Workshop on Models and Model-driven Methods for Enterprise Computing (3M4EC 2008)

Marten van Sinderen, João Paulo Andrade Almeida,
Luís Ferreira Pires and Maarten Steen (Eds.)

Munich, Germany, September 16, 2008
http://www.inf.ufes.br/~jpalmeida/3m4ec2008

Proceedings

Sponsored by

In conjunction with:

The 12th IEEE International EDOC Conference
The Enterprise Computing Conference (EDOC 2008)
15-19 September 2008, Munich, Germany
http://www.edocconference.org

Editors

Marten van Sinderen
Centre for Telematics and Information Technology
University of Twente
PO Box 217
7500 AE Enschede, the Netherlands
m.j.vansinderen@ewi.utwente.nl
http://wwwhome.ewi.utwente.nl/~sinderen/

João Paulo Andrade Almeida
Federal University of Espírito Santo
Av. Fernando Ferrari, s/n
Departamento de Informática – CT-VII
Vitória, ES, Brasil 29060-970
jpalmeida@ieee.org
http://www.inf.ufes.br/~jpalmeida

Luís Ferreira Pires
Centre for Telematics and Information Technology
University of Twente
PO Box 217
7500 AE Enschede, the Netherlands
l.ferreirapires@ewi.utwente.nl
http://wwwhome.ewi.utwente.nl/~pires/

Maarten Steen
Telematica Instituut
PO Box 589
7500 AN Enschede, the Netherlands
Maarten.Steen@telin.nl
http://www.telin.nl

Enschede, the Netherlands, September 2008
CTIT Workshop Proceedings Series WP08-04
ISSN 1574-0846

Table of Contents

List of Authors

Gašević, Dragan	11
Heerink, Lex	1
Iacob, Maria-Eugenia	1
Milanović, Milan	11
Poernomo, Iman	23
Steen, Maarten	1
Umarov, Timur	23
Wagner, Gerd	11
Weber, Gerald	35

Program Committee

Colin Atkinson	University of Mannheim, Germany
Mariano Belaunde	France Telecom R&D, France
Remco Dijkman	Eindhoven University, The Netherlands
Jeff Gray	University of Alabama at Birmingham
Giancarlo Guizzardi	Federal University of Espírito Santo, Brazil Laboratory for Applied Ontology, ISTC-CNR, Italy
Roy Grønmo	SINTEF, Norway
Slimane Hammoudi	ESEA, France
Patrick Hung	University of Ontario Institute of Technology, Canada
Maria-Eugenia Iacob	University of Twente, The Netherlands
Peter Linington	University of Kent, UK
Oscar Pastor	Universidad Politecnica de Valencia, Spain
Alfonso Pierantonio	University of L'Aquila, Italy
Dick Quartel	Telematica Instituut, The Netherlands
Richard Soley	Object Management Group, USA
Antonio Vallecillo	University of Málaga, Spain
Branimir Wetzstein	University of Stuttgart, Germany

Preface

Recent developments in metamodeling and model transformation techniques have led to increasing adoption of model-driven engineering practices. The increase in interest and significance of the model-driven approach has also accelerated its application in the development of large (distributed) IT systems to support (collaborative) enterprises. Shifting attention from source code to models enables enterprises to focus on their core concerns, such as business processes, services and collaborations, without being forced to simultaneously consider the underlying technologies. Different concerns are typically addressed by different models, with transformations between the models and ultimately to the source code. Although the model-driven approach offers theoretical benefits for the development, maintenance and evolution of enterprise computing systems, a number of issues for the practical application of the approach still exist. In order to solve these issues, further advances in models and model-driven methods (design concepts, languages, metamodels, profiles and specification frameworks) are necessary.

The International Workshop on Models and Model-driven Methods for Enterprise Computing (3M4EC) aims at helping the convergence of research on model-driven development and practical application of the model-driven approach in the area of enterprise computing. The workshop addresses questions with respect to the requirements on, concepts for, properties of, and experience with models and model-driven methods for enterprise computing in general and in specific application domains. Special attention is given to the application of the model-driven approach to enterprise service-oriented architecture computing.

This volume contains the proceedings of the first edition of the workshop, 3M4EC 2008, held on September 16, 2008, in Munich, Germany, in conjunction with the 12th IEEE International EDOC Conference – The Enterprise Computing Conference (EDOC 2008). Four papers were selected for oral presentation and publication, based on a thorough review process, in which each paper was reviewed by several experts in the field.

We would like to take this opportunity to express our gratitude to all people who contributed to the 3M4EC 2008 workshop. We thank the authors for submitting content, which resulted in valuable information exchange and will certainly lead to stimulating discussions during the workshop, and we thank the reviewers for providing useful feedback to the submitted content, which undoubtedly helped the authors to improve their work. Finally, we appreciated the possibility to have 3M4EC being held in conjunction with the EDOC 2008 conference, and we are grateful for the support we received from the EDOC 2008 organization.

Munich, Germany, September 2008

Marten van Sinderen, João Paulo Andrade Almeida, Luís Ferreira Pires, Maarten Steen
3M4EC Organizers

Reusable Model Transformation Patterns

Maria-Eugenia Iacob
University of Twente
m.e.iacob@utwente.nl

Maarten W. A. Steen
Telematica Instituut
maarten.steen@telin.nl

Lex Heerink
Telematica Instituut
lex.heerink@telin.nl

Abstract

This paper is a reflection of our experience with the specification and subsequent execution of model transformations in the QVT Core and Relations languages. Since this technology for executing transformations written in high-level, declarative specification languages is of very recent date, we observe that there is little knowledge available on how to write such declarative model transformations. Consequently, there is a need for a body of knowledge on transformation engineering. With this paper we intend to make an initial contribution to this emerging discipline. Based on our experiences we propose a number of useful design patterns for transformation specification. In addition we provide a method for specifying such transformation patterns in QVT, such that others can add their own patterns to a catalogue and the body of knowledge can grow as experience is built up. Finally, we illustrate how these patterns can be used in the specification of complex transformations.

1. Introduction

OMG's Model-Driven Architecture (MDA) ([9], [6]) has emerged as a new approach for the *design and realisation of software* and has eventually evolved in a collection of standards that raise the level of abstraction at which software solutions are specified. The central idea is that computational independent models (CIMs), platform independent models (PIMs) and platform specific models (PSMs) – defined at different levels of abstraction – are derived (semi-) automatically from each other through *model transformations*. Model transformations are thus a crucial element in OMG's vision on MDA. Transformations relate the different abstractions used in a model-driven development scenario. Model-to-model (M2M) transformations relate CIMs to PIMs and PIMs to PSMs, while Model-to-Text (M2T) transformations relate the PSMs to code. OMG has recently adopted standard languages for the specification of model transformations, for which a number of implementations are already available. The availability of these transformation engines, in addition to the existing metamodelling technology, brings us a lot closer to the realization of the MDA vision. Modelling engineers are now able to define their own Domain-Specific Languages (DSLs) and transformations between them and existing languages.

Since the technology for executing transformations written in high-level declarative specification languages (such as those included in the QVT standard) is of very recent date, we observe that there is very little knowledge available on how to write such declarative model transformations. This led us to the conclusion that there is a need for a body of knowledge concerning the emerging discipline of transformation engineering.

In this paper we aim to make an initial contribution to this emerging discipline. Recently we have had the opportunity to experiment with implementations of both the QVT Core language (from Compuware) and of the QVT Relations language (from IKV++). Based on these experiences we propose a number of useful problem-solution patterns, similar to the well-known design patterns in software development. In addition we provide a method for documenting and specifying such reusable transformation patterns, such that others can add their own patterns and the body of knowledge can grow as experience is built up. For this purpose we have recently started a Wiki catalogue [10] where transformation patterns can be documented and discussed.

The paper is organised as follows. In Section 2 and Section 3 we discuss briefly the QVT model transformation specification standard and a few modelling languages we use in this paper. In Section 4 the issue of documenting transformation patterns is addressed. Section 5 consists of a catalogue of model transformation patterns we believe to be relevant in the context of model-driven development. Each pattern is described using a template that includes (for illustration purposes) a pattern application example. In Section 6 we demonstrate how the patterns can be used combined by specifying a transformation for state chart models. Finally, Section 7 summarises our conclusions and gives some pointers to future work.

2. The QVT transformation languages

In order for design patterns to be understood and useable by a wide audience, they should be expressed in a well-known, preferably standardized language. QVT (Query/View/Transformation) provides such languages for M2M transformation specification. QVT actually defines three different transformation languages: *Relations, Core* and *Operational Mappings*. Relations and Core are both declarative languages at two different levels of abstraction, with a mapping between them. We briefly present the Relation language below that has been used for specification purposes throughout this paper. For a complete definition of these languages we refer the interested reader to the standard specifications [8].

The QVT Operational language extends both Relations and Core and provides a way of specifying transformations imperatively. As we focus on declarative transformation specification, we will not discuss the Operational language further in this paper. OMG has recently also approved the MOF Model-to-Text standard for specifying transformations from MOF models to text (i.e., code). However, M2T transformations are of a completely different nature and therefore also fall outside the scope of this paper.

In the **QVT Relations** language transformations are specified by defining the relations that should hold between source and target domains. Transformation rules are described in terms of relations that define a mapping between source and target elements and can be constrained in the when and where clauses. Only model elements that satisfy the constraints will be related. Such constraints typically deal with the properties of the model element, such as attributes and associations to other elements. The when-clause specifies a precondition. Only when all conditions in this clause evaluate to true the relation between the specified domains is established. The where-clause specifies a postcondition. Once the relation is established then the conditions specified here should be enforced to hold. When a domain is marked as **enforced**, the engine may create or update that domain in order to establish the relation.

3. Modelling languages

Before addressing the main topic of this paper - the *transformation rule patterns* - we briefly describe the experimental setting in which our results have been devised. The following modelling languages that have served as source and target languages in our transformation pattern specifications:

The **shape language** is a simple, purely syntactical language that has been defined in order to illustrate the model transformation patterns. It does not require any prior knowledge and it basically has only two concepts: simple shape and arrow. There are three types of simple shapes: square, circle and triangle. Furthermore, the Shape language contains a grouping concept called Block used to express hierarchy. A block may contain simple shapes and other blocks. Each shape model should have a unique root element, which is an instance of RootBlock, a specialization of Block. To express relations between simple shapes and blocks the Arrow concept is used. The Shape metamodel is given in the Figure 1.

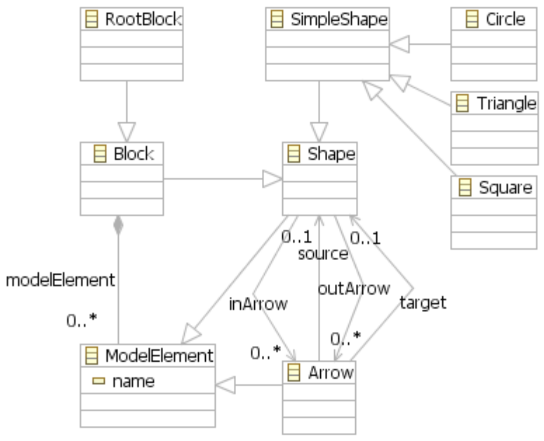


Figure 1. Shape language metamodel

In order to illustrate the transformation patterns proposed in this paper we have used well known diagramming notations, namely the **UML class, activity and statechart diagrams** (for the complete specifications see [7]). The used statechart metamodel can be found in the Figure 2.

4. Transformation design patterns

Since the publication of "Design Patterns" by Gamma et al. [4], patterns are well known in software engineering. Patterns describe which problems software engineers can encounter, the context in which such problems may appear, and a general solution to them. Analogously we propose to start a collection of reusable design patterns for specifying model transformation. A transformation design pattern, or *transformation pattern* for short, is then a reusable solution to a general model transformation problem.

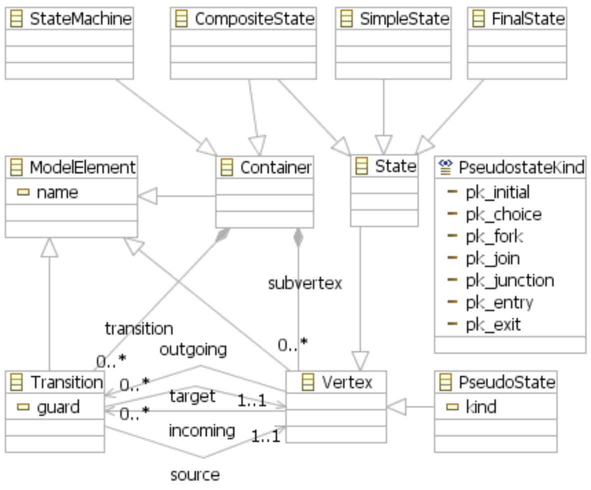


Figure 2. Statechart diagram metamodel

The need for transformation patterns emerged almost immediately after we first started writing model transformations in QVT. Transformations are often very similar. An existing transformation specification is often a good starting point for a new one. Unfortunately, the collection of existing and well-documented transformations is still very small. We also noticed that the same transformation can often be specified in subtly different ways. On the surface it seems to be just a matter of style, but such a different 'style' can have great consequences for performance, applicability and reusability of the transformation. Finally, our first solution to a particular transformation problem often was not entirely correct and had to be revised several times. A library of reusable transformation patterns should enable engineers to get it right more quickly.

4.1. Transformations, transformation rules and rule patterns

Transformations, transformation rules, transformation patterns, rule patterns are just a few concepts which are used with different meanings and sometime interchangeably in the literature. For example, in [5] a definition is given for a transformation pattern which corresponds to what we call a transformation definition. Somewhat similar definitions to the ones we propose are given by [3]. Another interesting view on transformation patterns is that taken by the project Modelware [1] that considers that a transformation pattern (as general repeatable solution to a commonly-occurring model transformation design problem) is not a finished design that can be transformed directly into a transformation specification. Although [1] proposes a catalogue of transformation patterns, their approach is different in two respects. Firstly, Modelware does not rely on the QVT standard for the specification of proposed patterns. Instead, the hybrid imperative/declarative ATL language is used. Secondly, the patterns included in [1] do not overlap with those proposed in this paper.

Therefore, before discussing specific transformation patterns as mentioned before, we feel compelled to provide further clarifications concerning the semantics we have attributed to these concepts and the relations between them (as depicted in Figure 3). In the remainder of this paper the following definitions will be used.

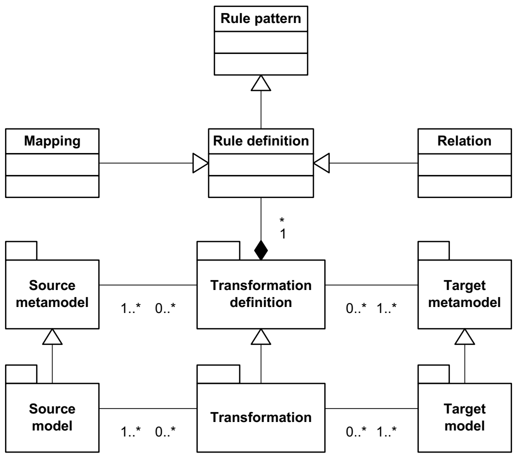


Figure 3. Model transformation concepts and relations between them

A *transformation definition* is a formal specification that consists of a set of rule definitions. A *rule definition* is a formal specification in the form of a *mapping* (in the sense of the QVT - Core language) or of a *relation* (in the sense of the QVT - Relations language). In its simplest form (and in line with the MDA), a *model transformation* is the process of converting a *source model* that conforms to a *source metamodel* into a target model that conforms to a *target metamodel*, using an existing transformation definition between the two metamodels. When a source model is transformed into the target model the transformation definition prescribes the manner in which the different rule definitions that are included in the transformation definition are "executed". In this paper we argue that rule definitions can be created by instantiating so called rule patterns. More specifically, we regard a *rule pattern* as a generic (possibly parameterized) formal specification that describes at a higher level of abstraction a whole class of recurring rule definitions.

4.2. Documenting transformation patterns

A design pattern names, abstracts, and identifies the key aspects of a common design structure, such that it can be reused and applied over and over again in creating new designs. According to [1] and [4], a pattern description should contain the following four essential elements: the pattern name, a description of the problem and the contexts in which it is applicable, the solution to the problem, and the consequences of using the pattern. In addition, pattern descriptions should provide an example to clarify the provided solution.
Likewise, we use a fixed template for documenting the transformation patterns, consisting of the following elements: the **name** of the pattern, the **goal** of the pattern, **motivation** for the pattern, describing the class of problems that the pattern solves, **specification** of the solution using the QVT Relations language, an **example** in which the pattern is applied and considerations regarding the pattern's **applicability**.

5. A catalogue of rule patterns

In this section, we document a number of transformation patterns using the template described above. These are: Mapping, Refinement, Abstraction, Duality and Flattening.

5.1. The Mapping pattern

Goal: Establish one-to-one relations between elements from the source model and elements from the target model.
Motivation: Mapping is the most common and straightforward transformation problem. It occurs when source and target models use different languages or syntax, but otherwise express more or less the same semantics. This pattern is used to a greater or lesser extent in virtually any transformation.
This is the most basic transformation pattern. Typical examples of transformation rules that are based on this pattern are 1-to-1 model transformation rules. It is in general bidirectional (unless different concepts from the left domain are mapped onto the same concept in the right domain). All other transformation patterns use/include this pattern.
Specification:

```
top relation XYMapping {
        nm: String;
        enforce domain left x: X {
        context = c1 : XContext {},
                name = nm };
        enforce domain right y: Y {
                context = c2 : YContext {},
                name = nm };
        when {
                ContextMapping(c1,c2);
        }
}
```

This rule specifies that some element x of type X is related to some element y of type Y, whenever their respective contexts are related by ContextMapping and their names are equal. When the respective model elements have more properties than a context and a name, these should also be mapped. Consider for example the case where the model elements to be mapped represent associations or relationships between other model elements, their sources and targets. The pattern for this case is specified below:

```
top relation RelationshipMapping {
        nm: String;
        enforce domain left a: A {
                context = c1 : AContext {},
                name = nm,
                source = as : AS {},
                target = at : AT {}
        };
        enforce domain right b: B  {
                context = c2 : BContext {},
                name = nm,
                source = bs : BS {},
                target = bt : BT {}
        };
        when {
                ContextMapping(c1,c2);
                ElementMapping(as,bs);
                ElementMapping(at,bt);
        }
}
```

Example: For an example of mapping pattern instance one may refer to the `relation TransitionMapping` in Section 6. Besides, we have applied this pattern to relate Circles to Squares in the Shape language. The complete specification of this transformation can be downloaded from our Wiki catalogue [10].
Applicability: The mapping pattern can be used to:

- translate a model from one syntax into another syntax, e.g. from ecore to XML, or from UML to Java;
- relate concepts one-to-one in source and target model.

5.2. The Refinement pattern

The refinement pattern is the key design pattern in stepwise refinement, which is a method to create lower level (or: concrete) models from models from higher level (or: abstract) models in a number of successive refinement steps. Refinement is a key ingredient of MDA, which advocates the realization of software systems through systematic stepwise refinement from models. Depending on the subject, different refinement types can be distinguished, e.g., relation refinement and node refinement.

Relation refinement pattern

Goal: To obtain a more detailed target model by refining an edge to multiple, possibly interrelated, edges.
Motivation: Relation refinement is typically used to detail steps (which are often modelled as edges) into sub steps. An example is e.g., by adding process steps to an existing UML activity diagram.
Specification: In relation refinement an edge is refined to (a set of) edges, possibly interleaved with nodes. The corresponding pattern is characterized by a single relation mapping on the left and multiple relation and/or node mapping on the right. The pattern for relation refinement is straightforward, and closely resembles the Mapping Pattern. The specification below demonstrates the mapping of an edge e1 to an edge-node-edge pattern.

```
top relation RelationRefinementMapping {
        n : String;
        enforce domain left e1 : Edge {
                name = n,
                context = c1 : Context {},
                source = s_left : Node {},
                target = t_left : Node {}
        };
        enforce domain right im_node {
                context = c2 : Context {}
        -- an intermediate node
                };
-- potentially more nodes and edges
        enforce domain right e2 : Edge {
                source = s_right : Node {},
                name = s_right.name + '_to_' +
im_node.name,
                context = c2,
                target = im_node
        };
        enforce domain right e3 : Edge {
                target = t_right : Node {},
                name = im_node.name + '_to_' +
t_right.name,
                block = c2,
                source = im_node
        };
        when {
                ContextMapping(c1,c2);
                ElementMapping(s_left,s_right);
                ElementMapping(t_left,t_right);
        }
}
}
```

Example: An example of relation refinement in the Shape language is the refinement of any Arrow into an Arrow-Square-Arrow combination. The corresponding specification in QVT Relations can be downloaded from our Wiki catalogue [10].

Node refinement pattern

To obtain a more detailed target model by refining a node to multiple, possibly interrelated, nodes a node refinement pattern (similar to the relation pattern) has been documented. However due to space limitations has not been included in this paper, but can be found on our Wiki catalogue [10]. Node refinement is used to provide more detail to a node. For example, an UML class diagram that leaves the methods and attributes unspecified can be refined to class diagrams that do specify methods and attributes. Another example is to refine a super state in a hierarchical statechart to several interrelated sub-states.

