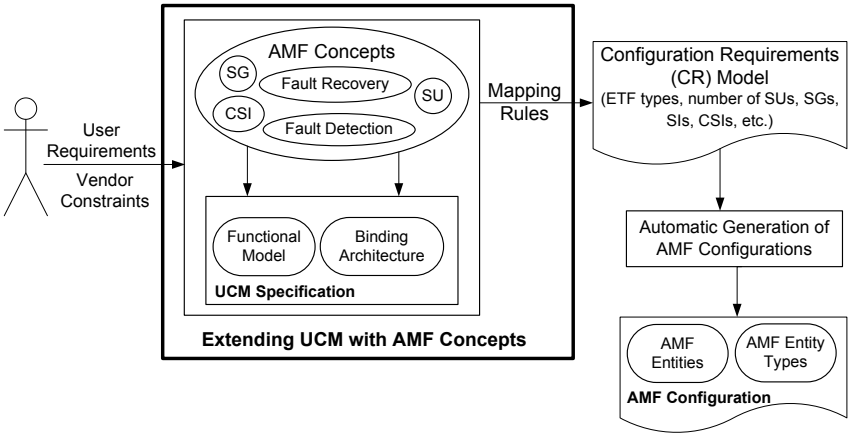


Fig. 2. AMF Configuration Generation Approach

The resulting UCM availability requirements can then be mapped to a configuration requirements (CR) model that describes the used ETF types and configuration parameters such as the number of SUs, CSIs, etc. Finally, based in the AMF requirements model (CR), an AMF configuration model can be generated by leveraging the AMF configuration generation approach proposed by Kanso et al. [12].

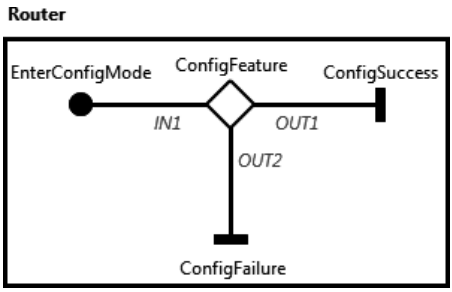
4 Use Case Maps Availability Modeling

In this section, we introduce the basic Use Case Maps constructs and we present our proposed UCM-based availability extensions to cover error detection (Section 4.3) and recovery (Section 4.4). For a complete description of the Use Case Maps language, interested readers are referred to [16].

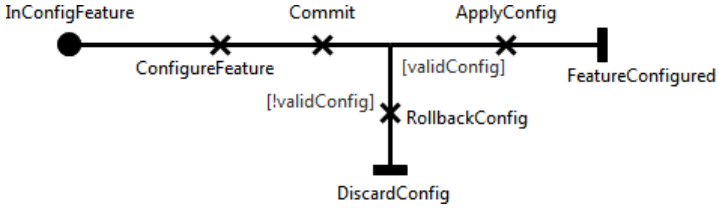
4.1 Use Case Maps Functional and Architectural Features

Use Case Maps (UCM) models are expressed by a simple visual notation allowing for an abstract description of scenarios in terms of causal relationships between responsibilities (×) (e.g., operation, action, task, function, etc.) along paths allocated to a set of components (▭). These relationships are said to be causal because they involve concurrency, partial ordering of activities, and they link causes (e.g., preconditions and triggering events) to effects (e.g., postconditions and resulting events). UCMs help in structuring and integrating scenarios (in a map-like diagram) sequentially, as alternatives (with OR-forks/joins; ⤙/⤚), or concurrently (with AND-forks/joins; ⊣/⊢).

When maps become too complex to be represented as one single UCM, a mechanism for defining and structuring sub-maps becomes necessary. Path details can be hidden in sub-diagrams called plug-in maps, contained in stubs (◇)



(a) Feature Configuration Scenario



(b) FeatureConfig Stub Plugin

Fig. 3. UCM Scenario: Configure a Feature on a Router

on a path. A plug-in map is bound (i.e., connected) to its parent map by binding the in-paths of the stub with start points (●) of the plug-in map and by binding the out-paths of the stub to end points (▮) of the plug-in map.

Figure 3(a) illustrates a UCM scenario for configuring a feature on a router setup. The feature configuration takes place when the router is in configuration mode (i.e., start point *EnterConfigMode*). The configuration steps are embedded within *ConfigFeature* stub, which has two outgoing paths (*OUT1* for successfully configuring the feature and *OUT2* for the rejection of the configuration). Figure 3(b) illustrates the plugin map of the *ConfigFeature* stub. AFter entering the configuration commands of the new feature (i.e., responsibility *ConfigureFeature*) and commit the new changes (i.e., responsibility *Commit*), the new configuration is applied (i.e., responsibility *ApplyConfig*) in case it is a valid config (i.e., condition *validConfig* part of the OR-Fork is true), otherwise the new changes are discraded (i.e., responsibility *RollbackConfig*).

One of the strengths of UCMs resides in their ability to bind responsibilities to architectural components. The default UCM component notation is generic and abstract allowing for representing software entities (e.g., objects, processes, databases, or servers) as well as non-software entities (e.g., actors or hardware). In the ITU-T standard [16], a UCM component (Figure 4) is characterized by its kind (Team, object, agent, process, actor) and its optional type (user-defined type) and may have several including components (i.e., more than one parent), therefore allowing the capture of several architectural alternatives in

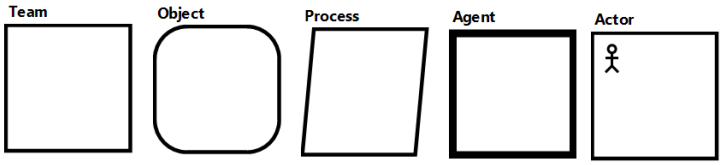


Fig. 4. UCM Components

one UCM model. A modeler may investigate various allocations of subcomponents to components and reason about trade-offs involving these alternatives.

In this research, we extend the Use Case Maps language with availability annotations. The proposed annotations are inspired from AMF concepts allowing for a smooth mapping of the resulting UCM specifications to AMF configurations. UCM generic components are used extensively to model AMF redundancy aspects, while functional constructs are used mainly to model component service instances (CSIs), modeled as UCM scenario paths.

4.2 Use Case Maps Redundancy Modeling

Error recovery focuses mainly on redundancy modeling in order to keep the system available in case of the occurrence of a failure. To accommodate the mapping to AMF configurations, we introduce five types (user-defined types) of UCM components: *node*, *application*, *serviceGroup*, *serviceUnit*, and *component*. The type is coded as a metadata attribute *Type*.

A UCM component of type *node* is annotated with the following metadata attributes:

- **NodeID**: Used to identify the node.
- **ClusterID**: Specifies the cluster to which the node belongs.

A UCM component of type *application* is annotated with the following metadata attributes:

- **ApplicationID**: Used to identify the hosted application.
- **ClusterID**: Specifies the cluster on which the application is hosted.

A UCM component of type *serviceGroup* is annotated with the following metadata attributes:

- **ServiceGroupID**: Used to identify the group to which a component belongs in a specific redundancy model. That is all components that belong to the same service group can collaborate to protected the offered services.
- **ApplicationID**: Used to specify the application the service group is implementing.
- **RedundancyModel**: Specifies the redundancy type that the service group implements. This attributes takes the following five values: *2N*, *N+M*, *N-Way*, *N-Way-Active*, and *No-Redundancy*.

A UCM component of type *serviceUnit* is annotated with the following metadata attributes:

- **ServiceUnitID**: Used to identify the service unit.
- **ServiceGroupID**: Used to identify the service group to which the service unit belongs.
- **SuActiveRole**: Lists all service instances for which the service unit is in active role.
- **SuStandbyRole**: Lists all service instances for which the service unit is in standby role.
- **SuSpareRole**: Lists all service instances for which the service unit is in spare role.

It is worth noting that *SuActiveRole*, *SuStandbyRole*, and *SuSpareRole* represent the "preferred" roles rather than static roles. At runtime, upon failure, roles may change.

A UCM component of type *component* is annotated with the following metadata attributes:

- **ComponentID**: Used to identify the component.
- **ETFComponentType**: Specifies the ETF component type.
- **ServiceUnitID**: Used to identify the service unit to which the component belongs.
- **ComponentServiceTypes**: Defines the list of service types a component can handle.

Service Instances (SIs) have no UCM graphical representation. A service instance is implicitly specified using the set of component service instances (CSIs) assigned to it. A component service instance (CSI), expressed using UCM scenarios (to model the workload), is characterized by a scenario *start point* with the following attributes:

- **CsiID**: Identifies the component service instance (CSI) that the UCM scenario implements.
- **SiID**: Identifies the service instance (SI) to which the CSI belongs.
- **ComponentActive**: Specifies the component for which the CSI is active.
- **ComponentStandby**: Specifies the component for which the CSI is standby.
- **ETFCSType**: Specifies the ETF component service type.
- **ComponentID**: Specifies the potential container of type *component*.

Figure 5 illustrates a UCM architecture describing two service units *ServiceUnit1* and *ServiceUnit2*, which are composed of components *Component1* and *Component2* respectively. The system implements one workload (CSI1), expressed as a UCM scenario path that is enclosed with the active component *Component1* (*Component2* of *ServiceUnit2* being in standby mode). The service group and its redundancy model (i.e., 2N in this case) are not shown. The characteristics of the component service instance *CSI1* are expressed as part of the start point

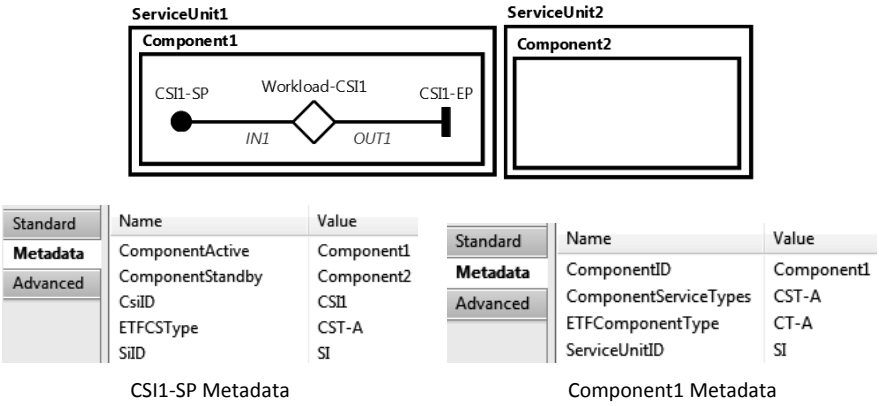


Fig. 5. Example of a Component Service Instance Representation and its corresponding Metadata

CSI1-SP metadata. The UCM plugin bound to the stub "workload-CSI1" contains the functional behavior of the component workload and it is not shown here.

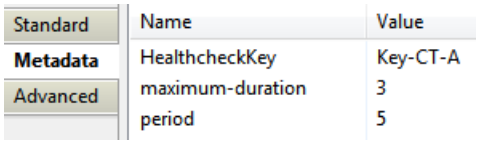
AMF [10] uses a rank-based mechanism to determine whether a service unit is active, standby, or spare. For a detailed description of the ranking system and how the service instances (SIs) are assigned to in-service service units, interested readers are referred to [10].

4.3 UCM Error Detection Modeling

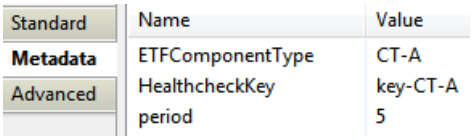
The specification of error detection mechanisms is a key factor in implementing any availability strategy. Error detection modeling involves the specification of liveness requirements (e.g., process heartbeat) and the description of potential errors. In [17] and [18], we have used the UCM comment constructor to describe error detection tactics such as *ping* and *heartbeat*. In this paper, we use metadata attributes and we introduce the concept of component healthcheck; a concept borrowed from the AMF framework. We introduce two types of component healthchecks: *framework-invoked* and *component-invoked*.

AMF supports the notion of *healthcheck* type, identified by a healthcheck key, that can be associated to a component type. A healthcheck can be invoked by the framework or by the component itself. A healthcheck configuration is composed of two attributes:

– *period*: specifies the period at which the corresponding healthcheck should be initiated. In case the healthcheck is started by the AMF framework and if a process does not respond to a given healthcheck callback (i.e., *saAmfHealthcheckCallback()*) before the start of the next healthcheck period, AMF would not trigger another callback.



(a) Healthcheck metadata associated with the UCM Specification



(b) Healthcheck metadata associated with a specific UCM component

Fig. 6. UCM-based Healthcheck

– *maximum-duration*: specifies the time-limit after which the AMF framework will report an error on the component. This attribute applies only to framework-invoked healthcheck variant.

The mapping between the UCM-based metadata attributes and AMF configurations is as follows:

– *period* is mapped to either *saAmfHealthcheckPeriod* (if the healthcheck is configured specifically for the component) or *SaAmfHctDefPeriod* (if the healthcheck is configured for the component type).
– *maximum-duration* is mapped to either *saAmfHealthcheckMaxDuration* (if the healthcheck is configured specifically for the component) or *saAmfHctDefMaxDuration* (if the healthcheck is configured for the component type).

Figure 6 illustrates two healthcheck descriptions, expressed using URN metadata feature. Both *framework-invoked* (Figure 6(a)) and *component-invoked* (Figure 6(b)) healthchecks use the "*HealthchekKey*" attribute. The component-invoked healthcheck (Figure 6(b)) specifies the type of component (e.g., using the attribute ETFComponentType) that can invoke the check. For the sake of clarity, only healthcheck related attributed are shown in Fig. 6(b).

Errors are reported to AMF by invoking the *saAmfComponentErrorReport_4()* API function that specifies, amongst others, the erronous component, the absolute time of error reporting (i.e., *errorDetectionTime*), and the recommended recovery action (i.e., *recommendedRecovery*). Section 4.4 discusses how recovery is implemented in Use Case Maps.

4.4 UCM Error Recovery Modeling

Upon failure detection, AMF would perform an automatic recovery by (1) taking a restart recovery action (restarts the erroneous component or restart all

components of the service unit), (2) performing a fail-over (e.g., Standby takes over), (3) restarting the application, or (4) resetting the cluster.

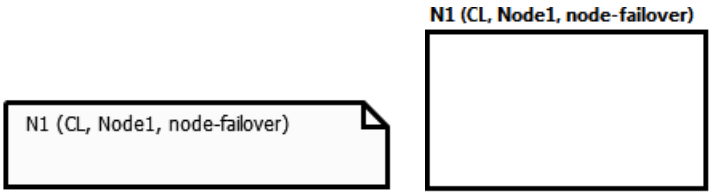
The recovery action can be encoded in component/node/application/cluster definitions using a metadata attribute *RecoveryAction* that may take the following values:

- *component-restart* and *component-failover* for UCM components of type *Component*. These two values map to *SA_AMF_COMPONENT_RESTART* and *SA_AMF_COMPONENT_FAILOVER* respectively in the AMF enumeration *SaAmfRecommendedRecoveryT*. Furthermore, a component fail-over may trigger a fail-over of the entire service unit. Such an option can be defined used the boolean attribute *SUFailOver* (mapped to AMF *saAmfSUFailover* with *SA_TRUE* and *SA_FALSE* as possible values).
- *node-failover*, *node-switchover*, and *failfast* for nodes. These three values map to AMF *SA_AMF_NODE_SWITCHOVER*, *SA_AMF_NODE_FAILOVER*, and *SA_AMF_NODE_FAILFAST* respectively. A detailed description of these three recovery mechanisms under different redundancy models can be found in [10].
- *cluster-reset* for clusters, which maps to AMF *SA_AMF_CLUSTER_RESET* enumeration value.
- *app-restart* for application components. The application should be completely terminated first by terminating all its service units. This value maps to AMF *SA_AMF_APPLICATION_RESTART* enumeration value.
- *No-recommendation*: The error report does not make any recommendation for recovery. It maps to the AMF *SA_AMF_NO_RECOMMENDATION*.

5 Discussion

Our proposed approach relies on extending the Use Case Maps language with AMF related concepts, allowing for the generation of AMF configurations at the early stages of system development process. Most the proposed extensions (e.g., application, node, service group, service unit attributes) are applied at the system architectural level and they are coded as metadata attributes (i.e., they are not represented visually in the UCM specification). Other representation options include the use of:

- **UCM comment option:** This option has been used in previous work [17] to add information about availability architectural tactics to a UCM model (see Fig. 7(a)). This option is sufficient for visualizing availability attributes in a model but does not lend itself to further analysis, because the availability information is captured in a non-formalized way. Another disadvantage of this approach is that comments cannot be attached to individual UCM model elements but only to UCM maps.
- **Construct name overloading option:** This option attaches availability attributes visually to individual UCM model elements (see Fig. 7(b)). However, similarly to the use of UCM comments, this option is informal and cannot be used in automated model analysis.



(a) UCM comments (b) Name overloading

Fig. 7. Other visual UCM-based availability representations

Contrary to the two options listed above, Our metadata approach formalizes availability attributes, making it easier to use this information in automated model analysis.

6 Illustrative Example

Figure 8 illustrates an example of a UCM system composed of one cluster of two nodes (*Node1* and *Node2*), implementing an application *App* that is composed of one service group *SG*. The relationship between the two nodes and the cluster is described using the metadata attribute *ClusterID*(Figure 9(a)). The service

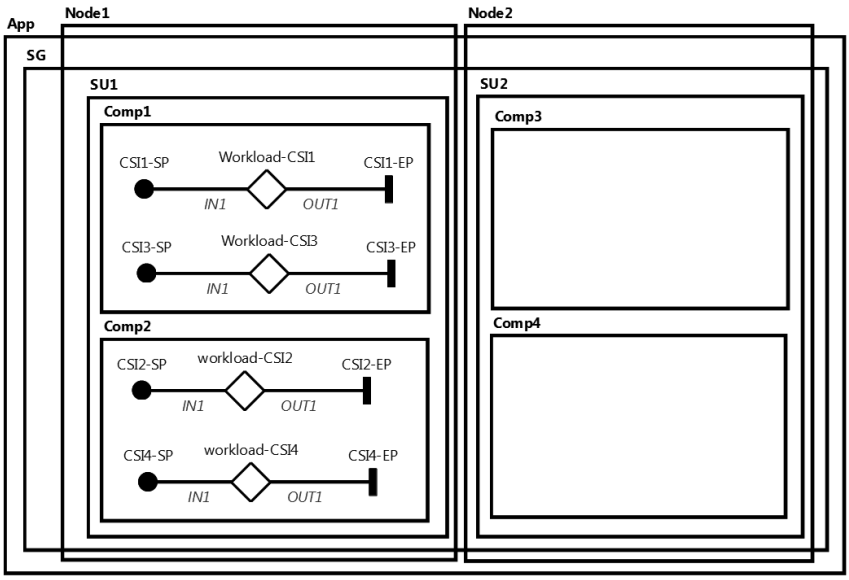
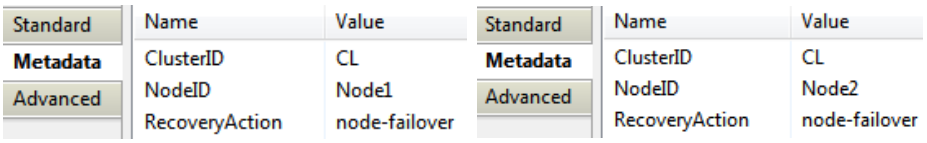
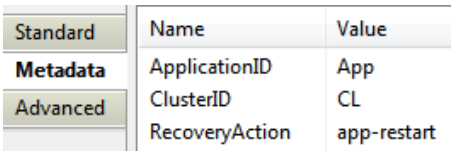


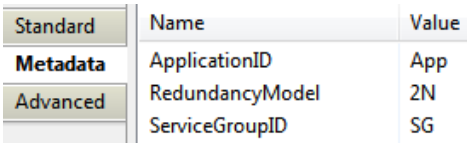
Fig. 8. A UCM Architecture with one SG having two SUs running in 2N redundancy model



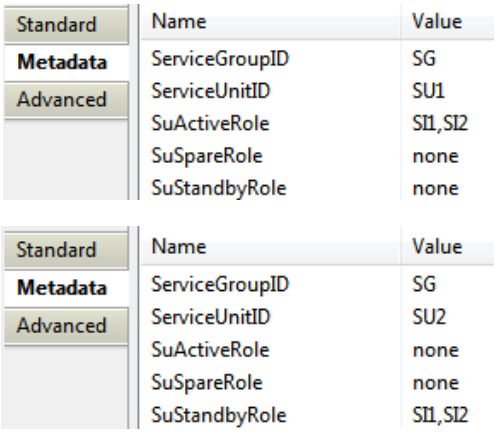
(a) Node1 and Node2 Metadata Attributes



(b) Application Metadata Attributes



(c) SG Metadata Attributes



(d) SU1 and SU2 Metadata Attributes

Fig. 9. Metadata Descriptions of the application, the participating nodes and service units

group identifier *SG* and its supported redundancy model *2N* are described using two metadata attributes *ServiceGroupID* and *RedundancyModel* (Figure 9(c)).

The service group *SG* is composed of two service units *SU1* and *SU2*. *SU1* is composed of components *Comp1* and *Comp2*, while *SU2* is composed of components *Comp3* and *Comp4*. The UCM specification defines 4 scenario paths specifying four workloads (referred to as component service instances (CSIs)). These CSIs are grouped into two service instances SI1 and SI2 (not shown graphically)

Standard	Name	Value
Metadata	ComponentID	Comp1
Advanced	ComponentServiceTypes	CST-A
	ETFComponentType	CT-A
	RecoveryAction	component-failover
	ServiceUnitID	SI1

Standard	Name	Value
Metadata	ComponentID	Comp3
Advanced	ComponentServiceTypes	CST-A
	ETFComponentType	CT-A
	RecoveryAction	component-failover
	ServiceUnitID	SI2

(a) Comp1 and Comp3 Metadata Attributes

Standard	Name	Value
Metadata	ComponentID	Comp2
Advanced	ComponentServiceTypes	CST-B
	ETFComponentType	CT-B
	RecoveryAction	component-failover
	ServiceUnitID	SI1

Standard	Name	Value
Metadata	ComponentID	Comp4
Advanced	ComponentServiceTypes	CST-B
	ETFComponentType	CT-B
	RecoveryAction	component-failover
	ServiceUnitID	SI2

(b) Comp2 and Comp4 Metadata Attributes

Standard	Name	Value
Metadata	CSIID	CSI1
Advanced	ComponentActive	Comp1
	ComponentStandby	Comp3
	SiID	SI1
	ETFCSType	CST-AA

Standard	Name	Value
Metadata	CSIID	CSI2
Advanced	ComponentActive	Comp2
	ComponentStandby	Comp4
	SiID	SI1
	ETFCSType	CST-BB

Standard	Name	Value
Metadata	CSIID	CSI3
Advanced	ComponentActive	Comp1
	ComponentStandby	Comp3
	SiID	SI2
	ETFCSType	CST-AA

Standard	Name	Value
Metadata	CSIID	CSI4
Advanced	ComponentActive	Comp2
	ComponentStandby	Comp4
	SiID	SI2
	ETFCSType	CST-BB

(c) CSI1, CSI2, CSI3, and CSI4 Metadata Attributes

Fig. 10. Metadata Descriptions of the particapting components and their corresponding component service instances

but described using the metadata attribute *SiID* (Figure 10(c)). For example, *CSI1* and *CSI2* are part of service instance *SI1*, while *CSI3* and *CSI4* are part of service instance *SI2*

The UCM shows the preferred active assignment of each component service instance. Since *SU1* has an active assignment with respect to service instances *SI1* and *SI2*, the four CSIs are described within the *SU1* components *Comp1* and *Comp2*. *SU2* has a standby assignment with respect to service instances *SI1* and *SI2*. Hence, *Comp3* and *Comp4* do not contain any CSI. Figure 10(c) shows the active (using the *ComponentActive* metadata attribute) and standby (using the *ComponentStandby* metadata attribute) assignments of the participating CSIs. For example, *CSI1* and *CSI3* are handled by component *Comp1*, while *CSI2* and *CSI4* are handled by component *Comp2*.

The specification of the healthcheck is exactly the same as in Fig. 6(a) (i.e., *framework-invoked* healthcheck), and is hence not repeated here. Recovery actions are expressed in terms of the metadata attribute *RecoveryAction*. In case of a failure, the application should be completely terminated and then started again by first terminating all of its service units and then starting them again. This is depicted in the *RecoveryAction* attribute, being equal to *app-restart*. When an error is identified as being at the node level, all service instances assigned to service units contained in the node are failed over to other nodes (i.e., *RecoveryAction* = *node-failover*). Hence, active components should also fail over to standby components (i.e., *RecoveryAction* = *component-failover*).

The metadata attributes (describing ETF types, SUs, CSIs, recovery actions, etc.) described in Fig. 9 and Fig. 10 correspond to the AMF requirements introduced in Fig. 1.

7 Conclusions and Future Work

In this work, we have extended the Use Case Maps language with Availability Management Framework (AMF) related concepts. The use of UCMs (supported by a feature-rich tool, *jUCMNav*) to describe system requirements, extended with AMF concepts, would empower analysis and validation of availability requirements at the very early stages of system development. Furthermore, we have provided a mapping between the introduced UCM-based availability requirements and AMF concepts. The resulting extensions would allow for the generation of AMF configurations from UCM specifications.

As a future work, we plan to investigate the possible integration of the UCM-based extensions (expressed with a metamodel) with a formal representation of AMF concepts, such as a UML profile for AMF.

Acknowledgment. Jameleddine Hassine would like to acknowledge the support provided by the Deanship of Scientific Research at King Fahd University of Petroleum & Minerals (KFUPM) for funding this work through project No. IN111017.

References

1. ISO/IEC/IEEE: 24765:2010(E) - Systems and software engineering – vocabulary, pp. 1–418 (2010), http://dx.doi.org/10.1109/IEEESTD.2010.5733835
2. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE Transactions on Dependable and Secure Computing 1(1), 11–33 (2004)
3. International Telecommunication Union: Recommendation E.800 (09/08) Definitions of terms related to quality of service, http://www.itu.int/rec/T-REC-E.800/en
4. Wang, D., Trivedi, K.S.: Modeling user-perceived service availability. In: Malek, M., Nett, E., Suri, N. (eds.) ISAS 2005. LNCS, vol. 3694, pp. 107–122. Springer, Heidelberg (2005), http://dx.doi.org/10.1007/11560333_10
5. Hatebur, D., Heisel, M.: A Foundation for Requirements Analysis of Dependable Software. In: Buth, B., Rabe, G., Seyfarth, T. (eds.) SAFECOMP 2009. LNCS, vol. 5775, pp. 311–325. Springer, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-04468-7_25
6. Laprie, J., Avizienis, A., Kopetz, H.: Dependability: Basic Concepts and Terminology. Springer (1991)
7. Service Availalbility^{TM} Forum: SAForum, http://www.saforum.org
8. Service Availalbility^{TM} Forum: Application Interface Specification – Software Management Framework SAI-AIS-SMF-A.01.02, http://www.saforum.org/hoa/assn16627/images/sai-ais-smf-a.01.02.pdf
9. Service Availalbility^{TM} Forum: Application Interface Specification – Overview SAI-Overview-B.05.03, http://www.saforum.org/hoa/assn16627/images/sai-overview-b.05.03.pdf
10. Service Availability^{TM} Forum: Application Interface Specification – Availability Management Framework SAI-AIS-AMF-B.04.01, http://www.saforum.org/hoa/assn16627/images/sai-ais-AMF-B.04.01.pdf
11. Salehi, P., Colombo, P., Hamou-Lhadj, A., Khendek, F.: A model driven approach for AMF configuration generation. In: Kraemer, F.A., Herrmann, P. (eds.) SAM 2010. LNCS, vol. 6598, pp. 124–143. Springer, Heidelberg (2011), http://dx.doi.org/10.1007/978-3-642-21652-7_8
12. Kanso, A., Toeroe, M., Hamou-Lhadj, A., Khendek, F.: Generating AMF configurations from software vendor constraints and user requirements. In: International Conference on Availability, Reliability and Security, ARES 2009, pp. 454–461. IEEE (2009), http://dx.doi.org/10.1109/ARES.2009.27
13. Colombo, P., Salehi, P., Khendek, F., Toeroe, M.: Bridging the gap between user requirements and configuration requirements. In: 17th International Conference on Engineering of Complex Computer Systems, pp. 13–22. IEEE (2012), http://doi.ieeecomputersociety.org/10.1109/ICECCS.2012.11
14. Kanso, A., Toeroe, M., Khendek, F., Hamou-Lhadj, A.: Automatic generation of AMF compliant configurations. In: Nanya, T., Maruyama, F., Pataricza, A., Malek, M. (eds.) ISAS 2008. LNCS, vol. 5017, pp. 155–170. Springer, Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-68129-8_13
15. Kanso, A., Khendek, F., Toeroe, M., Hamou-Lhadj, A.: Automatic configuration generation for service high availability with load balancing. Concurrency and Computation: Practice and Experience 25(2), 265–287 (2013), http://dx.doi.org/10.1002/cpe.2805

16. International Telecommunication Union: Recommendation Z.151 (10/12) User Requirements Notation (URN), http://www.itu.int/rec/T-REC-Z.151/en
17. Hassine, J.: Early Availability Requirements Modeling using Use Case Maps. In: Eighth International Conference on Information Technology — New Generations, ITNG, pp. 754–759. IEEE Computer Society (2011), http://dx.doi.org/10.1109/ITNG.2011.133
18. Hassine, J., Gherbi, A.: Exploring Early Availability Requirements Using Use Case Maps. In: Ober, I., Ober, I. (eds.) SDL 2011. LNCS, vol. 7083, pp. 54–68. Springer, Heidelberg (2011), http://dx.doi.org/10.1007/978-3-642-25264-8_6
19. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison-Wesley (2003)
20. jUCMNav v5.2.0: jUCMNav Project (tool, documentation, and meta-model), http://jucmnav.softwareengineering.ca/jucmnav

Modeling Early Availability Requirements Using Aspect-Oriented Use Case Maps

Jameleddine Hassine[1], Gunter Mussbacher[2], Edna Braun[2], and Mohammad Alhaj[3]

[1] Department of Information and Computer Science, KFUPM, Saudi Arabia
jhassine@kfupm.edu.sa
[2] School of Electrical Engineering and Computer Science, University of Ottawa, Canada
{gunterm,ebraun}@eecs.uottawa.ca
[3] Department of Systems and Computer Engineering, Carleton University, Canada
malhaj@sce.carleton.ca

Abstract. Non-functional requirements such as availability, reliability, and security are often crucial in designing and implementing distributed real-time systems. As a result, such non-functional requirements should be addressed as early as possible in the system development life-cycle. The widespread interest in dependability modeling and analysis techniques at the requirements elicitation and analysis stage provides the major motivation for this research. This paper presents a novel approach to describe high-level availability requirements using the Aspect-oriented Use Case Maps (AoUCM) language. AoUCM adds aspects-oriented concepts to the Use Case Maps (UCM) language, part of the ITU-T User Requirements Notation (URN) standard. The proposed approach relies on a mapping of availability architectural tactics to reusable AoUCM models, allowing availability tactics to be encapsulated early in the software development life-cylce. Initial tool support for the resulting availability extensions, is provided by the *jUCMNav* tool. We demonstrate the applicability of our approach using a case study of *Lawful Intercept (LI)*, an IP router feature.

1 Introduction

The Use Case Maps (UCM) language, part of the ITU-T User Requirements Notation (URN) standard [1], is a high-level visual scenario-based modeling language that has gained momentum in recent years within the software requirements community. Use Case Maps can be used to capture and integrate functional requirements in terms of causal scenarios representing behavioral aspects at a high level of abstraction, and to provide the stakeholders with guidance and reasoning about the system-wide architecture and behavior.

The Use Case Maps language, extended with aspect-oriented modeling, resulted in the Aspect-oriented Use Case Maps (AoUCM) language. AoUCM, part of the Aspect-oriented User Requirements Notation (AoURN), supports the

F. Khendek et al. (Eds.): SDL 2013, LNCS 7916, pp. 54–71, 2013.

modeling of scenario-based, crosscutting concerns during requirements activities, i.e., concerns that are difficult to encapsulate with UCM alone.

System non-functional aspects such as *availability* and *fault tolerance* are often overlooked and underestimated during the initial system design. To address this issue, the UCM language has been extended with availability features in [2] and [3]. In this research, we use the AoUCM language to model the well-known availability tactics, introduced by Bass et al. [4].

The widespread interest in dependability modeling, constitutes the major motivation of this paper. We, in particular, focus on the need to incorporate availability aspects at the very early stages of system development. This work builds upon and extends the work of Hassine and Gherbi [3], and serves the following purposes:

- It describes the availability tactics, introduced by Bass et al. [4], in a well-encapsulated way using the Aspect-oriented Use Case Maps language.
- It introduces an improved Aspect-oriented Use Case Maps language, capable of handling more concisely variations in an aspect such as availability.
- It provides a comparison between our approach and the availability modeling approach introduced in [2] and [3].
- It extends our ongoing research towards the construction of an Aspect-oriented User Requirements Notation (AoURN) framework for the description and analysis of dependability aspects in the very early stages of system development life cycle.

The remainder of this paper is organized as follows. Section 2 introduces the concept of availability and provides an overview of the existing availability description approaches. Section 3 describes the proposed AoUCM-based availability models. A brief discussion of the advantages and shortcomings of the approach is provided in Section 4. A case study of an IP-based router feature, named LI (Lawful Intercept) is presented in Section 5 demonstrating the applicability of our approach. Finally, Section 6 covers conclusions and future work.

2 Availability Requirements

Several definitions of availability have been proposed [5,6,7,8,9]. According to ISO [5], the availability of a system may be defined as the degree to which a system or a component is operational and accessible when required for use. The ITU-T recommendation E.800 [8] defines *availability*, as the ability of an item to be in a state to perform a required function at a given instant of time, or at any instant of time within a given time interval, assuming that the external resources, if required, are provided. Availability has been treated by the field of dependability [6,7,9]. Bass et al. [4] have introduced the notion of tactics as *architectural building blocks* of architectural patterns. The authors [4] have provided a comprehensive categorization of availability tactics based on whether

they address fault detection, recovery, or prevention. Figure 1 illustrates these four categories:

1. **Fault Detection** tactics are divided into
   (1) *Ping/Echo* (determines reachability and the round-trip delay through the associated network path),
   (2) *Heartbeat* (reports to system monitor when a fault is incurred), and
   (3) *Exception* (detects faults such as divide by zero, bus, and address faults).
2. **Fault Recovery-Preparation and Repair** tactics are divided into
   (1) *Voting* (A voter component decides which value to take in a redundant environment),
   (2) *Active Redundancy* (called also *hot redundancy*, refers to a configuration where all redundant spares maintain synchronous state with the active node(s)),
   (3) *Passive Redundancy* (called also *warm redundancy*, refers to a configuration where redundant spares receive periodic state updates from active node(s)), and
   (4) *Spare* (called also *cold redundancy*, refers to a configuration where the redundant spares remain out of service until a switch-over or fail-over occurs).
   It is worth noting that the application of one tactic may assume that another tactic has already been applied. For example, the application of *voting* may assume that some form of redundancy exists in the system.
3. **Fault Recovery-Reintroduction** tactics are divided into
   (1) *Shadow* (refers to operating a previously failed component in a *shadow mode* for a predefined duration of time),
   (2) *Rollback* (allows the system state to be reverted to the most recent consistent state), and
   (3) *State Resynchronization* (ensures that active and standby components have synchronized states).
4. **Fault Prevention** tactics include
   (1) *Removal from Service* (refers to placing a system component in an out-of-service state for the purpose of mitigating potential system failures),
   (2) *Transactions* (typically realized using *atomic commit protocols*), and
   (3) *Process Monitor* (monitors the health of a system).

In a closely related work, Scott and Kazman [10] have proposed a refined version of Bass et al. categorization [4]. However, their proposed classification considers tactics that are specific to inter-networking devices like switches and packet routers. Examples of such tactics include *Non-Stop Forwarding* (which maintains the proper functionning of user data plane in case of a failure) and *MPLS ping* (ensures timely ping responses in an MPLS-based network).

In this research, we adopt the more general availability tactics introduced by Bass et al. [4], as a basis for extending the Aspect-oriented Use Case Maps language [11] with availability annotations, allowing us to encapsulate the crosscutting availability concern in scenario models. These tactics have been proven in practice for a broad applicability in different industrial domains.

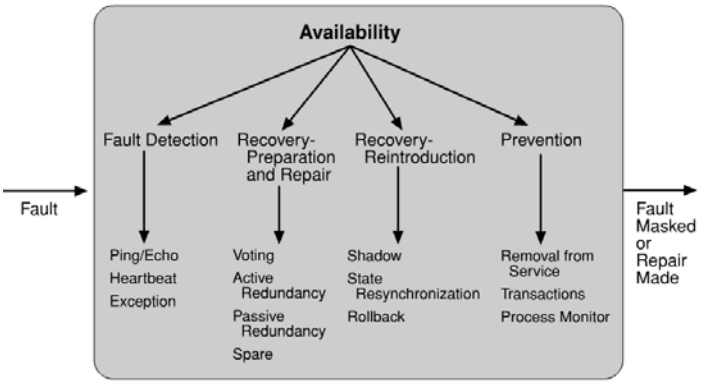


Fig. 1. Availability Tactics [4]

3 Aspect-Oriented Use Case Maps Availability Modeling

In this section, we introduce the AoUCM features and modeling elements that are relevant to our proposed availability extensions. For a complete description of the Aspect-oriented Use Case Maps language, interested readers are referred to [11,12,13,14]. AoUCM builds on the UCM language.

3.1 Use Case Maps

UCMs expressed by a simple visual notation allow for an abstract description of scenarios in terms of causal relationships between responsibilities (×, i.e., the steps within a scenario) along paths allocated to a set of components. These relationships are said to be causal because they involve concurrency, partial ordering of activities, and they link causes (e.g., preconditions and triggering events) to effects (e.g., postconditions and resulting events). UCMs help in structuring and integrating scenarios (in a map-like diagram) sequentially, as alternatives (with OR-forks/joins; ⤳/⤳), or concurrently (with AND-forks/joins; ⊣⊢/⊣⊢).

When maps become too complex to be represented as one single UCM, a mechanism for defining and structuring sub-maps becomes necessary. Path details can be hidden in sub-diagrams called plug-in maps, contained in stubs (◇) on a path. A plug-in map is bound (i.e., connected) to its parent map by binding the in-paths of the stub with start points (●) of the plug-in map and by binding the out-paths of the stub to end points (▮) of the plug-in map.

The UCM language supports path elements for failure points (⛛), which describe exceptions raised explicitly at a specific point on a path causing the cancellation of the rest of the path (it does not address other concurrent paths). In this research, we annotate responsibilities with failure metadata attributes instead of using failure points (see Section 3.3) to improve the readability of UCM and AoUCM scenario models.

One of the strengths of UCMs resides in their ability to bind responsibilities to architectural components. Several kinds ot UCM components allow system entities (▭) to be differentiated from entities of the environment (🯅). UCM component relationships depend on scenarios to provide the semantic information about their dependencies. Components are considered to be dependent if they share the same scenario execution path even though no actual/physical connections are drawn between the components.

3.2 Aspect-Oriented Use Case Maps

Aspect-oriented UCM (AoUCM) [11,12,13,14] adds three core aspect-oriented concepts *concerns*, *composition rules*, and *pointcut expressions* to UCM. A concern is a new unit of encapsulation that captures everything related to a particular idea, e.g., availability. AoUCM treats concerns as first-class modeling elements. If a concern is not crosscutting, it can be described with the standard UCM notation. However, if it is crosscutting like availability, then it is best described using the AoUCM notation. Figure 2 presents a simple example with a base concern consisting of two responsibilities *RespA* and *RespB*.

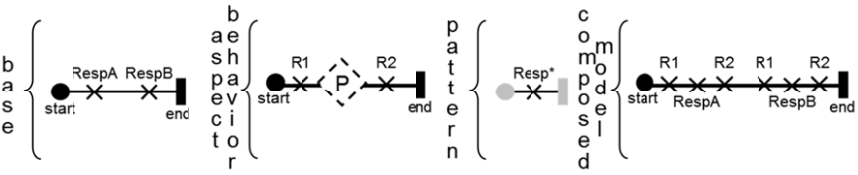


Fig. 2. AoUCM Example

Pointcut expressions are patterns that are specified by an aspect and matched in the base model. The pattern of the shown aspect matches against *Resp**, i.e., *RespA* and *RespB* in the base concern. If a match is found, the aspect is applied at the matched location in the base model. The behavior of an aspect is defined on a standard map. The only difference is that it contains a pointcut stub (🯅) that represents the locations matched by the aspect's pattern. The causal relationship between the pointcut stub and the rest of the aspect map defines the *composition rule*. In the example in Fig. 2, *R1* is added before the matched locations, because *R1* occurs before the pointcut stub, while *R2* is added after the matched locations. This results in the composed model shown in Fig. 2. An AoUCM model may also use a replacement pointcut stub (🯅) instead of a regular pointcut stub to remove any matched elements from the composed model. Furthermore, an element in the pattern may be defined as a variable ($) which allows the element to be reused in the definition of the aspect behavior.

In the following sections, we use AoUCM to describe high-level availability requirements. We adopt the availability tactics introduced by Bass et al. [4] as a basis for expressing availability requirements with AoUCM, making it possible to model availability as a properly encapsulated crosscutting concern. *Fault Recovery Preparation and Repair* as well as *Fault Recovery Reintroduction* categories (see Fig. 1) are merged to obtain what we refer to as *Fault Recovery*.

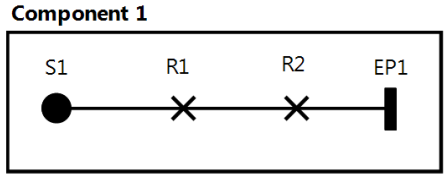
3.3 AoUCM Fault Detection Modeling

The specification of fault detection mechanisms is a key factor in implementing any availability strategy. Fault detection modeling involves the description of potential faults (i.e., *Exception tactic*) and the specification of liveness requirements using ping/echo and heartbeat [4].

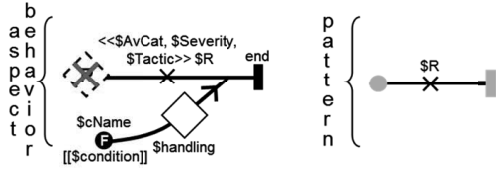
Exceptions. *Exceptions* are modeled and handled at the scenario path level. Exceptions may be associated with any responsibility along the UCM execution path. The availability requirements of a responsibility can be modeled using three metadata attributes. The metadata approach allows for a more nuanced description of availability which is not possible with only *failure points*.

1. *AvCat:* Specifies the availability category, if any, that the responsibility is implementing. In the case of exceptions, it is specified as "*FaultDetection*".
2. *Tactic:* Denotes the type of the deployed tactic. In the case of exceptions, it is specified as "*Exception*".
3. *Severity* denotes the severity of the potential fault that might occur as a result of the execution of the responsibility. Three severity levels are considered: "1" (causes the component to stop working), "2" (impacts the component operations), and "3 or higher" (minor fault, not service impacting).

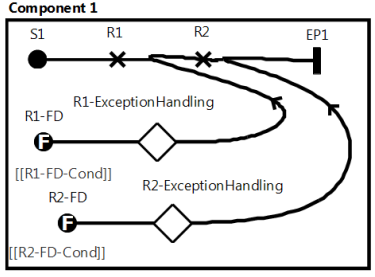
The realization of the exception tactic is assured by the definition of metadata attributes and by the existence of a related exception handling scenario. The actual handling of the exception (through the modeling of a failure scenario) is realized using the fault recovery tactic (Section 3.4). Figure 3(a) illustrates a simple UCM map with a main scenario executing in sequence two responsibilities *R1* and *R2*. Figure 3(c) shows an updated model where both responsibilities implement fault detection exception tactics specified using *AvCat* (i.e., *AvCat* = *FaultDetection*) and *Tactic* (i.e., *Tactic* = *Exception*) attributes. The metadata specifications are shown in Figure 3(d). Responsibility *R1* describes an exception with severity 1 (i.e., *Severity* = 1) while responsibility *R2* describes an exception with severity 2. Exceptions in responsibilities *R1* and *R2* are handled using two failure paths describing the recovery behavior. A failure path starts with a *failure start point* (Ⓕ) [1,11] and a guarding condition (e.g., conditions *R1-FD-Cond* and *R2-FD-Cond* in Fig. 3(c)). In this example, the actual handling of the exceptions takes place within two stubs named *R1-ExceptionHandling* and *R2-ExceptionHandling*. The corresponding plug-in maps *handlingR1* and *handlingR2* are not shown at



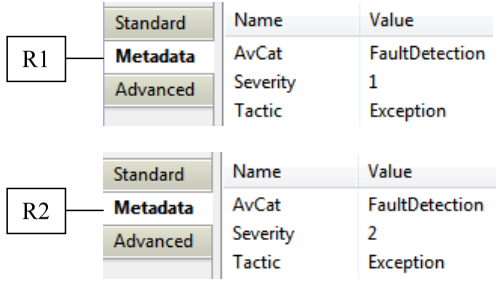
(a) Simple UCM with Two Responsibilities



(b) AoUCM Exception Handling Aspect



(c) UCM Exception Handling



(d) Metadata Attributes for R1 and R2 in (c)

Composition Condition	Assignments						Selection of Plug-in Map for
$R =	$AvCat	$Severity	$Tactic	$cName	$condition	$handling	$handling
R1	Fault-Detection	1	Exception	R1-FD	R1-FD-Cond	R1-Exception-Handling	handlingR1
R2	Fault-Detection	2	Exception	R2-FD	R2-FD-Cond	R2-Exception-Handling	handlingR2

(e) Composition Matrix for AoUCM Exception Handling Aspect

Fig. 3. UCM and AoUCM Exception Handling

this point because the details of the exception handling are irrelevant for this discussion on failure detection. The plug-in maps describe fault recovery (Section 3.4). After handling the *R1* and *R2* exceptions, the path continues explicitly with responsibility *R2* and the end point *EP1*, respectively.

As can be clearly seen from Fig. 3(c), the exceptions add significant complexity to the basic scenario in Fig. 3(a). Furthermore, great similarities can be observed for exceptions of responsibilities *R1* as well as *R2*, e.g., both require a failure path. These similarities are captured in the aspect-oriented model for the exception tactic presented in Fig. 3(b). Before explaining the details of the aspect-oriented model, it must be noted that, despite the similarities, there are also many small variations from the failure path of one responsibility to the one of another responsibility (i.e., the guarding condition is different and the actual exception handling is different; the metadata attributes of the responsibility may also be different). Nevertheless, the overall structure is the same. With conventional AoUCM, a separate AoUCM model would have to be created for each combination of variations. However, this does not scale to what is needed for availability tactics. Therefore, the Aspect-oriented Use Case Maps language has been extended with the concept of a *composition matrix* that allows variations to be specified in a concise manner. With this approach, the AoUCM model in Fig. 3(b) describes the generic reusable properties of exception handling, while the composition matrix factors out the adaptation of this generic model to its application context.

The aspect-oriented model in Fig. 3(b) describes a replacement as indicated by the replacement pointcut stub (), i.e., the model elements matched by the pattern in Fig. 3(b) are replaced with the model elements that follow the replacement pointcut stub. In this case, the pattern describes a single variable, the responsibility (*$R*). This single responsibility is replaced by itself (*$R* is shown on the path following the replacement pointcut stub which means that the matched responsibility is reused by the aspect) but with several metadata attributes added. In addition, the failure path is added which merges with the path of *$R* after the responsibility. The metadata attributes and several elements of the failure path are also specified with variables (e.g., *$AvCat* and *$condition*). However, these variables are not defined in the pattern, contrary to the *$R* responsibility. This is where the composition matrix comes into play. Any variable that is not bound by the pattern must be defined in the composition matrix as shown in Fig. 3(e).

In this example, the values of these unbound variables depend on the responsibility that is matched by the pattern, i.e., the bound variable. The first column in the composition matrix allows the composition condition to be specified. In this case, *$R* can either be *R1* or *R2*. The second set of columns specify assignments. For example, if *$R* is matched against *R1*, then the metadata variable *$AvCat* needs to be assigned the value *FaultDetection* and the condition variable *$condition* needs to be assigned the value *R1-FD-Cond*. The last column in the composition matrix allows for the specification of a specific plug-in map of the stub defined in the second row of this column (i.e., *$handling* in our case).

For example, when *R1* is matched, then the plug-in map called *handlingR1* must be used as the plug-in map for the *handling* stub. Note that, in addition to the name of the plug-in map, more detailed plug-in bindings can be specified for more complex stubs with several in-paths and out-paths. This, however, is not required for availability tactics. Similarly, regular expressions can be used in the column for the composition conditions, but this is also not required for the example in Fig. 3(b). In general, the composition matrix collects all metadata specifications required for availability in one location that are otherwise spread out over the model as shown in Fig. 3(d), hence allowing all specifications related to availability to be encapsulated in one aspect.

When the availability aspect is applied to the scenario in Fig. 3(a), the composed result is equivalent to the UCM map in Fig. 3(c) including the annotations.

Ping/Echo and Heartbeat Tactics. *Ping/Echo* and *Heartbeat* tactics can be used to determine how long it takes to detect a fault. This can be achieved using the round-trip time and the number of missed pings/heartbeats. In [2] and [3], we have reused the UCM comment constructor to describe ping and heartbeat tactics. In this paper, we use metadata attributes instead. While both alternatives allow for a global description of availability requirements (i.e., attached to the entire UCM model rather than to one specific UCM construct), using metadata provides a structural and more intuitive way of representing attributes. In AoUCM, these global descriptions are attached to the availability aspect instead of the entire UCM model, but their specification otherwise remains the same. For example, a ping initiated by component C1 that must result in an echo response from C2 received within 2ms is specified as metadata *Ping* = "*C1;C2;2*". Similarly, a heartbeat, periodic message exchange such as "I'm alive", that is sent from component C1 towards component C2 with a polling interval of 2000 ms is defined as metadata *Heartbeat* = "*C1;C2;2000*".

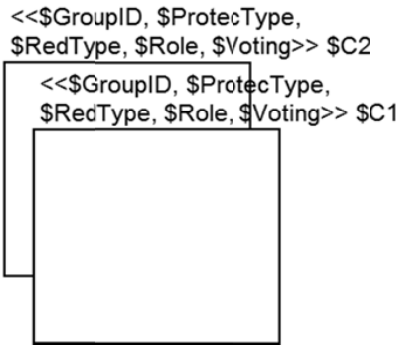
The *ping/echo* and *heartbeat* requirements can also be described using metadata attached to responsibilities (i.e., *AvCat* = "*FaultDetection*"; *Tactic* = "*Heartbeat*). Violations of *ping/echo* and *heartbeat* tactics are handled in scenario paths similar to the exception tactic. If this is the case, then an aspect similar to the AoUCM exception handling aspect is used to define the metadata and failure path for such responsibilities.

3.4 UCM Fault Recovery Modeling

Fault recovery tactics focus mainly on redundancy modeling in order to keep the system available in case of the occurrence of a failure. To model redundancy, we annotate UCM components with the following attributes:

- *GroupID*: A system may have many spatially redundant components of different types. The *GroupID* is used to identify the group to which a component belongs in a specific redundancy model.
- *Role*: Denotes whether a component is in *active* or *standby* role.
- *RedundancyType*: Specifies the redundancy type as *hot*, *warm*, or *cold*.

– *ProtectionType*: The minimal redundancy configuration is to have one active and one redundant node (commonly referred to as *1+1* redundancy). Other redundancy configurations are: *1:N* (refers to a configuration where one spare is used to protect multiple active nodes) and *M:N* (refers to a configuration where multiple spares are used to protect multiple active nodes).
– *Voting*: A boolean variable describing whether a component plays a voting role in a redundancy configuration.



(a) AoUCM Redundancy Aspect

Composition Condition	Assignments				
$C1 = ; $C2 =	$GroupID	$ProtecType	$RedType	$Role	$Voting
RP1;	G1	1+1	Hot	Active	false
RP2	G1	1+1	Hot	Stdby	false

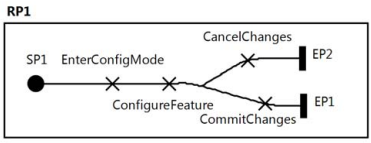
(b) AoUCM Composition Matrix

Fig. 4. AoUCM Redundancy Aspect with Composition Matrix

Since component redundancy has to be described visually and the involved redundant components share the same scenario path, an elegant way to illustrate such a configuration is to use overlapping components. Note that the current URN standard [1] does not allow overlapping components (while the *jUCMNav* tool [15] does support such a feature).

Figure 4 illustrates an example of a system with two components *RP1* and *RP2* participating in a 1+1 hot redundancy configuration. *RP1* is in active role while *RP2* is in standby role. None of these two components is taking part in a voting activity (i.e., Voting: *false*).

The presented redundancy annotations deal with the static description of component availability requirements. The operational implications of such availability requirements, in case of failure for instance, can be described using the



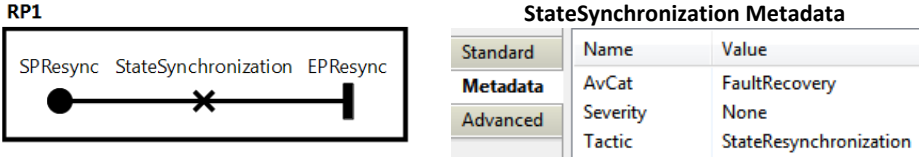
(a) Feature Configuration Example

Composition Condition	Assignments				
$C1 = ; $C2 =	$GroupID	$ProtecType	$RedType	$Role	$Voting
RP1;	G1	1+1	Hot	Active	false
RP2	G1	1+1	Hot	Stdby	false

(b) Redundancy AoUCM Composition Matrix

Composition Condition	Assignments						Selection of Plug-in Map for
$R =	$AvCat	$Severity	$Tactic	$cName	$condition	$handling	$handling
CommitChanges	Fault-Detection	2	Exception	SP	OutOf Synch	CC-Ex-Handling	Synchronize State

(c) Exception Handling AoUCM Composition Matrix



(d) Synchronize State Plug-in Map and Metadata Attributes

Fig. 5. Implementation of the State Resynchronization Tactic with AoUCM

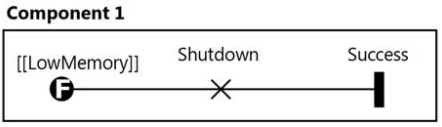
UCM scenario path. Hence, reintroduction tactics such as *Shadow*, *State Resynchronization*, and *Rollback* can be described using the metadata attributes associated to responsibilities. Typically, these tactics are used in the exception handling scenario in response to fault detection, i.e., they are used in the stub in the exception handling aspect as discussed in Section 3.3.

Figure 5 illustrates a feature configuration scenario on a dual route processor (RP) system. The configuration of a new feature may result in having the active and the standby route processors (respectively *RP1* and *RP2*) in Out-of-Sync state (e.g., configuration is not applied to the standby RP). The detection of such a situation would trigger an exception path (i.e., precondition *OutOfSych* is satisfied) and causes both RPs to synchronize again (using responsibility *StateSynchronization*). The basic scenario is defined in Fig. 5(a). Figures 5(b) and 5(c) show the composition matrices for the redundancy aspect and the exception handling aspect, respectively. The former insures that two redundant components, *RP1* and *RP2*, are defined using the redundancy aspect from Fig. 4. The latter composition matrix is for the exception handling aspect in Fig. 3(b). As stated earlier, the composition matrix defines which plug-in map to use. In this case, it is the *Sychronize State* plug-in map shown in Fig. 5(d) which may even be provided as a predefined specification of the state synchronization tactic as part

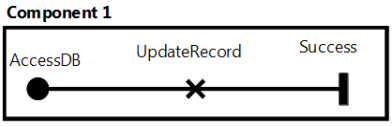
of the AoUCM availability aspect. Since this plug-in map and its responsibility *StateSynchronization* are part of the availability aspect, the required metadata attributes for the responsibility are specified directly for the responsibility.

3.5 UCM Fault Prevention Modeling

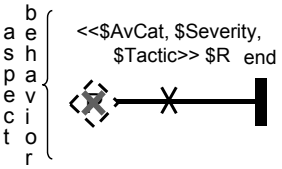
Annotations presented in Sections 3.3 can be used to accommodate this category. Indeed, responsibilities can be annotated with availability metadata attributes specifying a removal from service property, transactions properties, and process monitoring properties. For example, Fig. 6(a) illustrates a UCM scenario describing the placement of a component in an out-of-service state by shutting it down (i.e., responsibility *Shutdown*) to prevent potential system failures in case the component is running low on memory, and Fig. 6(b) provides a scenario of updating a database record using a two-phase-commit type of transaction (a.k.a. *2PC*). Failing to ensure the two phase commit requirement, would trigger an implicit rollback to undo the record update (not shown in the figure). The aspect in Fig. 6(c) uses the same pattern as specified in Fig. 3(b), but the aspect behavior is simpler as it only replaces the matched responsibility with the same but annotated responsibility. The composition matrix of the fault prevention aspect in Fig. 6(c) ensures that the responsibilities are annotated accordingly.



(a) Removal From Service Tactic



(b) Transactions Tactic



Composition Condition	Assignments		
$R =	$AvCat	$Severity	$Tactic
Shutdown	FaultPrevention	None	RemovalFromService
Update Record	FaultPrevention	None	Transactions

(c) AoUCM Fault Prevention Aspect with Composition Matrix

Fig. 6. AoUCM Fault Prevention Modeling

4 Discussion

Our proposed approach relies on a mapping of availability architectural tactics proposed by Bass et al. [4] to generic, reusable AoUCM models, which allows the tactics to be encapsulated at the early phases of the software development lifecycle. In previous work [2], the use of comments is suggested to add information

about the same availability tactics to a UCM model [3]. This paper uses metadata to describe availability tactics with an aspect-oriented approach but the same metadata could also be added directly to a conventional UCM model. These three options (comments, UCM metadata, AoUCM metadata) share the same goal of modeling the availability tactics at the early phases of development, which allows earlier detection of design errors and helps modelers to select between different design alternatives. We briefly compare the advantages and shortcomings of these three approaches.

1. **Comment-based option:** This option is sufficient for visualizing availability tactics in a model but does not lend itself to further analysis, because the availability information is captured in a non-formalized way. Another disadvantage of this approach is that comments cannot be attached to individual UCM model elements but only to UCM maps. The availability tactics, however, require information to be attached to individual UCM model elements. This option is hence not further considered.
2. **UCM metadata-based option:** This option formalizes availability information in metadata, making it easier to use this information in automated model analysis. However, availability information is not encapsulated well in one location in the model but rather distributed over the whole model, making maintenance, reuse, and evolution of the availability information more difficult.
3. **AoUCM metadata-based option:** This option also uses metadata but localizes the availability information in one availability aspect (which may be sub-divided into smaller availability aspects dealing specifically with exception handling, fault prevention, etc.). Previous comparisons of UCM and AoUCM models [13] indicate that AoUCM models exhibits better modularity, reusability, and maintainability than UCM models. Essentially, a larger vocabulary size (because aspects are introduced) is trade-off against better separation of concerns, less coupling, and stronger cohesion. These results also apply to the availability aspect. The effects are less pronounced for simple metadata assignments which could be maintained through additional tool management features without the need for aspect-oriented techniques. However, the availability tactics of exception detection and handling can greatly benefit from an aspect-oriented approach because of the high number of model elements affected by these tactics. Hence, the model can be simplified significantly. The addition of *composition matrices* further improves the handling of small variations in an aspect such as availability. The common, generic, reusable part of an availability tactic can be captured concisely in a rather simple AoUCM aspect, while the variations are factored out into the composition matrices. A disadvantage of the AoUCM metadata-based option is that the AoUCM model is more fragmented and an automated composition is required to merge the availability aspect into the system model. However, these disadvantages can be alleviated by available tool support with the jUCMNav editor [15] which helps modelers with the navigation through AoUCM models and the composition of AoUCM models.

5 Case Study: Lawful Intercept (LI) Feature

In this section, we illustrate our proposed approach using a case study of a Lawful Intercept (LI) feature, running on a Cisco CRS-1 router[1]. LI allows service providers to meet the requirements of law enforcement agencies (e.g., state and federal police, intelligence agencies, and independent commissions against corruption) to provide authorized interception of VoIP and data traffic at content IAP (intercept access point) routers.

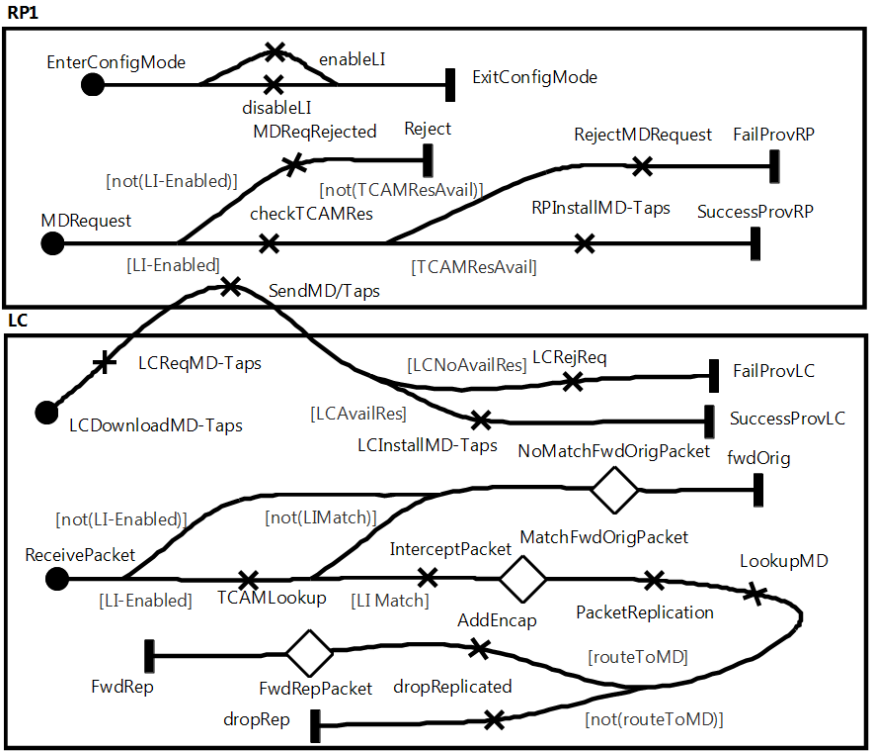
A minimal CRS-1 router architecture consists of one or many route processors (RP) cards (that provide route processing, alarm, fan, and power supply controller function), one or many ingress line cards (that process incoming traffic), one or many egress line cards (that process outgoing traffic), and a switch fabric (receives user data from ingress cards and performs the switching necessary to route the data to the appropriate egress cards). Since LI is an ingress feature (i.e., applied on ingress line cards), its description is abstracted from the switch fabric and the egress line cards. LI is described using an AoUCM scenario model (Figure 7) bound to an architecture composed of two Route Processors (RP) in a hot redundancy mode (RP1 is active role while RP2 is the standby) and one ingress line card (LC). The specification of the redundancy is exactly the same as in Fig. 5, and is hence not repeated here.

In a typical operation, a lawfully authorized intercept request is provisioned by the MD (Mediation Device) on the content IAP (Intercept Access Point), which is the device within the network used to intercept the targeted information. The IAP is responsible for identifying the IP traffic of interest and forwarding it to the MD, while remaining undetectable by the intercept subject. In a typical operation, the router (e.g., Cisco CRS-1) is the content IAP.

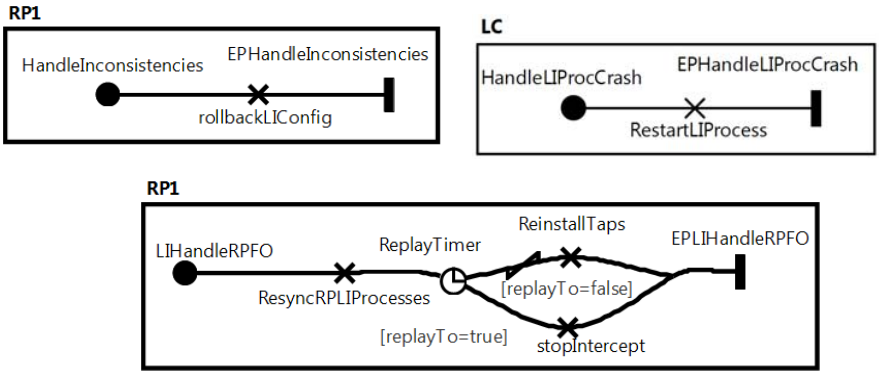
Figure 7(a) provides a high level description of basic LI scenarios. The Mediation Device (MD) crafts an interception request based on the content to be collected and sends it to the router. Upon reception (start point *MDRequest*), the MD request may be rejected (*MDReqRejected*) in case LI is disabled on the router, otherwise a check whether TCAM (Ternary content-addressable memory) resources are available (*checkTCAMRes*) is performed. Requested MD/Tap entries are programmed (*RPInstallMD-Taps*) when TCM resources are available, and they are rejected (*RejectMDRequest*) otherwise. Enabling and disabling LI may cause unexpected configuration inconsistencies. This exception is handled with the exception handling aspect introduced earlier. The composition matrix entries for *enableLI* and *disableLI* state that the *Rollback-LIConfiguration* plug-in map handles the exception by reverting back to the last consistent configuration, i.e., a rollback occurs as described by responsibility *rollbackLIConfig* and its composition matrix entry in Fig. 8.

The composition matrix also shows that there is a risk of an RP failover (RPFO), when installing new MD/taps in the TCAM (*RPInstallMD-Taps*) and the exception handling is defined on the *Resynchronize-Process-RPLI* plug-in

[1] `www.cisco.com/en/US/prod/collateral/routers/ps5763/prod_brochure0900aecd800f8118.eps`



(a) Lawful Intercept High Level Description



(b) Rollback-LIConfiguration, Restart-Process-LI, and Resynchronize-Process-RPLI Plug-in Maps

Fig. 7. Lawful Intercept AoUCM Modeling

Composition Condition	Assignments						Selection of Plug-in Map for
$R =	$AvCat	$Severity	$Tactic	$cName	$condition	$handling	$handling
enableLI	Fault-Detection	1	Exception	Handle-Inconsistencies	Config-Inconsistencies-eLI	CI-Ex-Handling	Rollback-LIConfiguration
disableLI	Fault-Detection	1	Exception	Handle-Inconsistencies	Config-Inconsistencies-dLI	CI-Ex-Handling	Rollback-LIConfiguration
RPInstallMD-Taps	Fault-Detection	1	Exception	LIHandle-RPFO	RPFO	RPFO-Ex-Handling	Resynchronize-Process-RPLI
LCInstallMD-Taps	Fault-Detection	1	Exception	HandleLI-ProcCrash	LIProcessCrash	LIPC-Ex-Handling	Restart-Process-LI

Composition Condition	Assignments		
$R =	$AvCat	$Severity	$Tactic
rollbackLIConfig or ResyncRPLIProcesses or RestartLIProcess	FaultRecovery	None	Rollback

Fig. 8. Lawful Intercept Exception Handling and Fault Recovery AoUCM Composition Matrices

map. At any point in time, the MD has the responsibility to detect the loss of the taps. LI uses a replay timer, an internal timeout that provides enough time for MD to re-provision tap entries while maintaining existing tap flows. It resets and starts on the active RP when an RPFO takes place. After replay timeout (the zigzag path leaving the timer in Fig. 7(b)), interception stops on taps that are not re-provisioned.

In the ingress LC, we distinguish two scenarios. The first one deals with downloading MD and Tap entries from the RP and installing them (*LCInstallMD-Taps*) on the LC if resources are available, otherwise the download request is rejected (*LCRejReq*). The second scenario deals with the interception of traffic. Packets received while LI is disabled are forwarded to their destination (stub *NoMatchFwdOrigPacket*). Packets matching Tap entries are intercepted (*InterceptPacket*), then forwarded to their original destinations (stub *MatchFwdOrigPacket*), and replicated (*PacketReplication*). If there is a valid route to MD (*LookupMD*), packets will be encapsulated (*AddEncap*) and sent to the MD (stub *FwdRepPacket*), otherwise the replicated packets are dropped (*dropReplicated*).

The LI process may crash while installing new MDs/Taps. This is again described with the exception handling aspect (Figure 7(b)). The composition matrix entry for *LCInstallMD-Taps* states that the exception handling is defined on the *Restart-Process-LI* plug-in map which restarts the process (*RestartLIProcess*).

6 Conclusions and Future Work

In this work, we have incorporated availability information at the very early stages of system development with the help of an aspect-oriented approach. To this end, we have extended the Aspect-oriented Use Case Maps (AoUCM) language (which is based on the ITU-T User Requirements Notation (URN)

standard) with availability metadata covering the well known availability tactics proposed by Bass et al. [4]. AoUCM adds aspect-oriented concepts to UCM which allow the availability architectural tactics to be encapsulated at the early phases of system development. Availability tactics typically need to be applied to numerous locations in a system. A characteristic of availability tactics is that each time a slightly different availability tactic needs to be applied. Therefore, we have extended the AoUCM language with the notion of *composition matrices* which enables these small variations of the availability tactics to be specified concisely. We envision that the proposed composition matrix is useful for other concerns in addition to availability. For future work, we plan to investigate which concerns could benefit from composition matrices. We also aim to study how to map AoUCM availability concepts into the Service Availability Forum's Availability Management Framework configurations. Our goal is to allow for the early reasoning about availability aspects and promote the portability and the reusability of the developed systems across different platforms.

References

1. International Telecommunication Union: Recommendation Z.151 (10/12), User Requirements Notation (URN) – language definition (2012), http://www.itu.int/rec/T-REC-Z.151/en
2. Hassine, J.: Early Availability Requirements Modeling using Use Case Maps. In: 8th International Conference on Information Technology – New Generations, ITNG 2011, pp. 754–759. IEEE Computer Society (2011)
3. Hassine, J., Gherbi, A.: Exploring Early Availability Requirements Using Use Case Maps. In: Ober, I., Ober, I. (eds.) SDL 2011. LNCS, vol. 7083, pp. 54–68. Springer, Heidelberg (2011)
4. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison-Wesley (2003)
5. ISO/IEC/IEEE: 24765:2010(E) - Systems and software engineering – vocabulary, pp. 1–418 (2010), http://dx.doi.org/10.1109/IEEESTD.2010.5733835
6. Avizienis, A., et al.: Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE Transactions on Dependable and Secure Computing 1(1), 11–33 (2004)
7. Hatebur, D., Heisel, M.: A Foundation for Requirements Analysis of Dependable Software. In: Buth, B., Rabe, G., Seyfarth, T. (eds.) SAFECOMP 2009. LNCS, vol. 5775, pp. 311–325. Springer, Heidelberg (2009)
8. International Telecommunication Union: Recommendation E.800 (09/08) Definitions of terms related to quality of service (2008), http://www.itu.int/rec/T-REC-E.800/en
9. Laprie, J., Avizienis, A., Kopetz, H.: Dependability: Basic Concepts and Terminology. Springer (1991)
10. Scott, J., Kazman, R.: Realizing and refining architectural tactics – Availability. Carnegie Mellon University – Software Engineering Institute (2009), http://www.sei.cmu.edu/library/abstracts/reports/09tr006.cfm
11. Mussbacher, G.: Aspect-oriented user requirements notation. Ph.D. thesis, University of Ottawa (2010), http://lotos.csi.uottawa.ca/ucm/pub/UCM/VirLibGunterPhDThesis/Aspect-Oriented_User_Requirements_Notation.pdf

12. Mussbacher, G., Amyot, D.: Extending the user requirements notation with aspect-oriented concepts. In: Reed, R., Bilgic, A., Gotzhein, R. (eds.) SDL 2009. LNCS, vol. 5719, pp. 115–132. Springer, Heidelberg (2009)
13. Mussbacher, G., Amyot, D., Araújo, J., Moreira, A.: Requirements Modeling with the Aspect-oriented User Requirements Notation (AoURN): A Case Study. In: Katz, S., Mezini, M., Kienzle, J. (eds.) Transactions on AOSD VII. LNCS, vol. 6210, pp. 23–68. Springer, Heidelberg (2010), http://dx.doi.org/10.1007/978-3-642-16086-8_2
14. Mussbacher, G., et al.: AoURN-based modeling and analysis of software product lines. Software Quality Journal 20(3-4), 645–687 (2012), http://dx.doi.org/10.1007/s11219-011-9153-8
15. jUCMNav v5.2.0: jUCMNav Project (tool, documentation, and meta-model) (2013), http://jucmnav.softwareengineering.ca/jucmnav

Model-Driven Engineering for Trusted Embedded Systems Based on Security and Dependability Patterns

Brahim Hamid[1], Jacob Geisel[1], Adel Ziani[1], Jean-Michel Bruel[1], and Jon Perez[2]

[1] IRIT, University of Toulouse
118 Route de Narbonne, 31062 Toulouse Cedex 9, France
{hamid,geisel,ziani,bruel}@irit.fr
[2] Ikerlan, Mandragon, Spain
jmperez@ikerlan.es

Abstract. Nowadays, many practitioners express their worries about current software engineering practices. New recommendations should be considered to ground software engineering on two pillars: solid theory and proven principles. We took the second pillar towards software engineering for embedded system applications, focusing on the problem of integrating Security and Dependability (S&D) by design to foster reuse. The framework and the methodology we propose associate the model-driven paradigm and a model-based repository of S&D patterns to support the design of trusted Resource Constrained Embedded System (RCES) applications for multiple domains (e.g., railway, metrology, automotive). The approach has been successfully evaluated by the TERESA project external reviewers as well as internally by the Ikerlan Research Center for the railway domain.

Keywords: Resource Constrained Embedded Systems, Security, Dependability, Repository, Pattern, Metamodel, Model Driven Engineering.

1 Introduction

The software of embedded systems [1] is not conventional software that can be built using usual paradigms. In particular, the development of resource constrained embedded systems (RCES) addresses constraints regarding memory, computational processing power and/or limited energy. Non-functional requirements such as Security and Dependability (S&D) [2] become more important as well as more difficult to achieve. The integration of S&D features requires the availability of both application domain specific knowledge and S&D expertise at the same time. In fact, capturing and providing this expertise by the way of S&D *patterns* can support embedded systems development.

In our previous work [3], we studied pattern modeling frameworks [4,5] and we proposed methods to model security and dependability aspects in patterns

F. Khendek et al. (Eds.): SDL 2013, LNCS 7916, pp. 72–90, 2013.

and to validate whether these still hold in RCES (Resource Constrained Embedded Systems) after pattern application. The question remains at which stage of the development process to integrate S&D patterns. In our work, we promote a new discipline for system engineering using a pattern as its first class citizen: Pattern-based System Engineering (PBSE). PBSE addresses challenges similar to those studied in software engineering. Closely related to our vision is the Component Based Software Engineering (CBSE) [6]. Therefore, PBSE focuses on patterns and from this viewpoint addresses two kind of processes: the process of *pattern development* and *system development with patterns*. The main concern of the first process is designing patterns for reuse and the second one is finding the adequate patterns and evaluating them with regard the system-under-development's requirements.

In this paper, we propose a methodology based on Model-Driven Engineering (MDE) and a model-based repository of S&D patterns for security and dependability engineering. At the core of the methodology is a set of Domain Specific Modeling Languages (DSML) [7] that allow modeling S&D patterns and repository structure. Such an approach, relying on an MDE too -suite supporting the methodology and thus in our context supporting automated model-based repository building and access in industry[1]. We discuss the benefits, such as reduced modeling effort and improved readability, achieved when applying the methodology to an industrial case study where we have used the modeling language to model the repository of S&D patterns for the domain of railway applications.

