Enterprise Architecture Analysis with XML

F.S. de Boer[1,2] M.M. Bonsangue[2*†] J. Jacob[1] A. Stam[2,3] L.van der Torre[1,4]

[1]*CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands*
[2]*LIACS, Leiden University, The Netherlands*
[3]*Ordina SI&D Technology Consulting, Amersfoort, The Netherlands*
[4]*Delft University of Technology, The Netherlands*

Abstract

This paper shows how XML can be used for static and dynamic analysis of architectures. Our analysis is based on the distinction between symbolic and semantic models of architectures. The core of a symbolic model consists of its signature that specifies symbolically its structural elements and their relationships. A semantic model is defined as a formal interpretation of the symbolic model. This provides a formal approach to the design of architectural description languages and a general mathematical foundation for the use of formal methods in enterprise architectures. For dynamic analysis we define transformations of models of architectures, modeled in XML, and for this purpose the XML vocabulary for an architecture is extended with a few constructs defined in the Rule Markup Language (RML). There are RML tools available that perform the desired transformations.

1. Introduction

Architectures as defined in the IEEE 1471-2000 standard [8] typically consists of conceptual models visualized as diagrams. Architectural description languages and UML have been used for information architectures, and more recently similar languages are used for modeling enterprise architectures, as described, for example, in the Zachman's framework [16].

The research question of this paper is how to design tools for analysis of enterprise architectures. We distinguish between static and dynamic analysis, and we use XML-based technology. Our approach is based on the following logical concepts [2].

Signature for static analysis. The signature of an architecture focuses on the symbolic representation of the structural elements of an architecture and their relationships, abstracting from other architectural aspects like rationale, pragmatics and visualization. It emphasizes a separation of concerns which allows to master the complexity of the architecture. Notably, the signature of an architecture can easily be expressed in XML for storage and communication purposes, and can be integrated as an independent module with other tools including, e.g., graphics for visualization.

Semantic model for dynamic analysis. The formal *semantics* of a symbolic model of an architecture provides a formal basis for the development and application of tools for the *logical analysis* of the dynamics of an architecture. A signature of an architecture basically only specifies the basic concepts by means of which the architecture is described, but an interpretation contains much more detail. In general, there can be a large number of different interpretations for a signature. This reflects the intuition that there are many possible architectures that fit a specific architectural description.

By applying the techniques for static and dynamic analysis discussed in this paper, we get a better understanding of how enterprise architectures are to be interpreted and what we mean with the individual concepts and relationships. In other words, these techniques allow enterprise architects to validate the correctness of their architectures, to reduce the possibility of misinterpretations and even to enrich their architectural descriptions with relevant information in a smooth and controllable way.

The layout of this paper is as follows. In Section 2 we introduce a running example to explain our definitions. In Section 3 we discuss tool support, XML, AML and RML. In Section 4 and 5 we explain static and dynamic analysis using these tools.

[*] The research of Dr. Bonsangue has been made possible by a fellowship of the Royal Netherlands Academy of Arts and Sciences

[†] Corresponding author. Phone: `+31.71.527 7095` Fax: `+31.71.527 6985` Email: `marcello@liacs.nl`

2 ArchiMate: a running example

To illustrate static and dynamic analysis in enterprise architectures, we use an example from the ArchiMate project. ArchiMate is an enterprise architecture modelling language [11, 12]. It provides through a metamodel concepts for architectural design at a very general level, covering for example the business, the application, and the technology architecture of a system. The Archimate language resemble the business language Testbed [5] but it has also a UML-flavor, introducing concepts like interfaces, services, roles and collaborations.

The example modelled using the ArchiMate language concerns the enterprise architecture of a small company, called *ArchiSell*. In *ArchiSell*, employees sell products to customers. The products are delivered to ArchiSell by various suppliers. Employees of ArchiSell are responsible for ordering products and for selling them. Once products are delivered to ArchiSell, each product is assigned an owner, responsible for selling the product.