5.3. The Node Abstraction pattern

Goal: Abstracts from nodes in the source model while keeping the incidence relations of these nodes.
Motivation: The node abstraction pattern removes specific nodes from the source model to create a target model whilst preserving the incidence relations. The node abstraction pattern can be used to abstract from specific information from models. The specification below shows a simplified node abstraction pattern that abstracts from a node X and produces an edge between the incidences. It is assumed that source and target have the same metamodel, that node X is a subtype of the abstract type Node, that each node contains references to its incidence edges, and that each edge contains references to its source and target nodes. The pattern below can only handle sequence of X of length 1, multiple in-sequence occurrences of X cannot be handled.
Specification:

```
top relation Node_X_Abstraction {
        enforce domain left s1 : X {
                inEdge = e_in : Edge {
                        name = na_in : String;,
                        source = ss1 : Node {}
                },
                outEdge = a_out : Edge {
                        name = na_out,
                        target = tt1 : Node{}
                }
        };
        enforce domain right a : Node {
                name = na_in + na_out,
                source = ss2 : Node {},
                target = tt2 : Node {}
        };

        when {
                NodeMapping(ss1,ss2);
                NodeMapping(tt1,tt2);
        }
}
```

Example As node abstraction is quite intuitive we do not provide a code fragment of node abstraction. However, specification of example transformations can be downloaded from our Wiki catalogue [10].
Applicability: Remove model elements from models, for example, remove processes that conform to certain criteria from a process diagram.

5.4. The Duality pattern

Goal: Given a model, to generate its semantic dual.
Motivation: Various modelling languages exist that rely on the (acyclic) directed graph formalism to represent dynamic behaviour (e.g., Petri nets, BPMN and UML statechart diagrams, sequence diagrams, collaborations diagrams and activity diagrams). Nevertheless the semantics attributed to nodes and arrows in these graph-like models differs. There are roughly two main categories of such languages:

- languages that focus on modelling the procedural flow of activities that make up a larger activity, namely a process - in this case vertices generally represent (branching, assembling) activities, while arrows depict causality relations between activities (e.g., BPMN, UML activity diagrams);
- languages that focus on modelling the flow of control from state to state for a particular object undergoing a process - in this case a vertex generally represent one state of that object, while an arrow depict the transition from one state to the other (i.e., indicating that the object being in the first state will enter the second state as a result of reacting to discrete events; e.g., Petri nets, UML statechart diagrams).

Defining transformations between modelling languages that belong to these two different categories requires the application of what we will refer to as *duality pattern* (explained in more detail in the sequel). This pattern is based on the dual character of these two types of languages. More specifically, an activity (in the sense of the first category of languages) can be seen as the procedure that leads to a state change of the object(s) undergoing a process, that is a transition in the sense of the second type of languages, while a causality relationship may be interpreted as the moment when the object(s) have reached a certain state as a result of an activity's completion, which makes possible the initiation of the subsequent one(s). In other words, the duality rule pattern will map vertices from the source model onto arrows in the target model and arrows from the source model onto vertices in the target model. However, it should be noted that the mapping of branching/assembling nodes deserves special consideration.
Specification: Our transformation strategy is as follows: All Arrows on the left are related Nodes on the right using the mapping rule pattern, as indicated below:

```
top relation ArrowNodeMapping {
        nm: String;
        enforce domain left a: Arrow {
                context = c1: AContext {},
                name=nm
        };
        enforce domain right v: Vertex {
                context = c2: VContext {},
                name=nm
        };
        when {
                ContextMapping(c1, c2);
        }
}
```

Rules must be defined for relating a node on the left with one or more arrows on the right for each of the following cases:

- a node on the left, having an incoming arrow e1 and an outgoing arrow e2, is related to an arrow a on the right if e1 has been related to the source of a and e2 to the target of a.

```
top relation NodeArrowMapping {
        nm: String;
        enforce domain left v:Vertex {
                context = c1: NContext{},
                incoming = e1: Arrow {},
                outgoing = e2: Arrow {},
                name = nm
        };
        enforce domain right a:Arrow {
                context = c2: AContext {},
                source = v1: Vertex {},
                target = v2: Vertex {},
                name = nm
        };
        when {
                ContextMapping(c1, c2);
               v.outgoing->size()=1;
               v.incoming->size()=1;
               ArrowNodeMapping(e1, v1);
                ArrowNodeMapping(e2, v2);
        }
}
```

- a node on the left that has an incoming arrow e1 and *n (n>1)* outgoing arrows (i.e., the node is a "split node") will be mapped on *n* arrows on the right (one for each outgoing arrow on the left) using the rule indicated below. As in the case of the previous rule, the rule fires when contexts have been related and the incoming arrow e1 has been related to the source of the arrow a (on the right) and an outgoing arrow e2 to the target of a.

```
top relation SplitArrowMapping {
        nm, nm2: String;
        enforce domain left e2:Arrow {
                source = v:SplitNode {
                        context = c1: SContext
{},
                        incoming = e1: Arrow
{},
                        name = nm
                },
                name = nm2
        };

        enforce domain right a: Arrow {
                context = c2: AContext {},
                source = v1: Vertex {},
                target = v2: Vertext {},
                name = nm.concat(nm2)
        };

        when {
                ContextMapping(c1,c2);
                v.outgoing->size()>1;
                v.incoming->size()=1;
                ArrowVertexMapping(e1, v1);
                ArrowVertexMapping(e2, v2);
        }
}
```

- a similar rules can be defined when the node on the left is a “join node”;
- rules must also be defined when the node on the left is a start node/final node (no incoming/outgoing arrows) or the node is simultaneously join and split node (two or more incoming arrows and two or more outgoing arrows). Because of space limitations we do not provide the specification of these rules, although for a complete transformation these situations must be equally considered.

Example: An example duality pattern application is the generation of a statechart diagram from an activity diagram. The corresponding QVT specification can be downloaded from our Wiki catalogue [10].
Applicability: The duality pattern can be used to related models expressed in languages between which a duality relationship can be established (i.e., nodes/constructs from the source language can be semantically related/mapped to arrows/relations in the target language and, relations/arrows in the source language can be related/mapped to nodes/constructs in the target language). For example it can be used to define transformations between UML activity diagrams and UML statechart diagrams. It should be noted, that situations may occur (depending on the metamodels of the involved languages) when this type of pattern is not bidirectional.

5.5. The Flattening pattern

Goal: Remove the hierarchy from the source model.
Motivation: Models are often hierarchically structured. Consider for example package hierarchy in UML, composite states in Statecharts or Hierarchical PetriNets. Such hierarchical structuring usually is intended to make the models easier to understand and do not have inherent semantics. In order to realize such hierarchical models in code or formally analyze them using some tool, it may be necessary to first flatten the model to a model without hierarchy.
Specification: We make the following assumptions:

- Source and target models have the same metamodel.
- Source and target models both have a unique RootElement, which are related by the RootMapping relation, an instance of the mapping pattern.
- Model elements in the source model belong to (have as their context) the RootElement or to a Composite element, representing the hierarchy.

Our transformation strategy is as follows. All Composites on the left are related to the RootElement on the right. The CompositeContext here is either the RootElement or another Composite. Thus the CompositeContext c1 should be related to the RootElement r via RootMapping or CompositeFlattening itself.

```
top relation CompositeFlattening {
        checkonly domain left c: Composite {
        context = c1 : CompositeContext {} };
        enforce domain right r: RootElement{};
                when {
                RootMapping(c1,r) or
CompositeFlattening(c1,r);
        }
}
```

All other elements will be simply copied using instances of the mapping pattern above. In these rules the ContextMapping should be replaced by the when clause of the CompositeFlattening rule.

```
relation ElementMapping {
        nm: String;
        enforce domain left x: Element {
              name = nm,
              context = c1 : Context {}
        };
        enforce domain right y: Element {
              name = nm,
              context = c2 : Context {}
        };
        when {
              RootMapping(c1,c2) or
CompositeFlattening(c1,c2);
        }
}
```

Examples: Below we have applied this pattern to flatten the Block hierarchy from a Shapes model. The additional condition `not(RootBlockMapping(b1,b2))` is required to make sure that the Block b1 is not the RootBlock.

```
top relation BlockFlattening {
        checkonly domain left b1: Block {
                block = c1 : Block {} };
        enforce domain right b2: RootBlock {};
        when {
                not(RootBlockMapping(b1,b2));
   RootBlockMapping(c1,b2) or
BlockFlattening(c1,b2);
        }
}
```

Applicability: The flattening pattern can be used to remove hierarchical structure from a model.

6. Applying transformation patterns

Transformation specifications are made up of rule definitions (see Section 4.1). Each rule tackles a small part of the transformation problem. Transformation patterns can help to identify solutions to these partial transformation problems. In this section, we show how a complete transformation definition can be constructed and specified by combining rule definitions, which in turn are obtained by applying the rule patterns. The example illustrates how several different rule patterns are combined to provide a complete solution for a particular transformation problem. To demonstrate the viability of the approach the transformation is applied to statecharts, which is a well-known and frequently used formalism of UML. The **problem statement** is:
Given a hierarchical statechart, i.e., a statechart with composite states, produce a flat statechart without any hierarchy describing the same behaviour (Figure 4).

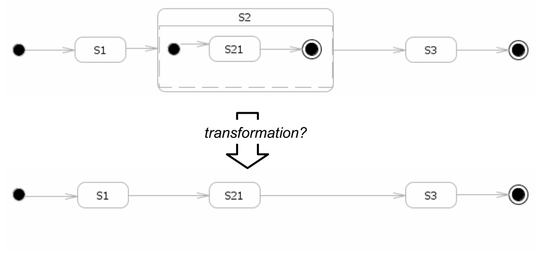
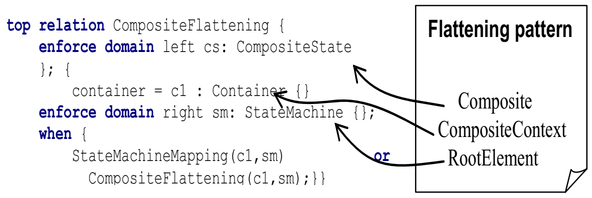


Figure 4. Statechart problem definition

We start the transformation specification with the declaration of the source and target domains. The source and target models are of the same type here, i.e., a statechart (see Figure 2 for the statechart metamodel).

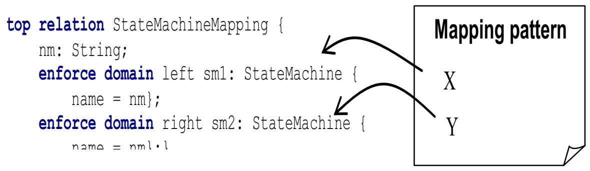
```
Transformation
StatechartFlattening(left:StateChart,right:Sta
teChart) {}
```

Obviously, the problem statement is a Flattening problem. Thus we first apply the Flattening pattern to define a rule for flattening composite states. Here the Composite is a CompositeState, the CompositeContext is a Container (which is another CompositeState or the StateMachine), and the RootElement is a StateMachine.



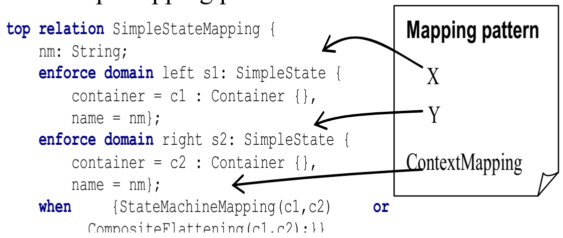
```
top relation CompositeFlattening {
    enforce domain left cs: CompositeState
    }; {
        container = c1 : Container {}
    enforce domain right sm: StateMachine {};
    when {
        StateMachineMapping(c1,sm)          or
          CompositeFlattening(c1,sm);}}
```

The above rule depends on a mapping between the root elements, i.e., the encompassing state machines. This relation is a simple instance of the Mapping pattern, in which a state machine on the left is related to a state machine on the right, such that their names are equal. As StateMachine is the root hierarchical concept no ContextMapping needs to be specified.



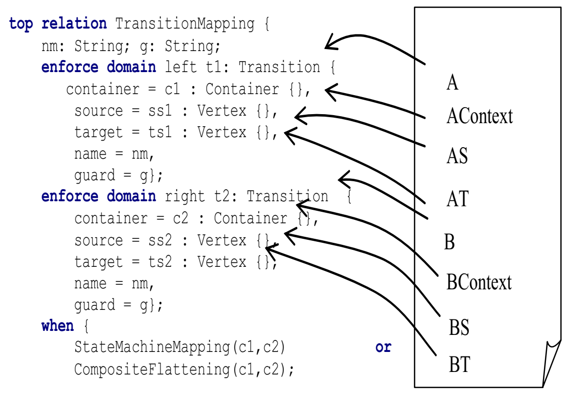
```
top relation StateMachineMapping {
    nm: String;
    enforce domain left sm1: StateMachine {
        name = nm};
    enforce domain right sm2: StateMachine {
        name = nm};}
```

The remaining elements of the state machine are also instances of the Mapping pattern. The rule for transforming SimpleStates, for example, is obtained by instantiating the Mapping pattern.
Also transitions between states from the source model are simply mapped onto transitions in the target model (as shown below), which is an instance of the Relationship Mapping pattern.



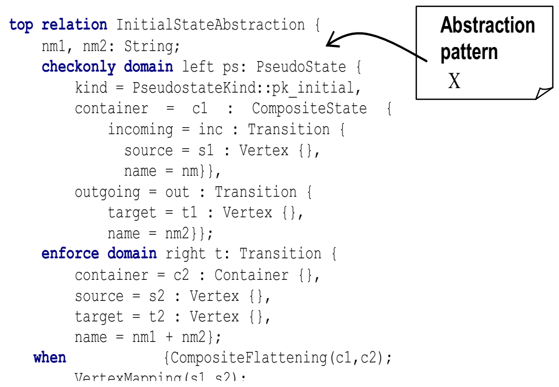
```
top relation SimpleStateMapping {
    nm: String;
    enforce domain left s1: SimpleState {
        container = c1 : Container {},
        name = nm};
    enforce domain right s2: SimpleState {
        container = c2 : Container {},
        name = nm};
    when     {StateMachineMapping(c1,c2)     or
          CompositeFlattening(c1,c2);}}
```

In the statechart metamodel, a Vertex is defined as a generalization of a State that can be used to distinguish between different types of states, e.g., Start State or Final State. It has been introduced to cope with a deficiency in the transformation execution engine to handle enumerations.



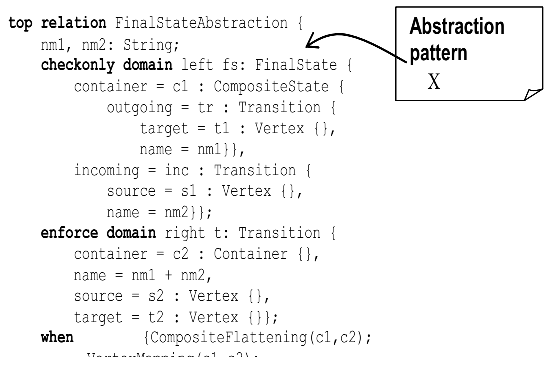
```
top relation TransitionMapping {
    nm: String; g: String;
    enforce domain left t1: Transition {
       container = c1 : Container {},
        source = ss1 : Vertex {},
        target = ts1 : Vertex {},
        name = nm,
        guard = g};
    enforce domain right t2: Transition  {
        container = c2 : Container {},
        source = ss2 : Vertex {},
        target = ts2 : Vertex {},
        name = nm,
        guard = g};
    when {
        StateMachineMapping(c1,c2)          or
        CompositeFlattening(c1,c2);
```

In order to obtain a semantically correct model, we additionally need to remove the initial and final states of all composite states. Moreover, we need to make sure that transitions that originally had a composite state as their target are now redirected to the target of the outgoing transition of the initial state of that composite state. And, conversely, that transitions that originally had a composite state as their source are now moved to the source of incoming transitions of the final state of that composite state. In principle we can do this by applying the node abstraction pattern, which takes a node and replaces it by a simpler structure. This pattern can be applied twice, first to remove the composite states and second to remove initial and final states. The next figure depicts the abstraction of pseudostates.



```
top relation InitialStateAbstraction {
    nm1, nm2: String;
    checkonly domain left ps: PseudoState {
        kind = PseudostateKind::pk_initial,
        container  =  c1   :  CompositeState  {
            incoming = inc : Transition {
              source = s1 : Vertex {},
              name = nm}},
        outgoing = out : Transition {
            target = t1 : Vertex {},
            name = nm2}};
    enforce domain right t: Transition {
        container = c2 : Container {},
        source = s2 : Vertex {},
        target = t2 : Vertex {},
        name = nm1 + nm2};
   when            {CompositeFlattening(c1,c2);
        VertexMapping(s1,s2);
```

The next figure depicts the abstraction of the FinalState by instantiating the node abstraction pattern.



```
top relation FinalStateAbstraction {
    nm1, nm2: String;
    checkonly domain left fs: FinalState {
        container = c1 : CompositeState {
            outgoing = tr : Transition {
                target = t1 : Vertex {},
                name = nm1}},
        incoming = inc : Transition {
            source = s1 : Vertex {},
            name = nm2}};
    enforce domain right t: Transition {
        container = c2 : Container {},
        name = nm1 + nm2,
        source = s2 : Vertex {},
        target = t2 : Vertex {}};
    when        {CompositeFlattening(c1,c2);
```

By combining all these transformation rules a transformation specification is obtained that is able to flatten the statechart.

7. Conclusions

Writing model-to-model transformations can be a tedious undertaking. In most model transformations there are certain underlying principles that can be used to facilitate the production of model transformations.
This paper has identified basic transformation patterns that frequently occur in model-to-model transformations such as the mapping pattern, the duality pattern, the refinement/abstraction pattern, the flattening pattern. These patterns have been described and specified in QVT Relations, resulting in a catalogue of basic transformation patterns. A simple Shape language has been introduced to illustrate most of the patterns. The catalogue of transformation patterns provided in this paper is a first attempt to categorize transformation principles in QVT Relations. This list is, however, not complete. A natural way to enrich this collection of pattern, would be to try to join our approach with similar initiatives in this area, such that the Modelware project ([1]). It remains however to investigate to what extent patterns proposed in [1] are implementable using the declarative QVT languages. Furthermore, we challenge the community to elaborate on this kind of work and extend the list of patterns.
Composition of model-to-model transformation should be guided, in our view, by the usability of the resulting transformation. In this respect we believe that it is not always meaningful to compose patterns. In practice, some particular compositions of patterns will occur more frequently than others. For example, the mapping pattern is often composed with many other patterns, but composition of node refinement with duality seems to make less sense in practical situations. Nevertheless, an analysis of pattern compositionality and parameterization makes the object of future work.

Acknowledgments

The work presented in this paper is part of the Freeband A-MUSE project (http://a-muse.freeband.nl), which is sponsored by the Dutch government under contract BSIK 03025.

References

[1] Allilaire, F., Bézivin, J., Olsen, G., Bailey, T., Bonet, S., Mantell, K., Vogel, R.: "D1.6-3 Identification of Transformation Patterns", FP6-IP 511731 MODELWARE, 04/09/06, http://www.modelware-ist.org/index.php?option=com_remository&Itemid=79&func=fileinfo&id=132 (1-2-2008).

[2] Alexander, C., Ishikawa, S., Silverstein, M.: "A Pattern Language: Towns, Buildings, Construction (Center for Environmental Structure Series)", Oxford University Press, 1977.

[3] Brahe, S. and Bordbar, B.: "A Pattern-based Approach to Business Process Modeling and Implementation in Web Services", In Workshop Proceedings of the 4th International Conference on Service-Oriented Computing ICSOC 2006, Chicago, IL, USA, December 4-7, 2006, Lecture Notes in Computer Science, Vol. 4652/2007, pp. 166-177, ISBN 978-3-540-75491-6.

[4] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: "Design Patterns: Element of Reusable Object-Oriented Software". Published by Addison-Wesley, 1995. ISBN 0201633612. 27th printing, November 2003.

[5] Judson, S.R., Carver, D.L., France, R.B.: "A Meta-Modelling Approach to Model Transformation". In: OOPSLA'03, p. 326-327, October 26-30, 2003, Anaheim, USA.

[6] Miller, J. and J. Mukerji (eds.): "MDA Guide Version 1.0.1", Object Management Group, June 2003.

[7] Object Management Group: "OMG Unified Modeling Language: Superstructure Version 2.1.1", 2003, http://www.omg.org/docs/formal/07-02-03.pdf (31-1-2008).

[8] Object Management Group: "Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification", Final Adopted Specification ptc/05-11-01, Nov. 2005, http://www.omg.org/docs/ptc/05-11-01.pdf (1-2-2008).

[9] Soley, R. and the OMG Staff Strategy Group: "Model Driven Architecture", Object Management Group White Paper, Draft 3.2, Nov. 2000.

[10] Wiki patterns catalogue: https://doc.telin.nl/dsweb/View/Wiki-90/HomePage.

Combining Rules and Activities for Modeling Service-Based Business Processes

Milan Milanović[1], Dragan Gašević[2], Gerd Wagner[3]
[1]FON-School of Business Administration, University of Belgrade, Serbia
[2]Athabasca University, Canada
[3]Brandenburg University of Technology at Cottbus, Germany
milan@milanovic.org, dgasevic@acm.org, wagnerg@tu-cottbus.de

Abstract

It is widely acknowledged that business process management would greatly benefit from integration with business rule management. But there is still no established solution to this integration problem, and the leading business process modeling language, BPMN, does not provide any explicit support for rules. In this paper, we are going to investigate the extension of BPMN by adding rules as a modeling concept in the form of a new gateway type, using the principles of Model-Driven Engineering. The integration will be done on the level of the metamodels of the involved languages, resulting in a new rule-based process modeling language called rBPMN (Rule-based BPMN).

1. Introduction

Recent research [26] has identified a lack of explicit formalism in the process modeling languages for capturing business rules. In this paper, we follow the business rules approach [5] by combining business process models and business rules in a way that makes rules first-class citizens in business process modeling. The key idea is to extract some parts of a business logic contained implicitly in business process models into explicit definitions of business rules. To achieve this, we propose adding a new gateway type, called *rule gateway*, to the business process modeling language BPMN. We also discuss how Web service compositions can be extracted from such rule-based process models and discuss a set of business process modeling (workflow) patterns for modeling SOAs. By adding rules to the BPMN we enable run-time updates of a business logic, which with regular BPMN cannot be done.