The rest of this paper is organized as follows. In Sect. 2, we present a review of the most important related work. In Sect. 3, we present the proposed methodology. Section 4 details the specification languages proposed in the context of the methodology. In Sect. 5, we introduce the tool-chain supporting the methodology. Then, in Sect. 6, we illustrate the methodology through the example of a railway application. Section 7 describes a first feedback on the methodology we propose. Finally, Sect. 8 concludes the paper with an outlook on future work.

2 Related Work

In developing software applications with security and dependability support, the use of patterns should lead to well structured applications. In [8] a hybrid set of patterns is used in the development of fault-tolerant software applications. These patterns are based on classical fault tolerant strategies such as N-Version programming and recovery block, consensus, voting. Extending this framework, [9] proposed a framework for the development of dependable software systems based on a pattern approach. They reused proven fault tolerance techniques in the form of fault tolerance patterns. The pattern specification consists of a service-based architectural design and deployment restrictions in form of UML deployment diagrams for the different architectural services.

In [10], a group of seven patterns is presented as a security framework for building applications. [11] introduced the concept of a security pattern system

[1] The approach is evaluated in the context of the TERESA project (`http://www.teresa-project.org/`).

as a set of patterns with a linkage between them and described how security patterns contribute to the security engineering process. [12] presented an extension of UML, called UMLsec, that enables to express security relevant information within diagrams in a system specification. UMLsec is defined in form of a UML profile using UML extension mechanisms, allowing the specification of security patterns and the integration of these patterns into system development.

Regarding the analysis aspects, [13] used the concept of security problem frames as analysis patterns for security problems and associated solution approaches. These frames are also grouped in a pattern system with a list of their dependencies. The analysis activities using these patterns are described with a highlight on how the solution may be set with a focus on the privacy requirement anonymity. For the software architecture, [14] presented an evaluation of security patterns in the context of secure software architectures. The evaluation is based on the existing methods for secure software development, such as guidelines, and on threat categories.

Another important issue is the identification of security patterns. [15] proposed a new specification template inspired on secure system development needs. The template is augmented with UML notations for the solution and with formal artifacts for the requirements properties.

In addition to the above, S&D patterns are studied as precise specifications of validated S&D mechanisms. [5] explains how this can be achieved by using a library of precisely described and formally verified security and dependability (S&D) solutions as mechanisms, while [16] reports an empirical experience, about the adopting and eliciting of these S&D patterns in the Air Traffic Management (ATM) domain. The results are of interest, mainly the use of patterns as a guidance to structure the analysis of operational aspects when they are used at the design stage. Recently [17] presented an overview and new directions on how security patterns are used in the whole aspects of software systems from domain analysis to the infrastructures.

3 Pattern-Based Security Engineering Methodology

We now present an overview of our modeling building processes as activity diagrams. In this description, we will give the main keys to understand why our process is based on a generic, incremental and a constructive approach. Additional and detailed information will be provided during the implementation of the related design environment. Moreover, we provide a set of definitions and concepts that might prove useful in understanding our approach. Then, we detail the description of the integrated process used for the development of the Safe4Rail application in Sect. 6.

3.1 Definitions

Adapting the definition of pattern language given by Christopher Alexander [18], we define the following:

Definition 1 (Modeling Artifact Language.). *A modeling artifact language is a collection of modeling artifacts forming a vocabulary. Such a collection may be skillfully woven together into a cohesive "whole" that reveals the inherent structures and relationships of its constituent parts toward fulfilling a shared objective.*

Definition 2 (Security Pattern.). *A security pattern describes a particular recurring security problem that arises in specific contexts and presents a well-proven generic scheme for its solution [11].*

Definition 3 (Security Pattern Language.). *We define a security pattern language as a modeling artifact language where its constituent parts are security patterns and their relationships.*

Definition 4 (Instantiation.). *An instantiation activity takes a pattern and its related artifacts from the repository and adds it to the end-developer environment. This task enables the pattern to be used while modeling.*

The *Instantiation* activity is composed of the following steps:

1. Define needs in terms of properties and/or keywords,
2. Search for patterns in the repository,
3. Select the appropriate pattern from those proposed by the repository,
4. Import the selection into the development environment using model transformation techniques.

Definition 5 (Integration.). *An integration activity happens within the development environment when a pattern and its related artifacts are introduced into an application design. Some development environments may come with native support for the integration.*

3.2 Development of Reusable Artifacts

The pattern development process supports a number of features including pattern design, validation, interaction with a verification framework, deposit to and retrieval and from the repository.

The process root, as shown in Fig. 1, indicates the start of the creation of a pattern (A1). It contains some initialization actions to define the pattern attributes (e.g, name, author, date,...). The next activity is the modeling of the pattern artifacts (A2) collecting data interacting with (1) *Domain knowledge and expertise* providing an informal pattern description and (2) the model-based Repository to refer to existing patterns. During this activity the pattern artifacts were built conforming to the pattern modeling language. An activity is added at this point to check the design conformity of the pattern (A3). The next activity (A4) deals with the pattern validation. It supports the formal validation of a pattern using an external process [3]. The result is a set of validation artifacts. At this point, the pattern designer may generate documentation (A6). If the pattern has been

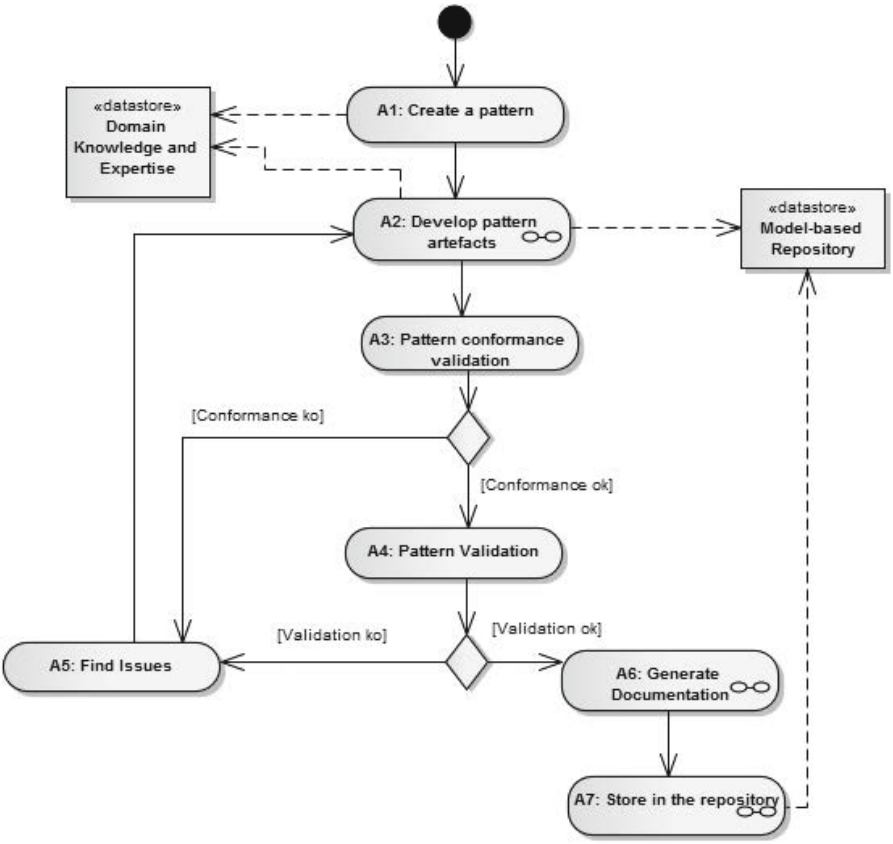


Fig. 1. Pattern development process

correctly defined (i.e. conforms to the pattern modeling language and is formally validated) the pattern is ready for the publication to the model-based repository (A7). Otherwise, we can find the issues and re-build the pattern (A5) by correcting or completing its relevant constructs.

3.3 Repository Designer View Point

The goal of this process is to organize the repository content, in our case patterns, to give them a structure of a set of pattern languages for application domains [19]. As visualized in the top part of Fig. 2, each pattern from a certain application domain is studied in order to identify its relationships with the other patterns belonging to the same application domain with respect to the engineering process' activity in which it is consumed (see the bottom part of Fig. 2).

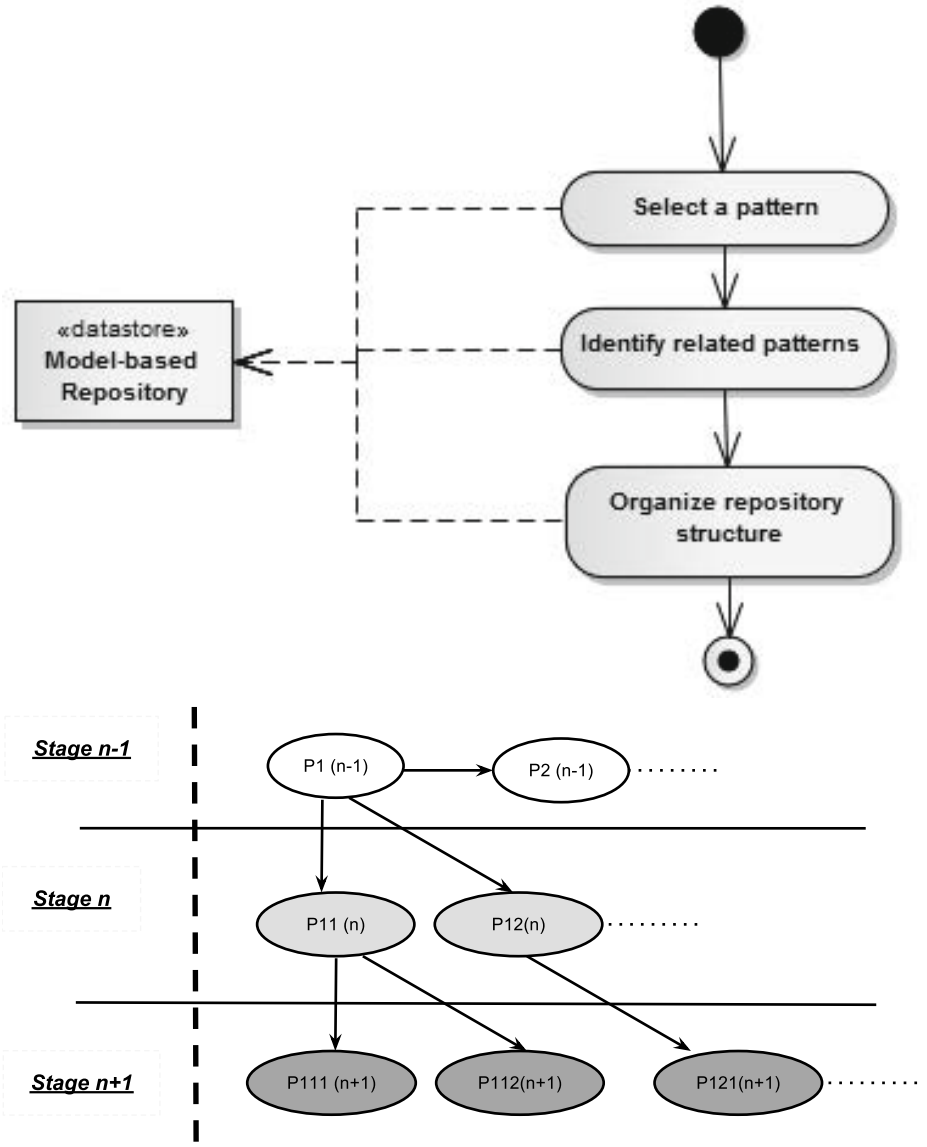


Fig. 2. Repository Process

3.4 Reuse of Existing Artifacts

Once the repository[2] is available, it serves an underlying trust engineering process. In the process model visualized in Fig. 3, the developer starts by system

[2] The repository system populated with S&D Patterns.

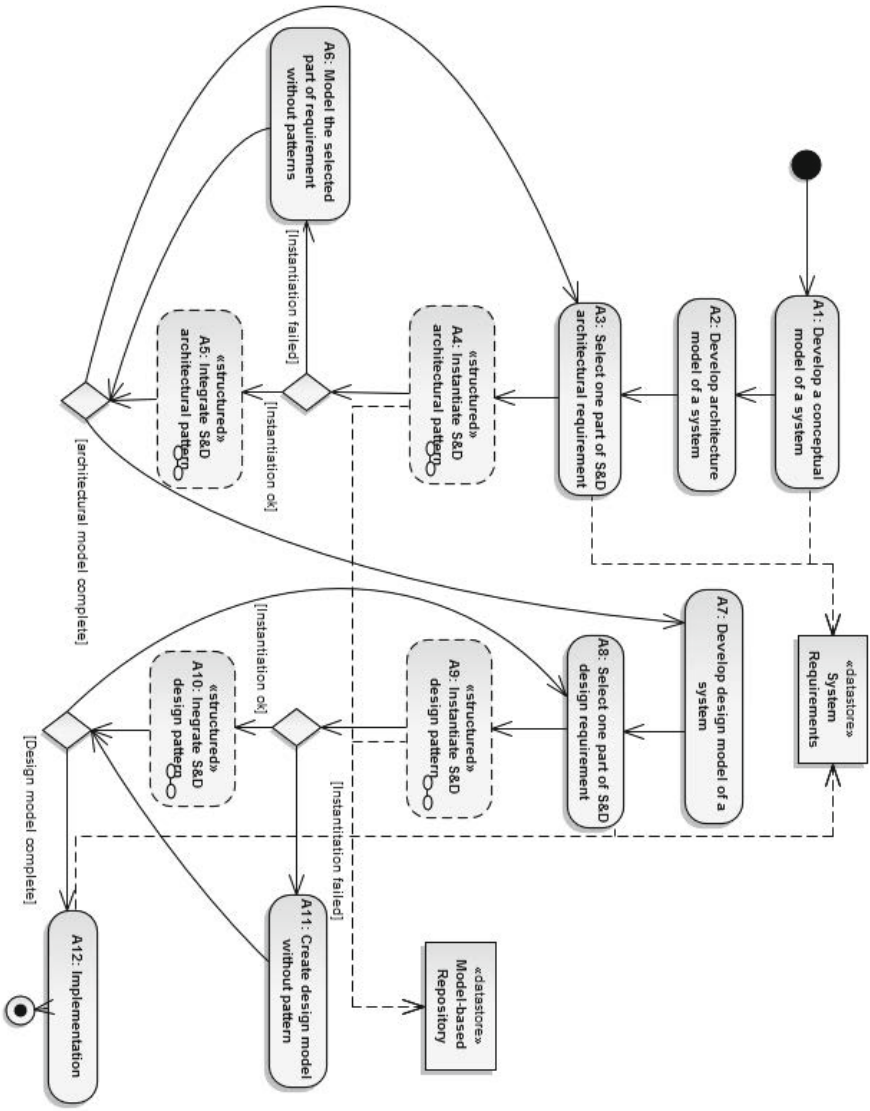


Fig. 3. The S&D Pattern-based Development Process

specification (A1) fulfilling the requirements. In a traditional approach (non pattern-based approach) the developer would continue with the architecture design, module design, implementation and test. In our vision, instead of following this phase and defining new modeling artifacts, that usually are time and effort consuming, as well as error prone, the system developer merely needs to select appropriate patterns from the repository and integrate them in the system under development.

For each phase, the system developer executes the search/select from the repository to instantiate patterns in its modeling environment (A4 and A9) and integrates them in its models (A5 and A10) following an incremental process. The model specified in a certain activity A$n-1$ is then used in activity An. In the same way, for a certain development stage n, the patterns identified previously in stage (phase) $n-1$ will help during the selection activity of a current phase. Moreover, the system developer can use the pattern design process, introduced previously, to develop their own solutions when the repository fails to deliver appropriate patterns at this stage. It is important to remark that the software designer does not necessarily need to use one of the artifacts stored in the repository previously included. He can define custom software architecture for some patterns (components), and avoid using the repository facilities (A6 and A11).

4 Specification Languages (DSLs)

In this section we present the specification languages to support the PBSE methodology: repository structure specification language (SARM) and pattern modeling language (SEPM).

4.1 Repository Structure Specification Language

A repository is a data structure that stores artifacts and that allows the user to publish and to select them for reuse and to share expertise. The specification of the structure of the repository is based on the organization of its content and the way it interacts with other engineering processes. The analysis of these requirements allows us to identify two main parts: the first one is dedicated to store and manage data in the form of *Compartments*, the second one is about the *Interfaces* in order to publish and to retrieve patterns and models.

The principal classes of the System and software Artifact Repository Metamodel (SARM) are described with Ecore notations in Fig. 4. The following part depicts more detailed the meaning of the principal concepts used to structure the repository:

- SarmRepository. Is the core element used to define a repository.
- SeArtifact. We define a modeling artifact as a formalized piece of knowledge for understanding and communicating ideas produced and/or consumed during certain activities of system engineering processes. The modeling artifact may be classified in accordance with engineering processes levels.

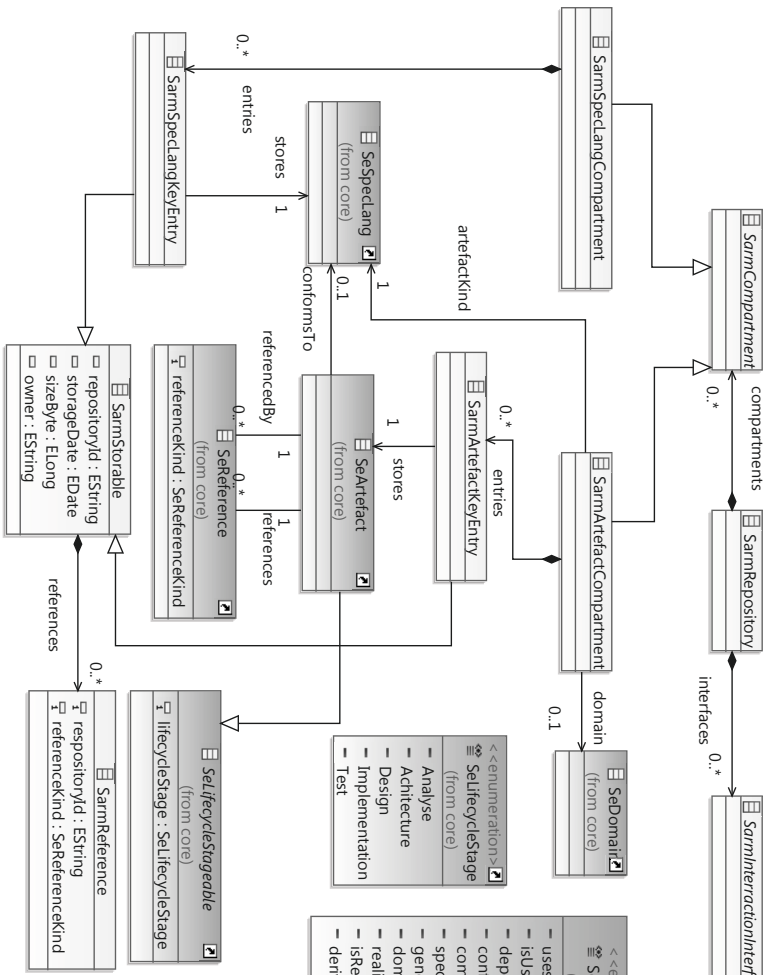


Fig. 4. Repository Specification Language -Overview

An SeLifecycleStage defines an enumeration to the development life-cycle stage in which the artifact will be used. In our study, we focus on S&D pattern models. In this context, we use the pattern classification of Riehle and Buschmann [4,19].

- SarmCompartment. Is used for the categorization of the stored artifacts. We have identified two main kinds of compartments: (1) SarmSpecLangCompartment to store the specification languages (SeSpecLang) of the modeling artefacts (SEPM), and (2) SarmArtefactCompartment to store the modeling artefacts (S&D pattern models).
- SeReference. This link will be used to specify the relation between patterns with regard to domain and software life-cycle stage in the form of a pattern language. For instance, a pattern at a certain software life-cycle stage *uses* another pattern at the same/or at different software life-cycle stage. The enumeration SeReferenceKind contains examples of these links.
- SarmStorable. Is used to define a set of characteristics of the modeling artifacts, mainly those related to its storage. We can define: *RepositoryID*, *StorageDate*, *SizeByte*, etc.... In order to keep the structure of pattern language as the set of patterns and their links for a certain domain, the concept SarmStorable includes a list of references (SarmReference).

4.2 Pattern Specification Language (SEPM)

The System and software Pattern Metamodel (SEPM), as depicted in Fig. 5, is a metamodel defining a new formalism for describing patterns. Note, however, that our proposition is inspired from GoF [20] specification, which we deeply refined in order to fit with the non-functional needs. The principal classes of the metamodel are described with Ecore notations in Fig. 5. In the following, we detail the meaning of principal concepts used to edit a pattern.

- SepmPattern. This block represents a modular part of a system representing a solution of a recurrent problem. It specializes the conceptual SeArtifact. An SepmPattern is defined by its behavior and by its provided and required interfaces. An SepmPattern may be manifested by one or more artifacts, and in turn, that artifact may be deployed to its execution environment. The SepmPattern has attributes [20] to describe the related recurring design problem that arises in specific design contexts.
- SepmInternalStructure. Constitutes the implementation of the solution proposed by the pattern. Thus the *InternalStructure* can be considered as a white box which exposes the details of the pattern.
- SepmInterface. A pattern interacts with its environment with *Interfaces* which are composed of *Operations.* We consider two kinds of interface:
  (1) SepmExternalInterface for specifying interactions with regard to the integration of a pattern into an application model or to compose patterns, and
  (2) SepmTechnicalInterface for specifying interactions with the platform.

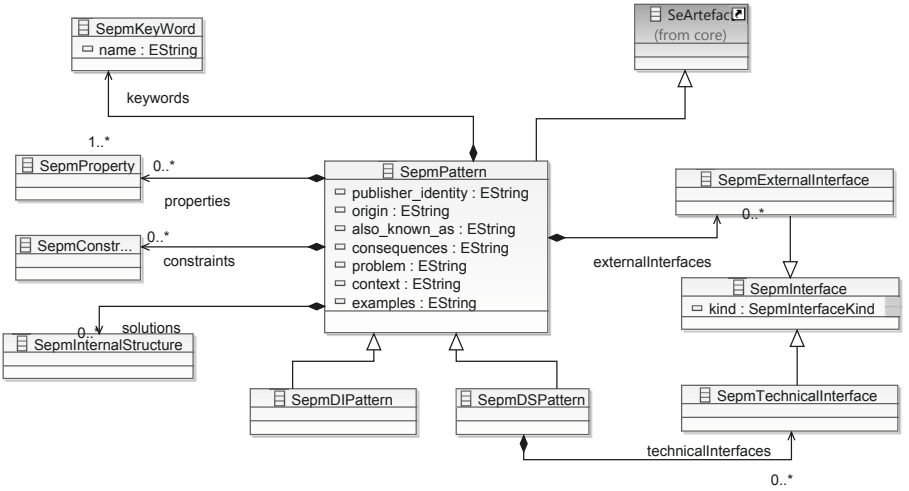


Fig. 5. Pattern Specification Language -Overview

– SepmProperty. is a particular characteristic of a pattern related to the concern dealing with and dedicated to capture its intent in a certain way. Each property of a pattern will be validated at the time of the pattern validation process and the assumptions used will be compiled as a set of constraints which will have to be satisfied by the domain application. Security attributes [21] such as *Confidentiality* and *Availability* are categories of S&D properties.

5 MDE Tool-Chain

Once these specification languages have been defined, it is possible to develop a repository in which modeling artifacts specifications and instances are stored. There are several Domain Specific Modeling Languages (DSML) [7] environments available. In our context, we use the Eclipse Modeling Framework (EMF) [22] open-source platform. Note, however, that our vision is not limited to the EMF platform. Using the proposed metamodels, ongoing experimental work is done under the hood of *semcomdt*[3] (IRIT's editors and platform as Eclipse plug-ins), testing the features of:

(1) *Gaya G* for the repository structure and API conforming to *SARM*,
(2) *Arabion(A)* for specifying patterns and documentation generation conforming to *SEPM*, and
(3) *Deposit* and *Retrieval* for repository access.

[3] `http://www.semcomdt.org`

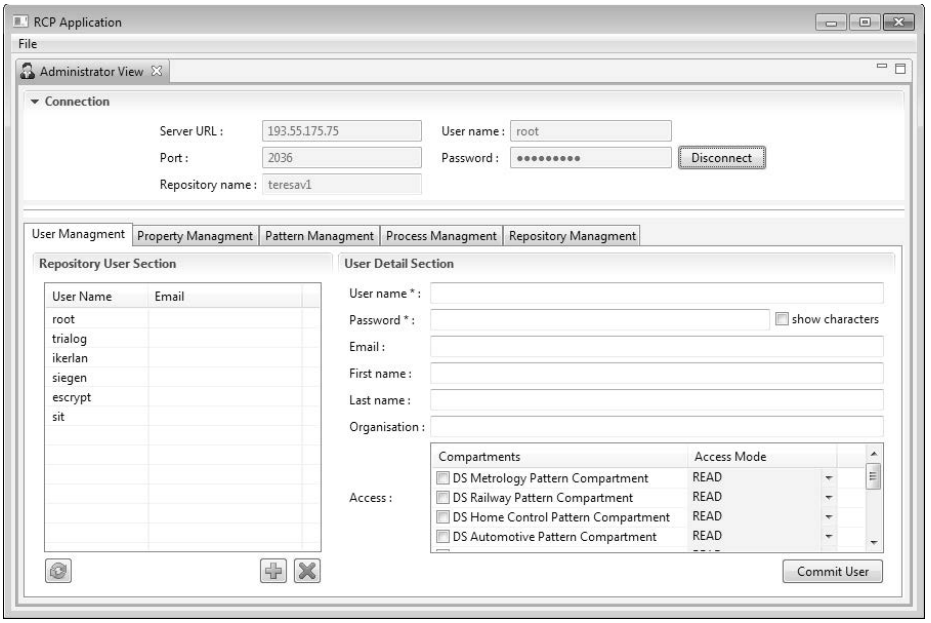


Fig. 6. Repository Implementation

5.1 Repository - Gaya

The implementation of Gaya is based on the SARM metamodel and the repository of S&D pattern structure presented in Sect. 4 on one hand and on Eclipse CDO[4] on the other hand. Repository management is provided via the Gaya tool. Gaya offers repository management with facilities such as pattern lookup, removal, sorting, exporting and categorization. We offered these facilities through a set of dialogs. The main dialog is shown in Fig. 6.

5.2 Pattern Designer - Arabion

The pattern designer called *Arabion* is an EMF tree-based editor for specifying S&D patterns. The design environment is presented in Fig. 7. There is a design palette on the right, a tree view of the project on the left and the main design view in the middle. The design palette is updated regarding the development stage to display suitable design entities for building patterns. These entities are *technical interfaces* and *resource properties*.

In our example, the *SecurityCommunicationLayer@DetailedDesign* uses the *HMAC* mechanism. The call of the method *send()* of the Sender calls internally *generateAH()* to prepare an appropriate authentication header for the data.

[4] `http://www.eclipse.org/cdo/`

Once this header is appended to the message, it is sent by the communication channel. On the Receivers side, the call of the method *receive()* returns the last received message from the sender. This message is checked by the method *checkAH()*. If the message is correct, it is passed to the application, in any other case it is discarded. The operations *generateAH()* and *checkAH()* are provided through an internal interface called *HMAC Computation*.

Moreover, Arabion includes conformance validation tool used to guarantee design validity conforming to the pattern metamodel. In our example, the Secure Communication pattern model can be validated, where a violation of a metamodel construct will yield an error message.

5.3 Pattern Deposit

Pattern publication is triggered by running the *Publication* tool. When executed, the pattern will be stored in the repository following the pattern designer's profile (compartment). The tool uses the *Gaya4Pattern API*, for publishing to the repository. Note, however that the deposit tool requires the execution of the validation tool to guarantee design validity.

5.4 Pattern Retrieval

The tool allows the search/selection of patterns which are used during a system development process. For instance, as shown in the right part of Fig. 8, the tool provides a set of facilities to help the selection of appropriate patterns. The results are displayed in search results tree as System, Architecture, Design and Implementation patterns. For example, the right part of Fig. 8 shows a pattern at design level targeting the *Confidentiality* S&D property[5], named communication and has a keyword *secure*. The tool includes features for exportation and instantiation. In our case, we select the *Secure Communication* pattern for instantiation (see the left part of Fig. 8).

6 Application of the PBSE Methodology to a Case Study

To illustrate our approach we have an industry control application from the railway domain called *Safe4Rail* acting as a TERESA case study. Our goal is also to assess whether the PBSE addresses the practical needs when modeling the trusted embedded system application of a realistic system and whether it can provide significant benefits in terms of reducing modeling effort and error-proneness.

The application is in charge of the emergency brake of a railway system. Its mission is to check whether the brake needs to be activated. Their implementation mainly depends on the safety level to meet, but also on the type and the

[5] In our case, this means that the pattern has a property with a confidentiality category type.

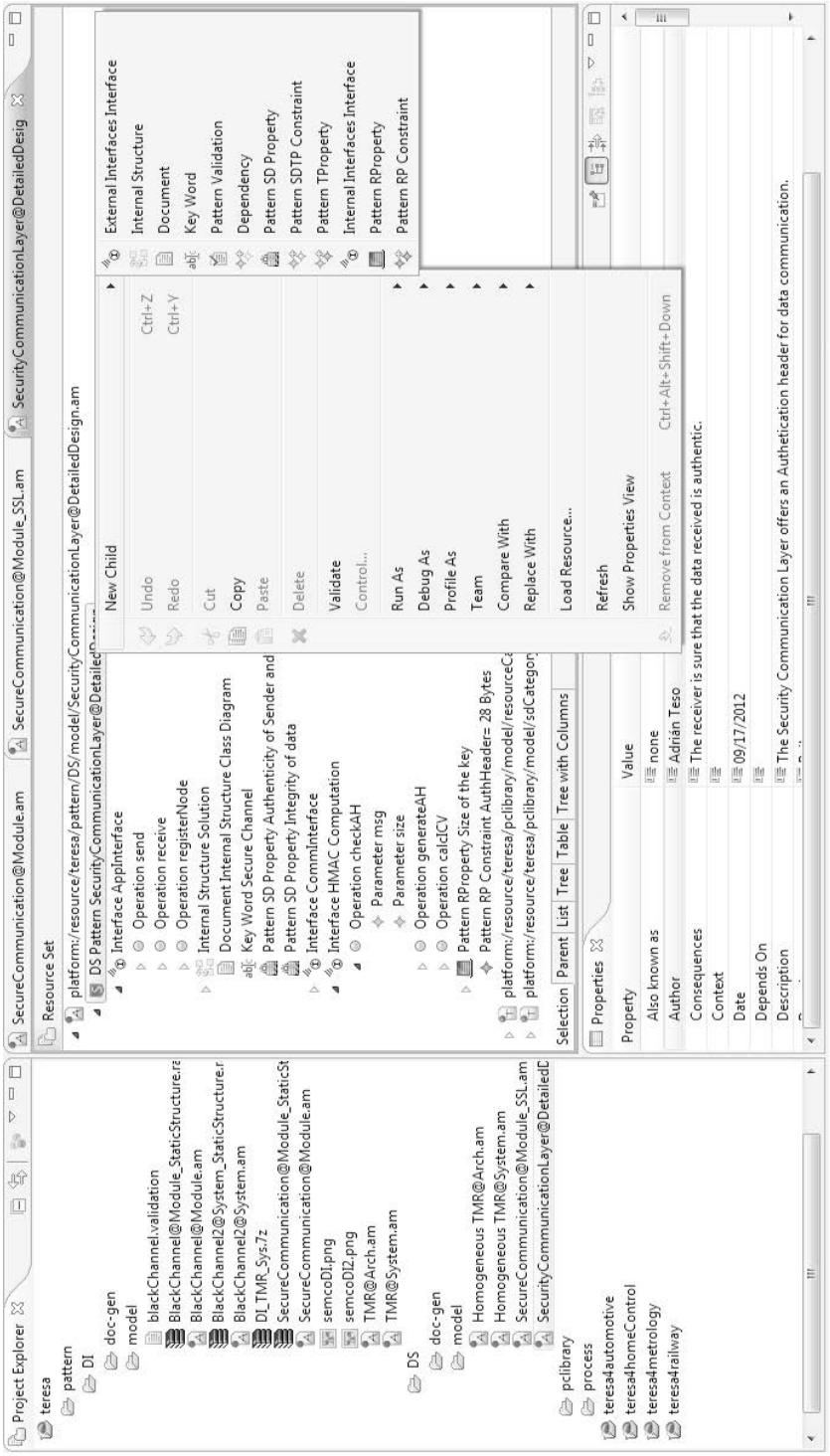


Fig. 7. Secure Communication Pattern at Design level

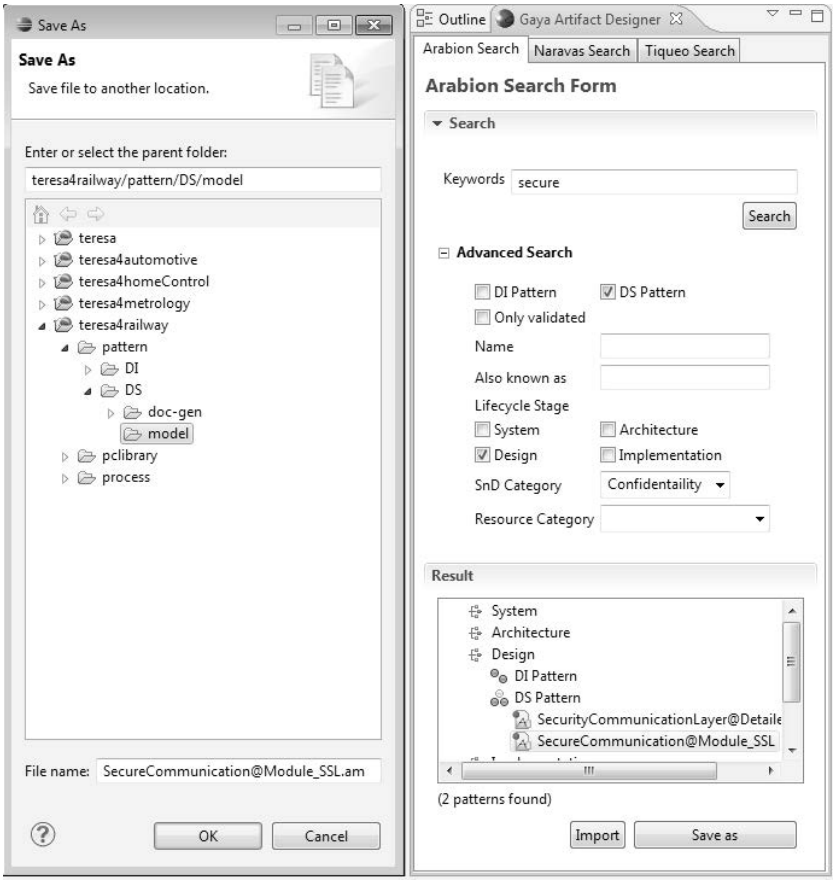


Fig. 8. Pattern Instantiation

number of sensors and actuators involved. These considerations greatly influence how each product is to be implemented (e.g, the number of channel redundancy, the diversity of the channels,. . .). In this case, SIL4 level is targeted. A number of design techniques from S&D are used, namely redundancies, voting, diagnostics, secure and safe communications. A very strict engineering process was followed, where specific activities were performed in order to achieve certification using the presented approach.

6.1 A Model-Based Repository of S&D Patterns Structure Model

The railway domain analysis led to identification of a set of patterns for the Safe4Rail application. We used Arabion to design these patterns and then the Deposit tool to store them in the repository. Figure 9 depicts an overview of the railway S&D pattern language.

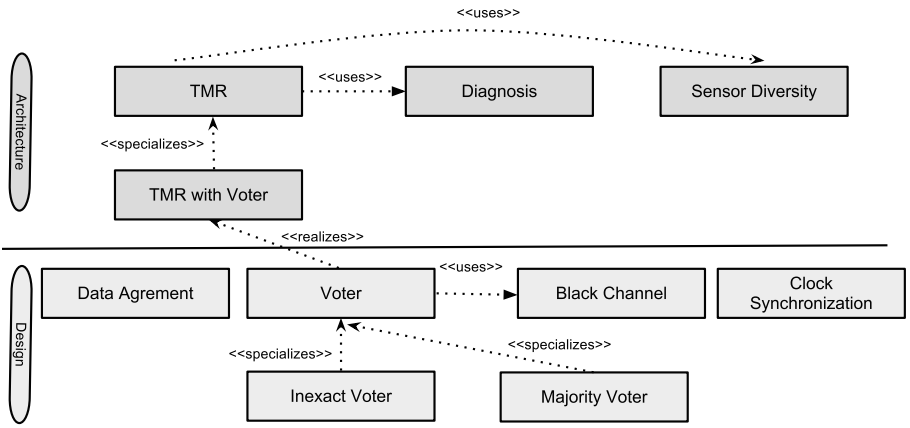


Fig. 9. Railway Pattern Language - Overview

6.2 System Developer View Point: Reuse of Existing Artifacts

Here, we examine the process flow for the example following the design process of Sect. 3.4. Once the requirements are properly captured and imported into the development environment [6], the process can be summarized with the following steps:

Activity A2: Develop architecture model of a system. The analysis of the requirements (A3) results in the needs of an architectural pattern for redundancy. Thus, activity A4 is the instantiation of S&D patterns from the repository using the repository access tools. The running of the Retrieval tool using keywords *Redundancy* and *SIL4*, suggests to use a *TMR* pattern at architecture level. In addition, some diagnosis techniques imposed by the railway standard are suggested, thanks to the repository organization for the railway application domain (see Fig. 9). Finally, at architecture level, we will integrate (A5) the following patterns:

(1) TMR (searched by the System Architect),
(2) Diagnosis techniques (suggested by the tool) and
(3) Sensor Diversity (searched by the System Architect).

Activity A7: Develop design model of a system. This activity involves the development of the design model of the system. The analysis of the requirements (A8), the architecture model and the identified architectural patterns will help during the instantiation activity (A9) of the design phase. Based on the selected patterns, the repository may suggest related or complementary patterns.

[6] Rhapsody is used by Ikerlan Center engineers.

For instance, if the TMR has been integrated, the following patterns may be proposed for the design model iteration: (1) Data Agreement, (2) Voter, (3) Black Channel and (4) Clock Synchronization.

7 Assessment

This section provides a preliminary evaluation of the approach along TAM (Technology Acceptance Model) and concerns the methodology as well as the tools (Arabion and Gaya). We have identified a set of measures to evaluate the usage of the models and the user-friendliness of the tools. Eleven TERESA members participated. The study was divided into three tasks. Before they started, a general description of the aim of the study was given (30'). Some running examples were introduced to them. After these two tasks, achieved during the TERESA MDE workshop in Toulouse (April 2012), a 6-months evaluation was conducted. All the subjects were already familiarized with MDE, S&D patterns and Eclipse. The procedure includes four tasks: SEMCO plug-in installation, pattern development, patterns instantiation and patterns integration.

We asked participants to give scores from 1 to 5 (5 is the best). We first evaluated the perceived usefulness of the solution itself (items 1-4). Next, we focus on the ease of the solution (items 5-6). We want also to measure the compatibility of the solution with existing environments. (items 7-9). Finally, we wanted to measure the willingness to use the approach in the future in the related activities (items 10-12). These scores indicate the degree of satisfaction of the users and provides a feedback to us in order to enhance our specification languages and the tool suite. The following table depicts an overview of the results of our experiment.

Item	**Mean**	**St. Deviation**
1. Design quality	3.5	0.4
2. Model completeness	4	0.3
3. Documentation and artifact generation readability	4	0.89
4. Effort spent on development	4.5	0.16
5. Model understandability	3.5	0.475
6. Effectiveness	3.5	0.6
7. Integration with other solutions	4	0.86
8. Standards compliance	4	0.2
9. Cost of adoption	3.5	0.48
10. Use the approach in the future	4.10	0.68
11. Exchange the approach in the future	3.60	0.56
12. Customize some of the proposed plug-ins in the future	3.60	0.76

Fig. 10. Satisfaction Results

8 Conclusion

We proposed a methodology and an MDE tool-chain to support the specifications and the packaging of a set of S&D patterns, in order to assist the developers of trusted applications for resource constrained embedded systems.

First evidences indicate that users are satisfied with the notion of 'model-based repository of S&D patterns'. The approach paves the way to let users define their own road-maps upon the PBSE methodology. First evaluations are encouraging with 85% of the subjects being able to complete the tasks. However, they also point out one of the main challenges: automatic search for the user to derive those 'S&D patterns' from the requirements analysis. We plan to perform additional case studies to evaluate both the expressiveness and usability of the methodology, the DSLs and the tools. Our vision is for 'S&D patterns' to be inferred from the browsing history of users built from a set of already developed applications.

References

1. Zurawski, R.: Embedded Systems. CRC Press Inc. (2005)
2. Ravi, S., et al.: Security in embedded systems: Design challenges. Transactions on Embedded Computing Systems (TECS) 3(3), 461–491 (2004)
3. Hamid, B., Gürgens, S., Jouvray, C., Desnos, N.: Enforcing S&D Pattern Design in RCES with Modeling and Formal Approaches. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 319–333. Springer, Heidelberg (2011)
4. Riehle, D., Züllighoven, H.: Understanding and Using Patterns in Software Development. Theory and Practice of Object Systems 2(1), 3–13 (1996)
5. Serrano, D., Mana, A., Sotirious, A.-D.: Towards Precise Security Patterns. In: 19th International Conference on Database and Expert Systems Application, DEXA 2008, pp. 287–291. IEEE Computer Society (2008), http://doi.ieeecomputersociety.org/10.1109/DEXA.2008.36
6. Crnkovic, I., et al.: Component-Based Development Process and Component Lifecycle. In: Proceedings of the International Conference on Software Engineering Advances, ICSEA 2006, p. 44. IEEE Computer Society (2006)
7. Gray, J., et al.: Domain-Specific Modeling. Chapman & Hall/CRC (2007)
8. Daniels, F., Kim, K., Vouk, M.A.: The reliable hybrid pattern – a generalized software fault tolerant design pattern. In: Pattern Language of Programs, PLoP 1997 (1997), http://hillside.net/plop/plop97/Proceedings/daniels.pdf
9. Tichy, M., Schilling, D., Giese, H.: Design of self-managing dependable systems with UML and fault tolerance patterns. In: Proceedings of the 1st ACM SIGSOFT Workshop on Self-Managed Systems, WOSS 2004, pp. 105–109. ACM (2004)
10. Yoder, J., Barcalow, J.: Architectural patterns for enabling application security. In: Pattern Languages of Programs, PLoP 1998 (1998), http://hillside.net/plop/plop97/Proceedings/yoder.pdf
11. Schumacher, M.: Security Engineering with Patterns. LNCS, vol. 2754. Springer, Heidelberg (2003)
12. Jürjens, J.: UMLsec: Extending UML for secure systems development. In: Jézéquel, J.-M., Hussmann, H., Cook, S. (eds.) UML 2002. LNCS, vol. 2460, pp. 412–425. Springer, Heidelberg (2002)

13. Hatebur, D., Heisel, M., Schmidt, H.: A security engineering process based on patterns. In: Proceedings of the 18th International Workshop on Database and Expert Systems Applications, DEXA 2007, pp. 734–738. IEEE Computer Society (2007)
14. Halkidis, S.T., Chatzigeorgiou, A., Stephanides, G.: A qualitative analysis of software security patterns. Computers & Security 25(5), 379–392 (2006)
15. Konrad, S., et al.: Using security patterns to model and analyze security requirements. In: Requirements Engineering for High Assurance Systems, RHAS 2003, pp. 13–22. Software Engineering Institute (2003)
16. Di Giacomo, V., et al.: Using security and dependability patterns for reaction processes. In: 19th International Workshop on Database and Expert Systems Application, DEXA 2008, pp. 315–319. IEEE Computer Society (2008)
17. Fernandez, E.B., et al.: Using security patterns to develop secure systems. In: Software Engineering for Secure Systems, pp. 16–31. Information Science Reference (2011)
18. Alexander, C., Ishikawa, S., Silverstein, M.: A pattern language – towns, buildings, construction, vol. 2. Oxford University Press (1977)
19. Buschmann, F., Henney, K., Schmidt, D.C.: Pattern-oriented Software Architecture, vol. 4. John Wiley & Sons (2007)
20. Gamma, E., et al.: Design patterns – Elements of reusable object-oriented software. Addison-Wesley (1995)
21. Avizienis, A., et al.: Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing 1(1), 11–33 (2004)
22. Steinberg, D., et al.: EMF: Eclipse Modeling Framework 2.0. Addison-Wesley (2008)

Static Analysis Techniques to Verify Mutual Exclusion Situations within SysML Models

Ludovic Apvrille[1] and Pierre de Saqui-Sannes[2]

[1] Institut Mines-Telecom, Telecom ParisTech, LTCI CNRS,
Campus SophiaTech, 450 route des Chappes, 06410 Biot, France
ludovic.apvrille@telecom-paristech.fr
[2] Université de Toulouse, ISAE,
10 av. Edouard Belin, B.P. 54032, 31055 Toulouse Cedex 4, France
pdss@isae.fr

Abstract. AVATAR is a real-time extension of SysML supported by the TTool open-source toolkit. So far, formal verification of AVATAR models has relied on reachability techniques that face a state explosion problem. The paper explores a new avenue: applying structural analysis to AVATAR models, so as to identify mutual exclusion situations. In practice, TTool translates a subset of an AVATAR model into a Petri net and solves an equation system built upon the incidence matrix of the net. TTool implements a push-button approach and displays verification results at the AVATAR model level. The approach is not restricted to AVATAR and may be adapted to other UML profiles.

Keywords: Modeling, Model verification, Structural analysis, SysML, Petri Nets, Invariants, Mutual exclusion.

1 Introduction

UML and SysML tools that implement formal verification of real-time systems models commonly reuse reachability analysis techniques and therefore face the state explosion problem [1]. Examples include Artisan Studio [2], SysML Companion [3], OMEGA SysML [4], TOPCASED [5], and TTool [6]. The latest release of TTool, which is addressed in the paper, contrasts with the aforementioned tools by the static analysis it implements for AVATAR, a real-time systems modeling language based on SysML. The paper indeed investigates static analysis of AVATAR models and even focuses discussion on proving mutual exclusion, for instance of shared resources.

The paper reuses the "invariant search" technique originally developed for Petri nets. Unlike papers that propose to write invariants and to check the model against them, the paper generates invariants from the model. In practice, TTool translates an AVATAR model into a Petri net and solves an equation system built upon the incidence matrix of the net. Then, TTool displays verification results at the AVATAR model level. Also, usual questions regarding states in mutual exclusion are asked at the AVATAR model level, not at the Petri net level, and mutual exclusions results are directly given at the AVATAR model level.

F. Khendek et al. (Eds.): SDL 2013, LNCS 7916, pp. 91–106, 2013.

The paper is organized as follows. Section 2 introduces the AVATAR modeling language and the TTool tool. Section 3 reminds the principles of invariant search from a Petri net. Section 4 details how mutual exclusion situations can be detected on AVATAR models. Section 5 presents a case study. Section 6 surveys related work. Section 7 concludes the paper.

2 Avatar: A SysML Environment

2.1 AVATAR Diagrams and Method

The AVATAR language reuses all SysML diagrams, excepted the package diagram. In the early stages of the method associated with AVATAR, a requirement diagram organizes captured requirements in a tree-like structure that shows their attributes, their interrelations and their connections with other elements of the model.

A text diagram lists the modeling assumptions that apply to the system's environment and to the system itself. Incremental modeling starts with strong assumptions that are progressively lowered.

As far as the AVATAR model is built up following an incremental approach, part of the limitations associated with the original modeling assumptions will progressively be removed from the list.

Analysis is use-case driven. A use-case diagram identifies the main functions or services the system offers in relation with external actors. Scenarios (sequence diagrams) and flowcharts (activity diagrams) document the use-cases. Sequence diagrams handle synchronous/asynchronous communications, absolute dates and time intervals. Activity diagrams depict basic actions, tests and loops.

An AVATAR design captures both architectural and behavioral matters [7,8,9]. First, a block instance diagram depicts the architecture of the system as a set of communicating block instances defined by their attributes, methods, and input/output ports. Each block instance has a behavior defined in terms of a finite state machine that supports most SysML state machine elements: input and output signals, variables, timers, time intervals on transitions, enabling conditions and composite states. Examples of non supported elements are history in composite states, and fork/join pseudo states.

The block instance diagram and its associated state machine diagrams have a formal semantics expressed by translation to timed automata for safety proofs, and to pi-calculus processes for security proofs. Design diagrams may be simulated and formally verified from TTool.

2.2 TTool

The AVATAR language is wholly supported by the open software tool *TTool* [6] developed for Linux, Windows and MacOS. The default installation of TTool comes with a diagram editor and a simulator. TTool implements gateways towards three tools that are developed by other laboratories : UPPAAL for the

formal verification of the logical and temporal properties [10], ProVerif for the formal verification of security properties [8,11], and SocLib for the virtual prototyping of the software and hardware of real-time systems [9]. The simulator enables step-by-step and random transition firing. All results are given at the AVATAR level: simulation traces in the form of sequence diagrams and on-the-model identification of the explored transitions. Similarly, the strong advantage of TTool as far as formal verification is concerned is the user-friendliness of the interface to UPPAAL. The user of TTool may indeed check for deadlock freedom, as well as for the reachability and liveness of actions and states, by mere identification of the actions and states on the AVATAR model itself, with no need for an inspection of the UPPAAL "code". Further, there is not need for writing logic formulae.

User friendliness - that is, not needing to know about underlying formal models and proof techniques - has also been a key concern in implementing the new verification approach presented in the paper.

3 Petri Nets and Invariants

This section is a short reminder about Petri nets and P-invariants, that is sufficient to understand our contribution. More information on Petri nets and invariants may be found for example in [12,13,14].

3.1 Verification Techniques for Petri Nets

A Petri net is a bi-partite graph made up of places and transitions. The transition firing policy and the way tokens move from place to place enable the representation of the operation semantics of systems modeled by Petri nets. More formally, Petri nets have been defined as follows [12]:

Definition 1. ***Petri net***
A Petri net is a 5-uple, $PN = (P, T, F, W, M_0)$ *where:*

- $P = \{p_1, p_2, \ldots, p_m\}$ *is a finite set of places,*
- $T = \{t_1, t_2, \ldots, t_n\}$ *is a finite set of transitions,*
- $F \subseteq (P \times T) \bigcup (T \times P)$ *is a set of arcs (called "flow relation" in [12])*
- $W : F \mapsto \{1, 2, 3, \ldots\}$ *is a weight function,*
- $M_o : P \mapsto \{0, 1, 2, \ldots\}$ *is the initial marking.*

By definition, $P \bigcap T = \emptyset$ *and* $P \bigcup T \neq \emptyset$.

In the paper, Petri nets are used to verify AVATAR models that may contain data and time. Nevertheless, the paper restricts discussion to basic Petri nets, as defined above: the limitations of our approach are discussed in section 4. Similarly, the purpose of the paper is not to survey all the verification techniques available for Petri Nets (see for example [12]), and the paper therefore restricts discussion to structural analysis based on place invariants (also called "*P-invariants*").

3.2 P-Invariants

P-invariants are defined from the Petri net incidence matrix. An incidence matrix represents the various transitions of each place. Rows are used for places p_i, and columns for transitions t_j. For example, the value v_{ij} at (p_i, t_j) means that v_{ij} tokens are added (or removed if the value is negative) from p_i whenever transition t_i is fired. More formally, an incidence matrix can be defined as follows [12]:

Definition 2. ***Incidence matrix***

The incidence matrix of a Petri net PN with n transitions and m places is a $n \times m$ *matrix* $A = [a_{ij}]$ *of integers with* $a_{ij} = a_{ij}^{+} - a_{ij}^{-}$ *where:*

- $a_{ij}^{+} = w(i,j)$ *represents the weight of the arc from transition i to the output place j,*
- $a_{ij}^{-} = w(j,i)$ *represents the weight of the arc from the input place j to transition i.*

Definition 3. ***P-invariants***

P-invariants of a Petri net PN are usually defined as $\mathcal{W}.A = 0$ *with* $\mathcal{W}$ *being a set of m weighted places of PN and A being the incidence matrix of PN.*

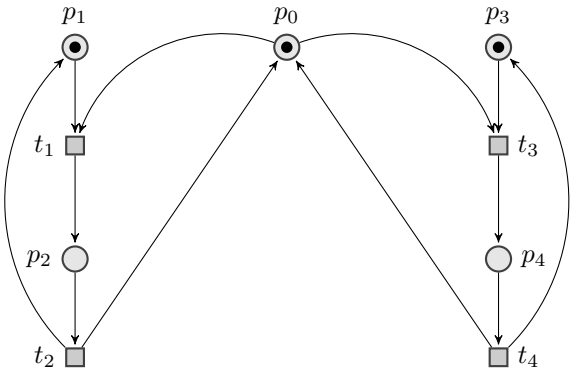
Finally, a P-invariant models a set of places in which the total number of tokens is constant in all reachable markings.

3.3 Algorithms for P-Invariants

Again, P-invariants are defined as $\mathcal{W}.A = 0$. This set of equations can be solved with the Farkas algorithm [15]. The latter allows to compute a set of minimal P-invariants. The complexity of this algorithm is exponential, but heuristics have been proposed in order to reduce this complexity [16].

3.4 Example

P-invariants can be used to prove mutual exclusion situations. Indeed, the mutual exclusion between two subnets s_1 and s_2 of a Petri net PN can be proved by showing that at most 1 token is present in the marking of places of s_1 and s_2. Let us illustrate mutual exclusion with the following Petri net:



The transpose incidence matrix A^t of this Petri net is as follows:

$$A^t = \begin{array}{c} \\ p_0 \\ p_1 \\ p_2 \\ p_3 \\ p_4 \end{array} \begin{array}{c} \begin{array}{cccc} t_1 & t_2 & t_3 & t_4 \end{array} \\ \begin{pmatrix} -1 & 1 & -1 & 1 \\ -1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & -1 & 1 \\ 0 & 0 & 1 & -1 \end{pmatrix} \end{array}$$

To resolve $\mathcal{W}.A = 0$ (see Definition 3), the matrix is made triangular, as for solving a linear system: lines can be exchanged, multiplied by a given integer value, or one line can be added to another one. Applying this to the A matrix gives:

$$A^t_{triangular} = \begin{array}{l} \\ p_0 \\ p_4 \\ p_0 + p_2 + p_4 \\ p_3 + p_4 \\ p_1 + p_2 \end{array} \begin{array}{c} \begin{array}{cccc} t_1 & t_2 & t_3 & t_4 \end{array} \\ \begin{pmatrix} -1 & 1 & -1 & 1 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

Finally, the P-invariants are p_1+p_2, p_3+p_4, $p_0+p_2+p_4$. Each of them represents a mutual exclusion situation. For example $p_1 + p_2$ models the fact that either there is a token in p_1 or in p_2. Similarly, $p_0 + p_2 + p_4$ proves a mutual exclusion between, in particular, places $p2$ and p_4.

4 Our Approach

4.1 Overview

The main contribution of the paper is the computations of "SysML model invariants". The latter are a list of SysML state machine elements of a model that are all executed in mutual exclusion. We propose to compute these invariants using P-invariants as defined in previous section. Fig. 1 depicts the approach implemented in TTool. As stated before, there are already two possibilities for proving properties from AVATAR design models. The first possibility is to generate a pi-calculus specification, and then using ProVerif for verifying security properties. The second possibility is to generate timed automata that are taken as input by UPPAAL to verify safety properties. The contribution based on P-invariants is displayed at the right side of Fig. 1:

1. AVATAR design models to be studied are first translated to a Petri net. The translation to a Petri Net could be applied to all Domain Specific Languages defined to model sets of entities communicating with synchronous or asynchronous channels, and whose behavior is described with state machines.

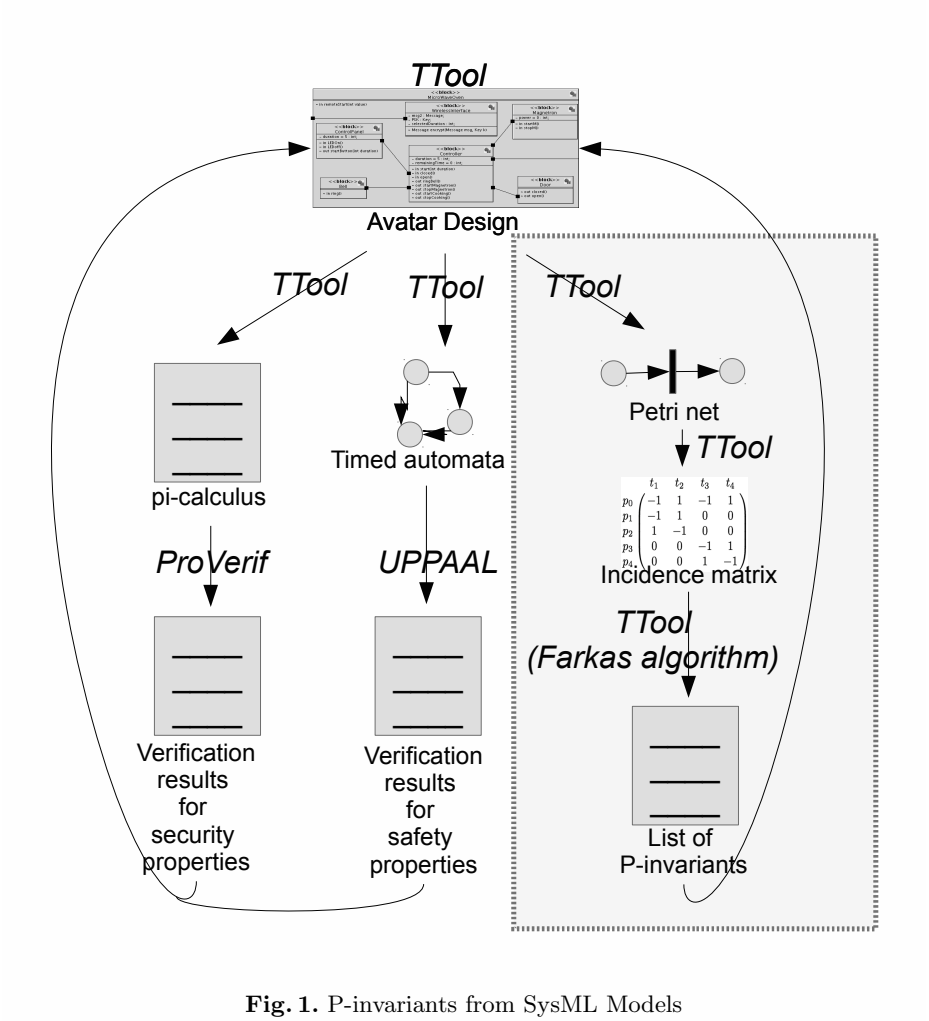


Fig. 1. P-invariants from SysML Models

2. The Petri net is in turn translated to an incidence matrix.
3. The Farkas algorithm [15] is used to compute the set of minimal invariants.
4. The invariants are filtered so as to keep only the relevant ones. Relevant invariants are the ones that are not concerning only one block.
5. The invariants are finally back-traced to the SysML model under the form of a SysML model invariant.

4.2 SysML Model Invariant

We have defined SysML model invariants for design diagrams only. A SysML model invariant is a list of model elements of the state machines: it may refer

to a message sending, a message reception or a state. Each element is translated into a set of places and transitions, and so, for back-tracing P-invariants to the SysML model, it is necessary to keep track of the correspondence between operators and places/transitions. The translation of an AVATAR design to a Petri net is now further detailed.

4.3 AVATAR Design Model Translation to Petri Nets

Again, an AVATAR model is made up, on the one hand, of the definition the architecture in terms of blocks and communication between blocks (synchronous, asynchronous), and on the other hand, of one state machine per block. AVATAR security-specific operators [8] in both architecture and behavioral diagrams are totally ignored in this translation process.

Translation of State Machines. AVATAR states machines are built upon the following operators: states, transitions (guards, actions on variables, time constraints), timers (set, wait for expiration, reset), non-deterministic choices, and communication operators (message sending, message receiving). Let us investigate the basics of the translation for all of them.

- **States**. Each state is translated as one place.
- **Transitions**. Transitions are constrained with time delays and Boolean guards. Both constraints are ignored by our translation process since the type of Petri nets we rely on does not support time nor variables. Similarly, actions on variables inside transitions are ignored. Finally, transitions are translated into a Petri net transition.
- **Timers**. Since time constraints cannot be represented in the Petri nets we rely on, timers are ignored.
- **Non-deterministic choices**. Each of them is translated to exactly one place.
- **Communication operators**. Their translation is of utmost importance since they model the synchronization between tasks. Their translation is the most complex one and is further explained in next subsection.

Finally, the translation process ignores time constraints and variables, which means the translation to Petri nets only takes into account block to block communications and state to state transitions as described by the state machines of the blocks. We now explain how AVATAR synchronous and asynchronous communications are translated.

Translation of Synchronous Communications between Blocks. Synchronous channels are declared at SysML block diagram levels. A synchronous channel can apply to several blocks. The translation of a given synchronous communication channel *ch* is a two-step process:

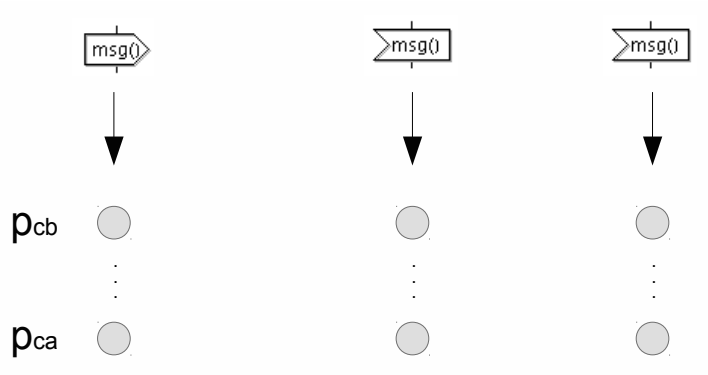


Fig. 2. Translating synchronous communications: step 1

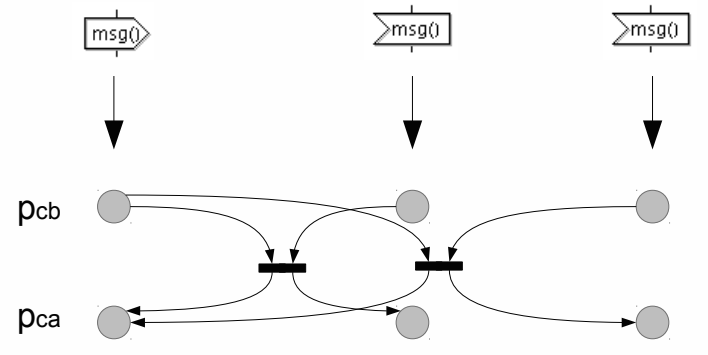


Fig. 3. Translating synchronous communications: step 2

1. For each communication operator c (sending in ch, receiving from ch) of all state machines, two places are generated: p_{cb} models the waiting for the synchronization, ("b" means before) and p_{ca} ("a" means after) models the situation after synchronization occurrence. For example, Fig. 2 represents three communications involving the same channel msg. Thus, for each operator of a design involving msg (one sending operator and two receiving operators) two corresponding places are created.
2. For possible synchronization of ch, i.e., for each possible couple c_{sr} (sending, receiving) of ch, we do the following: We create a new transition t_{csr} with two incoming edges from the p_{cb} places of sending and receiving, and two outgoing edges from t_{csr} to the after-synchronization places p_{ca} of the sender and the receiver. Figure 3 depicts how the places generated in the first step (see Fig. 2) are linked together through two transitions modeling the two possible synchronizations.

Translation of Asynchronous Communications between Blocks. AVATAR asynchronous communication is based on a finite FIFO, with two different writing policies:

1. Writers are blocked when the FIFO is full.
2. Writers are not blocked when the FIFO is full, that is, an element of the FIFO is considered to be removed whenever a write operation in a full FIFO is performed.

The two policies are translated as follows. For the first one (writers are blocked), the main idea is to have a place containing n tokens, with n being the maximum capacity of the FIFO (see Fig. 4). Then, one token is moved to another place when a write operation is performed, and one token is moved back to its initial location whenever a read operation occurs. Thus, write and read operations are blocked when the FIFO is full or empty, respectively.

The second FIFO policy is translated quite similarly to the previous one (see Fig. 5): the main difference relies in an additional transition that is used whenever the FIFO is full so as to "unblock" the writer.

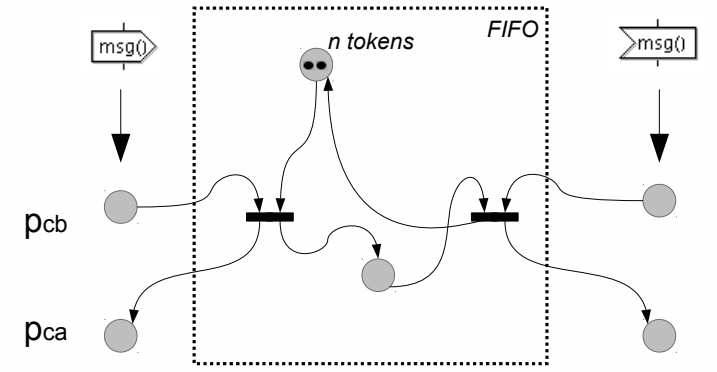


Fig. 4. Translating asynchronous blocking communications

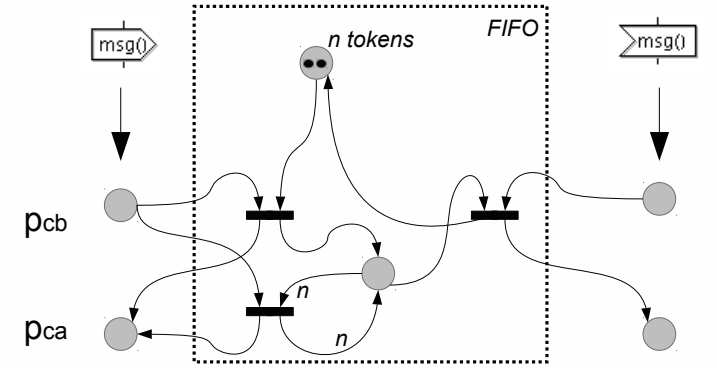


Fig. 5. Translating asynchronous non-blocking communications

5 Case Study

The objective of this case study is twofold. First, it intends to demonstrate the effectiveness of P-invariants to identify mutual exclusion situations in SysML diagrams. Second, it illustrates how P-invariants have been integrated in TTool to facilitate their usage.

5.1 Description of the System: A Microwave Oven

The microwave can be started using a button, or a remote control. Whenever the door is opened, the magnetron must be turned off (safety constraint). Also, the remote control must be secured, that is, a remote control must be attached to only one specific oven (authenticity constraint), and messages sent from the remote control must not be disclosed (confidentiality constraint). Finally, the oven model shall satisfy both safety and security constraints. Yet, security matters are out-of-scope of the paper, but have been explained in [8].

5.2 Design

The design is made upon several types of blocks and elements (see Fig. 6):

- A main block named "RemotelyControlledMicrowave" contains all other blocks modeling the system: the remote control, and the microwave oven itself composed of a wireless interface, a magnetron, a door, a bell and a control panel. Each block declares attributes, methods and signals.
- The declaration of two data types (Key, Message) at the lower left part of the diagram.
- The declaration of security-related constraints and properties in the note located at the top of the diagram.
- The declaration of communication channels between blocks. Ports filled in black represent synchronous communication whereas ports filled in white represent asynchronous communications. Signals and ports can be used by the block declaring them, and by the blocks it contains. For example, all blocks may use the asynchronous channels connecting "RemotelyControlled-Microwave" to itself.
- The declaration of an observer whose purpose is explained hereafter.

5.3 Formal Verification of Safety Properties

One safety property is at stake in this system: "the magnetron must be off whenever the door is opened".

A first way to prove this property in TTool is the usual way to do: adding an observer in the model (see Fig. 6 "ObserverProp1" block). The latter contains an "error" state whose reachability can be studied directly from TTool using UPPAAL. Of course, this technique requires to "execute" the model, and explore

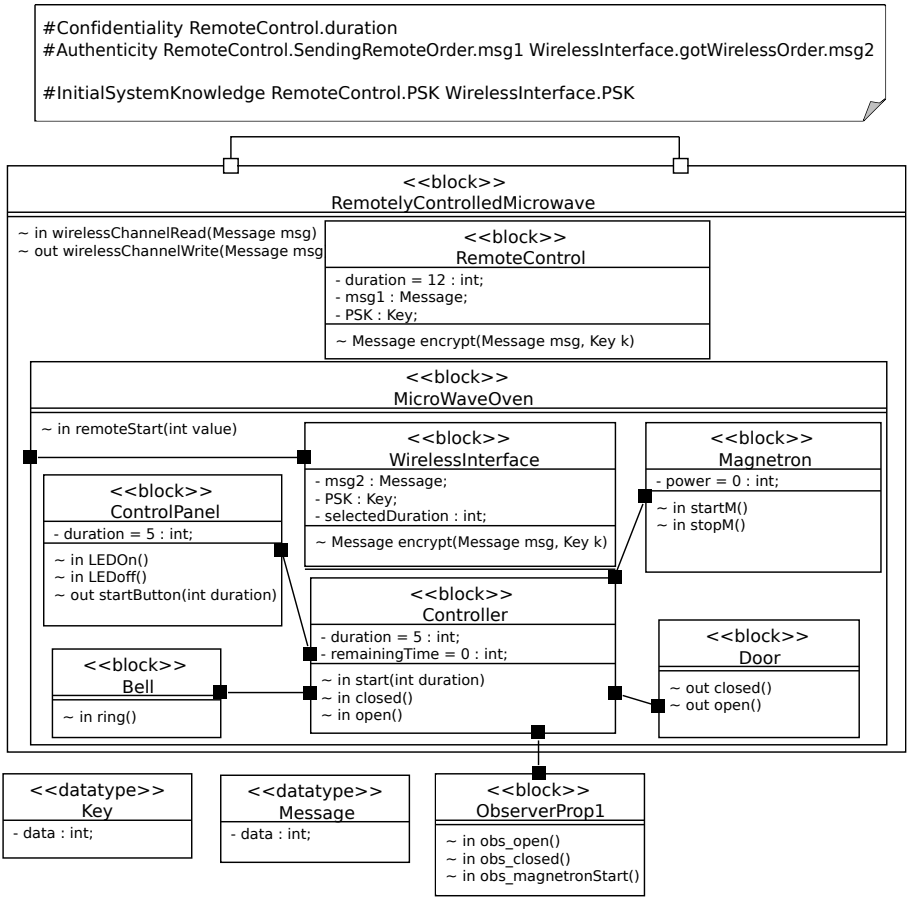


Fig. 6. AVATAR block diagram of the microwave system

all its branches to be certain that none of them contains that "error" action. A second way to do relies on P-invariants. TTool computes model invariants. Once computed, invariants are listed on the left part of the main window of TTool (see Fig. 7). The user of TTool may select one invariant. Then, all graphical elements of that invariant are underlined with an "inv" annotation (see the circle in Fig. 7), making it very easy to parse all elements of each invariant. All elements of the same invariant are mutually exclusive.

From invariants, TTool offers another nice graphical way to visualize mutual exclusive situations: putting the mouse on a given state displays the list of all states that are mutually exclusive with the selected one. For example, the "DoorIsOpened" state in the Door block (see Fig. 8), no state in mutual exclusion could be identified. Indeed, in a first model, it is possible to have the

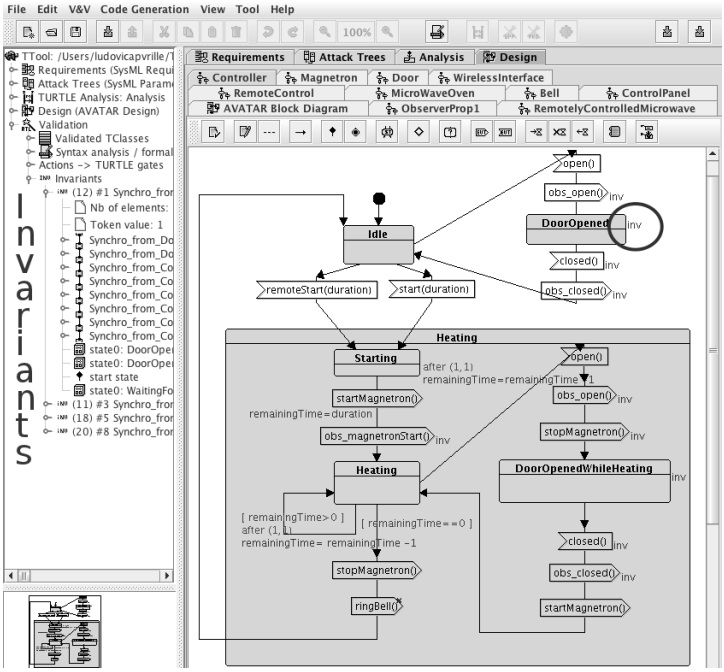


Fig. 7. Invariants as displayed in TTool

door opened while the magnetron is on. We have therefore modified the model as follows: we added a lock on the door, i.e., when the user wishes to open the door, the microwave first turns off the magnetron before releasing a lock on the door. With this model, the "DoorIsOpened" state is mutually exclusive with the state "Running" of Magnetron, see Fig. 9, which proves that the magnetron is off whenever the door is opened.

5.4 Discussion

Identifying mutual exclusion situations has now become a piece of cake in TTool thanks to the invariants. Heuristics we have defined and implemented - that are not detailed in the paper - make it possible to compute invariants in a few seconds on the most complex models we have made with AVATAR. This is in particular the case for an automotive application published in [9], and developed in the scope of the EVITA European project [17].

Yet, one must be aware of the main current limitation: not all model elements are taken into account to compute invariants, as explained in section 4. In particular, time constraints and variables are not considered in the translation process. In other words, invariants are computed for a reduced model that contains more

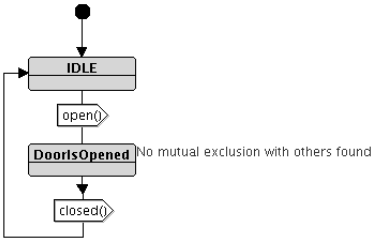


Fig. 8. Mutual exclusions of the state "DoorIsOpened" of the Door block (first version)

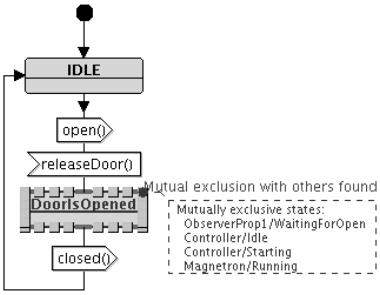


Fig. 9. Mutual exclusions of the state "DoorIsOpened" of the Door block (second version)

traces than the original model. And so, two states identified as mutually exclusive by invariants are really mutually exclusive ... but two states not identified as mutually exclusive might still be mutually exclusive. To address that issue, TTool puts a warning textnote ("Mutual exclusion could not be studied") next to the states for which the mutual exclusion could not be proved with invariants.

Last, but not least, the AVATAR-to-Petri Net transformation has been programmed in an ad-hoc manner. It could be interesting to describe this transformation with a model transformation language.

6 Related Work

Real-Time Dialects of SysML. The System Modeling Language [18] is a UML profile [19] that may be tailored in turn to fit in with an application domain. Given the concept of "profile of profile" has not been defined by the UML standard, a tailored version of SysML may be termed as a "SysML dialect". Examples of such dialects have been defined for real-time systems. For example, [20] suggests to simplify SysML and to formalize communication ports.AVATAR, which is the subject of the paper, ignores continuous flows and merges the block definition and internal block definition diagrams into one block instance diagram. AVATAR provides a semantics to most SysML state machine elements.