To describe this enterprise we use the ArchiMate concepts and their relationships as presented in Figure 1. In particular, we use structural concepts (product, role and object) and structural relationships (association), but also a behavioral concepts (process) and behavioral relationships (triggering). Behavioral and structural concepts are connected by means of the assignment and access relationships.

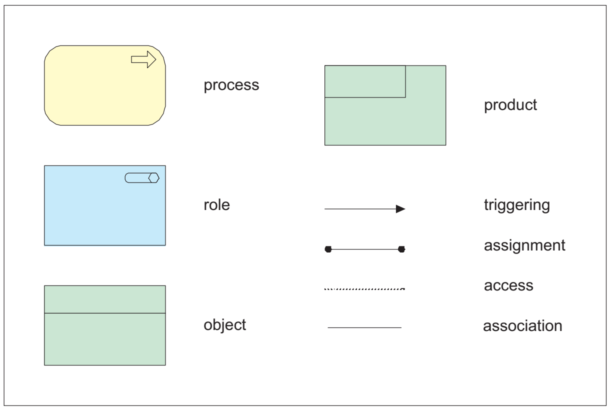

Figure 1. Some concepts and relations

A product is a physical entity that can be associated with roles. A role is the representation of a collection of responsibility that may be fulfilled by some entity capable of performing behavior. The assignment relation links processes with the roles that perform them. The triggering relation between process describes the temporal relations between them. When executed, a process may need to access data, whose representation is here called object.

We specifically look at the *business process architecture* for ordering products, depicted in Figure 2.

In order to fulfill the business process for ordering a product, the employee has to perform the following activities:

- Before placing an order, an employee must register the order within the Order Registry.
- After that, the employee places the order with the supplier.
- As soon as the supplier delivers the product(s), the employee first checks if there is an order that refers to this delivery. Then, he/she accepts the product(s).
- Next, the employee registers the acceptance of the product(s) within the Product Registry and determines which employee will be the owner of the product(s).

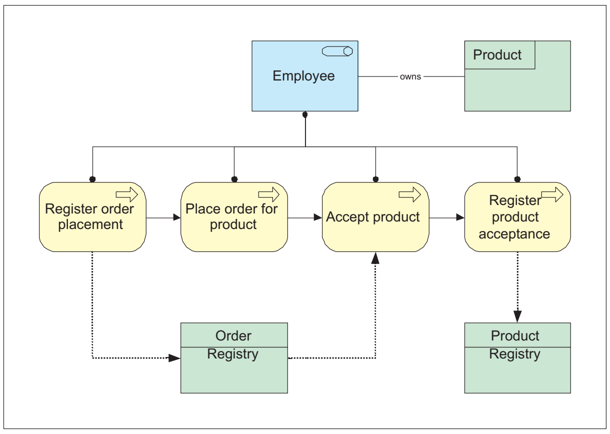

Figure 2. A Business Process Architecture

Despite the apparent simplicity of the diagram, there are several issues which can be analyzed. For example, when an architect presents this architecture, he may explain that the role of the order registry is to coordinate between the first two processes of placing orders and accepting them. Whereas the same employee should see that an order which is placed is also registered, there may be another employee which accepts the order.

Also variants can be analyzed. For example, given the fact that the coordination between order placement and order acceptance is regulated via the order registry, is it still necessary that placing the order for a product triggers the process that accepts the product. In other words, what is the impact if we change the architecture by removing this relation?

Before we can consider these questions, we need a language to represent the architecture. The ArchiMate language is a visual modelling language not well suited for representation or reasoning. We therefore represent architectures like the one above in XML.

3. The tools: XML, AML and RML

Before we start to analyze the enterprise architecture of the running example, we introduce our machinery. It consists of XML, AML and, most importantly, RML.