Our high-level modeling approach allows developers to focus on a problem domain rather than on an implementation technology. Following the principles of Model Driven Engineering (MDE), we integrate a new rule gateway type into BPMN business process models on the metamodel level, resulting in a language called rBPMN (Rule-based BPMN), which facilitates business process modeling by domain experts and allows to transform such process models into different SOA implementation platforms [7].

In order to make such abstract business process definitions modeled by business expert's executable, the ability to integrate heterogeneous enterprise systems is needed. Web Services (WS) technology provides this ability by encapsulating enterprise systems functionality into services. Service composition languages, such as the Business Process Execution Language (WS-BPEL [11]) and service interaction protocol languages such as the Web Services Choreography Description Language (WS-CDL [12]), make it possible to combine different services and to automate and standardize the execution of cross-organizational business process models [10]. Web services, service composition languages and service interaction protocol languages are key technologies for enabling SOA. In our approach, we propose obtaining service compositions from rule-based business process models to make those process models executable.

The rest of the paper is structured as follows. In the next section, a motivating example is presented which is used to explain different concepts used throughout the paper. In Section 3, requirements for modeling rule-enabled SOAs are given, together with proposed methodology for developing rule enabled SOAs. Section 4 introduces MDE, business rules and business process languages. In Section 5, we present different business process modeling patterns that our solution needs to support and rBPMN language. Before concluding the paper in Section 7, in Section 6, we summarize the related work.

2. Motivating Example

In this section, we use a well-known example of a travel agency process to show basic concepts of the rBPMN language for illustrating how BPMN models are enriched by rules. For simplicity, in Figure 1, we omit some of the model elements that would be needed to model the complete functionality of the described process (such as selecting Airline).

This scenario includes a Traveler role, which starts the process by requesting a trip arrangement from the TravelAgency. In Figure 1, the TravelAgency receives a request from a Traveler and communicates with Airline companies and Hotels. When all of the necessary information is collected, the TravelAgency returns the requested information to the Traveler, which denotes if the trip is planned or not. If yes, it provides the itinerary about the planned trip.

In this travel agency scenario, we added on a business process diagram (BPMN) additional annotations and rules, such as <<BpelProcess>> and <<WS>> annotations to the pools, and reaction rules (RR) represented with as diamond-based gateway called *rule gateway* with the boldface **R** tag. Reaction rules are also used to produce SOA artifacts in the Airline and Hotel pools. These rules describe mappings to Web service operations (see Section 4.3). By mapping these rules to Web service operations, we can extract service compositions (orchestrations and choreographies) from rBPMN models. We also added an additional constraint on the rule gateway in a form of a conditional expression on a construct *Flight*, which is equality of its property *price* and *wantedPrice* property of the incoming request message. In the subsequent sections, we describe how these additional concepts on the business process diagram can improve business process modeling and SOA generation.

3. Requirements

For integration of rules and processes in our case, we first need to define basic concepts that such integration should support for modeling SOAs. In order to model SOA-based processes by using abstract business process languages such as rBPMN in Figure 1, we need to define requirements that such business process models should have. We will define those requirements through a methodology for development of rule enabled SOAs. In Figure 2, we show our proposal for the methodology for developing rule-enabled SOAs. Here, we briefly explain only first five stages, which are the most relevant for our methodology:

1. *Requirements specification*. In this stage, a business analyst collects information about the application domain and business functions. The output of this specification is the project requirements document.
2. *Process design*. In this stage, using information from the requirements document from the previous requirements phase, a process modeler defines an abstract business process model (by BPMN). For this phase, we need to extend basic BPMN concepts by adding to them annotations for SOA-related concepts (e.g., the <<WS>> annotation of the Airline pool in Figure 1) in order to make them mappable to service compositions.
3. *Data design*. This stage includes defining a domain model (vocabulary) by using information collected during stage 1. This stage may include some existing vocabularies. For this phase, we need to extend BPMN by connecting to it an underlying data layer (i.e., some existing vocabulary language, such as UML class models [23]).
4. *Rule design*. In this stage, business (reaction) rules are added to the process. This activity includes rules which are already defined or rules that can be direct-

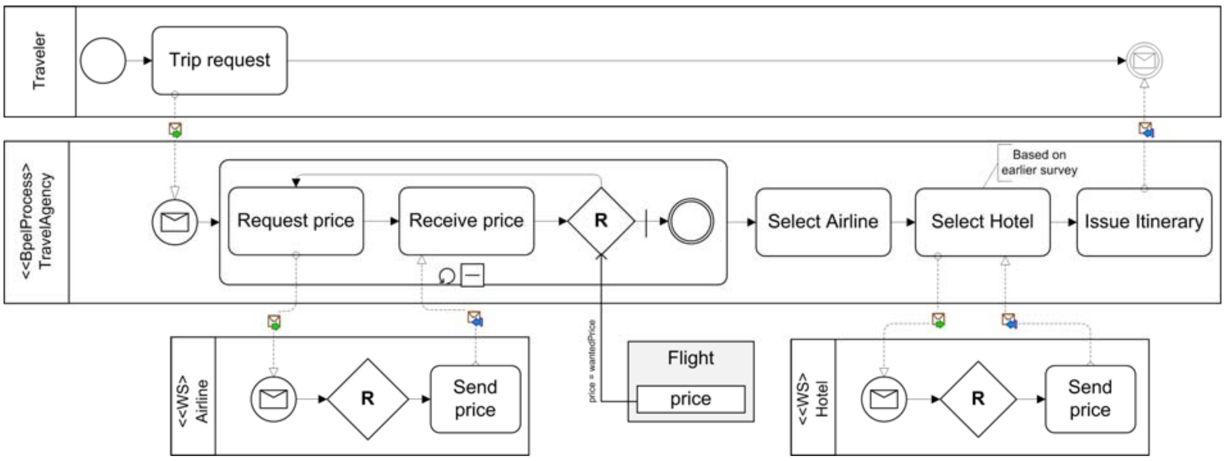


Figure 1. Travel Agency Process (rBPMN)

ly defined in the process diagram and later refined in a rule language. In this stage, we need to include reaction rules that can be added into process models. These rules must use data model from stage 2 of this methodology, as whole process is annotated with data from this model. Another important point here is a place where rules should be used. As rule gateway is one of the rBPMN gateways, we propose that this gateway should be commonly used in places where some kind of condition is needed to be defined that will fork the sequence flow in the process. Of course, every rule gateway must be logically put into rBPMN process as any other modeling element. In combination with BPMN elements, rules can be used for modeling Web Service in terms of Message Exchange Patterns (MEPs) [15]. This means that our rules can define how services can be used and service definitions, too [6].

5. *Orchestration and choreography generation.* In this stage, generation of executable orchestrations (e.g., WS-BPEL [11]) or choreographies (e.g., WS-CDL [12]) from the rule-based business processes model could be done by using a model transformation approach [21]. This step can include some existing choreographies and orchestrations. Choreographies are activities of the same process orchestration, but between activities of different process orchestrations [7], while orchestrations are modeled activities, with their relationships, that are performed within a single organization [7].

In the Section 5, we will present more about main stages of our methodology, namely, process design, data design and rule design phases regarding the orchestration and choreography phase. From this motivating example and requirements, we emphasize topics of the benefits for rBPMN, to be discussed about in the rest of the paper:

- Precisely defined descriptions of services (by using reaction rules), which can be defined and integrated into business processes.
- Definitions of rules along with Web services in business process models allow for generation rules that regulate how to use Web services.
- Ability to integrate declarative business logic via the use of rules into process-oriented models.
- Ability to generate SOAs, including both service and service composition definitions, where combination of rule and processes can be translated to full service definitions.

4. Background

In this section, we give a brief overview of the technologies and languages relevant to the problem under study. This includes a short description of the MDE, business process language – (BPMN) and rule language (R2ML).

4.1. Model Driven Engineering

MDE is a new software engineering discipline in which the process heavily relies on the use of models [3], while OMG's Model Driven Architecture (MDA) [17] is considered as an implementation of MDE [4]. A model is defined as a set of statements about some

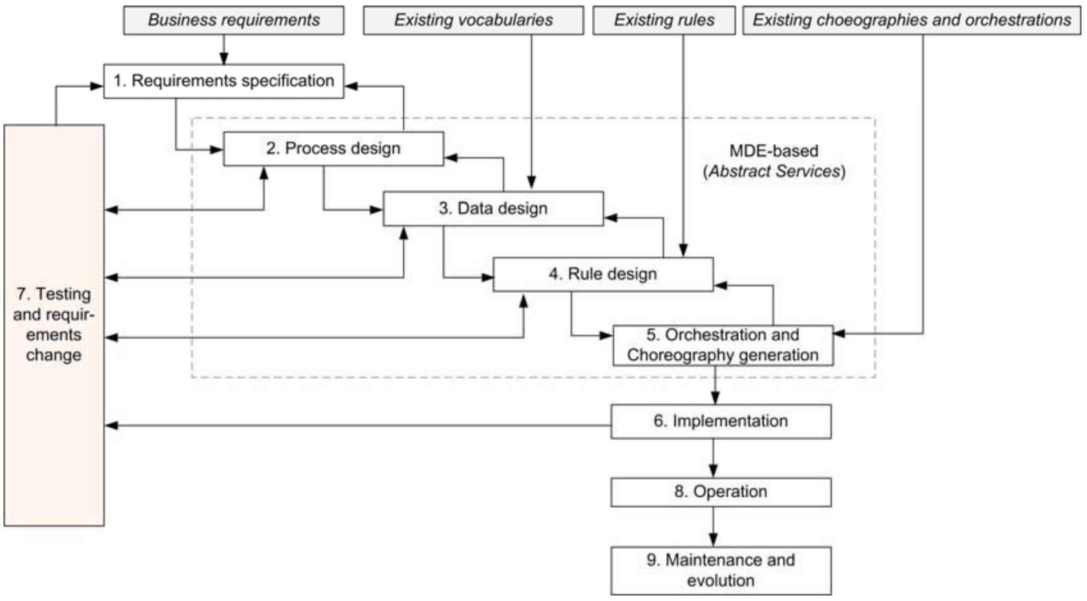

Business requirements
Existing vocabularies
Existing rules
Existing choeographies and orchestrations
1. Requirements specification
2. Process design
3. Data design
4. Rule design
5. Orchestration and Choreography generation
MDE-based (Abstract Services)
7. Testing and requirements change
6. Implementation
8. Operation
9. Maintenance and evolution

Figure 2. Methodology for developing rule enabled SOAs

system under study [17]. Models are usually specified using modeling languages (e.g., UML), while modeling languages can be defined by metamodels. A metamodel is a model of a modeling language. That is, a metamodel makes statements about what can be expressed in the valid models of a certain modeling language [28]. A typical meta-modeling framework (MDA) has three layers, namely:

- M1 layer or model layer where models are defined by using modeling languages;
- M2 layer or metamodel layer where models of modeling languages (i.e. metamodels) are defined (e.g., UML [23] or BPMN metamodel [19]) by using metamodeling languages such as MOF;
- M3 layer or metametamodel layer where the only metamodeling language is defined (i.e. MOF) by itself [20].

The relations between different meta-layers can be considered instance-of or conformant-to, which means that a model is an instance of a metamodel, and a metamodel is an instance of a metametamodel.

4.2. Business Processes: BPMN

BPMN represents an OMG specification [1918] whose intent in business process modeling is very similar to the intent of the UML for object-oriented design. It identifies the best practices of existing approaches and combines them into a new, generally accepted business process modeling language.

In BPMN, business process models are expressed in business process diagrams. Each business process diagram consists of a set of modeling elements. The notational elements in business process diagrams are control flows which are modeled using three different kinds of flow objects. Flow objects are: Events that occur at the start, during, or at the end of a process (represented by circles in BPMN), activities that are performed, and gateways for guiding, splitting and merging control flow. BPMN activities are represented by rectangles (with rounded corners) that can either stand for atomic tasks or so-called sub processes. An example of a BPMN activity is “Trip request” activity in Figure 1. The diamond shaped gateways represent decisions, merges, forks, and joins in the control flow. A gateway can be thought of as a question that is asked at a point in the process flow. The question has a defined set of alternative answers, which are in effect gates. The event-based XOR gateway represents a branching point where the alternatives are based on an event that occurs at that point in the process flow, while the data-based XOR gateway is similar to event-based with the difference that alternatives are based on defined conditions. Other gateways with special decision logic exist too, such as inclusive, complex and parallel gateways.

Connecting objects (i.e., different kinds of lines) connect the flow objects to create a basic skeletal structure of a business process. A Sequence Flow is represented by a solid arrow and is used to show the order that activities will be performed in the business process. An example of a sequence flow is given in Figure 1 between the task “Request price” and RR. A Message Flow is represented by a dashed line with an open arrowhead and is used to show the flow of messages between two separate business process participants. An example of message flow is shown in Figure 1 between the task “Receive price” and RR (rule gateway). Associations, represented as dotted lines, are used to associate data objects, text, and other artifacts with flow objects.

BPMN also has a concept called Pool, which represents a participant in a business process (an example of a Pool in Figure 1 is the “Traveler” Pool). A participant can be a specific business entity (e.g., a company) or can be a more general business role (e.g., buyer or seller). Graphically, a Pool is a container for partitioning a process from other Pools.

For a more detailed description of BPMN refer to [19]. By using MDE principles described in Section 4.1, our integration of business rules into BPMN is done on the level of metamodels, and for this purpose we use the BPMN metamodel proposal given in [19].

4.3. Business Rules: R2ML

A business rule is a statement that aims to influence or guide behavior and information in an organization [29]. There are different categories of business rules such as [31] integrity, derivation, reaction, and production. We decided to use REWERSE I1 Rule Markup Language (R2ML), as it supports abovementioned types of rules. The R2ML rule language is defined by a metamodel, by using the MOF metamodeling language [27] [31]. The R2ML attempts to address all the requests defined by the W3C working group for the standard rule interchange format [8]. As business process models (BPMN) are represented by using metamodeling principles, the R2ML choice is obvious. Integrity rules in the R2ML, also known as (integrity) constraints, consist of a constraint assertion, which is a sentence in a logical language such as first-order predicate logic or OCL [22]. Derivation rules are used to derive new knowledge (conclusion) if a condition holds. Production rules produce actions if the conditions hold, while post-conditions must also hold after the execution of actions. A reaction rule (RR) is a statement of programming logic [9] that specifies the execution of one or more actions in the case of a triggering event occurrence and if rule conditions are satisfied. Optionally, after the execution of the action(s), post-conditions may be made true.

R2ML also allows one to define vocabularies by using the following constructs: basic content vocabulary, functional content vocabulary, and relational content vocabulary. Here, we give a short description of vocabulary constructs that we use in this paper. *Vocabulary* is a concept (class) that can have one or more *VocabularyEntry* concepts. *VocabularyEntry* is an abstract concept (class) that is used for representing other concepts by its specialization. For example, one of *VocabularyEntry-s* is an R2ML *Class* concept which represents the class element similar to the notion of the UML Class. An R2ML *Class* can have attributes (class *Attribute*), reference properties (class *ReferenceProperty*) and operations (class *Operation*). Messages are defined in R2ML as *EventExpression*-s, while those *EventExpression* types are defined in the R2ML Vocabulary as event types.

Because of the space constraints, we describe here only reaction rules used in our motivating example in Figure 1, as our illustration of rBPMN. A more detailed description of other types of rules supported in R2ML can be found in [27]. Reaction rules (RR) represent a flexible way for specifying control flows, as well as for integrating events/actions from a real life [9]. Reaction rules are represented in the R2ML metamodel as shown in Figure 3: *triggeringEventExpr* is an R2ML *EventExpression*; *conditions* are represented as a collection of quantifier free logical formulas; *triggeredEventExpr* is an R2ML *EventExpression* and represents a system state change; and (optional) *postcondition* must hold when the system state changes.

The R2ML event metamodel defines basic concepts that are needed for dynamic rule behavior. R2ML *EventExpression* can be one of the following concepts (classes): *AtomicEventExpression, AndNotEventExpression*, *SequenceEventExpression*, *ParralelEventExpression,* and *ChoiceEventExpression*. We use *AtomicEventExpression* for modeling messages that are part of the business process diagram underlying data layer. This is because we are using R2ML elements (expressions) to represent message definitions in rBPMN. Each *AtomicEventExpression* has its own type – *EventType*. *EventType* is defined as a subclass of *Class* (in the R2ML Vocabulary). This means that each *EventType* has their own attributes, associations, and all other features of R2ML classes.

Along with its metamodel, R2ML has a graphical concrete syntax called UML-Based Rule Modeling Language (URML) [9] [15]. URML is developed as an extension of the UML metamodel to be used for rule modeling. In URML, modeling vocabularies is done by using UML class models. Rules are defined on top of such models, while URML models are stored in the R2ML XML (concrete syntax) format [27].

In Figure 4, we show an example of the URML definition of the reaction rule used Figure 1. This reaction rule returns a message that say if certain flight price is equal to wanted flight price. The URML class that represents the input message (*CheckPriceRequest* in Figure 4) of the reaction rule is *AtomicEventExpression* type instance, and it is represented with the <<atomic event expression>> stereotype on UML classes. The same stereotype is also the type of the reaction rule output message (*CheckPriceResposne*). The input message *CheckPriceRequest* is connected with *Class* instance type called *Airline*, by using association. While condition is represented with a *Flight* class that connects with RR and the condition expression defined on this connection (*price* = *wantedPrice*).

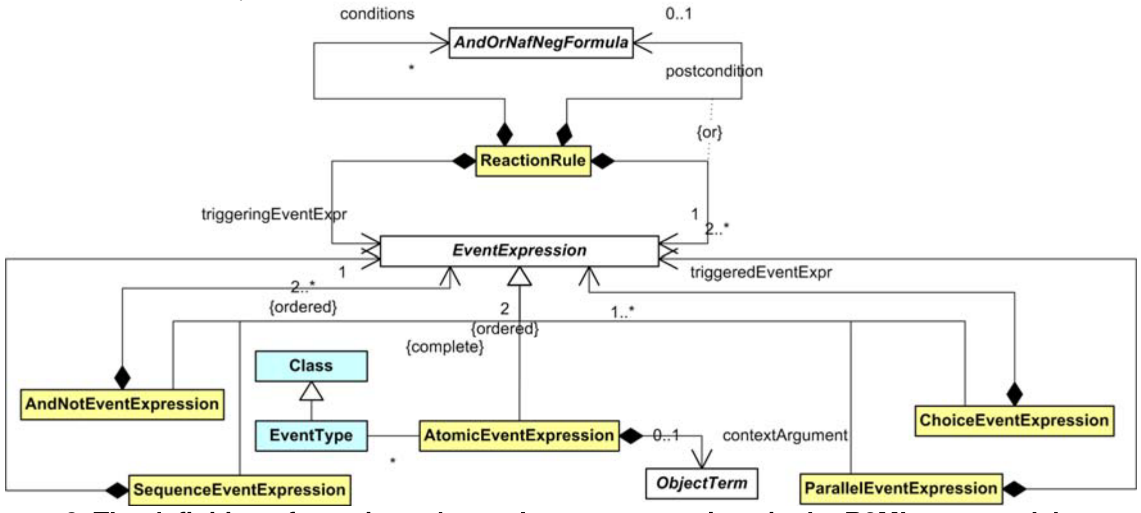


Figure 3. The definition of reaction rules and event expressions in the R2ML metamodel

In our previous work [15], we presented how reaction rules can be translated into Web services, i.e., WSDL descriptions. We have done this in the following way. A triggering event of a RR maps to the input message of a Web service operation. The action of the RR, which is triggered when a condition is true, maps to the output message of the Web service operation. To model condition constructs (e.g., *price* = *wantedPrice*) we use OCL filters [22]. OCL filters are based on a part of OCL that models logical expressions, which can be later translated to R2ML logical formulas, as parts of reaction rules. However, these OCL filters cannot be later translated to Web service descriptions (e.g., WSDL), as those languages cannot support such constructs. But, we can translate our URML models into rule-based languages (e.g., Jess or Drools). This means that for each Web service, we can generate a complementary rule, which fully regulates how its attributed service is used.

5. Integration of Business Rules and Processes: rBPMN

In this section, we describe the integration of the BPMN and R2ML languages in order to create a new rule-based process modeling language called rBPMN by using the MDE approach. We will show evaluation of this integration by using a set of different business process modeling patterns, namely, control flow and service interaction patterns.

5.1. Business Process Modeling Patterns

In order to show how rBPMN can be used to model SOAs, i.e., orchestrations (basic control flow patterns) and choreographies (service interaction patterns), we give here business process patterns that describe our approach in a rBPMN graphical concrete syntax.

5.1.1. Basic Control Flow Patterns. Control flow patterns represent a set of 21 workflow patterns created to show expressivity of workflow management systems [32]. These patterns can be used in business process modeling, and also to compare the expressiveness of process languages. Basic control flow patterns include sequence, and split, and join, as well as exclusive or split and exclusive or join. Control flow patterns are defined at the process model level. As some of these patterns can be defined in BPMN without using rules, by adding rules we enrich those diagrams in a way that such business processes can be changed in a real-time by changing only rules, and not by changing the whole process. As BPMN [19] has a weak support for rule-based gateways, where conditions are usually written in a natural language [26], by adding formal rules, we enable execution of such processes possible on some execution platform, such as BPEL [11].

Here, we evaluate the support needed by the rBPMN to represent these patterns, on an example of three workflow patterns represented in the rBPMN graphical concrete syntax due to lack of space, but we should note that we supported all of the 21 basic patterns shown in [32]. We should note that these patterns apply to business models that are used to model process orchestrations, because activities used in these patterns are performed within a single organization (i.e., BPMN Pool). Regarding mappings from rBPMN to execution languages, such as BPEL, these business models can be mapped into BPEL and WSDL constructs by using already defined mappings [19], where rules are mapped by using standard BPEL constructs, such as *invoke*, or by extending BPEL to support rules.