SysML Tools. A survey of the literature indicates that SysML tools that target real-time systems have been developed on top of UML tools in the form of SysML add-ons that reuse the capacity of the UML tool to translate a high level model into a formal model that may cater a formal verification tool.

- [21] uses Rhapsoldy and timed automata to formally verify the landing gear of a military aircraft.
- Artisan Studio [2] associates parametric diagrams with solvers such as Matlab or Excel.

- SysMLcompanion [3] translates a SysML model into a VHDL-AMS one.
- OMEGA SysML [4] translate a SysML model into a private intermediate form: IF.
- TOPCASED [5] also translates a SysML model into a private intermediate form: FIACRE.
- TTool [6], which is the subject of the paper, translates an AVATAR model into a timed automata, a pi-calculus specification or a Petri for temporal property verification, security flaw detection, and invariant search, respectively. What makes TTool really user-friendly for people not familiar with formal verification and formal methods in general, is the way the tool drives formal verification at the SysML model level and displays results at the SysML level too. The user of TTool is indeed not obliged to write logic formula to achieve formal verification. Nor he or she is obliged to inspect formal code to understand verification results.

Petri Nets and Invariants. Invariant search was introduced several decades ago for basic Petri nets. [22] reports a successful experience in applying invariant search to demonstrate token unicity on a local area network. The techniques is still implemented by Petri net tools such as TINA [23]. It has also been extended to search invariants in colored Petri nets [24] .

7 Conclusion

SysML tools that implement reachability techniques face the state explosion problem as far as complex real-time systems verification is at stake. So far, the open-source toolkit TTool has fallen in that category, for it translates a SysML/AVATAR model into timed automata, and model-check the latter using UPPAAL.

By contrast, the paper investigates formal verification of SysML/AVATAR models using a structural approach that does not require generating the state space of the model. The idea is to translate an AVATAR model into a Petri net and to search for invariants by solving an equation system derived from the incidence matrix of the Petri net. TTool implements invariant search algorithms and displays results at the SysML level. It also enables the designer to look for mutually exclusive actions or states in the SysML model. The user-friendliness added to the invariant interpretation phase is a real added value of the tool.

The overall contribution could be adapted to other UML and SysML environments structuring systems with classes / blocks and state machine diagrams.

An education case study of modeling a microwave oven has shown that invariant search usefully complements model checking. Risks of starting the oven before the door is actually closes have been revealed by invariant search only, and a handshake procedure has been added to the model.

References

1. Debbabi, M., Hassaïne, F., Jarraya, Y., Soeanu, A., Alawneh, L.: Verification and Validation in Systems Engineering: Assessing UML/SysML Design Models, p. 270. Springer (2010) ISBN 978-3-642-15227-6
2. Atego ARTiSAN Studio, http://www.atego.com/products/artisan-studio/
3. SysML Companion, http://www.realtimeatwork.com/software/sysml-companion/
4. Dragomir, I., Ober, I., Lesens, D.: A Case Study in Formal System Engineering with SysML. In: 17th International Conference on Engineering of Complex Computer Systems (ICECCS 2012), pp. 189–198. IEEE Computer Society (2012)
5. TOPCASED, http://www.topcased.org
6. TTool, http://ttool.telecom-paristech.fr
7. Knorreck, D., Apvrille, L., De Saqui-Sannes, P.: TEPE: A SysML Language for Time-Constrained Property Modeling and Formal Verification. ACM SIGSOFT Software Engineering Notes 36(1), 1–8 (2012)
8. Pedroza, G., Knorreck, D., Apvrille, L.: AVATAR: A SysML Environment for the Formal Verification of Safety and Security Properties. In: New Technologies of Distributed Systems (NOTERE), pp. 1–10. IEEE (2011), http://dx.doi.org/10.1109/NOTERE.2011.5957992
9. Apvrille, L., Becoulet, A.: Prototyping an Embedded Automotive System from its UML/SysML Models. In: Proceedings of Embedded Real Time Systems and Software (ERTSS 2012) (2012), www.erts2012.org/Site/0P2RUC89/3C-1.pdf
10. 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)
11. Blanchet, B.: Using Horn Clauses for Analyzing Security Protocols. In: Formal Models and Techniques for Analyzing Security Protocols. Cryptology and Information Security Series, vol. 5, pp. 86–111. IOS Press (2011)
12. Murata, T.: Petri Nets: Properties, Analysis and Applications. Proceedings of the IEEE 77(4), 541–580 (1989)
13. Diaz, M.: Modeling and analysis of communication and cooperation protocols using petri net based models. Computer Networks 6(6), 419–441 (1982)
14. Diaz, M.: Petri Nets: Fundamental Models, Verification and Applications, p. 768. John Wiley & Sons (2009)
15. Farkas, J.: Theorie den einfachen Ungleichungen. Journal für die Reine und Angewandte Mathematik (Crelle's Journal) 124, 1–27 (1902)
16. Colom, J.-M., Silva, M.: Improving the Linearly Based Characterization of P/T Nets. In: Rozenberg, G. (ed.) APN 1990. LNCS, vol. 483, pp. 113–145. Springer, Heidelberg (1991)
17. Kelling, E., Friedewald, M., Leimbach, T., Menzel, M., Séger, P., Seudié, H., Weyl, B.: Specification and evaluation of e-security relevant use cases. Technical Report Deliverable D2.1, EVITA Project (2009)
18. Object Management Group: OMG Systems Modeling Language (OMG SysML™) Version 1.3, http://www.omg.org/spec/SysML/1.3/PDF/
19. Object Management Group: Documents Associated With Unified Modeling Language (UML), V2.4.1, http://www.omg.org/spec/UML/2.4.1/
20. Ober, I., Ober, I., Dragomir, I., Aboussoror, E.A.: UML/SysML semantic tunings. Innovations in Systems and Software Engineering 7(4), 257–264 (2011)

21. da Silva, E.C., Villani, E.: Integrando sysml e model checking para v&v de software critico espacial. In: Brasilian Symposium on Aerospace Engineering and Applications (2009), http://www.cta-dlr2009.ita.br/Proceedings/PDF/59054.pdf
22. Ayache, J.-M., Courtiat, J.-P., Diaz, M.: REBUS, A Fault-Tolerant Distributed System for Industrial Real-Time Control. IEEE Transactions on Computers 31(7), 637–647 (1982)
23. Time Petri Net Analyzer, http://projects.laas.fr/tina/
24. Jensen, K.: Coloured Petri Nets and the Invariant Method. Theoretical Computer Science 14(3), 317–336 (2002)

Significantly Increasing the Usability of Model Analysis Tools through Visual Feedback

El Arbi Aboussoror, Ileana Ober, and Iulian Ober

IRIT, Université de Toulouse, 118 Route de Narbonne
F -31062 Toulouse, France
{El-Arbi.Aboussoror,Ileana.Ober,Iulian.Ober}@irit.fr

Abstract. A plethora of theoretical results are available which make possible the use of dynamic analysis and model-checking for software and system models expressed in high-level modeling languages like UML, SDL or AADL. Their usage is hindered by the complexity of information processing demanded from the modeler in order to apply them and to effectively exploit their results. Our thesis is that by improving the visual presentation of the analysis results, their exploitation can be highly improved. To support this thesis, we define a trace analysis approach based on the extraction of high-level semantics events from the low-level output of a simulation or model-checking tool. This extraction offers the basis for new types of scenario visualizations, improving scenario understanding and exploration. This approach was implemented in our UML/SysML analyzer and was validated in a controlled experiment that shows a significant increase in the usability of our tool, both in terms of task performance speed and in terms of user satisfaction.

1 Introduction

A plethora of theoretical results are available which make possible the use of dynamic analysis and model-checking for software and system models expressed in high-level modeling languages UML [1], SDL [2] or AADL [3]. These results have the virtue of allowing reasoning at the level of models instead of code, making possible early verification and validation, while taking advantage of the abstract nature of modeling languages. Unfortunately, the use of these advanced techniques is not as widespread as their capabilities could justify. One reason for this is that such techniques are often inaccessible to modelers with a basic software engineer training, as it demands advanced knowledge of these techniques and a high investment in learning the specificities of tools in order to apply them and to effectively exploit their results. Our thesis is that by improving the visual presentation of the analysis results, their exploitation by regular users can be highly improved. To support this thesis, we defined a trace analysis approach and we integrated it in Metaviz, a model-driven framework for simulation trace visualization introduced earlier in [4].

The typical functioning of a model-based formal analysis tool, consists in mapping the semantics of the high-level modeling language into a simpler language, more well-suited for formal analysis. Most of the time this language has

F. Khendek et al. (Eds.): SDL 2013, LNCS 7916, pp. 107–123, 2013.

a sound formal semantics that allows reasoning on the model and on its properties, using analysis tools. It is the case of vUML [5] that uses Promela [6], of IFx-OMEGA [7], that uses IF [8], etc. This change of context is in general not supported by a bi-directional mapping to and from the formal language, since the goal is to represent a complex modeling language with a limited set of primitives available in the formal language. Thus, the structure of a model may be very different between the two levels, and analysis results such as execution traces obtained from the lower-level model may be hard to interpret by a user whose knowledge is limited to the upper-level language and model. The trace analysis method is based on the extraction of high-level semantics events from basic events of the underlying semantics. It offers the basis for new types of scenario visualizations, improving the model execution understanding and exploration.

In order to validate these assertions we set up a controlled experiment that shows a significant increase in our model checking tool usability, both in terms of task performance speed and in terms of user satisfaction.

The rest of this paper is organized as follows: we start by an overview of existing methods for executable UML/SysML analysis, and by discussing why exploiting analysis results is challenging. Then we present the typical analysis tool integration pattern and finally describe our approach for extending the verification platform with trace analysis support. The evaluation of the new tool is detailed and discussed in Section 3.

1.1 Translational Semantics Approaches to Model Analysis

To be useful (not only descriptive) models need to have a well defined semantics. Among other advantages, formal definition of semantics provides the possibility to do analysis earlier in the design process. For example non-functional properties such as performance, schedulability or safety can be analyzed. For this purpose, mature formalisms and associated analysis techniques and tools already exist (Petri nets, queuing networks, Markov chains, etc.).

For Model Driven Engineering, the approach usually followed is to:

1. annotate the original models (e.g. UML [9]),
2. generate input artifacts for a formal analysis tool, and finally
3. perform analysis activities on the generated artifacts.

Target analysis formalisms can be process algebra, timed automata, queuing networks etc. Some examples of tools working in this way are IFx-OMEGA[10], vUML [5] or OPTIMUM [11].

Even if these model analysis techniques, called translational semantics approaches, are widely adopted in the industry they still suffer from limitations. Translational semantics approaches to model analysis bring a new complexity to the end user. The analysis results (e.g. model checking counterexamples) are not easily understood by the domain user. This is due to the gap between the two semantic levels: the original one and the target analysis formalism. Those results necessitate expertise in the low level analysis semantics. To enable a usable approach to model analysis a translation of the low level results to a more user friendly abstraction should take place in the process.

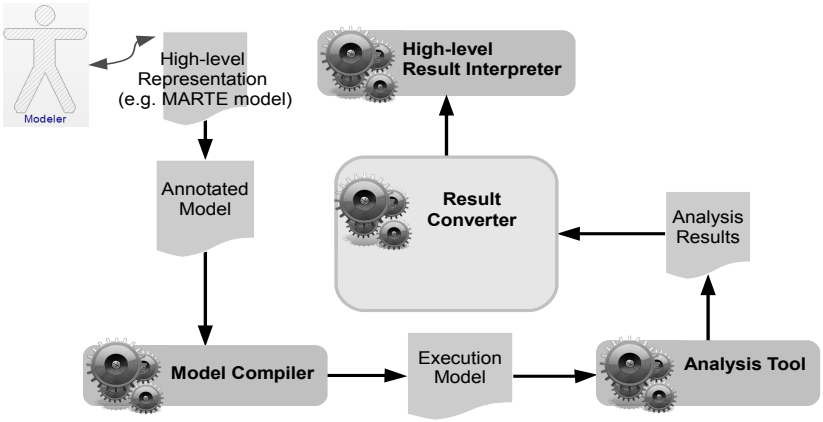


Fig. 1. Generic approach for analysis results exploitation. The annotated model is translated to a formal model for analysis. Results are then showed to the end user in a visual interpreter.

1.2 General Schema for Analysis Tool Integration

Analyzing models is assessing some properties of them. For example UML offers an extension mechanism, namely profiles that enable building a language to express non-functional properties. This approach offers a clear advantages like the reuse of the tools and techniques available for UML. Two well known examples are SPT and MARTE profiles. UML Profiles are integrated with analysis tools using the basic architecture described in Fig. 1. In this architecture model transformations are used to fill the gap between UML modeling technical space and the analysis technical space. This semantic gap is at the heart of the diagnostic problem described in the previous section1.1.

2 Extending a Verification Platform with Trace Analysis Support

IFx-OMEGA[1] platform integrates a compiler, a simulator and model-checker for a rich subset of UML and SysML [7]. The toolset relies on the automatic translation of models into a lower-level language (named IF) based on asynchronous communicating extended timed automata, and on the use of the extensive toolset available for this language [8]. The validation acts on a UML or SysML model, which is first translated to an IF model, and then compiled[2] to an executable program that will be used for automatic verification and interactive simulation.

[1] `http://www.irit.fr/ifx`

[2] In some cases, the model can be first simplified using automatic abstraction techniques.

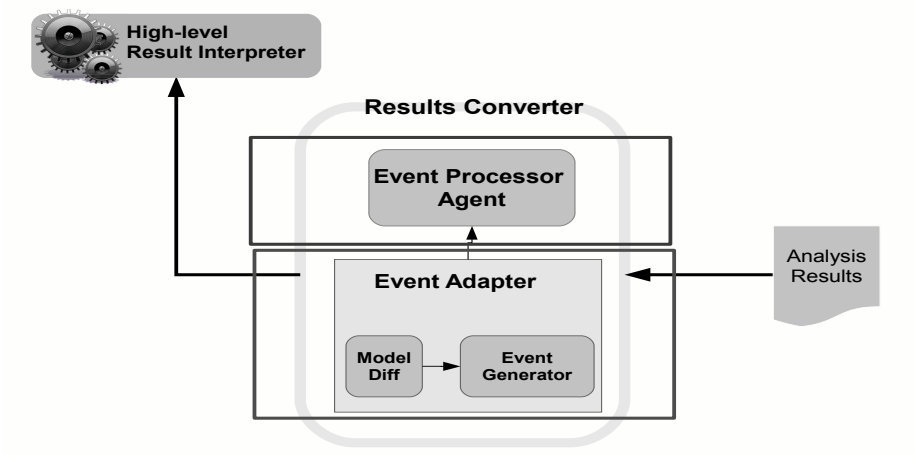


Fig. 2. Generic architecture for extracting high-level events from analysis results using a *Result Converter*. It is composed of an *Event Adapter* for generating relevant events from an operational semantics and an *Event Processor Agent* that translates those events to user level events.

The IFx validation approach has been applied to several industry-grade models such as Ariane-5 [12], MARS [13] and SGS [10], and has proven to be very effective in discovering design issues. As an application to this approach we propose to extend the IFx-OMEGA platform. The extension transforms low-level semantics analysis result. Figure 2 shows the architecture of this extension. The automatic verification and the simulation activities generates execution scenarios as analysis results. Those scenarios are expressed using the low-level semantics used by the IF model checker. The extension we propose translates the relevant information from the scenarios into a high-level like formalism syntax. To extend the IFx-OMEGA platform with result analysis facility we follow the generic architecture proposed in [14] and depicted in Fig. 1.

2.1 Extracting Execution Events Using Model Differences

The generic architecture has to be refined to enable a flexible and extensible implementation. We choose to take an event based approach to report the analysis results. The design rationale behind this choice is the decoupling introduced by this approach and its adequacy to our context. Indeed an event reports changes in monitored states [15] and since we are in the context of translating UML models to an operational semantics we have this notion of state changes within the execution model. Those state changes are captured using a model difference mechanism [16] and the difference model is then transformed into a set of events

by an *event generator*. Those events report what has occurred in the analysis results (e.g. state changes, message passing, ...). An example is shown in Fig. 3. This figure shows an original trace on the left (an XML file loaded in an tree-based view) alongside the extracted event. The event showed on the right correspond to the computed difference between two configurations of the system in the trace. In the example the difference between the system configurations (from step 15 to 16) is a state change of the *Model_EVC instance* from *Waiting* status to *Started*.

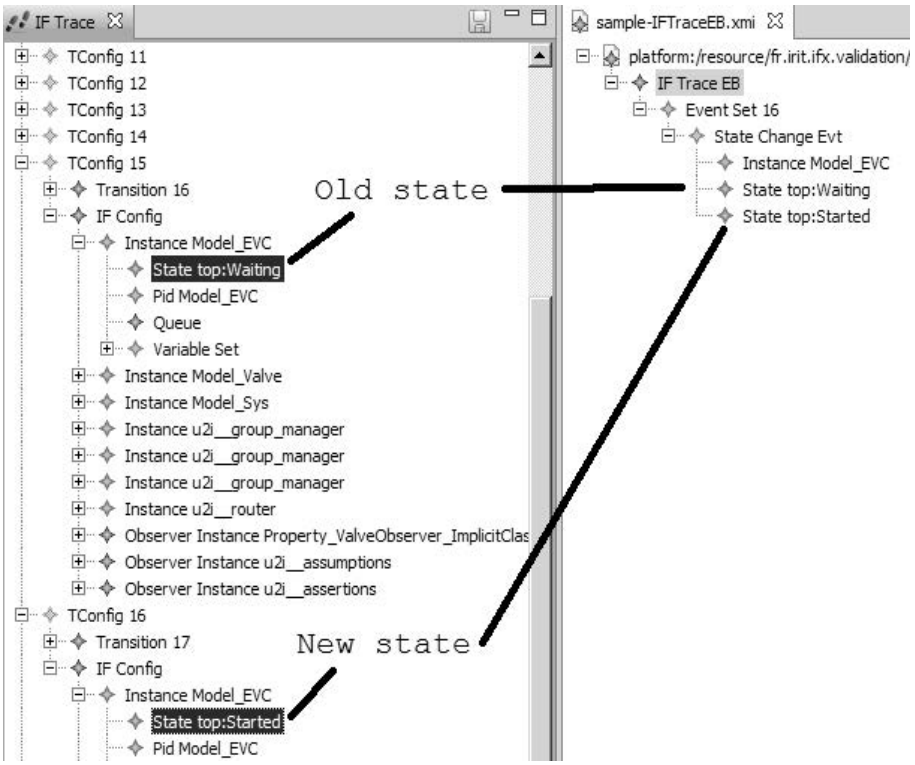


Fig. 3. Extracting execution events using model differences. State changes are captured using a model difference mechanism, the difference model is transformed into a set of events by an *event generator*.

2.2 Abstracting Execution Events to High-Level Semantics Events

Once the relevant execution events are extracted we have a clear view on what has changed in the execution semantics. But still, we needed a translation step to make the analysis results expressed in the high-level semantics. For this step we use an *Event Processor Agent* [15] implemented as a model transformation from low-level analysis results to a high-level set of events. Those events can be easily understood by the user and embed relevant concepts from the high-level design

model. High-level events can constitute a base for reporting the analysis results directly into the user models or for animating a part of the original specification. The two transformation rules in Listings 1.1 and 1.2 show an excerpt of the ATL transformation that abstracts execution events to high level events. The first rule transforms each *IF process state change event* into an *OMEGA state enter event*. The second rule extracts *OMEGA send events* from *IF enqueue events*.

```
StateChangeEvt2StateEnterEvent extends TraceEvent2OmegaEvent {
    from
        sEvt: IFTraceEB!StateChangeEvt
    to
        tEvt: OmegaTraceEB!StateEnterEvent (
            className <- sEvt.instance.type,
            name <- sEvt.instance.pid.name,
            stateName <- sEvt.newState.name,
            oldStateName <- sEvt.oldState.name
        )
}
```

Listing 1.1. ATL rule for extracting *OMEGA object state entering events*

```
rule EnqueueEvt2SendEvent extends TraceEvent2OmegaEvent {
    from
        sEvt: IFTraceEB!EnqueueEvt
    using {
         mes: IFTraceEB!Message = sEvt.messages -> first();
    }
    to
        tEvt: OmegaTraceEB!SendEvent (
            signal <- mes.signalType,
            by <- thisModule.pid2Object(mes."from")
            ...
}
```

Listing 1.2. ATL rule for extracting *OMEGA message sending events*

2.3 Interpreting High-Level Semantics Events

After abstracting execution events we get a set of high-level semantics events. High-level events gather state changes that can be easily understood by the user. The last step in the proposed process is the visual interpretation of the analysis results that is now transformed into high-level events. End user visualizations has to be carefully designed to help the user grasping the event information without visual effort. More on this point can be found in our previous work [4]. Figure 4 shows graphical rendering of *OMEGA state enter event* extracted from *IF process state change event*.

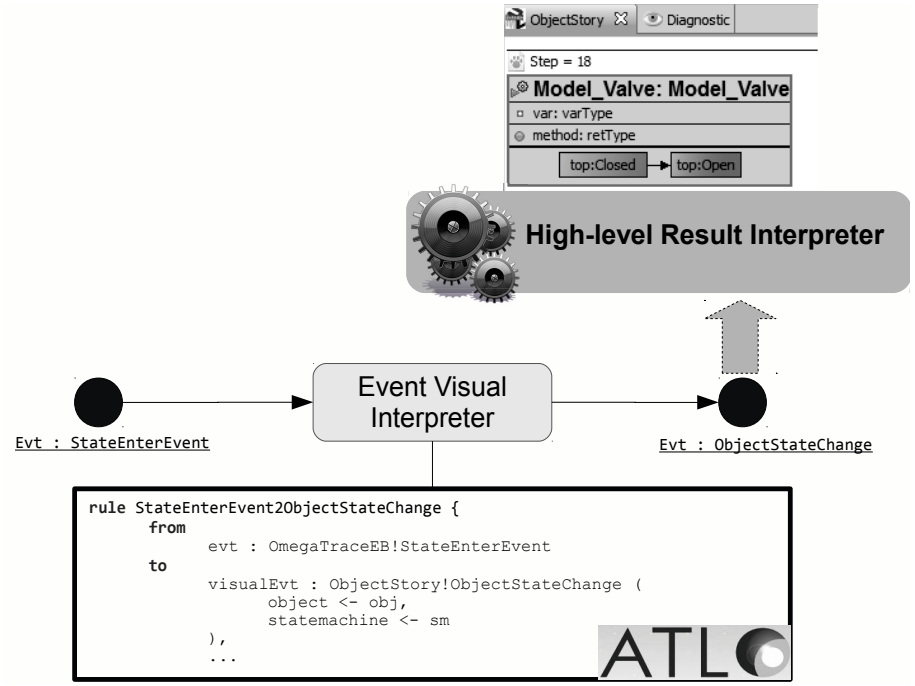


Fig. 4. Graphical rendering of high-level events. The *Event Visual Interpreter* executes a model transformation to generate the input model for the visual renderer.

3 Evaluation

In order to assess the advantages and limitations of the new visualization, we need to evaluate its usage. Several techniques can be used for such an evaluation. Some of these are purely subjective, while others use an objective and quantitative approach [17]. Moreover, evaluation techniques can be categorized according to the stakeholders: techniques such as *cognitive walk-through* or *heuristic evaluation* involve human factors experts, while *observational* or *experimental* techniques rely on user participation.

The premier goal in our study is to make our model analysis tool more accessible to a wider audience. Today, the overwhelming majority of modelers are not familiar with analysis techniques such as model checking. In order to assess whether we have achieved our premier goal, we need to evaluate the approach on a sample of users compliant with this profile. Therefore, due to its objective and quantitative orientation, we decided to use a *controlled experiment* that involves user participation. However, since this technique does not provide a detailed user impression and satisfaction overview, we complement our validation with a *subjective approach*, by means of a questionnaire technique based on the System Usability Scale [18].

3.1 Defining Formal Methods Usability

Before defining the hypothesis and designing an experiment, we need to understand the notion of *usability* in the context of formal analysis techniques. The ISO standard definition [19] for usability, defines it as:

> "the extent to which a system, product or service can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use."

To customize the usability definition in our context, we need to set its three characteristics - effectiveness, efficiency and satisfaction - in the context of formal methods:

- *Effectiveness* is the ability of users to understand, find and correct errors reported by the analysis process.
- *Efficiency* is the time and cognitive effort needed to perform each of the three above-mentioned tasks.
- *Satisfaction* is the subjective impression the users get after using the integrated tool suite that supports the three task.

3.2 Hypotheses Formulation

As mentioned above, our goal is to increase analysis tools usability. To achieve this goal, we propose to enhance tools with visualization techniques. Thus we can formulate the following hypothesis:

H. *Effectively supporting the user in understanding model analysis* ***results****, will increase the analysis tool* ***usability****.*

The notion of *analysis tool usability* was defined in the previous section. Based on this, our hypothesis **H** can be further refined:

H1. Participants using the new tool extension will spent less time understanding the trace.
H2. Participants using the new tool extension will have a better understanding of the trace.
H3. Participants using the new tool extension will spent less time locating the error in the trace.
H4. Participants using the new tool extension will locate the error in the trace with more precision.
H5. Participants using the new tool extension will spent less time understanding the error cause(s).
H6. Participants using the new tool extension will have better understanding of the error cause(s).

These refined hypothesis will allow more insights into the experiment results.

3.3 Experiment Design

In this section we will go further with the experiment design, by defining several of its characteristics. The terminology used in this section is the one defined in [20].

Participants. Participants were chosen from Master and PhD students from the University of Toulouse. The experiment was conducted with 10 subjects distributed in two groups, the experimental group (group A in next sections) and the control group (group B). All the participants were already familiar with UML, two of them have already used a formal analysis tool, but none of them has used ours. In order to assess their adequacy with the user population, participants were asked to fill a questionnaire. The questionnaire has 7 parts and 12 questions such as education level, experience with modeling, and verification techniques abilities. In the first group technical experience median is 4, mean is 4,4 and standard deviation is 1.67. In the second group mean is 4.6 with a median 4, the standard deviation is 1.52.

Experimental Unit. Participants were asked to visualize the execution trace of a small OMEGA UML model representing an Electronic Valve Controller. A timed constraint was set on the model. The OMEGA model is then used as input for the IF Model Checker. Since the model was intentionally violating the constraint, the model checker generated a counterexample. The model was built to get a representative counterexample of what a modeler would get from using the IFx-OMEGA toolbox. Participants will explore this counterexample and perform some well defined tasks.

Experimental Variable. It corresponds to the software product used by the participant to explore the analysis results. We consider the Metaviz [4] tool being the experimental unit. It offers support for IF traces exploration alongside with elaborate visualizations. Figure 5 presents a screen shot of the trace analysis support. The right panel contains the error trace browser, which is similar to what is available in other analysis tools. The middle panel contains the trace visualization feature that we added.

Factors. Also called independent variables, are those we are going to manipulate in the participants environment to see how other variables (response variables) are affected. Our experiment has the goal to analyze how a better support for the users would affect their understanding of the model analysis results. Thus we run the experiment with a two-level factor: Metaviz with elaborate trace visualization support activated and deactivated. Consequently we have the following levels:

level 1: only basic Metaviz views are activated
level 2: advanced visualization is activated

Response Variables. They correspond to the experimental aspects impacted. To investigate their quantitative values we have designed a set of user tasks distributed in three categories. Each category corresponding to a response variable. The categories are related to the following cognitive tasks:

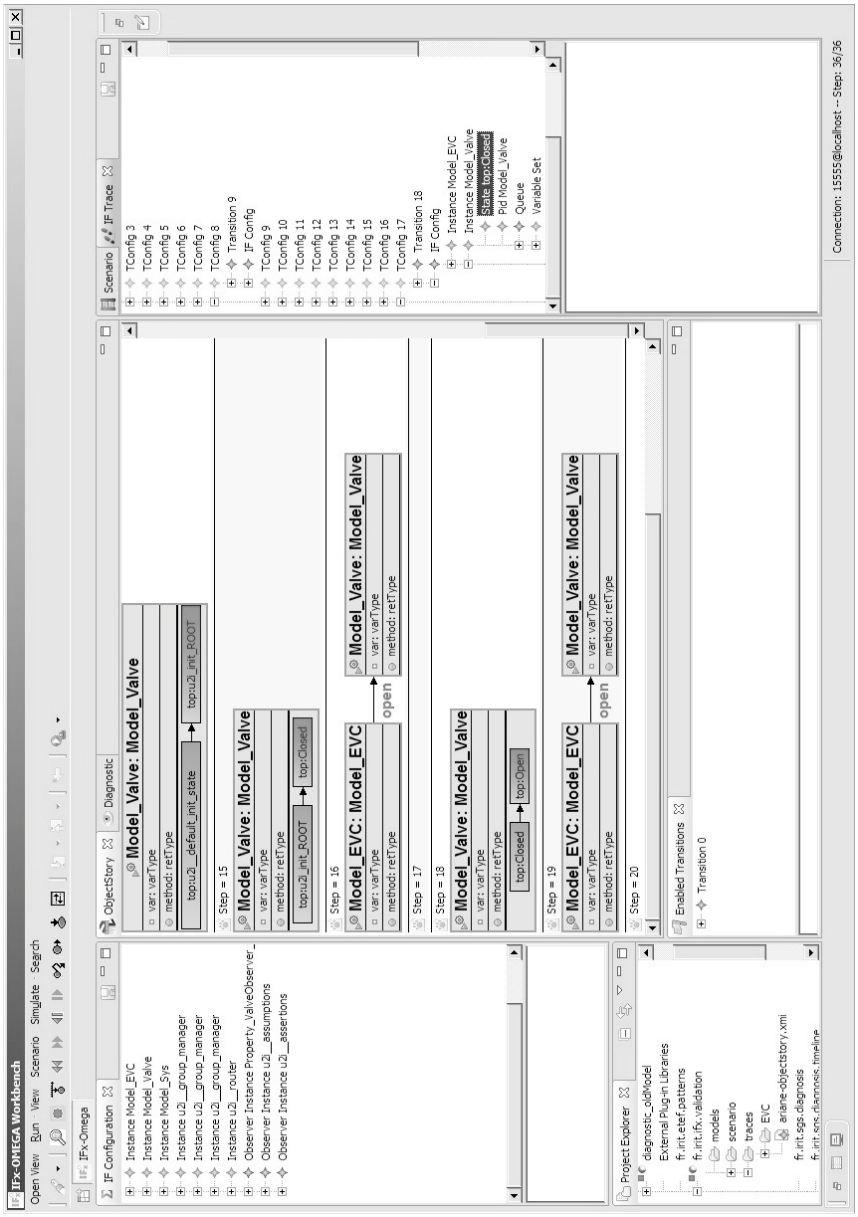


Fig. 5. Metaviz with the full support for trace analysis

– understanding the trace;
– locating the error;
– understanding the error's cause and fixing the model.

For each category we assess two characteristics: the speed and the quality.

Tasks. The set of tasks each participant has to perform was designed to cover the understanding of the trace, the error and its cause. Each participant was given a set of 7 tasks to perform using the trace analysis tool. Additionally, the time spent by each participant performing a task is monitored with an external stopwatch application. We do not take into account the time spent by the participant writing down the answers. The table 1 gives an overview of the tasks.

Table 1. Task per each cognitive category. Participants were asked to perform a set of tasks that spans three cognitive task types.

Cognitive Types	Tasks
Trace Understanding	1. What are the instances created during this execution and at what step
	2. What are the exchanged messages, at what step and between which instances
	3. At which step of the trace, the instance Model_Valve enters the state Closed
Error Locating	4. Find in which step the property is violated
Cause Understanding	5.Which diagram should be modified to satisfy the property ?
	6. Explain informally (in English) what modification you want to do
	7. Fix the error in the model (syntax is free).

3.4 Results

The goal of our experiment is to see whether extending an analysis tool with the event-based visualization mechanism described previously would improve analysis results exploitation. Figures 6 and 7 show the results for each task, in terms of time and success rate. Participants that use the Metaviz extension belong to the group A (experimental group) , while group B (control group) corresponds to participants using the basic version of the tool (i.e. without the Metaviz extension).

Time Spent in Each Task. Results for tasks *T1* and *T2* show that participants in the *group A* perform 5 times faster than the *group B* participants. For task *T3* they perform 8 times faster, while for the last set of tasks *T5* to *T7* they perform 2 times faster. The overall unbalanced improvement rate is

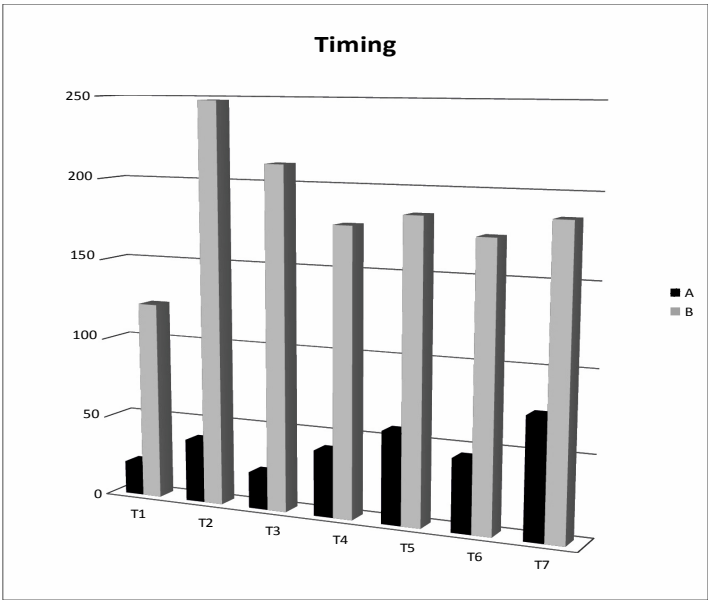


Fig. 6. Time (in secondes) to perform each task. Participants that use an elaborate visualization perform tasks 4 times faster.
A: experimental group, uses an elaborate visualization
B: control group, uses a basic visualization

therefore of 400%. This important increase in task performance speed is due to the cognitive nature of each task category. For instance, one can notice that the experimental group *group A* was 8 times faster in performing the task *T3*. This task is the most demanding in cognitive workload for the participants. Indeed, it asks participants to find the step in the trace where a certain instance enters a particular state. The participant has to recall the instance and state names while he goes step by step through the whole trace. This is where we can see the power of having a visualization that presents only what has changed in the trace. The Metaviz visualizations focus on the state change of the relevant instance and filter other irrelevant information. Figure 5 shows the basic visualization alongside the elaborate one. The visual vocabulary used is highly intuitive and thus enabled the control group to perform the task *T3* 8 times faster.

Quality. Concerning success rates, results revealed an increase of the participant's answers quality. Results for understanding the trace (task *T1* to *T3*) show that understanding of the instances interactions (i.e. message passing) is 25% better. For locating the error (task *T4*) the improvement is about 9%. For the last cognitive category, namely *understanding the cause* the improvement is of 40%. Figure 7 shows the success rate for each task.

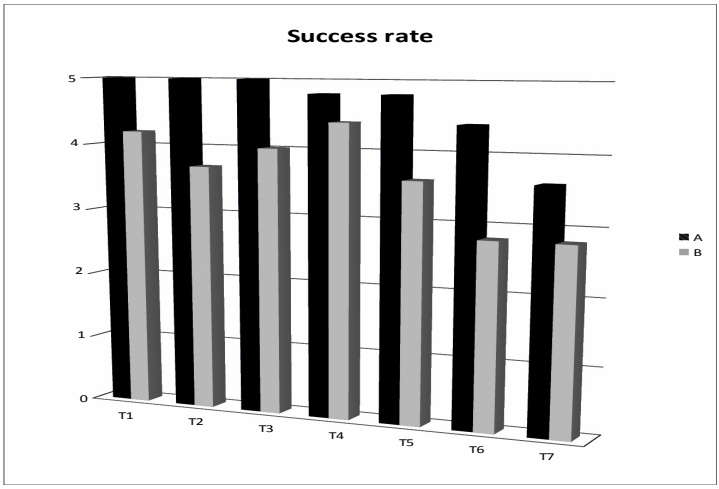


Fig. 7. Success rate for each task.Participants that use an elaborate visualization produce task outputs of a better quality.
A: experimental group, uses an elaborate visualization
B: control group, uses a basic visualization

To complement our analysis we have run a questionnaire based evaluation. This evaluation assessed the user satisfaction. For this purpose we relied on the System Usability Scale test[18]. We have chosen this analysis method since it is lightweight and reliable [21]. Results for user satisfaction are slightly higher for

Table 2. Statistical results by cognitive category. Results show means and Student's t hypothesis tests for each cognitive category of tasks.

	Trace Understanding (T1 to T3)		**Error Understanding (T4)**		**Cause Understanding (T5 to T7)**	
	Success	**Timing**	**Success**	**Timing**	**Success**	**Timing**
Group A	5	44.33	5	18	5	35
	5	27.33	5	21	4.33	55.67
	5	27	5	59	5	30.67
	5	8.33	5	51	3.33	62.67
	5	29.67	4	55	3.67	110
Group B	5	103	5	20	3	100
	3.17	325	5	540	0	600
	5	183.33	5	27	5	90.33
	5	120.67	5	32	5	23.33
	1.67	238	2	267	2.33	94.33
t	**0.16**	**0.004**	**0.54**	**0.22**	**0.26**	**0.28**

the experimental group (*group A*) with 68% versus 61% for the control group (*group B*). The results presented in table 2 show an increase of about 11% of the user satisfaction.

3.5 Hypothesis Tests

The null hypothesis corresponding to each hypothesis formulated previously can be summarized by *there is no difference between participants that use the elaborate visualization and those who use the basic one in performing the tasks from the category CTx*. Were *CTx* is one of the three cognitive task categories listed in 1. As mentioned previously task performance is measured with the speed and quality of three cognitive task types (i.e. understanding a trace, locating an error, understanding its causes). As we describe in section 3.3 the standard deviation can be considered equal and we can test the null hypothesis using the Student's t test. We applied a two-tailed Student's t test to the six null hypothesis. The results are showed in table 2. The experiment was run with five participants in each group, that gives us 8 degrees of freedom and a value for $t_{0.99}$ of 3,355. All the values of t are under the value of $t_{0.99}$, we can then reject the null hypothesis. For the user satisfaction results showed in table 3 are also statistically sound (t value of 0,7) and shows that the null hypothesis corresponding to user satisfaction can also be rejected. Initial hypotheses are thus accepted, meaning that *effectively supporting the user in understanding analysis results increases analysis tool usability.*

Table 3. Usability tests using the System Usability Scale[18]. Results show an increase of 11% in usability for the experimental group (group A). Results are statistically sound according to the Student's t test.

	System Usability Scale
Group A	67.7
Group B	61.0
t	**0.71**

3.6 Threats to Validity

Threats related to the conclusion. We have used statistical test, namely Student's t test. Our samples has the same number of individuals and similar variances which ensure the soundness of the test conclusions. *Construct threats.* The role of the trace abstractions is to reduce the amount of data to a relevant set of information for a specific user task, this avoid the cognitive overload. The visualization has a different role. It brings to the user these relevant information in a domain oriented way, that is, in terms of concepts already known by the user. No additional effort is needed from the user to understand the notations and semantics of the visualization constructs (perception overload). The visualization alone is not enough, this is why the control group (that uses a basic visualization of the low-level traces) has the worst results. The huge amount of

data gathered to the user in a basic view (without abstraction) is cognitively demanding and time consuming. *Internal threats*: The statistics shown in the section 3.3 ensures that there's is no significant difference in the participants technical level. *External threats*: While used with a small model, the technique is more likely to accelerate user exploring and understanding for bigger models. We think that for experts, the results may not be as significant as for users with average technical background. But we should be careful of this generalization, the *expertise reversal effect* as coined by Sweller [22] may arise. Thus, further experiments should be conducted to assess the results for expert users.

4 Related Work

Early work in analyzing UML models like [8,9] set among others the foundations of model-driven analysis. Recent work on the MARTE profile [11] introduces the use of user feedback in an integrated MDE approach. Results of MARTE [23] models analysis is used to annotate back the original user models. The RT-Simex research project [24] tackles the problem of user feedback with an elaborate user interface. This is very similar to our approach but the work focuses on clock reconciliation between independent traces and no controlled experiment is conducted to assess the effectiveness of the user feedback. In the work of [25] an emphasis is put on the lack of user friendly interfaces in model checkers. They effectively address the problem of understanding analysis results but directly at the low level semantics. A broader view to the issue of designing SE notations is addressed by Moody [26]. He emphasizes the lack of foundational work on the syntax of software engineering visual notation. Combemale et al. adressed the problem of extracting high-level traces from low-level executions in [27]. They show how to extract high level trace from low level traces. They also assume an existing execution model for the high-level formalism and editors. Our work emphasize the importance and the effectiveness of the user feedback but go further by validating the added value in a controlled experiment.

5 Conclusion and Future Work

Today's integrated development environments offer many debugging facilities, that allow the developer to follow closely the code execution, to debug it and to understand it. While modeling languages aim at raising the level of abstraction in software development and could support much more powerful analysis techniques, the tools existing to support them are still difficult to use [28]. Using languages such as UML, SysML, SDL, to model the software, performing model-based analysis on these models and understanding the analysis results is still a challenging task. Improving the existing tools and mechanisms for exploring and understanding model based analysis results is greatly needed for a larger adoption of formal methods. This paper presents our contribution in this direction. Our approach provides a semi-automatic technique to implement a feedback mechanism in a SysML/UML translational semantics approach. Based

on an existing SysML/UML tool that offers the possibility to perform verification, we develop an advanced and flexible trace analysis support mechanism. As described in this paper, this mechanism allows the analyst to reason at model level on the model execution, in terms of easily understandable high-level events. The pertinence of our approach is assessed through an evaluation, based on well established evaluation mechanisms (Usability Scale System, ISO notion of usability, etc). In order to perform such an evaluation, we needed to adapt the notion of usability to the context of formal methods usability, and to adapt the evaluation process to our setting. The goal of this experiment was to see whether extending analysis tools with a well designed event-based visualization would improve analysis results exploitation and the results are meeting our expectations. Several directions could be taken for future work. Beyond improving the existing approach/toolset, we intend to add new visualization techniques, based on specific tasks to be performed during the analysis (variable change impact, queue size evolution, etc), to identify new kinds of user profile based visualizations that may assist the user, depending on its level of expertise, to perform new experiments with different user profiles, etc. We strongly believe that by facilitating the access to analysis tools by regular users, the interest in using modeling techniques could significantly increase and the much advertised advantages of these techniques could finally get accessible to a larger panel of users.

Acknowledgments. We thank Anke Brock and Antonio Serpa for their support, the volunteer participants to the experiment, and the reviewers for their valuable suggestions.

References

1. Object Management Group: Unified Modeling Language, `http://www.uml.org/`
2. International Telecommunication Union: ITU-T Recommendation Z.100 (12/11) – Specification and Description Language – Overview of SDL-2010 (2011), `http://www.itu.int/rec/T-REC-Z.100-201112-I/en`
3. SAE International: SAE Architecture Analysis & Design Language (AADL) AS5506 Rev.B (2012), `http://standards.sae.org/as5506b/`
4. Aboussoror, E.A., Ober, I., Ober, I.: Seeing Errors: Model Driven Simulation Trace Visualization. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds.) MODELS 2012. LNCS, vol. 7590, pp. 480–496. Springer, Heidelberg (2012)
5. Lilius, J., Paltor, I.P.: vUML: A tool for verifying UML models. In: 14th IEEE International Conference on Automated Software Engineering (ASE 1999), pp. 255–258. IEEE Computer Society (1999)
6. Holzmann, G.J.: Design and Validation of Computer Protocols. Prentice Hall (1991)
7. Ober, I., Dragomir, I.: OMEGA2 – A new version of the profile and the tools. In: 15th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2010), pp. 373–378. IEEE Computer Society (2010)
8. Bozga, M., Graf, S., Ober, I., Ober, I., Sifakis, J.: The IF toolset. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 237–267. Springer, Heidelberg (2004)

9. Espinoza, H., Dubois, H., Gérard, S., Medina, J.L., Petriu, D.C., Woodside, C.M.: Annotating UML Models with Non-Functional Properties for Quantitative Analysis. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 79–90. Springer, Heidelberg (2006)
10. Conquet, E., et al.: Formal Model Driven Engineering for Space Onboard Software. In: Embedded Real Time Software and Systems (ERTS2) (2012), http://www.erts2012.org/Site/0P2RUC89/7B-4.pdf
11. Mraidha, C., Tucci-Piergiovanni, S., Gerard, S.: Optimum – a MARTE-based Methodology for Schedulability Analysis at Early Design Stages. SIGSOFT Software Engineering Notes 36(1), 1–8 (2011)
12. Ober, I., Graf, S., Lesens, D.: Modeling and Validation of a Software Architecture for the Ariane-5 Launcher. In: Gorrieri, R., Wehrheim, H. (eds.) FMOODS 2006. LNCS, vol. 4037, pp. 48–62. Springer, Heidelberg (2006)
13. Ober, I., Graf, S., Yushtein, Y., Ober, I.: Timing Analysis and Validation with UML – the case of the embedded Mars bus manager. Innovations in Systems and Software Engineering 4(3), 301–308 (2008)
14. Official reference MARTE Tutorial, http://www.omg.org/omgmarte/Tutorial.html
15. Etzion, O., Niblett, P., Luckham, D.: Event Processing in Action. Manning (2010)
16. Lin, Y., Gray, J., Jouault, F.: DSMDiff – a differentiation tool for domain-specific models. European Journal of Information Systems 16(4), 349–361 (2007), http://dx.doi.org/10.1057/palgrave.ejis.3000685
17. Dix, A.: Human-Computer Interaction. Pearson/Prentice-Hall (2004)
18. Brooke, J.: SUS – A quick and dirty usability scale (1996), http://hell.meiert.org/core/pdf/sus.pdf
19. International Standards Organisation: ISO 9241-11:1998 Ergonomic requirements for office work with visual display terminals (vdts) part 11 – Guidance on usability, http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=16883
20. Juzgado, N.-J., Moreno, A.-M.: Basics of Software Engineering Experimentation. Springer (2001)
21. Stanton, N., et al.: Human Factors Methods – A Practical Guide for Engineering and Design. Ashgate (2005)
22. Sweller, J.: Evolution of human cognitive architecture. Psychology of Learning and Motivation 43, 215–266 (2003)
23. The UML Profile for MARTE, http://www.omgmarte.org/
24. DeAntoni, J., et al.: RT-SIMEX – Retro-analysis of execution traces. In: Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2010), pp.377–378. ACM (2010)
25. Yokoyama, S., Sato, H., Kurihara, M.: User-friendly GUI in Software Model Checking. In: Proceedings of the 2009 IEEE International Conference on Systems, Man and Cybernetics (SMC 2009), pp. 468–473. IEEE Press (2009)
26. Moody, D.L.: The "Physics" of Notations: Toward a scientific basis for constructing visual notations in software engineering. IEEE Transactions on Software Engineering 35(6), 756–779 (2009)
27. Combemale, B., Gonnord, L., Rusu, V.: A Generic Tool for Tracing Executions Back to a DSML's Operational Semantics. In: France, R.B., Kuester, J.M., Bordbar, B., Paige, R.F. (eds.) ECMFA 2011. LNCS, vol. 6698, pp. 35–51. Springer, Heidelberg (2011)
28. Hutchinson, J., et al.: Empirical Assessment of MDE in Industry. In: Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011), pp. 471–480. ACM (2011)

Modeling Component Erroneous Behavior and Error Propagation for Dependability Analysis

Naif A. Mokhayesh Alzahrani and Dorina C. Petriu

Department of Systems and Computer Engineering, Carleton University
Ottawa, Ontario, Canada K1S 5B6
{nzahrani,petriu}@sce.carleton.ca

Abstract. Modeling erroneous behavior of software components along with normal behavior tends to be complex and hard to read or modify. However, ignoring the erroneous behavior and error propagation in models used for dependability analysis has a negative impact on the dependability assessment accuracy. In this paper, we propose a framework for automating dependability modeling and analysis that considers component erroneous behavior. Particularly, the paper focuses on our Component Erroneous Behavior Aspect Modeling approach (CeBAM), which captures component erroneous behavior and error propagation. We apply aspect-oriented modeling techniques to model erroneous behaviors separately from the normal behavior. The approach reduces the model complexity and improves its readability and modifiability. In addition, we propose a profile to extend the UML protocol state machine to capture both incoming and outgoing messages on components' ports. We automate the composition of normal and erroneous behavior by aspect weaving. This enables the next step: conformance verification between each component's complete internal behavior and its protocol state machines, as well as between component interfaces.

Keywords: erroneous behavior model, error propagation, aspect-oriented modeling, conformance verification.

1 Introduction

Model Driven Development (MDD) is a promising approach for software development that changes the focus from code to models. This change of focus facilitates also the analysis of different Non-Functional Properties (NFP) using formal analysis models obtained by model transformations from the software models. For instance, in this paper we are interested in the analysis of dependability attributes (such as reliability and availability) using analysis models automatically generated from software models extended with dependability annotations. In [1] the authors survey the works on dependability modeling and analysis based on UML and UML extensions for annotations. Another software development paradigm of interest is Component Based Development (CBD), which applies the "divide and conquer" principle to manage system complexity. Each component is a unit of composition that interacts with other components through

F. Khendek et al. (Eds.): SDL 2013, LNCS 7916, pp. 124–143, 2013.

predefined interfaces. CBD is a reuse-based approach that has an impact on the development time and system dependability attributes.

Combining MDD and CBD is an appealing approach for the development of real-time embedded systems, as it reduces the complexity, time and cost. In addition, MDD and CBD help to integrate dependability modeling and analysis during the design phase. The quantitative results of these analyses will support the developer in taking the right decisions for building dependable systems. Different approaches were proposed in literature to address reliability and availability modeling [1,2]. However, many existing approaches do not adequately consider error propagation in predicting system reliability [3].

The long-term goal of our research is to propose a framework based on standard modeling languages (such as UML and QVT), which would help developers to evaluate dependability attributes during a CBD + MDD process, taking into consideration component erroneous behavior and error propagation. We believe that including component erroneous behavior in dependability analysis and prediction will help developers to take the right design decisions. For instance, selecting proper fault tolerance mechanisms, placing error detection and using suitable recovery approaches are examples of critical decisions taken in the design phase based on quantitative values. The findings of [4,5] support this belief, since they show that error propagation may have significant impact on reliability prediction. Thus, in our approach the evaluation of dependability attributes is based on component behavior (normal and erroneous).

A software component has two views: internal and external. An internal view represents component's private properties realizing the provided services. An external view shows the public properties of the component in terms of provided and required interfaces. Modeling erroneous behavior of these views along with normal behavior in one model tends to be complex and hard to read or modify. Moreover, it is not easy to capture error propagation between components using existing behavior models such as UML2 state machines. As a result, developers often focus on the normal behavior of both views and tend to ignore the erroneous behavior.

To overcome these difficulties of modeling erroneous behavior and error propagation in CBD, we introduce the Component Erroneous Behavior Aspect Modeling approach (CeBAM). It captures the component erroneous behavior separately from the normal behavior. CeBAM uses aspect-oriented modeling [6] to simplify modeling the erroneous behavior and to automate its composition with the normal behavior.

CeBAM uses UML state machines to represent the component internal normal behavior and extended protocol state machine for port and interface behavior. Normally, UML protocol state machines capture only incoming messages, so we defined a profile to capture both incoming and outgoing messages. Another UML extension developed in this paper is an erroneous behavior profile to capture the chain of dependability threats for the component internal behavior, as well as for its ports and interfaces. In addition, this profile shows the error propagation from the component internal behavior to its ports and further to other components.

One of the CeBAM advantages is that it provides an easy and practical way to model component erroneous behavior separately from the normal behavior, using aspect-oriented modeling [6]. Indeed, this reduces the state machine complexity and makes the model easy to read and maintain. In addition, the automated composition of erroneous with normal behavior will be further used for: a) conformance verification between the internal behavior of each component and its protocol state machines, and between components interfaces, and b) derivation of dependability analysis model.

This paper is organized as follows. Section 2 presents our long-term objective to automate dependability modeling and analysis. In section 3 we explain briefly the Emergency Monitoring System (EMS) case study used in this paper. The CeBAM approach is introduced in section 4. Related work is discussed in section 5. In the last section, we conclude and summarize our ongoing work.

2 Overview of Dependability Analysis Framework

The long-term objective of our research is to provide software developers with automated techniques for architectural-based software dependability modeling and prediction. This paper is the first step on the road toward such an objective. Figure 1 illustrates the overall activities of our proposed framework in order to provide context for the contribution of this paper and to put it into perspective. We start with a component-based software architecture model that needs to be evaluated in terms of reliability and availability. For the most important scenarios, we identify the involved components and the interaction between them. Next, we build component behavioral models using CeBAM that considers erroneous behavior of the involved components. This model is also enriched with dependability annotations using the DAM profile [7]. Note that we do not show the dependability annotations in our case study because of the limited space. Aspect oriented modeling (AOM) approach [6] is adopted to allow for more flexibility in modeling erroneous behavior and to provide an automated composition of the erroneous with the normal behavior.

Reasoning on behavioral compliance of a component-based software architecture is required to validate the software architecture [8]. Thus, in our approach, after composing the normal and erroneous behavioral models, we validate the conformance in two stages. The first is conformance validation between component internal behavior and its protocol state machines. A mismatch would impact negatively the component reliability, since the internal behavior would receive unexpected messages that cannot be handled. So, any detected mismatch must be corrected. Second, we verify the compatibility between the components' provided and required interfaces.

In the literature, different approaches are suggested to check for component conformance by finding deadlocks in the formal model that is transformed from the main software model [9,10]. Different formalisms may be used, either various kinds of formal logic or of Petri nets. Since we are planning to use state-based dependability analysis based on Petri Nets, similar to the approach in [11], we

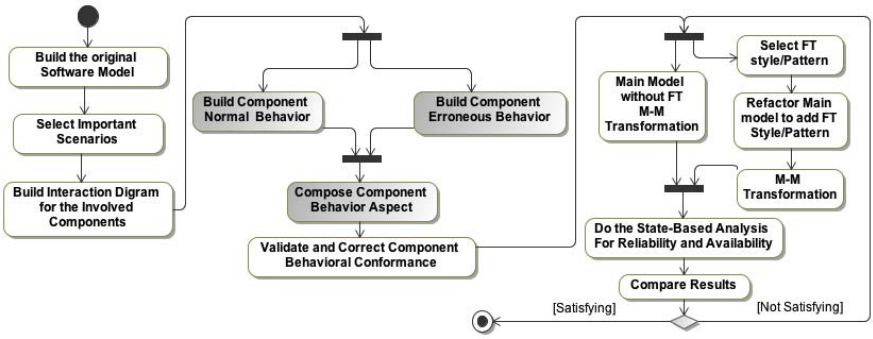


Fig. 1. Overview of the proposed framework (Shaded activities are the focus of this paper)

will also use a class of Petri Nets, namely Stochastic Reward Net (SRN) [12] for conformance validation instead of a model checker. We chose SRN because its marking dependency property helps in obtaining more compact models for complex systems, which helps in limiting the size of the state space during the analysis. Currently, we are working to automate the transformation of component behavioral model to SRN in order to validate conformance and predict reliability and availability as well.

Adding a fault tolerance mechanism may improve the system reliability, but the effects are non-trivial and depend on the context [13]. As shown in Fig. 1, our approach aims to provide developers with automated tools to assess the reliability (availability) of different fault tolerance mechanisms applied to the system under evaluation. By comparing the predicted results, a developer can select the best design alternative.

Automation is one of the key features in our approach. We utilize Query View Transformation (QVT) [14] to automate the composition of component erroneous behavior with the normal behavior (based on aspects) and to generate SRN models for conformance checking. Also, the process of refactoring the main architectural model by adding a selected fault tolerance mechanism from a predefined collection of fault tolerance styles (see Fig. 1) can be automated with the help of QVT model transformations. Moreover, automated model transformation will generate the SRN model used for state-based dependability analysis for the architecture without and with fault tolerance mechanisms. Comparing the results, the developer will be able to evaluate the effects of the selected fault tolerance style on the overall system reliability and availability.

3 Case Study

Emergency Monitoring System (EMS) is the case study that will be used throughout this paper. This case study was developed in [15] and our goal is to improve the design by modeling the normal and erroneous behavior of each

component. The erroneous behavior will capture any locally activated fault and show how it propagates to the connected components through its ports.

The EMS is a distributed system. It consists of a central monitoring service, operator presentation service and several distributed monitoring sensors. In addition, in some remote areas, a remote monitoring system is installed with its sensors. All the remote systems and distributed sensors are reporting regularly the current status of the external environment to the central monitoring service. The monitoring service stores and updates the recent status and presents it to the operator. As a result, the operator will have the updated status of each site and, accordingly, he can take action if the emergency alarm is detected.

COMET methodology [15] was used in this case study. Figure 2(a) shows the use case model and Fig. 2(b) the distributed component-based software architecture of EMS. Due to limited space, we select only one scenario called *generate alarm* to illustrate our approach for modeling the component behavior. In this scenario, three components are involved. Figure 2(c) shows the interaction between these components in case of generating an alarm. In general, component interaction takes place either via method calls (synchronous) of provided and required interfaces or via notification signals (asynchronous). According to [16], if an internal fault is activated but not properly handled inside the component, then this fault will end up in a failure, which will propagate to other components that depend on it. For instance, in the selected scenario, if the *AlarmService* component has failed due to an internal exception or hardware failure, the manifested failure will be propagated to the monitoring sensor (see Fig. 2(d)). Moreover, the detected emergency alarms cannot be reported to the operator since the core component *AlarmService* is down.

4 Component Erroneous Behavioral Aspect Modeling (CeBAM) Approach

A software component has two views: internal and external. The internal view represents component's private properties realizing the provided services. The normal behavior of this view can be described using UML behavioral state machine (BSM) [17]. An external view shows the public properties of the component in terms of provided and required interfaces. Interactions between components are actually methods calls (synchronous) or exchanging notification messages (asynchronous). Protocol state machine (PSM) can be attached to each interface to describe the legal sequence of operations calls [17].

A fault may be activated inside a component and then propagated to the interfaces and then to all dependent components if it is not handled internally or at the component port. Moreover, each fault type may have a different propagation path. In fact, modeling both the internal and external view of the component erroneous behavior will help to improve the software design. Unfortunately, BSM and PSM do not allow for an easy and practical way of modeling normal and erroneous behavior, due to model complexity and a lack of ability to capture error propagation.

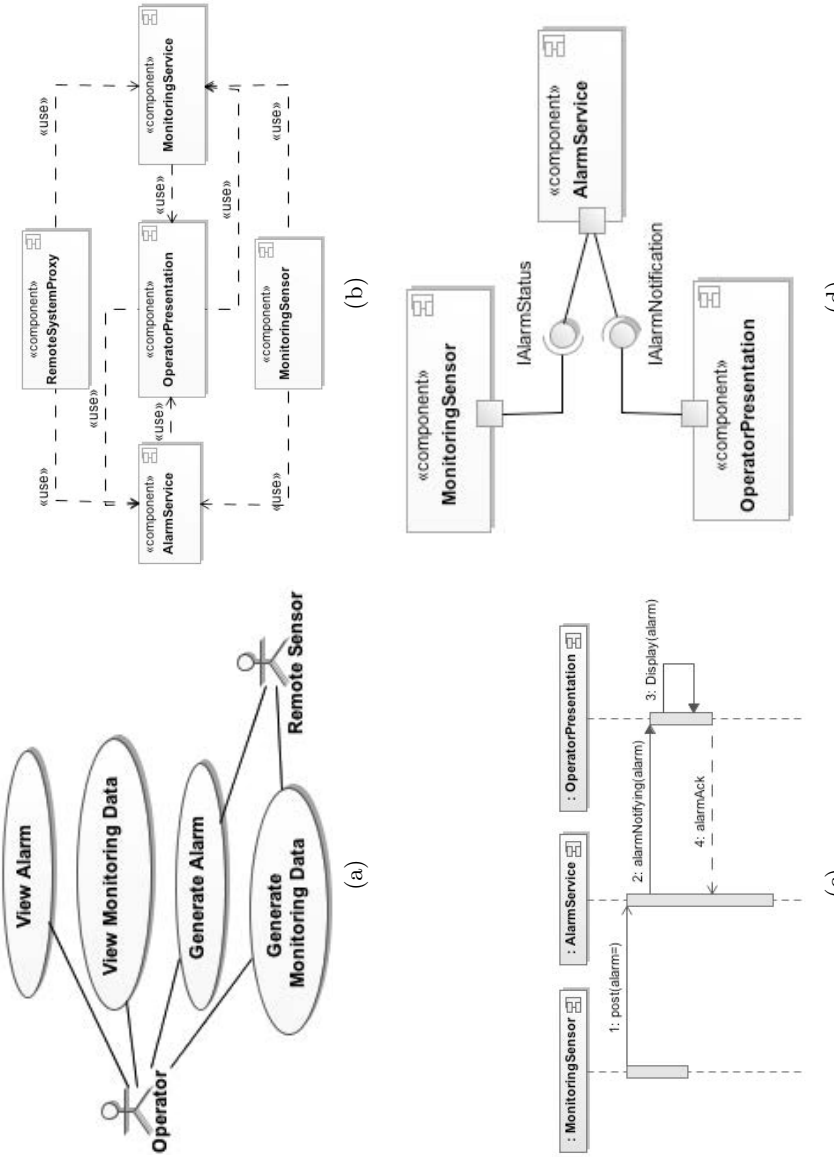


Fig. 2. EMS case study [15], (a) Use cases, (b) Component for the complete system, (c) Interaction of the generate alarm scenario, (d) Components involved in generate alarm scenario

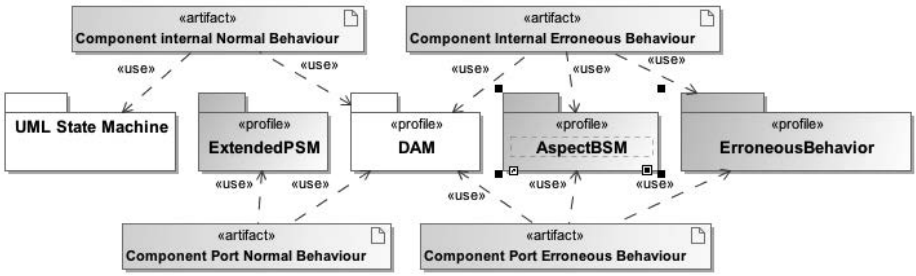


Fig. 3. Profiles and artifacts in CeBAM

In CeBAM, component behavior consists of normal and erroneous behavior that describe both component interfaces, ports and internal behavior. In [1,2] different approaches are presented for dependability modeling and analysis, and we noticed that most of these approaches focus only on the normal behavior, ignoring the erroneous behavior due to its complexity.

Our objective in this paper is to provide a practical solution to model complete component behavior for the internal and external views, by considering erroneous behavior and error propagation. The CeBAM approach is developed to realize this objective. Figure 3 shows the new profiles and artifacts used in CeBAM. For all defined profiles we followed the approach from [18], which suggests to start with defining the domain model as a starting point and then mapping the domain model concepts to the UML2.x meta-model, in order to identify the new stereotypes and attributes.

In CeBAM the internal component behavior is modeled using BSM according to the provided and required services. For instance, Fig. 4 shows the internal normal behavior for a single service of *AlarmService* component in the case study. Normal behavior of component ports or interfaces is modeled using extended PSM (as described in section 4.1).

The ErroneousBehavior profile and AspectBSM are used together to model the erroneous behavior (both internal and external views) separately from the normal behavior as aspect models. Then we automate the composition of the erroneous behavior with the normal behavior of both views (sections 4.2, 4.3 and 4.4). Since we consider also protocol state machines, we validate the conformance between component behavior and its PSMs, as well as between components' interfaces (section 4.6).

In CeBAM we adopt the aspect oriented modeling approach [6] to model component erroneous aspect. Actually, we consider erroneous behavior as a crosscutting concern that can be modeled separately and then we automate the composition with the base model (i.e., the normal behavior for both views). Using this approach, a developer will not have to learn new concepts in order to model the erroneous behavior with the dependability annotations. Additionally, there is full flexibility to update or change any behavior separately, since the composition of the complete behavior is automated.

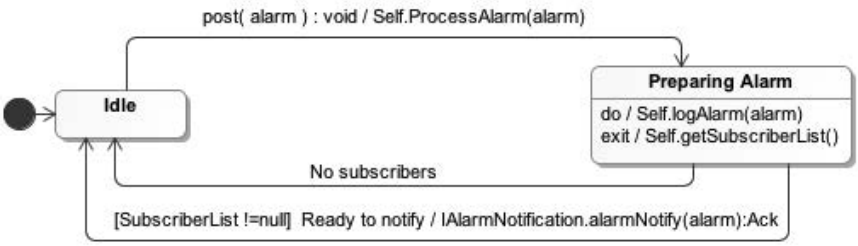


Fig. 4. Partial BSM of internal behavior of *AlarmeSerivce* Component

4.1 Extending Protocol State Machine

According to [17] a PSM is a specialized behavioral state machine defined in the context of a classifier, that can be used to specify which operations of the classifier can be called in which state and under which conditions. PSM is used to describe only the legal usage of any classifier. PSM does not show any specific behavioral implementation since actions are not allowed on transitions or in the states. Actually, states in PSM do not have entry, exit, do activities. On the other hand, composite state and concurrent regions are allowed, but history pseudo-sates are not. (We are not using composite states or concurrent regions in this paper).

A protocol transition captures the legal transition of the context classifier. It has a pre-condition, a trigger, and a post-condition. Protocol transition shows that the associated operation can be called under a specific condition (pre-condition) and then after the complete execution the destination state can be reached if the post condition is satisfied. Moreover, PSM inherits run-to-completion semantics from BSM, i.e. the action on a transition is uninterruptible. This implies that no other event can be accepted during the transition. For instance, if a fault is activated during the execution of the called operation, the transition will not be completed. Additionally, nested calls cannot be captured.

Due to the restrictions applied to PSM, only unidirectional communications can be captured [8,17]. For instance, in Fig. 2(d) we can use PSM to model the communication of the provided interface connected to the *AlarmService* component. In this case PSM can only capture incoming calls to that interface. Moreover, for each interface we have to create a separate PSM since it can only capture the communication on a single classifier.

In UML, a port is a property of a classifier [17]. A port can be associated with a component (i.e., the UML classifier) to specify an interaction point between the components and its environment and between component and its internal parts. A combination of required and provided interfaces can be associated with a port; thus, a port may specify provided services to other components as well as required services. In the case study, we assume that each component has only one port for interaction with the environment; thus, the external view of the component is actually the port behavior.

We can describe the external view of the component behavior by describing its protocol state machine. As mentioned before, PSM can be used to precisely capture that behavior, but it captures only a single direction (incoming) and it does not allow recursive calls due to the run-to-completion semantics. To overcome these limitations, we extend the PSM by introducing a new profile as shown in Fig. 5. This profile will be used to model "extended" protocol state machines. *PortTransition* stereotype is extending the *ProtocolTransition* metaclass with different attributes.

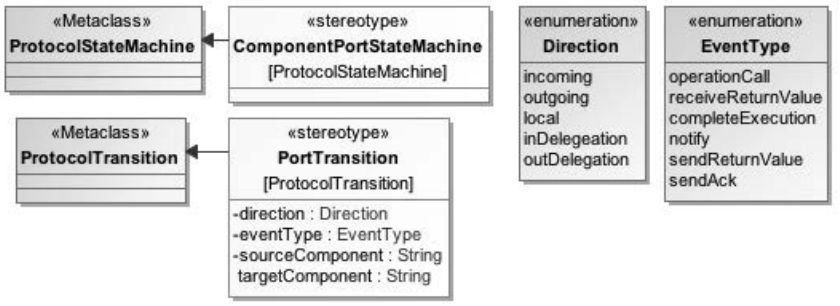


Fig. 5. ExtendedPSM profile

In *PortTransition* we can capture the direction of each passed message, either sent to an associated component or received from the environment. Sometimes the PSM state changes because of an internal event.

In this profile we respect the run-to-completion semantics and we can show atomic events. For each event in *PortTransition* we specify the direction (incoming, outgoing and delegation) and the type of that event (operation call, notify signals, receive return value from the called operation locally or externally and complete execution signals). Moreover, we show the source and target component associated with that event.

The *AlarmService* component of the EMS case study has two provided interfaces and one required interface. These interfaces are associated to a single port (see Fig. 2(d). To describe the external view of the component, we use UML and the ExtendedPSM profile to model the protocol state machine (see Fig. 6). Initially, it receives incoming message from the monitoring sensor to execute one of its provided services *post(alarm)*. The *post* method is implemented by the *AlarmService* component and therefore, different actions will be done internally, for instance, storing the reported alarm and then notifying the operator. These actions will change the state of the PSM, as precisely captured using the ExtendedPSM profile. In Fig. 6, each transition of the alarm service PSM is atomic and it has run-to-completion semantics. In addition, the direction and the source of the messages are also captured.

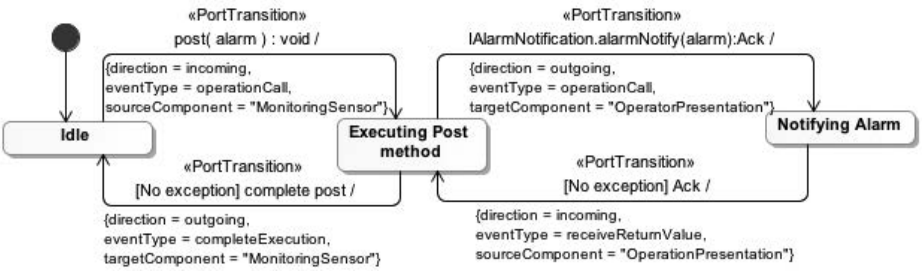


Fig. 6. Fragment of *AlarmService* normal behavior protocol state machine

4.2 Aspect Composition

We adopt AOM mechanism [6] to model separately component erroneous behavior and then to compose it with the normal behavior. Figure 7 shows the domain model for the proposed AspectBSM profile and Fig. 8 shows the actual profile according to the domain model.

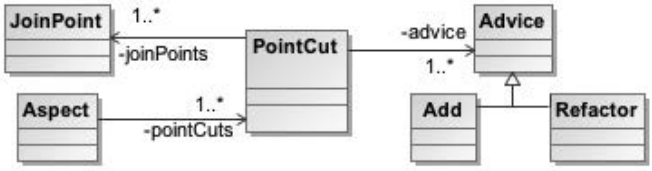


Fig. 7. Aspect domain model in erroneous behavior context

This profile is based on the main concepts of aspect-oriented modeling and is similar to the approach from [19]. *Aspect* describes a crosscutting concern; in our context, the aspect will be the erroneous behavior of both component views. For each crosscutting behavior we have a *pointCut*, which is a condition of a query that identifies the place(s) where the new behavior should be added in the base model or which model element needs to be refactored. A candidate element in the base model that corresponds to a *pointCut* is called *joinPoint*. In other word, the *pointCut* will select one or more *joinPoint* where the new behavior can be applied. In our approach, the *pointCut* will be an OCL query that selects states or transitions. Note that we will not add any new stereotype in the base model to identify the *joinPoint*. *Advice* is a new behavior introduced to the base model at the *joinPoint* and it could be *add* or *refactor advice*.

As mentioned before, transitions in behavioral state machines have run-to-completion semantics according to [17]. In some cases, the action associated with a transition is an operation call. The operation must be executed successfully before entering the new state. However, during its execution, faults maybe activated that will interrupt the transition. Our objective is to model any fault

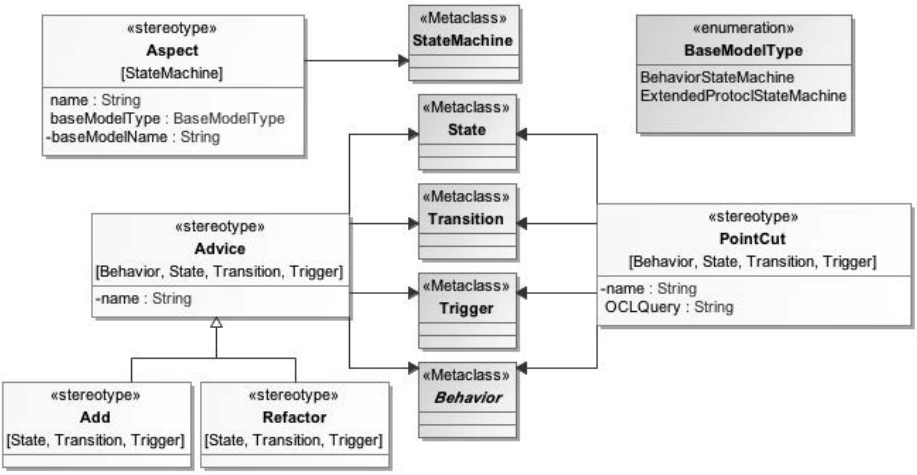


Fig. 8. AspectBSM profile

that may be activated during the transition, but at same time we want to respect the run-to-completion semantics. To achieve that, we introduce a *refactor* advice applied to the respective BSM transition. Before adding the erroneous aspect of that operation we should introduce a new state called *intermediate state* and a new transition called *done*. Figure 9 illustrates an example of the *refactor* aspects applied to the internal behavior of Alarm Service component.

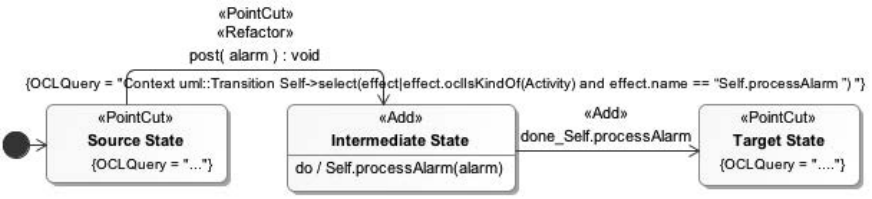


Fig. 9. Refactor aspect of *processAlarm* activity

For instance, *processAlarm(alarm)* is an operation executed as an effect of the transition from the state *Idle* to *PreparingAlarm* in the base model (see Fig. 4). First we identify the source and target states of that transition and then we add a new state (called *IntermediateState*) reached from the source state with the original transition, but without the call to the effect operation. We move the *processAlarm(alarm)* operation to the newly introduced state as a *do activity*. Finally, we add a new transition *done* from the *IntermediateState* to the target state. This new transition represents the successful execution of the transition action from the base model. In this way, we preserve the run-to-completion semantics and we can add later an erroneous transition from the *IntermediateState*.

We use the *refactor* aspect only in BSM describing the internal behavior of a component, but we do not need refactoring in the protocol state machines developed with ExtendedPSM because all transitions are already atomic.

4.3 Modeling Component Erroneous Behavior

Different error states and failure modes can be identified for a single component. Each failure may have a different propagation path. Our objective is to model the error propagation between components separately as crosscutting concerns in order to study how this propagation impacts the overall reliability and availability of the system. We develop a new profile to model the component internal and external erroneous behavior, as well as the error propagation between components. Figure 10 shows the profile stereotype and attributes. This profile captures the two kinds of states and transitions: error, failure mode states and erroneous, recovery transitions. For each transition type it depicts the direction, event, and source operation and target operation. Using this profile we can model different kinds of failure caused by software exceptions or hardware failures. In CeBAM we use this profile and AspectBSM profile together to model erroneous internal component behavior and PSM erroneous behavior as aspects, separately from the normal behavior. This approach will allow for flexibility in the modeling of erroneous behavior, creating models easy to read and modify.

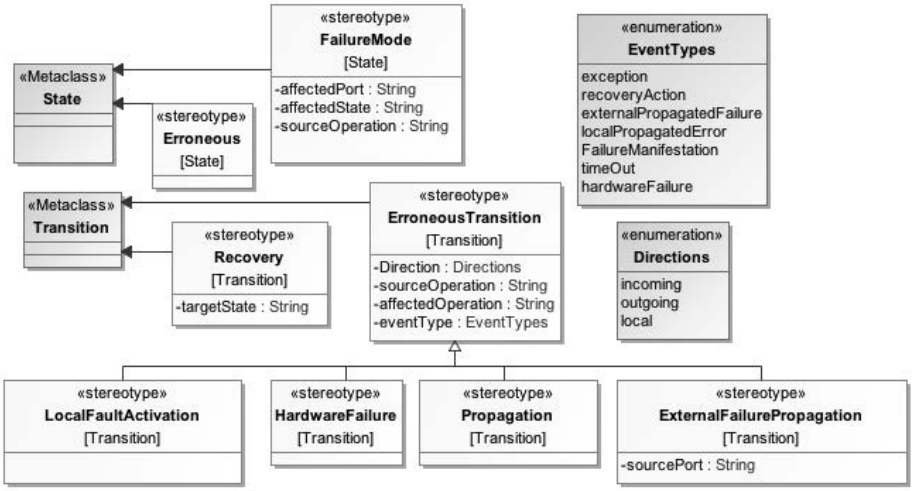
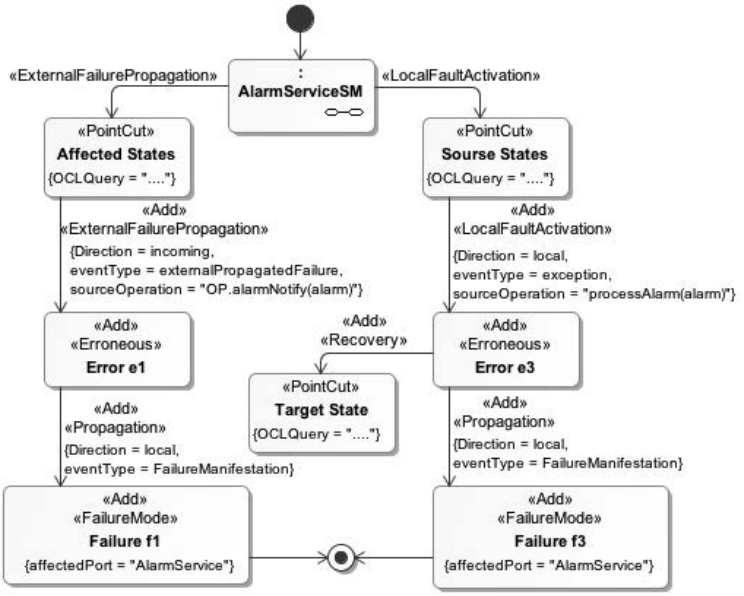


Fig. 10. Erroneous Behavior profile

For instance, Fig. 11 (a) shows different errors and failure modes activated internally in the *AlarmService* component. A local failure is activated because of an internal exception occurring in *processAlarm(alarm)* method and an external failure is propagated from *OperatorPresentation* component. The internal failure



(a)

(b)

Fig. 11. Fragments of *AlarmService* erroneous behavior: (a) internal (a) external

will be propagated to the component port and then to the connected component causing another error and failure types according to [16]. In Fig. 11 (b) we notice that the external failure is captured as well in the component port behavior and propagated to the internal component behavior. The profiles in CeBAM were designed to capture all required details described in [3,11,16] to model component internal and external erroneous behavior.

4.4 Behavior Composition

We follow AOM [6] approach to compose both behaviors. Composition directives are describing the sequence and the order in which aspects need to be composed with the base model. For instance, in behavioral state machine of component internal behavior *refactor* aspects will be processed and applied first before any *add* aspects. Figure 12 shows fragments from the BSM and PSM for *AlarmService* after composition. The states shaded in gray are erroneous states.

4.5 Guidelines for Using CeBAM

Modeling component behavior using CeBAM can be done in two phases. In the first phase we just model the normal behavior of both component views (internal and external). BSM will be used for the component internal normal behavior (see Fig. 4) and extended PSM for the external view (see Fig. 6). The second phase is focusing on modeling component erroneous behavior separately using two profiles: AspectBSM and ErroneousBehavior profile. The outcome of this phase is represented by two aspect models: one for the erroneous behavior of the internal view and another for the external view. We may need a few iterations to build these two models. First we capture the local failures and then in the next iteration(s) we may have to add propagated failures that originated in other components. The iterations will end when all errors/failures have been "propagated". In some cases we may need to create refactor aspects to preserve the run-to-completion semantics of BSM transitions.

4.6 Components Behavior Conformance

A system is built from different component interacting with each other to accomplish specific scenarios. In UML 2 [17] conformance is considered, but the definition of the conformance semantics is limited [8]. Several approaches in the literature propose different solutions for checking conformance. For instance, in [10] labeled Petri net are used to check the compatibility between component interfaces. In [9] a model checker is used to automate conformance validation between components PSM and its internal behavior in the context of UML-RT.

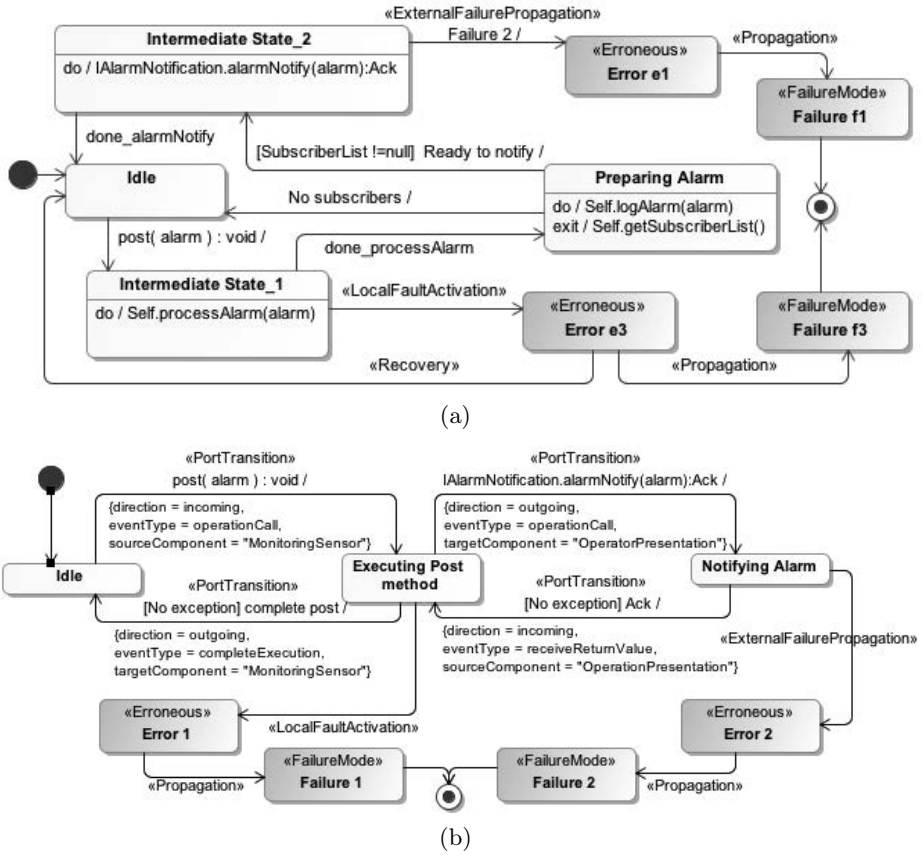


Fig. 12. Fragments of *AlarmService* complete component behavior after composition: (a) internal behavior, (b) external behavior

In our case we are working to address the conformance validation in two levels: between required and provided interfaces and between protocol state machines and internal component behavior. This conformance validation will be done for the composed state machines, which capture both the normal and erroneous behavior. In the first stage, we will automate the conformance validation between internal component behavior and its ports to fix any incompatible behavior. In the next stage we will check the conformance between components interfaces.

5 Related Work and Discussion

Different approaches can be found in literature for predicting the reliability and availability of component based systems. The proposed approaches are classified in [20] into states-based, path-based and additive models. We are following the

state-based approach in our work. Our approach considers that erroneous behavior is an important part of the dependability assessment. It was inspired by different works, such as [4,5,11,19,21]. In [19] a methodology for modeling system robustness behavior using aspect-oriented modeling is proposed. The authors defined an aspect profile for robustness behavior which inspired our definition of the AspectBSM profile. However, we customized our profile for modeling component erroneous behavior.

Some limitation of the UML behavioral and protocol state machines for capturing component port behavior are identified in [8]. The author proposes a Port State Machine (PoSM) to capture the interleaving operation calls on a port, which is defined by modifying the UML meta-model, specializing some of the meta-classes. PoSM focuses only on the operation calls between components and does not capture other type of triggers, i.e. failure propagation and signals. PoSM is not supported by current UML2 tools since the UML meta-model was changed, and this is a major limitation for its applicability. Our approach is addressing the same basic limitations of PSM, but we utilize the UML2 profile mechanism for the required extensions, to make our approach supported by existing UML tools.

A new development framework for dependability analysis was introduced in [11] based on a new intermediate dependability-specific modeling language CHESS ML. A component error model view, represented with a special kind of state machine in CHESS ML, shows faults that can be activated internally or propagated from other components. CHESS ML models are transformed into Petri nets for state-based reliability analysis. Many differences can be identified between our approach and the CHESS approach. First, in our case we do not use an intermediate model and we plan to transform to Petri net directly from the base model. Second, our approach considers the origin of the fault in the normal behavior and uses composed state machines for conformance verification and for generating the Petri nets models. Last but not least, we are following the UML standard for our software models.

The importance of including error propagation in the reliability assessment was identified in [4,5]. The work in [4] considers an error propagation probability and proves the significant impact of error propagation in reliability predictions. The approach in [5] takes into account the error propagation and error propagation path, but does not consider fault tolerance. In [3] a framework for compositional reasoning on the error model is proposed; a new error classification and failure modes are introduced. The execution environment and component usage profile are considered in [22]. The work in [13] studies the effects of software fault tolerance mechanisms in varying architectural configurations in models built in the Palladio Component Model (a non-standard software modeling language for component-based systems). A framework for predicting component reliability by studying component internal behavior is developed in [23], but it focuses on internal components and ignores the error propagation.

The approach in [24] employs application blocks to represent application functionalities and the internal behavior is described using activity diagrams.

An application block may include an activity and be a part of another one. External State Machine (ESM) and its extended version (EESM) are used to model the UML pins behavior, which model the interaction between activities. In our CeBAM approach we follow a component-based approach, using standard UML components with ports and connectors to represent the system architecture. The component internal behavior is modeled using state machines and we extend the UML protocol state machine to describe the component port behavior according to the required and provided interfaces for both incoming and outgoing messages. Moreover, in [24] External Reliability Contract (ERC) are introduced to describe the failures during the communication between activities. The ERCs are modeled separately and they are composed with EESMs using an aspect-oriented modeling approach to reduce the modeling complexity. In CeBAM we also use AOM for the same reasons. However, in CeBAM the erroneous behavior will capture the source of failure inside a component and it shows how it propagates and affects other connected components. In [24] a model checker is used for formal verification, while in our case we are working on transforming the UML model to Stochastic Reward Networks (SRN) [12] for both dependability analysis and conformance checking.

The separation of concerns principle is applied in component-based architectures to reduce the complexity and to improve the quality. This principle is realized by separate protocol behaviors that describe the provided and required component functionality. However, the challenge consists in the composition of these separate protocol behaviors that may be interdependent. Moreover, in the embedded real-time systems where safety is considered, the composition can be even more complicated. This problem is addressed in [25], which provides an automated approach for synthesizing component behavior based on real-time coordination pattern that describe the behavior of connected component interaction. Each component participating in a coordination pattern has a role described by a protocol statechart. In order to synthesize the component external behavior the user needs to define a set of composition rules that explicitly describe the dependencies of components roles. This approach was implemented in the FUJABA Tool Suite [26]. The focus of this approach is on the external behavior of the components. However, in our case we will consider the conformance between the component internal behavior and all its ports, and will also check the compatibility between provided and required interfaces of the connected components.

WEAVER [27] is an Aspect-Oriented modeling tool developed by Motorola, which uses Specification and Description Language (SDL) to model the behavior due to its unambiguous semantics. WEAVER supports model execution, code generation and dependencies between aspects.

The CHARMY framework [28] uses model-checking techniques for validating software architecture conformance. An UML-like notation is used and the tool automates the validation. Static views show the component and connector relationships, while dynamic views describe the internal component behavior. The tool transforms these models into Promela code. To test a specific scenario described by a sequence diagram showing the exchanged messages between

components, the tool transforms the sequence diagram to Buchi Automata. Both the Promela code and Buchi Automata will be passed to SPIN to identify potential deadlocks.

6 Conclusions and Future Work

The paper introduces Component Erroneous Behavior Modeling (CeBAM), an aspect-oriented approach which provides a practical solution for modeling component erroneous behavior and error propagation. We illustrate, with the help of a case study, how to apply aspect-oriented techniques to model the erroneous behavior separately and then to compose automatically erroneous and normal behavior for each component. CeBAM is a part of a larger framework aiming to provide developers with automated tools for assessing the reliability (availability) of different fault tolerance mechanisms applied to a system under development. The composed state machines that are an outcome of CeBAM are used for conformance validation and for generating SRN models for dependability analysis.

We are working now on conformance validation of port behavior with internal behavior. Moreover, we are in the process of developing QVT transformations to generate automatically SRN analysis models from the software model. This transformation will be used for conformance validation and dependability attribute assessment. We are also investigating how to limit the state explosion in the analysis model without affecting too much the reliability and availability prediction results.

Acknowledgments. Naif A. Mokhayesh Alzahrani would like to acknowledge the support provided by Albaha University, KSA as well as the Ministry of Higher Education, KSA. This research was partially supported by the Natural Sciences and Engineering Research Council (NSERC).

References

1. Bernardi, S., Merseguer, J., Petriu, D.C.: Dependability modeling and analysis of software systems specified with UML. ACM Computing Surveys (CSUR) 45(1), Art. 2 (2012)
2. Immonen, A., Niemelä, E.: Survey of reliability and availability prediction methods from the viewpoint of software architecture. Software & System Modeling 7(1), 49–65 (2008)
3. Aysan, H., Punnekkat, S., Dobrin, R.: Error Modeling in Dependable Component-Based Systems. In: Proceedings of the 2008 32nd Annual IEEE International Computer Software and Applications Conference (COMSAC 2008), pp.1309–1314. IEEE Computer Society (2008)
4. Popic, P., Desovski, D., Abdelmoez, W., Cukic, B.: Error propagation in the reliability analysis of component based systems. In: Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering (ISSRE 2005), pp. 53–62. IEEE Computer Society (2005)

5. Cortellessa, V., Grassi, V.: A Modeling Approach to Analyze the Impact of Error Propagation on Reliability of Component-Based Systems. In: Schmidt, H.W., Crnković, I., Heineman, G.T., Stafford, J.A. (eds.) CBSE 2007. LNCS, vol. 4608, pp. 140–156. Springer, Heidelberg (2007)
6. Yedduladoddi, R.: Aspect oriented software development: an approach to composing UML design models. VDM Publishing (2009)
7. Bernardi, S., Merseguer, J., Petriu, D.C.: A dependability profile within MARTE. Software & Systems Modeling 10(3), 313–336 (2011)
8. Mencl, V.: Specifying component behavior with port state machines. Electronic Notes in Theoretical Computer Science 101, 129–153 (2004)
9. Moffett, Y., Beaulieu, A., Dingel, J.: Verifying UML-RT protocol conformance using model checking. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 410–424. Springer, Heidelberg (2011)
10. Craig, D.C., et al.: Compatibility of Software Components - Modeling and Verification. In: Proceedings of the International Conference on Dependability of Computer Systems (DEPCOS-RELCOMEX 2006), pp. 11–18. IEEE Computer Society (2006)
11. Montecchi, L., Lollini, P., Bondavalli, A.: Dependability Concerns in Model-Driven Engineering. In: Proceedings of the 2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW 2011), pp. 254–263. IEEE Computer Society (2011)
12. Muppala, J., Ciardo, G., Trivedi, K.S.: Stochastic reward nets for reliability prediction. Communications in Reliability, Maintainability and Serviceability, 9–20 (1994)
13. Brosch, F., Buhnova, B., Koziolek, H., Reussner, R.: Reliability prediction for fault-tolerant software architectures. In: Proceedings of the Joint ACM SIGSOFT Conference – QoSA and ACM SIGSOFT Symposium – ISARCS on Quality of Software Architectures – QoSA and Architecting Critical Systems (QoSA–ISARCS 2011), pp. 75–84. ACM (2011)
14. Object Management Group: Query View Transformation (QVT) v1.1 formal/2011-01-01, `http://www.omg.org/spec/QVT/`
15. Gomaa, H.: Software Modeling and Design. Cambridge University Press (2011)
16. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing 1(1), 11–33 (2004)
17. Object Management Group: Unified Modeling Language (UML) - Superstructure v.2.4.1 formal/2011-08-06, `http://www.omg.org/spec/UML/2.4.1/`
18. Selic, B.: A systematic approach to domain-specific language design using UML. In: Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC 2007), pp. 2–9. IEEE Computer Society (2007)
19. Ali, S., Briand, L.C., Hemmati, H.: Modeling robustness behavior using aspect-oriented modeling to support robustness testing of industrial systems. Software & Systems Modeling 11(4), 633–670 (2012)
20. Goseva-Popstojanova, K., Trivedi, K.S.: Architecture-based approach to reliability assessment of software systems. Performance Evaluation 45(2-3), 179–204 (2001)
21. Abdelmoez, W., et al.: Error propagation in software architectures. In: Proceedings of the Software Metrics 10th International Symposium (METRICS 2004), pp. 384–393. IEEE Computer Society (2004)

22. Reussner, R.H., Schmidt, H.W., Poernomo, I.H.: Reliability prediction for component-based software architectures. Journal of Systems and Software 66(3), 241–252 (2003)
23. Cheung, L., Roshandel, R., Medvidovic, N., Golubchik, L.: Early prediction of software component reliability. In: Proceedings of the 30th International Conference on Software Engineering (ICSE 2008), pp. 111–120. ACM (2008)
24. Såtten, V., Kraemer, F.A., Herrmann, P.: Towards automatic generation of formal specifications to validate and verify reliable distributed systems – A method exemplified by an industrial case study. ACM SIGPLAN Notices - GCPE 2011 47(3), 147–156 (2012)
25. Eckardt, T., Henkler, S.: Synthesis of Component Behavior. In: Proceedings of the 7th International Fujaba Days, Eindhoven University of Technology, The Netherlands, pp. 40–44 (2009)
26. FUJABA Tool Suite, `http://www.fujaba.de`
27. Cottenier, T., Van Den Berg, A., Elrad, T.: Motorola WEAVR: Aspect orientation and model-driven engineering. Journal of Object Technology, 51–88 (2007)
28. Inverardi, P., et al.: CHARMY – An extensible tool for architectural analysis. In: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE-13), pp. 111–114. ACM (2005)

An IMS DSL Developed at Ericsson

Pascal Potvin, Mario Bonja, Gordon Bailey, and Pierre Busnel

Ericsson, 8400 boulevard Décarie, Mont-Royal, Québec, H4P 2N2, Canada
{pascal.potvin,mario.bonja,gordon.bailey,
pierre.busnel}@ericsson.com

Abstract. In this paper, we present how we created a Domain Specific Language (DSL) dedicated to IP Multimedia Subsystem (IMS) at Ericsson. First, we introduce IMS and how developers are burdened by its complexity when integrating it in their application. Then we describe the principles we followed to create our new IMS DSL from its core in the Scala language to its syntax. We then present: how we integrated the IMS DSL into existing projects, how it can save time for developers, and the readability of the IMS DSL syntax.

Keywords: Domain Specific Language, IP Multimedia System, application development, industrial experience.

1 Introduction

The IP Multimedia Subsystem (IMS) relies on a complex architecture. The whole system is made up of different components, each with a very specific purpose, such as the Call Session Control Function (CSCF), which aggregates several roles related to sessions (routing, registering, etc.), the Home Subscriber Server (HSS) for managing user identities, authentication, subscription information, etc., the Presence and Group Management (PGM) for handling presence information about users and groups, and the Media Resource Function (MRF) for mixing, selecting and converting media sources and playing announcements and tones. These are essential components, but the IMS architecture contains many more. Knowing them is important for a developer building an IMS based application to understand the behavior of an IMS network and how to interact with it.

The CSCF contains a proxy which serves as an entry point to IMS functionalities. The client communicates with the proxy using the Session Initiation Protocol (SIP). IMS also works with many other communication protocols, such as the Session Description Protocol (SDP) for negotiating media properties during a SIP Invite request, the Message Session Relay Protocol (MSRP) for transferring files as well as instant messaging, HTTP for updating presence documents through the XML Configuration Access Protocol (XCAP), H.248 for media mixing, and playing tones and announcements, etc. Thus, IMS application developers have to learn the different processes to register with IMS, to publish a presence document, to send an instant message to another user or to handle media mixing between users, sending tones and announcements.

F. Khendek et al. (Eds.): SDL 2013, LNCS 7916, pp. 144–162, 2013.

All the necessary information about IMS in general, the specifics about the CSCF, HSS, PGM and MRF nodes and details about the SIP, SDP, MSRP and XCAP protocols is disseminated in multiple Internet Engineering Task Force (IETF) Request for Comments (RFC), the standards defining internet technologies [1]. The information about the HTTP and XML standards can be found on the World Wide Web Consortium standards pages [2]. Finally the information about the H.248 standard is published by the International Telecommunication Union (ITU) as a Recommendation [3]. Even for basic operations, such as registering and subscribing, developers have to refer to several documents. This process is both time-consuming and frustrating for developers who only want to use simple IMS functionalities in an application without the hassle of learning IMS in fine detail. Facing a steep learning curve, developers must immerse themselves in IMS and become experts in order to use it. In an industrial context this situation usually leads to the need to establish large teams of specialists covering the different areas of knowledge required to develop a given functionality, hence it is a limiting factor to the spontaneous development of new services by enthusiasts.

We begin this essay with an introduction to DSLs, and then explain our choice of Scala for the implementation of our IMS DSL. Following this we discuss our experience implementing four prototypes using the IMS DSL. Finally we present the architecture and features of the IMS DSL and conclude our discussion.

2 DSL

A Domain Specific Language (DSL) [4,5,6,7,8] is simple and concise with the expressive power focused on a particular problem domain. It is custom-built to be very intuitive and fluent for a domain expert to use. It allows one to efficiently and quickly build applications for that domain, thus reducing development time and increasing productivity.

By definition, a good DSL is at a higher level of abstraction than a high-level General-Purpose programming Language (GPL). The goal of a DSL is to digest the complexity of the problem domain, de-cluttering it of the implementation details through a syntax built around familiar terms and concepts from the domain. This allows domain experts to save time and effort and focus on the tasks of interest, such as enabling IMS communications without having to worry about the programming details of traditional libraries or APIs such as initialization and default handling which must be done explicitly in traditional libraries. This also enables domain neophytes with minimal knowledge of the domain to create sound IMS applications without having to learn all the intricacies of IMS first. Ideally, domain experts could provide the required knowledge to further develop the IMS DSL. This simplification and clarification of the problem domain allows IMS application development to be left to the neophyte, making prototyping and developing proof-of-concept applications much more practical, especially in cases where the complexity of the problem domain is high, such as the case of IMS.

We have decided to build the IMS DSL on top of a GPL, thus following the embedded approach [9,10] in order to save us the effort required to develop all the peripheral functionality of a complete language e.g. conditional handling, loops, etc. and also to enable us to easily use existing libraries covering the protocols we want to offer functionality for. The host language of the resulting IMS DSL is Scala. While it is a relatively young language, it will offer much more flexibility than other GPLs.

3 Scala

Scala [11] is a GPL designed to build software components in a concise and type-safe way. It integrates features of object-oriented and functional programming paradigms. The source code sizes of applications written in Scala are typically smaller by a factor of two or three compared to equivalent Java applications.

Existing Java code and programmer skills are fully re-usable. Scala is compiled into bytecode to be run on the Java Virtual Machine (JVM), which allows Scala programs to access functionality defined in Java. Scala code can be called from Java code and vice versa.

Below we list a few of the interesting properties of Scala with respect to the development of a DSL:

- Scala provides a lightweight syntax for defining anonymous functions, supports higher-order functions, allows functions to be nested, and supports currying. This helped us develop a more readable syntax for the IMS DSL.
- Scala code can run on the Android platform, .Net platform, and anywhere else Java code can run. This provides us with the potential for deploying our IMS DSL not only on dedicated network nodes, but eventually also on end user equipment to facilitate development on a wider range of platforms.
- Scala provides type inference, everything-is-an-object, function passing, and many other features which cut away unneeded syntactic overhead. Scala is a pure object-oriented language in the sense that every value is an object; it is a functional language in the sense that every function is a value; and it is a statically typed programming language. Types and behaviors of objects are described by classes and traits. For the IMS DSL these traits mean a simpler syntax and structure without the need for complex class hierarchies.
- Scala code can be run in an interactive shell where Scala expressions are interpreted interactively. A Scala program may also be run as a shell script or as a batch command. Initially this helped us develop and test the IMS DSL. At the current state we have less need for the interpreter but might look back at its use in the future.

We take advantage of the fact that in Scala's syntax dots are optional in most method calls, and parentheses are not required for method calls with zero or one parameter, to develop a DSL that is more readable than most GPLs. For instance, the chain of method calls userA.send("Hello World").to(userB) can also be written as userA send "Hello World" to userB. Thus, the syntax of our IMS DSL

can be given a format which is similar to English and is simpler to understand, since it is close to natural language as a result of the absence of dots and parentheses, which are mandatory in most modern GPLs. Despite its resemblance to natural language, our DSL syntax is well-defined and unambiguous in terms of method invocation order. It is worth noting, as we will see later on, that when the IMS DSL is used within a java program this intuitiveness is limited e.g.: we need to use the dot notation.

4 IMS DSL Development in Scala

The IMS DSL must be simple to understand and use correct IMS terminology so that software engineers who have already acquired high-level knowledge of IMS can learn its syntax quickly and put it in practice immediately. Thus, our IMS DSL must avoid constructs peculiar to GPLs like variable declaration. It must get rid of any irrelevant programming details which are a legacy of GPLs, while remaining extensible and flexible enough for the domain of IMS communication.

To develop the IMS DSL, we have made use of the relaxed syntax of Scala as well as its implicit conversion between types. Other useful features of Scala, such as functional decomposition and identifier names entirely made up of arithmetic operator characters, have not yet been exploited. However, as more and more functionalities are added to the IMS DSL in the future, these features will be indispensable for the conciseness and maintainability of the IMS DSL.

The IMS DSL features have been built following a few principles:

- List the domain concepts to be expressed in the DSL and the relationship they have with respect to each other's.
- From that list define the domain notation and syntax.
- Make sure that each syntax element has an intuitive, logical and functional default behavior, while still enabling more specific behaviors when required.
- Keep the syntax free of anything which does not come from the problem domain itself, only include host language specific artifacts if absolutely necessary.

Those propositions may sound simple but discipline is required to fulfill them properly. In order to properly perform these steps, a good understanding of the domain is required as well as programming competency. However one must not let his previous experience in programming taint or limit the syntax to be developed. It is really easy for one to simply dilute the DSL aspect to a point where it would be indistinguishable from a traditional library.

In the context of the IMS DSL the domain is defined as an application using the available IMS interfaces. Those interfaces are pre-provisioned and the details of the provisioning are controlled by the IMS DSL developed application through configuration files which are application specific. Thus an IMS network must already be provided and configured and some of the details of that configuration need to be available to the IMS DSL application.

5 Development Experiences Using the IMS DSL

The development of the IMS DSL as described here took place over the course of the last two years. It followed an iterative approach where new functionality was introduced and refined in the IMS DSL syntax as new requirements in the prototype projects developed using the IMS DSL arose.

The first three projects listed below were developed as proofs of concept of how IMS technology can be an enabler for machine-to-machine (M2M) communication in the vision of "More than 50 billion connected devices" [12] that Ericsson is bringing forward in the industry. The core functionalities of an IMS network include authentication, security, and quality of service management, which greatly simplifies the development of M2M applications.

The last project we discuss below, still a proof of concept, was basically chosen as a vehicle to augment the IMS DSL's provided functionality.

5.1 Tolmie 2 - Assisted Living Project

The objective of the Tolmie project is to build the basics of the IMS DSL and demonstrate the advantages of that approach compared to a more traditional general purpose language approach. The Tolmie project had previously been implemented in C++. By re-implementing the Tolmie project using the IMS DSL we can firstly develop a base for the IMS DSL and secondly compare the new Tolmie 2 implementation to the pre-existing Tolmie C++ prototype for the same functionality. We can compare how the IMS DSL improves the efficiency of development by considering the time it takes for both approaches and we can qualitatively compare how much more expressive the IMS DSL is compared to the direct library calls used in the initial prototype.

For those reasons the Tolmie project, also known as the Assisted Living project, was chosen as a basis for comparison. In the Tolmie project we demonstrate that IMS can be used as a smart bit-pipe for machine-to-machine communication, allowing health professionals to monitor their patients remotely. The patient wears a life-monitoring sensor that periodically sends health data to a server. The server contains an agent manager that can interact with IMS. The caregivers are able to observe the evolution of patient data on an Android application called eRCS client, which receives data over IMS.

The Tolmie Agent Manager establishes IMS access for six functional requirements; we use the IMS DSL to implement those six IMS functionalities.

When the agent manager receives sensor data from a new device, it creates a new agent to act on behalf of the device. The newly created agent then needs to register with the IMS network. After completing the registration, the agent manager will publish data about itself to the PGM so that the caregivers will be notified.

In order for a caregiver to request or stop a live feed through the eRCS client, each agent must be ready to receive a SIP instant message from the eRCS client and process its content to determine whether to add or remove a live feed subscriber. The agent must then be able to send live feed data to

the eRCS client through SIP instant messaging. Once the maximum live feed duration has passed, the agent must send a SIP instant message to uncheck the live feed button on the eRCS client of each live feed subscriber.

In brief, the required IMS functionalities in Tolmie are as follows: SIP Register, SIP Publish, and sending and receiving SIP Instant Messaging.

After completion of the IMS DSL implementation of the prototype, a comparison can be made between the original code in C++, which uses a C SIP stack (PJSIP), and the new one in Java, which uses the IMS DSL for accessing IMS functionalities, to observe how the IMS DSL can lead to more concise code and much shorter development times.

The original development time (coding and unit testing only) for Tolmie was 12 man-weeks. The re-implementation of Tolmie 2 using the IMS DSL took 3 man-weeks and implementing the IMS DSL itself took 9 man-weeks. It is to be noted that this figure intentionally excludes the high level design and testing of the solution in both cases since it is obvious that it was in part re-used for implementing the IMS DSL version of it, and thus would have made it look better than it should in reality. Secondly one should also know that some of the developers were common between the developments of the two versions thus leading to some non-measureable benefits while developing the IMS DSL version.

Table 1 below presents a comparison between the amounts of code required to implement various IMS related functions in C++ and in the IMS DSL:

Table 1. Required lines of code in C++ versus using the IMS DSL

Functionality	*C++ Lines Required*	*IMS DSL Lines Required*
Initialization	27	0
SIP Register	23	1
Send Message	11	1
Receive Message	49	1
SIP Publish	131	1

5.2 Tolmie 3 - Automotive Telemetry Project

The objective of the Tolmie 3 project, also known as the Automotive Telemetry project, is to further our knowledge and expertise in using IMS as a smart bit-pipe for machine-to-machine communication, by allowing an engineer or a mechanic to obtain real-time data from an automobile on-board computer. A device is linked to the automobile CAN Bus, periodically forwarding data to a server via IMS steams on a mobile network (3G or LTE). The engineers or mechanics can monitor the automobile data via a web interface linked to the server.

The goal of implementing Tolmie 3 using the existing IMS DSL is to observe how the IMS DSL facilitates re-use of existing functionality and judge the ease of implementing new features to support the new requirements.

As an improvement over the Tolmie 2 project, where the data was sent over IP to an IMS gateway where an agent manager was translating the data into an IMS format, the Tolmie 3 project performs the IMS encapsulation as the first step. Thus the device linked to the CAN Bus is the IMS User Agent and communicates directly with the IMS network, providing authentication and security directly from the device itself.

Comparing the development time of Tolmie 3 with that of Tolmie 2, which had similar functionality in a different configuration, we can verify that there is a re-use factor provided by the IMS DSL between the two projects. The development time (coding and unit testing) for Tolmie 3 took four man-weeks and practically no re-work was required on the IMS DSL in order to implement the prototype. This development time is quite close to the three weeks required to develop Tolmie 2 (excluding the development of the DSL itself), proving the potential for re-use of the IMS DSL.

5.3 Area 51 - Home Automation Project

The objective of the Area 51 project, also known as the Home Automation project, is to verify whether the IMS DSL can be used by IMS neophytes to develop an automated house where the IMS DSL is used to develop an IMS smart bit-pipe for machine-to-machine communication with a server which can be accessed by the home owner. The project was driven by enthusiasts volunteering occasionally in their spare time, and two eight hours sessions where the group gathered to work in a more structured fashion.

A model house is used to represent the home. A number of sensors and actuators are installed in the house: a doorbell and a motor to open and close the entrance door remotely; lighting and switches to control the main rooms' light levels; a light detector to control the exterior lighting based on ambient light levels and a temperature sensor and electric fan to represent the climate control of the house. These components are connected to an Arduino [13] microcontroller, which communicates via a USB connection with the Home Gateway we built using an Odroid-X [14] an ARM based micro-computer platform. The Home Gateway makes use of the IMS DSL to encapsulate the data from the house into an IMS pipe going to a server, and to receive commands from the server via the same IMS pipe. On the server side, the IMS DSL is used for the same purpose in order to communicate with a web server accessible by the home owner.

We can observe in this development environment if the IMS DSL is simple enough that enthusiasts can use it in order to implement the required functionality within the time constraints imposed by the schedule.

As this project was executed on a voluntary basis, we do not have a strict accounting of the spent time to develop the prototype. However, based on the

feedback of the group of volunteers, we can make the qualitative statement that the IMS DSL is convenient and easy to use for IMS neophytes, enabling them to quickly create a functioning prototype application.

5.4 A Core IMS DSL (ACID) Telephony Application Server (TAS) Project

The objective of the ACID-TAS project, also known as the Light Weight Telephony Server, is to further our knowledge and expertise in implementing an IMS DSL. For this project we are developing the following telephony services: originating and terminating call handling; incoming and outgoing call barring; originating and terminating identity presentation and restriction; call diversion (on busy, on no reply, on not logged-in, deflection and unconditional) and a conference call service. For this project, the IMS DSL can no longer act merely as a user agent. It needs to provide back-to-back user agent functionality to the developers. Moreover, it needs to address new interfaces, such as the H.248 protocol for conference handling with the MRF.

A light weight telephony server had been implemented as a proof of concept within Ericsson using Java and the JSR-289 SIP Servlet framework a few years ago. The goal of the ACID TAS project was to test our IMS DSL on an existing and more demanding IMS based prototype and compare it to the previous project. A comparison can then be made between the original project's code and that of the new prototype which makes use of the IMS DSL. Again, we can observe how the IMS DSL leads to more concise code and much shorter development times.

The original development time (coding and unit testing) for the Light Weight Telephony Server in Java using the JSR-289 framework had been 22 man-months. The re-implementation through ACID-TAS using the IMS DSL took 4 man-months and implementing the new required functionality in the IMS DSL itself took 9 man-months. Again it is to be noted that this figure intentionally excludes the high level design and testing of the solution in both cases since it is obvious that it was in part re-used for implementing the IMS DSL version of it, and thus would have made it look better than it should in reality. In this trial however, none of the original developers (non-DSL version) participated in coding of the IMS DSL version of the Telephony Application Server. We also consider that the general level of proficiency of the developers in both projects were of equivalent levels. Finally, the objectives and the acceptance criteria were the same for both proofs of concept which ended with a demonstration to the financing stake holders. This shows again the advantage of using a DSL on the development time and the re-use factor in doing so.

From the point of view of readability, we demonstrate later on in this essay how simple the conference server portion of the ACID-TAS is to understand.

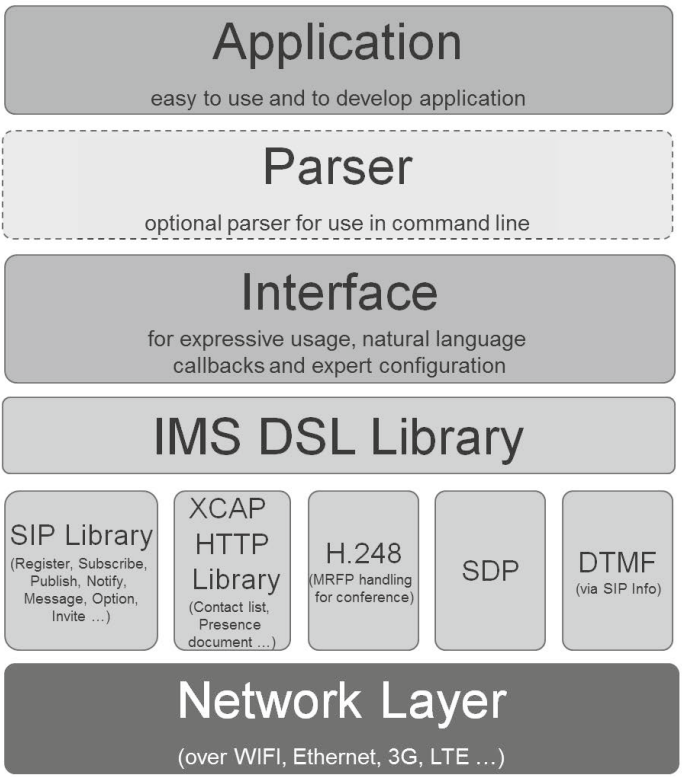


Fig. 1. IMS DSL architecture layers

6 Architecture

Figure 1 below presents the multiple layers of the architecture of the resulting DSL design positioned in the IMS architecture. Starting from the bottom to the top, we rely on the network layer to access the IMS network from our IMS library. The IMS DSL Library layer is a standalone java-written interface that contains IMS primitives such as register, subscribe, publish, etc. It uses both SIP and XCAP protocols for interacting with IMS. HTTP based protocols like XCAP are implemented using the standard Java HTTP library.

The Session Initiation Protocol is implemented using Jain-SIP, an open-source Java SIP library. Jain-SIP only provides a low-level API to instantiate, send and receive basic SIP messages. To get the high level SIP user-agent functionalities like registering, publishing, subscribing to users etc., we developed a complementary layer acting as the missing user-agent. One instance of this layer can then be manipulated as a SIP user would. The IMS library layer uses these sub-libraries transparently.

The IMS library interfaces with the DSL in such a way that there is little coupling between the two layers. Thus, it will be easy to switch the set of libraries it depends upon in the future to better match general Ericsson architectures.

The IMS DSL layer is the DSL itself, and is the core of our development. It is the result of the grammar development and the selection of the abstraction level. The DSL is developed in Scala and relies on the IMS Library for accessing IMS functionalities. It is composed of objects, classes and methods to support the DSL syntax and internal operations related to IMS.

The Interpreter is an optional layer that can be used to call the DSL from another language.

Finally the application can be any program coded in Java using the embedded IMS DSL to build a service from the provided functionality.

6.1 IMS DSL Features

The current version of the IMS DSL supports a limited number of features, with each feature's syntax designed to express concepts from the IMS domain as simply and naturally as possible. The following is a list of the current IMS functionalities supported and how they are expressed in the IMS DSL. Thanks to the interoperability between Scala and Java, it is possible to use the IMS DSL as an API directly in Java code.

This leads to the obvious question of why we call the IMS DSL a DSL and not simply an API? Through the development of the IMS DSL, the primary focus has been to provide a simple and concise syntax to express concepts in the IMS domain. Once that syntax had been defined, we implement it using Scala. As a final step we devise a scheme for java applications to access the IMS DSL syntax. This is what makes an embedded DSL different from a traditional library or API. Libraries and APIs are developed to provide functionality through a host language, using all the facilities and conventions of the host language. In an embedded DSL, the first priority is to define a syntax that clearly and concisely expresses ideas in the problem domain. Once this syntax is defined, it is implemented as completely as possible within the constraints imposed by the host language. Intuitive and simple expression of domain concepts always takes precedence over the established conventions and idioms of the host language.

6.2 Registering with IMS

This entails sending a SIP REGISTER Message to the Call Session Control Function in order to register a SIP user with the Home Subscriber Server. If successful, the server will reply with a positive response, and the registration will be valid for a certain duration after which the user must register again. Such technical details have been abstracted by the IMS DSL, so one can register by writing the following code:

```
<USER> hasCredentials (<USERNAME>, <DOMAIN>, <PASSWORD>)
```

6.3 Sending a SIP Request or a Status

After a SIP user has been registered, it can send SIP requests to other users via their SIP URI. A request body and various headers may optionally be added.

```
<USER> sendRequest <REQUEST TYPE>
  [ withHeader(<NAME>,<VALUE>) ]
  to <SIP_URI>
```

The same can also be done to respond with a SIP status. The send response method is quite powerful in the sense that it uses the supplied request information to build the response and send it.

```
<USER> sendStatus <STATUS_CODE>
  inResponseTo <INCOMING_REQUEST>
```

6.4 Sending a SIP Message (Instant Message)

A registered user can send a message to another user via their SIP URI. A content type and multiple headers may optionally be added. Sending an instant message is actually a special case of sending a SIP request.

```
<USER> send <MESSAGE>
  [ withContentType <TYPE> ]
  [ withHeader(<NAME>,<VALUE>) ]
  to <SIP_URI>
```

6.5 Publish XCAP

The IMS DSL allows the user to publish its current presence state by using the XCAP protocol. The XML data within a specific XML tag is published to the presence document in the PGM.

```
<USER> publish <XML_DATA> as <TAG_NAME>
```

6.6 Managing Contact Lists

The user's contact list can be managed with the following methods:

```
<USER> [ addContact | removeContact ] <USER_URI>

<USER> [ newContactList | removeContactList ] <LIST_NAME>

<USER> add <USER_URI> to <LIST_NAME>

<USER> remove <USER_URI> from <LIST_NAME>
```

6.7 Incoming Message Handling to Perform Actions

An action (user-supplied method) can be bound to the reception of a message with optional conditions. The action will be executed only if all the conditions are met. If not, the action is passed to the next configured handler. If none of the handlers are configured to handle this message, it is simply replied to with a positive acknowledgement. It is worth noting that the handling of instant message reception is managed by the IMS DSL, so the developer will not have to write event handlers to specify what to do.

The `MESSAGE TYPE` can either be a request type (Any, Bye, Invite...) or a response type (Any, Ringing, Ok...).

```
<USER> onReceive <MESSAGE_TYPE>
  [ withContentType <TYPE> ]
  [ withBody <BODY> ]
  [ withHeader(<NAME>,<VALUE>) ]
  [ from <SIP_URI> ]
  Do <ACTION>
```

6.8 Incoming Dual Tone Multiple Frequencies (DTMF) Handling

Similarly to the incoming message handling, the IMS DSL allows the user to execute specific actions when receiving DTMF digits encapsulated in SIP INFO messages. Handling DTMF digits is actually a special case of the onReceive method shown above.

```
<USER> onDtmf Do <ACTION>
```

6.9 Conferencing

The IMS DSL offers conferencing capabilities.

First, the conferencing engine needs to be initialized. This can be done either when the server user is created or later. This must be done once for each registered server user.

```
<SERVER> supportingConference
```

When the conferencing engine is ready, an ad-hoc conference can be established on that server user. It starts with the participants of a 2-party call inviting a third party to the call. It is assumed the 2 first participants are already in an active call. The IMS DSL needs the full URI and call ID of the 2 first participants and the phone number of the third party.

The complexity of the conferencing feature is hidden from the end user. The IMS DSL will send the appropriate SIP messages to the initial participants to move them from a point-to-point call to the conference bridge. The MRFP H.248 signalling is also handled in the process.

```
<SERVER> createConf <CONFERENCE_URI>
  withInitialParticipant
    <PART_A_URI> <PART_A_CALLID>
    <PART_B_URI> <PART_B_CALLID>
```

The following code will add the new participant to the conference. This step can be repeated for each new participant. Again, the IMS DSL handles all the SIP and H.248 signalling.

```
<SERVER> updateConf <CONFERENCE_URI>
  withNewParticipant <PART_NUMBER>
```

When the participants are leaving the conference, their SIP client will send a SIP BYE message. The following code will remove them from the conference.

```
  <SERVER> removeParticipant <PARTICIPANT_URI>
```

6.10 A Typical IMS DSL Usage

The previous sections describing the various capabilities were expressed in the IMS DSL syntax. When used in a Java environment, the IMS DSL syntax is invoked using the Java API. For instance, the methods that create an ad-hoc conference:

```
Server createConf "15141234000@ims.server.ericsson.com"
  withInitialParticipants
    "15141234567@ims.server.ericsson.com" "12345634567"
    "15141234568@ims.server.ericsson.com" "12345634568"
Server updateConf "15141234000@ims.server.ericsson.com"
  withNewParticipant "15141234568"
```

Would look like this:

```
Server.createConf("15141234000@ims.server.ericsson.com")
  .withInitialParticipants(
    "15141234567@ims.server.ericsson.com", "12345634567",
    "15141234568@ims.server.ericsson.com",
    "12345634568");
Server.updateConf("15141234000@ims.server.ericsson.com")
  .withNewParticipant("15141234568");
```

The application code example shows how the IMS DSL conferencing feature is actually used in a java context. Basically Scala provides a java front end so that the IMS DSL primitives can be directly used in a java application.

The processData() method was called by the DTMF action method when a user entered DTMF digits during a call. The request parameter contains the incoming SIP INFO message with the entered digit in the message body. The method uses data accessors such as request.getRType() and request.getHeader() to easily access SIP header information. The incoming data manipulation is

minimal and the incoming SIP INFO message is used unmodified when passed to the conferencing methods. The only computation done in the processData method is to gather the digits entered by the end user and fetch the other participant's call ID from the local session info database.

This example uses 7 lines of IMS DSL code. If we were to implement the same functionality in Java, the code size would grow by at least 300 lines. Bold text indicates IMS DSL code as accessed via the java API to the IMS DSL.

```
public void processData (Request request)
{
  // Respond right away with OK.
  server.sendStatus(StatusCode.OK).inResponseTo(request);
  // We only handle SIP Info messages.
  if ( request.getRType().equals(RequestType.Info) )
  {
    // Add the received digit to buffer.
    // Expected format that is: "#10-digit-number#".
    Buffer += request.getBody().charAt(request.getBody()
           .indexOf("=") + 1);

    // pattern is ".*#
d{10,}#.*"
    if (buffer.matches(pattern))
    {
      // We have a pattern match. For this sample,
      // we only have one conferencing server.
      String conferenceURI =
        "15148500002@ims.server.ericsson.com";

      if ( !request.getReceiverUri()
      .contentEquals(conferenceURI) )
      {
        // The sender is not part of a conference,
        // let's add everyone.
        // First, we need to extract the call id of
        // both legs. It is in the form of
        // "Call-ID: <callerId string>".
        String fromCallID = request.getHeader("Call-ID");
        // Since the SIP Info message does not contain
        // the call ID of the other leg, we get it from
        // the database we keep of the various users.
        String toCallID = SessionInfo.getSessionInfo()
                          .getForwardCallID(fromCallID);
        server.createConf(conferenceURI)
          .withInitialParticipants(
            request.getHeader("From"),
```

```
            fromCallID,
            request.getHeader("To"),
            toCallID);
      }

      // Format the joining number URI.
      int start = buffer.indexOf("#");
      int end = buffer.indexOf("#", start + 5);
      String joiningNumber =
        buffer.substring(start + 1, end);
      String joiningNumberURI =
        SipUri.getShortUri(joiningNumber,
        "ims.server.ericsson.com");

      server.retrieveConf(conferenceURI).
        addNewParticipant(joiningNumberURI);
      // We are done with the buffer, clean it.
      buffer = "";
    }
  }
}
```

6.11 Graphical Representation

In order to help visualize the flow of events generated by the IMS DSL on a network level, a graphical representation was built to accompany the IMS DSL. The goal of the graphical representation is to complement the IMS DSL and enhance its capacity to simplify development, both for IMS domain experts and for neophyte developers. The graphical representation runs in parallel with multiple IMS DSL applications, collecting information about the IMS activity they generate, and displaying that information to the IMS DSL user in a simple and intuitive way.

Currently the graphical representation shown in fig. 2 below is limited to a dynamic view of the execution of the IMS DSL code. Our ambition in the future is to provide a way to model the static structure of the IMS DSL code and its dynamic execution in a similar way. Then, the current representation would be used to compare the designed model to the executed behavior.

The graphical representation is integrated in the IMS DSL, and collects all of the IMS packets that it sends and receives. Multiple IMS DSL applications may be analyzed simultaneously; each program is synchronized against a common reference clock, allowing packets from multiple applications to be displayed in the correct order. The collected packets are analyzed, and logically connected packets are grouped together. As packets are collected, they are displayed to the user in a sequence diagram. In the diagram, logical groups of packets are visually connected, and are color-coded to indicate their status.

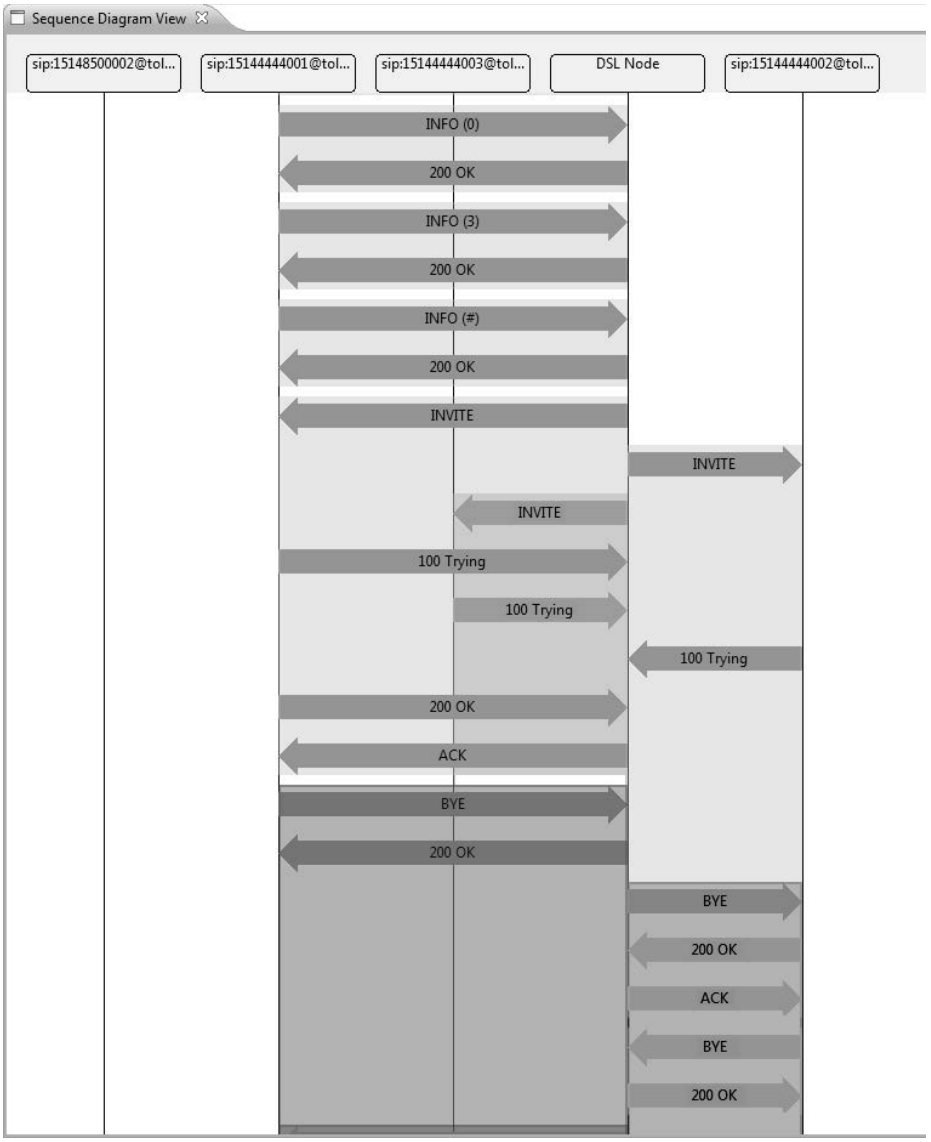


Fig. 2. Graphical Representation

The graphical representation tool is implemented in three parts. The first part of the graphical representation tool is code which is integrated into the source of the IMS DSL. This integrated component captures all outgoing and incoming packets, as well as full Java stack traces for each packet.

The second part of the graphical representation is a server program which accepts, synchronizes and analyzes information from running IMS DSL programmed applications, and makes it available to the graphical client program.

The third part is the graphical client program, which receives information from the server component and visualizes it in the sequence diagram display.

The features of the graphical representation tool provide advantages for both experienced software developers and IMS domain experts. For software developers who are not well versed in IMS communication, the intuitive graphical representation provides insight into how their program operates, and where problems might lie. For domain experts, the graphical representation gives a convenient high-level view that offers a lot of information at a glance, but also allows the expert the freedom to examine the details of their program's operation.

The graphical representation sequence diagram display and color-coded groupings make it possible for domain experts and developers to quickly understand how their applications interact with IMS on a high level of abstraction. The sequence diagram layout is familiar to domain experts and software developers alike, and is intended to be simple for both classes of user to understand. Color-coded packet groups add additional structure to the familiar layout, and reduce the amount of time required to understand the IMS communication represented by the diagram. For domain experts, the colored groups highlight patterns which are already well known, and for software developers without a strong knowledge of IMS, they are a useful learning tool, hinting at the meaning of the underlying data.

Like the IMS DSL itself, the graphical representation provides a high level view, but does not restrict its users. It allows them to view IMS activity in detail. Each packet can be inspected to reveal its complete contents, enabling domain experts to understand the behavior of their applications on a much lower level of abstraction, and debug complex IMS communication problems.

Developers and domain experts both benefit from the ability to link sent IMS packets back to their IMS DSL source code. For domain experts this facilitates understanding the DSL, as it allows the familiar area of IMS communications to be mapped very concretely to DSL commands. For the experienced software developer, this feature is useful for understanding the DSL, as well as for aiding in the understanding of IMS. Linking high-level DSL commands to the exchanges of IMS messages that they produce provides insight into how logical actions, such as the initiation of a telephone call, are accomplished through exchanges of multiple packets in IMS.

7 Conclusion

Two of the projects implemented using the IMS DSL, Tolmie 2 and ACID TAS, were initially created using different languages, paradigms and team members than the current ones. Comparing the recorded working hours for the coding and unit testing of the original projects with the recorded hours for the current IMS DSL incarnations of those projects, we can claim at least four fold increases

in efficiency for those phases of software design and development. This figure is not taking into account the time required to produce the actual IMS DSL. If we factor in this additional time, we arrive at approximately equal costs for the original and DSL implementations for one of the developed projects. Hence developing an application and the domain language supporting this application does not account for a higher cost. However, having at hand a DSL speeds up any subsequent project making use of it and also facilitates the domain comprehension for non-experts as shown from the feedback received from a group of enthusiast coders on the Area 51 project.

Through the development of the IMS DSL we have gained knowledge and experience on the process of developing a Scala embedded DSL. Through the projects developed using the IMS DSL we have been able to measure and observe the benefits in terms of code simplicity, expressiveness and conciseness. We have also been able to measure and observe the benefits in terms of an increase by a factor of three to four in the speed of development times for the coding and unit testing phases, as well as the ease of and potential for re-use of the IMS DSL in different projects. Lastly we have received positive feedback regarding the ease of use, and the simplicity and clarity of the code produced with the IMS DSL.

This positive outlook will be further pursued in the coming year as we will evaluate the potential of an IMS DSL embedded in the action language of a UML based Model Driven Development workflow.

At this point in time, the IMS DSL has been developed as a proof of concept to showcase the potential benefits of the DSL approach. The projects conducted using the IMS DSL were also proofs of concept. It is obvious to us that productizing the IMS DSL would involve a great deal of work, especially to integrate it in the Ericsson software infrastructure. However, the benefits observed at least warrant the study of the business case of doing so. Based on our experience, the main hindrance to the development of a DSL is one's ability to accept the DSL paradigm and maintain discipline to avoid falling back on developing it as he would any other software library.

References

1. Internet Engineering Task Force: Request for Comments (RFC) Pages, `http://www.ietf.org/rfc.html`
2. World Wide Web Consortium:Standards – W3C, `http://www.w3.org/standards/`
3. International Telecommunication Union: Gateway control protocol, `http://www.itu.int/rec/T-REC-H.248.1/en`
4. Hunt, A., Thomas, D.: The Pragmatic Programmer – From Journeyman to Master, pp. 70–76. Addison Wesley (1999)
5. Wikipedia: Domain-specific language, `http://en.wikipedia.org/wiki/Domain_specific_language`
6. van Deursen, A., Klint, P., Visser, J.: Domain-Specific languages. SEN-R0032 November 30 (2000), `http://homepages.cwi.nl/ paulk/publications/ Sigplan00.pdf` ISSN 1386-369X

7. Raja, A., Lakshmanan, D.: Domain Specific Languages. International Journal of Computer Applications 1(21) Art. 18 (2010); Foundation of Computer Science, http://oaj.unsri.ac.id/files/wwwijcaonline/journal/number21/pxc387640.pdf
8. Taha, W.M.: Domain-Specific Languages. Plenary Presentation paper for 2008 IEEE International Conference on Computer Engineering and Systems (ICCES 2008) (2008), http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.139.6989&rep=rep1&type=pdf
9. Mernik, M., Heering, J., Sloane, A.M.: When and How to Develop Domain-Specific Languages. ACM Computing Surveys (CSUR) 37(4), 316–344 (2005), http://www.rose-hulman.edu/Users/faculty/young/OldFiles/CS-Classes/OldFiles/csse490-mbse/Readings/DSL-Survey-WhenHow.pdf
10. Hudak, P.: Modular Domain Specific Languages and Tools. In Proceedings of the 5th International Conference on Software Reuse (ICSR 1998). IEEE Computer Society (1998), http://www.cis.uab.edu/courses/cs793/spring2010/dsel-Hudak.pdf
11. École Polytechnique Fédérale de Lausanne (EPFL): The Scala Programming Language, http://www.scala-lang.org/
12. Ericsson: More than 50 billion connected devices, http://www.ericsson.com/res/docs/whitepapers/wp-50-billions.pdf
13. Arduino, http://www.arduino.cc
14. Hardkernel Co., Ltd: Odroid-X, http://www.hardkernel.com/renewal_2011/products/prdt_info.php?g_code=G133999328931

Efficient Development of Domain-Specific Simulation Modelling Languages and Tools

Andreas Blunk and Joachim Fischer

Humboldt-Universität zu Berlin
Unter den Linden 6
D-10099 Berlin, Germany
{blunk,fischer}@informatik.hu-berlin.de

Abstract. This paper presents an approach which facilitates efficient development of domain-specific simulation modelling languages and tools for discrete-event systems. The work is motivated by a set of properties which in combination are not well supported by established frameworks. These include the provisioning of object-oriented description means, means for specifying domain-specific concepts with a distinct notation and semantics, the possibility of including general-purpose concepts into domain-specific ones, low cost tool support including an editor, a debugger, and a simulator, simulation primitives with fast execution, and extensibility means for enabling access to externally implemented simulation-specific functionality. We present a prototype that partly implements these properties. It combines established techniques derived from metamodel-based language development and extensible simulation modelling. The value is demonstrated by applying the approach to an example language from the domain of reactive systems and by comparing it to related approaches.

1 Introduction

While simulation modelling is as old as computational machinery, we are still learning how to best utilise it alongside modelling and experimentation. As the availability of low-cost computational power increases and the complexity of systems grows, the importance of simulation rises as well. The investigation of efficient, robust, explorative and problem area-adapted simulation methodologies is an important part of this activity.

Research in simulation methodology ranges from the development of efficient and effective algorithms, tools and programming languages, to the creation of new software engineering technologies, visualisation, data processing and storage methods, and even the philosophy and epistemology of simulation modelling [1].

1.1 Objective

In this paper, we argue in favor of an approach that allows to *efficiently develop domain-specific simulation modelling languages and tools* for discrete-time event-driven systems which in addition allow for fast simulations. The approach

F. Khendek et al. (Eds.): SDL 2013, LNCS 7916, pp. 163–181, 2013.

complies with two general desirable properties of simulation systems:

1. A system model should be expressed with structural and behavioural equivalence to the original system and
2. simulations should be executed efficiently.

In the next paragraph, we present accepted simulation systems which implement these two general properties and what we can learn from them.

1.2 Lessons Learned and Challenges

The introduction of model abstraction concepts by Simula [2], which later became known as *object-oriented modelling concepts*, is a major step towards achieving structural and behavioural equivalence [3]. With its class concept, Simula introduces the powerful principles of classification/exemplification and generalisation/specialisation. In addition, class instances (objects) can be divided into passive and active objects. Active objects are combinations of states and actions, which cause state changes in dependence of a model time.

However, object-orientation combined with expression means provided by a universal modelling language are not sufficient to concisely capture *domain-specific concepts*. This is because a universal modelling language defines a strict syntax in which domain-specific concepts have to expressed in. There are cases in which this preset syntax hinders application and understanding of a concept. An example is the implementation of state machines by an object-oriented pattern [4]. Applying this pattern results in a behaviour description spread over several classes which have to be created for each state of a state machine.

One possible solution to having a distinct syntax is to *extend* the syntax of a language combined with semantic foundations in the host language. The Simulation Language with Extensibility (SLX) [5] is a modern simulation language that has such possibilities, although they are limited.

Above all, we can see that a clear structure in the model alone is not causing a simulation language to be widely accepted. An assembly level language, whose many versions enjoyed great popularity since decades, is Gordon's GPSS (General Purpose Simulation System) [6]. The reason for its widespread use is (despite its structural weaknesses) the *efficient realisation of the next-event-scheduling* paradigm, which also takes so-called state events into account.

When we look at simulation languages today, we observe that both of these basic aspects can be incorporated in one language. This is achieved in languages like SLX, Modelica [7], and ODEMx [8], regardless of whether the modelled system is continuous or discrete in time.

Another aspect of simulation modelling is *experiment design*. Current research suggests that purely declarative ways of describing experiments [9] can already be sufficient to derive complete scientific experimentation workflows. These workflows are models of the experiment processes. They cannot only serve as specifications for repeated experimental procedures but also as inputs for workflow

engines that trigger complex series of experiments, and that monitor and automatically evaluate their results. It is obvious that different descriptions are necessary:

(i) description of a parameterised simulation model,
(ii) description of the experiments,
(iii) description of the computational platform for execution and evaluation [10].

Equally desirable are *programmatic interfaces* to existing programming languages inside a special simulation language. They allow us to reuse statistical methods, solvers, or optimization methods that have already been implemented [11].

Self-standing solutions already exist for many of the above mentioned issues. However, there is no combined approach due to major technological differences. A recent development promising to facilitate the combination of such different aspects is object-oriented metamodelling (OOMM) [12]. In OOMM, the central part that connects all other aspects is an object-oriented metamodel. It defines the concepts of a language in an abstract way.

Other language aspects are defined with specialised language description languages, e.g. textual notation, static semantics, and execution semantics. For each such language aspect, tools can be derived automatically. Thus, OOMM is a key technology for creating domain-specific modelling languages (DSMLs or DSLs) and tools at an acceptable cost. In addition, DSLs allow models of dynamic systems to be expressed in a more concise way and with increased structural and behavioural equivalence than with object-oriented description means alone.

However, adding domain-specific concepts alone is not enough. One often needs to include or mix concepts of general-purpose programming languages, e.g. expressions and statements, with domain-specific ones. Therefore, we believe that an approach is required that allows to combine general-purpose as well as domain-specific concepts. These ideas are also described in [13], in which the authors propose to develop a unified approach that can be applied to combine different established concepts of modelling and programming. Our work also explores this direction of a combined approach. However, we believe that the efficient development of new language concepts and tools is a key requirement here.

1.3 Properties of Our Approach

Our approach exhibits a number of positive properties regarding the efficient development of domain-specific simulation modelling languages and tools.

It allows to create models with increased structural and behavioural equivalence by providing means for defining domain-specific concepts with a distinct notation and semantics. At the same time it allows to efficiently execute simulations, which make use of such concepts. This includes the possibility to attach already existing and efficiently implemented functionality, e.g. implementations of random number generators.

Adding domain-specific concepts to simulation modelling languages requires a frame in which they can be correctly applied. This frame can be provided by an object-oriented base simulation language. In addition, modelling concepts often make use of programming concepts. Therefore, an approach that allows to combine modelling and programming concepts is desirable.

However, for such an approach to be practical at all, a general requirement has to be fulfilled. The development of a new domain-specific simulation modelling language has to pay off, i.e. development effort including customised tools has to be low. The approach we present fulfills this requirement. It is based on a general simulation language, which can be dynamically extended by domain-specific concepts and which provides immediate support by a customised editor and simulator.

We summarise these positive properties, which our approach partly already implements, for further reference as follows. The approach exhibits

- (P1) object-oriented description means,
- (P2) means for specifying domain-specific concepts with a distinct notation and semantics,
- (P3) the possibility of including general-purpose concepts in domain-specific ones,
- (P4) low cost tool support including an editor, a debugger, and a simulator,
- (P5) simulation primitives with fast execution,
- (P6) and a programming interface enabling access to and from simulation-specific but externally implemented functionality with high efficiency.

1.4 Objective and Outline

We present a prototypical implementation of our approach by a framework named DMX (Discrete-Event Simulation Modelling Framework with Extensibility) [14] which implements properties (P1)-(P4) and partially (P5). DMX combines established techniques of metamodel-based language development and extensible languages. It consists of an extensible object-oriented base language and mechanisms for automatically deriving tools at a low cost. Property (P6) will be discussed as part of our future work. Furthermore, we confine our work to textual languages.

The paper is structured as follows. Section 2 gives an overview of related work that implements some of the properties. This motivates the presentation of our combined approach in Sect. 3. We introduce the concepts of the extensible base language and present the prototypical implementation of the framework. The value of the approach is demonstrated by applying it to an example language from the domain of reactive systems – a first use case on the path to more complex languages. In Sect. 4, we briefly discuss the fulfillment of each property and we compare our approach with two related ones in which we define the same example language. The comparison also takes into account development effort, which we identify as a general requirement of such approaches. We conclude the paper in Sect. 5 and discuss future work in Sect 6.

2 Related Work

In this section, we discuss related work which considers some of the proposed properties. Most of them have a more general focus and are not specifically designed for creating simulation languages. We give an overview of these works and point out their important characteristics.

The related work can be divided into 4 branches: extensible programming languages, extensible simulation languages, comprehensive DSL development frameworks, and further approaches for describing the execution semantics of metamodel-based languages.

Extensible Programming Languages became popular in the 1970s. The technique keeps being applied to state-of-the-art programming languages [15]. There is some recent works for the Java and the C++ language. An example is the Java Syntactic Extender (JSE) [16]. It is a pre-processor for Java allowing to add extensions of a few syntactic shapes. In addition, extensions may include certain Java constructs. Despite limitations in the syntax of extensions, customised modelling tools cannot be derived.

Extensible Simulation Languages are rare. The only such language that we know about is the Simulation Language with Extensibility (SLX) [5]. SLX is an object-oriented language that has a small but powerful C-like kernel language, in which constructs of the C language which are prone to error or primarily intended for systems programmers are excluded or restricted. On the contrary, discrete event simulation primitives for expressing concurrency, scheduling, and synchronization are added. In addition, the language can be extended by new statements and expressions with a distinct notation and semantics. The class of languages that is supported is a subset of regular languages. Semantics is defined as a mapping to SLX itself, which is the foundation for runtime efficient simulation execution. SLX has an efficient implementation of time-delays as well as state events and unconditional blockages with explicit reactivation. This is achieved by a specially developed compiler that, for simulation, has advantages regarding execution speed compared to compilers of general-purpose languages.

Comprehensive DSL Development Frameworks with the possibility of including general-purpose concepts into DSLs are a more recent trend. Outstanding representatives are Xtext [17] and the Meta Programming System (MPS) [18].

In Xtext [17], development starts with a concrete syntax from which a metamodel is derived. Semantics have to be described as a mapping to Java. This is achieved by writing a transformation which programmatically works on a Java program represented as an abstract syntax tree.

MPS [18] is more powerful than Xtext because extensions can be used jointly with DSL concepts. However, the editor is a projectional one that is unusual to operate. One does not enter the single characters that make up a construct, but instead has to choose from a set of possible constructs insertable at the current cursor position. Then the fixed textual parts of a construct are expanded and the

cursor can be moved from one variable part to the next. For example, for a class construct, one can move from the name to other variable parts like attributes and operations and instantiate further constructs there.

Both frameworks automatically provide text editors. However, these are only available after a manual software generation step. The generation has to be initiated every time the language is changed. This manual step hinders rapid development of DSLs. Furthermore, both frameworks do not provide simulation primitives. These have to be made available manually by a simulation library.

There are Some Alternative Approaches for Describing the Execution Semantics of Metamodel-Based Languages. Some of them are based on operational semantics, e.g. MAS [12], M3Actions [19], and EProvide [20]. Runtime data and runtime states are described as a part of the same metamodel that also defines the abstract syntax of a language. Semantics is described by stepwise transformations of the runtime state. For such descriptions, programming languages like Java, but also UML activity diagrams or languages like Prolog and Haskell can be used. However, these approaches do not consider the necessity of runtime efficient executions of simulations. Furthermore, simulation primitives are not available and have to be added manually.

3 Approach

Our approach is based on a framework that combines an extensible object-oriented language with the immediate provisioning of essential tools at a low cost. In this section, we describe the basic concepts of the approach and its implementation. In the following, we present a use case that applies the approach to the definition of a state machine language. These explanations lay the foundation for a discussion of the approach regarding the fulfillment of the proposed properties in the next chapter.

3.1 Basic Concepts

Object-Oriented Concepts. At its core, the approach consists of a base language (BL) that includes object-oriented description means and a small set of essential simulation primitives. Its object-oriented features are confined to single inheritance between classes and multiple inheritance between interfaces combined with the well-established concept of type polymorphism. The base language and its concepts are similar to those of SLX. However, in contrast to SLX, the concepts of pointers to an object and an object as a value are not distinguished. Variables of type class are always handled as references to objects. This is a simplification derived from Simula and Java.

Simulation Concepts. The set of provided simulation primitives is short but sufficient. Concurrent processes are defined by the concept of an *active class*. Each such class defines the behaviour of its objects as a sequence of statements in

an *actions part*. The behaviour starts when objects of active classes are explicitly *activated*. Further statements specify event-based process interactions: *advance* of model time, indefinite *waiting* of a process and *reactivation* by another one, *interruption* of a process and *rescheduling* it at a certain time, *yielding* control to another process, and *waiting for a certain condition* (defined as an expression accessing model structures) to become true (wait until). These are simulation primitives known to be sufficient for modelling all kinds of discrete-event systems. The provided simulation primitives are inspired by DEMOS [21] and SLX [5].

Domain-Specific Concepts. Domain-specific concepts are defined by specifying extensions to the base language. An extension specification consists of two parts. First, an extensions syntax is specified and then a mapping to concepts of the base language is defined.

Syntax Definition. The syntax is defined in an attributed BNF-like grammar language. A syntax definition consists of a set of grammar rules that extend the grammar of the BL. The first of these rules refers to the BL grammar rule that is extended by a new rule. Subsequent rules, consisting of terminals and non-terminals, define the concrete syntax of an extension. Each rule may refer to already existing BL rules, e.g. Statement and Expression, and thus reuse BL constructs. Non-terminals prefixed by a dollar sign designate references to already existing language constructs.

The class of languages that can be defined is a subset of context-free languages which can be defined by an LALR[1] grammar. Furthermore, semantic additions for specifying references between language constructs can be made by using the dollar sign in front of non-terminals.

In the definition of an execution semantics, the syntax parts have to be accessed. Therefore, syntax parts are prefixed by symbolic names. These names allow to access and evaluate the elementary or structured value of a part in semantics definitions. In addition, prefixed syntax parts define an implicit mapping to an abstract syntax. This abstract syntax is internally represented as a metamodel which extends the metamodel of the BL.

```
extension ForLoop {
   Statement -> ForLoop ;
   ForLoop -> "for" "(" it:$Variable "in" set:Expression "with"
       condition:Expression ")" "{"
           ManyStatements
       "}";
   ManyStatements -> ;
   ManyStatements -> statements:list(Statement) ManyStatements;
}
```

Listing 1. Syntax definition of a for-loop statement as an extension

[1] Look-Ahead Left to Right, Rightmost derivation.

An example is given in Listing 1. The extension defines the syntax of a for-loop as an additional kind of statement. In contrast, to the BL for-loop that iterates over all elements of a given set, the for-loop extension can be equipped with a condition expression which selects specific elements of the set. For the non-terminal $Variable, an identifier referring to an already existing Variable construct has to be supplied.

The example also shows syntax parts prefixed by symbolic names. An example is the condition expression which is prefixed by the name condition. In the semantics definition, the concrete condition can be accessed by this name.

In [22], we introduce the parts of the approach that deal with syntax extensions. We also show a prototype that implements this aspect. It allows to syntactically extend the concepts of a general-purpose language by domain-specific ones.

Semantics Definition. Models created in the BL are used for simulation. Therefore, domain-specific concepts have to define an execution semantics. The semantics of a concept is defined by a mapping to concepts of the BL. Furthermore, the semantics of the BL concepts are informally defined by a mapping to an executable target simulation language, i.e. an existing language for which there is already a compiler. This can be an exclusive simulation language like SLX but also a general-purpose language like Java combined with a simulation library. The sole requirement for the target language is that one can write a mapping for each of the concepts of the BL.

The semantics of an extension is defined in a *semantics part* right below the syntax part. The mapping is defined by a sequence of regular BL statements combined with a special gen statement. For each concrete use of an extension, these statements are executed. The target BL code is derived from executions of the gen statements. In the next step, the resulting target BL code is included at the exact place where an extension is used. In addition, there is a special statement for changing the current substitution context to other parts of a BL model. The concept is very generic in the way that arbitrary constructs of the BL can be referred to by their abstract syntax definition. As an example, an extension of type statement could add a class definition in a BL module that is required in the substitution code of the statement extension itself.

```
extension ForLoop {  } semantics {
    gen "for (" it ":" set ") {";
    gen "if (" condition ") {";
    for (Statement stm: statements) {
        gen "" stm ";";
    }
    gen "} }";
}
```

Listing 2. Semantics definition of a for-loop statement as an extension

In Listing 2 we revisit the example of the for-loop statement again and present the definition of its semantics. Executing this definition results in a regular BL for-loop in which the conditional selection of elements is simply implemented by encapsulating the statements of the for-loop body in an if statement. Syntax parts like it and condition are implicitly replaced by their concrete syntax representation when used inside semantics parts. An example use of the for-loop extension and the corresponding target code is depicted in Listings 3 and 4.

```
list(int) is; int i; for (i in
      is with i > 0) {
   print i;
}
```

Listing 3. Example use of the for-loop extension

```
list(int) is; int i; for (i:
     is) {
   if (i > 0) {
      print i;
   }
}
```

Listing 4. Resulting BL target code for an example use of the for-loop extension

3.2 Implementation

Editor. The BL editor is implemented by using the Textual Editing Framework (TEF) [12] and the Eclipse Modeling Framework (EMF) [23]. TEF is used for deriving a BL editor by defining the concrete syntax of the BL. EMF is used for the definition of a metamodel for the BL, which is required by TEF, and also for the extensions, which make additions to the BL metamodel.