We should note that for control flow patterns the focus is on control flow and not on message exchange, so we do not show messages in control flow patterns figures in Section 5.1.1.

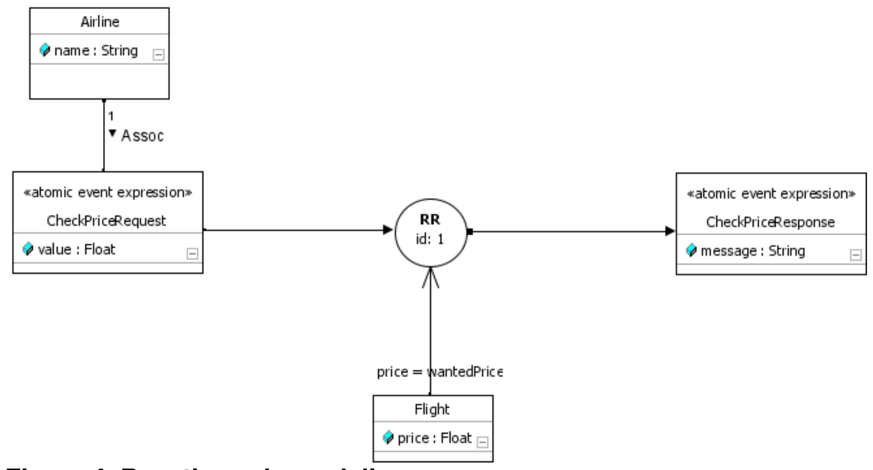


Figure 4. Reaction rule modeling

5.1.1.1. Sequence. The pattern "Sequence" is represented in Figure 5. An activity of the type *activity2* is started after the completion of an activity of the type *activity1*. The rule gateway symbol for the reaction rule is usually omitted from a graphical representation of this pattern, but we show it in order to present how reaction rule can be simply located in a sequence flow. This rBPMN pattern can also be represented in BPMN, as it is very simple.

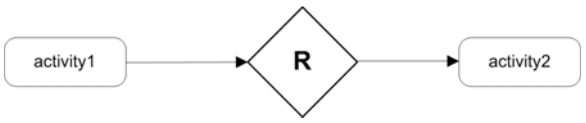


Figure 5. The "Sequence" pattern

5.1.1.2. Parallel Split. The pattern "Parallel Split" splits an activity into two or more activities which can be performed in parallel, thus allowing activities to be performed simultaneously or in any order. This pattern is shown in Figure 6. After the end of an *activity1*, activities of the types *activity2 … activityn* are started to be performed in parallel. In this case, the *triggeredEventExpr* of the RR is a *ParralelEventExpression* that contains two or more activities (i.e., activity2 ... activityn) so that they can perform in parallel. This rBPMN pattern can be represented in BPMN too, but with using a reaction rule we have flexibility to use event-based logic to choose its outgoing alternatives.

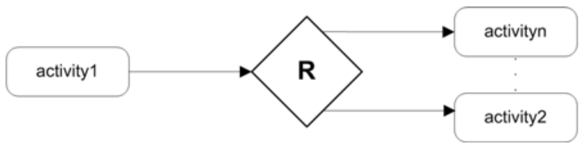


Figure 6. The "Parallel Split" pattern

5.1.1.3. Exclusive Choice. The pattern "Exclusive Choice" chooses one of several activities to be performed based on a control data. In an example of Figure 7, after the end of an *activity1*, if the condition specified by *predicate* and *condition* is true, an *activity2* is started. Otherwise, *an activity3* is started (this choice is denoted with a cross line on a line between R and activity 3 in Figure 7). This pattern is also shown in Figure 1, in a place where reaction rule (RR) (after "Receive price" task) is used. This pattern can be represented in BPMN, but without using reaction rule when Entities' predicate changes in run-time it is not possible to affect a process.

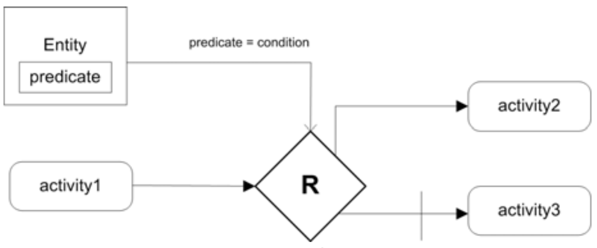


Figure 7. The "Exclusive Choice" pattern

5.1.2. Basic Service Interaction Patterns. The workflow patterns presented in Section 5.1.1 describe control flow that is characteristic for process orchestrations. However, there are several differences between process orchestrations and process choreographies that need specific consideration: choreographies are based on message exchange, and potentially many participants interact in choreography, while orchestrations are based on control flow between the activities of a single process performed by a single organization.

Service interaction patterns aim at filling this gap by proposing small granular types of interactions that can be combined to choreographies [1]. As with workflow patterns represented in Section 5.1.1, these service interaction patterns can be modeled with BPMN, but by modeling them with rBPMN, we show how decision logic can be changed at run-time and how these patterns are annotated for mapping them to service compositions. Rules, activities and events are combined on a level of concrete and abstract syntax (see Section 5.2). In the following subsections, we give an example of three service interaction patterns modeled by means of the rBPMN language. We should note that we supported all of the 13 service interaction patterns [1]. These examples can be mapped to a process choreography language, such as WS-CDL in phase 5 of our methodology (Figure 2).

5.1.2.1. Send. The send pattern represents a one-way interaction between two participants seen from the perspective of the sender. There are different flavors of this pattern, considering, for instance, the moment when the sender selects the receiver: The receiver is known either at design time of the choreography or only during the execution of a conversation.

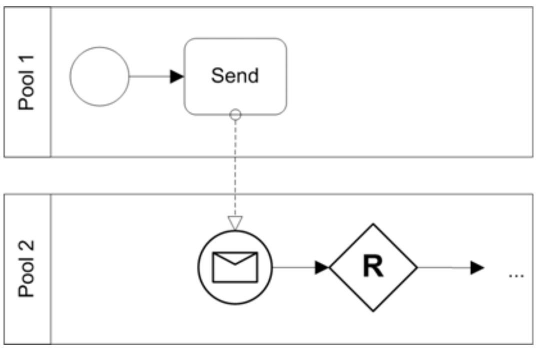


Figure 8. The "Send" pattern

Figure 8 illustrates an example where a one party (represented with BPMN Pool) is sending a message to another party. The sending of the message is realized by a Send task, while the receiving is realized using message event and reaction rule that receive message and continues sequence flow in Pool 2. We should not that we do not show message between the Send task and reaction rule as it is not mandatory (we describe message annotation in the next pattern – Section 5.1.2.2). In addition, this rBPMN pattern can be directly mapped into the Web Service In-Only message exchange pattern (MEP), as we have shown in [15]. The In-Only MEP consists of exactly one input message: service expects one message and it is not obliged to send a response back.

5.1.2.2. Send/Receive. In the send/receive pattern, a participant sends a request to another participant who then returns a response message. Both messages belong to the same conversation. Since there could be several send/receive interaction instances happening in parallel, corresponding requests and responses need to be correlated.

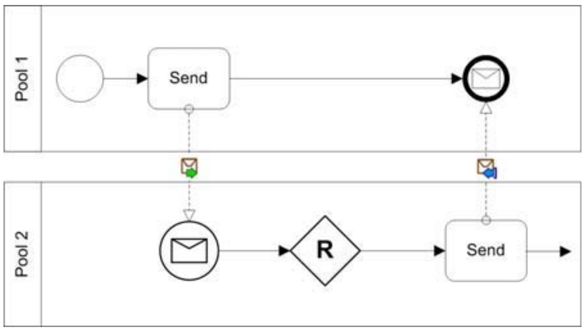


Figure 9. The "Send/Receive" pattern

If, for instance, one party (e.g., Pool 1 in Figure 9) requests information from different parties (e.g., Pool 2 in Figure 9), the different request/response pairs belong to different conversations. In this situation, the first party must be able to tell which quote belongs to which request. Therefore, correlation information must be placed inside the messages. For instance, the request could carry a request identifier which is then also contained inside the response message.

We should note that message exchange between tasks and rules is annotated with messages by using symbols 📩 and 📩 for request and response messages, respectively. These messages are defined as a part of the R2ML Vocabulary and shown in Figure 9, and also in Figure 1 between TravelAgency and Airline and Hotel pools. A reaction rule defined in a process diagram, such as in Figure 1 or Figure 9, can be imported into a process diagram if it is already defined (by following the methodology steps of Figure 2).

In addition, this pattern can be directly mapped into Web Service In-Out MEP [15]. The In-Out MEP consists of exactly two messages: when a service receives an input message, it has to reply with an output message.

5.1.2.3. Racing Incoming Messages. Racing incoming messages are common in business-to-business scenarios; this pattern is described as follows: a participant is waiting for a message to arrive, but other participants have the chance to send a message. These messages by different participants "race" with each other. Only the first message arriving will be processed. The type of the message sent or the category the sending participant belongs to can be used to determine how the receiver processes the message. The remaining messages may be discarded or kept for later consumption. This aspect is not covered by the racing incoming messages pattern.

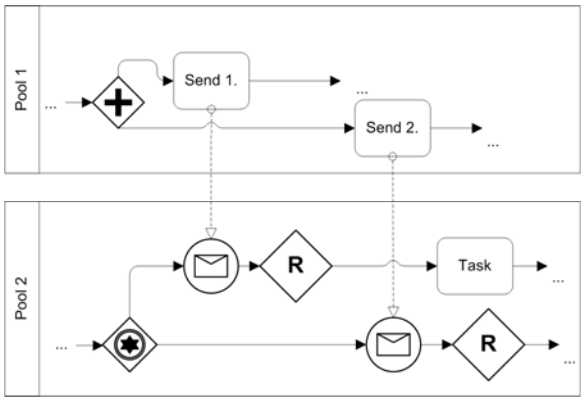


Figure 10. The "Racing Incoming Messages" pattern

Figure 10 shows a scenario where a Pool 2 has done some tasks and now waits for Pool 1 message. If the "Send 1." task sends the message, the first RR in the Pool 2 fire, but in the case that "Send 2" task sends the message, the second RR in the Pool 2 fire and continue sequence flow.

An example of such scenario is where a travel agent (Pool 2) has reserved a flight for a customer (Pool 1), and now waits for a confirmation or a notification that the flight details are not acceptable. In the case of confirmation the payment is initiated, and in the case of rejection a new flight reservation might be needed, where customer can have more requests which can be rejected or accepted by travel agent by using rules.

5.2. rBPMN Language

Business processes are represented by business process models. In order to express process models, there needs to be a notation in place that provides notational elements for the conceptual elements of process meta-

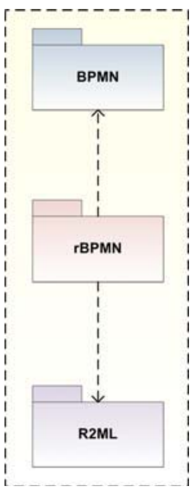


Figure 11. rBPMN metamodel

models. The rBPMN process notation is associated with the rBPMN process metamodel level and with the rBPMN process model level (by using MDE approach), while each rBPMN process model is expressed in the rBPMN process notation associated with the rBPMN process metamodel that describes the rBPMN process model.

Here, we present the rBPMN metamodel (i.e., abstract syntax, and also note that we omit some concepts because of the space limitation and that work on rBPMN metamodel is in progress). The use of the MDE approach for defining abstract syntax of the rBPMN language enables us to have first leverage metamodels as means for checking validity of concrete expressions of process models. We should note that we are using here UML class diagrams notation for representing rBPMN metamodeling concepts. The overall rBPMN organization is shown in Figure 11, where we can see that the rBPMN package includes elements from the BPMN and R2ML packages (metamodels). The core package in the rBPMN metamodel is shown in Figure 12. This package is not changed from the BPMN metamodel proposal [19]. We have chosen this BPMN metamodel proposal because it comprises the whole BPMN language specification and it is clearly mappable to BPEL. The rBPMN core package consists from the following elements:

- A *Process* (class) describes a sequence or flow of activities in an enterprise with the objective of carrying work. In BPMN, a *Process* is depicted as a graph of *FlowElements*, which are a set of activities, events, gateways and sequence flows that define finite execution semantics (see Figure 13).
- *Collaboration* is used to describe interactions between two or more business entities or business roles, which are represented as *Participants* within *Pools*. *Collaboration* shows interactions, that is, the *Messages* exchanged between *Participants* that take part in the *Collaboration*. The Collaboration contains two or more *Pools*, representing the *Participants* in the *Collaboration*. The interactions between the *Participants* are shown by a *MessageFlow* that connect two *Pools*.
- *MessageFlow* connect either to the *Pool* boundary or the *Flow* objects within the *Pool* boundary (they are represented as dashed lines with arrow on one the side and circle on the other side, for example, as between the "Request price" task and the RR in Figure 1). Every *MessageFlow* can have zero or one *Message* attached (see Figure 14).
- *Pool* represents a *Participant* in *Collaboration*. A *Participant* can be a specific business entity, such as TravelAgency, Traveler, Airline and Hotel in Figure 1. Every *Participant* has a *roleRef* attribute (of the enumeration type *Role*) that we have added in metamodel and that defines a business role that the *Participant* plays in the *Collaboration* (such as << WS >> and << BpelProcess >> for *Pools* in Figure 1). This Role enables generation of service compositions because with them we know which pool should be Web service and which pool is main BPEL process. A *Pool* acts as the container for a *Process*.

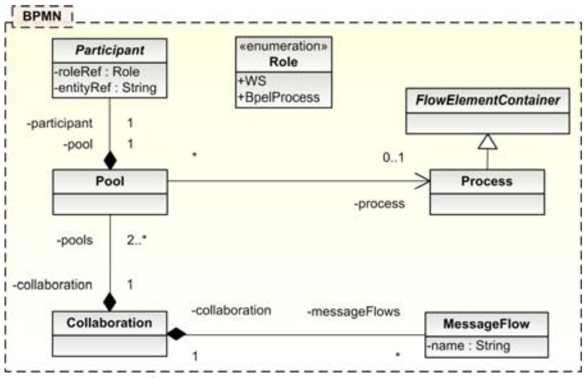


Figure 12. Core package in the rBPMN metamodel

In the Figure 13, we show the Process package of the rBPMN metamodel. The Process package contains classes which are used for modeling the flow of activities, events, messages, and how they are sequenced within a *Process*. This package consists from different elements (classes) from the BPMN metamodel:

- A *SequenceFlow* is used to model the transition of control from one *FlowElement* (the source) to another (the target). It determines the sequencing of *FlowElements* within a *Process* flow. *SequenceFlow* is represented with a solid line with black arrow between *Tasks* in Figure 1.
- *Activities* represent points in a *Process* flow where work is performed. They are the executable elements of a BPMN *Process*. The *Activity* class is an abstract element, the types of activities that are a part of a *Process* are: *Task*, *SubProcess*, and *CallActivity*. In Figure 13, we show only the *Task* element, as the most important activity type. A *Task* is an atomic *Activity* within a *Process* flow, which is used when the work in the *Process* cannot be broken down to a finer level of detail. Generally, an end-user and/or

applications are used to perform the *Task* when it is executed.

- The Process package also includes *Event*s and *Gateway*s. *Gateway*s are used to control how *SequenceFlow*s interact as they converge and diverge within a *Process*. If the flow does not need to be controlled, then a *Gateway* is not needed. The term "Gateway" implies that there is a gating mechanism that either allows or disallows passage through the *Gateway*. An *Event* is something that "happens" during the course of a *Process*. These *Event*s affect the flow of the *Process* and usually have a cause or an impact and in general require or allow for a reaction. In BPMN, there are different types of start, intermediate, and end events [19].
- *R2MLRule* is an element which we added in the Process package of the BPMN metamodel which actually represents an R2ML *Rule*. In this way, we enabled that an R2ML *Rule* (that is *Reaction*, *Derivation*, *Production* and *Integrity* rule) can be placed into a *Process* as a *Gateway*, but in the same time not to break R2ML *Rule* semantics. In Figure 13, we can see that *R2MLRule* as a *Gateway* can be connected by using *SequenceFlow* with other *FlowElement*s such as *Task*s, *Event*s, and *Gateway*s. This enables us to use rules in different places in rBPMN process models, as shown in workflow and service interaction patterns in Section 5.1.

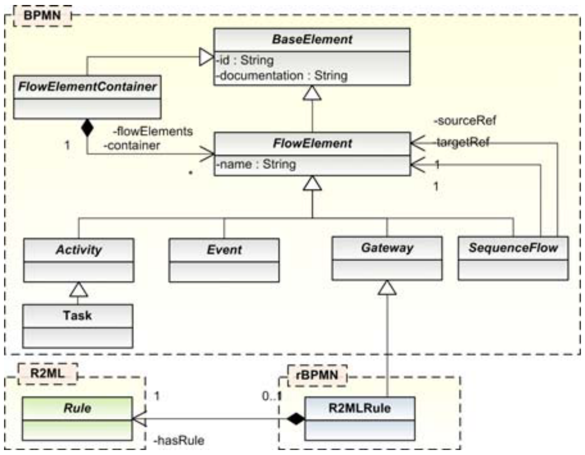


Figure 13. Process package in rBPMN metamodel

In the way presented in Figure 13, we can have a rule as a valid element in a business process, but we should also have a way to connect underlying data model to the business rule. In rBPMN, we use R2ML Vocabulary as an underlying data model (see Figure 14).

In the rBPMN metamodel, the *StructureDefinition* element is used to specify a *Message* structure. The *Message* is connected to the *StructureDefinition* through the *structure* relation, i.e., a *Message* can have exactly one *StructureDefinition*. We extended these BPMN elements by subclassing *StructureDefintion* with the *R2MLMessageType* that can have one R2ML *EventExpression,* through the relation *hasVocEntry*. In this way, rBPMN messages can be directly mapped to the *ReactionRule*'s triggering event or triggered event expression, and later into Web service WSDL descriptions [15], as we have described in Section 4.3.

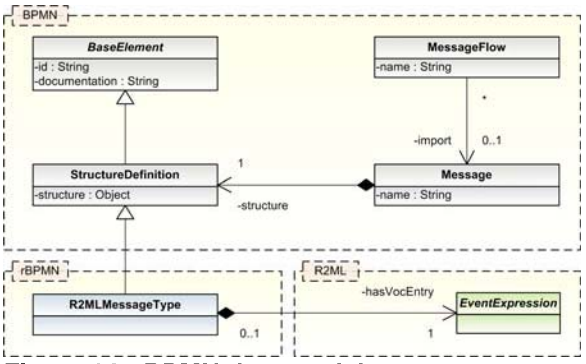


Figure 14. rBPMN data model

As described through Section 5.1, reaction rules can be mapped into Web service descriptions (i.e., MEPs), and we have defined that mapping [15]. The triggered event or triggering event messages of reaction rules defined in a rBPMN processes can be directly mapped (and transformed) into XML Schema elements (i.e., *complexTypes*) as XML Schema is used as a vocabulary in WSDL [15].

6. Related Work

Integration of rules and processes has been the subject of early investigation in the research community. While in [14], a merge of business processes and rules to improve the capturing of temporal information for information systems development was first introduced, in [16], the authors extended that approach by proposing a technique for associating reaction rules in a process modeling languages, by using the ERL rule modeling language. In that work, the authors presented only use of reaction rules which are used to make precise statements about some activity. Actually, they show how business processes can be translated into reaction rules, while in our approach different types of business rules can be modeled as a part of a business process in a design time and later whole process can be translated into some execution platform.

In [13], authors introduced a framework where some basic process flow constructs (such as parallel and sequence) could be represented by means of the reaction rules (Event-Condition-Action–ECA format). They also offer a methodology to stepwise refine these ECA rules to support the transition from the semi-formal process models to a formal workflow specification. The main idea of that approach is that business process can be decomposed into rules of the form event-condition-action-event. Our approach instead of decomposing rules into business rules makes business processes more flexible.

In [24], the authors present an integrated approach to service composition that covers the entire service composition life-cycle. This composition life-cycle can be divided into five broad composition phases that span abstract service definition, scheduling, construction and execution. The authors analyze the types of rules required for each phase and demonstrate how the rules can be used to drive the service composition process. Using composition rules, the authors construct a concrete service composition specification from basic composition elements. The authors try to make the whole business process modeling lifecycle more flexible by starting with basic composition elements (such as activities, condition, events) and using business rules (such as structure, data, constraint resource and exception rules) to assemble them into overall composition specification. As the result they get a dynamically-assembled Web Service composition. However, such composition is still static at run-time. Our approach make business processes more flexible, because in our approach changing rules in such process is able to do at run-time.

The authors of [6] propose a hybrid approach to Web service composition. They break down the composition logic into a core component, the process, and several well-modularized business rules that exist and evolve independently. They propose an aspect oriented extensions for BPEL (named AO4BPEL) for integrating rules with processes. This approach enables the mapping of business rules to aspects and weaving these aspects into the BPEL code by using an aspect-aware orchestration engine. This custom orchestration engine allows dynamically activating or deactivating aspects during process run-time and thus allows adapting the composition. In our approach, we do not use any special AOP technology for modeling or realization, but our solution is executable by using BPEL (extended with rule-based support [18]).

In [25], authors propose a new workflow management system language called DECLARE. It uses constraint-based process modeling language for developing declarative models. DECLARE models are mapped into a set of Linear Temporal Logic formulas (which adds several symbols to classical logical operators: always, eventually, until and next time). This language can detect conflicting errors and changing process during execution. DECLARE support only simple constraint specification and consider events regarding execution of activities, while rBPMN by using R2ML support more complex constraints. And also DECLARE processes cannot be mapped into service compositions. It does not have deadlines, and support only constraint templates with multiple parameters and only two activities can be supported, while rBPMN as BPMN can support more than two activities.