The outstanding feature of the editor is its immediate awareness of the syntax of domain-specific extensions. This feature automatically derives a DSL editor at runtime. It offers well-established editor features like syntax highlighting and content assistance. Its implementation is feasible because TEF is based on the runtime parser generator RunCC [24], which can be supplied with changing versions of a grammar at runtime. This makes the implementation of a TEF variant feasible in which extension definitions are instantly recognised by the BL editor. For each extension, the grammar rules defined by the extension are added to the grammar of the BL. The extended BL editor and its parser continue to work with the extended version of the grammar. When an extension definition is modified, the corresponding rules in the BL grammar are updated as well.

Simulator. The BL simulator is implemented by a mapping to an executable target simulation language. The simulator is derived by compiling and executing the target language program.

BL concepts as well as simulation concepts have to be considered in the mapping description. In a first prototype, a mapping to Java in combination with the simulation library DESMO-J is defined. The mapping is described in Acceleo [26], which is a template language implementing the OMG MOF Model

to Text Standard [27]. This concrete mapping is 513 LOC[2] in size, composed of template statements and target code.

DSL simulators are derived by substituting all the extensions with BL concepts as defined in their semantics parts. At the end of this process, the resulting model solely consists of BL concepts. In the final step, the model is mapped to the target simulation language that is used for execution. An overview of the whole compilation workflow is available online [14].

A major problem in the whole transformation process is the rather static nature of the used metamodeling framework EMF, as explained in [22]. In summary, it is hard to have a changing metamodel be supported by EMF at runtime. Because of this problem, the time for executing a transformation is rather long. For the simple for-loop example, it takes around 8 seconds to transform, compile, and execute a corresponding model on a high-end computer[3]. Initiating the EMF generation process for creating all the Java classes, which have to be present for metaclasses, takes most of the overall execution time.

3.3 An Example Language – State Machines

We successfully apply the approach to the definition of state machines as an example language, which we refer to as SML. The simplified use case of this language is designed as a preliminary study for the development of a language for industrial workflows in the field of supply chain management.

SML is defined by a set of domain-specific extensions. The language is an enhanced version of the one presented in [22]. It is a subset of state machines as defined by the UML [28]. The subset includes simple states, initial and final states, transitions with signal, completion, and time event triggers, and also guards and effects. The semantics of event processing is implemented as run-to-completion as defined by UML. SML state machines can be used to define behaviour inside an active class of the BL. They can also refer to properties of their enclosing class.

An example, which defines the structure and the behaviour of a simple counter, is depicted in Listing 5. After each time step, a count variable is increased until a certain limit is reached. In addition, the counter may be started, paused, and resumed by external signals. Signals are send to objects by a special send statement, which is also defined as an extension.

The syntax definition is similar to the one presented in [22]. The new semantics definition has been added. It is defined by using the core simulation constructs of the BL. An excerpt of the semantics definition is depicted in Listing 6. The complete definition of SML is 158 LOC in size. It is available online [14].

The main idea in the semantics definition is to add a variable of type list of object to the class containing the state machine. This variable serves as an event

[2] Description effort is measured in lines of code (LOC), excluding comments and blank lines.

[3] Intel Core i7 2.6 GHz processor, 8 GB main memory, and a Solid State Hard Disk.

pool into which signal and time events are placed. In the next step, the behaviour is implemented by waiting for events to arrive in this set. Each event is processed one after the other, including the evaluation of guards and the execution of effects. For time events, a special active class Timer is created by using the special statement setGenContext. It allows to change the current generation context inside semantics parts. This concept allows to add constructs in other places of the enclosing model in which an extension is used. In the example, the class Timer is created in the same module in which the state machine is used. Objects of class Timer represent time events. When a time event occurs, a Timer object is placed into the corresponding event pool.

```
class Start {} class Pause {} class Resume {} class Reset {}


active class Counter {
    int count;
    int limit = 10;
    int step = 1;


    stateMachine CounterBehaviour {
        initial -> StandBy;
        state StandBy (
            Start / { count=0; } -> Active,
            Resume -> Active
        );
        state Active (
            [count >= limit] / { trace("Finished."); } -> final,
            after(step) [count < limit] / {
                count=count+1; trace("Tick " + count); } -> Active,
            Pause -> StandBy,
            Reset / { count=0; } -> Active
        );
    }
}

void main() {
    Counter c = new Counter;
    activate c;
    send new Start to c;
    advance 20;
}
```

Listing 5. Counter state machine as an example extension use

```
semantics {
       gen "list(Object) eventPool; string currentState;";
...
       gen "actions { ... while (currentState != null) {  ";
...
       gen "  empty eventPool;
           wait; ...
           while (eventPool.size > 0 and currentState != null) {
              Object ev = eventPool.first;
              remove ev from eventPool; ";

       for (State state: states) {
           gen "if (currentState == \"" state.name "\") {";
           for (Transition tr: state.outgoing) { ...
              if (tr.effect != null) {
                 for (Statement stm: tr.effect.getStatements()) {
                    gen "" stm ";";
                 }
              }
...

       ClassContentExtension ext = self;
       Clazz clazz = ext.eContainer() as Clazz;
       Module mod = clazz.eContainer() as Module;

       setGenContext after mod.getClassifiers().first;
       gen "active class Timer {
           " clazz.getName() " sm; ...
           actions {
              advance delay; ...
              place self into sm.eventPool; ...
           }
       } ";
}
```

Listing 6. Excerpt of the semantics definition for the state machine extension

4 Discussion

In this section, we discuss to what extent the proposed properties are present in our approach. We also compare the approach to two related ones, namely SLX [5] and Xtext [17]. We investigate to what extent each property is present in each approach. We relate this investigation to the general requirement of having low development effort for domain-specific concepts including customised tools. We measure description effort in LOC and point out characteristics of each approach.

4.1 Object-Orientation (P1)

The base language (BL), which is included in DMX, is object-oriented. Thus, property (P1) is present. Object-orientation is restricted to single class inheritance. This kind of object-orientation is used in various established modelling languages like Simula, SLX, and the System Description Language (SDL) [29]. Although single inheritance seems to be sufficient, other languages like Modelica and UML [28] provide multiple inheritance instead.

However, UML [28] does not define a precise semantics and leaves this part unresolved as a semantic variation point. Problems arise when multiple implementations of the same kinds of elements are inherited. Modelica solves this problem by merging the contents of the base class and the derived class. Thus, similar elements become one. This is a feasible solution in Modelica because operations, which can be a source of ambiguities, cannot be defined as parts of classes but only as functions on a global level. Therefore, ambiguities as a result of inheriting the same operation multiple times with different implementations cannot occur. Multiple inheritance can be a helpful feature. However, further investigation is required of how it should be supported to be a helpful instrument.

Xtext includes a base language named Xbase, which defines a large set of expressions and statements. The semantics of Xbase is defined as a mapping to Java. Therefore, Xbase also takes over Java's type system, i.e. single class and multiple interface inheritance. However, object-oriented descriptions means for defining structures like classes and relations are not present in Xbase directly. These have to be defined in Java and then referenced from languages defined with Xtext and Xbase.

Another aspect of the BL is its simplification of the type concept for class typed variables to object references only. This simplification exempts the modeller of considering runtime efficiency aspects, i.e. whether an object should be placed on the stack or on the heap. This kind of decision is intentionally left to an optimizing compiler. In SLX, there is no such simplification. In Xtext, as a result of its Java-based semantics, the same simplification is present.

4.2 Domain-Specific Additions (P2 and P3)

Modelling with increased structural and behavioural equivalence is achieved by the extension concept. It allows to define domain-specific concepts with their own notation and execution semantics (P2). The advantage of an extensions-based approach is that concepts of the base language can be included into domain-specific ones (P3). In addition, BL concepts and extensions can be used jointly within the same model.

In SLX, domain-specific concepts of two kinds can be defined with their own notation and semantics: statements and expressions. The syntax definition of such concepts is bound to a subset of regular languages, which is not suitable for complex DSLs. Therefore, property (P2) is only present in parts.

In addition, DSL concepts can only include SLX expressions, but they cannot include other kinds of contructs like SLX statements. This is a major obstacle in defining effects in the example language SML. In SLX, one has to code them as strings. Therefore, property (P3) is only present in parts as well. The execution semantics of an extension is defined by a sequence of SLX statements in combination with a special expand statement. They define a mapping to the SLX core language. The target code is derived by executing this mapping for each extension instance and replacing it by its target code. The availability of simulation primitives make the description of an execution semantics surprisingly short. The complete definition of SML is 103 LOC in SLX.

In DMX, the set of languages that can be defined is more comprehensive. It already allows to define such complex languages as state machines. Important features making this definition feasible is the support of context-free languages in extensions. In addition, DMX includes a special statement setGenContext which allows to change the current generation context inside semantics parts. This concept allows to add constructs in other places of the enclosing model in which an extension is used. Such constructs can be helpful when defining semantics. As an example, in the semantics definition of SML, a class Timer is created in the same module in which a state machine is used. In the semantics definition the class Timer is used in order to define the semantics of time events. In contrast, SLX only allows to create constructs at the same place in which an extension is used, which limits the possibilities of defining certain semantics. The size of the SML definition is with 158 LOC in DMX compared to 103 LOC in SLX slightly larger. Yet, language definition as well as editor support are more comprehensive as well.

In Xtext, DSLs in the set of context free languages can be defined, thus property (P2) is present. In addition, general-purpose constructs like expressions and statements can be added to a language, which is property (P3). However, there is a major problem with Xtext: domain-specific additions cannot be embedded into regular Java programs and thus cannot extend the Java language. In the semantics description, expressions have to be positioned in a suitable place in the resulting Java code, where they can be correctly evaluated. As Java does not contain simulation primitives, this part of the mapping has to be defined by using a Java-based simulation library. In this comparison, DESMO-J [25] is used. The definition of SML in Xtext consists of 373 LOC (15 LOC for the syntax and 358 LOC for the semantics definition).

In DMX, domain-specific concepts can be directly embedded into a BL model. In addition, they can be used jointly with the BL. The size of the syntax definition of SML is with 25 LOC in DMX compared to 15 LOC in Xtext slightly larger. However, this can be explained by Xtext providing EBNF and our own approach providing BNF for syntax definition. With the semantics definition it is different. In Xtext the size is 358 LOC, which is more than double the size of the definition in DMX (158 LOC). There are multiple reasons for this increase in description size in Xtext: i) the programmatic construction of target code as

a Java abstract syntax tree, ii) the missing integral part of simulation primitives, and iii) the need to embed BL elements like statements and expressions into a suitable evaluation context. In addition, semantics is defined in Xtend, which in comparison to Java includes many simplifications to Java as well as additions of high-level concepts like lambda expressions. This makes descriptions considerably more concise than comparable Java code.

The use of BNF notation in DMX syntax parts could be reduced by adding Extended BNF (EBNF) description means. Because EBNF is already defined as an extension of BNF, it could be made available by the very same principles of our approach. However, such an extension is of a different kind. In order for the editor to recognise EBNF-based syntax definitions, it is required to carry out extension substitution for the semantics part at runtime as well. Currently this is only implemented for the syntax part.

4.3 Low Cost Tool Support (P4)

In our approach, there is immediate tool support with an editor and a simulator. This makes the definition and the application of a domain-specific concept with a distinct notation as simple as writing an ordinary method. Editor features like syntax highlighting and content assistance, available for methods, are equally present for domain-specific extensions. All of these tools are provided at a low cost because the description effort required for defining syntax and semantics is low in comparison to SLX and Xtext. Thus, property (P4) is present.

In SLX, there is an integrated programming environment including an editor, a compiler, a launcher, and a debugger. These tools also support defined extensions. However, the syntax of extensions is not immediately recognised. Instead, the syntax is highlighted after a complete program has been compiled successfully. Expressions and statements used in extensions have to be supplied as unchecked strings.

In Xtext, a textual DSL editor can be generated from a DSL description. It features syntax highlighting and content assistance. In contrast to SLX, there is even support for expressions and statements used in extensions. There is a generic builder, which executes the mapping description and automatically compiles the resulting Java code. The Java representations of DSL constructs are available in regular Java programs. However, the development process is rather slow because tools have to be generated first before they can be applied. Especially in an iterative process, this kind of development is time-consuming. In addition, description effort is rather high.

In comparison to SLX, tool support by an editor is more comprehensive in DMX. Editor features likes syntax highlighting and content assistances are equally available as in Xtext. However, in DMX, an editor is immediately available for domain-specific concepts. In addition, the description effort required is lower than in Xtext.

In each approach, a compiler for domain-specific extensions is automatically derived from semantics descriptions. The compilers differ in compilation time. In DMX, compiling a BL program which includes extensions is rather slow at the current stage of implementation. Compiling the example counter state machine (Listing 5) takes around 10 seconds.

4.4 Simulation Primitives with Fast Execution (P5)

In DMX, runtime efficient executions are preserved by efficient implementations of the BL simulation primitives. This part is delegated to the selected simulation target language. For some of these languages efficient implementations of compilers already exist. An impressive example is SLX. By writing a mapping of the BL to SLX, one can benefit from fast executions combined with an increased expressive power in defining domain-specific extensions. Also, extensions benefit from fast executions because their semantics are defined using the very same concepts of the BL.

The same argument holds for simulation libraries written in Java, although they may not be as runtime efficient. An example with poor execution speed is DESMO-J. That is because its coroutine implementation is based on Java threads. However, there are more efficient libraries. A library which is prominent in the network simulation community is JiST [30]. It implements coroutines by Java Byte Code rewriting which makes it faster than DESMO-J. A mapping to JiST, which would be similar to the one already implemented for DESMO-J, can result in a viable simulator as well.

Execution time of SLX and Xtext/DESMO-J is measured in an experiment[4] with an example model which includes two counters (as presented in Sect. 3.3), limited to 10^6 counts. In SLX, the simulation is finished after 0.06 seconds. In Xtext/DESMO-J, it takes 186 seconds to complete. The corresponding DMX model executes in 144 seconds when using DESMO-J. There is a slight increase in execution time in the Xtext-based SML. This might be because the state machine semantics for processing events are implemented in an object-oriented way by following a state machine pattern [4]. In contrast, the semantics of the DMX-based SML is defined by determining the current state with a number of simple if statements.

Furthermore, the execution time of DMX could easily be increased to the same time as pure SLX by defining a BL-to-SLX transformation. Thus, DMX could benefit from the very efficiently implemented SLX simulation core. In addition, DMX models already created can be used without any changes.

This is only a first measurement with a simple example which focusses on process switching times. Depending on the concrete model, other aspects might be more important. Nevertheless, the example can serve as a first indicator of execution times.

[4] Intel Core i7 2.6 GHz processor, 8 GB main memory, and a Solid State Hard Disk.

4.5 Externally Implemented Functionality (P6)

The possibility of connecting different target simulation languages (mentioned in Sect. 4.4) is important for several reasons. It is important i) for creating models which are independent of some current state-of-the-art simulator platform, ii) for running simulations with the most efficient simulator platform available, and iii) for integrating external functionality provided by libraries or tools without too much effort. The last remark can be an implementation of property (P6).

Although we have not yet investigated this aspect in depth, we believe that the BL to target language mapping already offers a good solution. It should be feasible to access external functionality implemented by tools written in the target language with not much effort. One can declare a BL function as native in which calls to this function are forwarded to their implementation in the target language. A prerequisite is that an external tool has to offer an interface to its functions accessible in the chosen target language.

In SLX, external functions implemented in C/C++ can be invoked. This is achieved in a number of steps. A model has to 1) declare a function as natively available via a Dynamic Link Library (DLL), 2) generate a C/C++ header and implementation file, 3) implement the function in C/C++, and 4) compile it as a DLL so that it can be accessed from a SLX model.

In Xtext, external functions implemented in Java are instantly accessible because its semantics are already defined as a mapping to Java.

5 Conclusions

We present an approach exhibiting a number of properties which are important in order to develop domain-specific languages used in simulation in a more efficient way. This includes the development of the language as well as its tools. We measure description effort of our approach for defining a state machine language and present the derivation of tools like an editor at a low cost. In comparison to related approaches, description effort and the cost for having tools is reduced while maintaining the expressive power and execution efficiency required for domain-specific simulation languages. Further research has to show if languages that are even more complex can be developed analogously.

6 Future Work

The provisioning of debuggers is an area of special interest to us. Currently, debuggers still have to be implemented by hand, e.g. for simulation languages like SLX. Furthermore, dedicated debuggers do not even exist for simulation libraries written in programming languages. We believe that our approach can be extended to an immediate provisioning of DSL debuggers (part of P4). In [31], we already describe the debugging aspect of a DSL, so that a debugger can be derived automatically. The work is based on EProvide, which is a framework

for defining the execution semantics in an operational way. However, other approaches [32] show that such tools could also be derived for execution semantics described as transformations.

The second area of interest is investigating the integration of externally available functionality as described in Sect. 4.5.

References

1. Law, A.M.: Simulation Modeling and Analysis. McGraw-Hill (2007)
2. Dahl, O.J., Nygaard, K.: SIMULA: an ALGOL-based simulation language. Communications of the ACM 9(9), 671–678 (1966)
3. Møller-Pedersen, B.: Scandinavian Contributions to Object-Oriented Modeling Languages. In: Impagliazzo, J., Lundin, P., Wangler, B. (eds.) HiNC3. IFIP AICT, vol. 350, pp. 339–349. Springer, Heidelberg (2011)
4. Shalyto, A., Shamgunov, N., Korneev, G.: State Machine Design Pattern. In: NET Technologies 2006 Short Communication Papers Proceedings, pp. 51–58 (2006), http://dotnet.zcu.cz/NET_2006/Papers_2006/!Proceedings_Short_Papers_2006.pdf
5. Henriksen, J.O.: SLX – The X is for Extensibility. In: Proceedings of the 32nd Conference on Winter Simulation (WSC 2000). Society for Computer Simulation International (2000), http://informs-sim.org/wsc00papers/027.PDF
6. Gordon, G.: The development of the General Purpose Simulation System (GPSS). ACM SIGPLAN Notices 13(8), 183–198 (1978)
7. Fritzson, P.A.: Principles of Object-Oriented Modeling and Simulation with Modelica 2.1. John Wiley & Sons (2004)
8. Fischer, J., Ahrens, K.: Objektorientierte Prozeßsimulation in C++. Addison-Wesley (1996)
9. Bowers, S., McPhillips, T., Ludäscher, B., Cohen, S., Davidson, S.B.: A Model for User-Oriented Data Provenance in Pipelined Scientific Workflows. In: Moreau, L., Foster, I. (eds.) IPAW 2006. LNCS, vol. 4145, pp. 133–147. Springer, Heidelberg (2006)
10. Kühnlenz, F., Fischer, J.: A Language-centered Approach for Transparent Experimentation Workflows. In: Proceedings of the CSSim 2011-Conference on Computer Modelling and Simulation (2011), http://metrik.informatik.hu-berlin.de/grk-wiki/index.php/Expwf
11. Zeigler, B.P.: Theory of Modeling and Simulation, 2nd edn. Academic Press (2000)
12. Scheidgen, M.: Description of Computer Languages Based on Object-Oriented Meta-Modelling. Doctoral Thesis, Humboldt University Berlin (2008), http://www2.informatik.hu-berlin.de/~scheidge/downloads/thesis.pdf
13. Madsen, O.L., Møller-Pedersen, B.: A Unified Approach to Modeling and Programming. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part I. LNCS, vol. 6394, pp. 1–15. Springer, Heidelberg (2010)
14. Blunk, A.: Discrete-Event Simulation Modelling Framework with Extensibility (DMX), http://ablunk.github.com/dmx
15. Zingaro, D.: Modern Extensible Languages. McMaster University (2007), http://www.cas.mcmaster.ca/sqrl/papers/SQRLreport47.pdf
16. Bachrach, J., Playford, K.: The Java Syntactic Extender (JSE). In: Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2001), pp. 31–42. ACM (2001)

17. Itemis AG: Xtext, http://www.eclipse.org/Xtext/
18. JetBrains: Meta Programming System (MPS), http://www.jetbrains.com/mps
19. Soden, M., Eichler, H.: An Approach to use Executable Models for Testing. In: Enterprise Modelling and Information Systems Architectures Concepts and Applications (EMISA 2007). Lecture Notes in Informatics, vol. P-117, pp. 75–86. Gesellschaft für Informatik (2007), http://subs.emis.de/LNI/Proceedings/Proceedings119/gi-proc-119-006.pdf
20. Sadilek, D.A., Wachsmuth, G.: Prototyping Visual Interpreters and Debuggers for Domain-Specific Modelling Languages. In: Schieferdecker, I., Hartman, A. (eds.) ECMDA-FA 2008. LNCS, vol. 5095, pp. 63–78. Springer, Heidelberg (2008)
21. Birtwistle, G.: DEMOS – a System for Discrete Event Modelling on Simula. Springer (1987)
22. Blunk, A., Fischer, J.: Prototyping Domain Specific Languages as Extensions of a General Purpose Language. In: Haugen, Ø., Reed, R., Gotzhein, R. (eds.) SAM 2012. LNCS, vol. 7744, pp. 72–87. Springer, Heidelberg (2013)
23. Eclipse Foundation: Eclipse Modeling Framework, http://www.eclipse.org/modeling/emf
24. Ritzberger, F.: RunCC – Java Runtime Compiler Compiler, http://runcc.sourceforge.net
25. Page, B., Kreutzer, W.: The Java Simulation Handbook: Simulating Discrete Event Systems with UML and Java. Shaker Verlag (2005)
26. Acceleo: Transforming Models into Code, http://www.eclipse.org/acceleo
27. Object Management Group: MOF Model To Text Transformation Language (MOFM2T), 1.0 (2008), http://www.omg.org/spec/MOFM2T/1.0/
28. Object Management Group: Documents Associated With Unified Modeling Language (UML), V2.4.1 (2011), http://www.omg.org/spec/UML/2.4.1/
29. International Telecommunication Union: Z.100 series, Specification and Description Language, http://www.itu.int/rec/T-REC-Z.100/en
30. Barr, R.: An efficient, unifying Approach to Simulation using Virtual Machines. Doctoral Dissertation, Cornell University (2004), http://jist.ece.cornell.edu/docs/040517-thesis.pdf
31. Blunk, A., Fischer, J., Sadilek, D.A.: Modelling a Debugger for an Imperative Voice Control Language. In: Reed, R., Bilgic, A., Gotzhein, R. (eds.) SDL 2009. LNCS, vol. 5719, pp. 149–164. Springer, Heidelberg (2009)
32. Wu, H., Gray, J., Mernik, M.: Grammar-driven Generation of Domain-specific Language Debuggers. Software – Practice & Experience 38(10), 1073–1103 (2008)

FTG+PM: An Integrated Framework for Investigating Model Transformation Chains

Levi Lúcio[1], Sadaf Mustafiz[1], Joachim Denil[2,1], Hans Vangheluwe[2,1], and Maris Jukss[1]

[1] School of Computer Science, McGill University, Canada
{levi,sadaf,joachim.denil,hv,mjukss}@cs.mcgill.ca
[2] University of Antwerp, Belgium
{joachim.denil,hans.vangheluwe}@ua.ac.be

Abstract. In this paper, we describe our ongoing work on *model transformation chains*. Model transformation chains refer to the sequences of model transformations in Model Driven Engineering (MDE). The transformations represent and formalise typical model/software engineering activities, and their chaining is the natural composition of such activities. Model transformation chains found in industrial practice vary widely, depending on the specific domain they are used in. By explicitly modelling development activities, these activities can be analysed and the MDE process may be improved. As a step towards such analyses, we propose an integrated framework to describe all the artifacts involved in model transformation chains, as well as the means to execute "enact" those chains. We describe the Formalism Transformation Graph + Process Model (FTG+PM) which is at the heart of our framework in detail.

1 Introduction

Model Driven Engineering (MDE) is currently the mainstream top-down approach to software development. The philosophy behind MDE is that software development should start by building *domain specific* structural and behavioral models of the system under development. By *domain specific* we mean that initially models of the system should be described in a language close to the domain being tackled. During the software development process those models are then improved, augmented and refined by the application of model transformations – possibly with the automatic or manual injection of additional information.

Model transformations have been called the *heart and soul* of MDE [1]. Chaining model transformations is a natural step in MDE as such chains allow describing the composition of activities in software construction and provide explicit means for MDE automation. However, to the best of our knowledge little work is devoted to understanding the underlying structure of such chains when they are used in domain specific software development. This work is crucial for the following (non-exhaustive) list of reasons:

– **Reuse**: Model transformation chains are typically devoted to building software within certain domains. In this paper, we provide an example of the

F. Khendek et al. (Eds.): SDL 2013, LNCS 7916, pp. 182–202, 2013.

usage of model transformation chains for building automotive software. As in traditional software development, the modularity and possibility of reuse of such chains is extremely relevant from an engineering viewpoint. It seems natural that subsets of a transformation chain developed for a given software engineering purpose can be reused without much changes for a similar engineering purpose. Moreover, by identifying and classifying subsets of transformation chains responsible for high level activities in domain specific software development (e.g. requirements development, domain-specific design, verification, simulation, analysis, calibration, deployment, code generation, execution, etc), it is possible to achieve a finer level of understanding and control of such activities – in a domain specific or in a more general context;
- **Traceability**: Traceability is increasingly required in software development at the stakeholder level (e.g. to ensure a given requirement has been implemented in the system), but also at the software development level (e.g. to ensure traceability as high level models are refined along the development process). Because transformation chains explicitly model the relations between the several steps of an MDE process, traceability is a natural consequence of using such chains;
- **Certification**: Finally, and possibly most importantly, by having an explicit representation of such transformation chains and the models (and metamodels) they work on, the certification of such processes becomes possible. In certain domains such as embedded systems, automotive or aerospace, strict norms exist to ensure each step in software production is performed correctly and is properly documented. A large effort has been devoted in the last two decades to developing verification methods for software. The MDE community is now missing studies on how and when those techniques should be applied, but also how they can be composed in a meaningful way. Again, model transformation chains are the ideal context to study the usage and utility of such verification methods for software certification in MDE.

Several studies such as [2,3,4,5,6,7,8], among others, have addressed model transformation chains. However, to perform an investigation on the nature and pragmatic uses of transformation chains we require an environment where all the artifacts involved in such chains are explicitly formalized, easily accessible and easily manipulated. The majority of the approaches in the literature dealing with transformation chains are concerned with automated execution. The explicit and integrated representation of all artifacts involved in model transformation chains in a way that makes them amenable to the formal study of those chains' characteristics is typically less of a concern. In order to address this issue and to have a solid basis to study the issues mentioned above, we need a framework allowing the modelling of model transformation chains that addresses the following requirements:

1. An explicit representation of both the languages used in the model transformation chains and the relations between those languages should be provided;
2. An explicit representation of the individual model transformations should be available and the means to execute those transformations should exist;

3. Explicit process modelling of MDE activities should be possible such that transformation chains can be built;
4. Automatic execution of transformation chains should be possible. In order to study the execution of transformation chains and which parts of those chains should be performed manually, we require that a model transformation chain execution engine exists.

In order to address these requirements, we propose in this paper the FTG+PM framework. The proposed framework is completely supported by our tool AToMPM, A Tool for Multi-Paradigm Modelling [9], which allows explicit modelling of and access to, all used artifacts.

This paper is organised as follows: Sect. 2 provides background information on meta-modelling, model transformation, and our tooling environment. In Sect. 3 we introduce our running example, the *power window case study.* Section 4 introduces the FTG+PM framework. Section 5 presents the the explicit execution semantics of the FTG+PM. Section 6 describes in detail an automotive power window case study and by doing so illustrates the artifacts involved in a model transformation chain. Section 7 discusses related work. Finally, Sect. 8 draws some conclusions on how the FTG+PM addresses the aforementioned requirements and proposes future studies on model transformation chains.

2 Background

Within the context of this paper we have chosen to follow the terminology as presented in [10]. A *model* is completely described by its abstract syntax (its structure), concrete syntax (its visualisation) and semantics (its unique and precise meaning). A *language* (also called *formalism*) is a possibly infinite set of (abstract syntax) models. This set can be concisely described by means of a grammar or a metamodel. No semantics or concrete syntax is given to these models. Several such languages, called *metamodels*, are used to describe families of models of computational artifacts that share the same abstraction concerns. Each *metamodel* is a language that may have many *model* instantiations.

Domain Specific Modelling (DSM) captures the fact that certain languages or classes of languages, called Domain Specific Languages (DSLs) are appropriate for expressing models in certain domains.

Model transformations involve the mapping of source models in one or more formalisms to target models in one or more formalisms using a set of transformation rules.

In this work, we use rule-based graph transformation as the means for model transformation [11]. This requires (meta-)models to be stored as graphs, thus allowing model manipulations to be defined as graph grammars.

In our work, we have used *AToMPM* [9]*, A Tool for Multi-Paradigm Modelling,* to build metamodels, transformations, and execution support for the FTG. AToMPM (the successor of AToM^3[12]) rigorously applies the "model and conforming meta-model" workflow to all facets of domain specific modelling. It allows

modelling of language syntax (abstract and concrete) and semantics. The tool supports rule-based graph transformations and pre- and post-condition pattern languages to allow specification of model transformations. AToMPM runs on a web browser and provides support for real-time, distributed collaboration.

3 The Power Window Case Study

In order to explain how our FTG+PM framework addresses the requirements stated in Sect. 1 we will use a running example. In Fig. 1 we show a slice of the FTG+PM we have built for developing the power window control software. The power window FTG+PM was built based on our experiences with developing automotive software. Further details on this case study can be found in [13,14].

A power window is basically an electrically powered window. The basic controls of a power window include lifting and descending the window, but an increasing set of functionalities is being added to improve the comfort and security of the vehicle's passengers. When given the task to build the control system for a power window, a software engineer considers several variables, such as:

(1) the physical power window itself, which is composed of the glass window, the mechanical lift, the electrical engine and some sensors for detecting for example window position or window collision events;
(2) the environment with which the system (controller plus power window) interacts, which will include both human actors as well as other subsystems of the vehicle – e.g. the central locking system or the ignition system.

This idea is along the same lines as that presented by Mosterman and Vangheluwe in [15]. According to control theory [16], the control software system acts as the *controller*, the physical power window with all its mechanical and electrical components as the *process* (also called the *plant*), and the human actors and other vehicle subsystems as the *environment*.

The FTG+PM slice in Fig. 1 presents the design and verification part of developing the power window software. The case study begins with three domain-specific languages built for the modelling of power windows (*PlantDSL*, *EnvDSL* and *ControlDSL* in the FTG part of Fig. 1, allowing respectively modeling the *plant*, *environment* and *controller* for a power window), plus a network language (not shown in Fig. 1) that allows the connection of the components defined in those DSLs. Those domain specific components are separately transformed into modular Petri nets (*EncapsulatedPetriNet* in the FTG part of Fig. 1). When all the modular Petri nets have been built, they are composed into a single Petri net (*PetriNet* in the FTG part of Fig. 1). This Petri net can then be used to verify that the system cannot enter a non-safe state. While the left side of Fig. 1 presents the FTG part of the model detailing the required formalisms and transformations, the right side of Fig. 1 shows how executions of those transformations are chained. Note also that in Fig. 1 dotted elements *PlantToPN*, *EnvToPN* and *ControlToPN* denote automatic transformations, while other elements without dots denote manual ones. The following section elaborates on the syntax of the FTG+PM language.

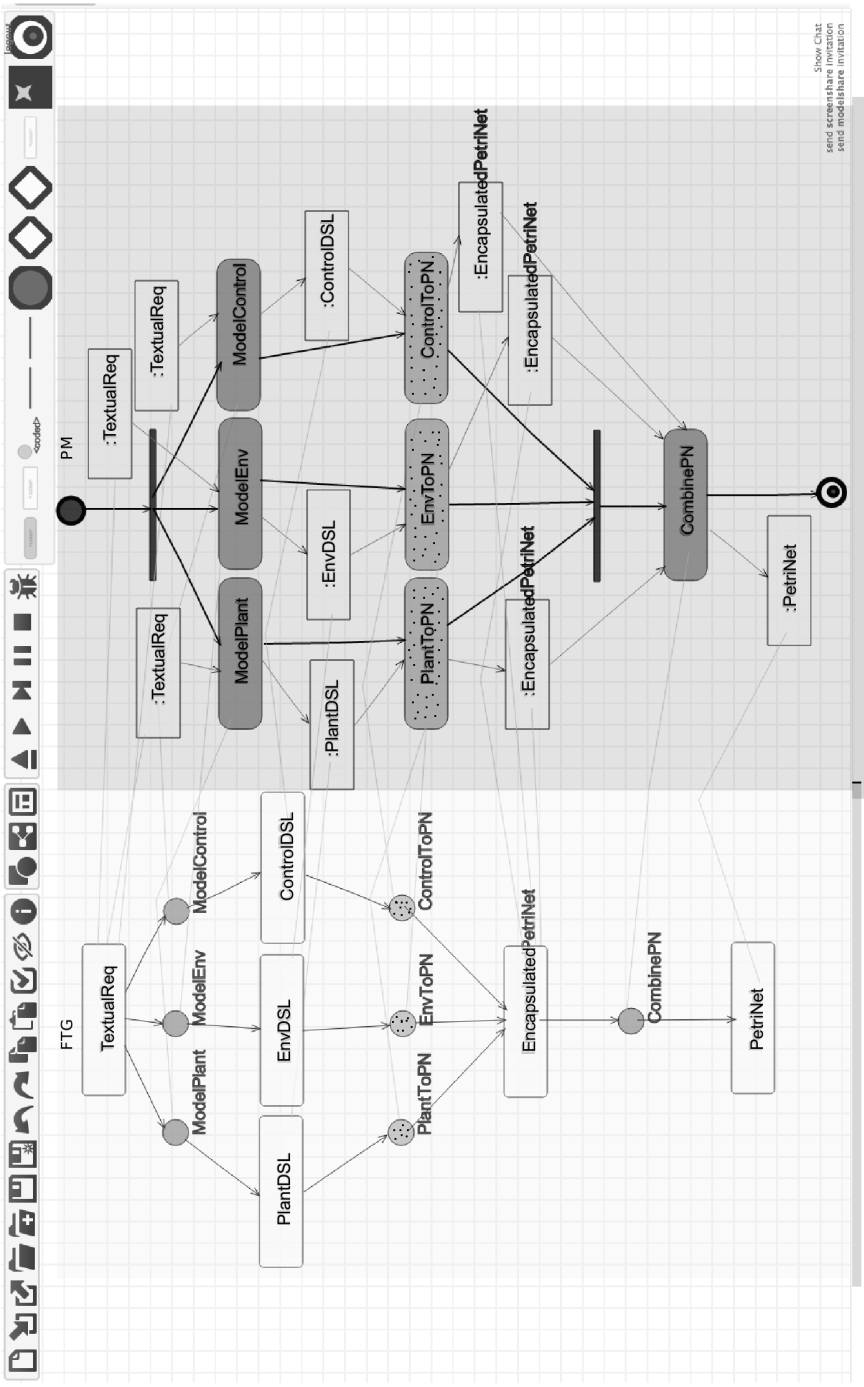


Fig. 1. Example FTG+PM Model

4 The FTG+PM Language

The FTG+PM language is defined using two sub-languages: the Formalism Transformation Graph (FTG) language and a Process Model (PM) language. We give a brief overview of FTG+PM in this section. The formalization of the language along with further details on the framework can be found in [17]. A unified metamodel of the FTG+PM language is shown in Fig 2.

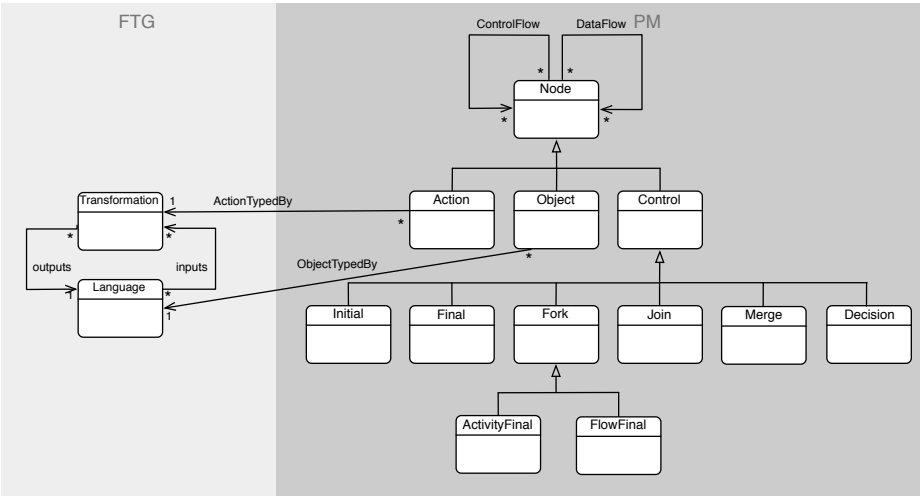


Fig. 2. Formalism Transformation Graph and Process Model (FTG+PM) Metamodel

The *Formalism Transformation Graph (FTG)* is a hypergraph with *languages* as nodes and *transformations* as edges. It lays down the relationships among the multitude of languages and transformations used for the development of a particular system or systems within a domain. The framework takes into account the heterogeneous nature of the MDE process, and integrates the MDE paradigms: multi-abstraction, multi-formalism, and metamodelling. The languages at each level in the FTG are used to represent and model knowledge at different levels of abstraction starting from requirements to code synthesis. Depending on the activity involved, we build our FTG by choosing the most appropriate formalism based on the nature of the problem and the intention: discrete-event formalisms, continuous time formalisms, hybrid formalisms, or others. All the languages in the FTG are metamodelled, and the transformations are specified using rule-based graph grammars. Languages in the FTG are denoted by labelled rectangles, and transformations are denoted by labelled circles on edges. The incoming edges show the source languages of the transformation, and the outgoing edges point to the target languages. Fig 1 (discussed in detail in Sect. 6) shows a slice

of a FTG+PM model for the automotive domain (complete model presented in [14]), and describes a part of the artifacts and the process necessary to build software to control power windows of automobiles. The FTG model may include self loops to languages (for example, when a transformation is endogenous in nature).

The *Process Model (PM)* (see sample PM in Fig 1 highlighted in gray) is used in conjunction with the FTG to model the MDE process. Having a process model integrated with the FTG allows us to precisely and in detail model the MDE process we follow, and to provide execution support for it when needed. The PM exhaustively describes the control flow and data flow in the MDE process. Our process model is a subset of the UML 2.0 activity diagram metamodel. In the PM language, the labelled roundtangles (actions) in the Activity Diagram correspond to executions of the transformations declared within the FTG. This typing relation is made explicit in the FTG+PM model by the thin horizontal links connecting the action nodes in the PM to the transformation elements in the FTG. Labelled rectangles (object nodes) in the PM correspond to models that are consumed or produced by actions. A model is an instance of a language declared in the FTG part of the model with the same label. This typing relation is again made explicit by horizontal links connecting the object nodes to the language elements in the FTG. Notice that in a PM model thin edges denote data flow, while thick edges denote control flow. Notice also that for each model input and output edge of a PM action a corresponding edge exists for the transformation typing it on the FTG side. The input and output models of an action are typed according to the input and output languages of the FTG transformation that types that action. Finally, the join and fork Activity Diagram flow constructs represented as horizontal bars, allow us to represent concurrent activities.

The FTG defines a set of transformations and the PM describes the chaining of the transformations and the execution order for a particular intent. The FTG+PM can thus be considered to be a model transformation chaining language for describing the composition of transformations by defining their order of execution, source and target model types, and the relationships and dependencies among them.

Various business process modelling or workflow languages exist in the literature. Our intention is to model the MDE process as a chain of model transformations rather than a business process with models as first class artifacts and with model transformations as the core of the approach, hence we have chosen to use UML 2.0 activity diagrams for our purpose. In addition, UML 2.0 is a standard in the MDE community, and our tool, AToMPM (A Tool for Multi Paradigm Modelling) [9] also provides support for UML. Our framework is supported by AToMPM for creating metamodels, describing graph transformations, and for building execution support for the FTG.

5 FTG+PM Semantics: Transformation and Tool Support

The proposed FTG+PM language is implemented in our AToMPM tool. AToMPM contains its own transformation language. Transformations and transformation rules, in AToMPM, are treated as normal models conforming to an appropriate meta-model. Transformation rules, consisting of a left hand side (LHS), a right hand side (RHS) and a set of negative application conditions (NAC), are tried in an order given by a rule scheduling model, in this case described in a finite state automaton-like formalism. Since transformation rules and their scheduling are explicitly modelled within AToMPM using appropriate meta-models, defining higher-order transformations is straightforward.

To execute a FTG+PM model, we transform the PM to the native transformation scheduling language of AToMPM. The result of the transformation of the power window FTG+PM shown in Fig. 1 to the native AToMPM transformation language is depicted in Fig. 3. A PM action which is mapped to a transformation can be either automatic (e.g. see dotted elements *PlantToPN* etc. in Fig. 1) or manual (other elements in Fig. 1 without dots). Manual transformations are not implemented using graph transformations, but involve actions in which the output models need to be created by the user(s).

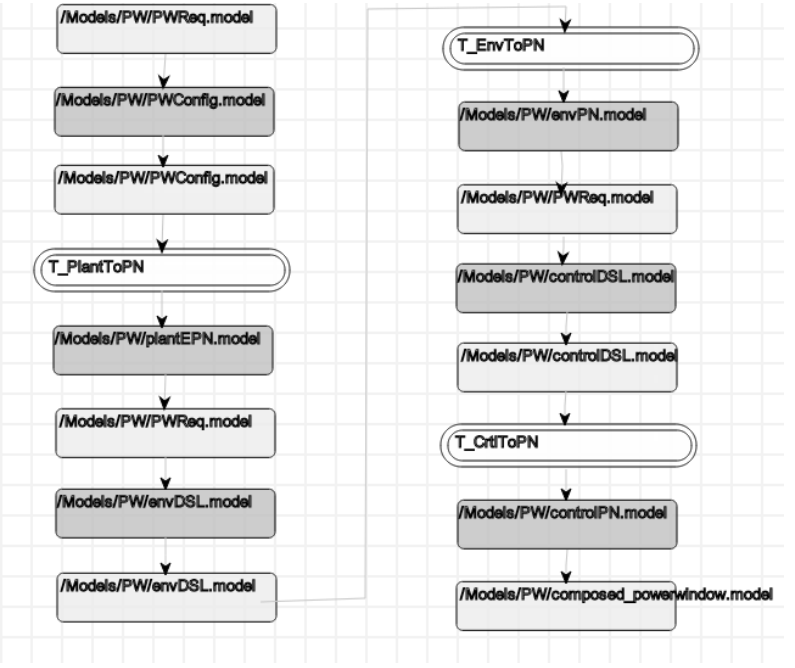


Fig. 3. The resulting transformation model of the example FTG+PM

The transformation schedule is created as follows:

(1) a PM *Action* node tagged as *automatic* corresponds to the execution of a transformation defined in the FTG *Transformation* node typing it;
(2) transformations are scheduled according to the control-flow defined in the PM.

An example rule of this transformation from FTG+PM into AToMPM's transformation scheduling language is shown in Fig. 4. Note that the LHS of a rule matches a pattern in the input model including a PM *Action* (round-tangle) typed by a FTG *Transformation* (circle), while the RHS rewrites it by building the scheduling of the transformation execution as a double round-tangle (a composite transformation application in AToMPM's rule scheduling language). The double round-tangle is then used to execute this transformation within the AToMPM environment. For example, the *PlantToPN* action in Fig. 1 is mapped to the transformation *T_PlantToPN* (inside the double round-tangle node) in Fig. 3.

The scheduling language additionally includes rectangular nodes corresponding to the execution of a single transformation step to handle opening of input models (shaded as */Models/PW/PWReq.model* in Fig. 3) or writing (includes editing and saving) of output models (shaded as */Models/PW/PWConfig.model* in Fig. 3), and control flow arrows to impose the ordering of the scheduling of the transformations.

When executing a FTG+PM model, the input of a scheduled transformation depends on whether there are incoming dataflow arrows:

(a) if there are incoming dataflow arrows into the action node, for each of these dataflow arrows a transformation step is created that opens the specified input model in the appropriate formalism in the current canvas. The transformation rules that open the specified models are scheduled before the execution of the transformation defined by the action node;
(b) If there is no incoming data flow arrow, the result of the previous transformation (present on AToMPM's modelling canvas) is used as the input.

A similar solution is used for the output of an action:

(a) when a dataflow arrow emanates from an action node, a transformation step is created to save the target model (specified in the location by the object node) and clears the modelling canvas. The transformation step is scheduled after the transformation defined by the action node;
(b) If no dataflow arrow exits the node, the canvas is not cleared.

Manual transformations (such as *ModelPlant* highlighted in dark gray in Fig 1) are not mapped to a transformation, i.e. a double round-tangle node, in the resulting schedule. They are mapped to transformation steps that first open the input models and then to transformation steps that write/save the output models. One transformation step corresponds to one open/save model and one or more associated formalisms. For instance, the *ModelPlant* action is mapped to a pair of transformation steps in Fig. 3:

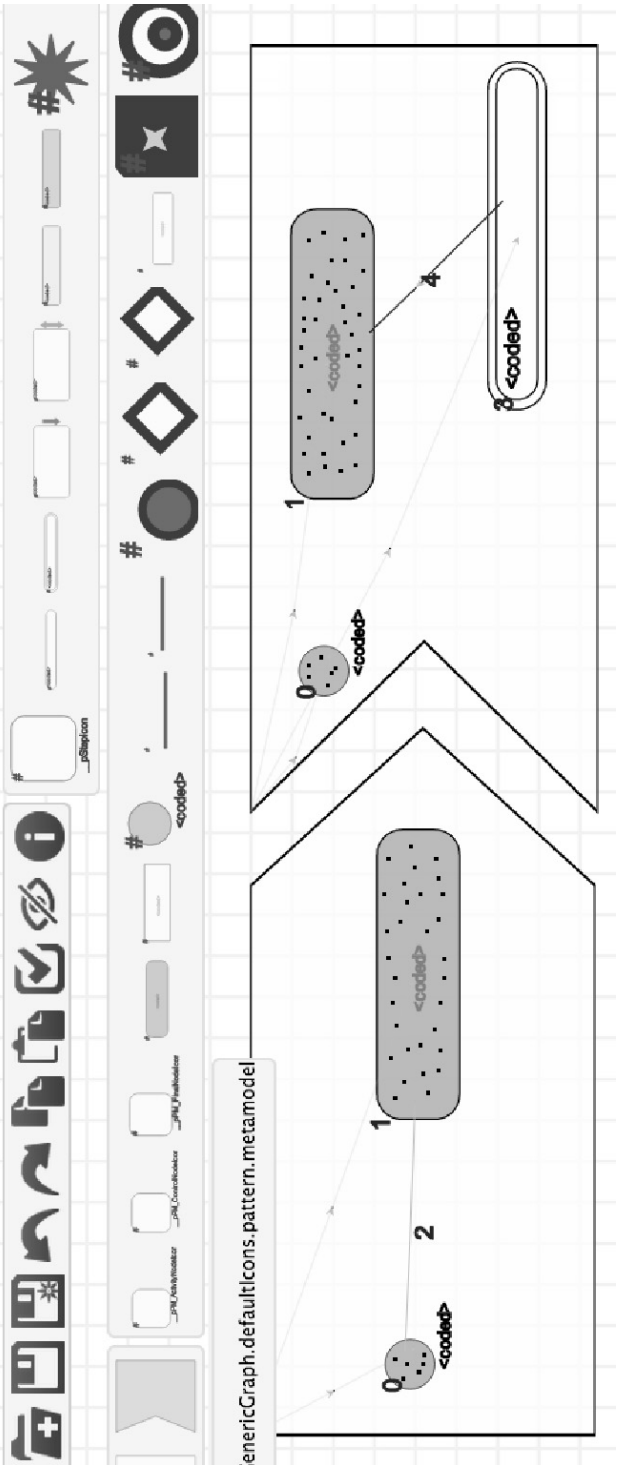


Fig. 4. Transformation Rule to Map an Action Node to a Transformation

(1) a step that opens the input requirements model, `/Models/PW/PWReq.model`, which is a model instance of *TextualReq*;
(2) a step that writes `/Models/PW/PWConfig.model`, the output plant configuration model, which is an instance of *PlantDSL*.

In case of multiple input and output models, a transformation step is created in the schedule corresponding to each open and write step. When output model(s) are produced by manual transformations, a new AToMPM window is spawned for each output model, which loads the model if it already exists (to allow for further editing) or opens an empty canvas with the formalism toolbars loaded otherwise. Once the user is done creating or modifying the model, a button needs to be pressed to save the model and to return to the parent AToMPM window where automatic transformation resumes.

In the current implementation, there is no support for the (semi-)parallel execution of fork and join nodes since the current transformation language in AToMPM does not allow this. Instead, the transformation towards the AToMPM transformation language makes sequential the different branches between the joins and the forks. This is done in the same way as described in [18] where a marker is made at the top of the fork. Another marker is used to follow the chain until the join node is found. Afterwards the full branch is scheduled before the join node. This is done until all branches are made sequential.

When nesting occurs, the inner fork/join pairs are made sequential first.

Since the canvas can be used as the input for the next action node, the state of the canvas has to be saved before the fork node. This is done by inserting an object node, connected to the action node before the fork node. The output goes to the first actions of each of the branches after the fork. At the last action of each branch, a similar object node is inserted that is connected to the first action after join node.

Because all models are saved and closed after the action node has executed and reloaded before starting a new action, the sequential process model preserves the original semantics.

6 Languages and Transformations in the Power Window Case Study

In this section, we present some of the languages and transformations that allow building and executing the transformation chains in the power window FTG+PM. Note that all the metamodels, models and transformations we present in Sections 6.1 and 6.2 have been built and are readable and/or executable using the AToMPM modelling environment. Note also that in the sections that follow the metamodels, models and transformations are not explained in complete detail, as the goal of their presentation in this paper is to illustrate the usage of the FTG+PM, rather than the case study itself. Again, for further details on the models presented in the sections that follow we refer the reader to [13].

6.1 Building the Domain Specific Languages

The design and verification part of the power window FTG+PM in Fig. 1 makes use of several domain specific languages (DSLs) for defining *controller*, the *plant* and the *environment* models.

Due to space reasons, we only present in this text the *plant* DSL which allows the specification of the hardware necessary for a given power window configuration.

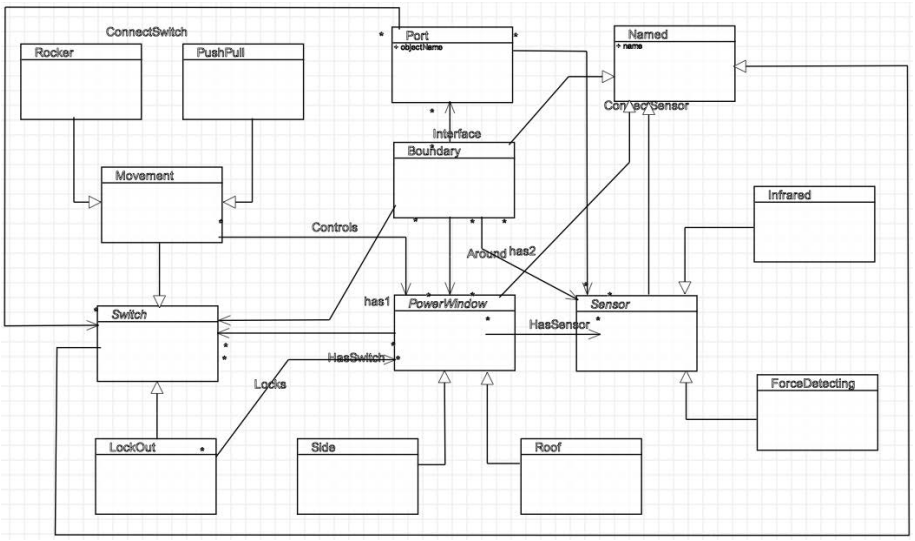


Fig. 5. Plant DSL Metamodel

In Fig. 5, the metamodel of the *plant* DSL can be observed. The main class of the language is the *PowerWindow* class, which is abstract and can be instantiated as a *Side* window or a *Roof* window. A physical power window includes a set of switches of two kinds: *Lockout* switches allow removing control from other power windows in the car (as specified by the *controls* association); *Rocker* or *PushPull* switches allow controlling window movement. Finally, a power window may also have sensors of types *Infrared* or *ForceDetecting* for detecting if an object is blocking the window from going up.

In Fig. 6, we present a model instance of the Plant DSL, where a configuration of two power windows of an automobile is described. The model includes a *driver* and a *passenger* power window, where the driver's window has three buttons: a pushpull button for controlling the driver's window, a pushpull button for controlling the passenger's window, and a lockout switch for disabling/enabling the control of the passenger's window. The passenger's window includes a rocker button and a infrared sensor meaning the window automatically stops rolling up when an object obstructs its path.

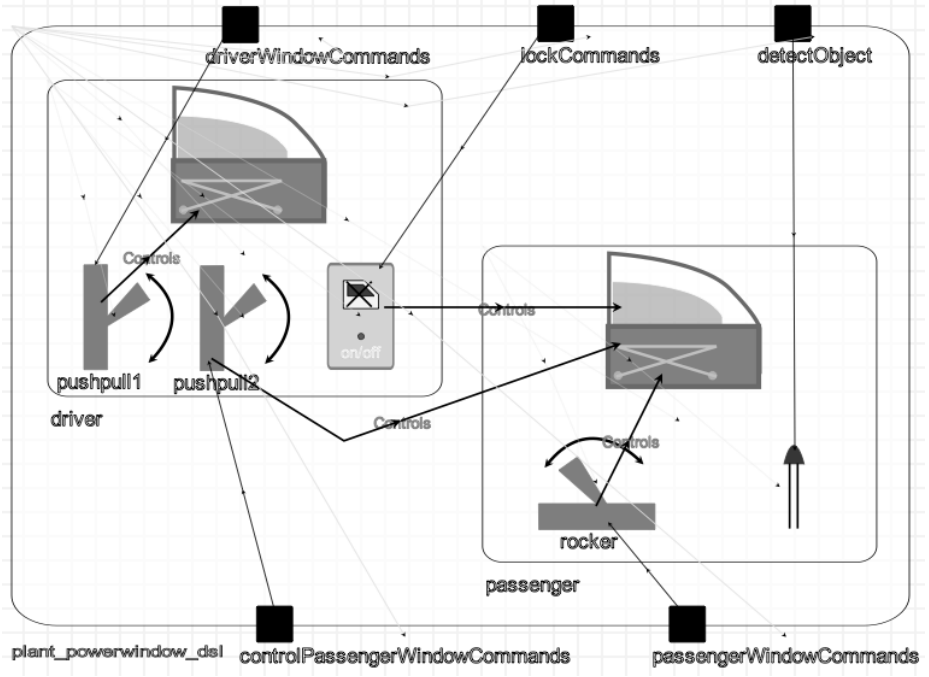


Fig. 6. Plant DSL: Example Model

6.2 Transformations

From Domain Specific Models to Modular Petri Nets. Two types of modular Petri nets are generated from the Plant DSL model by the *PlantToPN* transformation in Fig. 1, depending on the power window configuration. In Fig. 7, the Petri net modelling the discrete behavior of a power window with an obstacle detecting sensor can be observed. During operation the window can either be at the *bottom* of the frame (*bot* place, meaning the window is completely open), somewhere in the *middle* of the frame (*mid* place, meaning the window is partially open), or at the top of the frame (*top* place, meaning the window is closed). Additional places in Fig. 7 (*midDetObj*, *topDetFrame* and *danger*) are used to model object detection during window operation. The modular Petri net in Fig. 7 also includes ports (having as concrete syntax black squares) for synchronisation with other modular Petri nets. An example rule of the *PlantToPN* transformation in Fig. 1 is shown in Fig. 8. This particular rule builds the behavior of a power window without obstacle detection. Notice that the negative application condition of the rule (inside the dashed square) prevents the power window that is matched by the LHS of the rule from having a sensor.

Due to space constraints, we are unable to present here the similar transformations into modular Petri nets defined for both the control and environment models (called *EnvToPN* and *ControlToPN* in Fig. 1).

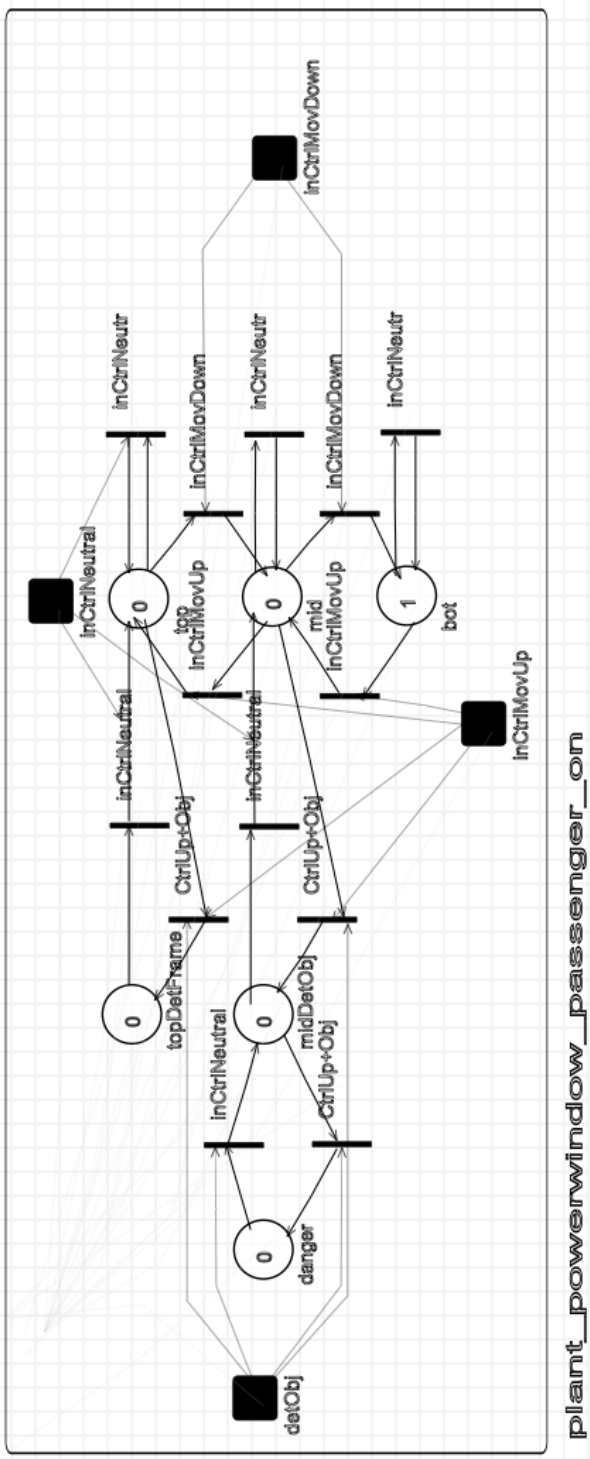


Fig. 7. Transformed Passenger Window Plant Model

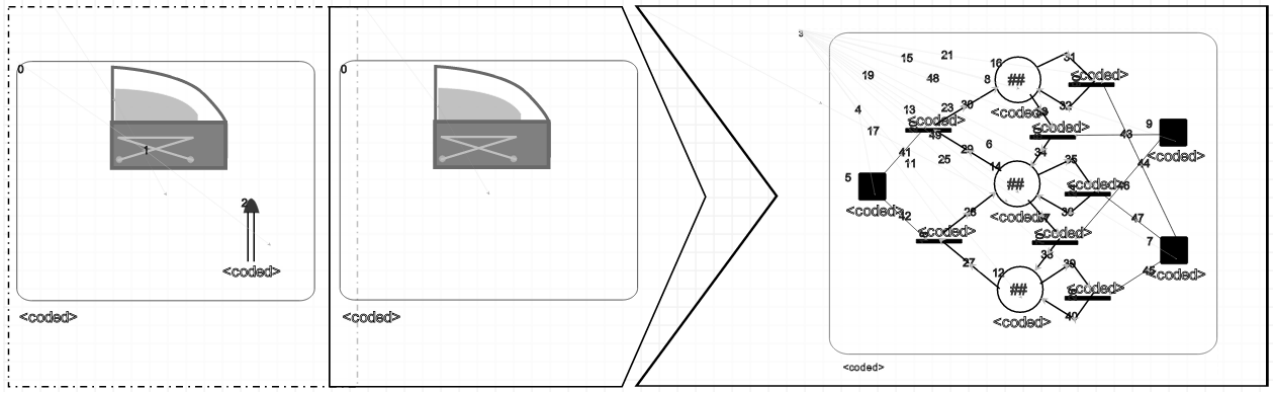


Fig. 8. Transformation Rule for Building a Petri Net Representation of a Power Window without Obstacle Sensor Plant

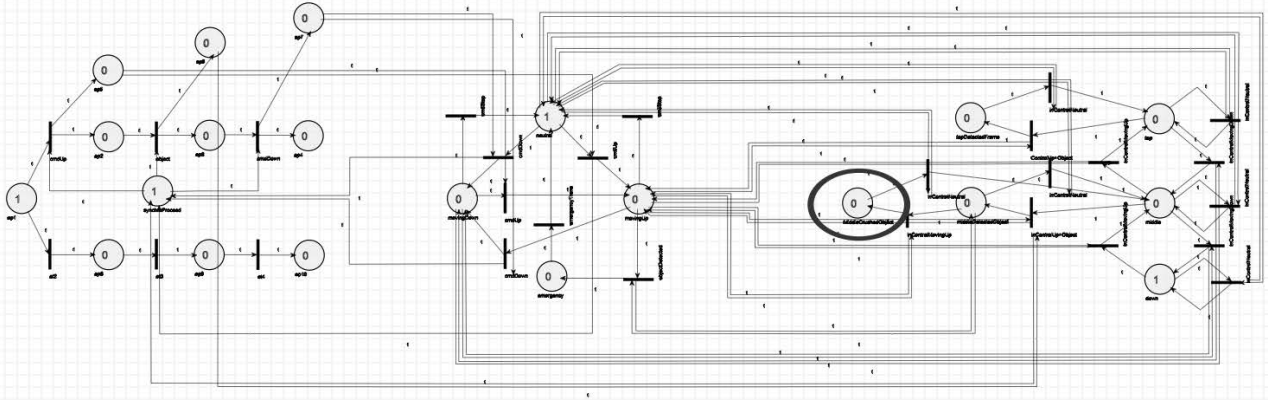


Fig. 9. Composed Petri Net Model for the Power Window Control Software

Composition of the Modular Petri Nets. Once the environment, plant, and control models are transformed to the modular Petri nets, it is necessary to compose those models. This last transformation, called *CombinePN* in Fig. 1, allows to manually[1] build the complete Petri net of the power window example using the produced modular Petri nets and an additional network model (not shown here). This composed Petri net is an instance of the PetriNet formalism in the FTG part of Fig. 1. An example of such a model (produced from our example models in Fig. 6 and Fig. 7) can be (partially) observed in Fig. 9. This composed Petri net is used for the validation of the safety requirements of the power window. In particular we have used it to automatically check that a state where an object obstructing the window has been detected and the window is still going up is never reached. In this state a token exists in the "danger" place in the EncapsulatedPetriNet model in Fig. 7. This place can be found in the rightmost subnet of the composed model in Fig. 9, highlighted by a red ellipse. Note additionally that in the transformation chain found in the complete power window FTG+PM defined in [14], the Petri net verification step is itself built as a transformation.

7 Related Work

We consider different approaches for the composition of model transformation chains. We have looked at work which have applied mega-modelling concepts and/or process modelling concepts in their approach. A *megamodel* is a conceptual framework used to reason about MDE and represents the global view of the considered artifacts (models, metamodels, and other global entities) in a system and the relationships between them [19,20]. Key in their approach is that not only models, but also tools and the services and operations they provide are also represented as models, with all sorts of relations in between.

The approaches are compared on a number of properties. The first criteria is whether the approach uses mega-modelling and therefore has an explicit representation of the modelling languages and relations between the languages by means of transformation definitions. The second is whether the approach allows the composition of chains by means of an explicit representation of the process. Finally, we consider both automatic transformations where the execution of the transformation is completely automated and manual transformations where a modelling environment is setup in the defined language(s). Table 1 shows the comparison of the different approaches.

Most approaches allow for the data-flow composition of model transformations where input and output relations of the transformations are used to chain different transformations. The control-flow of these approaches is inferred from this data-flow composition. Oldevik proposes a framework for the data-flow composition of transformations in [2]. It uses UML activities like our FTG+PM to model these relations, though control flow is not taken into account. A definition

[1] We are currently building the transformation to automatically execute this composition.

Table 1. Comparison of the approaches (supports (✓), does not support (x) , unknown/unclear (∼))

Tool	Explicit Megamodel	Explicit Process Model		Transformations	
		Control Flow	Data Flow	Automatic	Manual
Oldevik et al. [2]	✓	x	✓	✓	∼
Vanhooff et al. [3]	✓	x	✓	✓	x
UniTI [4]	✓	x	✓	✓	x
TraCo [5]	✓	x	✓	✓	x
Wagelaar [6]	x	x	✓	✓	x
MoTCoF [7]	∼	x	✓	✓	x
Wires* [21]	x	x	✓	✓	x
transML [22]	✓	∼	✓	✓	x
Epsilon [23]	∼	x	✓	✓	∼
MCC [8]	x	x	x	✓	x
Aldazabal et al. [24]	x	✓	✓	✓	∼
Diaw et al. [25]	✓	x	✓	✓	∼
FTG+PM	✓	✓	✓	✓	✓

for manual transformations is present, though it is not described how the framework copes with these transformation types. In [3], a data-flow composition of transformation framework is presented similar to the UniTI framework [4]. The concepts of these frameworks are extended by the TraCo framework [5] where additional validation checks are performed on the composition of the transformations. Wagelaar [6] presents a DSL for the composition of transformations. The models are transformed to ANT scripts for execution. Seibel et al. present the MoTCoF framework [7] for the data-flow and context composition of model transformations. The meta-model of the approach is not shown, but most likely an explicit megamodel is present. Wires* [21] provides a graphical language for the orchestration of ATL model transformations. It has modelling elements for complex data-flow for example decision nodes, parallel execution and support for loops. It does not however take manual activities into account. The transML framework [22] is created for transformations in the 'large'. It provides meta-models for requirements, analysis, architecture and testing of transformations. The tool supports data-flow chaining of transformations by transforming to ANT-tasks. The Epsilon Framework, presented in [23], provides a model management framework where ANT-tasks can be used to build chains of transformations. It is not clear if the Generic Model Manipulation Task can be used for the loading of a modelling environment though models can be loaded and stored using ANT tasks. Finally, Kleppe proposes a scripting language MDA Control Center (MCC) [8] for combining multiple transformations in sequence and in parallel.

In the process modelling community, frameworks for MDE are proposed as well, though these usually do not focus on transformation chaining, for example [26,27]. Two examples however do take transformation chaining into account.

In [24], Aldazabal et al. present a framework for tool integration where transformations can be chained. The process is modelled in SPEM or BPMN (Business Process Modelling Notation) and is transformed to BPEL (Business Process Execution Language) for execution support. They do not however have a megamodel to validate input-output relations. In [25], Diaw et al. present an adaptation of SPEM [28] for the use in an MDE context. The composition is a data-flow composition like most transformation chaining approaches discussed above. Both frameworks allow the modelling of manual activities, though it is not clear how the frameworks handle these manual activities.

Our approach, combines the explicit modelling of the languages and transformations (megamodel) together with a process model that supports complex control-flow constructs. This allows the modelling of non-linear transformation chains for building complex applications. Transformations can either be executed automatically or require manual intervention. In the manual case the framework opens a modelling environment for the activity and continues the process when the activity is finished. The explicit modelling of all the components allows to reason about these complex chains of transformations.

8 Conclusion and Future Work

In this paper, we have presented a framework for explicitly describing model transformation chains within MDE. We have introduced the FTG+PM language, composed of the Formalism Transformation Graph (FTG) and its complement, the Process Model (PM). The building blocks of the FTG are formalisms (nodes in the graph) and transformations (edges in the graph). The FTG describes the different languages that can be used at each stage of model development. The transformations model development activities, and the control flow and data flow between each transformation action are explicitly modelled in the PM.

In its current form, the FTG+PM framework satisfies the requirements stated in Sect. 1. We have explicitly described the abstract and concrete syntax of the FTG+PM language by metamodelling them in our tool, AToMPM. In addition, the syntax of each of the languages appearing as a node in the FTG is also explicitly modelled. The transformations defined as activities in the PM are all modelled as rule-based graph transformations using AToMPM's transformation language (which was itself modelled explicitly). The FTG+PM language allows transformations to be defined as *automatic* or *manual*. Our framework allows user interventions in the MDE process, and provides means for creating artifacts using manual activities. The process model connects the transformations using control flow and data flow links. UML 2.0 activity diagrams were chosen as the language to describe the PM. This allows us to model the chaining of transformations as a process model and to build execution support for it. For execution, we map the process model to the native transformation scheduling language of AToMPM. The mapping takes into consideration whether a transformation is automatic or manual. In case of manual activities, the users can complete the task at hand and resume the execution of the process model which continues with the execution of the next scheduled transformation. The FTG+PM

approach was applied to a concrete problem in the automotive domain: the power window case study.

As mentioned in Sect. 1, the goal of having a framework that allows us to thoroughly describe and automate model transformation chains is to give use the means to study and optimize such chains. As such we are currently developing the following:

- We currently use the power window case FTG+PM to study the notion of *intent* in model transformations. In our work in [29], the *intent* of a model transformation is defined as "a description of the goal behind the model transformation and the reason for using it". The FTG+PM model of the power window model transformation chain helped us to construct a transformation intent language. We are currently building a catalogue of model transformation intents (akin to design patterns in the OO world) and are formalising the properties of such intents. As mentioned in [30], the study of the formal properties of model transformations is in its infancy;
- As a result of our *transformation intent* work, we are now attaching *intent*-related annotations to the transformations described in the PM part of an FTG+PM model. Such annotations may serve to identify formal properties that should be proved for a model transformation. As transformation chaining is a form of relational composition, the formal composition of the properties of individual transformations in the chain is of great importance;
- Using the concrete power window case, we are also investigating the multi-paradigm modelling aspects of the FTG+PM [14]. We expect the study to help in identifying methodological and reusability concerns when developing model transformation chains for the automotive domain, that can hopefully be extrapolated to other domains.

Acknowledgements. Part of this work has been developed in the context of the NECSIS project, funded by the Automotive Partnership Canada.

References

1. Sendall, S., Kozaczynski, W.: Model Transformation – The Heart and Soul of Model-Driven Software Development. IEEE Software 20(5), 42–45 (2003)
2. Oldevik, J.: Transformation Composition Modelling Framework. In: Kutvonen, L., Alonistioti, N. (eds.) DAIS 2005. LNCS, vol. 3543, pp. 108–114. Springer, Heidelberg (2005)
3. Vanhooff, B., Van Baelen, S., Hovsepyan, A., Joosen, W., Berbers, Y.: Towards a transformation chain modeling language. In: Vassiliadis, S., Wong, S., Hämäläinen, T.D. (eds.) SAMOS 2006. LNCS, vol. 4017, pp. 39–48. Springer, Heidelberg (2006)
4. Vanhooff, B., Ayed, D., Van Baelen, S., Joosen, W., Berbers, Y.: UniTI – A Unified Transformation Infrastructure. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 31–45. Springer, Heidelberg (2007)
5. Heidenreich, F., Kopcsek, J., Aßmann, U.: Safe composition of transformations. Journal of Object Technology 10(7), 1–20 (2011), http://dx.doi.org/10.5381/jot.2011.10.1.a7

6. Wagelaar, D.: Blackbox composition of model transformations using domain-specific modelling languages. In: Proceedings of the First European Workshop on Composition of Model Transformations, pp. 15–20 (2006), http://doc.utwente.nl/66171/1/00000179.pdf
7. Seibel, A., Hebig, R., Neumann, S., Giese, H.: A Dedicated Language for Context Composition and Execution of True Black-Box Model Transformations. In: Sloane, A., Aßmann, U. (eds.) SLE 2011. LNCS, vol. 6940, pp. 19–39. Springer, Heidelberg (2012)
8. Kleppe, A.: MCC – A Model Transformation Environment. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 173–187. Springer, Heidelberg (2006)
9. Mannadiar, R.: A Multi-Paradigm Modelling Approach to the Foundations of Domain-Specific Modelling. PhD thesis, McGill University (2012), http://msdl.cs.mcgill.ca/people/raphael/files/thesis.pdf
10. Giese, H., Levendovszky, T., Vangheluwe, H.: Summary of the Workshop on Multi-Paradigm Modeling: Concepts and Tools. In: Kühne, T. (ed.) MoDELS 2006. LNCS, vol. 4364, pp. 252–262. Springer, Heidelberg (2007)
11. Dörr, H.: Efficient Graph Rewriting and Its Implementation. LNCS, vol. 922. Springer, Heidelberg (1995)
12. de Lara, J., Vangheluwe, H.: AToM^3: A Tool for Multi-formalism and Meta-Modelling. In: Kutsche, R.-D., Weber, H. (eds.) FASE 2002. LNCS, vol. 2306, pp. 174–188. Springer, Heidelberg (2002)
13. Lúcio, L., Joachim, D., Vangheluwe, H.: An Overview of Model Transformations for a Simple Automotive Power Window. McGill University, Technical Report SOCS-TR-2012.2 (2012), http://msdl.cs.mcgill.ca/people/levi/30_publications/files/tech_report_mcgill_SOCS-TR-2012.2.pdf
14. Mustafiz, S., et al.: The FTG+PM framework for Multi-Paradigm modelling – An automotive case study. Paper for 6th International Workshop on Multi-Paradigm Modeling (2012), http://avalon.aut.bme.hu/mpm12/papers/paper17.pdf
15. Mosterman, P., Vangheluwe, H.: Computer Automated Multi-Paradigm Modeling – An Introduction. Simulation 80(9), 433–450 (2004)
16. Dorf, R.C.: Modern Control Systems, 12th edn. Pearson (2010)
17. Lúcio, L., et al.: The formalism transformation graph as a guide to model driven engineering. McGill University, Technical Report SOCS-TR-2012.1 (2012), http://msdl.cs.mcgill.ca/people/levi/30_publications/files/tech_report_mcgill_SOCS-TR-2012.1.pdf
18. Bottoni, P., Saporito, A.: Resource-based enactment and adaptation of workflows from activity diagrams. Electronic Communications of the EASST 18 (2009), http://journal.ub.tu-berlin.de/eceasst/article/view/233
19. Favre, J.-M.: Foundations of Model (Driven) (Reverse) Engineering: Models - Episode I: Stories of The Fidus Papyrus and of The Solarus. In: Language Engineering for Model-Driven Software Development, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany (2004), http://drops.dagstuhl.de/opus/volltexte/2005/13
20. Favre, J.-M.: Foundations of Model (Driven) (Reverse) Foundations of Meta-Pyramids: Languages vs. Metamodels - Episode II: Story of Thotus the Baboon. In: Language Engineering for Model-Driven Software Development, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany (2005), http://drops.dagstuhl.de/opus/volltexte/2005/21

21. Rivera, J., et al.: Orchestrating ATL model transformations. Model Transformation with ATL, 34–46 (2009), http://docatlanmod.emn.fr/MtATL2009Presentations/PreliminaryProceedings.pdf
22. Guerra, E., de Lara, J., Kolovos, D.S., Paige, R.F., dos Santos, O.M.: *trans*ML: A Family of Languages to Model Model Transformations. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part I. LNCS, vol. 6394, pp. 106–120. Springer, Heidelberg (2010)
23. Paige, R., et al.: The Design of a Conceptual Framework and Technical Infrastructure for Model Management Language Engineering. In: Proceedings of the 2009 14th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2009), pp. 162–171. IEEE Computer Society (2009)
24. Aldazabal, A., et al.: Automated model driven development processes. In: ECMDA Workshop on Model Driven Tool and Process Integration 2008, pp. 43–54. Fraunhofer IRB Verlag (2008), http://www.modelbus.org/modelbus/images/stories/docs/7AutomatedDevelopment.pdf
25. Diaw, S., Lbath, R., Coulette, B.: Specification and Implementation of SPEM4MDE, a metamodel for MDE software processes. In: Proceedings of SEKE 2011, pp. 646–653. Knowledge Systems Institute Graduate School (2011), http://www.ksi.edu/seke/Proceedings/seke11/173_Samba_DIAW.pdf
26. Chou, S.-C.: A process modeling language consisting of high level UML-based diagrams and low level process language. Journal of Object Technology 1(4), 137–163 (2002), http://dx.doi.org/10.5381/jot.2002.1.4.a3
27. Bendraou, R., et al.: Definition of an executable SPEM 2.0. In: Proceedings of the 14th Asia-Pacific Software Engineering Conference (APSEC 2007), pp. 390–397. IEEE Computer Society (2007)
28. Object Management Group: Software & Systems Process Engineering Metamodel Specification (SPEM) Version 2.0 formal specification – formal/2008-04-0 (2008), http://www.omg.org/spec/SPEM/2.0/PDF/
29. Amrani, M., et al.: Towards a Model Transformation Intent Catalog. In: Proceedings of the First Workshop on the Analysis of Model Transformation (AMT 2012), pp. 3–8. ACM (2012)
30. Amrani, M., et al.: A Tridimensional Approach for Studying the Formal Verification of Model Transformations. In: Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST 2012), pp. 921–928. IEEE Computer Society (2012)

Traceability Links in Model Transformations between Software and Performance Models

Mohammad Alhaj and Dorina C. Petriu

Dept. of Systems and Computer Engineering, Carleton University,
1125 Colonel By Drive, Ottawa, Ontario, Canada, K1S 5B6
{malhaj,petriu}sce.carleton.ca

Abstract. In Model Driven Engineering, traceability is used to establish relationships between various software artifacts during the software life cycle. Traceability can be also used to define dependencies between related elements in different models, to propagate and verify properties from one model to another and to analyze the impact of changes. In this paper we describe how to define typed trace-links between different kinds of models in our model transformation chain PUMA4SOA, which generates Layered Queuing performance models from UML software models of service-oriented applications. The goal of PUMA4SOA is to help evaluate the performance of SOA systems in the early development phases. In our approach, the traceability links are stored externally in a new model, which maintain traces separately from the source and target models they refer to. We illustrate how traceability links can be used to propagate the results of the performance model back to the original software model.

Keywords: Software Performance Engineering, SOA, Traceability, Trace-Links, Aspect-oriented modeling, Model transformation, Performance Analysis.

1 Introduction

Model-Driven Engineering (MDE) is a software development paradigm that changes the focus from code to models. Many models of different types are used to describe the software under development in different lifecycle phases and at different levels of abstractions. Models in different modeling languages are created, updated and transformed either manually or automatically. This raises challenges related to the ability of managing and configuring the software models. In order to improve the coherence, and consistency of models used in an MDE process, it is useful to establish and maintain trace-links between models. Traceability is a known software approach used to establish relationships between various software artifacts (including all kinds of models) during the software life cycle. This allows the developers to understand the relationships and dependencies between artifacts, to maintain their consistency and to analyze the impact of changes in different artifacts.

F. Khendek et al. (Eds.): SDL 2013, LNCS 7916, pp. 203–221, 2013.

A wide range of traceability approaches have been discussed in the literature. The survey in [1] discusses the state of the art of traceability approaches in MDE, classifying them into three categories: 1) *requirement-driven*, 2) *modeling*, and 3) *transformation* approaches. In requirement-driven approaches, traceability is defined in the requirement models as "the ability to describe and trace the requirement specifications forward and backward in the life cycle during the software development" [2]. The modeling approaches are focusing on using metamodels and models to define trace-links. In the transformation approaches, the traceability details are generated by using model transformations. This can be done by creating trace-links between the source and target model elements during the model transformation. In terms of storing and managing traceability, two approaches are proposed in [3]: the *intra-model* and the *extra-model* approach. In the intra-model approach, traceability links are embedded inside the models they refer to as new model elements. In the extra-model approach, traceability links are stored externally in a new model, to maintain traces separately from the model they refer to. In terms of capturing the trace-links, [4] proposes two categories: *explicit trace-links* captured directly in the models using a suitable concrete syntax (such as UML dependencies), and *implicit trace-links* generated by a model operation (such as transformation or comparison).

Performance from Unified Model Analysis for SOA (PUMA4SOA) is a modeling framework introduced by the authors in [5,6] and [7], which generates a Layered Queueing Network (LQN) model from the UML design model of a SOA system; the LQN model is then used for analyzing the performance characteristics of the SOA system in early phases of the software life cycle. PUMA4SOA extends the PUMA framework developed previously in our research group [8], as presented in the next section. PUMA4SOA in its present state does not provide trace-links between the elements of the source and target models, which are needed for tracing, analyzing or propagating the impact of changes between different models.

The focus of this paper is on defining a traceability model for PUMA4SOA, which establishes trace-links between the elements of its different models. The paper is organized as follows: Section 2 gives a high-level view of the transformation chain in PUMA4SOA. Section 3 presents the proposed traceability metamodel, which defines trace-links between UML, CSM (Core Scenario Model) and LQN model elements; the metamodel is also extended to handle cases where aspect models are used. Section 4 illustrates the use of the traceability model with a Purchase Order system example. Section 5 presents related work and Section 6 gives the conclusion and directions for future work.

2 PUMA4SOA Transformation Principles

The PUMA4SOA transformation chain is described in Fig. 1. It takes as input three UML design models: 1) platform independent model (PIM), 2) deployment diagram, and 3) aspect platform models. The SOA systems are modeled

in UML [9] extended with two OMG standard profiles: MARTE [10] for adding performance annotations and SoaML [11] for describing the service architecture.

After getting the UML input design and selecting the generic aspect models for the platform operations required in the model, the next step is to transform the UML PIM model and the aspect models into intermediate models called Core Scenario Models (CSM [12]. The purpose of CSM is to bridge the semantic gap between the UML input design model and various performance models that could be generated. At the CSM level, the aspect platform models are composed with the platform independent model to generate the platform specific model (PSM). Transforming the CSM PSM to LQN is the final step in the model transformation chain [6,8]. The LQN model is an extension of queuing networks with the capability of representing nested services [13]. An LQN model defines a set of tasks representing software processes (threads) or hardware devices, which offer services called entries. An entry of a task can make a request to an entry of another task. Once the LQN model is generated, an existing LQN solver is used to produce the performance results (such as response time, throughput, and utilizations). The results are then fed back to the UML input design for further analysis.

The UML platform independent model (PIM) describes the structural and behavioral views of SOA systems at three levels of abstractions: a) the workflow model representing the business process layer, b) the service architecture model describing the invoked services, the participants, ports and service contracts, and c) the service behavior model giving details about the behavior of the invoked services. The deployment diagram represents the configuration of the SOA system, showing the allocation of software to hardware resources. The aspect platform models represent platform operations provided by the underlying service middleware, such as service invocation, service publishing and service discovery. Each aspect model can be seen as a template with generic parameters, which will be bound eventually to concrete values just before the respective aspect will be composed with the PIM in all places (join points) where a platform operation needs to be executed. For instance, a "service invocation" aspect will be composed with the PIM for every service invocation contained in the PIM. The aspect composition can take place at three levels, as discussed in [7]: UML, CSM or LQN level. In Fig. 1, the aspect composition is performed at the CSM level, which has certain advantages [7]. The final result of the composition is a platform specific model (PSM) expressed in this case in CSM.

PUMA4SOA also defines a so-called *performance completion (PC) feature model* that represents the variability (i.e., alternatives) in the service platform. It provides the choice to select between multiple aspects based on the business requirements for the given application. The "performance completion" concept was introduced in [14], where "completions" close the gap between the abstract design models and the functions provided by a platform external to the design model. In [15], a PC feature model is used to define the variability in platform choices, execution environments and other external factors that might impact the system performance. A concrete example of PC-feature model can be found in [7].

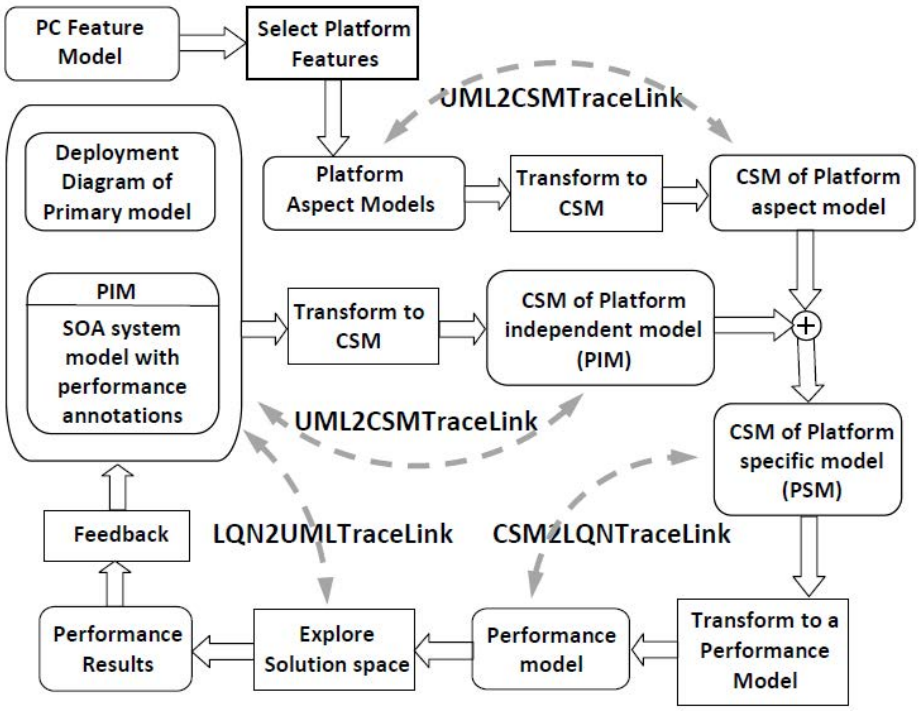


Fig. 1. PUMA4SOA approach

A high-level view of the transformation chain from UML software models to CSM and then to LQN is shown in Fig. 2. Please note that only the behavioural view of the UML model is presented in Fig. 2; the service architecture, the deployment and the platform aspect models are not shown, although they do contribute to the CSM derivation. Figure 2 emphasizes the fact that workflow and the service-providing components are represented separately in the UML model, contributing to distinct parts of the CSM and LQN models. The basic transformation principles are as follows:

a) A UML workflow model (normally an activity diagram) will generate a top-level scenario in CSM, which in turn will generate a LQN reference task that embeds an LQN activity graph corresponding to the workflow activities.
b) A service-providing run-time component in UML will generate a component in CSM executing subscenarios that represent the behaviour of services; each service subscenario is invoked by the workflow steps or other services. In turn, this will generate an LQN task with entries modeling each provided service.
c) An activity or an execution occurrence corresponding to a message in UML will generate a Step in CSM and an activity or phase in LQN.

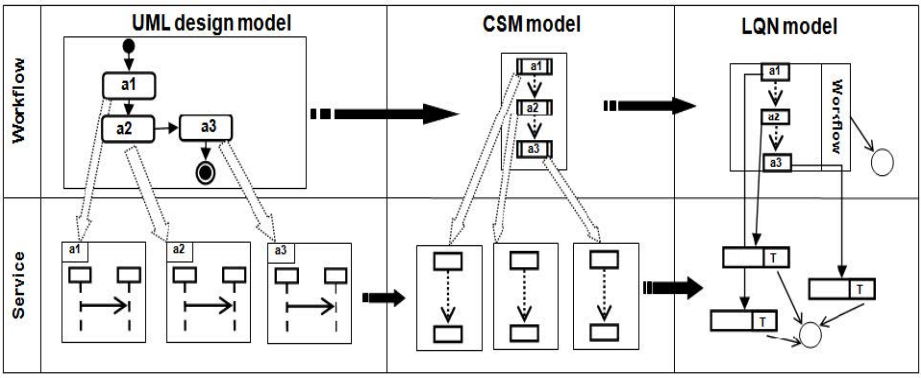


Fig. 2. Transformations in PUMA4SOA

d) A processing node in UML will generate a processor in CSM and a hardware device representing a processor in LQN.
e) An aspect model representing a platform operation will generate a subscenario in CSM, which will be woven into the PIM model when such operations are called.

3 Traceability Metamodel of PUMA4SOA

We used an approach similar to [16] for defining the PUMA4SOA traceability metamodel. The trace-links are classified into three groups: *UML2CSMTraceLink* between the UML and CSM elements, *CSM2LQNTraceLink* between CSM and LQN elements and *LQN2CSMTraceLink* between LQN and UML elements. The first two correspond to the model transformations shown in Fig. 2, while the third can be derived from the first two and it is used for feeding back to the UML model the LQN results. The three trace-links groups are aggregated into *TraceModel* (see Fig. 3).

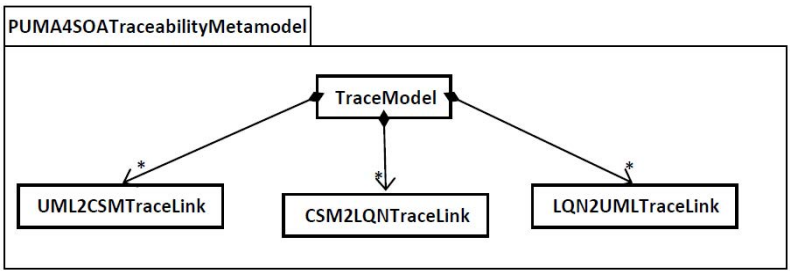


Fig. 3. PUMA4SOATraceabilityMetamodel top level

3.1 UML to CSM Traceability

The first group of trace-links in PUMA4SOA is defined between the elements of the UML input design models and the CSM model. The UML models are built using several types of UML diagrams, i.e. activity diagram (AD), component diagram, sequence diagram (SD) and deployment diagram. For simplicity, we will show here the trace-links for a subset of UML model elements.

To define *UML2CSMTraceLink*, we use the UML metamodel and the CSM metamodel, as the purpose is to capture trace-links between models elements which conform to those two metamodels. Each trace-link between an element of the source model and an element of the target model has its own type. It also has two associated properties (modeled as association roles in UML): the *source* refers to a metaclass in the *UMLMetamodel* and the *target* to a metaclass in the *CSMMetamodel*. An example of trace-link type between an *Action* element in the UML metamodel and a *StepType* element in the CSM metamodel is *Action-StepTypeTL*. The trace-links are derived during model transformation from to the mapping between corresponding UML and CSM elements. The relationships between the source and target model elements can be one-to-one (such as *Node* and *ProcessingResource*), one-to-many, many-to-one (such as *ActivityPartition*, *LifeLine* and *ComponentType*), or many-to-many.

All trace-link metaclasses inherit from *UML2CSMTraceLink*, which is aggregated into *TraceModel*. Figure 4 presents a subset of the traceability metamodel between UML and CSM. A subset of UML and CSM metamodels are represented at the top and the bottom of the figure, respectively.

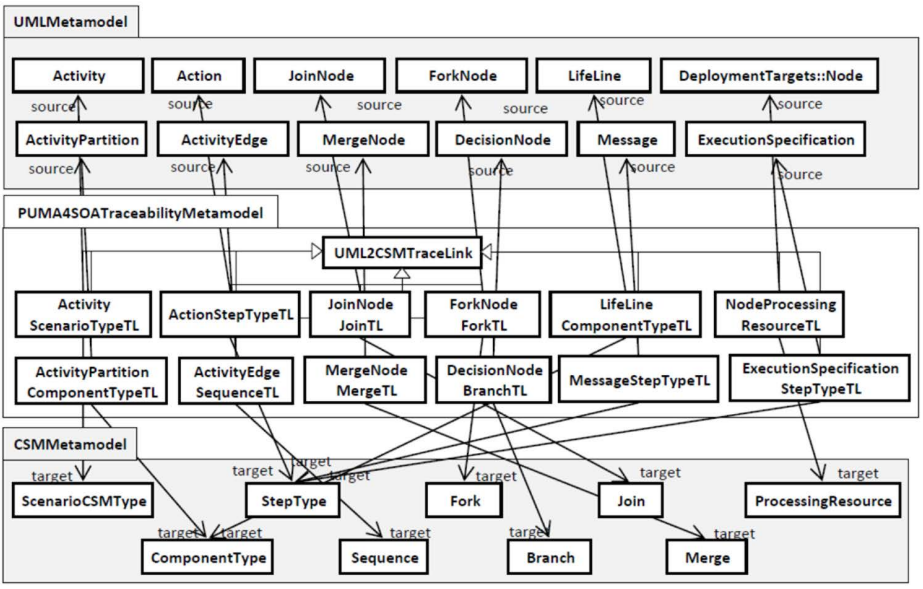


Fig. 4. Trace-Links between the elements of UML and CSM

3.2 CSM to LQN Traceability

The next group of trace-links in PUMA4SOA is defined between the elements of CSM and LQN model. We use the same procedure as in the previous section. The purpose is to capture the traceability between models that conform to the CSM metamodel and to the LQN metamodel. The metaclasses in PUMA4SOA *TraceabilityMetamodel* have two associated properties: the *source* refers to a metaclass in the *CSMMetamodel* and the *target* to a metaclass in the *LQN-Metamodel*. When an element in the source or the target model is not mapped during the model transformation, it means that it does not have equivalent element(s) in the other model. In this case a trace-link will not be defined for this element. As an example, the *OutputResultType*, which is defined in the LQN metamodel to create elements that store the results, is not linked with a CSM element; however it will have trace-links to an UML element to propagate the LQN output results. Figure 5 presents a sample of trace-links between the CSM and LQN.

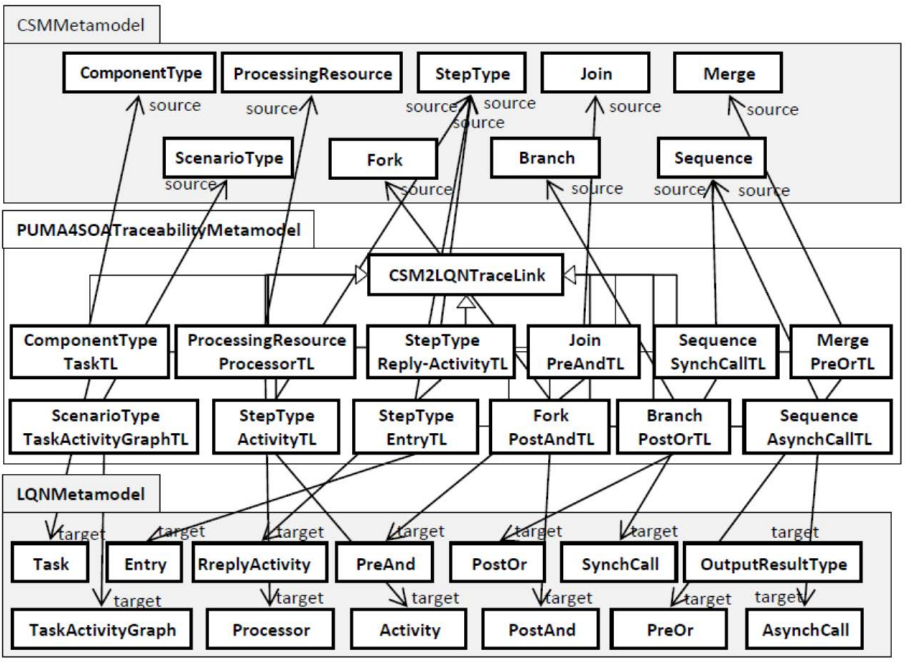


Fig. 5. Trace-Links between the elements of CSM and LQN

3.3 LQN to UML Traceability

The third group of trace-links in PUMA4SOA is defined between the LQN and UML model elements, and can be derived from the combined effect of the two

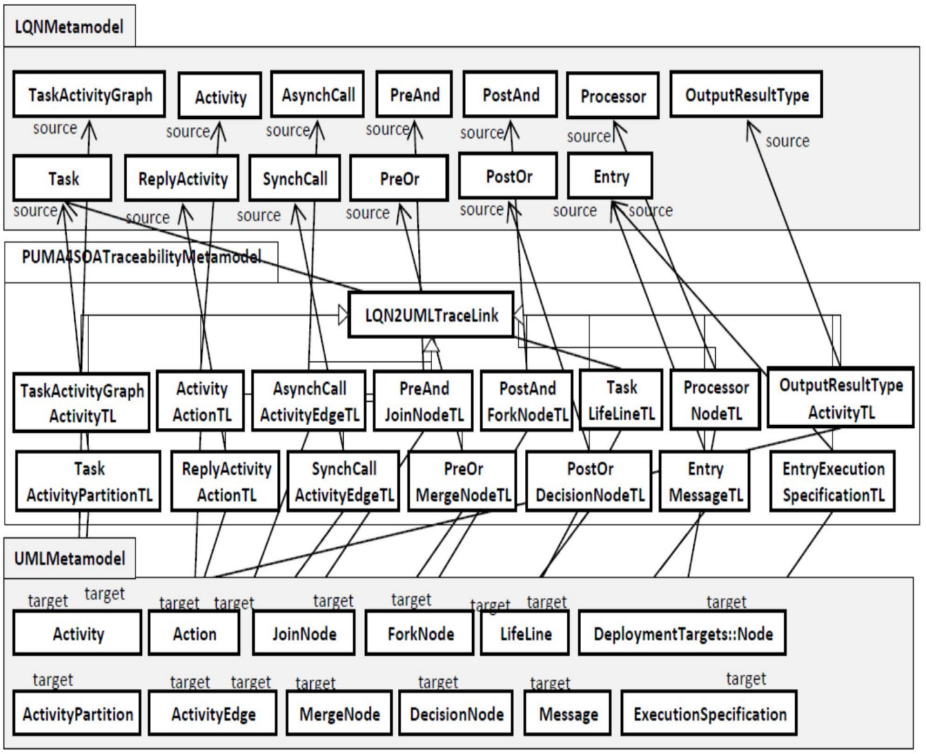


Fig. 6. Trace-Links between the elements of LQN and UML

transformations from Fig. 2, as there is no direct transformation from UML to LQN. The LQN2UML metaclasses define two associated attributes: the *source* refers to a meta-class in the *LQNMetamodel* and the *target* to metaclass in the *UMLMetamodel*. Figure 6 presents a sample of trace-links between LQN and UML.

As shown in Fig. 1, the LQN model is derived by a model transformation from the platform dependent CSM model, which in turn was generated by composing the platform aspect models into the PIM at the CSM level. Since the UML model does not contain a PSM, the mapping from LQN to UML encounters some difficulties, as discussed in the next section.

3.4 Trace-Links Related to Aspect Models

The trace-links between the LQN and UML models have not been properly defined yet in Section 3.3, because the LQN model is a Platform Specific Model (PSM), while the UML input design models do not contains a PSM, only the PIM and Generic Aspect Models (see Fig. 7). For instance, in this example we modeled the service invocation operation as a generic aspect, which involves

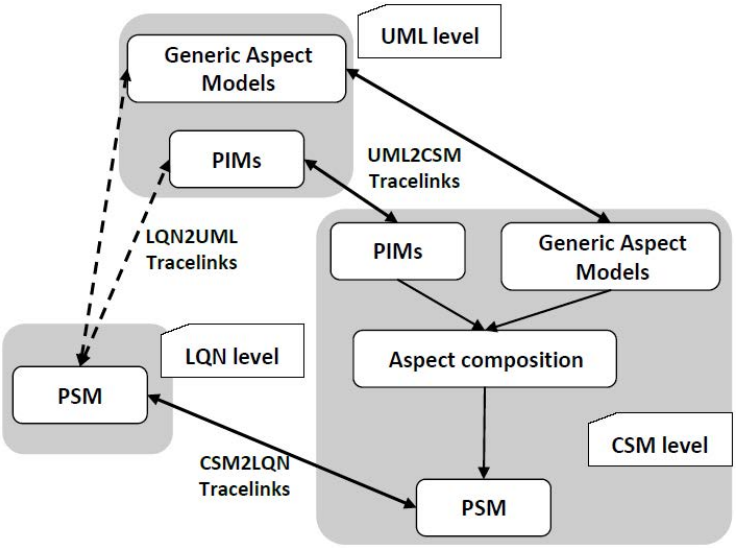


Fig. 7. Traceability for aspect models

XML parsing and marshaling/unmarshaling of the SOAP messages exchanged between the service client and the service provider. These actions are usually performed by a service middleware process, which may be instantiated multiple times on different processors, depending on the deployment of the components providing and requesting services.

Some of the concrete LQN elements obtained from the instantiation of a generic aspect model (such as the concrete instance of a service middleware used for a certain service invocation) cannot be traced back to a UML element, since only the generic middleware counterpart exists in the UML model. In this section, we extend the traceability metamodel defined in the previous sections to address this issue. The extension is used to define additional trace-links between some of the already defined trace-links. These extended trace-links allow for the mapping of the generic elements from the CSM level to LQN level. If LQN carries information about the generic model element corresponding to each concrete model element, trace-links between the concrete LQN elements can use that information to point to the generic UML counterpart.

Figure 8 shows an example of how the PUMA4SOA *TracibilityMetamodel* was extended. Two extra trace-links are created that make it possible to link a concrete LQN element (such as a concrete middleware task) with information about its generic counterpart (the generic role representing the middleware process in the aspect model). The first trace-link *ProcessorProcessorTL* defines the trace-links between *NodeProcessingResourceTL* (UML2CSM) and *ProcessingResourceProcessorTL* (CSM2LQN), and it owns two associated attributes: the *concreteSource* to define the concrete LQN *Processor*, and the *genericTarget* to define the generic CSM *ProcessingResource*. The function *SetProcessor* is

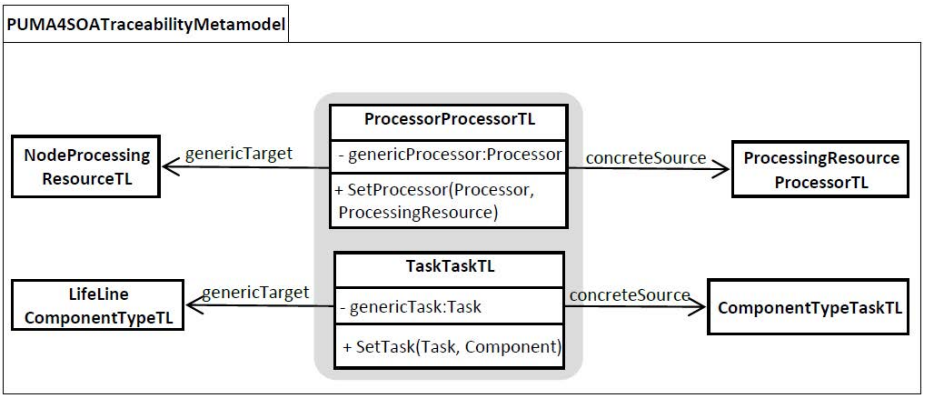


Fig. 8. Extension of PUMA4SOATraceabilityMetamodel

used to set an instance of the defined attribute *genericProcessor* to its equivalent generic CSM *ProcessingResource*. The second *TaskTaskTL* defines the trace-links between *LifeLineComponentTypeTL* (UML2CSM) and *ComponentTypeTaskTL* (CSM2LQN), and it also owns two associated attributes: the *concreteSource* to define the concrete LQN *Task*, and the *genericTarget* to define the generic CSM *ComponentType*. The function *SetTask* is used to set an instance of the defined attribute *genericTask* to its equivalent generic CSM *ComponentType*. By using such trace-links, a concrete middleware task running on a concrete processor in the LQN model can be traced to the generic UML counterparts and can also indicate the context in which they were instantiated.

4 Example: Traceability Model of Purchase Order System

In previous work [7], we used PUMA4SOA to build the UML design model of a Purchase Order (PO) system and to generate a LQN model in order to study its performance properties. In this section, we use the same example to create the traceability model which defines the trace-links between the model elements at UML, CSM and LQN levels.

A brief description of the PO case study is given first. The platform independent model (PIM) contains two parts: a) the workflow (Fig. 9) represented as a UML activity diagram, which describes the actions of receiving, invoicing, scheduling and shipping an order; and b) the service behavior models, which describe the details of each activity in the workflow.

ProcessSchedule (Fig. 10) is an example of service behavior model; the detailed models for the rest of the activities from Fig. 9 are similar, but not shown.

The deployment diagram (Fig. 11) shows the allocation of software components to the hardware nodes. The aspect platform models describe the structure and behavior of the platform operations (in this case the service middleware used

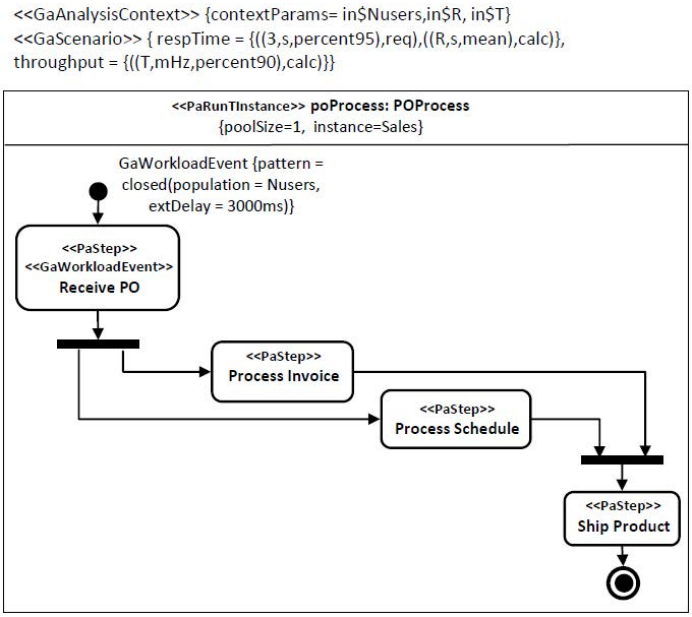


Fig. 9. Workflow model of Purchase Order system

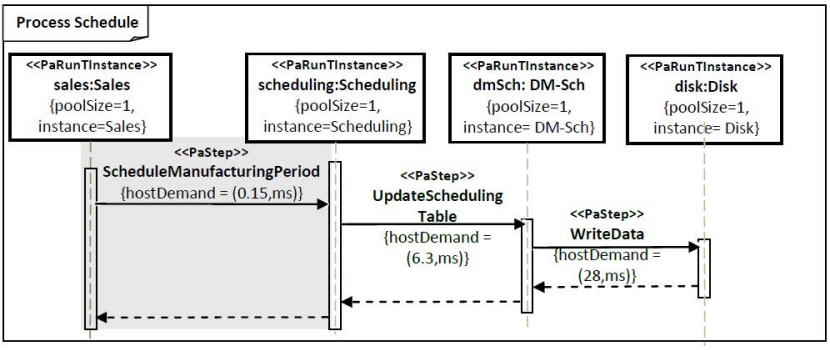


Fig. 10. Service behavior model of *ProcessSchedule*

for service invocation, discovery, publishing) in a generic format. Each middleware operation is represented by a different aspect model that has a structural and behavioural view. In this example we use only the Service Invocation operation, which describes the message construction (including XML parsing and marshaling/unmarshaling) and sending/receiving of the SOAP messages for service request and service reply, respectively (for more details, please refer to [7]).

At the CSM level, two separate CSM models are generated: one from the UML PIM (Fig. 12) and the other from the UML generic platform aspect model (service invocation in our case). Figure 12 contains two CSM scenario graphs

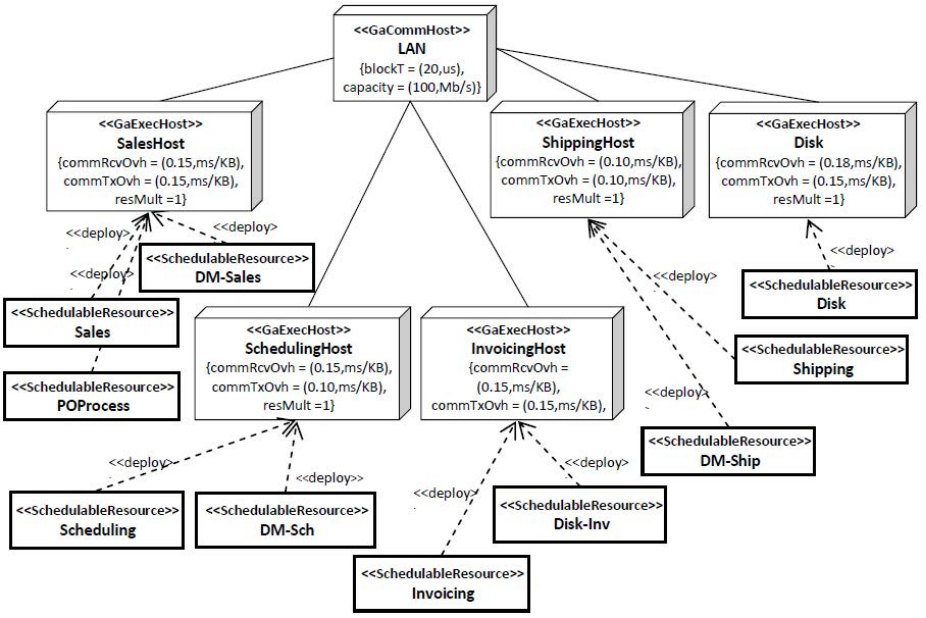


Fig. 11. Deployment diagram of Purchase Order system

composed of steps: the left one represents the workflow CSM corresponding to Fig. 9, and the right one the details of the composite step *ProcessSchedule* obtained from Fig. 10. The Aspect Oriented Modeling (AOM) technique is then used to generate a platform specific model (PSM) in CSM by composing the instantiated aspect models into the primary model (Fig. 13). The gray steps from Fig. 13 represent the woven aspects for the service request and reply.

The LQN model (Fig. 14) is generated next from the CSM Platform Specific Model. After generating the LQN model, the LQN solver produces complete performance results for the PO system, such as utilization of all resources, response times and throughput for scenarios, etc. By identifying the performance hotspots in the system, the results are fed back to the UML level to improve the software design models. Trace-links are used to propagate the performance results and to feed back the suggested improvements from the LQN to UML.

The workflow model has two MARTE stereotypes in Fig. 9. The first stereotype *<<GaAnalysisContext>>* defines the *contextParams* attribute which declares two variables: *$Nusers* for number of users and *$R* for response time. The second stereotype *<<GaScenario>>* captures system-level behavior and attaches allocations and resource usage to it. It defines many attributes, such as *respTime*, *utilization* and *throughput*. In our case, the *respTime* has two values: a required value (no more than 3 seconds) and a calculated value that will be assigned to *$R*, (the variable defined in the *contextParams*). The *$Nusers*

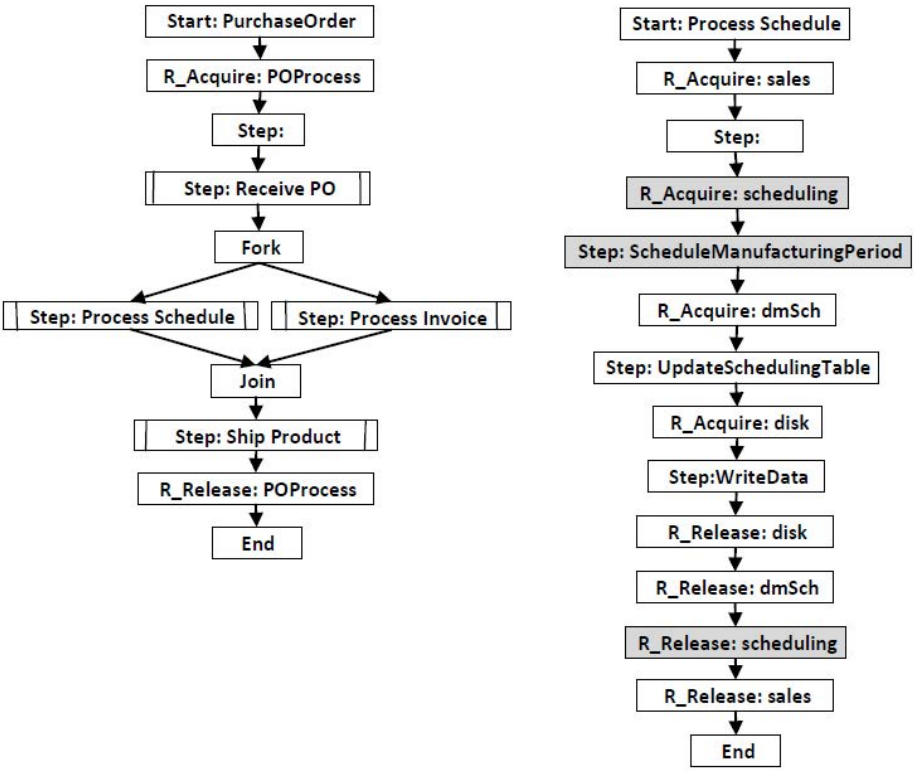


Fig. 12. The CSM model of PIM for Purchase Order system

variable is initialized by the modeler, and the $R variable will be assigned output results from the LQN model using trace-links.

PUMA4SOA allows for different types of performance analysis, such as sensitivity analysis, pass/fail and finding optimal values, which may require multiple iterations of the model transformation chain. In our case, the performance requirement specifies that the response time should be maximum 3s in average, which may require multiple model changes to achieve it. Hence, to manage the propagation of the model changes, we define a traceability model as in Fig. 15. For simplicity, trace-links for only three UML elements are defined: *POSystem:Activity*, *Sales:LifeLine* and *InvoicingHost:Node*. Some of the model elements have more trace-link relationships. For example, there are two trace-links defined between some LQN elements and the UML element *POSystem:Activity*. The trace-link between *POSystem:TaskActivityGraph* and *POSystem:Activity* corresponds to the relationship caused by the model transformation that generated LQN from UML. The other trace-link between *results:OutputResultType* and *POSystem:Activity* is used to propagate the generated output result of the LQN, such as response time and throughput. The *OutputResultType* is defined

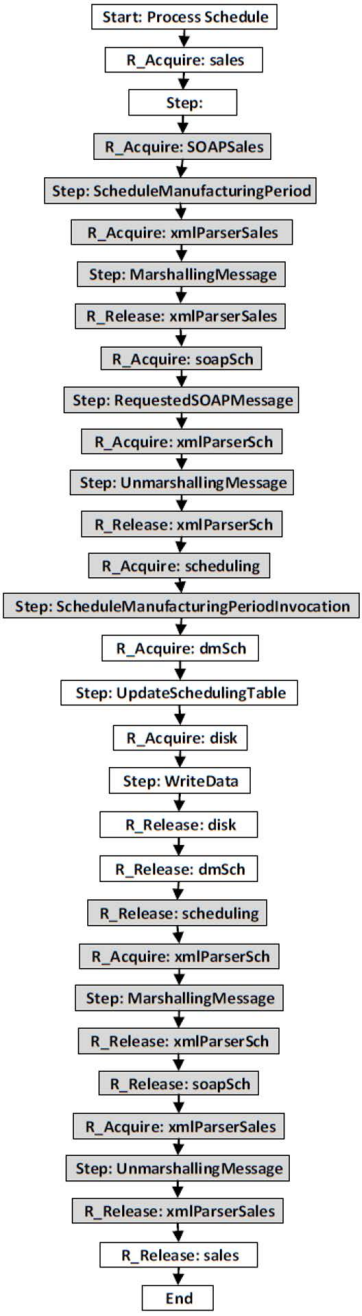


Fig. 13. Result of composition for *ProcessSchedule*

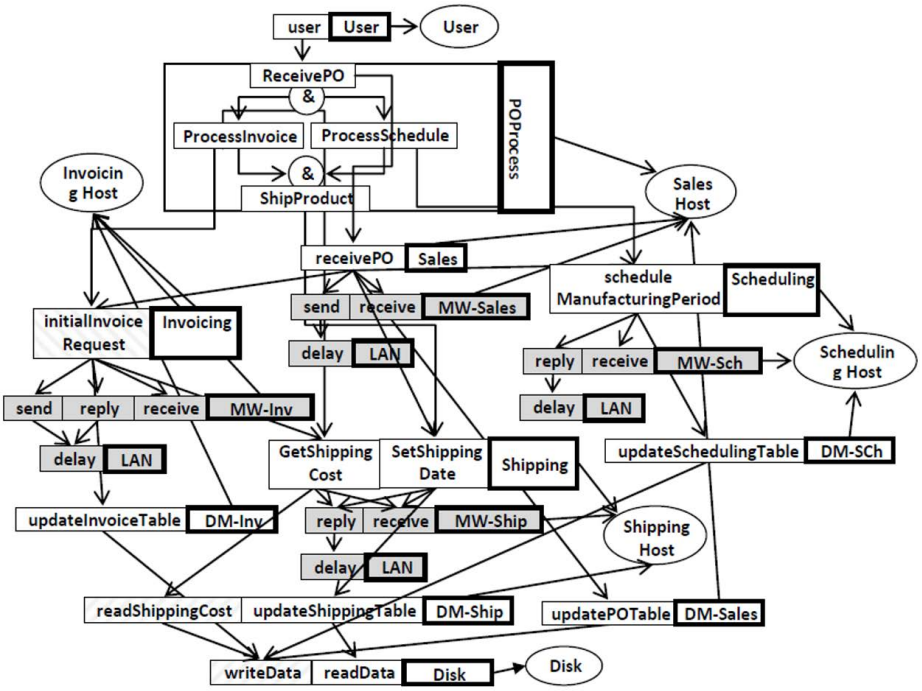


Fig. 14. LQN model of Purchase Order system (PSM)

in the LQN metamodel to create elements that store the results. To meet the performance requirement defined for the PO workflow (required mean response time <= 3s), three iterations of the model transformation chain have been executed as described in Table 1 for *$Nusers* = 100.

The first iteration represents the base case, where the multiplicity of all tasks and hosts equals one. The response time and throughput are calculated and passed to the *POSystem:Activity* through the trace-link *perfResultsTL: OutputResultTypeActivityTL*. Based on the LQN solver results, the *Sales:Task* is found to be the bottleneck server.

This is a case of software bottleneck, usually solved by increasing the concurrency level by having more threads in the pool. Thus we change the multiplicity of Sales to 15. The new value is propagated through the trace-links to their corresponding element in UML and CSM. The same procedures happen in the second iteration, except that the bottleneck is the processor *InvoicingHost:Node*. We increase its multiplicity value to 5 (which means using 5 cores instead of one). In the third iteration, the calculated response time is less than 3s, so it meets the performance requirement. More iterations could be done to find out if the requirement can be met with fewer resources.

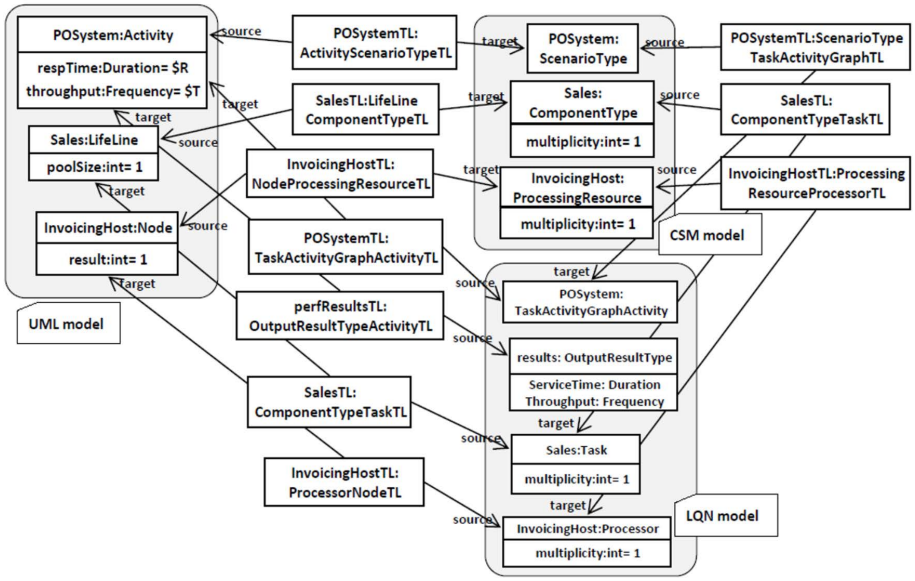


Fig. 15. Sample of traceability model of PO system

Table 1. Performance analysis of PUMA4SOA

$Nuser= 100, respTime = ((3,s, mean), req)			
	UML	CSM	LQN
1	POSystem:Activity {respTime = $R} *After feedback $R=7014ms*	POSystem: ScenarioType	POSystem: TaskActivityGraph results: OutputResultType {serviceTime = 7014ms}
	Sales: LifeLine {poolSize = 1}	Sales: ComponentType {multiplicity = 1}	Sales: Task {multiplicity = 1} **Solve software bottleneck -> multiplicity=15**
	InvoicingHost: Node { resMult = 1}	InvoicingHost: ProcessingResource {multiplicity= 1}	InvoicingHost: Processor {multiplicity = 1}
2	POSystem: Activity {respTime = 7014ms} *After feedback $R=3442ms*	POSystem: ScenarioType	POSystem: TaskActivityGraph results: OutputResultType {serviceTime = 3442ms}
	Sales:LifeLine { poolSize = 15}	Sales: ComponentType {multiplicity = 15}	Sales:Task { multiplicity = 15}
	InvoicingHost: Node { resMult = 1}	InvoicingHost: ProcessingResource {multiplicity = 1}	InvoicingHost: Processor { multiplicity = 1 } **Solve hardware bottleneck -> multiplicity=5**
3	POSystem: Activity {respTime=3442,ms} *After feedback $R=1607ms*	POSystem: ScenarioType	POSystem: TaskActivityGraph results: OutputResultType {serviceTime = 1607, ms}
	Sales: LifeLine {poolSize = 15}	Sales:ComponentType {multiplicity = 15}	Sales: Task {multiplicity = 15}
	InvoicingHost:Node { resMult = 5}	InvoicingHost: ProcessingResource {multiplicity = 5}	InvoicingHost: Processor { multiplicity = 5}

5 Related Work

Traceability information is used to manage the artifacts of a software system during its development life cycle. The survey in [1] has classified three traceability approaches: a) the *requirement-driven* approach which describe the traceability of requirement specification to its subsequent deployment and use; b) the *modeling* approach, which is mainly focused on defining tracing mechanisms for a modeling language at the metamodel level (such as special tracing relationships); and c) the *transformation* approach, where the tracing process is performed between a source and a target model during the model transformation, by creating trace-links between the source and the target model elements. PUMA4SOA uses the transformation approach to define trace-links between the elements of its three modeling languages, i.e. UML, CSM and LQN.

There are several papers in the literatures using the transformation approach; in [17] the author presented a method of attaching traceability generation code to pre-existing ATL programs [18]. The method produces a loosely coupled traceability, which can be used for any kind of traceability range and format. In [19] the authors presented a method of generating annotated models which contain traceability information, by merging the primary models with their defined trace models. The generated trace-links can be stored internally, where the trace-links are embedded as new elements inside the target models they refer to, or externally where the trace-links are stored separately in a new model. In [20] the authors proposed a traceability framework, implemented in the model-oriented language Kermeta, to facilitate modeling transformations. Using a trace metamodel, the framework allows for tracing the transformation chain within Kermeta. Model transformation trace-links are defined in the metamodel as a set of source nodes and target nodes.

Two ways to manage the complexity of traceability information in MDE where introduced in [16]. One is by identifying the trace-links through a process called Traceability Elicitation and Analysis Process (TEAP), which is mainly used to extract and analyze traceability relationships within an MDE process, to determine how these relationships would fit into a trace-link classification. The second way is by describing a strict metamodeling approach, which defines semantically rich trace-links between the elements of different models. Three characteristics are defined for the semantically rich trace-links: a) to be typed, b) to conform to a case-specific traceability metamodel, and c) to define a set of constrains within the case specific meatmodel to validate the requirements that cannot be captured by the metamodel itself. In this paper, we used a metamodeling approach similar to [16] to create a traceability metamodel for our PUMA4SOA, which defines trace-links between the elements of the UML, the CSM and the LQN elements. Also, in order to handle the problem of traceability loss when applying aspect-oriented techniques we extended the metamodel by defining new trace-links between the previously defined trace-links.

Similar to our work, in [21,22], the authors use trace-links between a software model (Smodel) and the corresponding Performance model (Pmodel) of a service-oriented system to study the change propagation when applying design patterns

to a Smodel. The purpose is to develop methods for incremental propagation of changes from Smodel to Pmodel, in order to study the performance effects of design patterns.

6 Conclusions

The paper focuses on defining a traceability metamodel for PUMA4SOA, which is used to define trace-links between different types of models, namely UML, CSM and LQN. We also addressed the problem of trace-links when applying aspect-oriented modeling techniques. Trace-links are used in PUMA4SOA to analyze the impact of changes at different model levels, i.e. UML, CSM and LQN, and to feed back the performance results from the LQN model to the UML model. We illustrate the proposed approach with a simple PO system.

In the future, we are planning to define formally the constraints within PUMA4SOA traceability meatmodel to express well-formedness rules that cannot be captured by the metamodel itself. We are also working on implementing the traceability metamodel proposed in this paper, integrating it with the existing PUMA4SOA model transformations.

Acknowledgements. This research was partially supported by the Natural Sciences and Engineering Research Council (NSERC) and industrial and government partners, through the hSITE Strategic Research Network.

References

1. Galvao, I., Goknil, A.: Survey of Traceability Approaches in Model-Driven Engineering. In: Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference, pp. 313–326. IEEE Computer Society (2007)
2. Gotel, O.C.Z., Finkelstein, A.C.W.: An Analysis of the Requirements Traceability Problem. In: Proceedings of the International Conference on Requirements Engineering, pp. 94–101. IEEE Computer Science Press (1994)
3. Kolovos, D., Paige, R.F., Polack, F.A.C.: On-Demand Merging of Traceability Links with Models. In: From: 3rd ECMDA Traceability Workshop (2006)
4. Paech, B., von Knethen, A.: A Survey on Tracing Approaches in Practice and Research. Technical Report IESE Report Nr. 095.01/E, Fraunhofer - Institute of Experimental Software Engineering (2002)
5. Alhaj, M.: Automatic generation of performance models for SOA systems. In: Proceedings of the 16th International Workshop on Component-Oriented Programming (WCOP 2011), pp. 33–40. ACM (2011)
6. Alhaj, M., Petriu, D.C.: Approach for generating performance models from UML models of SOA systems. In: Proceedings of the 2010 Conference of the Center for Advanced Studies on Collaborative Research (CASCON 2010), pp. 268–282. IBM (2010)
7. Alhaj, M., Petriu, D.C.: Using Aspects for Platform-Independent to Platform-Dependent Model Transformations. International Journal of Electrical and Computer System (IJECS) 1(1), 35–48 (2012)

8. Woodside, C.M., Petriu, D.C., Petriu, D.B., Shen, H., Israr, T., Merseguer, J.: Performance by Unified Model Analysis (PUMA). In: Proceedings of the 5th International Workshop on Software and Performance (WOSP 2005), pp.1–12. ACM (2005)
9. Object Management Group: Unified Modeling Language Superstructure Version 2.2 formal/2009-02-02, http://www.omg.org/spec/UML/2.2/Superstructure/PDF
10. Object Management Group: UML Profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE) Version 1.1 formal/2011-06-02, http://www.omg.org/spec/MARTE/1.1/PDF
11. Object Management Group: Service oriented architecture Modeling Language (SoaML) formal/2012-03-01, http://www.omg.org/spec/SoaML/1.0/PDF
12. Petriu, D.B., Woodside, C.M.: An intermediate metamodel with scenarios and re-sources for generating performance models from UML designs. Software and Systems Modeling 6(2), 163–184 (2007)
13. Woodside, C.M., Neilson, J.E., Petriu, D.C., Majumdar, S.: The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-like Distributed Software. IEEE Transactions on Computers 44(1), 20–34 (1995)
14. Woodside, C.M., Petriu, D.B., Siddiqui, K.H.: Performance-related Completions for Software Specifications. In: Proceedings of the 24th International Conference on Software Engineering (ICSE 2002), pp. 22–32. ACM (2002)
15. Tawhid, R., Petriu, D.C.: Automatic Derivation of a Product Performance Model from a Software Product Line Model. In: Proceedings of the 2011 15th International Software Product Line Conference (SPLC 2011). IEEE Computer Society (2011)
16. Paige, R.F., Drivalos, N., Kolovos, D.S., Fernandes, K.J., Power, C., Olsen, G.K., Zschaler, S.: Rigorous identification and encoding of trace-links in model-driven engineering. Software and Systems Modeling (SoSyM) 10(4), 469–487 (2011)
17. Jouault, F.: Loosely Coupled Traceability for ATL. In: Traceability Workshop at European Conference on Model Driven Architecture (ECMDA-TW), pp. 29–37 (2005)
18. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
19. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: Merging Models with the Epsilon Merging Language (EML). In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 215–229. Springer, Heidelberg (2006)
20. Falleri, J., Huchard, M., Nebut, C.: Towards a Traceability Framework for Model Transformations in Kermeta. In: Traceability Workshop at European Conference on Model Driven Architecture (ECMDA-TW), pp. 31–40 (2006)
21. Mani, N., Petriu, D.C., Woodside, C.M.: Propagation of Incremental Changes to Performance Models due to SOA Design Pattern Application. In: Proceedings of the International Conference on Software Engineering (ICPE 2013) (2013)
22. Mani, N., Petriu, D.C., Woodside, C.M.: Studying the Impact of Design Patterns on the Performance Analysis of Service Oriented Architecture. In: Proceedings of the 2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2011), pp. 12–19. IEEE Computer Society (2011)

Refactorings in Language Development with Asymmetric Bidirectional Model Transformations

Martin Schmidt[1], Arif Wider[2], Markus Scheidgen[2], Joachim Fischer[2], and Sebastian von Klinski[1]

[1] Beuth Hochschule für Technik Berlin
- University of Applied Sciences -
Luxemburger Straße 10, D-13353 Berlin, Germany
{maschmidt,klinski}@beuth-hochschule.de
[2] Humboldt-Universität zu Berlin
Department of Computer Science
Unter den Linden 6, D-10099 Berlin, Germany
{wider,scheidge,fische}@informatik.hu-berlin.de

Abstract. Software language descriptions comprise several heterogeneous interdependent artifacts that cover different aspects of languages (abstract syntax, notation and semantics). The dependencies between those artifacts demand the simultaneous adaptation of all artifacts when the language is changed. Changes to a language that do not change semantics are referred to as refactorings. This class of changes can be handled automatically by applying predefined types of refactorings. Refactorings are therefore considered a valuable tool for evolving a language.

We present a model transformation based approach for the refactoring of software language descriptions. We use asymmetric bidirectional model transformations to synchronize the various artifacts of language descriptions with a refactoring model that contains all elements that are changed in a particular refactoring. This allows for automatic, type-safe refactorings that also includes the language tooling. We apply this approach to an Ecore, Xtext, Xtend based language description and describe the implementation of a non-trivial refactoring.

Keywords: DSL evolution, language description, refactoring, bidirectional model transformations.

1 Introduction

Software languages evolve continuously [1] and software language engineering does not only include the initial development but also the continuous adaptation of software languages. Especially the engineering of domain-specific languages (DSLs) requires an agile process to evolve a language along rapidly changing user requirements.

F. Khendek et al. (Eds.): SDL 2013, LNCS 7916, pp. 222–238, 2013.

During this process, two distinct problem sets arise. First, languages are already used while they evolve, and artifacts written in a language need to be co-adapted to language adaptations. Secondly, the different artifacts that constitute the description of a language (and eventually the tooling of that language) need to be co-adapted when one of those artifacts changes. We call the former *vertical* and the latter *horizontal co-adaptation*.

In this paper, we are only concerned with horizontal co-adaptation. A language description (depending on the nature of that language) consists of several artifacts: an abstract syntax (i.e., a metamodel, e.g., an Ecore model), concrete syntax (e.g., Xtext grammar, GMF model), and description of semantics (e.g., model transformation rules or code generator). Fig. 1 depicts the different artifacts in language development and their dependencies. When one of these artifacts is changed to evolve the language (e.g., the multiplicity of a metamodel feature is changed), the other artifacts need to be changed, too (e.g., the code generator rules need to be adapted towards the changed metamodel).

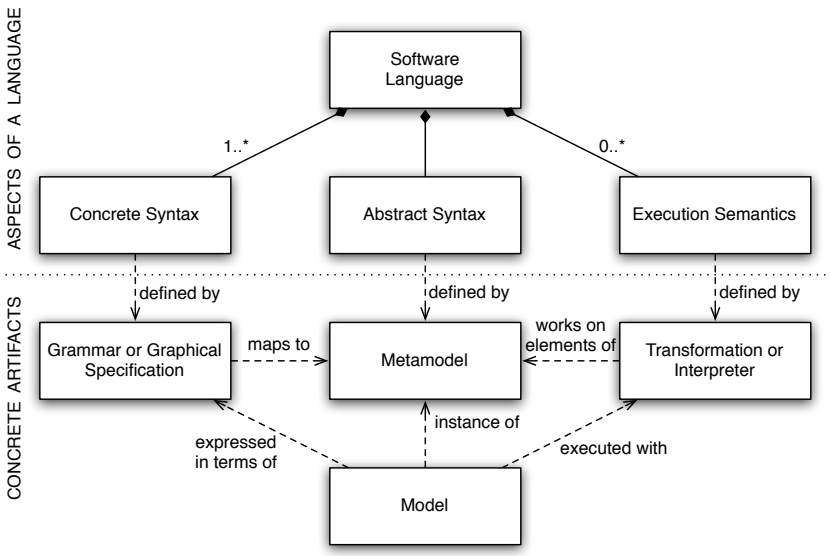


Fig. 1. Aspects of a language and concrete artifacts in metamodel-based language development

Refactorings play an integral role during the evolution of a language. Whereby a *refactoring* is described as a semantically invariant change of the language description [2]. This includes for example changing metamodel identifiers, moving features within the inheritance hierarchy (pull-up, push-down), changing the organization of grammar and transformation rules, etc.

In this paper, we present an approach that allows the refactoring of language description artifacts with automated co-refactoring of depending artifacts of the

same language description. We model these co-refactorings as *view-update relations* between a common view capturing the refactoring-specific information and the dependent artifacts using asymmetric bidirectional model transformations [3]. This allows for a more declarative and modular description of refactorings which allows for more possibilities of verification and better reuse. We demonstrate the practicability of our approach by implementing refactorings on Ecore metamodels, Xtext grammars and Xtend-based code generators as artifacts.

The paper is organized as follows: In the first part we present different areas of language evolution and give an overview of existing and related work in language evolution. The following section describes a specific case study, which was decisive for this paper. Section 4 describes our approach for handling refactorings of interdependent heterogeneous artifacts in language development. Afterwards, we give an outlook of an implementation using model transformations. Finally, we conclude the paper including some discussion and show up possible directions for future work.

2 Background and Related Work

When we discuss the evolution of languages, the need for co-adaptation arises. We have multiple interdependent artifacts and if we change one the others have to change as well.

2.1 Horizontal vs. Vertical Co-adaptation

We can distinguish two forms of co-adaptation.

- First, when we change the language description (especially the metamodel) language instances have to be changed as well. We call this vertical co-adaptation: Changes need to be propagated from the meta-layer (top) down to the instance layer (bottom). Notable contributions to this kind of co-adaptation comes from Wachsmuth [4] and Hermannsdörfer [5].
- The second form of co-adaptation happens within the same layer, hence horizontal co-adaptation. Software engineers might use several languages to create one piece of software on the instance-layer, and language engineers might use different meta-languages to describe a single language (e.g., a DSL) on the meta-layer. In both cases heterogeneous interdependent artifacts are created. Since we mainly discuss horizontal co-adaptation (more specifically co-refactoring) in this paper, we discuss the related work in this field more detailed in the following subsections.

2.2 Horizontal Co-adaptation in Software Engineering

Software systems in general are often mixed-language systems. They are constructed with declarative descriptions for the user interface, imperative application logic implemented with a general-purpose programming language (GPL),

and other specific languages, e.g., configuration scripts, styling or plug-in management. Strein et al. investigated the evolution of a given *inter-language* software system [6]. They described the interdependencies in an object-oriented web application that is implemented with *ASP.NET*, *HTML*, *C#* and *Visual Basic*. They identified that modern *integrated development environments* (IDEs) support the evolution of certain artifacts through offering refactorings or simple adaptations like introducing getters and setters, but these IDEs are limited concerning the co-adaptation of dependent artifacts written in different languages. For transferring adaptations to other parts of a system Strein et al. developed an IDE called X-Develop. They captured refactoring-relevant information, concerning more than just one language, in a model and adapt this model, which is a typical approach for implementing refactorings in software engineering.

2.3 Horizontal Co-adaptation in Language Engineering

As languages evolve too, the development process of a domain-specific language has similarities to general software development [1]. Changes in a language specification can have an impact on the corresponding language tools [7]. In metamodel-based language development (i.e., where the metamodel that describes the abstract syntax is the central artifact) other meta-descriptions often reference the types defined by the metamodel. Therefore, many of the needed co-adaptations that are necessary when the metamodel changes can be detected by checking these references. Although multiple (meta-) languages are involved in DSL development, these languages are interconnected via the metamodel and form the description of one language. Therefore, we call this *intra-language evolution*. In this paper, we are concerned with intra-language evolution of DSLs and other software languages (e.g., modeling languages).

Pizka and Jürgens already captured the difficulties of DSL evolution and the need of co-adaptations for these systems [8]. For handling the evolution of a language they implemented *Lever* (Language Evolver) [9]. Lever provides different integrated DSLs for the description of grammars, the tooling, and the coupled evolution of these parts. Lever focuses on textual DSLs and allows only for adaptation of tooling after the grammar changes.

In this paper we present an approach for describing refactorings and show an exemplary implementation that works with established technologies (e.g., *EMF*, *Xtext*) and allows for co-adaptations resulting from changes to different kinds of artifacts at the metalevel. We also believe that our approach is applicable to the evolution of graphical DSLs.

3 Motivating Example: The NanoWorkbench

The motivation for the work that we present in this paper emerged from practical experiences during the development of an Xtext-based DSL for developing *optical nanostructures* (NanoDSL) and a corresponding integrated tool-suite for that

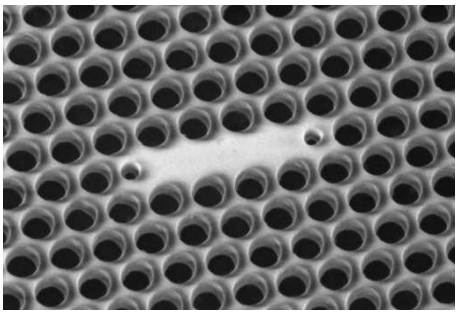
DSL (a *domain-specific workbench* called NanoWorkbench [10]). This project is subject of a cooperation with the nano-optics research group at the physics department.

The members of this group design geometrical structures that are smaller than the wavelength of optical light in order to affect the motion of photons in a similar way a semiconductor crystal affects the motion of electrons. The properties of these *photonic crystals* are tested by simulating the propagation of an electromagnetic pulse within the structure. There are different simulation methods for that, e.g., the *finite difference time domain method* (FDTD) or the *finite element method* (FEM). Fig. 2(a) shows a picture of a photonic crystal and Fig. 2(b) shows a schematic overview of the workbench incorporating different DSLs and different simulation methods, as well as a model-driven data management and model-driven communication channels for performing external experiments or computations.

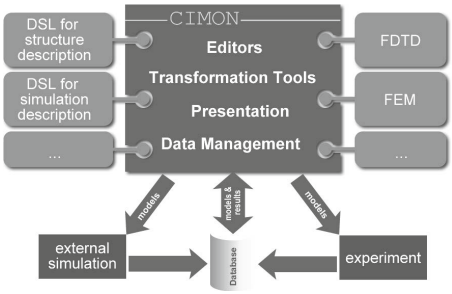
As we pursued an agile, iterative process to develop the DSL and its domain-specific workbench (with continuous consultation of the domain-experts), we identified problems similar to general software development: When implementing changes requested by the domain experts we had to change the design of the language specification several times. After solving change requests we manually adapted the generator and artifacts concerning the tooling to preserve consistency. Figure 3 gives an overview of corresponding generators that have to be adapted after the language changes.

As a concrete example, the metamodel of the NanoDSL started with a description/class for only adding cylinders as geometrical objects to the photonic crystal. This is the application the physicists mostly used. Later they wanted to add other structures like truncated cones or cuboids. With these structures they wanted to simulate manufacturing faults. For this change request we introduced a superclass for geometrical objects in general. The existing structure was renamed to *Cone* and became a subclass of the introduced class as shown in Figure 4. We identified that some attributes are more general, e.g., height or position in space, than others. To handle this redundancy we started to pull up these attributes.

For solving this change request, we applied at least three well-known refactorings - *rename, introduce superclass, pull up feature.* Although we could easily rewrite the core language description, we had to update the existing tooling and in our special case had to adapt two complex model-to-text transformations describing different execution semantics and one model-to-model transformation for a 2D-visualisation. Additionally, we implemented a transformation rule for each added subclass. For calling the rules we need to check the object's type via the `instanceof`-operator first for providing the correct transformation. These manual changes are time-consuming and error-prone. Thus, we identified the practical need for automatic co-refactorings of interdependent artifacts in DSL development.



(a) A photonic crystal [11]



(b) Overview of the domain-specific workbench

Fig. 2. A domain-specific workbench for the development of optical nanostructures

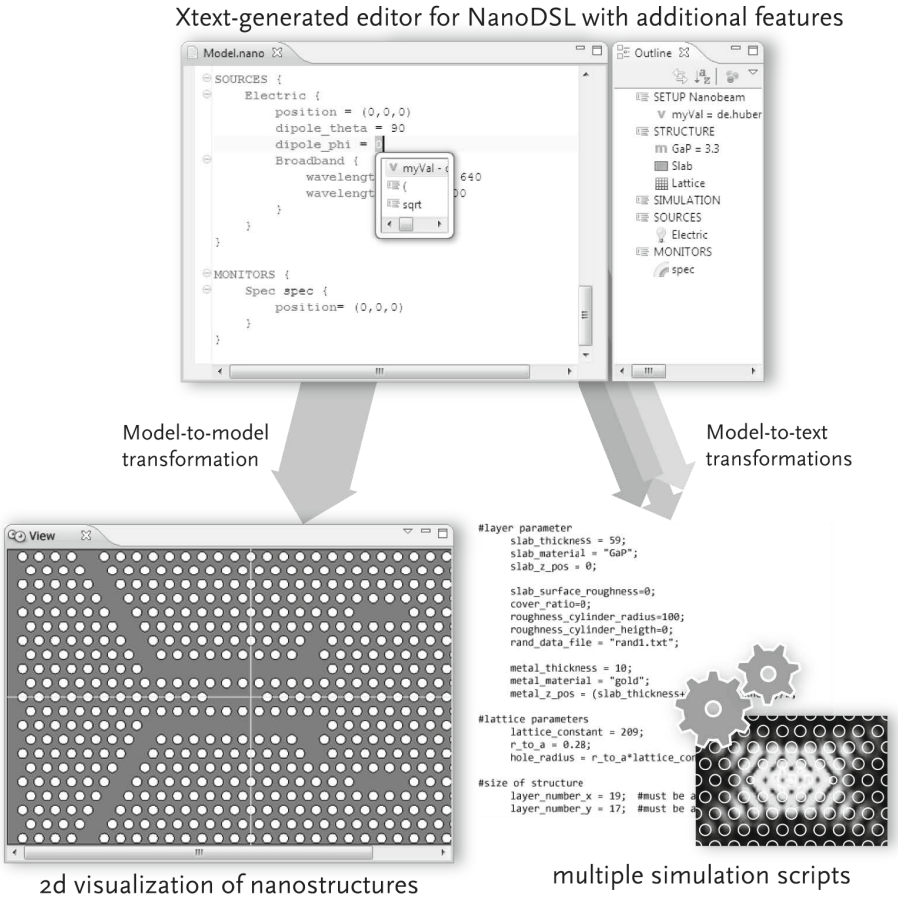


Fig. 3. Implemented parts of the domain-specific workbench

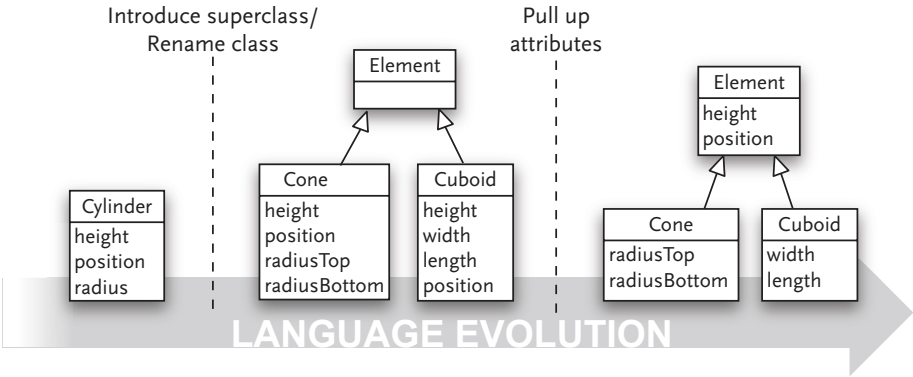


Fig. 4. Example for changes of NanoDSL

4 A Model Transformation-Based Approach for Evolution of Interdependent Artifacts in Language Development

In the following subsections, we present our model transformation based approach to refactoring of software languages. First, we will sketch the traditional, imperative approach that is used to realize refactoring for, e.g., general purpose programming languages like Java (homogeneous artifacts). Secondly, we identify differences between the refactoring of, e.g., Java programs and DSLs. Thirdly, we describe refactoring of software languages as a model-view-synchronization problem. Fourthly, we present *lenses* as one concrete method for model-view-synchronization. Finally, we summarize our approach and identify the components necessary to describe a refactoring within our approach.

4.1 Traditional Imperative Approach

Bäumer, Gamma & Kiezun [12] propose the following three steps to perform a refactoring.

- First, create a *program database* (e.g., an AST) that stores information about declarations and references (independent from a compiler). The database needs to provide a search interface for queries like *What program elements reference a certain method?*. This general step is independent from concrete refactorings.
- Secondly, perform structural program analysis using the previously established database. In this step the *cone of influence* is determined for concrete refactorings: all affected compilation units and program elements are identified and refactoring-specific preconditions are checked.
- Thirdly, the actual changes are performed. Elements of the previously determined cone of influence are changed according to the refactoring.

4.2 Refactoring of Software Languages

It is hard to extend the traditional imperative approach to the refactoring of software languages like DSLs. The existing refactoring capabilities for one type of artifact (homogeneous artifacts, e.g., refactorings for Ecore models) need to be extended to other types of artifacts (heterogeneous artifacts, e.g., Ecore refactorings that also affect related Xtend rules).

There are two specific problems. First, there is no common program database that includes elements for all types of artifacts involved in DSL development. Secondly, there are explicit relations between artifacts of different types and there are also implicit or indirect relations. A code generation rule for example is not only connected to meta-classes it directly references, but also from its super classes and its features. Furthermore, in some cases there are whole artifacts that are implicit. An example is the generated metamodel in Xtext-based DSL development, where the abstract syntax is fully generated from the concrete syntax. Code generator rules are explicitly linked to the generated metamodel, but also indirectly connected to grammar rules.

4.3 Modeling Language-Refactorings as a Model-View-Synchronization Problem

To solve the previously stated problems, we apply the steps of the traditional imperative approach with declarative methods. The information stored in a program database is already contained in the original artifacts and can be extracted through model transformation rules (step 1).

The cone of influence is a view (i.e., an abstraction) on the model that is the sum of all artifacts (i.e., the language description). This *refactoring view* can be described as model transformations between all types of artifacts and the refactoring view (step 2). These transformations filter all elements in all artifacts for those elements that are affected by the refactoring. In that sense, the refactoring view aggregates all refactoring related information scattered in all artifacts into a single view.

The actual manipulation of the model can be described by an in-place model transformation that changes the refactoring view and as model transformations from the refactoring view into all types of artifacts (step 3). Here, the aggregated information about all refactoring related elements is used to change all those elements accordingly.

Pairs of model transformations between artifacts and refactoring view and between refactoring view and artifacts (i.e., forward and backward transformations) can be described as bidirectional model transformations and, thus, the application of a refactoring can be described as a model-view-synchronization problem. Conclusively, each type of refactoring (e.g., a pull down) is described by (1) a metamodel of the refactoring view, (2) one model transformation for altering that view, and (3) bidirectional model transformations between each artifact type and the refactoring view.

4.4 Asymmetric Bidirectional Transformations

We model the refactoring of DSLs as a model-view-update problem. There are many approaches to bidirectional model transformations for solving a model-view-synchronization problem. For refactorings we favor asymmetric bidirectional model transformations (more specifically *lenses*). This specific kind of bidirectional model transformation fits the needs of a model-view-synchronization [14].

Lenses, as introduced by Pierce et al. [15], are asymmetric bidirectional transformations, i.e., one of the two structures that are synchronized has to be an abstraction of the other. This asymmetric approach is inspired by the *view-update problem* known in the database community, where a database view – the abstraction – has to be updated when the database changes and vice versa.

Given a *source* set S of concrete structures and a *view* set V of abstract structures, a lens comprises two functions:

$$\begin{aligned} get &: S \to V \\ put &: V \times S \to S \end{aligned}$$

The forward transformation *get* derives an abstract view structure from a given concrete source structure (e.g., filtering for a cone of influence). The backward transformation *put* takes an updated abstract view structure and the original concrete source structure to yield an updated concrete structure (e.g., propagate refactoring related changes into an artifact). Fig. 5 depicts a lens and its two functions.

Lenses, as presented by Pierce et al., is a combinator-based approach to asymmetric bidirectional transformations, i.e., lenses are composed from other lenses. There are primitive and combinator lenses. Formal properties of lenses can be proved for combinations if the used combinator preserves these properties. The

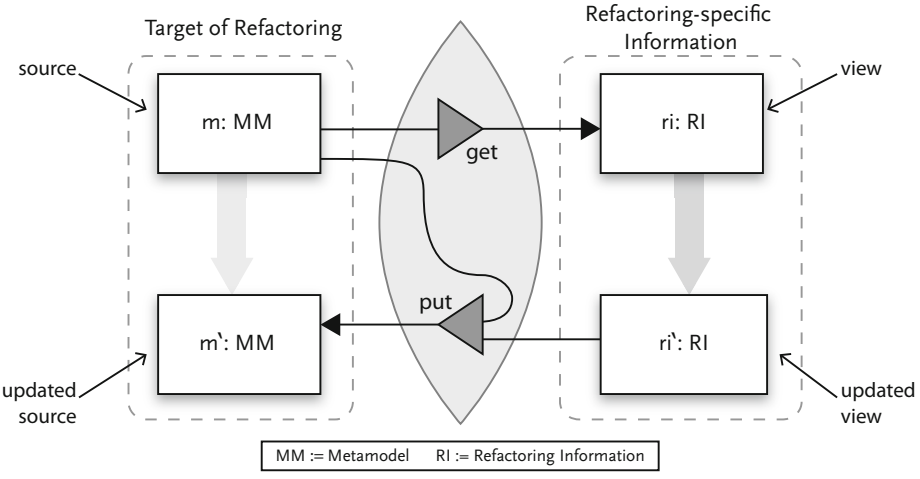


Fig. 5. Asymmetric Bidirectional Model Transformation (based on [13])

lenses framework therefore provides a flexible model transformation technique with strong capabilities for reuse and formal verification. However, the more general notion of a lens (an encapsulated tuple of the two functions) is already useful when modeling a model-view-synchronization problem.

The lenses approach stands in contrast to directly manipulating the source (which would be the naive object-oriented approach), as it describes the synchronization as two side-effect-free functions. However, the lenses approach is only possible when at a given time always only one artifact is changed and immediately synchronized with its interdependent artifacts, i.e., with no concurrent changes. This is always the case in our refactoring scenario.