In [30], the author proposed a methodology for Agent-Oriented Business modeling and investigated the combination of reaction rules with activities and shown which of workflow patterns are supported by this combination. This methodology show only reaction rules, while in our solution we support all four types of rules and mappings to SOAs.

7. Conclusion

In this paper, we have proposed an approach to integration of business rules (R2ML) and business processes (BPMN), by using the MDE principles. In this approach, we created a new rule-based process modeling language called rBPMN by integration of metamodels of two languages (i.e., BPMN and R2ML), which we have defined by using metamodeling. By using rBPMN, it is possible to have rules as first-class citizens in a business process and to change these rules while processes are in run-time. We have shown how rules can be modeled directly as part of the business process models by introducing new advanced rule-based gateway called *rule gateway*. Besides this, we also provided a better integration of the state structure of a pool and of state conditions with the process definition, since the BPMN constructs for state structure modeling and for relating state conditions to control flow are rather weak. rBPMN has been proposed by bearing in mind an idea to have support methodology for developing rule-enabled SOAs, and a set of requirements to realize steps in the proposed methodology. We also presented a set of workflow and service interaction modeling patterns that our methodology supports, and how these patterns can be translated into SOAs (i.e., orchestrations and choreographies).

Currently, we are working on defining some additional example scenarios with other rule types, such as integrity and derivation rules. We also need to extend our rBPMN metamodel to support state conditions, action events, and rule post conditions. We are also working on rBPMN semantics, and we are currently considering the use of process algebra with support for condition composition [2].

References

1. Barros, A., Dumas, M., Hofstede, T., A., "Service Interaction Patterns: Towards a Reference Framework for Service-based Business Process Interconnection", *Technical Report FIT-TR-2005-02*, Faculty of Information Technology, Queensland University of Technology, Brisbane, Australia, March 2005.
2. Bergstra, A., J., Ponse, A., "Process algebra and conditional composition", *Inf. Process. Lett.* 80, 1 (Oct. 2001), 41-49.
3. Bézivin, J., "On the unification power of models", *Software and System Modeling*, vol. 4, no. 2, pp. 171–188, 2005.
4. Brahe, S., Osterbye, K., "Business Process Modeling: Defining Domain Specific Modeling Languages by Use of UML Profiles," in ECMDA-FA 2006, lNCS 4066, pp.241-255, 2006.
5. Business Rules Group. The business rules manifesto. The principals of rules independence. http://www.businessrulesgroup.org/brmanifesto.htm, 2003.
6. Charfi, A., Mezini, M., "Hybrid web service composition: business processes meet business rules", In *Proceedings of the 2nd international conference on Service oriented computing*, New York, NY, USA, pp.30-38, 2004.
7. Erl, T., Service-Oriented Architecture: Concepts, Technology, and Design, Prentice Hall PTR, 2005.
8. Ginsberg, A., "RIF Use Cases and Requirements," W3C Working Draft, http://www.w3.org/TR/rif-ucr/, 2006.
9. Guirca, A., Lukichev, S., Wagner, G., "Modeling Web Services with URML", *In Proceedings of Workshop Semantics for Business Process Management*, 2006.
10. Graml, T., Bracht, R., Spies, M., "Patterns of Business Rules to Enable Agile Business Processes", *In proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007)*, Annapolis, USA, 2007, pp. 365-375.
11. IBM Developerworks, "Business process execution language for web services, version 1.1.", online: http://www-128.ibm.com/developerworks/library/specification/wsbpel/, 2003.
12. Kavantzas, N., Burdett, D., Ritzinger, G., Fletcher, T., Lafon, Y, "Web Services Choreography Description Language Version 1.0", *W3C Candidate Recommendation*, November 2005.
13. Knolmayer, G., Endl, R., Pfahrer, M., "Modeling processes and workflows by business rules", *In Business Process Management*, pp. 16–29, 2000.
14. Korgstie, J., McBrien, P., Owens, R., Selveit, H., A., "Information Systems Development using a Combination of Process and Rule-based Approaches", in *Third Nordic Conference of Advanced Information Systems Engineering:* LNCS, Springer-Verlag, 1991.
15. Lukichev, S., Giurca, A., Wagner, G., Gaševic, D., Ribaric, M., "Using UML-based Rules for Web Services Modeling," *In Proceedings of the 2nd Int'l Workshop on Service Engineering at the 23rd Int'l Conference on Data Engineering*, pp. 290-298., 2007.
16. McBrien, P., Seltveit, H., A., "Coupling Process Models and Business Rules", In Proceedings of the IFIP 8.1 WG Conference, Chapman Hall, Pages 201-217, 1995.
17. Miller, J., Mukerji, J., (eds.) "MDA Guide Version 1.0.1", OMG, 2003.
18. Nagl, C., Rosenberg, F., & Dustdar, S. "VIDRE– A distributed service oriented business rule engine based on RuleML", In Proc. of the 10th IEEE International Enterprise Distributed Object Computing Conference (pp. 35–44), 2006.
19. Object Management Group, "Business Process Model and Notation (BPMN) Specification 2.0", initial submission, http://www.omg.org/cgi-bin/doc?bmi/08-02-06, 2008.
20. Object Management Group, Meta Object Facility (MOF) Core, v2.0, OMG Document formal/06-01-01, http://www.omg.org/cgi-bin/doc?formal/2006-01-01, 2005.
21. Object Management Group, MOF QVT Final Adopted Specification, OMG document 05-11-01, http://www.omg.org/docs/ptc/05-11-01.pdf, 2005.
22. Object Management Group, Object Constraint Language, OMG Specification, Version 2.0, formal/06-05-01, http://www.omg.org/docs/formal/06-05-01.pdf, 2006.
23. Object Management Group, Unified Modeling Language 2.0, Docs. formal/05-07-04 & formal/05-07-05, 2005.
24. Orriëns, B., Yang, J., "A Rule Driven Approach for Developing Adaptive Service Oriented Business Collaboration", In *Proceedings of the IEEE International Conference on Services Computing*, pp. 182 - 189, 2006.
25. Pesic, M., Schonenberg, H., van der Aalst, M. P., M., Wil, "DECLARE: Full Support for Loosely-Structured Process", In Proceedings of the 11th IEEE international Enterprise Distributed Object Computing Conference (October 15 - 19, 2007). EDOC. IEEE Computer Society, Washington, DC, 287, 2007.
26. Recker, J., Indulska, M., Rosemann, M., Green, P., "How Good is BPMN Really Insights from Theory and Practice", in *Proceedings 14th European Conference on Information Systems*", Goeteborg, Sweden, 2006.
27. REWERSE I1 Rule Markup Language (R2ML). http://oxygen.informatik.tu-cottbus.de/rewerse-i1/?q=node/6, 2008.
28. Seidewitz, E., "What Models Mean", IEEE Software, pp. 26-32, 2003.
29. Steinke, G., Nikolette, C., "Business rules as the basis of an organization's information syste*m", Industrial management + Data Systems*, vol. 103, p. 52, 2003.
30. Taveter, K. "A Multi-Perspective Methodology for Agent-Oriented Business Modelling and Simulation", PhD Thesis, Tallinn University of Technology, 2004.
31. Wagner, G., Giurca, A., Lukichev, S., "A Usable Interchange Format for Rich Syntax Rules Integrating OCL, RuleML and SWRL", *In Proceedings of WWW2006 conference*, Edinburgh, UK, 2006.
32. Wohed, P, van der Aalst, W., M., P., Dumas, M., ter Hofstede, W., M., P., Russell, N., "Pattern-based Analysis of BPMN", *BPM Report BPM-06-17*, BPMcenter.org, 2006.

Normative Ontologies for Data-Centric Business Process Management

Iman Poernomo
Department of Computer Science
King's College London
Strand, London, UK, WC2R2LS
iman.poernomo@kcl.ac.uk

Timur Umarov
Department of Computer Science
King's College London
Strand, London, UK, WC2R2LS
timur.umarov@kcl.ac.uk

Abstract

This paper addresses the problem of describing and analyzing data manipulation within business process workflow specifications. We apply a model-driven approach. We begin with business requirement specifications, consisting of an ontology and an associated set of normative rules, that define the ways in which business processes can interact. We then transform this specification into a Petri Net workflow model and, separately, an Event B specification. The former models can be submitted to further behavioural analysis to ensure, for instance, satisfaction of liveness and safety properties. The latter specifications are important as we can use theorem proving techniques to check and refine data representation with respect to process evolution. An important property of the transformation is semantic equivalence between the Petri net model and Event-B model.

Keywords—*Petri nets, MEASUR, Business Process Management, Formal Specification, Semantic Augmentation*

1. Introduction

Business process management (BPM) is an increasingly challenging aspect of the enterprise. Middleware support for BPM, as provided by, for example, Oracle, Biztalk and the recent Windows Workflow Framework, has met some challenges with respect to performance and maintenance of workflow.

The central challenge to BPM is complexity: business processes are becoming widely distributed, interoperating across a range of inter- and intra-organizational vocabularies and semantics. It is important that complex business workflows are checked and analyzed for optimality and trustworthiness prior to deployment. The problem becomes worse when we consider the enterprise's demand to regularly adapt and change processes. For example, the growth of a company, changes to the market, revaluation of tasks to minimize cost. All these factors often require reengineering or adaptation of business processes along with continuous improvement of individual activities for achieving dramatic improvements of performance critical parameters such as quality (of a product or service), cost, and speed [16]. Reengineering of a complex workflow implementation is dangerous, due to existing dependencies between tasks.

Formal methods can assist in meeting the challenge of complexity, as their mathematical basis can assist in analysing and refining a system specification. However, complex systems often involve a number of different aspects that entail separate kinds of analysis and, consequently, the use of a number of different formal methods.

Petri nets are a formal method that has successfully assisted in workflow design and analysis. While Petri nets are good for expressing the dynamics of a workflow, the representation of data as tokens do not provide the full depth of specification necessarily to by developers. Petri nets model the *possible flow* of information in a business process, but do not specify the nature of the information nor how information is to be manipulated during the business process. Petri net lack modeling power and mechanisms for data abstraction and refinement [6].

In contrast, a business process implementation within a BPM middleware requires detailed treatment of both information flow and information content. The abstraction gap is identified by Hepp and Roman in [9]: an abstract workflow that ignores information content provides an abstract view of business processes that does not fully define the key aspects necessary for BPM implementation.

We argue that this abstraction gap can be addressed by developing *semantically compatible* PN models and data models from an initial business process requirements specification. We employ a Model Driven Architecture approach.

The overall framework is depicted in Fig. 1. For our purposes, we consider transformations between models of three languages, a Computation-Independent Model (CIM) and two Platform Independent Models (PIMs). The CIM is an initial model of business requirements. It describes business functionality without treating any architectural or computational aspects of the system implementation. The two PIMs describe complementary aspects of the overall structure of the system to be implemented: workflow descriptions from PN models and data and data exchange mechanisms from Event B specifications.

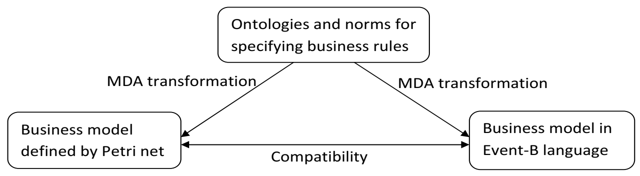

Ontologies and norms for specifying business rules
MDA transformation
MDA transformation
Business model defined by Petri net
Compatibility
Business model in Event-B language

Figure 1. The data-centric workflow framework.

The initial CIM might be written as models within a number of requirements specification frameworks. We use the ontologies and normative language of the MEASUR method [12]. The method has a 20 year history and is widely used within the organizational semiotics community, but less well-known in Computer Science. Its roots lie in the philosophical pragmatism of Pierce, the semiotics of Saussure and Austin's speech act theory. It is model-based, with ontologies and normative constraints forming the central deliverables of a requirements document. We employ MEASUR notation because our starting point is information systems analysis, where MEASUR has found the most application. We have also found its normative constraints lend themselves to transformation into our PIM languages. However, our approach should be readily adaptable to a number of similar notations in use in the multi-agents and normative specification research communities.

The Event B language is used in specifying, designing, and implementing software systems. The language may be used to develop software by a process of gradual refinement, from an abstract, possibly nonexecutable system model, to intermediate system models that contain more detail on how to treat data and algorithms, to a final, optimised, executable system. In this process,

- the first abstract model in this refinement chain should be verified for consistency and
- each step in the refinement chain should be formally checked for semantic preservation.

Consistency will then be preserved throughout the chain. This means that the final executable refinement can be trusted to implement the initial abstract specification. This is the main reason why we are using Event-B method: once we have an initial abstract model of the system we can refine it to get more concrete and executable model. The efficiency of using formal methods in software development is proven by significantly low number of errors in a developed system and high degree of reliability. Development-by-proving approach allows develop more efficient and less error-prone software system.

Our approach defines a transformation MEASUR models to

- PN models, permitting the usual workflow analysis results
- Event B machines, permitting
  - a full B-based formal semantics for vocabularies and data manipulation that is carried out within the modelled workflow, which can be validated for consistency.
  - an initial, abstract B model that can be further refined using the B method to a final optimal executable system in an object-oriented workflow middleware, such as Windows Workflow Foundation.

A notion of semantic compatibility holds over the transformed models, so that any property derived over the PN view of the system will hold over potential processes that arise from the Event B machine.

The paper proceeds as follows:

- In section 2, we sketch the nature of our CIM, the normative ontology language of MEASUR.
- Section 3 provides a brief introduction to Event B specifications, focusing on the main points relevant to our formal semantics and the notion of semantic consistency between PNs and Event B specifications.
- Section 4 then outlines the transformation approach to generating B specification and Petri nets from our ontologies. We discuss how the resulting specification provides a formal semantics of our data-centric business process, and how this enables consistency validation checks.
- Section 5 discusses related work and conclusions.

2. MEASUR models

The MEASUR can be used to analyse and specify an organization's business processes via three stages [12]:

1. Articulation of the problem, where a business requirements problem statement is developed in partnership with the client.
2. Semantic Analysis, where the requirements problem statement is encoded as an ontology, identifying the main roles, relationships and actions.

3. Norm Analysis, where the dynamics of the statement are identified as social norms, deontic statements of rights, responsibilities and obligations.

Space does not permit us to detail the first stage. Its processes are comparable to other well known approaches to requirements specification. The last two stages require some elaboration. For our purposes, they provide a Computation Independent Model, consisting of an ontology and collection of norms, that formally define the structure and potential behaviour of an organization and its processes. We hereafter refer to the combination of an MEASUR ontology and associated norms as a *normative ontology*.

2.1. Ontologies

The ontologies of semantic analysis are similar to those of, for example, OWL, decomposing a problem domain into roles and relationships. As such, our ontologies enable us to identify the kinds of data that are of importance to business processes. A key difference with OWL is the ability to directly represent *agents* and *actions* as entities within an ontology. This is useful from the perspective of business process analysis, as it enables us to identify tasks of a workflow and relate them to data and identify what agent within the organization has responsibility for the task.

Semantic Analysis has its roots in semiotics, the philosophical investigation of signs. MEASUR applies to information system analysis a number of ideas and approaches from philosophy of language, drawing on the pragmatism of Pierce, semiotics of Saussure and the epistemology of Wittgenstein and Austin. The method's core assumption is knowledge and information exists only in relation to a knowing *agent* (a single human or a social organization). There is no Platonic reality which defines Truth. Instead, Truth is a derived concept that might be defined as *agreement* between a group of agents. An agent is *responsible* for its knowledge. When a group of agents agree on what is true and what is false, they accept responsibility for that judgement. Following Wittgenstein, MEASUR considers an information system as a "language game", a form of activity involving a party of agents that generates meaning. In an information-system-as-language-game, the meaning of data derives from usage by agents, rather than from a universal semantics.

Semantic Analysis represents the information system as language game in the form of an ontology diagram, identifying agents, the kinds of actions agents can perform and the relationships and forms of knowledge that can result from actions.

These concepts are identified as types of *affordance*. An affordance is a collection of patterns of behaviour that define an object or a potential action available to an agent. Every concept in a MEASUR ontology is an affordance.

MEASUR subclasses the notion of affordance as follows. A *business entity* – such as a user account or a bank loan – is an affordance in the sense that it is associated with a set of permissible behaviours and possibilities of use. For the purpose of business process analysis, business entities are used to identify the main kinds of data that are of importance in an organization's processes. A *relationship* – such as a contract – between business entities or agents is an affordance in the sense that is is defined by the behaviour it generates for the parties involved in the contract. *Agents* are affordances in terms of the actions they can perform and the things that may be done to them. Agents then occupy a special status in that they take responsibility for their own actions and the actions of others and can authorize patterns of behaviour. The structure of a business entity, relationship or agent is given via a list of associated properties, called *determiners*. Determiners are properties and attributes of affordances, such an address or telephone number associated with a user account. *Units of measurement* are typical data types that type determiners and other values associated with affordances. The latter two concepts are considered as affordances as their values constrain the possible behaviour of their owners.

In our treatment, affordances can be treated as types of things within a business system, with an ontology defining a type structure for the system. An actual executing system consists of a collection of affordance *instances* possess the structure prescribed by the ontology and obey any further constraints imposed an associated set of norms.

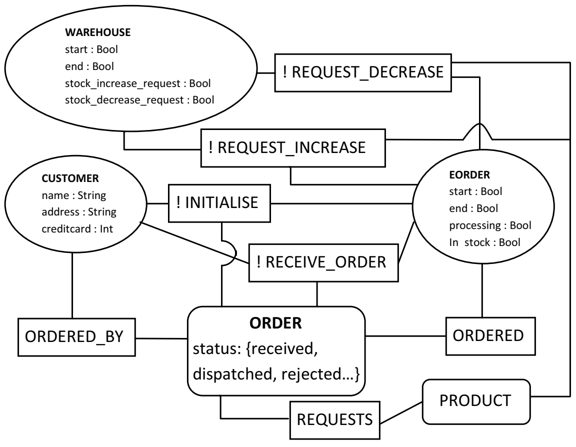


Figure 2. Example normative ontology

Example 2.1 *The ontology for a purchasing system is given in Fig. 2. Agents are represented as ovals and business entities as rectangles with curved edges. Communication acts and relations as rectangles, with the former differentiated by the use of an exclamation mark* ! *before the act's name.*

All affordances (including agents and business entities) have a number of typed attributes, defining the kinds of states it may be in. We permit navigation through an affordance's attributes and related affordances in the object-oriented style of the OCL.

The system involves processes that cross the boundaries of two subsystems: an order processing system, and a product warehouse system. These two subsystems are represented as agents in the ontology, eOrder *and* ProductWarehouse*, respectively. By default all agents contain start and end attributes.*

Orders *are* requests *for* products*, both represented as entities in the ontology with a* requests *relationship holding between them (multiplicities could be associated with the relationship to define the possibility of a number of products contained in an order). A customer can initialise the order processing system with a given order, denoted by a communication act* !immortalize *between the corresponding* customer *and* eOrder *agents. An order is associated with its customer, defined by the* ordered_by *relationship holding between the* customer *agent and* order *entity. An order can stand in an* ordered *relationship with the* eOrder *agent, after it has been successfully processed.*

Another communication act, !receive_order*, corresponds to the initial reception of data.*

2.2. Norms

Norms are constraints and rules that determine how agents interact and control affordances. They also control the initialization and termination of particulars (affordance instances).

We have adopted a typed language of deontic and normative logic to express logical constraints over business processes, using ontologies as atomic classes, relations, objects and actions for the logic. Our constraints take the form

$$A,B := R(\bar{a}) \mid \neg A \mid A \vee B \mid A \wedge B \mid A \rightarrow B \mid \forall x : C.A(x) \mid \exists x : C.B(x) \mid ObA \mid PA \mid NPA \mid E_xA \quad (1)$$

where C is an affordance (that acts as a type of a particular instance); $R(\bar{a})$ is an affordance with one or two antecedents $\bar{A}$ and $\bar{a}$ is one or two particular instances of $\bar{A}$; the meaning of ObA is that A is obliged to happen; the meaning of PA is that A is permitted to happen; the meaning of NPA is that A is prohibited to happen; the meaning of E_xA is that A results from, and is the responsibility of, agent particular x; the meaning of the other connectives follows standard first order logic.

A *behavioural* norm is the general form for a constraint over our ontologies, and has the following form (Liu, 2000):

$$\text{Trigger} \rightarrow \text{pre-condition} \rightarrow E_{\text{agent}} Ob/P/NP\,\textit{post-condition} \quad (2)$$

The informal meaning of the norm might be written:

if *Trigger* occurs and
the *pre-condition* is satisfied,
then *agent* performs an action so that
post-condition is
Obliged/Permitted/Prohibited from resulting

The idea of a behavioural norm is to associate knowledge and information with agents, who produce and are responsible for it. From a philosophical perspective, truth is then defined as something that an agent brings about and is responsible for.

As shall be seen, from the perspective of determining how to *implement* a normative ontology as a workflow-based system, we view agents as corresponding to subsystems, business entities to specify data and behavioural norms to expected dynamic interaction protocols between subsystems.

Example 2.2 *Consider the communication act* !receive_order *from our example, corresponding to the initial reception of data by the order processing system. The idea that this reception can only occur over orders that are not yet processed is captured by the following behavioural norm:*

$$\forall\, oo : Order.\forall\, e : eOrder.\neg ordered(oo,e) \rightarrow E_e\ Ob\ receive\_order(oo,e) \quad (3)$$

Both relationships and communication acts are represented as logical relations in our language, but communication acts are not used in pre-conditions, and may only be placed after a Deontic operator.