4.5 Describing Refactorings with Model Transformations

A refactoring type R is described as an x-tuple of one refactoring view metamodel MM_{rv}, a number of asymmetric bidirectional transformations $\leftrightarrows_{sv}$ between the metamodels of all involved artifact types and the refactoring view metamodel, and one unidirectional view-change-transformation $\rightarrow_{vv}$ (view to view) on the refactoring view model.

$$R = \langle MM_{RV}, [\leftrightarrows_{sv}], \rightarrow_{vv} \rangle$$

In principle, this framework is independent of the concrete types of artifacts (unless they are not applicable to the same model transformation method) and independent of the concrete model transformation (unless it does not allow for asymmetric bidirectional transformations). Fig. 6 shows a refactoring type (pull up) based on Ecore and Xtend as artifact types.

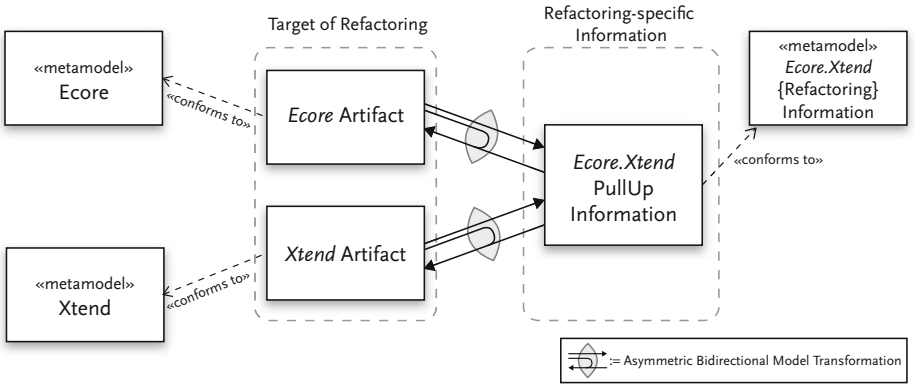


Fig. 6. The Pull-up refactoring involving an Ecore and an Xtext artifact

4.6 Advantages and Disadvantages

The proposed approach provides the following advantages compared to traditional imperative approaches. Firstly, the approach allows for more comprehensible and reusable descriptions of refactorings as a clear set of models and transformations. Refactoring related logic is not scattered across multiple parts of the language description anymore. Secondly, the declarative model transformation based approach clearly separates meta-language tooling (e.g., Xtext, Ecore, Xtend tools) from refactorings. The refactoring description only depends on models and not on tools. Thirdly, the approach provides a history of changes and refactorings as a set of model transformation traces. Finally, the declarative and side-effect-free lenses based approach provides good verification capabilities, especially when implemented in a functional manner.

As a disadvantage, a potentially large number of transformations has to be created for each refactoring – up to two times the number of involved artifacts. Firstly, this number can be halved by using special languages for bidirectional transformations that provide special notations for defining both the forward and the backward transformation at the same time. Secondly, we argue that the refactoring logic is the same as in the traditional imperative approach, it is just structured differently. This different structure can result in code duplication when implemented naively, e.g., when transformations for multiple involved artifacts are very similar. However, by defining reusable building blocks for transformations and by composing transformations from them, it is possible to not only keep code duplication within a refactoring minimal, but also code duplication across refactorings, which is one of the advantages of our approach.

5 Implementation with Model Transformations

The general approach presented so far is independent from concrete modeling technologies or model transformation languages. It can be implemented using special languages for the description of bidirectional transformations or pairs of unidirectional transformations which again can be described with special model transformation languages or GPLs.

In this section we demonstrate how to implement our approach using Java as a GPL and how to integrate it with Eclipse-based modeling technologies. As an example, we use the 'Pull Up Item' refactoring in the Ecore, Xtext, and Xtend based scenario that we motivated in Sect. 3.

5.1 Abstract Refactoring Structure

Listing 1.1 shows the abstract structure of refactorings where three artifacts are affected: A grammar describing a textual concrete syntax, a metamodel describing the abstract syntax, and a generator describing one execution semantics. Thus, a refactoring consists of three asymmetric bidirectional model transformations (i.e., lenses, see List. 1.2) that synchronize the grammar, the model,

and a generator, respectively, with a refactoring-specific refactoring view. In addition to that, an abstract method for implementing the (typically trivial) change on that refactoring view is provided. It takes a second parameter of type `ChangeInfo`, if there are different possibilities how to perform the change.

```
public abstract class
    Refactoring<RefactoringView,ChangeInfo,Grammar,Model,Generator>
    {

   public Lens<Grammar, RefactoringView> grammarLens;
   public Lens<Model, RefactoringView> modelLens;
   public Lens<Generator, RefactoringView> generatorLens;

   public abstract RefactoringView changeView(RefactoringView
       oldView, ChangeInfo info);
}
```

Listing 1.1. Java implementation of the Refactoring structure

```
public interface Lens<Source, View> {
   public View get(Source src, ISelection sel);
   public Source put(Source src, View view, ISelection sel);
}
```

Listing 1.2. A lens interface with parameterizable lens functions

5.2 Implementation of 'Pull Up Item' for Xtext, Xtend and Ecore

Based on this general structure of a refactoring description, the following listings show parts of an exemplary implementation of the 'Pull Up Item' refactoring [2]. First, a refactoring-specific view type is defined which, in this case, holds the attribute that is to be pulled up, the class it originally belongs to (the subclass), and a list of this class' superclasses (List. 1.3). From the list of superclasses one is to be chosen for the attribute to be pulled up to.

```
public class PullUpRefactoringView {
   public EAttribute selectedAttribute;
   public EClass subClass;
   public List<EClass> superClasses;
}
```

Listing 1.3. RefactoringView for 'Pull up Item'

Next (refer to List. 1.4), the concrete refactoring type is defined by extending the abstract refactoring type and by providing appropriate type parameters: Obviously, the view type is the previously defined `PullUpRefatoringView`. The `ChangeInfo` contains the target superclass. As multi-inheritance in principle is allowed in model-driven engineering, we provide a wizard for selecting the target superclass if there is more than one option. Before an instance of this view can

be created, refactoring-specific pre-conditions have to be checked like *Exists at least one superclass?* or *Are there already attributes with the same signature in the selected superclass?*.

The remaining type parameters are specific to the involved technologies: An Xtext resource for the grammar description, `EObject` for the root object of the (meta-) model describing the abstract syntax, and again an Xtext resource for the Xtend-based generator because Xtend is based on Xtext. Apart from providing these type parameters (and, thus, typing the three lenses accordingly) the declaration of the refactoring type only provides a concrete implementation of the `changeView`-method, which here, only changes the attribute in the view so that it belongs to the selected superclass of its originally containing class. Listing 1.4 shows the complete definition of the PullUpRefactoring type (except the trivial, field initializing constructor).

```
1  public class PullUpRefactoring extends
       Refactoring<PullUpRefactoringView, PullUpChangeInfo,
       XtextResource, EObject, XtendResource>{
2
3     public Lens<XtextResource, PullUpRefactoringView> grammarLens;
4     public Lens<EObject, PullUpRefactoringView> modelLens;
5     public Lens<XtendResource, PullUpRefactoringView> generatorLens;
6
7     @Override
8     public PullUpRefactoringView changeView(PullUpRefactoringView
          oldView, PullUpChangeInfo info) {
9       // ..
10      // checking parameter and relevant preconditions
11
12      PullUpRefactoringView newView = oldView;
13      EAttribute changedEAttribute = clone(oldView.getAttribute());
14
15      for (EClass superClass : newView.getSuperClasses()) {
16        if (superClass.getName() ==
              info.selectedSuperClass.getName()) {
17          // Change the container for the attribute
18          superClass.getEStructuralFeatures().add(changedEAttribute);
19      }}
20
21      newView.setAttribute(changedEAttribute);
22      return newView;
23 }}
```

Listing 1.4. The PullUpRefactoring class

Now, the vital parts of the refactoring are the bidirectional model transformations, which are defined separately and are then passed as constructor arguments to the refactoring during instantiation. As we show an implementation without the use of special languages for bidirectional transformations, a total of six transformations have to be provided in this case (three forward and three backward transformations.) For brevity, we only show two selected transformations.

Listing 1.5 shows the forward transformation `get` of the lens synchronizing between the Xtend-based generator and the refactoring view. First, we are extracting the element, which is the target of the refactoring from the text selection `sel`. Afterwards we collect the relevant information – the containing class and its superclasses – and build the refactoring view. To gain this information, we navigate through the containment hierarchy of the resolved element. Finally, the refactoring view is returned.

```
1  public PullUpRefactoringView get(XtextResource src, ISelection sel) {
2     PullUpRefactoringView pullUpRefView = new PullUpRefactoringView();
3
4     EObject elementUnderChange = getElement(sel);
5
6     // Attribute which will be pulled up
7     pullUpRefView.attribute = (EAttribute)elementUnderChange;
8     // Containing class of attribute
9     pullUpRefView.subClass = getContainerOfType(elementUnderChange);
10    // List of possible superclasses
11    pullUpRefView.superClasses = getSuperClasses(elementUnderChange);
12
13    return pullUpRefView;
14 }
```

Listing 1.5. Forward transformation of the generator lens of 'Pull Up Item'

The backward transformation of the lens synchronizing between the Ecore-based (meta-) model and the refactoring view is shown in Listing 1.6. Additionally to the (potentially altered) refactoring view, this transformation takes the original artifact as the `src` argument – here, the original model. A copy of this model is created and the selected attribute is replaced by the one contained in the refactoring view (including the attribute's updated reference to its containing class).

```
1  @Override
2  public EObject put(final EObject src, final PullUpRefactoringView
       view, ISelection sel) {
3     EObject newModel = copy(src); // returns a copy of src
4
5     EAttribute selectedEAttribute = getSelectedAttribute(sel);
6
7     // replace the selected attribute with containment hierarchy
8     newModel.eSet(selectedEAttribute, view.getAttribute());
9
10    return newModel;
11 }
```

Listing 1.6. Backward transformation of the model lens of 'Pull Up Item'

5.3 Alternative Implementation Approaches

In the previous subsections, we demonstrated a pragmatic implementation of our model transformation based approach using Java as a GPL, because it is well-known and eases integration with existing EMF-based technologies. However, the advantages are more apparent, when using special model transformation languages for the implementation. For a Java-like integration of such languages with EMF-based technologies, we showed how to implement a unidirectional rule-based model transformation language as an internal DSL in the *Scala* programming language [16]. Using this language or another unidirectional model transformation language like ATL [17], the forward and backward transformations of refactorings could be described more concisely.

However, in order to use our approach to its full capacity, special languages for describing bidirectional model transformations like *QVT Relations* could be used instead of describing pairs of unidirectional transformations. Unfortunately, QVT Relations suffers from weak tool support and from semantic issues regarding non-bijective relations [18]. This especially affects the presented scenario of constructing and synchronizing an abstracted view. The combinator-based approach of lenses that was presented by Pierce et al. is especially strong in such a scenario. Therefore, we are working on an implementation of such combinator-based lenses that integrate well with EMF-based technologies [14] and work on an implementation of our refactoring approach using these lenses.

Furthermore, an issue that arises with our current implementation approach, is that some logic, e.g., for finding all occurrences of an element, is needed in both the forward and the backward transformation (because it takes the original source as an argument) but is currently not shared, resulting in code duplication. Therefore, we are investigating into splitting the process of building a view into two steps: First, the abstraction step that only collects relevant information and, second, the aggregation step that merges redundant information so that it can be used as a shared view on different artifacts. This way, the abstraction step could be shared by forward and backward transformation.

6 Conclusions and Future Work

We presented a declarative approach to horizontal co-refactorings in language development. Our approach employs asymmetric bidirectional model transformations to extract all information important for a refactoring into a refactoring view first, and then to synchronize all artifacts of a language description with the changes made to that refactoring view.

This approach allows for describing types of refactorings independent from tooling and concrete notation of meta-languages. All that comprises a refactoring is put into its description – which is a clear set of models and transformations – and refactoring related logic is not spread over and mixed with meta-tools. This allows for extending refactorings towards new meta-languages. Sequences of changes can be recorded through traces of model transformations. Different model transformation languages can be used to implement our abstract approach

as long as they allow for asymmetric bidirectional transformations. However, the advantages of our declarative approach can be leveraged in conjunction with special (often more declarative) transformation languages for bidirectional model transformations. Therefore, our approach should benefit from ongoing research in that area [3,14,19].

We showed the principle feasibility of our approach based on the *'Pull up item'* refactoring, which we applied to an Ecore, Xtext, Xtend based DSL. We are working on a full catalog of refactorings in order to better show advantages of our approach in terms of comprehensibility and reuse of transformations across multiple refactorings. Furthermore, we want to provide an implementation that makes use of special languages and frameworks for bidirectional model transformations. Therefore, we are evaluating which of such languages and frameworks work best for our scenario. Finally, we are investigating how our approach can be generalized towards co-adaptations that are not refactorings.

References

1. Favre, J.-M.: Languages evolve too! changing the software time scale. In: Proceedings of the Eighth International Workshop on Principles of Software Evolution (IWPSE 2005), pp. 33–44. IEEE Computer Society (2005)
2. Fowler, M., Beck, K.: Refactoring – improving the design of existing code. Addison-Wesley Professional (1999)
3. Diskin, Z., Xiong, Y., Czarnecki, K.: From State- to Delta-Based Bidirectional Model Transformations – the Asymmetric Case. Journal of Object Technology 10, 6:1–6:25 (2011), http://www.jot.fm/issues/issue_2011_01/article6.pdf
4. Wachsmuth, G.: Metamodel adaptation and model co-adaptation. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 600–624. Springer, Heidelberg (2007)
5. Herrmannsdoerfer, M., Ratiu, D., Wachsmuth, G.: Language evolution in practice –The history of GMF. In: van den Brand, M., Gašević, D., Gray, J. (eds.) SLE 2009. LNCS, vol. 5969, pp. 3–22. Springer, Heidelberg (2010)
6. Strein, D., Kratz, H., Lowe, W.: Cross-language program analysis and refactoring. In: Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006), pp. 207–216. IEEE Computer Society (2006)
7. Favre, J.: Meta-model and model co-evolution within the 3D software space. In: Proceedings of the Interantaional Workshop on Evolution of Large-scale Industrial Software Applications (ELISA), pp. 98–109 (2003), http://plg.math.uwaterloo.ca/~migod/papers/2003/ELISAproceedings.pdf
8. Pizka, M., Jürgens, E.: Tool-supported multi-level language evolution. In: Software and Services Variability Management Workshop, vol. 3, pp. 48–67. Helsinki University of Technology (2007), http://citeseerx.ist.psu.edu/viewdoc/doi=10.1.1.190.9220&rep=rep1&type=pdf
9. Jürgens, E., Pizka, M.: The Language Evolver Lever-Tool Demonstration. Electronic Notes in Theoretical Computer Science 164(2), 55–60 (2006), http://dx.doi.org/10.1016/j.entcs.2006.10.004
10. Wider, A., Schmidt, M., Kühnlenz, F., Fischer, J.: A Model-Driven Workbench for Simulation-Based Development of Optical Nanostructures. In: Proceedings of the 2nd International Conference on Computer Modelling and Simulation (CSSim 2011) (2011) IEEE CD with ISBN 978-80-214-4320-4

11. Barth, M., Kouba, J., Stingl, J., Löchel, B., Benson, O.: Modification of visible spontaneous emission with silicon nitride photonic crystal nanocavities. Optics Express 15(25), 17231–17240 (2007), http://dx.doi.org/10.1364/OE.15.017231
12. Bäumer, D., Gamma, E., Kiezun, A.: Integrating Refactoring Support into a Java Development Tool. In: OOPSLA 2001 Companion. ACM (2001), http://people.csail.mit.edu/akiezun/companion.pdf
13. Foster, J.: Bidirectional Programming Languages. PhD thesis, University of Pennsylvania (2009), http://repository.upenn.edu/cgi/viewcontent.cgi?article=1967&context=cis_reports
14. Wider, A.: Towards Combinators for Bidirectional Model Transformations in Scala. In: Sloane, A., Aßmann, U. (eds.) SLE 2011. LNCS, vol. 6940, pp. 367–377. Springer, Heidelberg (2012)
15. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for Bi-Directional Tree Transformations: A Linguistic Approach to the View Update Problem. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2005), pp. 233–246. ACM (2005)
16. George, L., Wider, A., Scheidgen, M.: Type-Safe Model Transformation Languages as Internal DSLs in Scala. In: Hu, Z., de Lara, J. (eds.) ICMT 2012. LNCS, vol. 7307, pp. 160–175. Springer, Heidelberg (2012)
17. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL – A model transformation tool. Science of Computer Programming 72(1-2), 31–39 (2008)
18. Stevens, P.: Bidirectional model transformations in QVT – Semantic issues and open questions. Software and Systems Modeling 9(1), 7–20 (2010)
19. Sasano, I., Hu, Z., Hidaka, S., Inaba, K., Kato, H., Nakano, K.: Toward Bidirectionalization of ATL with GRoundTram. In: Cabot, J., Visser, E. (eds.) ICMT 2011. LNCS, vol. 6707, pp. 138–151. Springer, Heidelberg (2011)

SDL Real-Time Tasks – Concept, Implementation, and Evaluation

Dennis Christmann, Tobias Braun, and Reinhard Gotzhein

Networked Systems Group
University of Kaiserslautern, Germany
{christma,tbraun,gotzhein}@cs.uni-kl.de

Abstract. *Real-time tasks* are a concept used in real-time systems to structure and schedule execution, in order to handle load situations, and to meet deadlines. In previous work, we have transferred this concept to the Specification and Description Language (SDL), by incorporating the notion of real-time task into SDL's formal syntax and semantics. More specifically, we have defined an SDL real-time task as a set of transition executions, which may span different SDL processes and are ordered by a strict partial order with a least element. In this paper, we extend this concept by the notion of *distributed* real-time task, which may span SDL processes of different SDL systems, thereby supporting tasks executed on several nodes. In addition, we introduce the notion of task types, which support task multiplexing in SDL processes. We then outline our implementation of real-time tasks in our SDL tool chain, consisting of the SDL transpiler ConTraST, the SDL Runtime Environment (SdlRE), and the SDL Environment Framework (SEnF). To evaluate the gain in real-time performance, we have devised an SDL specification of an Adaptive Cruise Controller taken from the automotive domain, and have executed it on an Imote2 hardware platform. The results clearly show that task-based scheduling outperforms ordinary and priority-based scheduling in terms of processing delays and reaction times to critical events.

1 Introduction

The Specification and Description Language (SDL) [1] has been devised as a formal design language for distributed systems. Yet, due to its notion of time (`now`) and its timer mechanism, it also provides expressiveness to specify certain aspects of real-time systems. To broaden this expressiveness, we have proposed, defined, implemented, and evaluated several language extensions, in particular SDL real-time signals [2] and SDL process priorities [3]. These extensions have proven valuable to enhance the predictability of networked control systems, which we have developed in a model-driven way with SDL as design language [4].

To further enhance the real-time capabilities of SDL, we have considered the concept of *real-time task* (or *task* for short), which is used in real-time systems to structure and schedule executions, in order to handle load situations and to meet deadlines. Tasks are code unit executions, and may be initiated

F. Khendek et al. (Eds.): SDL 2013, LNCS 7916, pp. 239–257, 2013.

dynamically when a significant change of state occurs (event-triggered) or at determined points in time (time-triggered). In the context of SDL, these code unit executions may be structured into several ordered SDL transition executions associated with one or more SDL processes. After some consideration, we came to the conclusion that our previous extensions, i.e., SDL real-time signals and SDL process priorities, were not sufficient to express real-time tasks in SDL. Therefore, we have devised further language extensions, which were necessary to enable real-time tasks in SDL [5]. Thus, we have established the notion of real-time task[1] in SDL's formal syntax and semantics.

In this paper, we continue our previous work conceptually and, in particular, by presenting the implementation and evaluation of SDL real-time tasks. Conceptually, we introduce the notion of *distributed* real-time task, which may span several SDL processes of *several* SDL systems (Sect. 2). In practical situations, this means that several nodes may be involved in the completion of a given real-time task. In addition, we propose *task types*, which can be used to naturally specify task multiplexing in SDL processes.

The focus of this paper, however, is on the implementation and evaluation of SDL real-time tasks and task scheduling. In Sect. 3, we outline the implementation in our SDL tool chain [4], which consists of the SDL transpiler ConTRaST, the SDL Runtime Environment SdlRE, and the SDL Environment Framework SEnF. In Sect. 4, we present extensive experimental results showing the gain of SDL real-time tasks and task scheduling w.r.t. the predictability of reaction times compared to existing scheduling approaches. To run these experiments, we have specified an Adaptive Cruise Controller (ACC) taken from the automative domain with SDL, and have executed it on an embedded hardware platform. From the experiments, it is obvious that real-time task scheduling outperforms existing scheduling strategies, in particular with increasing system load, and substantially improves the predictability of reaction times. The paper is completed by a survey of related work (Sect. 5) and conclusions (Sect. 6).

2 Distributed SDL Real-Time Task

In this section, we extend our previous work [5] by introducing the concept of distributed real-time tasks in SDL (see Sect. 2.1). With the extension, SDL tasks can not only be used to group functionality-related behavior of a single SDL system, but also to identify and prioritize behavior spanning several network nodes. To incorporate distributed real-time tasks in SDL, several language extensions are presented in Sect. 2.2.

2.1 Concept of SDL Real-Time Task

The formal definition of real-time task is based on a set of transition executions and a strict partial order with a least element, i.e., each real-time task starts

[1] Not to be confused with the existing notion of task in SDL, which is a sequence of statements.

with a single transition execution. Real-time tasks are dynamic in the sense that the set of transition executions is determined at run-time and may depend on the internal state of the system, that is, for instance, the current time or states of SDL processes. They terminate after all transition executions are finished. Let N be the set of network nodes. Then, a real-time task is defined as follows.

Definition 1. *A **real-time task** τ is a tuple (id_τ, $T_e(\tau)$, f_{prio}, f_{node}, $<_{eo}$), where id_τ is a globally unique task id, $T_e(\tau)$ is the set of transition executions, $f_{prio} : T_e(\tau) \to \mathbb{N}$ is a function assigning a priority to each transition execution, $f_{node} : T_e(\tau) \to N$ is a function to allocate each transition execution to a network node, and $<_{eo} \subsetneq T_e(\tau) \times T_e(\tau)$ is an execution order on $T_e(\tau)$ with following properties:*

- *$<_{eo}$ is a strict partial order, i.e., $<_{eo}$ is irreflexive, transitive, and antisymmetric*
- *$\exists t_e \in T_e(\tau).\forall t'_e \in T_e(\tau).(t'_e \neq t_e \Rightarrow t_e <_{eo} t'_e)$, i.e., there is a least element defining the starting point of the task, which is the first transition execution.*

We note that the definition allows concurrent transition executions within a real-time task, if they are not ordered by $<_{eo}$. A real-time task may be non-terminating, if its set of transition executions is infinite. Thereby, a real-time tasks τ may consist of cyclic executions of transitions, since executing the same transition multiple times results in different transition executions, i.e., different entries in $T_e(\tau)$. An example for such a task is the periodical calculation of control values. A real-time task itself is non-recurring, i.e., it is executed only once.

The definition of real-time tasks so far covers node-local and node-spanning tasks. Based on the general definition, we define distributed tasks as follows.

Definition 2. *A **distributed SDL real-time task** τ_{dist} is a real-time task, for which the image of f_{node} contains at least two distinct elements:*

$$\exists t_1, t_2 \in T_e(\tau_{dist}) : f_{node}(t_1) \neq f_{node}(t_2).$$

To fulfill this definition, there must be at least two SDL systems deployed on different network nodes, each running at least one transition execution of the task. Thereby, distributed behavior, such as an Request-to-Send/Clear-to-Send (RTS/CTS) handshake or a distributed leader election, can be expressed as a single task.

The definition of distributed SDL real-time task does not model communication explicitly. Instead, consecutive transition executions of different nodes are ordered by the execution order only, and the required synchronization and value passing is left to the environment of the SDL system. A distributed SDL real-time task may run on several nodes in parallel, if there is suitable concurrency according to the execution order $<_{eo}$. Otherwise, the transitions of the distributed SDL real-time task are processed successively by the nodes.

The distinction between SDL transitions as code units and transition executions as execution units results in a very generic definition of real-time task. Some properties of this definition are:

- *Loose temporal ordering*: The definition does not make a statement on the time between transition executions.
- *Flexible activation paradigm*: Every transition execution can be event- as well as time-triggered. In particular, a real-time task can be temporarily suspended or wait for the activation of the next transition execution trigger.
- *Transition repetitions*: The same transition can be executed by one task several times, particularly with different priorities.
- *Priority-independent transition definitions*: A transition as code unit has no (static) priority assigned. Instead, the priority is (dynamically) associated with its execution, thereby allowing the same transition to be executed with different priorities.
- *Transition sharing*: Several real-time tasks may execute the same transition.

Though a real-time task is nonrecurring, it usually describes actions to execute a recurring system task like the response to a specific event that is observed by the environment of the node. To enable the association of a real-time task with the system task it fulfills, the notion of *task type* is introduced. During a transition execution, the information on the task type is, for instance, helpful for task multiplexing within the system or for changing the priorities of future transition executions. The relation between SDL task and task type has an analogy in object-oriented programming languages, because a real-time task τ (object) states the execution (instantiation) of a task type (class), and multiple executions of a task type result in multiple real-time tasks with distinct task (object) identifiers id_τ. Formally, the relation between real-time tasks and task types is as follows: Assuming τ is a real-time task and Γ is the set of all task types. Then, there is a function f_{type} with $f_{type}(\tau) \in \Gamma$ returning the task type of the real-time task.

2.2 Language Extensions to SDL

In [5], real-time tasks have been incorporated in SDL's syntax and semantics. By dynamically associating transition executions with task attributes, consisting of task id id_τ and a priority, a transition runs in the context of the real-time task τ. To accommodate task types, task attributes are now extended by $f_{type}(\tau)$. The task attributes are transported by SDL signals (so-called task signals), which are used to trigger task execution. Consuming a task signal transfers the task attribute to the triggered transition execution. The priorities given by the task attributes have implications on the transition selection order, such that task signals are consumed according to their priority and have additionally preference over plain SDL signals.[2] Thereby, transition executions of time-critical real-time tasks take precedence over all other transition executions.

[2] There is an exception for this rule, if the signal with highest priority is saved in the current process state.

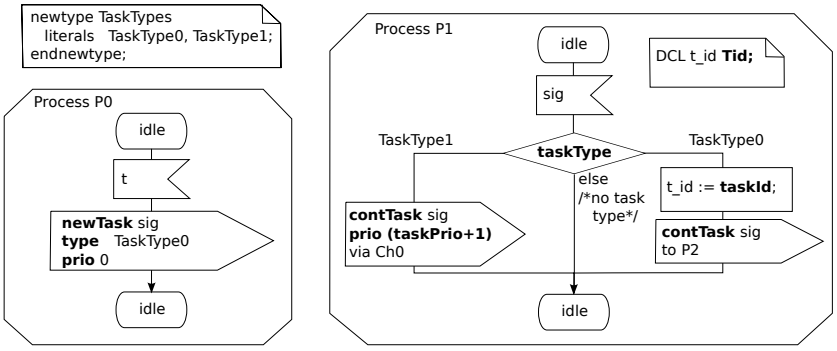


Fig. 1. Example showing the use of real-time tasks in SDL. Bold characters state new keywords/operators

Due to the consideration of priorities, transitions may be executed in a different order compared to standard SDL. Thus, existing tools – like tools performing reachability analysis to find deadlocks or implicit consumptions – must be extended to consider priorities. Since transition priorities are included in each signal's task attributes, all required information is, however, available in the SDL system and analysis of the system specification is still possible. This is even an additional benefit compared to many implementation methods introducing priorities in a separate implementation phase, because with such approaches, priorities are not available for system analysis on design level.

To control the execution of real-time tasks in SDL specifications, [5] presents several syntactical extensions, in particular, regarding task creation and forking (continuation of an existing task). Additionally, a new data type to store task ids (**Tid**) has been introduced together with a function returning the task id of the current task. To support task types and relative changes of priorities, the syntax has been extended with functions returning type and priority of the current task.

The example in Fig. 1 presents the use of real-time tasks in SDL: In process `P0`, a new task is created by the output of signal `sig`. This new real-time task is scheduled with task priority 0 and is of type `TaskType0`. In `P1`, task execution starts by consuming the task signal and by identifying the task type by means of the **taskType** operator. Because the task type is `TaskType0`, the right branch is taken, i.e., the transition stores the task id in the variable *t_id* and triggers the next transition execution of this task by sending `sig` to process `P2` (not shown). Because no priority is provided, the task is continued with priority 0. The figure only shows excerpts of SDL processes highlighting language elements that have been extended to enable the control and processing of real-time tasks in SDL. In a real scenario, there are usually further application-related actions in the transitions' bodies (see also Sect. 4.1).

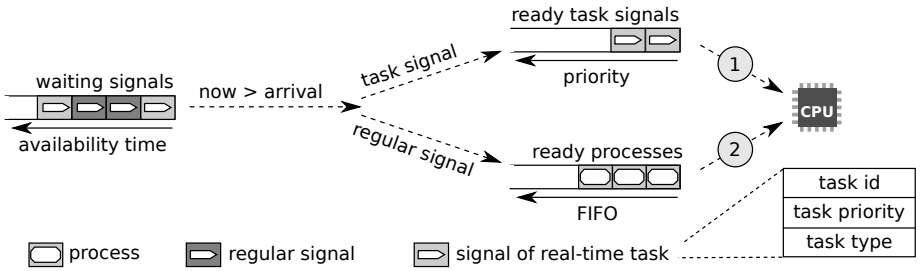


Fig. 2. Schematic outline of the task scheduler implementation

3 Implementation of Real-Time Tasks

The concept of real-time tasks has been implemented in our SDL tool chain, which supports an embedded ARM platform, Linux/PC, and various simulators. The tool chain consists of three main components: The code generator ConTraST, the SDL Runtime Environment (SdlRE), and the SDL Environment Framework (SEnF) providing interfaces and drivers of the SDL environment. To support real-time tasks, changes to all components were necessary.

Though we would prefer the extension of SDL's concrete syntax to control real-time tasks (see Sect. 2), our implementation is based on annotations, thereby allowing the re-utilization of the graphical editor and analyzer of IBM's Rational SDL suite [6]. When generating C++ code, ConTraST analyzes the real-time task annotations in SDL/PR and generates relevant C++ instructions.

To support and schedule real-time tasks during system runtime, SdlRE has been extended by task signals and a non-preemptive scheduler realizing the task scheduling strategy (short: $\text{Priorities}_{tasks}$). Additionally, SdlRE provides an implementation of the **Tid** datatype and an interface to access the task id, priority, and task type of the transition that is currently executed.

Task signals are implemented by extending the existing SDL signal class with task attributes, i.e., task id, priority of the triggered transition execution, and task type. Further information of the real-time task, such as the node executing the transition, is implicitly available and not stored explicitly.

Different to Sect. 2, where task priorities affect the transition execution order of SDL processes[3] only locally, $\text{Priorities}_{tasks}$ enforces priorities system-wide. A schematic overview of $\text{Priorities}_{tasks}$ is presented in Fig. 2. In total, the scheduler operates on three global queues: A queue holding signals with future arrival times (e.g., timers), a queue with task signals sorted by task priorities, and a queue with runnable processes. When searching for the next transition to be executed,

[3] Due to their background of Abstract State Machines, the dynamic semantics of SDL-2000 is based on different types of agents [7]. Thus, to be precise, we would have to use the notion of agents when referring to the execution of an SDL system. Nevertheless, we use the term SDL process in the rest of the paper, because all schedulers of SdlRE affect the scheduling of agents that evolve from SDL processes.

the process holding the first consumable signal in the queue of task signals runs to fire the corresponding transition. If there is no such signal, the first process of the process queue is dispatched to execute one transition that is either a transition consuming a regular signal or a continuous signal.

Following our annotation-based approach, the runtime environment can be configured to use the task scheduling strategy $\text{Priorities}_{tasks}$ by annotations in the head symbol of the system. Besides $\text{Priorities}_{tasks}$, our tool chain supports the following non-preemptive strategies:

- *Signal-based First-Come-First-Served* strategy (short: $\text{FCFS}_{signals}$)
  The transition execution order of $\text{FCFS}_{signals}$ is determined by the arrival times of the triggering signals. For this purpose, a global First-In-First-Out (FIFO) queue of SDL signals is maintained. When searching for the next transition to be executed, the first consumable signal in the queue is taken.
- *Process-based First-Come-First-Served* strategy (short: $\text{FCFS}_{process}$)
  $\text{FCFS}_{process}$ is also based on a FIFO queue, but, different to $\text{FCFS}_{signals}$, the queue is filled with processes. Thus, a process with several signals in its inport, is scheduled only once. The process at the front of the FIFO queue is executed as long as it has firable transitions, thereby reducing the overhead of the scheduler compared to $\text{FCFS}_{signals}$.
- *Process Priority Scheduling* (short: $\text{Priorities}_{process}$)
  In [3], $\text{Priorities}_{process}$ has been introduced in order to privilege time-critical SDL processes. Static priorities are assigned to processes in the SDL specification, where lower values represent higher priorities. The scheduling strategy works on a queue of executable processes, which is sorted by their priorities.

Due to the non-preemptive character of all strategies, there is in general a non-zero waiting delay for all schedulers, increasing in particular reaction times in systems with long-running transitions. However, independent of the scheduling strategy, such transitions should already be avoided by design rules, because they always delay other transition executions of the same SDL process.

To support distributed SDL real-time tasks, the environment (implemented by SEnF) has been extended, such that task attributes are attached to outgoing data, e.g., to CAN messages, before leaving the node. By extracting task attributes from received data and adding them to generated SDL signals, the environment continues existing real-time tasks on the local node. Thereby, the SDL runtime environment can treat incoming signals according to their priority and task type, though the continuation of the real-time task is transparent to the system.

A special element of a task attribute is the task id. To guarantee its uniqueness also in case of distributed real-time tasks, further measures became necessary. In our implementation, the uniqueness is ensured by composing task ids of the node's id and a locally unique identifier, which is incremented each time a new task is generated.

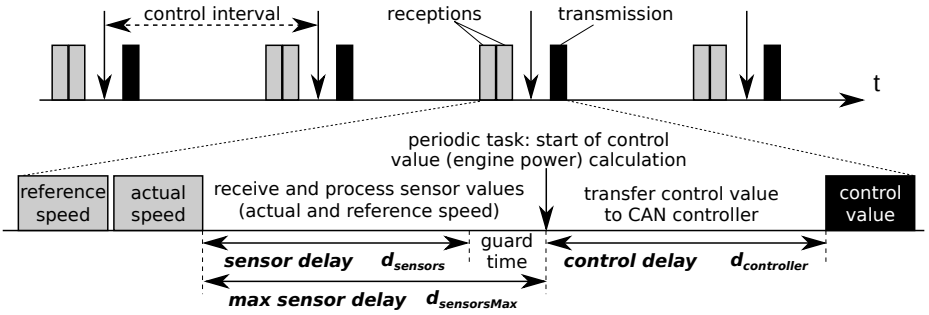


Fig. 3. Message schedule of the node hosting the PID controller in the ACC scenario

4 Evaluation of Real-Time Tasks in a Control System

In this section, we present experimental results of our SDL task implementation. To assess its impact, we compare the task scheduling strategy with standard schedulers of SDL implementations as well as an SDL process priority scheduler. The scenario evaluates a network node that is connected to a Controller Area Network (CAN) bus and hosts an Adaptive Cruise Control (ACC), a realistic scenario from the automotive domain. An ACC is an enhanced cruise control system focused on retaining a reference speed against disturbance variables such as the current gradient or aerodynamic resistance. In contrast to a simple cruise control, a radar sensor is used to detect the distance to obstacles in front of the car. Depending on the speed of and distance to the obstacles, the reference speed is adjusted to keep a minimal safety distance or an emergency braking is initiated. Our realization of an ACC uses a Proportional-Integral-Derivative (PID) controller to minimize the difference between desired and actual speed.

An abstract schedule of the ACC is shown in Fig. 3. The ACC periodically (every 20 ms) calculates new control quantities, which are sent via CAN bus to the engine control unit. The duration to calculate and transfer the control value to the CAN controller is given with $d_{controller}$ in the figure. For correct operation of the controller, control values must be calculated and transferred on schedule with low delay, and sensor values of the reference and actual speed must arrive at the PID controller on time, taking the processing delay of the system into account. In Fig. 3, the sensor delay in the system, i.e., the duration between reception of the last sensor value at the CAN controller and the updating of the values at the PID controller process, is denoted by $d_{sensors}$. Because this delay may vary, a maximal sensor delay $d_{sensorsMax}$ has to be considered in the schedule. The best quality of control is achieved, if the sensor values are as new as possible, i.e., if the processing delay of the node generating the sensor value, the communication delay, and $d_{sensors}$ is small and almost constant, allowing a small $d_{sensorsMax}$ before the periodic control task. In addition to the periodic speed values, the node also receives sporadic radar messages that are used to keep an adequate distance to other objects by correcting the controlled speed

and/or by enforcing to brake. To test the system under different workloads, additional sporadic load messages are sent to the system.

Since the scenario is based on a networked system, a comprehensive analysis must consider delays that are introduced by all nodes as well as the network itself. Because the system designer must consider the worst case when planning the global schedule, all delays must not only be low but also free of large jitter. This, in particular, demands high requirements to scheduling decisions in cases of secondary system load, which must be deferred on behalf of relevant system tasks. In a first step, this requires the assessment of delays at each single network node. This evaluation focuses on the impact of SDL schedulers on the behavior of the node hosting the PID controller.

4.1 Evaluation Setup

Hardware. To obtain reliable and reproducible results, all experiments ran on an Imote2 node, an embedded hardware platform that can be linked to various peripherals and communication technologies. E.g., in [8], a FlexRay [9] communication controller is connected to the Imote2 via Serial Peripheral Interface (SPI). The Imote2 is equipped with 256 kB SRAM, 32 MB SDRAM, and 32 MB flash ROM. Its processor is based on an ARM architecture providing up to 416 Mhz. Due to energy aspects, the processor frequency was fixed to 104 Mhz in all experiments. Since our implementation on the Imote2 is a bare implementation (without further operating system), SEnF and SdlRE have full control over the system's execution and interrupts. Therefore, all measured times can be attributed to the execution of the SDL system and its runtime environment.

Because the experiment's objective is not the evaluation of the communication technology, we did not use a real CAN bus. Instead, we simulated all CAN events taking the minimal interarrival time of CAN messages into account. As a side effect, this approach avoids distortion of results due to communication errors. To additionally avoid faulty measurements, results of experimental runs were stored in the local memory of the node and transferred to a PC via UART after the end of the run. Thus, the measurement overhead is minimized and uniform for all evaluated scheduling strategies.

System Under Test. The SDL system used in the evaluations is shown in Fig. 4 and consists of four blocks. The `CAN` block is the interface to the environment and contains two processes. The `CANMac` process converts between CAN identifiers and internal event expressions. `ConcatCoder`, on the other hand, encodes and decodes the data of CAN messages into SDL types. On top of the `CAN` block are two blocks (`Speed` and `Distance`) that are part of the cruise control. The third block `Load` processes background load that is stimulated by messages from the environment. I.e., load messages are forwarded by `CANMac` and `ConcatCoder` before arriving at the `Load` block, in which they trigger further transition executions. Because the origin of load is in the system's environment, the generation of additional load is independent of the SDL system. By changing the average frequency of load messages, different load situations are emulated.

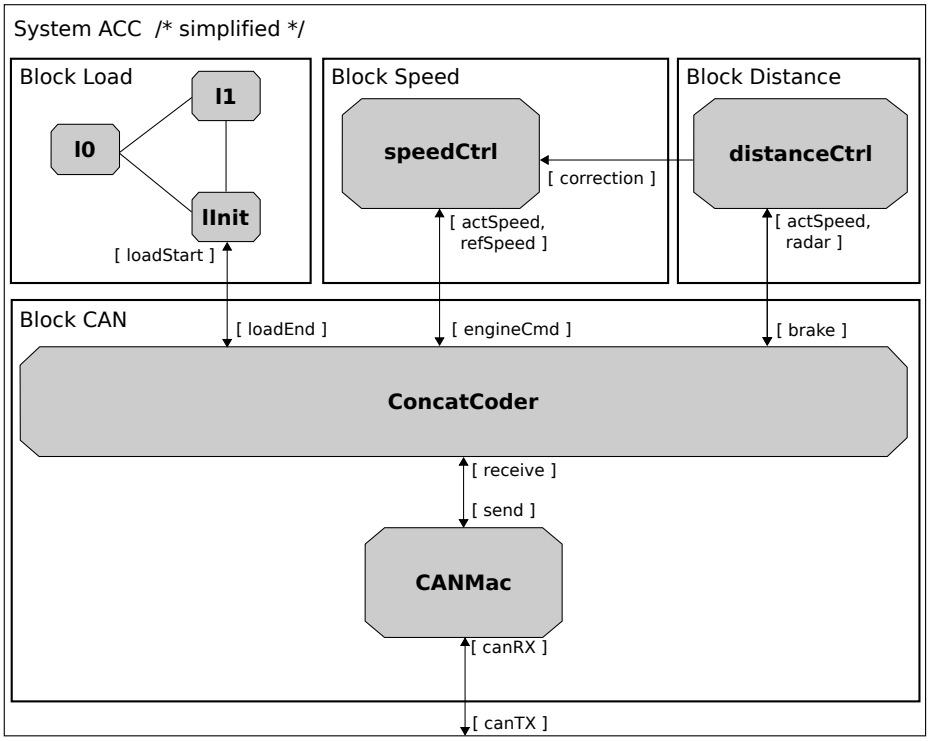


Fig. 4. SDL specification of the evaluated Adaptive Cruise Control system

In the evaluations, the system receives four different types of CAN messages, which are sent as `canRX` signals by the environment to process `CANMac`. After forwarding them through the `CAN` block, they are delivered to their responsible SDL processes. In the case of task scheduling, task types can be used in `ConcatCoder` to determine the target process of a signal. The CAN messages with the actual and reference speed are sent to the processes `speedCtrl` and `distanceCtrl`, in which they are received as SDL signals `actSpeed` or `refSpeed`. On the other hand, the radar and load messages are delivered as SDL signals `radar` or `loadStart` to process `distanceCtrl` and `lInit` respectively. CAN messages sent by the system are received by the SDL environment as `canTX` signals. They are either engine and brake control values, or load information. The engine control values are periodically calculated by `speedCtrl` and initially sent as SDL signal `engineCmd`. Brake control values and load information are generated reactively as responses to `radar` or `loadStart` signals and have their origins in `distanceCtrl` and `lInit`. `distanceCtrl` also creates a `correction` signal, which is considered by `speedCtrl` to calculate the engine control values.

The presented SDL system is executed with four different scheduling strategies (see Sect. 3). The priorities used in experiments with $\text{Priorities}_{process}$ are

given in Tab. 1. They are assigned such that the environment obtains highest priority and processes in the `Load` block have lowest priority. For the task scheduling strategy $\text{Priorities}_{tasks}$, Tab. 2 summarizes the task types of the system, their priorities, and the affected signals. Tasks with sources or destinations in italics are distributed tasks and include communication via CAN bus. By deriving task types and task ids from received CAN messages, the environment continues existing tasks in the evaluated system, considering their privileges as well. On the other hand, before transmitting CAN messages, task attributes are appended to the messages.

Table 1. Process priorities used by $\text{Priorities}_{process}$

processes	priority
CANMac, ConcatCoder	3
speedCtrl	2
distanceCtrl	1
lInit, l0, l1	4
environment	0

The control of real-time tasks in the ACC system is illustrated by means of an exemplary excerpt of process `speedCtrl` in Fig. 5. The figure shows the start transition and four transitions that are executed in the context of real-time tasks. As discussed in Sect. 3, the control of real-time tasks is specified by annotations to be compatible with the graphical tool and analyzer of IBM's Rational SDL suite [6]. In the process, a periodical real-time task computing new engine control values is created with priority 3 in the start transition. This task is processed by executing the transition consuming `controlTimer`. In this transition, a further real-time task is created propagating the new engine control value. The calculation of engine control values takes the reference speed, the current speed, and a correction value given by the distance controller into account. Each of these values is received in a separated transition. Though the executions of the transitions receiving these values are part of a corresponding real-time task, no task control actions are specified, because the real-time tasks end after receiving the values.

Table 2. Task types of the system. Sources and destinations in italics state processes on other nodes.

task type	source	destination	priority	task signals
reference speed	*refSpeedInput*	speedCtrl	4	canRX, receive, refSpeed
actual speed	*actSpeedSensor*	ConcatCoder	4	canRX, receive
	ConcatCoder	speedCtrl	4	actSpeed
	ConcatCoder	distanceCtrl	5	actSpeed
speed control value	speedCtrl	speedCtrl	3	controlTimer
engine regulation	speedCtrl	*engineCtrl*	3	engineCmd, send, canTX
radar	*radarSensor*	distanceCtrl	2	canRX, receive, radar
collision avoidance	distanceCtrl	*brake*	1	brake, send, canTX
	distanceCtrl	speedCtrl	4	correction
load	*loadSimulator*	*loadSimulator*	8	canRX, receive, loadStart loadEnd, send, canTX

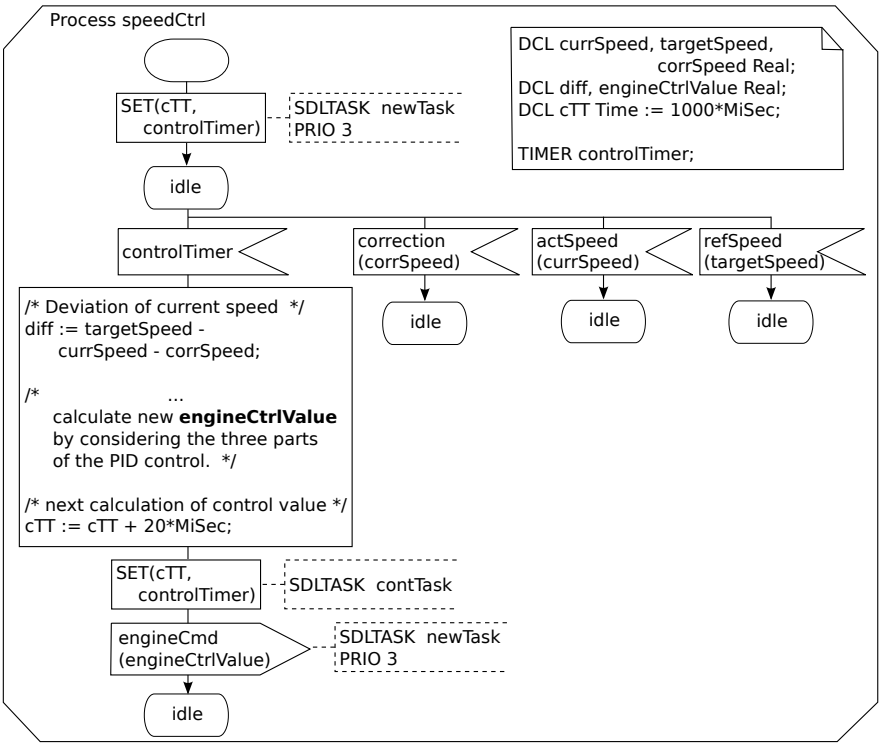
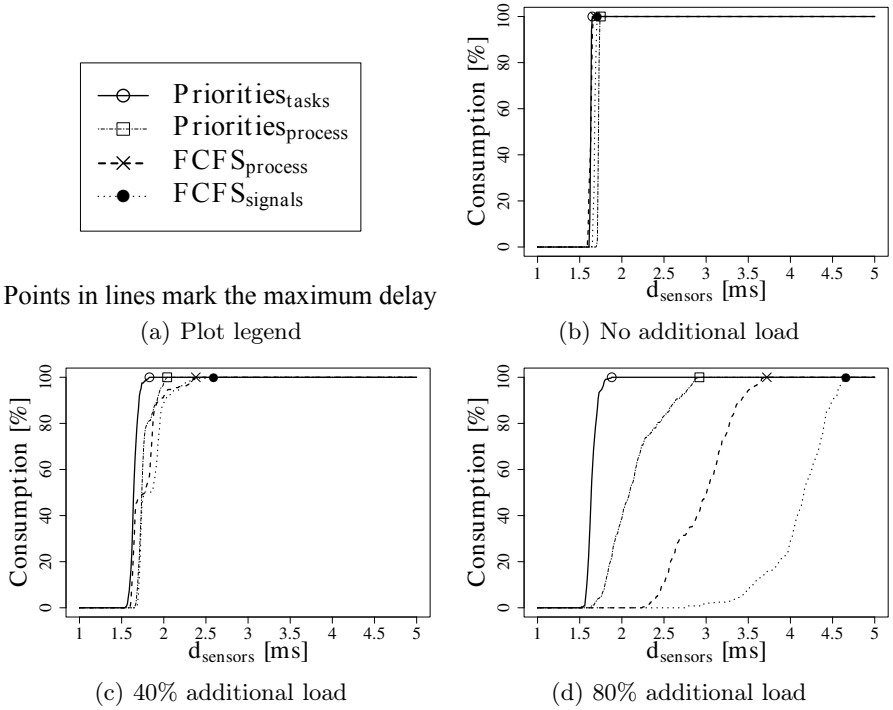


Fig. 5. Excerpt of `speedCtrl` showing the usage of real-time tasks exemplarily

4.2 Evaluation Results

In two series of experiments, the evaluations focus on three delays: Sensor delay $d_{sensors}$, control delay $d_{controller}$, and reaction delay $d_{reaction}$ to radar messages. The sensor and control delays are evaluated in the first series consisting of 125 runs for each scheduling strategy. In each run of this series, in which no radar messages are used, 200 sensor values (reference and actual speed) are received by the system and 100 new control values are calculated and sent. The second series comprises 200 runs and includes additionally 50 radar messages per run that are sent to the system sporadically with a minimal interarrival time of 35 ms.

The runs of each series are divided into 25 different load situations, ranging from no additional load up to approximately 80% additional load. Since the regular system load is about 15%, the heaviest load situation takes the system almost to its limits. The load is processed by the processes within the `Load` block after the reception of special sporadic CAN messages and regulated by changing the average frequency of these messages.



(a) Plot legend (b) No additional load

(c) 40% additional load (d) 80% additional load

Fig. 6. Ratio of sensor value consumptions at `speedCtrl` as a function of the sensor delay $d_{sensors}$ in three different load situations

Accuracy of Sensor Values. To enable correct operation of the PID controller, delays between taking the sensor values and running the control algorithm must be as low and as constant as possible. In particular, this implies high requirements on the processing delay $d_{sensors}$ at the controller node.

In Fig. 6, the percentage of consumed `actSpeed` and `refSpeed` signals is plotted against $d_{sensors}$ for three different amounts of load. The lines in each plot show how many sensor values have been received by the `speedCtrl` process after a given delay $d_{sensors}$. On each line, there is a point marking the maximal delay, i.e., the time after which the latest sensor value was updated in `speedCtrl`.

In case of no additional load (Fig. 6(b)), only sensor values are sent to the system and all schedulers perform almost similar. In detail, the *best* scheduler (Priorities$_{tasks}$) delivers the latest sensor signal after 1.67 ms and the *worst* scheduler (Priorities$_{process}$) requires 1.76 ms. These small differences are basically due to two reasons: First, `ConcatCoder` forwards `actSpeed`, one of the sensor signals, to `distanceCtrl` and to `speedCtrl`, thereby introducing a serialization delay. The second reason is due to different overhead of the scheduling algorithms.

However, in situations with load, the sensor delays increase and the differences between the scheduling strategies become observable. In Fig. 6(c), about

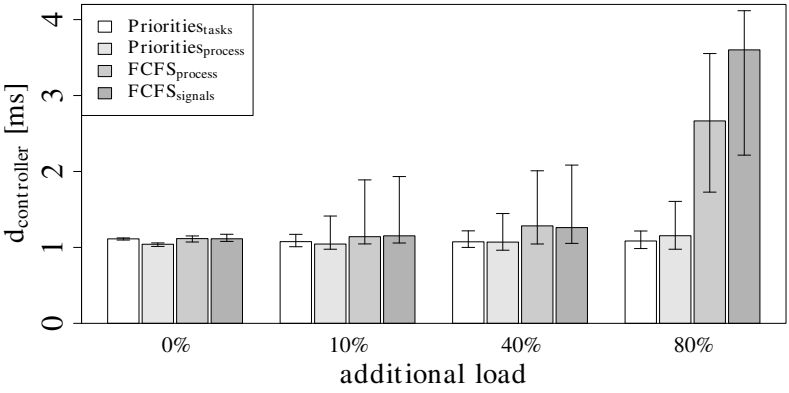


Fig. 7. Delay of calculating the control value and transfer to the SDL environment

40% additional load is added to the system, resulting in a maximal sensor delay of 1.83 ms for the best scheduler Priorities$_{tasks}$ and 2.59 ms for the worst scheduler FCFS$_{signals}$. Here, the increased delay with Priorities$_{tasks}$ is due to its lack of preemption, i.e., before the SDL environment is executed to transfer an SDL signal with sensor values into the system, the running transition must finish. A second reason is attributed to software and hardware caches performing more replacements in case of load. But all other strategies additionally suffer from an inadequate transition execution order. Though the second best strategy Priorities$_{process}$ benefits from the rejection of processes in the `Load` block (maximal delay 2.13 ms), sensor delays are increased due to the execution of transitions in `ConcatCoder` and `CANMac` that are triggered by signals belonging to the background load.

In the high load situation (Fig. 6(d)), the differences become even larger and only Priorities$_{tasks}$ is almost insusceptible against the load. As result, the maximal sensor delay with task scheduling is 41% lower than with the next best scheduling strategy Priorities$_{process}$. Another big advantage is the low sensor delay jitter with Priorities$_{tasks}$ that is only 370 μs. In contrast, the second best scheduling strategies suffers from a jitter of 1560 μs.

These results clearly show that a schedule as shown in Fig. 3 can be realized very accurately with task scheduling. All other scheduling strategies require a more pessimistic value for $d_{sensorsMax}$, thereby decreasing the quality of control.

Control Delay. This section assesses the four scheduling strategies w.r.t. control delay, because the best quality of control is achieved if the periodical computation of new control values is on time and if the new control values are transferred to the actuators rapidly.

Figure 7 depicts the measured control delays in terms of a bar diagram. The plot shows average, minimum, and maximum delays for each scheduling strategy

in four different load situations. The four bars in the left part of Fig. 7 present the results in case of no load. Similar to the sensor delays, control delays differ only slightly in this case. However, when load is added to the system, task scheduling again outperforms the other scheduling strategies.

If the additional system load is about 10%, the average delay remains almost unchanged, but the maximal delay increases for all scheduling strategies. However, the increase with $\text{Priorities}_{tasks}$ is much lower than with the other scheduling strategies. In detail, the maximal control delay with $\text{Priorities}_{tasks}$ is about 240 μs less than with the second best scheduling strategy $\text{Priorities}_{process}$. This difference is basically due the shared transitions in `ConcatCoder` and `CANMac` for which $\text{Priorities}_{process}$ can not distinguish between load and control signals. If the number of signals or shared processes would be higher, reaction times with $\text{Priorities}_{process}$ would even get worse.

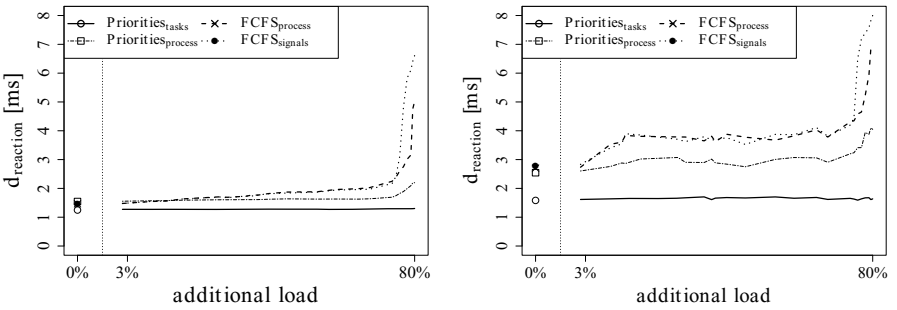
With increasing system load, both FCFS strategies suffer more and more from an inadequate transition execution order, whereas the priority-based strategies are less prone to the load. Thus, comparing situations with 10% and 80% additional load, there are only small differences of control delay with $\text{Priorities}_{tasks}$ as well as $\text{Priorities}_{process}$. Maximum and average control delays with $\text{FCFS}_{signals}$ and $\text{FCFS}_{process}$, however, are more than 2.5 times higher than with $\text{Priorities}_{tasks}$ in case of 80% additional load, thereby demonstrating that fair strategies are not adequate if predictability is required.

Similar to the sensor delays, task scheduling does not only achieve a lower control delay but is also attended by a lower jitter, thereby increasing predictability significantly and resulting in a better quality of cruise control.

Reaction Times to Time-Critical Events. Though the fair scheduling strategies are prone to load, reading sensors and calculating control values are periodic events, thereby allowing to determine an off-line event schedule as shown in Fig. 3 by taking the maximal delays for each scheduling strategy into account. However, with sporadic events, such a schedule is often not possible or requires very pessimistic assumptions on the events' interval. In the following scenario, we add sporadic time-critical radar messages to the system that require fast reactions and measure the reaction times between the reception of the radar messages and the sending of brake commands for each scheduling strategy.

The average and maximal reaction times are presented in Fig. 8, each with 25 different load situations. In case of no load, the average reaction times are almost equal. However, increasing load results in increased average reaction times with $\text{FCFS}_{signals}$, $\text{FCFS}_{process}$, and $\text{Priorities}_{process}$, whereas the reaction times with $\text{Priorities}_{tasks}$ stay constant. For instance, the average delay with $\text{Priorities}_{tasks}$ is only 60% of the average delay with the next best scheduler $\text{Priorities}_{process}$ at 80% additional load.

The maximal delay already differs without load (see Fig. 8(b)), because only task scheduling can entirely prefer radar and brake events, which are more time-critical than the actual and reference speed or engine control values. Thus, the maximal reaction time with $\text{Priorities}_{tasks}$ is about 38% less than with



(a) Average reaction delay (b) Maximal reaction delay

Fig. 8. Reaction delays: Time between radar messages and brake messages

Priorities$_{process}$, though there is no additional system load. The differences become even larger if the system is stressed with load, and with 80% additional load, task scheduling is actually 2.4 times faster than process priority scheduling.

Discussion. The results of all experiments demonstrate the benefits of task scheduling regarding both shorter and less variable reaction times of time-critical system tasks. They also point out that existing language elements of SDL are not sufficient to develop applications with real-time requirements if the hardware platform is predetermined and severely limited. These shortcomings can, in particular, be ascribed to the gap between the concurrent execution of all processes according to the SDL semantics and the required serialization of transition executions on embedded hardware. Though there is language support in SDL to prefer transition executions within single SDL processes (e.g., priority inputs)[4], the scheduling non-determinism of transition executions in different SDL processes is not addressed by SDL. Thus, the preference of specific system tasks distributed across several SDL processes can not be expressed in SDL.

To serialize transition executions, several scheduling strategies have been proposed. In our evaluation, we compare task scheduling with three common SDL scheduling strategies. In summary, the results demonstrate that existing approaches have drawbacks: FCFS strategies are in general inadequate, because they can not prefer time-critical system tasks at all. Strategies with static priorities are more adequate but suffer from their dependence on the static system specification. With task scheduling, these limitations are removed by dynamically adding information about the context of a transition execution, i.e., the system task it contributes to. Though these extensions improve delays significantly, the amount of additional information and overhead is only very low.

[4] We note that these language elements are limited, because they are based on static elements in the system's specification. In contrast, real-time tasks are created at run-time and assign priorities to transition executions dynamically.

5 Related Work

SDL real-time tasks enrich SDL in two respects: First, they improve the language expressiveness by enabling the specification and identification of SDL process- and node-spanning functionalities. Thus, there is a similarity to Message Sequence Charts (MSCs) [10], which is a common technique to describe the communication within and between nodes. The second impact of real-time tasks is on the scheduling of SDL systems, in which task priorities determine transition execution orders, thereby improving the predictability of reaction times. Hence, this section also outlines related work regarding SDL schedulers.

Because MSCs are a high-level way to specify distributed behavior, there are several proposals to transform MSCs to SDL [11,12]. In [11], Dulz et al. present the transformation of MSCs to synthetic SDL specifications used for early performance predictions. The intention of the MSC to SDL transformation proposed by Khendek et al. [12] is to achieve consistency between both specifications. For this purpose, the authors present a tool called MSC2SDL using an MSC and a target SDL architecture as input. Since MSCs are not suitable for describing complete systems, the influence of such approaches on the run-time behavior is limited. Yet, MSCs are a useful method to visualize and identify SDL task types.

Due to the scheduling non-determinism of the SDL semantics, there are many proposals dealing with the implementation of SDL schedulers. These proposals can be divided into two categories: The activity thread model, mapping SDL signal transfer to procedure calls [13,14], and the scheduling of the SDL system by means of priorities [6,15,16]. An overview of alternatives of SDL implementations can be found in [17,18].

The activity thread model differs from the SDL semantics, because it is synchronous and dissolves the distinction between communication and transition execution [17]. Yet, it is an efficient and often standard-compliant way of implementing SDL. However, different from SDL tasks, the activity thread model is not able to prefer specific transitions. Additionally, due to its synchronous execution model, deadlocks may occur in systems with cyclic signal flows [19]. To overcome these limitations, several measures, e.g., the reordering of signal outputs, have been proposed [13,14]. Similar to SDL tasks, the system execution with activity threads is driven by SDL signals and not by SDL processes.

Priority-based scheduling solutions operate either on process [6,15] or signal priorities [6,16]. E.g., C-micro [6], which is part of IBM's Rational SDL suite, supports static signal priorities as well as static process priorities. An extension of SDL's execution model with dynamic process priorities is introduced in [15], where priorities are derived from fixed transition priorities, forming the basis of a preemptive scheduler and schedulability analysis. Other than task priorities, prioritization based on processes or static signal priorities is not well-suited if transitions of a process are used to fulfill both time-critical and non-time-critical functionalitites. In [16], a scheduling approach with dynamic signal priorities, called Message Earliest Deadline First (MEDF), is proposed, sorting transition executions by means of message deadlines. For this purpose, several language extensions, e.g., annotations to specify timing constraints, are presented. Similar

to SDL tasks, the proposed execution model can pass priorities on to signal outputs during transition executions. A drawback of MEDF compared to SDL real-time tasks is the limitation of SDL's language constructs. Additionally, EDF scheduling is in general more costly.

6 Conclusions

In this paper, we have continued our previous work on SDL real-time time tasks both conceptually, and, in particular, with a strong focus on implementation and evaluation. Conceptually, we have introduced *distributed SDL real-time tasks*, which may now span transition executions of SDL processes of several SDL systems. This enables distribution of real-time functionality across nodes while preserving tight control of global scheduling decisions and transition execution priorities. Furthermore, we have added *SDL real-time task types*, which can be used to naturally specify task multiplexing in SDL processes.

We have then presented an overview of the implementation of SDL real-time tasks in our tool chain, with emphasis on transition scheduling strategies. Based on this implementation, we have conducted extensive experiments to provide evidence of the benefit of SDL real-time tasks. In particular, we have measured the gains of SDL real-time task scheduling w.r.t. standard SDL scheduling strategies and SDL process priority scheduling. In summary, the delays of time-critical transition executions grouped into SDL real-time tasks was considerably lower when using task scheduling, in particular, in situations with increasing CPU load. In the evaluated ACC system, it has, for instance, been shown that the worst case reaction delay with task scheduling is 2.4 times less than with SDL process priority scheduling, which suffers from processes executing time-critical and none-time-critical transitions. In addition, we have observed a dramatic drop of jitter with task scheduling, which accounts for far better predictability of reaction times. Our experiments also show that these real-time performance gains have been achieved without creating additional overhead during transition selection.

As future work, we are considering applications of SDL real-time tasks in real control systems to assess its benefit to the quality of control. Additionally, we plan to extend our simulation framework by Imote2 nodes as hardware-in-the-loop in order to evaluate distributed real-time tasks in large networked systems.

Acknowledgments. This work is supported by the Carl Zeiss Foundation.

References

1. International Telecommunication Union: ITU-T Recommendation Z.100 (12/11) - Specification and Description Language - Overview of SDL 2010 (2012), `http://www.itu.int/rec/T-REC-Z.100-201112-I`
2. Krämer, M., Braun, T., Christmann, D., Gotzhein, R.: Real-Time Signaling in SDL. In: Ober, I., Ober, I. (eds.) SDL 2011. LNCS, vol. 7083, pp. 186–201. Springer, Heidelberg (2011)

3. Christmann, D., Becker, P., Gotzhein, R.: Priority Scheduling in SDL. In: Ober, I., Ober, I. (eds.) SDL 2011. LNCS, vol. 7083, pp. 202–217. Springer, Heidelberg (2011)
4. Gotzhein, R.: Model-driven by SDL - Improving the Quality of Networked Systems Development (Invited Paper). In: Proceedings of the 7th International Conference on New Technologies of Distributed Systems (NOTERE 2007), pp. 31–46 (2007), http://vs.cs.uni-kl.de/publications/2007/Go07/Go07.pdf
5. Christmann, D., Gotzhein, R.: Real-time Tasks in SDL. In: Haugen, Ø., Reed, R., Gotzhein, R. (eds.) SAM 2012. LNCS, vol. 7744, pp. 53–71. Springer, Heidelberg (2013)
6. IBM Corporation: Rational SDL Suite, http://www-142.ibm.com/software/products/us/en/ratisdlsuit
7. International Telecommunication Union: ITU-T Recommendation Z.100 Annex F: Formal Semantics Definition (2000), http://www.itu.int/rec/T-REC-Z.100
8. Braun, T., Gotzhein, R., Wiebel, M.: Integration of FlexRay into the SDL-Model-Driven Development Approach. In: Kraemer, F.A., Herrmann, P. (eds.) SAM 2010. LNCS, vol. 6598, pp. 56–71. Springer, Heidelberg (2011)
9. FlexRay Consortium: FlexRay Communication System Protocol Specification Version 3.0.1 (2010)
10. International Telecommunication Union: ITU-T Recommendation Z.120 (02/2011) Message Sequence Charts (MSC) (2011), http://www.itu.int/rec/T-REC-Z.120-201102-I/en
11. Dulz, W., Gruhl, S., Lambert, L., Söllner, M.: Early Performance Prediction of SDL/MSC Specified Systems by Automated Synthetic Code Generation. In: SDL 1999 –The Next Millennium, pp. 457–471. Elsevier (1999), http://dx.doi.org/10.1016/B978-044450228-5/50030-8
12. Khendek, F., Zhang, X.J.: From MSC to SDL: Overview and an Application to the Autonomous Shuttle Transport System. In: Leue, S., Systä, T.J. (eds.) Scenarios. LNCS, vol. 3466, pp. 228–254. Springer, Heidelberg (2005)
13. Langendörfer, P., König, H.: Automated Protocol Implementations Based on Activity Threads. In: 7th International Conference on Network Protocols, ICNP (1999)
14. König, H., Langendörfer, P., Krumm, H.: Improving the Efficiency of Automated Protocol Implementations using a Configurable FDT Compiler. Journal of Computer Communications 23(12), 1179–1195 (2000)
15. Álvarez, J.M., Díaz, M., Llopis, L., Pimentel, E., Troya, J.M.: Integrating Schedulability Analysis and Design Techniques in SDL. Journal of Real-Time Systems 24(3), 267–302 (2003)
16. Kolloch, T.: Scheduling with Message Deadlines for Hard Real-Time SDL Systems. PhD thesis, Technische Universität München (2002), http://tumb1.biblio.tu-muenchen.de/publ/diss/ei/2002/kolloch.pdf
17. Bræk, R., Haugen, Ø.: Engineering Real Time Systems. Prentice Hall (1993)
18. Mitschele-Thiel, A.: Engineering with SDL – Developing Performance-Critical Communication Systems. John Wiley & Sons (2000)
19. Sanders, R.: Implementing from SDL. In: Telektronikk 4.2000, Languages for Telecommunication Applications. Telenor (2000)

Definition of Virtual Reality Simulation Models Using Specification and Description Language Diagrams

Pau Fonseca i Casas[1], Xavier Pi[2], Josep Casanovas[1], and Jordi Jové[3]

[1] Universitat Politècnica de Catalunya, Barcelona-Tech, Barcelona, Spain
{pau,josepk}@fib.upc.edu
[2] Sinetric systems, Barcelona, Spain
xpi@sinetric.com
[3] Arbora-Ausonia, Barcelona, Spain
jove.j@arbora-ausonia.com

Abstract. A full representation of a simulation model encompasses the behavior of the elements that define the model, the definition of the probability distributions that define the delays of the events that control the model, the experimental framework needed for execution, and the graphical representation of certain model elements. This paper aims to use specification and description language to achieve a full model representation by adding two extensions to the language, which allows for a complete and unambiguous definition of a discrete simulation model that is similar to a common discrete operations research simulation tool.

Keywords: Formal Languages, Specification and Description Language, SDL, Operations Research, Discrete Simulation, Virtual Reality.

1 Introduction

A discrete simulation model can be described using formal languages that allow a clear separation between the definition of the model and its implementation. However, for discrete simulation (for operations research), the use of formal languages is desirable but not common. Many of the important discrete simulation tools do not work with a formal language and often are based on proprietary syntax and tools. This proprietary representation of the simulation model often presents a challenging problem for transforming the model to a different implementation.

This paper aims to use a formal language, the Specification and Description Language [1], to achieve a complete representation of a simulation model. This representation encompasses the behavior and structure of the model as well as the graphical representation of the model execution, which simplifies the model validation as suggested in [2]. We developed two simple extensions to the 2000 version of the language (SDL-2000), which allow for a complete definition of a

F. Khendek et al. (Eds.): SDL 2013, LNCS 7916, pp. 258–274, 2013.

discrete simulation model that is similar to common discrete operations research simulation tools; however, we use an unambiguous graphical and standard formal language to improve the model description and reuse, while maintaining the benefits of a formal language as suggested in [3]. To achieve this objective, it is necessary to define the behavior of the elements that define the model: the characterization of the probability distributions that define the delays of the different events that control the model, the experimental framework for execution, and the structures for model representation.

Operations research simulation tools apply different paradigms to represent the real world. The discrete-event paradigm includes classical languages, such as GPSS/H [4,5] and SLAM II [6]. These tools have features that are similar to Simprocess [7], Arena [8], Simio [9], and Simul8 [10]. The paradigms that are usually represented with these tools include process-interaction or event-scheduling [11,12]. Other simulation tools do not exactly follow these paradigms. For example, Witness [13] constructs processes using push/pull rules for the different elements in the model. This type of software often allows for **if... then... else rules** for the definition of resources and attributes and allows for the use of dynamic link libraries (DLLs) to use specialized code defined with C++, C# or Visual Basic.

These software tools always allow for a complete definition of the model behavior, structure, time and representation.

Similar to the current paper, there are certain programming libraries and infrastructures that allow for defining a simulation model following a formal language. The tools related to DEVS [14] and PetriNets formalisms [15,16] often represent an excellent alternative to define and implement a simulation model with the previously mentioned simulation tools. An interesting example of extending Petri Nets can be reviewed on [17] . For the DEVS tools and infrastructures, a non-exhaustive list can be reviewed in [18]. For Petri Nets, a similar list is available in [19].

CD++ [20] and DEVSJAVA [21] are examples of DEVS infrastructures. CD++ is mainly a toolkit for Discrete-Event modeling and simulation and the environment is based on the DEVS (Discrete-Event systems Specifications) formalism. Currently, a plug-in exists that allows for using a graphical interface with the Eclipse platform. Figure 1 includes the CD++ plug-in on the eclipse platform.

One of the strengths of DEVS is that it supports the transfer of models from one concrete tool to another (due to the use of XML). Several attempts have been made to define a standard XML representation for DEVS with a complete and common XML schema [22,23].

Specification and Description Language also has different tools that allow for the implementation of a simulation model [24,25,26]. These tools allow the generation of code from the model representation. In this study, we use *Specification and Description Language Parallel Simulator*, SDLPS [23,27].

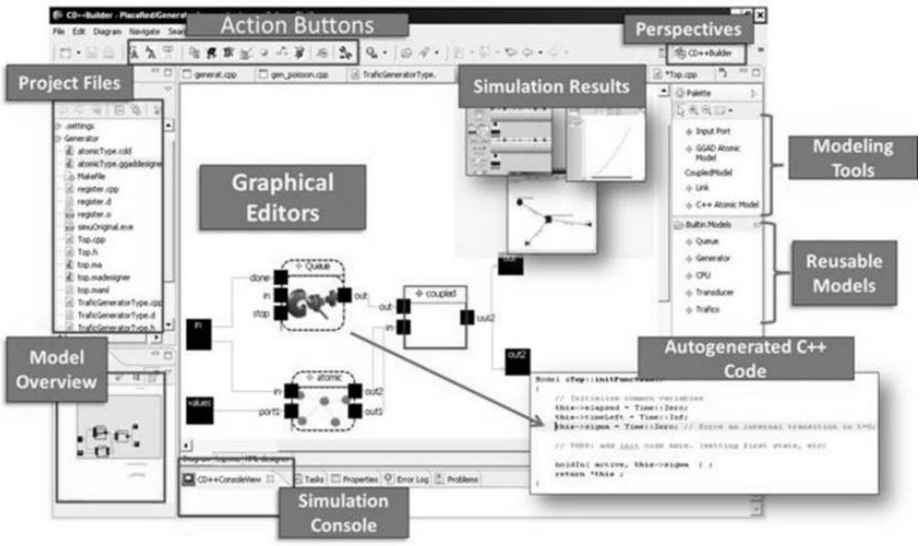


Fig. 1. CD++ infrastructure on the Eclipse Platform (source: `http://cell-devs.sce.carleton.ca/mediawiki/index.php/` Screenshots)

2 Our System

This study describes a small part of a system from an ongoing project with the Arbora-Ausonia enterprise. The main elements of this sub-system include a conveyor and robot.

Fig. 2. An example of a roller conveyor similar to that modeled in the project (Source Wikipedia)

For confidentiality, we only show the behavior of one of the more common elements in industry, a roller conveyor belt (see Fig. 2). However, for our purposes this is enough to explain the system. Although it is a common element, the belt is

complex enough to justify the use of the extensions to SDL-2000. Additionally, a conveyor includes the complex behavior of boxes that continuously travel the length of the unit and requires minimization of the number of events (or signals).

2.1 Used SDL-2000 Extensions

We define models using the Specification and Description Language, but to fully define a simulation model, as was performed with certain tools presented, it is not enough to only allow for the definition of the model structure (that is defined for the different elements in the model) and behavior (that is defined for each element). A complete definition of the behavior (including the time) and graphical representation of the simulation model are required.

2.2 Time and Priority Management

For a discrete simulator, a complete definition of the behavior of a model is needed to describe the time related to the execution of each event that manages its evolution. Usually each type of event has a specific probability distribution, which determines when the event is executed. For an event scheduling simulator, the engine manages the time for all the events and decides where and when these events must be sent (to other simulation elements, which are the agents in a Specification and Description Language model).

SDL-2000 has two main structures for time management, *Timers* and *Delaying Channels* [1].

Delaying Channels of SDL-2000 were not acceptable for representing the delays in a simulation model because in SDL-2000 there were no existing mechanisms to define the required time to reach the destination with these channels. The *Delaying Chanel* represents a delay in the transmission of the signal, but the probability distribution of this delay cannot be defined. The other mechanism, *Timers*, is inadequate, because each different instance of a signal that can travel in parallel requires the definition of a new *Timer*. For example, if we need to send a signal to represent the arrival of new entities to a machine, when a new arrival is sent to this machine, the *Timer* is reprogrammed; thus, the signal has not arrived to its final destination and is reprogrammed. Only one instance of the signal represented by the *Timer* can travel through the system. Additionally, *Timers* cannot represent the priorities. This represents a strong limitation in order to perform a more readable representation of a dynamic system where the delays and priorities must be completely defined.

Specification and Description Language time management has been studied by several groups [28,29]. Specifically, [29] presented an extension that defines three kinds of transitions; (i) eager (consumed without delay), (ii) lazy (not urgent) and (iii) delayable (an enabling condition depending on time). For a discrete simulator, all the transitions can be considered delayable because all the transitions have a defined time. An eager transition is equivalent to a delayable transition with the temporal condition set to now=x [29].

Considering these issues, we implemented extensions of SDL-2000 in SDLPS. In SDLPS, all the signals carry the parameters with the following elements: (i) *ExecutionTime* or *delay*, the time when the event must be executed. (ii) *Priority*, the priority of the event, which is used to eliminate simultaneity of events. (iii) *CreationTime*, the time when the event is created. (iv) *Id*, an identifier of the event. (v) *Time*, the clock reading for the process (represents the time related to the last event that was processed by the process). (vi) *Destination*, the final destination (process PId) of the signal.

The parameter *event* delays or sorts (by priority) the different signals. When a *signal* is received, SDLPS uses the *event* parameter to manage the time and the priorities of the signal. In SDLPS, we can use extension elements to define this parameter related to the signal, as shown in Fig. 3. Not all the parameters of the *event* structure have to be defined: only those required to fully define the behavior of the model.

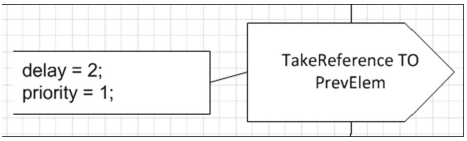


Fig. 3. A *delayable* signal. This *signal* requires 2 time units to reach its destination. Additionally, priorities can be defined to avoid ambiguity when two signals reach the destination at the same time.

Summarizing this section, to manage time we add to the language:

1. All signals can have a time delay: each signal instance output has a parameter that defines the time needed to travel to its final destination (i.e., a *delay* or the value of the *ExecutionTime* value minus the current time) and an input queue schedule parameter defining the priorities with respect to the other signal instances that arrive at the destination at the same time (i.e., the *ExecutionTime* value). A signal instance is therefore only available in the destination input port when the current time is greater or equal than the *ExecutionTime*.
2. The signals in the input port are scanned in the following order to determine if there is an enabled signal: *ExecutionTime* and priority. Signal *priority* determines the signal that is processed first. If two signals with the same *ExecutionTime* and *priority* exist, the implementation decides which signal is executed first (the model does not specify this scenario).

These proposed extensions are included in the SDL-2010 release of the standard [1].

2.3 To Represent the Model

To represent the model, we assume that all SDL-2010 *agents* (*system*, *block* and *process*) can be represented. Thus, the *agent* that represents the conveyor has a real position on the model graphical representation (or layout) and also has a file that describes its shape. This representation suggests that the definition of an *agent* can also have information related to its representation (information regarding the visual behavior of the *agents* in a 2D or 3D environment). Our current implementation provides this representation in an XML file that describes the initial position of the *agents* and a file that describes its shape (see Fig. 4).

```
<Agent name='BCinta2_PCinta2'>
        <state name='ROLLING'>
                <mesh scale='1'>default.obj</mesh>
                <pos x='0' y='0' z='0' />
                <rot x='0' y='0' z='0' />
        </state>
</Agent>
```

Fig. 4. XML representation of an agent with the extensions used to represent the agent in a virtual reality environment. For agent *BCinta2_PCinta2* in the state of *ROLLING, default.obj* represents the agent and the initial position and orientation is (0,0,0).

In this case in the shape for the agent BCinta2_PCinta2 (a process) is in a file (default.obj) and its initial position is 0,0,0 with no rotation.

To represent the model with a 3D (or 2D) animation we define a library that can be accessed during the execution time using a *Procedure Call*. In Fig. 5 there is an example of this *Procedure Call* called *AnimTo*.

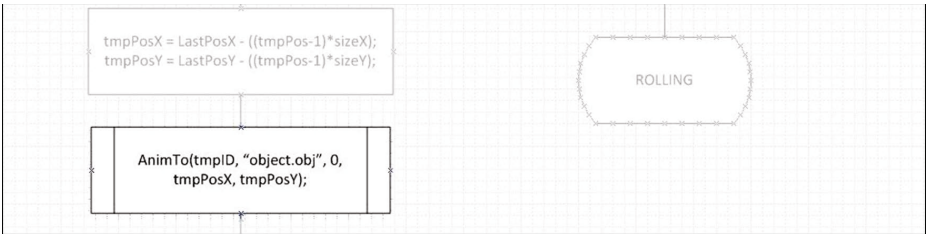


Fig. 5. The *AnimTo* procedure allows for defining the animations to represent the model in a 3D (or 2D) environment

When the simulator executes the procedure it creates a representation in the 3D environment. The parameters of this call are described as follows:

AnimTo(ID, meshPath, delay, x, y, z)

In this case, the element is identified by the *ID* identity and moves to (x, y, z) coordinates at the current *execution time* plus the time *delay*. The shape is defined by a mesh in a WaveFront OBJ format that is stored in the *meshPath* path parameter.

A sequential execution of this procedure creates an animation between the specified points. For the conveyor case, this procedure is used to represent the elements that the conveyor transports. Thus, to create an animation from the beginning to the end of the conveyor we must execute:

AnimTo(ID, meshPath, 0, x0, y0, z0)
AnimTo(ID, meshPath, delay, x, y, z)

Where (x0, y0, z0) is the initial position, (x, y, z) is the last position and *delay* is the time to move the distance of the conveyor.

2.4 Conveyor Model

It is not our intention to perform a complete description of the model. Instead, we detail the behavior of the most complex element (with representation), the conveyor. The conveyor has different parameters including the speed of the boxes, defined by the rotation speed of the rollers. Additionally, we can define the number of boxes that can be carried (this parameter depends on the size of each box). All the parameters that can be configured by the user are represented on the SDLPS diagrams using DCL statements. The behavior of the conveyor is described as follows.

Elements can be added to the beginning of the conveyor. The conveyor continuously moves the elements to the end of its structure. Because this conveyor is composed of different cylinders that move the elements (not by a continuous belt), when the first elements reaches the end of the conveyor (but cannot leave because the next element is blocked), the other elements can continue moving. This relationship implies that the conveyor behaves as a buffer that can store elements until *MaxElems* boxes is reached (defined in the declarations).

Because we are planning to use SDLPS to interpret and generate the model, the notation of the declarations and the code for the diagrams is written in ANSI C language to simplify the DLL needed to perform the execution of the system. This coding implies that the language we are using is similar to SDL-RT [26] or C-language binding as in SDL-2010 [1]. We define following terminology to describe the diagrams.

Entity: the entities are the elements that move through the system using different facilities to define different processes. Each entity "travels" through different agents that perform operations on them.

Process: despite having an agent titled process in SDL-2010, a theoretical process in operations research represents the set of operations that must be performed to the entities. The definition of these operations is based on the SDL-2010 process agents.

Event: An event occurs in the simulation model and implies the modification of select state variables of the model. In our approach, the events are represented using SDL-2010 *signals*.

With these considerations for the model definition, there are two main events (signals), *NewReference* (used to indicate that a new element reaches and attempts to enter another element) and *TakeReference* (the other element, i.e., agent, attempts to take one of the elements that was completed in the process). From these two signals, we can define a model that is similar to a PUSH/PULL paradigm for a process interaction engine and similar to the simulation tools that were described previously.

For the conveyor, we only consider the ROLLING state. In this state, we can receive events, including *NewReference*, *Full*, *Reroll*, *TakeReference*, *Unblock* and *Roll*. The behavior of these elements is described in Fig. 6, Fig. 7 and Fig. 8.

Figure 6 shows the declarations that contain the elements of the conveyor that can be modified to define different scenarios. For example, the variable *double MaxElems=10* can be changed to test the difference between using a short (*MaxElems=5*) or a long (*MaxElems=20*) conveyor.

3 The Implementation and Execution of the Model

The model was implemented with SDLPS software, developed in the InLab FIB of the Polytechnic University of Catalonia [23,27]. This tool uses SDL-RT (the code for tasks is defined using C language) and the the extensions to SDL-2000. Regarding the infrastructure, SDLPS was built with C++ and C languages. The model code (written in C for the tasks and procedures of the SDL-2000 blocks) are used through a DLL. The SDL-XML model is generated with a plug-in on Microsoft Visio®. This coding implies that the model can be mainly validated and verified by reviewing the graphic diagrams on Microsoft Visio®. This property dramatically simplifies the interaction between the different parts involved in the project.

SDL-2010 does not define the ordering of events when two events with the same *ExecutionTime* have the same priority. This situation must be defined in the model or by the simulation engine. SDLPS eliminates this ambiguity by storing these events in a FIFO queue.

From Fig. 9, the simulator shows the model diagrams that will be simulated. The system uses an XML representation of the model that is obtained from Microsoft Visio® with the SanDriLa© plug-in.

Because we use this infrastructure, no specific implementation was performed for this project, simplifying the verification process required for simulation projects [30].

Figure 10 shows the steps that are simplified (red in the online paper/ grey in the printed paper) by this methodology that are needed for a simulation.

The obtained results from the model emulation trace can be presented in Microsoft Excel® or SDLPSEye, which is capable of representing information in a 3D environment (for the representation events described in the previous section). All agents can have unique representations, and due to the time extensions, we can determine the movement of the simulation entities.

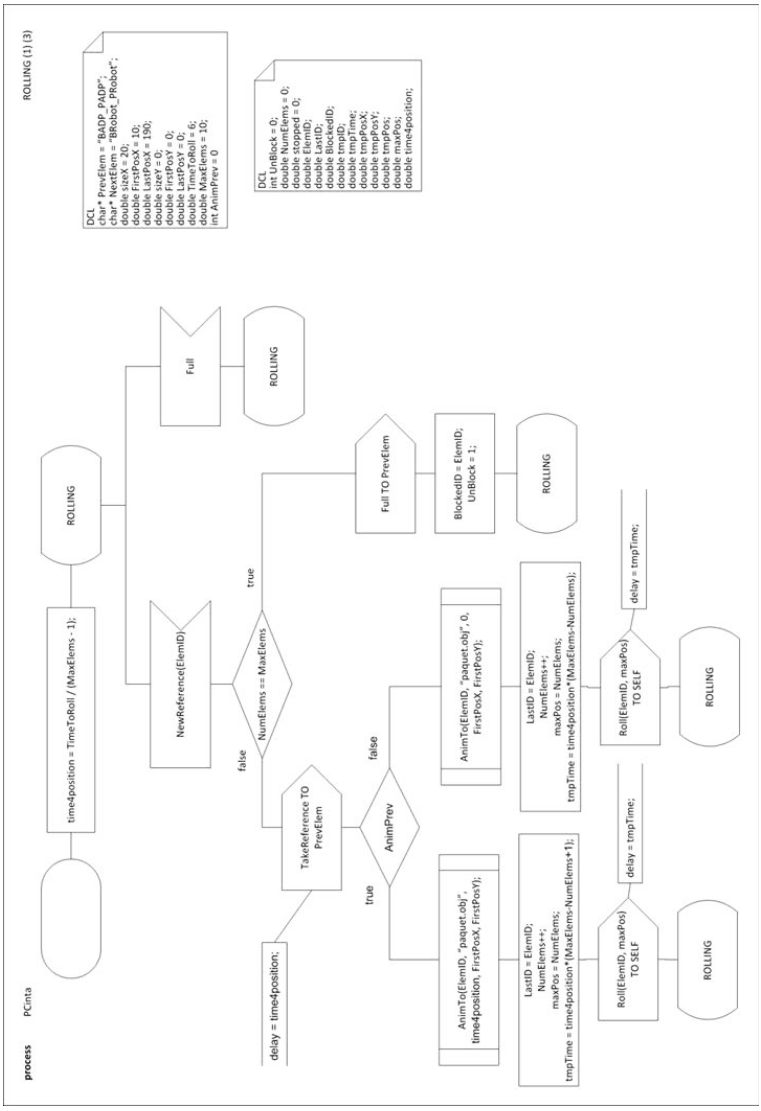


Fig. 6. PCinta for the ROLLING state (1/3). This figure shows how the process is instantiated. The time4position variable is defined from the START symbol and is the time that is required for each of the pieces to reach a specific position on the conveyor. From the "rolling" state, we can receive a *NewReference* signal. We then analyze the number of elements that are in the conveyor. If this number is equal to the conveyor capacity, the conveyor sends the "Full" signal to the previous agent. Otherwise, the conveyor processes the element (sends the *TakeReference* signal to the previous agent). Then, the program determines if the animation must be completed from the beginning or if this element is connected to another conveyor *AnimPrev* value. In this example, the *AnimPrev* value is 0, such that the previous element is not a conveyor.

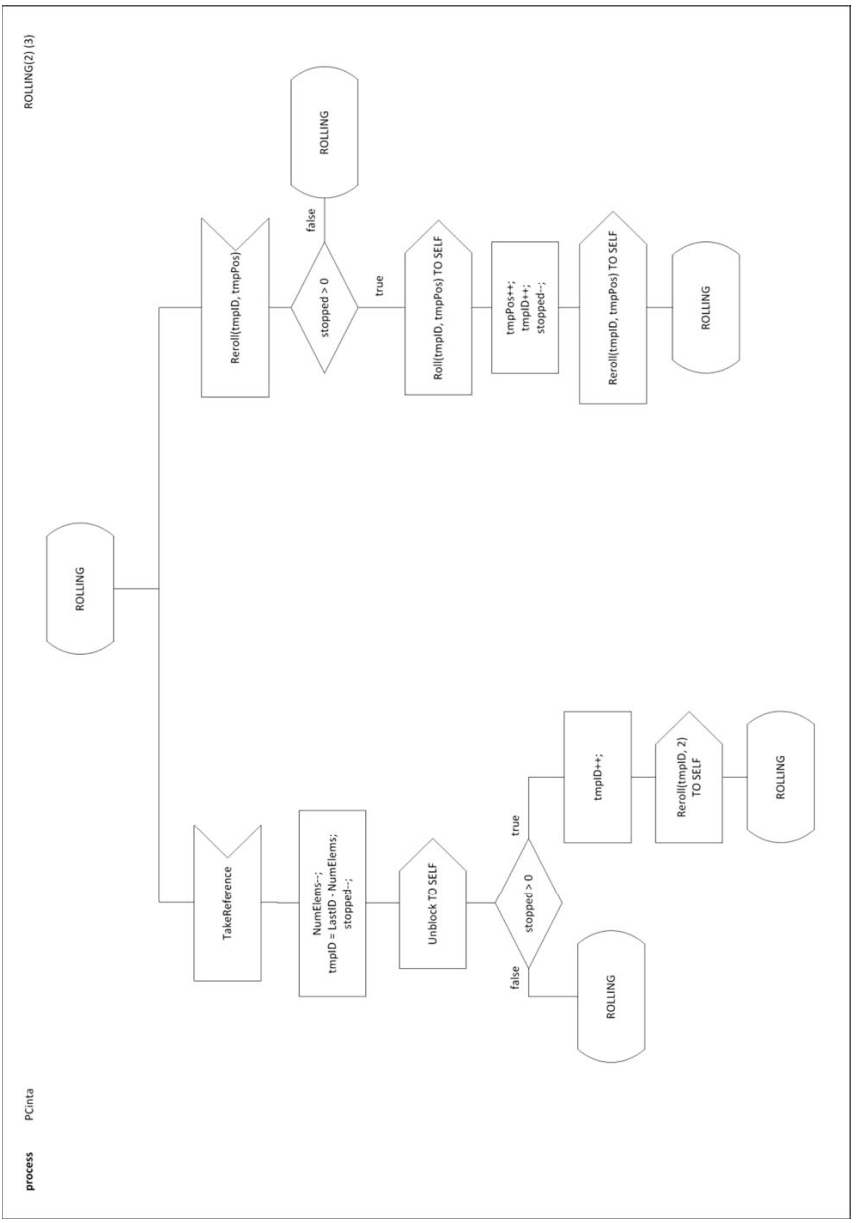


Fig. 7. PCinta for the ROLLING state (2/3). When a conveyor receives a *TakeReference* signal, the *NewReference* is accepted by the next agent. Thus, after receiving this signal, we can decrease the number of elements of our agent and unblock the previous agent (in case that agent was blocked by our agent). We also can observe the behavior of the conveyor when it receives a *Reroll* signal. This signal is sent by the agent (self-signal) and is used to start the motion of the conveyor after a blocking occurs.

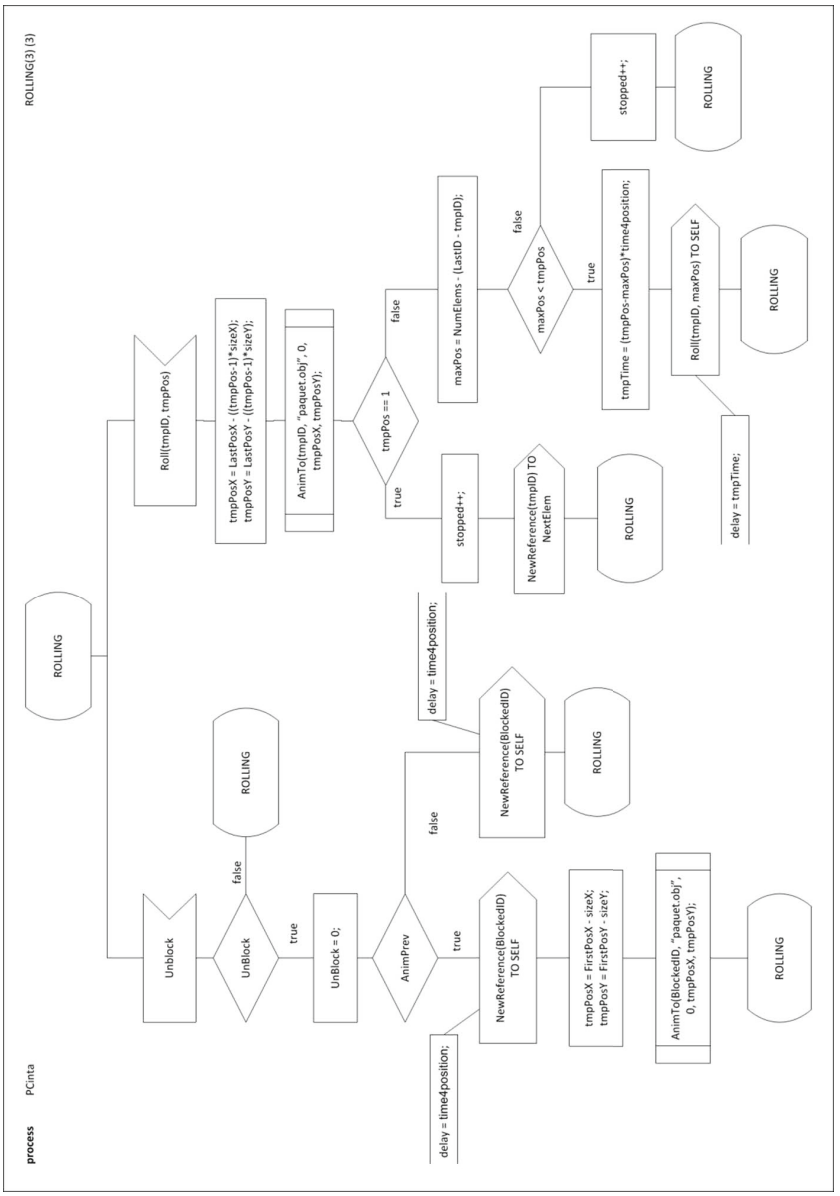


Fig. 8. PCinta for the ROLLING state (3/3). A conveyor receives an *Unblock* signal when we need to unblock the previous agent. To unblock the previous agent, we simulate receiving a *NewReference* signal. We also observe the formalization of the *Roll* signal. This signal is used to simulate the traveling time of the elements along the conveyor. When an element attempts to travel from the current position to position number 5, we send a *Roll* signal to the self conveyor with a delay, which represents the time for traveling from one position to the next.

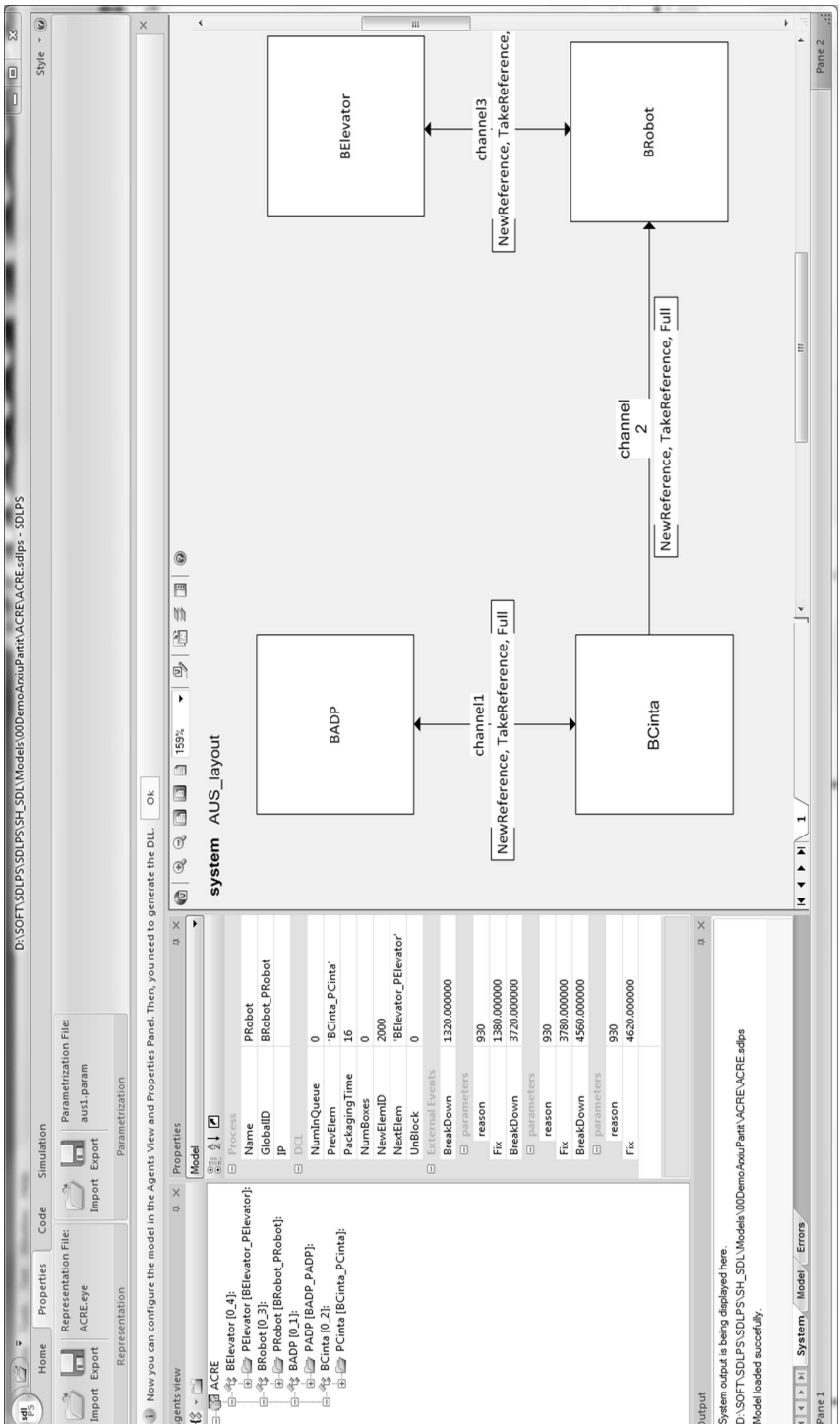


Fig. 9. SDLPS interface with the model

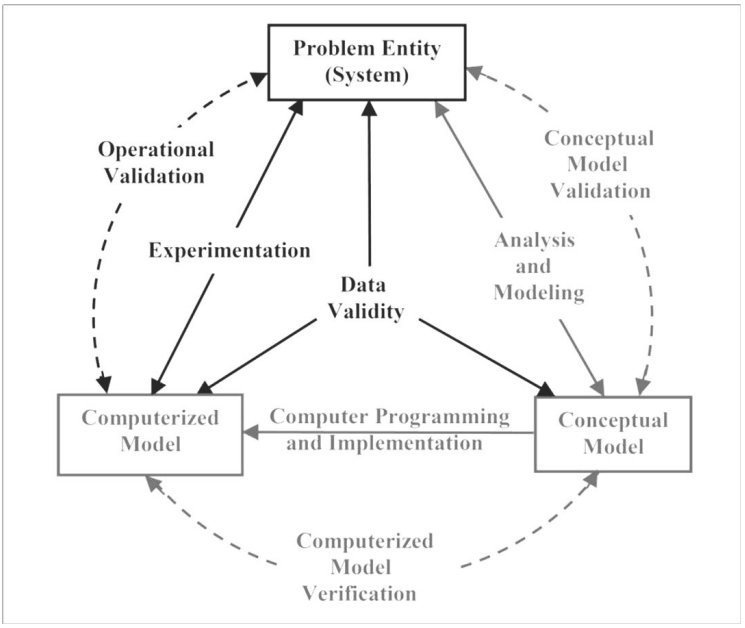


Fig. 10. Simplified version of the modeling process [30]

3.1 The Model Eyes: SDLPSEye

The representation of the model is stored in a trace file. This file contains the representation events from all the simulations. The events can be generated by the *AnimTo* procedure or by changing any process of the model. For the latter, the event must be completed with a representation of every possible state of the given process. Figure 11 is an example with a defined state representation of every agent. In this case, the agent *BRobot_PRobot* has 3 states (IDLE, BLOCKED and WORKING). Figure 12 is an example of a representation of an event sequence. In this case, there are two representation events types, which include *EYE_SetState* and *EYE_AnimTo.*

Figure 13 and Fig. 14 represent the model that was obtained from the description of the SDL diagrams. The boxes are the *mesh* elements (the representation of the agent) defined on the extensions.

4 Concluding Remarks

Despite that the obtained results from the simulation model can be used for a decision process in industry, the Specification and Description Language becomes an excellent language to fully describe the behavior of the enterprise elements, due the added time and representation capabilities. As we see in the introduction

```
<ModelInfo>
    <ModelName>Ausonia</ModelName>
    <Agents>
        <Agent name='BRobot_PRobot'>
            <state name='IDLE'>
                <mesh scale='1'>box_green.obj</mesh>
                <pos x='420' y='0' z='0'/>
                <rot x='0' y='0' z='0'/>
            </state>
            <state name='BLOCKED'>
                <mesh scale='1'>box_red.obj</mesh>
                <pos x='420' y='0' z='0'/>
                <rot x='0' y='0' z='0'/>
            </state>
            <state name='WORKING'>
                <mesh scale='1'>box_yellow.obj</mesh>
                <pos x='420' y='0' z='0'/>
                <rot x='0' y='0' z='0'/>
            </state>
        </Agent>
        <Agent name='BElevator_PElevator'>
            <state name='RUNNING'>
```

Fig. 11. XML file defining the representation of the model

```
<Events>
    <EYE_SetState xTime='0.000000' agent="BRobot_PRobot" state="IDLE"></EYE_SetState>
    <EYE_SetState xTime='0.000000' agent="BElevator_PElevator" state="RUNNING"></EYE_SetState>
    <EYE_SetState xTime='0.000000' agent="BADP_PADP" state="RUNNING"></EYE_SetState>
    <EYE_SetState xTime='0.000000' agent="BCinta_PCinta" state="ROLLING"></EYE_SetState>
    <EYE_AnimTo xTime='0.000000' agent='1000.000000'>
        <mesh scale='1'>paquet.obj</mesh>
        <pos x='10.000000' y='0.000000' z='0'/>
    </EYE_AnimTo>
    <EYE_AnimTo xTime='0.555556' agent='1001.000000'>
        <mesh scale='1'>paquet.obj</mesh>
        <pos x='10.000000' y='0.000000' z='0'/>
    </EYE_AnimTo>
    <EYE_AnimTo xTime='2.111112' agent='1002.000000'>
        <mesh scale='1'>paquet.obj</mesh>
        <pos x='10.000000' y='0.000000' z='0'/>
    </EYE_AnimTo>
    <EYE_AnimTo xTime='4.777780' agent='1003.000000'>
        <mesh scale='1'>paquet.obj</mesh>
        <pos x='10.000000' y='0.000000' z='0'/>
    </EYE_AnimTo>
    <EYE_AnimTo xTime='5.000004' agent='1001.000000'>
        <mesh scale='1'>paquet.obj</mesh>
        <pos x='170.000000' y='0.000000' z='0'/>
    </EYE_AnimTo>
    <EYE_SetState xTime='5.000004' agent="BRobot_PRobot" state="BUSY"></EYE_SetState>
    <EYE_AnimTo xTime='5.000004' agent='1000.000000'>
        <mesh scale='1'>paquet.obj</mesh>
        <pos x='190.000000' y='0.000000' z='0'/>
    </EYE_AnimTo>
    <EYE_AnimTo xTime='5.666671' agent='1000.000000'>
```

Fig. 12. A trace of the representation, representing the complete behavior of the model

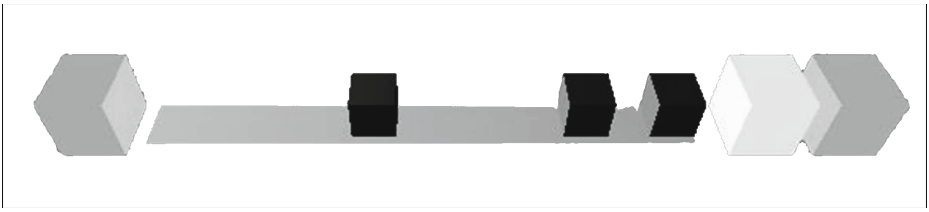


Fig. 13. The conveyor contains three boxes. The two boxes at the end are waiting for service.

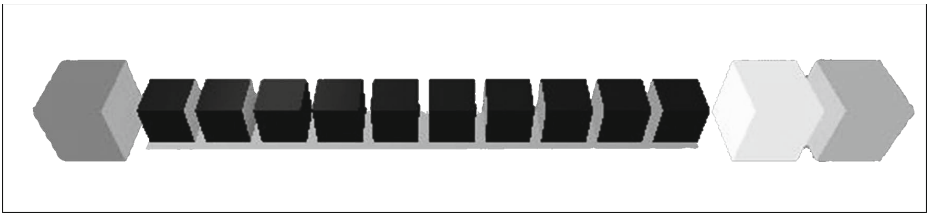


Fig. 14. The conveyor is full and blocking the box generator (in red). In this example, 4 blocks are represented. From left to right, the first, third and fourth elements have the simplest representation, consisting of a color representation of each state. For the conveyor block, a more accurate representation is presented using the *AnimTo* procedure.

there are certain formal languages, programming libraries and infrastructures that allow defining a simulation model. However, any of these languages allows the definition of the model representation. To achieve this it is needed to use proprietary infrastructures and tools. This obviously is far from the objective to achieve a complete and formal representation of a simulation model independent of the tool or infrastructure used to finally perform the implementation.

Thus, once the system is fully described, a client can modify the structure of the model using only Microsoft Visio® SDL-2010 diagrams with the SanDriLa [31] plug-in. Additionally, because the more important parameters of the model are defined in the declarations and can be modified directly in the SDLPS infrastructure, simple modifications (new parameterizations) of the model are not time intensive. Thus, in industry, the managers can validate the accuracy of certain proposed alternatives using common tools such as Microsoft Visio® .

Additionally, this tool can validate the accuracy of several of the proposed solutions. Thus, the diagrams that represent the tacit and explicit knowledge of the industry can be validated, allowing for the representation and validation of these types of knowledge.

We are currently continuing with the project implementation in industry and installing the system for the clients so they can modify and define their own models. The main elements of the system can be predefined with SDL-2010 blocks that implement a library, so that several elements can be reused.

As shown in this study, specification and description language can be used in operations research to fully represent discrete simulation models with temporal extensions and a library to represent the basic operations to render a 3D environment.

Because the validation of the model is performed in the SDL-2010 representation of the model, non-simulation specialists that are experts in system behavior can understand the model's behavior. Thus, all the actors involved in the project can participate in the model validation. Thus, SDL-2010 diagrams can be used to represent the tacit knowledge that expresses the behavior of the complex interactions between several actors in industry.

References

1. International Telecommunication Union: Recommendation Z.100 (12/11) Specification and Description Language – Overview of SDL-2010, http://www.itu.int/rec/T-REC-Z.100
2. Earle, N., Henriksen, J.: Proof Animation – Better Animation For Your Simulation. In: Proceedings of the 25th Conference on Winter Simulation (WSC 1993), pp. 172–178. ACM (1993), http://doi.acm.org/10.1145/256563.256617
3. Brade, D.: Enhancing modeling and simulation accreditation by structuring verification and validation results. In: Proceedings of the 32nd Conference on Winter Simulation (WSC 2000), pp. 840–848. Society for Computer Simulation International (2000)
4. Gordon, G.: The Development of the General Purpose Simulation System (GPSS). ACM SIGPLAN Notices 13(80), 183–198 (1978), http://portal.acm.org/citation.cfm?doid=960118.808382
5. Fonseca i Casas, P., Casanovas, J.: JGPSS, an Open Source GPSS Framework to Teach Simulation. In: Winter Simulation Conference (WSC 2009), pp. 256–267. Winter Simulation Conference (2009)
6. Pritsker, A.: Introduction to simulation and SLAM II. Halsted Press (1986)
7. CACI: Simprocess, http://simprocess.com/
8. Rockwell Automation: Arena Simulation Software, http://www.arenasimulation.com/
9. Simio LLC: Simio forward thinking, http://www.simio.com/index.html
10. Simul8 Corporation: Simul8, http://www.simul8.com/
11. Guasch, A., Piera, M., Casanovas, J., Figueras, J.: Modelado y simulación. Edicions UPC (2002)
12. Law, A., Kelton, W.: Simulation Modeling and Analysis. McGraw-Hill (2000)
13. Lanner: Witness, http://www.lanner.com/en/witness.cfm
14. Zeigler, B.P., Kim, D., Praehofer, H.: DEVS formalism as a framework for advanced distributed simulation. In: Proceedings of the 1st International Workshop on Distributed Interactive Simulation and Real-Time Applications (DIS-RT 1997), pp. 15–21. IEEE Computer Society (1997)
15. Peterson, J.: Petri Net Theory and the Modeling of Systems. Prentice-Hall (1981)
16. Petri, C.: Kommunikation mit Automaten. University of Bonn, Bonn (1962)
17. Liu, R., Kumar, A., van der Aalst, W.: A formal modeling approach for supply chain event management. Decision Support Systems 43(3), 761–778 (2007)
18. Wainer, G.: DEVS tools, http://www.sce.carleton.ca/faculty/wainer/standard/tools.html

19. University of Hamburg: Petri Nets Tool Database, http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/complete_db.html
20. Wainer, G.: CD++: a toolkit to develop DEVS models. Software, Practice and Experience 32(3), 1261–1306 (2002)
21. Zeigler, B.P., Sarjoughian, H.S.: Creating Distributed Simulation Using Devs M&S Environments. In: Proceedings of the 32nd conference on Winter simulation (WSC 2000), pp. 158–160. Society for Computer Simulation International (2000)
22. Risco-Martín, J.L., Mittal, S., López-Peña, M.A., De la Cruz, J.M.: A W3C XML Schema for DEVS Scenarios. In: Proceedings of the 2007 Spring Simulation Multiconference (SpringSim 2007), vol. 2, pp. 279–286. Society for Computer Simulation International (2007)
23. Fonseca i Casas, P., Casanovas, J.: Towards a SDL-DEVS Simulator. In: The Third International Conference on Advances in System Simulation (SIMUL 2011), pp.188–194. International Academy, Research and Industry Association (2011)
24. Cinderella ApS: Cinderella SDL 1.3, http://www.cinderella.dk
25. IBM: Rational SDL Suite, http://www-947.ibm.com/support/entry/portal/overview/software/rational/rational_sdl_suite
26. Specification & Description Language - Real-Time (2006), http://www.sdl-rt.org/
27. Fonseca i Casas, P.: SDL distributed simulator (poster) In: Proceedings of the 40th Conference on Winter Simulation (WSC 2008). Winter Simulation Conference (2008), http://www-eio.upc.edu/~pau/?q=node/67
28. Bozga, M., Graf, S., Mounier, L., Kerbrat, A., Ober, I., Vincent, D.: SDL for Real-Time: What Is Missing? Interval project publication, http://www-interval.imag.fr/Pub/sam2k.ps
29. Bozga, M., Graf, S., Mounier, L., Ober, I., Roux, J.-L., Vincent, D.: Timed Extensions for SDL. In: Reed, R., Reed, J. (eds.) SDL 2001. LNCS, vol. 2078, pp. 223–240. Springer, Heidelberg (2001)
30. Sargent, R.G.: Verification and validation of simulation models. In: Proceedings of the 39th Conference on Winter Simulation (WSC 2007), pp.124–137. Winter Simulation Conference (2008)
31. SanDriLa Ltd.: www.sandrila.co.uk/visio-sdl/

Integration of SDL Models into a SystemC Project for Network Simulation

Pavel Morozkin, Irina Lavrovskaya,
Valentin Olenev, and Konstantin Nedovodeev

Institute of High-Performance Computer and Network Technologies,
Saint Petersburg State University of Aerospace Instrumentation,
Saint Petersburg, 190000, Russia
{pavel.morozkin,irina.lavrovskaya,
valentin.olenev,konstantin.nedovodeev}@guap.ru

Abstract. The paper proposes an approach for integration of a number of SDL model instances into a SystemC project. It is done by conversion of an SDL model into a C/C++ library. Implementation of the library is performed by means of post-processing of previously auto-generated C code by the CAdvanced code generator. The main benefit of the approach is reducing of the project work effort and achieving a better quality of the simulation results.

1 Introduction

Key interests of industrial telecommunication companies are increased quality of products with reduced time-to-market. Modeling is a mechanism that meets these requirements by its application at the early stages of product development. This paper focuses the communication protocol development. The ITU Specification and Description Language (SDL) [1,2] and SystemC [3–5] language can both be used separately for the modeling of the communication protocols. SystemC network models are used to explore the behavior, functional and non-functional properties of protocols. Formal SDL models focus primarily on the protocol behavior exploration at the specification development stage. In general, the development of protocol stack models in SystemC and in SDL is performed in parallel. However, resources are often limited, so this way of development carries project delay or commercial risks. This paper proposes a new approach for integrating the SDL model instances into the SystemC network model by means of a special protocol stack library. This approach minimizes the potential risks and decreases the complexity of a project. The library implemented in accordance with the proposed approach contains an SDL model of a protocol stack.

2 The Problem Statement

Nowadays, modeling plays an important role in protocol stack development. It is one of the most efficient methods of protocol mechanisms development, and

F. Khendek et al. (Eds.): SDL 2013, LNCS 7916, pp. 275–290, 2013.

is performed by implementation of high-level behavior models and testing them. SDL and SystemC are widely used languages for these purposes.

SDL is a formal description technique (FDT) [6] based on a formal semantics and is widely used for specification and investigation of event-driven communication systems. Moreover, the formal SDL specification can be taken as a part of the official protocol specification as a reference. The SystemC language is a C++ library, which provides a capability of event-driven simulation and system design using the Object Oriented Programming (OOP) paradigm and software design patterns. These features make SystemC an appropriate tool for exploration of functional and performance characteristics of protocols by simulation of network models operation. However, implementation of large SystemC projects can lead to difficulties during debugging. On the other hand, SDL has a graphical representation that helps to design a protocol stack model rapidly. Therefore, the problem that is faced is how to efficiently use the SDL and SystemC languages together.

The common use of SDL and SystemC in one model could decrease time costs for protocol development. To provide this common use, we use an SDL/SystemC co-modeling method [7,8]. In this method the SystemC tester manages the simulation of the SDL model, configures it, generates test sequences, etc. This method partly solves the efficiency problem, but has a list of drawbacks. The most important issue is that it is impossible to create different numbers of SDL model instances for the tester environments to use in network simulations. Another complexity is that the SystemC developer should understand the principles of SDL model operation.

The problem can be solved by means of a special library, which can be applied during networks simulation in SystemC. This library should implement the original SDL model and SDL simulation kernel as well as provide special services for the user (term 'user' stands for the SystemC developer, who uses the library).

3 Overview of SDL/SystemC Co-modeling Approach

3.1 Tool Choice

The approach of SDL/SystemC co-modeling described in this paper assumes that we have a SystemC project that corresponds to the whole model to be considered. The whole model contains SDL and SystemC parts. Consequently, this approach uses a C/C++ representation of the SDL system [9]. Before starting a description of the discussed approach, we need to introduce general principles of co-modeling with some requirements and important notions for modeling. Consider some abstract SDL tool. This SDL tool should meet the following requirements:

1. Provide a possibility to generate C/C++ code for the implemented model that is the equivalent of the SDL.
2. The generated C/C++ code operation should be controlled by some kind of a manager engine (*SDL_kernel*).

3. The *SDL_kernel* should provide a number of functions for initialization and simulation of the SDL model. For the further discussion it is necessary to introduce declarations for two main functions: *SDL_Init()*, which is responsible for initialization, and *SDL_Simulate()*, which is responsible for emulation of the SDL system, so that one SDL transition is executed during each call of this function. One SDL transition is a system state change from one to another.

It should be pointed out that this kind of the SDL tool already exists. For example, all these features are provided by the IBM Rational SDL Suite [10].

3.2 Modeling with SDL and SystemC

Modeling with SDL and SystemC is an approach that focuses on modeling of systems that include SDL and SystemC models. This model consists of an SDL model of a protocol layer and a SystemC model of the same layer. The SDL/SystemC co-model is represented by a SystemC project, which contains SDL and SystemC parts. The SystemC model is a master and it provides all the mechanisms for simulation. The SDL part is represented by C/C++ code, which was generated from the original SDL system. Generation of code is performed by means of the CAdvanced code generator, which is a part of the IBM Rational SDL Suite.

The process for connection of SDL and SystemC parts can be subdivided into the following stages:

1. Preparation of the SDL system to be the part of the whole model.
2. Generation of C/C++ code on basis of the SDL system.
3. Insertion of this C/C++ code code to the *SDL_kernel*.
4. Preparation of the SystemC part of the model.
5. Integration of the *SDL_kernel* with the generated C code into the whole model.

According to this approach, the SystemC model is a master and the SDL model is a slave. So SystemC provides the mechanisms for modeling. Co-modelling organization starts after implementation of the SDL and the SystemC parts of the model. The SystemC project should contain a special thread, which is intended for the SDL part (*SDL_thread*). This thread calls the *SDL_kernel* function *SDL_Simulate()*. Control of the *SDL_thread* can be specified in any acceptable way. The choice of this way depends on the requirements of the modeled system. Initialization of the SDL part of the model requires a call to the *SDL_Init()* function.

One of the most interesting questions in the area of SDL/SystemC co-modeling is how scheduling is organized, because of the difference in the notions of the SDL and SystemC modeling times. According to the SDL/SystemC co-modeling approach, SystemC provides all necessary mechanisms for scheduling of events. Each point of the modeling time corresponds to a number of delta-cycles, which trigger in zero time. According to the SDL/SystemC co-modeling approach one

execution of the SDL transition is performed in one delta-cycle. Each transition of an SDL process from one state to another can result in scheduling of the new events. There are two ways for events scheduling – signals and timers. Using of signals means that the event should be performed at the current moment of modelling time. Such an event is processed during the next delta-cycles after a delta-delay. The timer expiration is scheduled at another moment of modelling time. So it causes a new event, which is processed when all current time events will be performed [7].

Fig. 1 shows a simple example of the SDL/SystemC co-model structure. This is an example, when two nodes interact with each other through the channel, but one node is implemented in SDL while another node and channel – in SystemC [11].

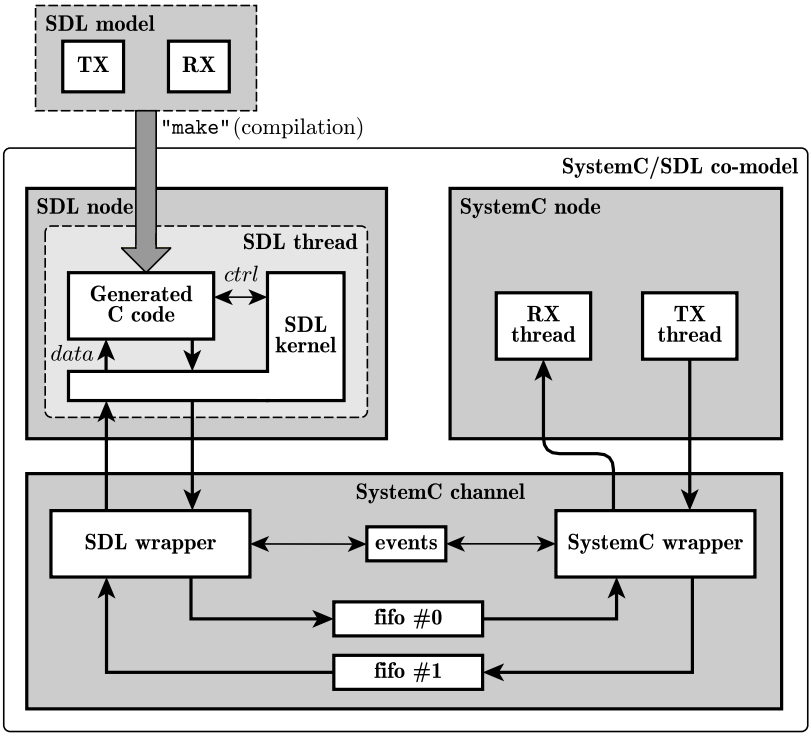


Fig. 1. SDL/SystemC co-modeling example

SDL/SystemC co-modeling approach can be successfully applied for validation of SDL models within the SystemC tester. In this case, the SDL model contains two instances of either a protocol layer or a protocol stack which work independently. Communication is performed through SystemC channels which can operate in accordance with different algorithms for error generation, signal

loss, etc. SystemC test environment is a master component that fully controls the slave SDL system.

The implementation of the tester can be divided into the following general stages taking into account the general SDL/SystemC co-modeling principles:

1. Implementation of an SDL model of a protocol.
2. Implementation of special wrappers for conversion of SDL data types to the SystemC data types and vice versa.
3. Implementation of the SystemC test engine, channel for communication of the nodes and control components.
4. Writing test cases. Test cases are implemented as SystemC components, which work in accordance with different algorithms. By switching between these components, developers can change test scenarios.

The architecture of a co-model is shown in Fig. 2. It includes three main parts – SystemC test control components, an SDL part (SDL model and SDL simulation kernel) and SystemC channels. In Fig. 2 the SDL model is essentially the generated C code which implements the original graphic model.

SDL/SystemC co-modeling approach has been successfully used for validation and testing in such projects as UniPro [12] and SpaceWire-RT [13].

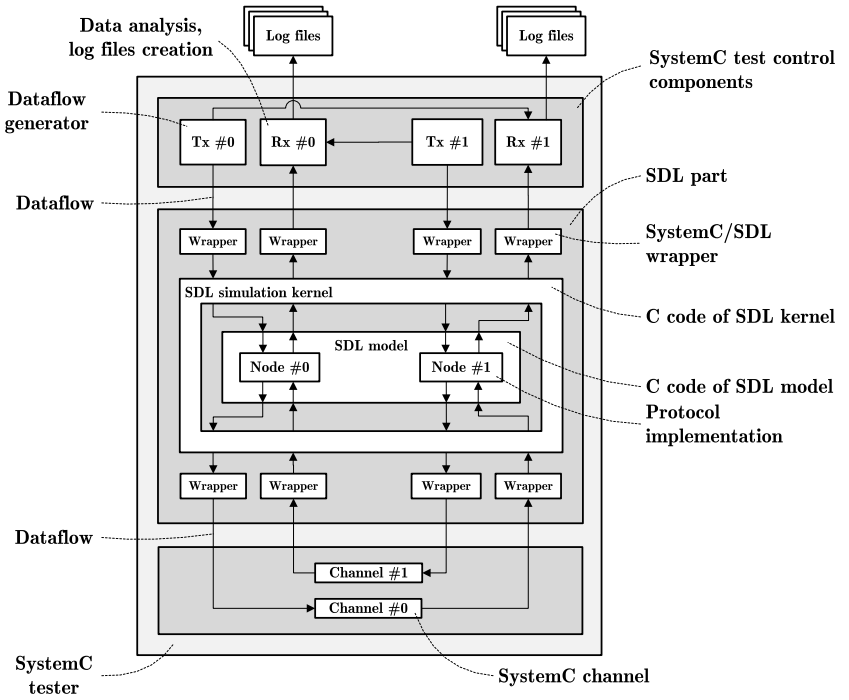


Fig. 2. Testing SDL under SystemC

4 Different Approaches to a Solution

Our goal is to develop an approach that allows creating several instances of the SDL model. Moreover, since we use the IBM Rational SDL Suite, our approach is tool specific. Especially for the SDL/SystemC co-modeling we use the CAdvanced code generator and C source code of the SDL model, which has been generated by it.

Since SystemC is based on C++ and since CAdvanced generates plain C code, there appears a task of combining C and C++ parts of the model. The use of available C++ code generators can probably solve some problems. But in the case of using our codebase, embedding a new tool requires making global changes to our projects.

We consider it is important to understand how the original SDL model is implemented in the C source code and, moreover, what are the source code equivalents for different elements of the SDL language. Another question is how we can reuse the code to have an opportunity to instantiate more than one SDL model. There are not many publications that describe the design of generation of source code from SDL and the principles of communication with an SDL simulation kernel. The paper [14] observes some details of this code generation and the principles of its functionality. Another publication [15] (a thesis) that proposes an integrated design flow for embedded systems, where the author also uses the CAdvanced code generator, describes several mechanisms that are used in SDL Simulator, and how the generated code interacts with the simulation kernel.

Based on the IBM Rational SDL Suite documentation together with [14, 15] we have conducted research in the design of model generated to find the ways for acheving our specific goal.

4.1 Integration of C Code into C++ Environment

We need to localize the SDL model instances in a memory. The most obvious approach is integration of the generated C code of the SDL model and the SDL simulation kernel into a C++ environment for its further operation in the user's code. If integration is possible, then the target library can be developed with the use of different OOP patterns. However, practical application of this approach has shown that this way entails a number of technical problems. Since the generated code of the SDL model is represented by C code that strictly conforms the ANSI C standard, the most complex problem is integration of the C code for operation in the C++ project. Consequently, the significant part of the SDL simulation kernel and generated code of the SDL model should be changed. Therefore, it can be concluded that the implementation of this approach takes a considerable time.

4.2 Code Post-processing

Another way of solving the problem is post-processing of the SDL model generated C code in order to have an opportunity of creating different numbers of

SDL model instances by the use of dynamic memory allocation. The main feature of the CAdvanced code generator is that the implementation of an SDL model represents the hierarchical structure that is called *symbol table* and organized as a tree [15]. The symbol table contains objects that represent SDL entities (system, blocks, processes, signals, etc). These objects are global variables, so static memory allocation is used.

To have an opportunity to create different numbers of SDL model instances, we need to change the memory allocation mechanism from static memory allocation to dynamic memory allocation. This can be done by the code post-processing. Ideally, we need to change the implementation of the CAdvanced code generator, but this is almost impossible as we use an existing industrial tool. On the other hand, it is a well known approach to develop an auxiliary toolchain for existing products.

5 An Approach of SDL Model Instances Integration

5.1 The Library Development Flow

The solution is aimed to develop an environment that allows creating the target library. This library provides an ability to use a different number of SDL model instances in the SystemC user's project and contains both the SDL model and the SDL simulation kernel. The library development flow and library usage in a project is shown in Fig. 3.

These are the steps of the proposed library development flow:

1. **Analysis of requirements** and **implementation of an SDL model**.
2. **Obtaining a PR-model** using the GR-to-PR converter.
3. **Obtaining C code** of the SDL model with use of CAdvanced. The code consists of three parts: a symbol table, which corresponds to the SDL model architecture, a set of initialization functions and a set of PAD (Process Activity Description) functions [10] which implement the behavior of SDL processes.
4. **Code post-processing** of the obtained C code. Generation of initialization functions and patching of some parts of PAD functions.
5. **Building a target library** according to the proposed approach. Creation of the symbol table selector. Development of a user's code interface, which is a set of C++ classes.

Then all the generated source code is compiled and linked, so the user gets a target library 'component.lib'. The implementation of the SDL kernel stays unchanged during the library development flow, but the new functionality for operating with a different number of SDL model instances is added. The user's project operates with the target library and the SystemC library simultaneously. The user interface is intended for using services provided by the library.

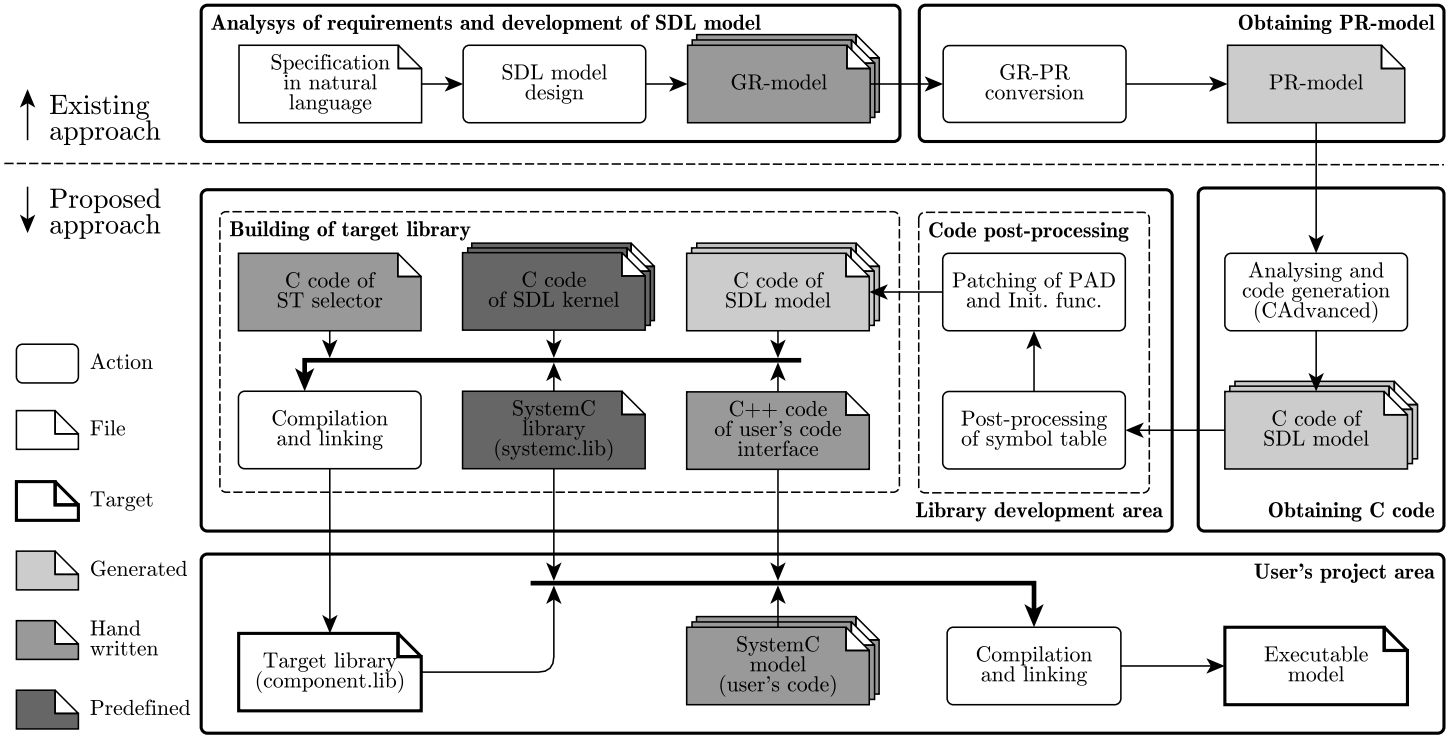


Fig. 3. Library development and usage

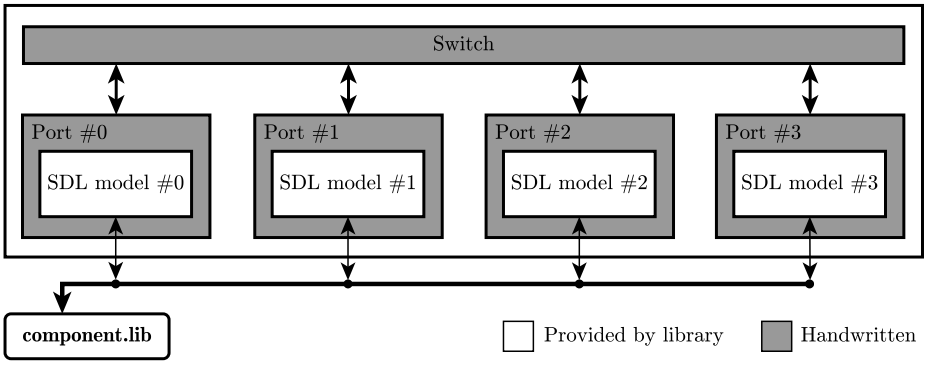


Fig. 4. SystemC model structure

5.2 Application Structure

Let us consider a simple example. Fig. 4 shows the architectural diagram of the SystemC model of SpaceWire MCK-01 switch [16].

The model contains a *Switch* module and four ports connected to four independent SDL model instances. The *Switch* and *Port* modules are implemented in SystemC. According to the proposed approach it is possible to design a switch model, where each port includes the implementation of full protocol stack in SDL. In this case the network layer is implemented in SystemC while the bottom ones – in SDL. The structure of the application implemented in accordance with the proposed approach is shown in Fig. 5.

It consists of the following parts:

1. **The SystemC library**, which includes SystemC kernel.
2. **The SystemC model** implemented by a user.
3. **The target library**, which provides an ability to create a number of different SDL model instances. The library is divided into three parts: the user's model interface, which describes the services for communication between the users SystemC model and the SDL kernel; the SDL kernel, which performs scheduling of the generated SDL model and the SDL model itself. For communication with C++ classes a basic `xInEnv/xOutEnv` [10] mechanism is used. Implementation of the SDL model has four parts:
   (a) A set of PAD functions. These functions implement behavior of SDL model processes.
   (b) A selector of a symbol table (ST).
   (c) A set of SDL model instances. Each of them has its own symbol table, but all instances have a set of common PAD functions.
   (d) A set of initialization functions. These functions are used for instantiation of each new symbol table with the use of the dynamic memory allocation.

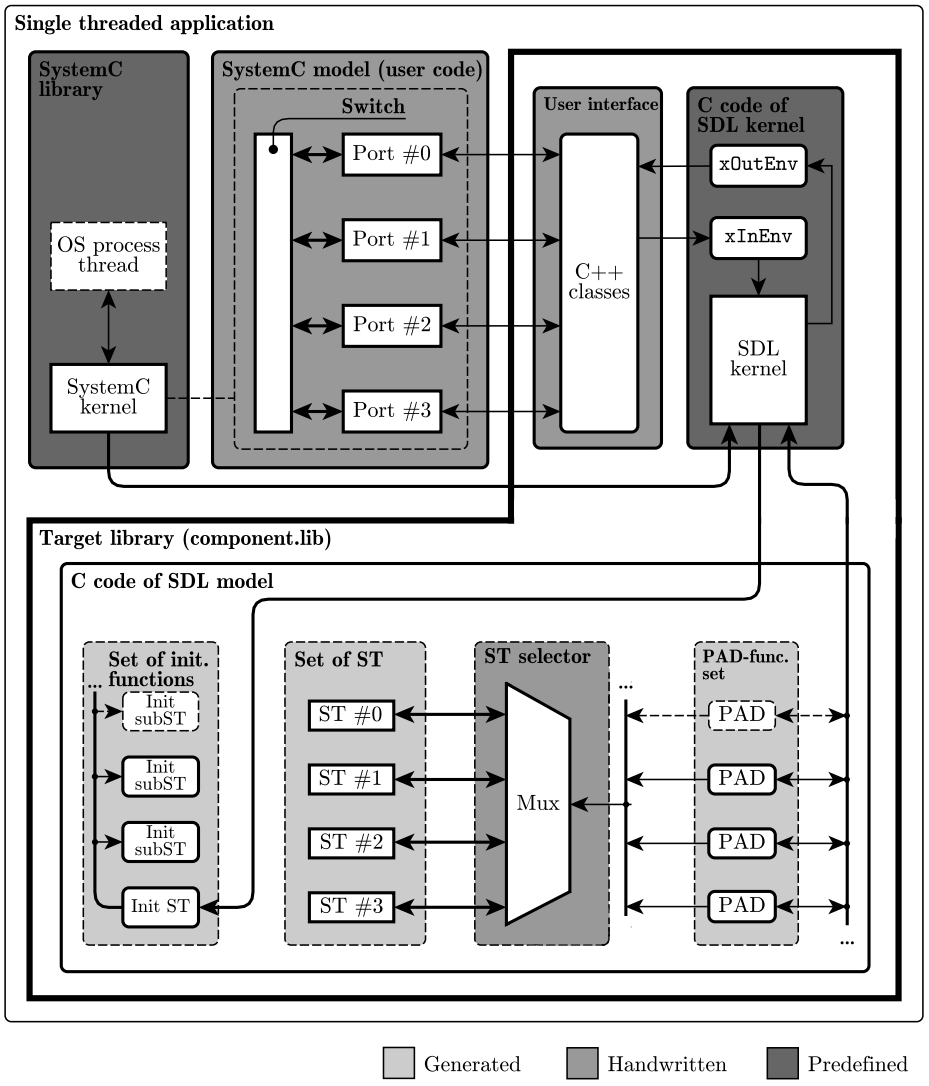


Fig. 5. Application structure

User's SystemC model is a single threaded application, which is controlled by the SystemC kernel. The Switch module with its ports communicates with the SDL kernel via user's interface. The SDL kernel is responsible for scheduling of SDL model processes. The kernel calls different PAD functions and each PAD function chooses an SDL model instance by means of ST selector. ST selector uses a symbol table identifier, which is generated by a user's C++ object which represents the SDL model (for example *Port #0*). The ST selector clearly indicates the required SDL model instance.

6 Main Principles of Model Memory Organization and Management

As an example we take the simple SDL model to clearly explain the proposed approach. The SDL model, which is taken as the basis for the example, consists of one block communicating with the environment by means of signals 'sig.req' and 'sig.rsp'. This block contains only one process, which operates with the same signals.

First we should generate the C code from the SDL model. It is done by means of CAdvanced code generator. This C code contains a set of interacting data structures and each of them could be a huge hierarchical tree. These structures represent the SDL model symbol table. Thereafter, it is possible to convert the generated C code to the XML. The XML representation contains 226 nodes (a node comprises a C data structure and its fields) with 251 connections between them for such a simple example.

According to the proposed approach the C code should be divided into a number of post-processing steps. Initialization functions were generated and PAD functions were patched. Initialization functions are called each time a new SDL model instance is initialized. In the case when the original SDL model was designed with use of packages, the CAdvanced generates an implementation of each package and places them into separated source file. Each package has its own initialization function. Therefore, we need to post- process all source files to have opportunity of build the symbol table in memory using dynamic memory allocation. After the initialization each SDL model instance is separately stored in the heap. Since initialization is performed with the use of the same function, which does not returns any value, it is not possible to get access to all symbol tables (as each new instance is initialised by a consecutive calling of initialization functions). To solve this problem some special nodes of the SDL model symbol table are added to special arrays. These arrays are used to determine the necessary nodes of symbol table of SDL model instance while sending signals from environment [1] to SDL model or sending signals from SDL model to an environment. Communication mechanisms are shown in Fig. 6.

The heap stores two symbol tables of the SDL model. A set of arrays, which are global variables, is stored in a data segment and contains pointers to signals, channels and environment processes, since every SDL model instance has its own environment. The SDL kernel extracts the first process from the *ready queue* [10] and calls associated PAD function.

A PAD function must obtain information about SDL model instance before it can send a signal to any process. It is done by using of a multiplexer, which is able to choose an instance depending on the information from arrays. The signal array, the channel array and the environment processes array are used for a required SDL model instance choice. Multiplexer does it by using traverse of hierarchical part of symbol table. The system identifier is stored on a system level of the hierarchy of SDL entities [1]. Such an identifier is associated with each new SDL model during the initialization stage.

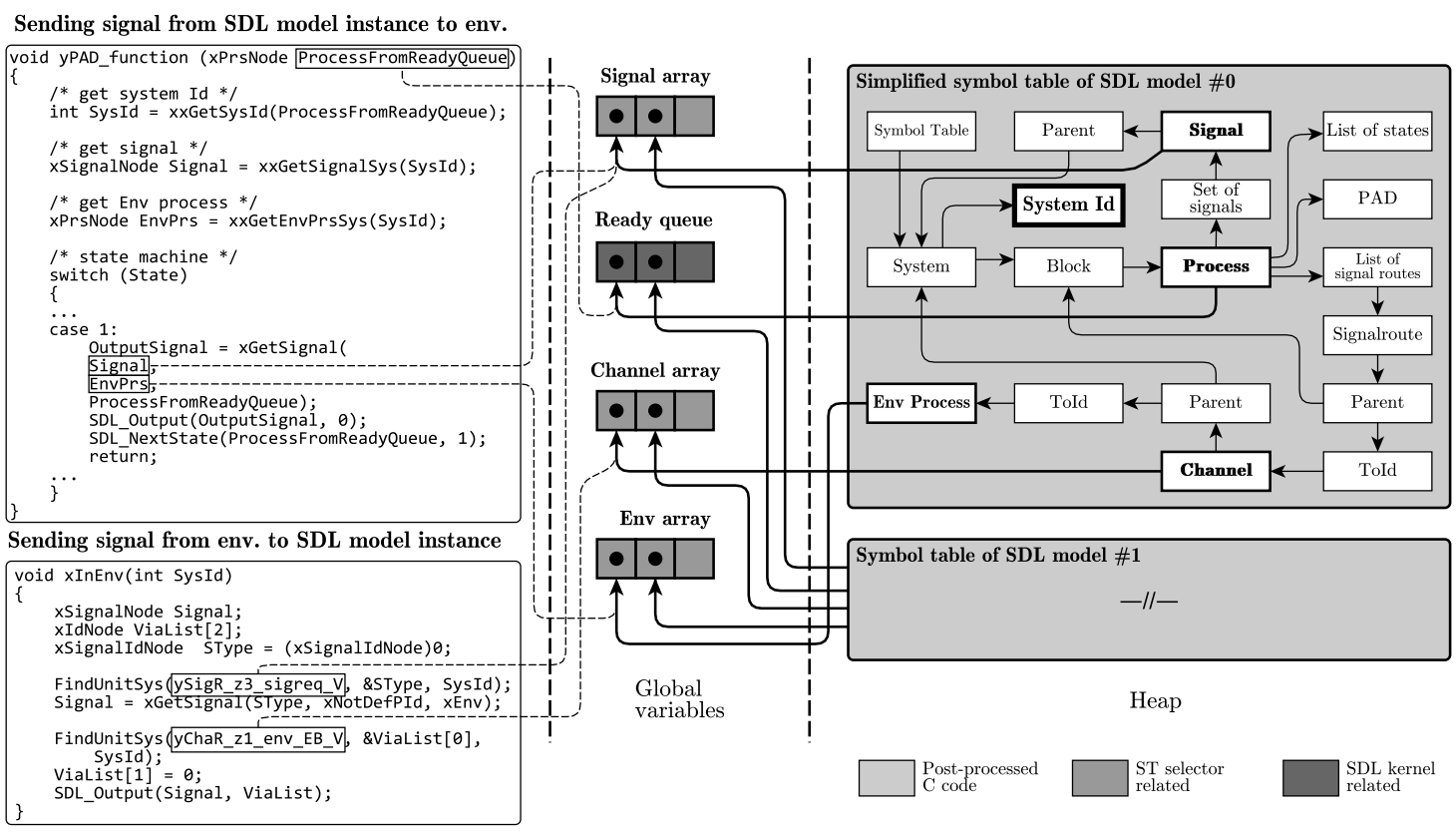


Fig. 6. Communication mechanisms

So during the execution of a PAD function and sending a signal to environment by `xOutEnv` function or to other process by `SDL_Output` function, it uses the multiplexer to determine a symbol table of the SDL model instance. When signal is sent from environment (from user's SystemC model) to SDL model instance by `xInEnv` function, it uses the multiplexer which performs search for a channel and a signal in arrays for identification of the SDL model instance.

7 An Example of the Approach Application

This example gives more details of the proposed approach and shows how the SystemC developer can use it in his project. Let us assume that we need to create a network model in SystemC and also we need to use it for an exploration of non-functional properties of a protocol while the SDL model of a protocol has already been implemented. To simplify the SDL model we use the same SDL model as we used in section 6. The example is shown in Fig. 7.

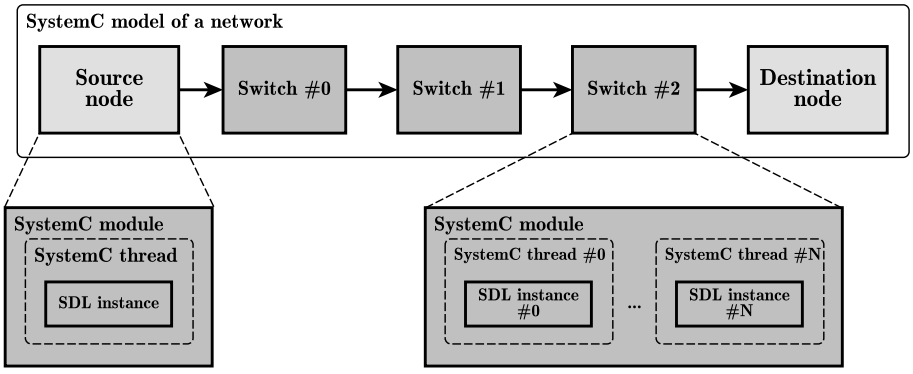


Fig. 7. Example of the approach application

The SystemC model includes the *Source* node, three switches and the *Destination* node. The *Source* node is responsible for data generation while the *Destination* node is responsible for its reception. Each switch contains the SystemC module, which includes a number of SystemC threads, each of which corresponds to an independent instance of the SDL model. All these instances are created by the user in C++. The library controls all of them.

A fragment of source code of the SystemC module of a *Switch* is shown in Listing 1.1. This fragment shows the part of source code of the `switch_module` class and `sdl_model` class. Constructor of the `switch_module` class is responsible for initialization of a module and creation of a corresponding new instance of the SDL model. The `sdl_model` class contains functions which form the user's interface and which are used in the `switch_module` class. The `sdl_model_thread` firstly waits for a request event. After the event has been generated the thread

```
1  /**** part of user's interface ****/
2  class sdl_model {
3  public:
4      // initialization of the new instance
5      sdl_model ();
6
7      // function for sending sig.req to the instance
8      void send_sig_reg ();
9      ...
10 };
11
12 /**** part of user's code ****/
13 class switch_module : ... {
14 public:
15     ...
16     void sdl_model_thread ();
17 private:
18     sc_event sig_req_event;
19 };
20
21 // switch module ctor
22 switch_module::switch_module(sc_module_name name) :
       sc_module(name){
23     // thread creation and event setting
24     SC_THREAD (sdl_model_thread);
25     sensitive << sig_req_event;
26
27     // creation of a new instance of the SDL model
28     sdl_model_instance = new sdl_model;
29     ...
30
31 }
32
33 // switch module thread
34 void switch_module::sdl_model_thread (){
35     while(1){
36         wait(sig_req_event); // waiting for input event
37
38         // sending sig.req signal to the instance
39         sdl_model_instance->send_sig_reg();
40     }
41 }
```

Listing 1.1. Part of SystemC module source code

handles it and the signal is sent to SDL model instance using function call. The function is provided by the library and it is one of a set of functions of the user's interface. Therefore, the instance can be used in such a manner as if it is a SystemC component. Thus, SystemC developer can work with any instance of the SDL model of a protocol not knowing anything about its implementation.

The proposed approach gives an opportunity to focus on implementation of the SystemC model of a network comprising hundreds of nodes rather than on implementation of the SDL model of a protocol.

8 Conclusion

This paper gives an overview of the problem of integration of different numbers of SDL model instances into the SystemC project. The paper proposes and explains the elaborated approach. The SDL model is encapsulated inside a self-contained C++ class and could be easily instantiated. During instantiation of every new copy of the SDL model, it is placed on heap. For this opportunity, the generated source code of the SDL model has to be post-processed and the memory allocation mechanism should be changed from the original static memory allocation to a dynamic memory allocation. In addition, we provide an example of the successful application of the approach. The proposed approach is expected to reduce the project work effort and help in achieving a better quality of the simulation results. However, there is still a number of open questions and tasks for future work: definition of rules for code post-processing, memory management, proving of a model implementation and behavior correctness.

The future work would be mostly focused on the creation of a special tool. This tool is planned to be applied during the SpaceWire-RT standard model implementation and validation.

References

1. International Telecommunication Union: Recommendation Z.100 (12/11) Specification and Description Language - Overview of SDL-2010, http://www.itu.int/rec/T-REC-Z.100/en
2. Mitschele-Thiel, A.: Systems Engineering with SDL: Developing Performance-Critical Communication Systems. John Wiley & Sons (2001)
3. Institute of Electrical and Electronics Engineers: IEEE Standard for Standard SystemC Language Reference Manual, http://standards.ieee.org/findstds/standard/1666-2011.html
4. Black, D.C., et al.: SystemC: From the Ground Up. Springer (2010)
5. Grötker, T., Liao, S., Martin, G., Swan, S.: System Design with SystemC. Kluwer Academic (2002)
6. Turner, K.J.: Using Formal Description Techniques – An Introduction to Estelle, LOTOS and SDL. John Wiley & Sons I (1993)
7. Balandin, S., et al.: Co-Modeling of Embedded Networks Using SystemC and SDL. International Journal of Embedded and Real-Time Communication Systems 2(1), 24–49 (2011), http://www.igi-global.com/article/modeling-embedded-networks-using-systemc/51648

8. Gillet, M.: Hardware/software co-simulation for conformance testing of embedded networks. In: Finnish-Russian University Cooperation Program in Telecommunications seminar
9. Olenev, V., et al.: SystemC and SDL Co-Modelling Methods. In: Proceedings of 6th Seminar of Finnish-Russian University Cooperation in Telecommunications Program, pp. 136–140. State University of Aerospace Instrumentation (2009)
10. IBM Rational. IBM Rational SDL Suite User's Manual. IBM Rational (2009)
11. Stepanov, A., et al.: SystemC and SDL Co-Modelling Implementation. In: Proceedings of 7th Conference of Finnish-Russian University Cooperation in Telecommunications Program, pp. 130–137. State University of Aerospace Instrumentatio (2010), http://www.fruct.org/publications/fruct7/files/Ste.pdf
12. UniPro protocol stack by Mobile Industry Processor Interface Alliance, http://mipi.org/specifications/unipro-specifications
13. SpaceWire-RT project, http://spacewire-rt.org
14. Haroud, M., Biere, A.: SDL Versus C Equivalence Checking. In: Prinz, A., Reed, R., Reed, J. (eds.) SDL 2005. LNCS, vol. 3530, pp. 323–338. Springer, Heidelberg (2005)
15. Dietterle, D.: Efficient Protocol Design Flow for Embedded Systems. Brandenburg University of Technology (2009), http://systems.ihp-microelectronics.com/uploads/downloads/diss_dietterle.pdf
16. SpaceWire switch MCK-01, http://multicore.ru/index.php?id=850

Author Index