Communication acts often define resulting changes of state on related agents and entities. We define them as further definitions of norms which contain expressions semantically equivalent to the effects of communication acts. In this case, the reception of an order entails a change of state in the order (its status becomes set to "received") and order processing system (its processing attribute is set to true). This norm definition is depicted below:

$$\forall\, oo : Order.\forall\, e : eOrder.receive\_order(oo,e) \rightarrow ordered(oo) \wedge oo.status = received \wedge e.processing = true \quad (4)$$

When the eOrder system is processing an order, it will request an increase in the stock from the warehouse. This is prescribed by the following norm:

$$\forall\, oo : Order.\forall\, e : eOrder.\forall\, p : ProductWarehouse. \neg e.processing = true \rightarrow E_e\ Ob\ request\_increase(oo,p)$$

where the communication act entails a change of state in the warehouse subsystem:

$$\forall oo : Order.\forall e : eOrder.\forall p : ProductWarehouse.\ request\_increase(oo, e) \rightarrow p.stock\_increase\_request = true$$

A number of other norms are required to define the entire business process. For example, after processing this information the system sends response to the customer depending on data provided. If the data are valid, then system sends an appropriate availability request to the warehouse. If the product is available, it is sent to the customer. If not validated, the order is rejected. The structure of the norms

There have been a number of attempts to use semantic analysis normative ontologies as the language for a business process management engine. The most widely used is Liu's NORMBASE system [12]. In such systems, the ontology serves as a type system for data, while norms define the conditions under which tasks may be invoked to create and manipulate data.

Our approach is different: we treat normative ontologies as a useful and semantically rich requirements analysis document. However, we intend to implement these requirements using a standard business process management infrastructure. We believe that further refinement and analysis a necessary step to this goal. In particular, it is important to ensure that

- the possible communication act traces permitted by a set of norms do not deadlock unexpectedly (in our example, this might happen if the order processing system waits indefinitely for a response from the warehouse that stock is available);
- the ontology and its associated norms do not allow for an inconsistent state of the system (in our example, this happens if an action entails that an order is processed and rejected at the same time).

The first kind of error can be removed by providing the normative ontology with a Petri Net representation and applying standard behavioural analysis techniques. The second kind of error can be eliminated by checking and ensuring that our generated invariants and guard conditions of the machines are not overlapping.

3 Event B and Petri nets

This section provides an overview of the Event B notation. We define the notion of semantic consistency between Petri Nets and Event B specifications.

3.1. Event B

Event B specifies a software system in terms of encapsulated modules, called machines, that consist of a mutable state and a number of related operations, called events, whose execution changes the values of the state. Each event consists of a logical *guard* and an *action*. The guard is a first order logical statement about the state of the machine and defines the conditions under which an action may occur. The action defines the way the machine's state may be modified as a first-order logical statement relating the initial values of the state prior to the action occurring and the final values of state.

Machines therefore have a formal operational semantics, that models system execution as a sequence of events. If an event's guard holds over the machine's state, its action may be executed. This will change machine's state, which may cause another event's guard to hold, and an action to be executed. The sequence continues until the system has halted (it is deadlocked). Note that execution is potentially nondeterministic: when a number of event guards are true, then *one* of the corresponding event actions is chosen at random.

A common requirement over business process descriptions is the preservation of certain properties throughout the whole course of execution of events. These properties are called *invariants*: they represent predicates built on the state variables that must hold permanently. This is achieved by proving that under this invariant and guards of events, the invariant still holds after modifications made to the state variables associated with event executions.

Fig. 3 demonstrates the structure of an Event-B model. Every model written in Event-B is represented as a machine/context pair. The relationship between these two constructs is that the machine "sees" the context (read-only access, with no modification possible). The context contains the main sets, constants, axioms, and theorems. Carrier sets and enumerated sets are declared in the Sets section. Since, an enumerated set is a collection of elements, additionally, its members are defined as constants in the Constants section. An axioms section contains assignments of the names of each enumerated set to its values and declaration of rules according to which constants are defined as not being equal to each other. There can also be one or several theorems defined in the Context of a model definition.

A machine consists of state variables, invariants, and events. State variables represent states which the machine can be in. The Invariants box is comprised of the conditions that should hold throughout the whole execution of the machine. The events box contains the initialisation construct and all events of the machine. Each event contains one or several guards and one or several actions.

Definition 3.1 (Consistent B Machine) *An event B ma-*

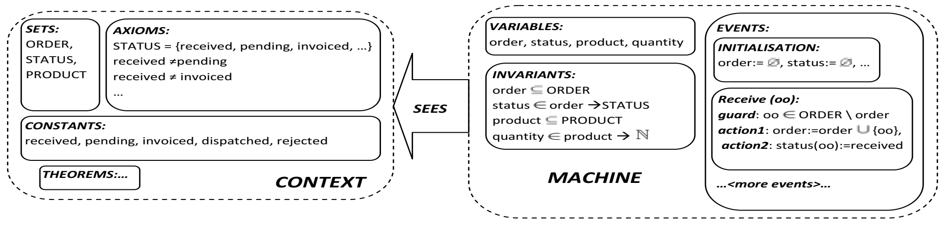


Figure 3. The structure of an abstract Event-B model.

chine is consistent if the following conditions hold:

- *Invariant preservation : for any event, assuming the invariant and guard are true, then the invariant and action are consistent (do not result in a contradiction).*
- *Feasibility: given any event, if the guard holds, then it is possible for the action to be performed.*

It is possible to define an operational semantics for event B machines, over which the runtime execution of the modelled system can be understood. Essentially, this is done by assuming the initialization constraints to hold over the state of the machine (actual values assigned to its set of variables), and then successively selecting events based on guard checks over the variables. Each event selection will result in the action condition changing the state of the system. The resulting sequence of events is a *trace* of the machine. A machine will usually have a potentially infinite number of traces, due to the nondeterminism of guard selection (and the nondeterminism within actual actions, which space does not permit us to discuss here).

B machines do not lend themselves easily to simulation. This is because guards and actions are first order logical formulae and, consequently, the selection of a guard and the determination of how an action affects state is not decidable and requires human proof. Simulation of the possible ordering of tasks in a workflow are better handled via a Petri Net specification.

3.2. Petri nets and semantic compatibility

Space does not permit a full overview of the Petri Net notation. The reader is referred to, for example, [16] for an detailed introduction. A Petri net specifies a business process workflow in terms of *places*, representing a main business activity, tokens, representing some data or document that can be passed between activities, *transitions*, representing permitted flow of data between activities. Directed arcs are used to related transitions to places. An example Petri Net is given in Fig. 4: places are denoted by circles and transitions as rectangles. Petri nets permit the usual business process workflow notions of joins and forks from transitions to places, allowing us to represent parallel and synchronizing processes and nondeterministic choice.

An Event-B machine is *semantically compatible* with a PN if there is a bijection from transitions of the PN to events of the Event-B machine such that all possible traces of the machine's events correspond to possible traces of transitions in the PN, and vice versa. That is, a machine is compatible with a PN if the PN simulates every possible sequence of events of the machine, and vice versa.

4 Semantic Embedding of Normative Ontologies in Event-B

This section describes in detail the actual mapping approach that was used to implement the transformation.

4.1 General Mapping Strategy

We now sketch our transformations from normative ontologies to PN and Event-B machines. The purpose of the transformations is threefold: 1) it provides a formal semantics for MEASUR's normative ontologies that can be analysed for consistency and correctness using B-based tools, 2) it serves to produce the first B model in a chain of refinements that leads to a final implementation and 3) Petri net model obtained from normative ontologies can be further formally analysed for liveness errors.

We have implemented our transformations using Kermeta metamodelling language.

The transformation from normative ontologies to Petri nets focuses on identifying the traces of communication acts that are possible from a collection of norms. The transformation involves *discarding* information about changes to states of entities and agents, and mapping communication acts to places. The transformation is *interactive*, not automatic, requiring a domain expert to identify the type of ordering expected to hold between communication acts. This is then used to define the possible transitions between places.

The domain expert need not be an expert in formal methods or workflow implementation – our transformation might be incorporated at the requirements analysis stage immediately after the normative ontology has been developed. The resulting Petri Nets can be seen as a further refinement of the normative ontology requirements, further constraining workflow enabled by the norms.

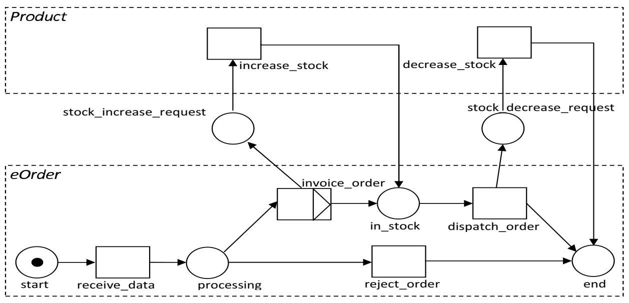


Figure 4. Petri net model for eOrder

Example 4.1 *A Petri net model obtained from our example via the interactive transformation depicted in Fig. 4. The process eOrder is comprised of four main* transitions*, receive_data, invoice_order, dispatch_order, and reject_order, and four* places*, start, processing, in_stock, and end. The process begins with the place "start" by containing a token in it. This fires the transition "receive_data" after which the token transfers to the next place "processing". This place contains an implicit condition which selects which transition to fire next. If the total price of the purchase is lower than allowed credit limit then "invoice_order" transition fires, otherwise the transition "reject_order" fires. If the ordered products are in stock the token moves to the place "in_stock". The presence of the token in the place "in_stock" causes the transition "dispatch_order" to fire. Otherwise, if the product is currently not available in the stock then the token moves to the place "stock_increase_request" after which the transition "increase_stock" fires which starts an appropriate subprocess for increasing the stock and returns the token to the place "in_stock". Successful execution of the transition "dispatch_order" decreases the stock and ends the process.*

Normative ontologies are mapped to a B machine in the following way.

The mapping of affordances is straightforward. Actors are mapped to machines. Business entities and relations are mapped to Event-B sets and relations.

The transformation of normative constraints is more difficult. Conceptually, norms of the form (2) appear similar in form to an machine event:

- A *trigger* and *pre-condition* correspond to a *guard*. The former define the situation that must hold before an agent can act. The latter defines the state that must hold before a machine can perform an action.
- The responsibility modality E_a corresponds to the location of the event within the machine corresponding to agent a.
- The deontic modality $Ob/Ppost-condition$ identifies whether the action corresponding to $post-condition$ should be necessarily performed, or whether execution of another (skip action) is possible instead. The *NP* deontic modality means the negation of the post-condition holds.

Because the normative constraints are essentially abstract business rules, while the conditions of the B machine define further implementation-specific detail, the mapping will depend on how we interpret relations and functions of the ontology. For this purpose our transformation must be based on a given semantic mapping of individual relations and functions to B relations and functions. We assume this is defined by a domain expert with the purpose of wide reusability for the ontology's domain.

Fig. 5 shows an excerpt of the metamodel for normative ontology written in Kermeta (the source model). The

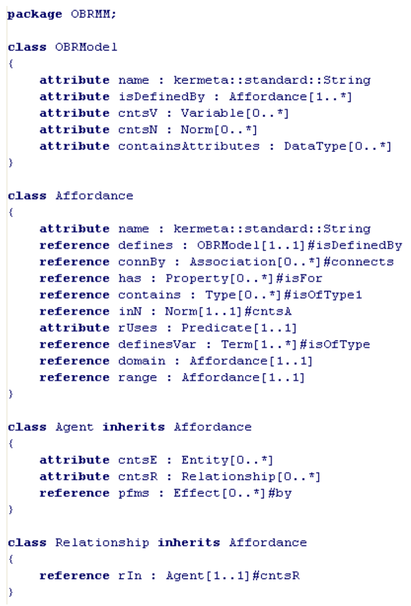
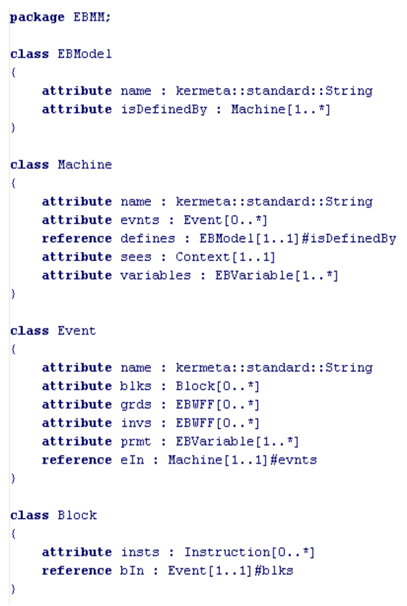

```
package OBRMM;

class OBRModel
{
    attribute name : kermeta::standard::String
    attribute isDefinedBy : Affordance[1..*]
    attribute cntsV : Variable[0..*]
    attribute cntsN : Norm[0..*]
    attribute containsAttributes : DataType[0..*]
}

class Affordance
{
    attribute name : kermeta::standard::String
    reference defines : OBRModel[1..1]#isDefinedBy
    reference connBy : Association[0..*]#connects
    reference has : Property[0..*]#isFor
    reference contains : Type[0..*]#isOfType1
    reference inN : Norm[1..1]#cntsA
    attribute rUses : Predicate[1..1]
    reference definesVar : Term[1..*]#isOfType
    reference domain : Affordance[1..1]
    reference range : Affordance[1..1]
}

class Agent inherits Affordance
{
    attribute cntsE : Entity[0..*]
    attribute cntsR : Relationship[0..*]
    reference pfms : Effect[0..*]#by
}

class Relationship inherits Affordance
{
    reference rIn : Agent[1..1]#cntsR
}
```

Figure 5. Metamodel for a normative ontology

main containing metaclass is OBRModel that is mainly defined by affordances, which can effectively be an agent, a relationship, an entity, a communication act, etc. Fig. 6 depicts an excerpt of the metamodel for an Event-B machine written in Kermeta (the target model). The main containing metaclass in this metamodel is EBModel which is defined



```
package EBMM;

class EBModel
{
    attribute name : kermeta::standard::String
    attribute isDefinedBy : Machine[1..*]
}

class Machine
{
    attribute name : kermeta::standard::String
    attribute evnts : Event[0..*]
    reference defines : EBModel[1..1]#isDefinedBy
    attribute sees : Context[1..1]
    attribute variables : EBVariable[1..*]
}

class Event
{
    attribute name : kermeta::standard::String
    attribute blks : Block[0..*]
    attribute grds : EBWFF[0..*]
    attribute invs : EBWFF[0..*]
    attribute prmt : EBVariable[1..*]
    reference eIn : Machine[1..1]#evnts
}

class Block
{
    attribute insts : Instruction[0..*]
    reference bIn : Event[1..1]#blks
}
```

Figure 6. Metamodel for an Event-B machine

by machine, set of events and variables. Events are defined by well-formed formulas for preconditions and blocks of instructions for actions.

4.2 Mapping Rules

Mapping from normative ontologies to Event-B machines consists of several rules to be implemented in the transformation φ. The following rules are used:

$$R : \mathit{Entity} \times \mathit{Agent} \mapsto_{\varphi} R'_{A'}, \tag{5}$$

where R is from $\mathit{Entity} \times \mathit{Agent}$ superset and represents a relationship type between *Entity* and *Agent*, R maps to R', which represents a set variable inside the machine A';

For any variable a from the entity E

$$a : E \mapsto_{\varphi} a' \in E', \tag{6}$$

a maps to a' which is a variable from the generated set E';

Predicate P of the form $P(a,b)$ maps to a local variable a from the set variable P:

$$P(a,b) \mapsto_{\varphi} a \in P, \tag{7}$$

where b is used for identifying the machine, which contains the set variable;

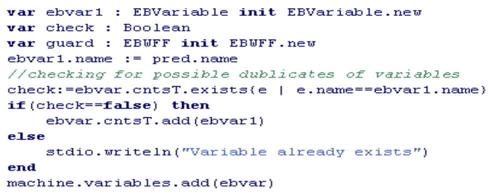
$$\neg F \mapsto_{\varphi} \neg\varphi(F) \tag{8}$$

According to the rule (5) any given relationship r holding between a given entity e and a certain agent a maps to a set variable r' in the set e' of the machine a'. For example, let us consider the following expression of a norm:

$$\neg \mathit{ordered}(oo : \mathit{ORDER}, e : \mathit{EORDER})$$

This statement declares that a particular order *oo* has not been ordered, or is not yet in the system (agent) *EORDER*. We can transform this expression to several constructs and expressions in Event-B. By applying rule (5) we first generate a set (state) variable *ordered* and machine *EORDER* (if it is not already created).

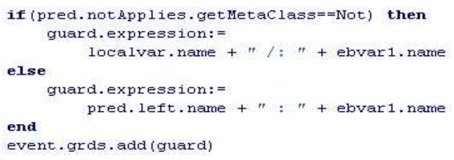
The following Fig. 7 – 12 show the excerpts of algorithms of transformation implementation. Fig. 7 depicts the transformation algorithm for this mapping.



```
var ebvar1 : EBVariable init EBVariable.new
var check : Boolean
var guard : EBWFF init EBWFF.new
ebvar1.name := pred.name
//checking for possible dublicates of variables
check:=ebvar.cntsT.exists{e | e.name==ebvar1.name}
if(check==false) then
    ebvar.cntsT.add(ebvar1)
else
    stdio.writeln("Variable already exists")
end
machine.variables.add(ebvar)
```

Figure 7. Algorithm for set variables

By applying rule (6) we generate a "general" guard $oo \in \mathit{ORDER}$ which declares local variable *oo* of type *ORDER*. Applying rule (7) will associate local variable *oo* with the set (state) variable *ordered*: $oo \in \mathit{ordered}$. Fig. 8 represents



```
if(pred.notApplies.getMetaClass==Not) then
    guard.expression:=
        localvar.name + " /: " + ebvar1.name
else
    guard.expression:=
        pred.left.name + " : " + ebvar1.name
end
event.grds.add(guard)
```

Figure 8. Algorithm for set variables

the algorithm for this mapping rule. It considers both cases: with and without negation. Since our example expression contains "$\neg$" sign, which means that the order *oo* has not been ordered, then (applying rule (8)) the resultant expression will effectively take the form of $oo \notin \mathit{ordered}$. This *guard* will reside in the event which is generated from the behavioural norm (3) following the responsibility modality. The content of the event is generated from the behavioural norm definition (4), which is semantically equivalent to the meaning of the effect of the norm. The norm's expression $e : \mathit{EORDER}$ is used only to identify the responsible agent that maps to a corresponding machine where the event resides. Hence, we are not applying the rules to this statement. Fig. 9 shows how we assign names to the newly created events from predicates' type information. For example, a predicate *receive_order* is of type COMMUNICATION

```
if(pp.getMetaClass==Implies) then
    if(pp.E.action.getMetaClass==DeonticWFF) then
        var deontic : DeonticWFF
        deontic?=pp.E.action
        if(deontic.modality=="Ob") then
            deontic.cntsWFF.cntsP.each(pp |
                //assigning event names
                event.name := pp.isOfType.isOfType1.name
            )
        end
    end
else if(pp.getMetaClass==ForAll) then
  ...
```

Figure 9. Algorithm for norms' effects

ACT RECEIVE_ORDER, taken from normative ontology (see Fig. 2), generates a new event with the same name, given that the deontic modality is "Ob". These newly created events from normative ontologies are added to the ma-

```
else if(pp.getMetaClass==ForAll) then
    var check : Boolean
    pp.cntsT.each(tt |
    if(tt.isOfType.getMetaClass==Entity) then
        localvar.name := tt.name
    else if(tt.isOfType.getMetaClass==Agent) then
        if(nn.cntsA!=CommunicationAct) then
            var checkMachine : Machine init Machine.new
            checkMachine.name := tt.isOfType.name  + "_M"
            check:=result.isDefinedBy.exists(e |
                //checking for possible dublicates of machines
                e.name==checkMachine.name
            )
            if(check==false) then
                checkMachine.name := tt.isOfType.name  + "_M"
                context.name := tt.isOfType.name + "_C"
                checkMachine.sees := context
                checkMachine.evnts.add(event)   //adding event
                result.isDefinedBy.add(checkMachine) //adding machine
                machine:=checkMachine
            else
                if(machine.name!=checkMachine.name) then
                    machine:=checkMachine
                    machine.evnts.add(event) //adding event
                else
                    machine.evnts.add(event) //adding event
            end end
    end end end
    )
```

Figure 10. Adding machines and events

chines as shown in Fig. 10. This part of the transformation is responsible for creating new machines, adding new events to these machines and assigning new contexts to these machines. The events are further elaborated by adding new

```
nd.meaning.cntsT.each(aa |
    var prmts : EBVariable init EBVariable.new
    if(aa.isOfType.getMetaClass==Entity) then
        prmts.name:=aa.name
        prmt.cntsT.add(prmts)
    end
)
```

Figure 11. Adding parameters for events

parameters to them as shown depicted in Fig. 11. These parameters are taken from norm definitions, which contain necessary variables as input arguments.

One norm definition can contain several instructions to execute. Fig. 12 demonstrates how parallel executions are generated from such norms.

```
result.isDefinedBy.each(mm |
    mm.evnts.each(ee |
    nd.meaning.compound.each(cc |
    if(cc.getMetaClass==And) then
        var andOperator : And
        andOperator?=cc
        var parallel : ParallelExecution
            init ParallelExecution.new
        var ebterm : EBTerm init EBTerm.new
        if(ee.name==andOperator.cntsRPredicate.
                isOfType.isOfType1.name
                and ee.name==andOperator.cntsLPredicate.
                isOfType.isOfType1.name) then
            ee.prmt.add(prmt) //adding parameter to event
            parallel.left:=PredicatesToInstructions
                (andOperator.cntsLPredicate)
            parallel.right:=PredicatesToInstructions
                (andOperator.cntsRPredicate)
            ebterm.containsInstruction:=parallel
            instruction.isBuiltOf:=ebterm
            block.insts.add(instruction)
            ee.blks.add(block)
        end
    else if(cc.getMetaClass==Predicate) then
```

Figure 12. Handling parallel executions

Given a PN associated with a normative ontology, it is possible to further extend our transformation so that the PN and the Event B machine are semantically consistent.

It is possible for norms to specify invariant properties over a system: these take the form of norms whose trigger is always true: the invariant is then specified as a postcondition in first-order logic whose Deontic modality is *Ob*. These invariants are then mapped to invariants of the B machine associated with the responsible agent. Initial conditions are handled in a similar fashion.

Example 4.2 *The example order processing normative ontology can be transformed to an Event-B model, part of which is shown in Fig. 13. Each normative act within the ontology is mapped to Event-B events.*

Fig. 14 shows the context for the model defined in Fig. 13.

Variable order (a subset of ORDER) is a set of order data currently in the system. Variable product (a subset of PRODUCT) is a set of products currently involved in the ordering process. Variable status is a total function over the variable order and represents the status of the order. Variable quantity describes the quantity of ordered products within a given order and is a function over the product variable.

One of the generated events which is shown in Fig. 13 is receive. This event creates a new instance of the order and assigns the status of that order to received. The receive event contains "any-where-then-end" substitution construct. See the whole definition of "eOrder" and "Product" and their operations in [15].

One of the requirements of technical specification development is to prove that this specification does not violate its initial requirements and is internally consistent, and that eventually the final software system which is generated in the course of successive refinement steps is indeed correct.

```
MACHINE $EOS\_0$
SEES $EOS\_C0$
VARIABLES $order\ status\ product\ quantity$
INVARIANT
    inv1: $order \subseteq ORDER$ inv2: $product \subseteq PRODUCT$
    inv3: $status \in order \rightarrow STATUS$
    inv4: $quantity \in product \rightarrow \mathbb{N}_1$
EVENTS
Initialisation
    begin
        act1: $order := \varnothing$ act2: $status := \varnothing$
        act3: $product := \varnothing$ act4: $quantity := \varnothing$
    end
Event receive $\widehat{=}$
    any $oo$
    where
        grd1: $oo \in ORDER \setminus order$
    then
        act1: $order := order \cup \{oo\}$
        act2: $status(oo) := received$
    end
END
```

Figure 13. Machine for the model defined in Event-B

On the one hand, we are checking the correctness of the generated model. Since, we are performing transformation from normative ontologies to the Event-B model and the Petri net model, it is of high importance to check whether the both target models do not contain errors and are internally consistent. If this is the case, then we can state that the generated Event-B model is compatible with the generated Petri net model and vice versa. On the other hand by demonstrating the compatibility between two generated models we can also assert that our transformation is indeed correct. As [10] states, to be useful at all, an transformation must have specific characteristics. The most important characteristic is that a transformation should preserve meaning between the source and the target model. In our case, it is done by checking whether the target model functions in a way, as it was "prescribed" by the source model.

```
CONTEXT $EOS\_C0$
SETS
    $ORDER\ STATUS\ PRODUCT$
CONSTANTS
    $received\ pending\ invoiced\ dispatched\ rejected$
AXIOMS
    axm1: $STATUS = \{received, pending, ...\}$
    axm2: $received \neq pending$
    axm3: $received \neq invoiced$ ...
    axm11: $dispatched \neq rejected$
END
```

Figure 14. Context for the model defined in Event-B

Table 1. Proof obligations

Machine	Proof Obligations	Automatic	Interactive
eOrder	22	22	0
Total	22	22	0

We are particularly interested in checking whether events and initialization preserve the guards and invariants of the generated Event-B model. The Event-B language (similarly to B-method) defines proof obligations for substitutions (events and initializations). Discharging these proof obligations presents a form of specification validation.

There are several theorem provers used for specification validation. We used the Rodin platform, an Eclipse-based IDE [1], to generate the proof obligations. As it was mentioned earlier, the proof obligations generated were only for initializations and events, since only these elements of the abstract machine modify state variables.

Example 4.3 *Continuing our example, we can validate the generated specification. Validation of the specification requires that initialization T is guaranteed to establish the invariant I. It is also necessary to prove that all events preserve the invariant. In other words, if invariant I and precondition P are both true when the event is executed, then the event should be guaranteed to re-establish I:* $I \wedge P \Rightarrow [S]I$.

Table 1 illustrates a summary of statistics for proof obligations. According to these numbers all 22 generated proof obligations for the Event-B model were proven automatically.

5 Conclusion and Related Work

In this paper, we have shown how normative ontologies can be used for generating data-aware and semantically-rich business process models in the form of B specifications. We have shown the results MDA transformation from normative ontologies to both Petri nets model and and Event-B machines. Petri nets specify possible flow of the information but do not specify the nature of the information. In order to avoid this, we are generating additional Event-B machines from normative ontologies to provide missing semantics for the business processes.

There are a number of related approaches for enriching workflow models [3, 4]. One of them is showing transformation from BPEL4WS to full OWL-S ontology to provide missing semantics in BPEL4WS. BPEL4WS does not present meaning of a business process so that business process can be automated in a computer understandable way [2]. They are using an overlap which exists in the conceptual models of BPEL4WS and OWL-S and perform mapping from BPEL4WS to OWL-S to avoid this lack of semantics.

Another work is using the example of BPEL processes which should be converted to semantically enriched specifications. All data (stored in process models) must be augmented by references to ontologies [9]. They refer this augmentation to as *ontological lifting* because input business processes must be expressed using richer constructs provided by ontologies. However, from the perspective of web services, systems support only part of the process space representation which is reduced to the patterns of message exchange (choreography) and the control and data flow in the combination of multiple Web services (orchestration) [8].

[4] advocates the idea of ensuring the correctness of a workflow by making protocol specifications data-aware through expressing actual data content rather than message names. In other words, workflow validation cannot be complete unless this abstraction is eliminated. They present CTL-FO+ tool, an extension over Computation Tree Logic that includes first-order quantification on state variables in addition to temporal operators, and which is adequate for expressing data-aware constraints.

There is also a variety of research directed towards semantic enriching of Petri net business processes. The authors were proposing to enrich the semantics of Petri nets by combining it with OWL language. Representing Petri nets in combination with OWL is a way to make data computer-interpretable for flexibility, ease of integration and significant level of automation of loosely coupled business processes [11]. In this work, authors are trying to define Petri net models using OWL framework which entails horizontal way of integration. In other words, they are defining semantic metadata for business processes described by Petri nets. [3] was proposing an approach for (semi-)automatic detection of synonyms and homonyms of the process element names in order to support semantic process model interconnectivity and interoperability with use of OWL.

There were also several results on integrating Petri nets and Z [6, 7]. In [6] describes a thorough integration definition of Petri nets and Z which results in so called PZ nets for specifying concurrent and distributed systems. In this work, Petri nets are used to define the overall structure, control flow and dynamic properties and Z is applied for specifying tokens, labels and constraints of the system. This result is based on the previous more preliminary work on integrating Petri nets and Z outlined in [5]. Another work in [17] have used Z to specify certain aspects of restricted hierarchical coloured Petri nets. Namely, the authors have used Z schemas to define the metamodel of a hierarchical coloured Petri net and operation for specifying the transitions in a specific coloured Petri net.

[13] describes the model-driven transformation between the Semantic Web Rule Language with Web Ontology Language (OWL/SWRL) and Object Constraint Language with UML (UML/OCL). The implementation of the transformation is performed by using ATLAS Transformation Language involving several MOF based metamodels, XML schemas, and EBNF grammars. Whereas, [19] presents a new interchange format for rules for integrating the Rule Markup Language, the Semantic Web Rule Language and the Object Constraint Language. Since these languages are capable of providing a rich syntax for expressing rules, it is possible to make conceptual distinctions of different types of terms and different types of atoms. They also adopt the Model-Driven Approach, particularly specifying the computation-independent level as a domain containing set of rules and business policies.

The advantage of our work over the before-mentioned approaches is that we are incorporating norms into our source models and for this purpose using the MEASUR language to provide semantical information and additional constraints to the business processes of the source model (the CIM). Another advantage of our approach is that from the CIM model we are generating two PIM models defined in Petri nets and in Event-B. The Petri net model can be checked for liveness errors. With respect to the Event-B model we can use theorem proving techniques to check and refine data representation to obtain an executable system.

In multiagent systems, there are several quite successful works on developing and using norms in order to specify the expected behaviour of agents in a certain organization. For instance, one of the most interesting works in this area is [21, 20] which describes the agent architecture SMART which is based in an agent specification framework developed in the Z modeling language. Interesting aspect of this work is that it provides an analysis of different kinds of norms and agent societies based on these norms. Moreover, they are modelling norms as objects rather than as static constraints. As a result, these norms can have several states which in its turn completes the Norm lifecycle.

[18] makes norms operational rather than purely declarative by focusing on how norms should be operationally implemented in MAS from an institutional perspective. [14] view an electronic institution based on agents as dialogical system where all the necessary interactions between agents are made through dialogic activities (message exchanges). These interactions, also called illocutions, follow a certain well-defined protocol and are structured through agent group meetings, scenes. Such a division of all possible interactions among agents in scenes is in line with modular approach of systems design with the classical modular design principles and methodologies (e.g. Modular Pro-

gramming and Object-Oriented Programming) taken as a foundation.

Future work will investigate how our B-based PIMs can be further transformed into an actual platform specific solution utilizing industrial BPM solutions. We hope that our rich specifications involving data and operations will map naturally onto the modular technologies employed in, for example, Windows Workflow Foundation.

References

[1] J.-R. Abrial, M. Butler, S. Hallerstede, and L. Voisin. An open extensible tool environment for event-b. *Proceedings of the Eighth International Conference on Formal Engineering Methods, ICFEM*, pages 588–605, 2006.

[2] M. A. Aslam, S. Auer, and M. Böttcher. From bpel4ws process model to full owl-s ontology. *ESWC2006 Proceedings, Lecture Notes in Computer Science*, 2006.

[3] M. Ehrig, A. Koschmider, and A. Oberweis. Measuring similarity between semantic business process models. *Proceedings of the fourth Asia-Pacific Conference on Conceptual Modelling, Ballarat, Australia*, 67:71–80, 2007.

[4] S. Hallé, R. Villermaire, O. Cherkaoui, and B. Ghandour. Model-checking data-aware temporal workflow properties with ctl-fo+. *forthcoming*, 2007.

[5] X. He. Pz nets - a formal method integrating petri nets with z. *Proceedings of the 7th International Conference of Software Engineering and Knowledge Engineering SEKE'95*, pages 73–180, 1995.

[6] X. He. Pz nets - a formal method integrating petri nets with z. *Information and Software Technology*, 43(1):1–18, 2001.

[7] X. He and C. Yang. Structured analysis using hierarchical predicate transition nets. *Proceedings of the 16th International Computer Software and Applications Conference, Chicago*, pages 212–217, 1992.

[8] M. Hepp, F. Leymann, J. Domingue, A. Wahler, and D. Fensel. Semantic business process management: A vision towards using semantic web services for business process management. *Proceedings of the IEEE ICEBE, Beijing, China*, pages 535–540, October 2005.

[9] M. Hepp and D. Roman. An ontology framework for semantic business process management. *8th international conference Wirtschaftsinformatik, Karlsruhe*, 2007.

[10] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained. The Model Driven Architecture: Practice and Promise*. Pearson Education, Boston, USA, 2003.

[11] A. Koschmider and A. Oberweis. Ontology based business process description. *Proceedings of the CAiSE-05 Workshops, Lecture Notes in Computer Science, Springer, Porto, Portugal*, (13):321–333, 2005.

[12] K. Liu. *Semiotics in Information Systems Engineering*. Cambridge University Press, 2000.

[13] M. Milanović and et al. On interchanging between owl/swrl and uml/ocl. *Proceedings of 6th Workshop on OCL for (Meta-)Models in Multiple Application Domains (OCLApps) at the 9th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, pages 81–95, 2006.

[14] P. Noriega. Agent-mediated auctions: The fishmarket metaphor. *Number 8 in IIIA Monograph Series, Institut d'Investigació en Intelligència Artificial (IIIA), PhD Thesis*, 1997.

[15] PALab. The predictable assemble laboratory. http://palab.dcs.kcl.ac.uk/, November 2007.

[16] W. van der Aalst and K. van Hee. *Workflow Management: Models, Methods, and Systems*. The MIT Press, Cambridge (USA), London (England), 2002.

[17] K. van Hee, L. Somers, and M. Voorhoeve. Z and high-level petri nets. *Lecture Notes in Computer Science*, 551:204–219, 1991.

[18] J. Vázquez-Salceda, H. Aldewereld, and F. Dignum. Implementing norms in multiagent systems. *Multiagent System Technologies: Second German Conference, MATES 2004, Erfurt, Germany*, pages 313–327, September 2004.

[19] G. Wagner, A. Giurca, and S. Lukichev. A usable interchange format for rich syntax rules integrating ocl, ruleml and swrl. *Proceedings of Reasoning on the Web*, 2006.

[20] L. y López and M. Luck. Towards a model of the dynamics of normative multiagent systems. *Proceedings of the International Workshop on Regulated Agent-based Social Systems: Theories and Applications (RASTA '02)*, pages 175–194, July 2002.

[21] L. y López, M. Luck, and d'Inverno. A framework for norm-based interagent dependence. *Proceedings of the Third Mexican International Conference on Computer Science*, pages 31–40, 2001.

Technology-Independent Modeling of Service Interaction

Gerald Weber
Department of Computer Science
The University of Auckland
38 Princes Street, Auckland 1020, New Zealand
Email: gerald@cs.auckland.ac.nz

Abstract

Systems based on a service-oriented architecture (SOA) can be implemented with many different technologies, and in particular, they can be implemented with a heterogeneous set of technologies. An enterprise service bus (ESB) is a typical option for bridging the technology boundaries. It is desirable to have technology-independent models of the core services in the IT system. We present here computation-independent models (CIMs) and platform-independent models (PIMs) for service oriented architectures. Our models have the following advantages: Some of the CIMs are closely related to Petri net approaches; the PIMs are expressed in the same formalism as the CIMs; a canonical PIM is easily derived from a CIM; the semantics of the PIMs matches the operation of a typical enterprise service bus architecture. Finally, both CIM and PIM are defined as core semantic data models and can therefore be created with most semantic data modeling tools.

1 Introduction

In this paper we discuss technology-independent models for message-based communication of information systems. One key idea of a model-driven approach is to use platform-independent models that can be translated into several platform-dependent models, thus enabling reuse. For service-oriented architectures, many frequently discussed languages are not platform-independent; BPEL, for instance, is tailored towards web-services. In the same vein, it is important to realize that a (mis-)understanding of service-orientation as mere migration to web-services is an implementation technology and not an architecture. Today's enterprise computing projects are covering areas as diverse as healthcare [7] and e-commerce [15]. In such projects, a plethora of different message-based technologies is used; compare for example e-commerce with classical EDI [10] and AS2 [12], a novel e-commerce standard that is similar to web-services [4, 14]. In order to understand such diverse systems is helpful to assume a technology-independent viewpoint. The reference model for ODP defines a viewpoint as "a form of abstraction achieved using a selected set of architectural concepts and structuring rules, in order to focus on particular concerns within a system." [1]. We will propose our own viewpoints, which are mostly related to the enterprise viewpoint of ODP; our viewpoints will then motivate the different technology-independent models that we introduce. We introduce computation-independent model (CIMs) and platform-independent models (PIMs). More important than the individual labeling of these models as either CIMs or PIMs is however that they are in a clear semantic relationship to each other.

In Section 2 we discuss one of our key modeling principles, namely the heavy use of immutable datatypes, and derive CIMs that are not yet tied to a datamodel. In Section 3 we introduce the key aspects of our definition of core data models. In Section 4 we explain another key modeling principle, namely the use of datamodels to represent business integrity constraints that have a process-like character. We introduce CIMs based on this approach. In Section 5 we introduce CIMs that express synchronization. In Section 6 we reflect on the importance of message-based communication for enterprise computing. The message-based viewpoint presented in Section 7 captures the high-level architecture of state-of-the-art systems, particularly enterprise service bus architectures. Here we introduce our platform-independent models. In Section 8 we revisit the advantages of using core data models. In Section 9 we summarize our findings.

2 The memo-based viewpoint

Our first viewpoint is the *memo-based viewpoint*. In this system view, all IT-related business activities proceed by creating documents and locking them at a certain time. After that they are immutable and remain in the sys-

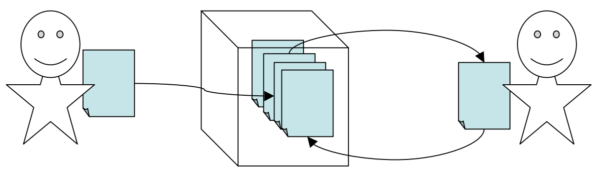


Figure 1. Memos are kept in a repository.

tem state. We call these immutable, archived documents *memos*. Hence the memo-based viewpoint models archiving on an analysis level as the repository of memos, as illustrated in Figure 1. User-generated as well as relevant system-generated memos should be kept. The first type of computation-independent model are therefore data models that contain only memos, we call them *memo models* (MMs).

The memo-based viewpoint is an enabling viewpoint for later process-oriented and message-based models. Memos are a precursor to messages, but in memos the aspect of transporting information is not yet emphasized, while persistence is emphasized more.

One remarkable semantic aspect is that the immutability constraint for the memos is inseparably connected with a timestamp concept. Every immutable object has a unique timestamp where it becomes immutable. We want to call this process *locking* in order to indicate that there could possibly be a previous edit phase. The locking of an immutable data object is an atomic event, and this means that the immutability constraint asks naturally for a transactional concept. This again gives rise to the importance of submit/response style systems.

The semantic essence of the memo model is the collection of memos, that allows only inserts of new memos, and the association of a timestamp with each memo. The concept of memo models is however not bound to a particular data model for the description of the individual memos. Therefore we understand memo models as being compatible with for example textual memos as well as with memos that consist of structured data. The space of versions of a wiki would constitute a memo model. The memo-based viewpoint is significant because via the immutability constraint and the ensuing differentiation of distinct timestamped memos it structures the information space.

The way we introduced the concept of timestamps for every memo is an example of the way we understand analysis in the software engineering process: analysis can add rich attributes like timestamps even where they might not be supported by every modeled system (i.e. the system might keep no, or unreliable timestamps) in order to foster system understanding.

The memo model allows strictly no change of the memo content. Extensions of this concept that in turn allow amendments of submitted data, akin to a concept of minor edits, are thinkable but can often be naturally defined on top of the memo model concept.

3 Core data models

The memo-based viewpoint does not require a certain data model, however for the upcoming process-oriented viewpoints we will use a relational model to describe the interconnection of memos. Therefore we employ a data model of the relational family. In our view this family especially includes conceptual models such as ER and some parts of UML. We, however prefer a semantically more minimalist approach and therefore only use core data models. A core data model for our purposes is a data model that has entity types and binary relation types, and the relation types have set-relational semantics. Generalization as a separate notion is not required. In form-oriented analysis [6] we call our modeling language the *parsimonious data modeling language*. Its models are called PD models for short, they are core data models to begin with. The PD models can be seen as simplifications of UML and ER models. A PD model in our approach is described by a mathematical object, its so-called model graph. Because we conceive PD model graphs as mathematical objects, the visual representation is not prescribed. We prefer a visual notation which is similar to graph notation. A PD model consists of *entity types*, *relation types*, and *roles*. An entity type has entity instances. A relation type is a predicate which says whether a number of entity instances are thought of as being connected. Each role *role(a, t)* is connected with one relation type a and one entity type t. The *model graph* is defined to be a tuple (E, P, R, e, p), where E is the set of entity types, P is the set of relation types, and R is the set of roles. $e : R \mapsto E$, $p : R \mapsto P$ are the functions giving for each role its entity type and its relation type. The *static semantics* of a PD model are the set of all the possible *states* of this model. An entity type represents the collection of all *entity instances*. Entity instances are opaque identities. Each entity type is assumed to be a *repository*, i.e., a countably infinite set of entity instances. We call an entity instance that has never been used before a *fresh entity instance*. In PD models the primitive data types are just entity types, and there is no difference between attributes of a type and relation types. In each state a relation type is represented by a finite relation between the connected entity types.

The *dynamic semantics* of a PD model describe the possible updates on the state of the PD model. Updates are the transition from one state to another. This definition gives rise to a typed automaton model of the system. The system changes its state through updates, while the PD model remains the same. Here we distinguish two important notions of updates: primitive updates and transactions. The *primitive updates* are simply the insertion or the deletion of a single link between two entity instances. The operation *insert of link l on relation type a, insert(a, l)* means that a becomes $a \cup \{l\}$. The *delete of link l on relation type a, delete(a,l)* means that a becomes $a \setminus \{l\}$. The insert operations can make use of a new() operator that delivers fresh identifiers. The *transactions* are complex updates which are still executed atomically. They will be clarified through an automaton model later.

4 Using core data models for modeling business logic

The key semantic approach in this paper is that we will model process-like aspects of the business logic with core data models. Memos in a business process are often based on earlier memos. As an example let us consider two types of memos in a banking application, a memo type representing the opening of a new account and a memo type representing the granting of a personal loan. Both memos are related to an important business process in a bank, the granting of personal loans. A business constraint is, for instance, the constraint that a loan must only be granted after an account has been opened. This is a process-like constraint in the business logic.

In order to model such process-like constraints we use relation types with the following additional constraints. We consider *arrow heads* as annotations on roles of relations. The entity type at the annotated role is the target and the other entity type is the source. A *time* arrow head is defined on relation types between timestamped entity types. The defining condition of a time arrow head is that the timestamp for each target instance t is not earlier than the timestamp of any source instance that is linked to t via the annotated relation type. We call these source instances the predecessors of t and t the successor of them. In other words, time arrow heads always point in positive time direction. A relation type with a time arrow head is a *time relation type*. We will consider here only time relation types between memos. A set of time relation types has the property that in every state the directed links of these relation types form a directed acyclic graph. We call such constraint *partial oder* constrains. Im UML, aggregations can be used to some extent, but the time constraints are not implied by aggregations.

In the rest of the paper we will model process aspects with time relation types between memo types. The personal loan example is modeled by a time relation types between the two memo types so that the loan memo is the successor. A 1..1 multiplicity on that time relation type at the role of the account creation memo type makes sure that every loan grant has to refer to one earlier account creation. This example illustrates that we can model process aspects with time

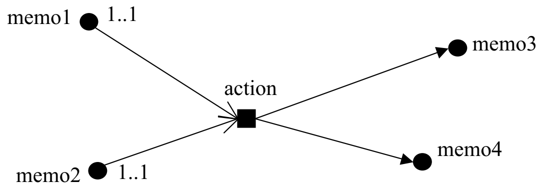


Figure 2. A Memo Flow Model

relation types in core data models alone, without the need for a separate process language. The time relation type and the multiplicity alone ensures that the intended process must be followed. If $t_1, t_2, \ldots, t_n$ are the generation times of the different memos in the process, then our approach enforces $t_1 \leq t_2 \leq \ldots \leq t_n$. The immutability of memos is applied here as well.

This use of time relation types gives rise to a computation-independent model for process-like aspects, and we call this the *memo order model* (MOM). This model only contains memos and time relation types between these memos. In the following we will see that we can create more specialised models for processes based on this. These models will be specializations of MOMs. The previous definition of the more general MMs is helpful, because there are specialization of MMs other than MOMs. A strongly typed reimagination of email for example would only support single precursor memos, either of the Re: style or of the Fwd: style. Indeed the DTIMs introduced later, which are platform-independent models, use such an approach. Wikis on the other hand, where memos are page versions, not pages, also support only one predecessor memo, with the additional constraint that the predecessor relation establishes a total order. The concept of a changeable page is a special case of a topic bundle [6], this is an identifier named in a memo as a topic. For wikis we usually have the additional constraint that each page version names only one such identifier as topic and all page versions that name this topic are part of a single total order of memos. In Figure 1 the right hand actor performs a wiki style edit. It has the benefit that it looks like a change to the old page, hence appealing to intuition, but it is in fact the creation of a new immutable page, without destruction of the old version.

5 The action viewpoint

An often discussed phenomenon is synchronization between subprocesses. Not every new memo must lead immediately to a subsequent action. Assume, we need a quote and an approval in order to confirm a travel booking. Only if both messages have arrived, a confirmation can be cre-

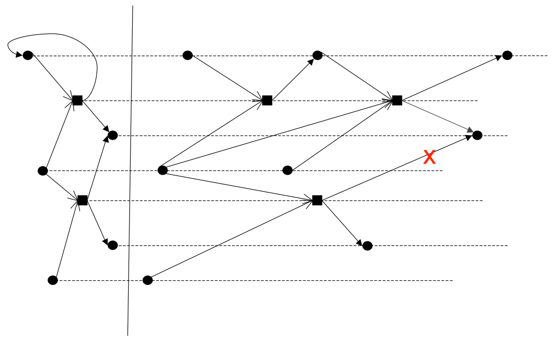


Figure 3. Data model and example state of a memo flow model.

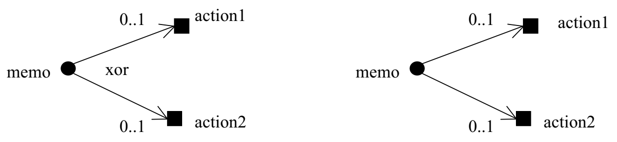


Figure 4. Nondeterminism in a Memo Flow Model

ated. We therefore want to define a viewpoint that expresses this abstraction, the *action viewpoint*. Actions that are performed get their own identity, and they can result in several memos being produced. An action type is an entity type and it has relation types to all memos that are required and all memos that are produced. The action viewpoint results in a bipartite action model resembling a Petri net as shown in Figure 2. The two partitions are actions and memos. The actions are depicted as square nodes, the memos are depicted as round nodes. The actions themselves do not hold additional data, but they help in defining what we call a superparameter [6], that is a data type that acts as a parameter list. This is in contrast to many languages where a parameter list might be a type of the type system, but is not a first class citizen data type. All the memos linked to an action are input to that action. In this model, it is in general not necessary that the action is performed immediately as soon as all necessary predecessor memos are present. Such a constraint would however be frequent for individual action types. The action viewpoint gives rise to a computation-independent model (CIM). We call a data model that shows memos and actions a memo flow model (MFM). A memo flow model is a directed bipartite graph; this is a well-formedness condition on memo-flow models.

Multiplicities that refer back from an action to predecessor memos describe which predecessors must be there at the start of the action. If for example the multiplicities are all 1..1 multiplicities as in Figure 2, then they represent a synchronization, and the action behaves similar to a synchronization bar in a Petri net.

5.1 Semantics of memo flow models

In Figure 3 a memo flow model is shown on the left and an example state over this model is shown, by indicating with (here horizontal) swimlanes, to which type each entity instance is belonging. This type of diagram was introduced in form-oriented analysis to discuss semantic properties of models (The diagram works best if the relation types are unique between two entity types). A first semantic property that we require from relation types in a MFM is that they are time relation types. This means, that the state over the data model is a directed acyclic graph. A memo flow model, as a directed bipartite graph, has two types of edges: action-memo edges and memo-action edges. The second semantic property only affects the action-memo transitions; these are the arrows going from actions to memos. The requirement is that every memo is only created by one action. This is equivalent to UML composition (in UML composition the diamond would be on the opposite end of the relation type from the arrowhead given in the MFM). This property is illustrated in Figure 3. Note the edge that is crossed out. The targeted memo is also produced via an edge from a different action. One edge must be deleted. In contrast, one action can be targeted by an arbitary number of memo-action transitions, if no other constraints prevent that.

5.2 Nondeterminism and the action viewpoint

The action viewpoint is reminiscent of advanced Petri nets since it is bipartite. In contrast to Petri nets however, there is no implicit consumption of messages by their successor messages. This becomes obvious if we consider nondeterminism as in the following examples. In the two small MFMs in Figure 4, in the left picture each message is giving rise to either action1 or action2 but not both. In this sense the message is consumed, but only because of the xor constraint (note here that the xor constraint and the multiplicities are stated following the PD model convention, not the UML convention). On the right-hand side, however, the message may give rise to two actions because of the multiplicities. The reason we prefer MFMs over sophisticated Petri net variants is the tight integration of MFMs with core data models.

6 Importance of message interchange

Message-based data interchange is used in many mature technologies. EDI is an implementation technique for

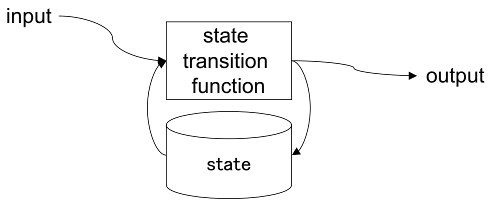


Figure 5. Unit systems are modeled as automata

business message interchange [8]. It allows communication with high Quality of Service. The interchange is traditionally on dedicated networks, called value-added networks. The more recent technology of web services [2] is an implementation technique for message communication. Web services use XML as a semi-structured format [3] for messages. Notations for distributed systems based on Web services include Web service orchestration languages, for example BPEL. Web services use a type system given through the XML Schema concept. Deployed Web services are described by the Web Service Description Language WSDL. This language can be used for the specification of configuration information, i.e., for the data transmission options chosen. This approach is technology-dependent: BPEL primarily describes Web service communications. A higher degree of abstraction is needed for modeling at the design or analysis stage.

7 The message-based system viewpoint

The *message-based* viewpoint allows us to capture the architecture of state-of-the-art message-based systems. In this viewpoint all memos are considered to be messages. Each model in this viewpoint is called a *data type interchange model* (DTIM). DTIMs are high-level design models for message-based communication. They fit well for example to enterprise service bus architectures [11], but also to similar systems used in electronic data interchange [5].

The core architectural feature of such systems is that the business logic is composed of components that are triggered by messages. This common high-level design might be implemented with various concrete component technologies, such as message-driven enterprise java beans (EJBs) or XML style sheets.

7.1 Unit systems as automata

In form-oriented analysis, distributed systems are conceived as a net of single systems, called *unit systems*. Such a unit system is a *computational automaton* with a state as illustrated in Figure 5. This automaton takes in messages

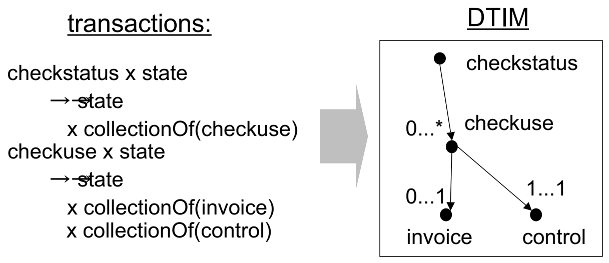


Figure 6. DTIMs fit to the typed automaton model.

and produces other messages. In an untyped view, such an automaton is specified by a single *state transition function*:

```
stateTransitionFunction:  message × state
                 → state × collectionOf(message)
```

In the statically-typed view, each transaction has an associated message type. The state transition function is conditional and invokes for each message type a different *transaction*, which is the state transition for this message type. In Figure 6 on the left-hand side, two transactions for the messages *checkstatus* and *checkuse* are shown.

We can therefore represent the transactions by their message types. A single message type acts as the superparameter of the transaction. Each such transaction may represent one deployed transformation component from Figure 7, such as workerA or workerB. Message-based components working on persistent message queues are an old type of component, known from classical transaction monitors. A recent name for a directly analogous technology of message-based middleware today is enterprise service bus (ESB). The automaton model fits very good to the general ESB architecture; the transactions are equivalent to worker threads in such a system as illustrated in Figure 7. They are crucially important for a workable enterprise platform, and it was a major drawback for recent object-oriented enterprise platforms that they did not have message-driven components from the start; their later addition was eagerly awaited by practitioners. Such message-based components are an exact replica on the implementation level of our platform-independent concept of transactions. Indeed such message-based components follow the superparameter concept, in that the message is their sole parameter and it is of course a structured data object. As an additional side remark one might add that these concepts bear a resemblance to the coordination language Linda [9], but in practice the central requirements for users are nonfunctional requirements, chiefly persistence of the messages. There is also

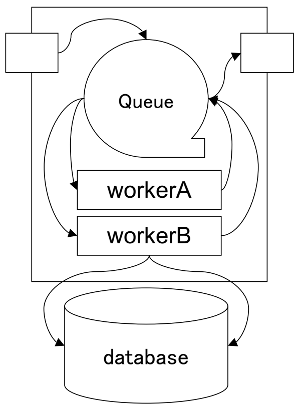


Figure 7. Transactions are analogous to components in an ESB architecture.

a direct correspondence with active database technologies: the transaction for message m is in principle a trigger on insert on table m, and it follows the event-condition-action pattern. The transactional execution style used here is called *detached* in active database terminology.

7.2 Data type interchange models

The message-based viewpoint motivates the platform-independent Data Type Interchange Models (DTIMs) that we are going to introduce now. They are first of all a natural translation of the above mathematical definition into a data model. The DTIM in its elementary form contains as entity types (depicted as nodes) just message types that represent the transactions as explained before.

If the transaction for message type A may send messages of type B, then the DTIM contains a time relation type from A to B. The connection between the mathematical notation for a single transaction and the node in the DTIM is shown in Figure 6. Again, since the edges are just relation types, DTIMs can immediately be annotated with constraints such as multiplicities at the targets of relation types. A $1...1$ multiplicity at B, for example, indicates that a message of type A always causes a message of type B. These multiplicities are written with three dots, since they have asynchronous semantics as will be explained in Section 7.3.

A message type can have many ingoing edges as well, coming from all those transactions that can produce such a message. At the source of each edge, a composition diamond is implied by the above definition of transactions, since every message was produced by exactly one transaction. These diamonds are shown in Figure 8 to illustrate this fact, but they will usually be omitted and assumed implicitly in DTIMs.

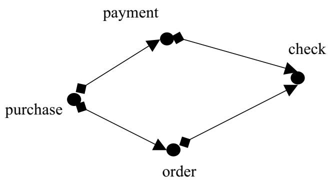


Figure 8. The transitions in DTIMs have implicit composition diamonds at their source.

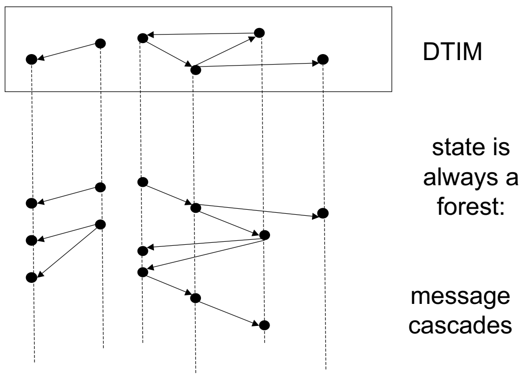


Figure 9. The state of a DTIM is a forest.

This means that the object net over the DTIM is not only cycle-free but in fact it is a forest, as shown in Figure 9, where the DTIM is shown on the top and the state is again shown with swimlanes. A message is processed in a transaction, then the transaction may trigger new messages, and so a *message cascade* is started which is supposed to terminate. A message cascade is always a tree.

7.3 Context of multiplicities

Constraints on DTIMs such as multiplicities can be tied to different checkpoints. For multiplicities this is necessary to accommodate the fact that not all the messages produced during the operation of a system are processed at once but in a consecutive way. To state this in intuitive terms: asynchronous messages call for asynchronous multiplicities. We can formalize this the following way.

A multiplicity tied to the transaction boundaries is valid after each transaction. Such transactional multiplicities are depicted with two dots between the upper and lower bounds. But for the multiplicities given at the targets of DTIM-edges we have the following situation. Mostly, no message at the target of the edge is produced in the same transaction in which the source was produced, therefore the transactional lower multiplicity is zero. However, in DTIMs the processing of messages in subsequent transactions is guaranteed,

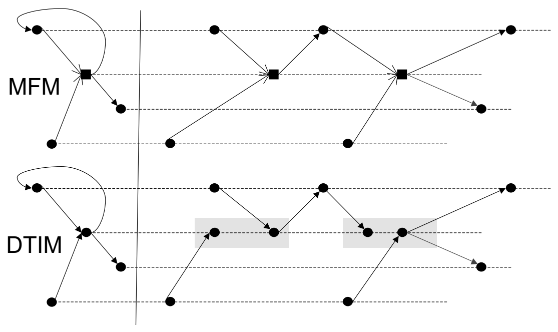


Figure 10. An MFM and a DTIM implementing the former.

and in many cases we know that one subsequent message must be produced later. Here we also want to use multiplicities – we call them *DTIM multiplicities* – to model this knowledge. Since these multiplicities have a different meaning, they are shown with three dots. This distinction is actually only necessary for the lower multiplicity. In DTIMS, these multiplicities typically refer to the end of message cascades. The lower DTIM multiplicities define conditions that must hold before the cascade can terminate, and the upper DTIM multiplicities define limits for the message cascade, although they in general do not suffice to ensure finiteness of the cascade.

7.4 Memo flow and data type interchange

A memo flow model can be implemented by an isomorphic DTIM. An example is given in Figure 10. The DTIM is obtained by removing the distinction between actions and memos. The semantics of the DTIM that is implementing a MFM are the following. If we consider one message type in the DTIM that is implementing an action (in the example there is a single action), then the transaction for this message type has to have the following specification: Only if several instances of this message type have been received, one from each memo required for this action, then the last of these instances sends out all the messages for that action. If we consider the example, then we see that the memo flow model abstracts from the mechanics of data interchange. In the state of the memo flow, the two instances of the action type have an isomorphic star of ingoing edges. In the DTIM state, however, it is denoted that the messages have been received each time in a different order. Therefore the MFM offers a further abstraction that can be helpful but hides the actual operational history, how the memo flow was executed. This is why we call the MFM a computation-independent model.

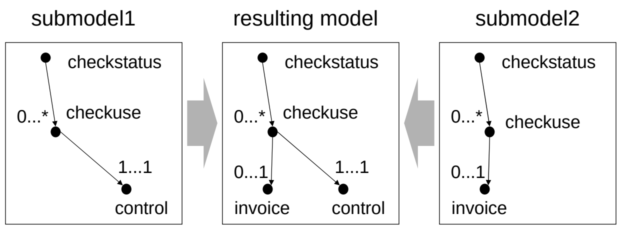


Figure 11. Model Decomposition

7.5 Model decomposition

For the technology-independent models presented here we use the model composition mechanism from form-oriented analysis. Models are conceived as a set of model elements and composition is the set union of these sets. As a consequence, we can compose models that contain partly the same and partly different elements as shown in Figure 11.

In this way we can use one DTIM L to model a subsystem in another DTIM R. We can form larger diagrams, by modeling subsystems and connecting them with a diagram on a higher level. A further discussion of this can be found in [13].

8 Advantages of using core data models

The fact that our models are still core data models has several advantages. This opens up the possibility to create the presented models with many data modeling tools. Core data models are naturally a minimal functionality supported by a majority of modeling tools. Therefore, all these modeling tools can be used to model this type of process model. For example, many tools for UML class diagrams or for ER diagrams can be used. Secondly, because our models are core data models, we can use multiplicities to express integrity constraints on process execution, as seen in our examples. Thirdly the integration with other parts of the data model is immediately possible as shown in Figure 12. The type *shipment* is an entity from the datamodel. The model expresses that for every shipment received by the organization running this system there has to be one or more notification messages sent.

In general, a data modeling tool might not enforce the semantic constraints of our models. For example, it might not enforce the defining constraints on time relation types, and specifically it might not enforce the immutability of memos. In some cases it might even be necessary to rely on standard role names in order to express the directions of time relation types. Also, the well-formedness condition on MFMs, namely that they must be bipartite, will often be not automatically enforceable. Nevertheless, the semantics

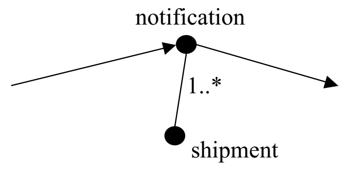


Figure 12. DTIMs can be combined with data models

presented here ensure that such a tool use is more than the trivial option of using a core data modeling tool to draw arbitrary labeled graphs.

9 Conclusion

We need a platform-independent way to describe systems that must fulfil stringent requirements. There is no single definition of the boundary between a computation-independent model and a platform-independent model, despite all strong opinions about these notions. This is aggravated by the fact that what appears to one person as a formal notation for a PIM may appear to someone else as just another data format and hence a proprietary platform. Therefore the main focus should be to ensure that CIMs are on a higher abstraction level as PIMs, and PIMs are not obviously tied to a particular technology. Message-based architectures can be set up with a plethora of technologies. The high-level design of such systems is, however, often very similar. In this paper we have presented computation-independent models and matching platform-independent models that fit well to current state-of-the-art architectures such as enterprise service bus technologies. We have presented semantics for these models by defining them as core data models. This gives us a parsimonious yet powerful modeling approach for process-like business constraints that is easy to integrate with the information model of typical enterprise applications.

References

[1] ITU-T Rec. X.902 — ISO/IEC 10746-2. Open distributed processing - reference model - part 2: Foundations, 1996.

[2] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services: Concepts, Architecture and Applications*. Springer Verlag, 2004.

[3] Peter Buneman. Semistructured data. In *PODS '97: Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 117–121. ACM Press, 1997.

[4] Christoph Bussler, Dieter Fensel, and Alexander Maedche. A conceptual architecture for semantic web enabled web services. *SIGMOD Rec.*, 31(4):24–29, 2002.

[5] Barry Dowdeswell and Christof Lutteroth. A message exchange architecture for modern e-commerce. In Dirk Draheim and Gerald Weber, editors, *TEAA*, volume 3888 of *Lecture Notes in Computer Science*, pages 56–70. Springer, 2005.

[6] Dirk Draheim and Gerald Weber. *Form-Oriented Analysis - A New Methodology to Model Form-Based Applications*. Springer, October 2004.

[7] Marco Eichelberg, Thomas Aden, and Jörg Riesmeier. A survey and analysis of electronic healthcare record standards. *ACM Computing Surveys*, 2005.

[8] Margaret A. Emmelhainz. *EDI: Total Management Guide*. John Wiley & Sons, 1992.

[9] David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[10] Paul Kimberley. *Electronic Data Interchange*. McGraw Hill, 1991.

[11] Min Luo, Benjamin Goldshlager, and Liang-Jie (LJ) Zhang. Designing and implementing enterprise service bus (esb) and soa solutions. In *SCC '05: Proceedings of the 2005 IEEE International Conference on Services Computing*, page .14, Washington, DC, USA, 2005. IEEE Computer Society.

[12] D. Moberg and R. Drummond. MIME-Based Secure Peer-to-Peer Business Data Interchange Using HTTP, Applicability Statement 2 (AS2). RFC 4130 (Proposed Standard), July 2005.

[13] Gerald Weber. A platform-independent approach for auditing information systems. In *HDKM '08: Proceedings of the second Australasian workshop on Health data and knowledge management*, pages 65–73, Darlinghurst, Australia, Australia, 2008. Australian Computer Society, Inc.

[14] Sanjiva Weerawarana, Francisco Curbera, Frank Leymann, Tony Storey, and Donald F. Ferguson. *Web Services Platform Architecture*. Prentice Hall PTR, 2005.

[15] Han Zhang, Gerald Weber, William Zhu, and Clark Thomborson. B2b e-commerce security modeling: A case study. In *Computational Intelligence and Security, 2006 International Conference on*, pages 1549–1554, 2006.