The Extensible Markup Language (XML) [15] is a universal format for documents containing structured information using nested begin and end labels, which can contain attributes. For example, a such as:

```
<product>
<weefer color="green">zyx</weefer>
<wafer color="blue">cis</wafer>
<weefer color="green">zyx</weefer>
</product>
```

The nested structure of the labels corresponds to a tree. They can be used over the Internet for web site content and several kinds of web services. It allows developers to easily describe and deliver rich, structured data from any application in a standard, consistent way. Today, XML can be considered as a lingua franca in computer industry, increasing interoperability and extensibility of several applications. Terseness and human-understandability of XML documents is of minimal importance, since XML documents are mostly created by applications for importing or exporting data.

The ASCII Markup Language (AML) [9] used to show examples in this paper is an alternative for XML syntax. AML is designed to be concise and elegant and easy to use. AML uses indentation to increase readability and to define the XML tree hierarchy: indentation level corresponds to depth, sometimes called level, in the tree. No indentation is required for the set of attributes that immediately follows each attribute name.

```
product
  weefer color="green"
    zyx
  wafer color="blue"
    cis
  weefer color="green"
    zyx
```

The Rule Markup Language (RML) is a tool for transforming XML documents that can be used for analysis of architectural description, and in particular for the definition and simulation of the system behavior. It consists of a set of XML constructs that can be added to an existing XML vocabulary in order to define RML rules for that XML vocabulary. These rules can then be executed by RML tools to transform the input XML according to the rule definition. Consider for example the following rule which removes duplicates from an XML document.

```
div class=rule name="Removeduplicates"
    div class=antecedent
        product
            rml-list name=rml-A
            rml-tree name=rml-B
            rml-list name=rml-C
            rml-tree name=rml-B
            rml-list name=rml-D
    div class=consequence
        product
            rml-use name=rml-A
            rml-use name=rml-B
            rml-use name=rml-C
            rml-use name=rml-D
```

The example illustrates the main constructs. First, there is an input and an output part of the rule, called antecedent and consequent. The antecedent contains a set of variables, rml-A, rml-B, rml-C and rml-D. The second variable occurs twice, and will therefore only match with a duplicate. Finally, rml-list matches with a list of elements, and rml-tree with one element; they can be considered the analogues of * and ? in regular expressions as used in for example grep.

The antecedent matches with the product given above, and binds the variables so that rml-A and rml-E are empty, rml-B is the weefer and rml-C is the woofer. The consequent of the rule explains the output of the rule. It reproduces the content of the variables rml-A, rml-B, rml-C and rml-D, but it does not reproduce the second instance of rml-B. In this way, the duplicate is removed.

```
product
  weefer color="green"
    zyx
  wafer color="blue"
    cis
```

There are a few more constructs, dealing for example with variables for attributes such as color. The set of RML constructs is concise and shown in Table 3. Things that can be stored in RML variables are element names, attribute names, attribute values, whole elements (including the children), and lists of elements.

The example illustrates that a pattern can be matched that is distributed over various parts of the input XML. Such pattern matching is hard to define with other existing approaches to XML transformation because they do not use of the problem domain XML for defining transformation rules: transformations are defined either in special purpose language like the Extensible Stylesheet Language Transformation (XSLT), or they are defined at a lower level by means of programming languages like DOM and SAX. RML captures transformations defined by a single rule, but interaction among rules is dealt with by other tools. Moreover, XML transformations normally involve creating links between elements by means of cross-referencing attributes, or reordering elements, or adding or removing elements, but does typically not include things like integer arithmetic and floating point calculations. In case of such transformations RML tools will have to be combined with other tools that can do the desired calculation.

	Elements that designate rules
`div`	`class="rule"`
`div`	`class="antecedent" context="yes"`
`div`	`class="consequence"`

element	attribute	A	C	meaning
				Elements that match elements or lists of elements
`rml-tree`	`name="X"`	*		Bind 1 element (and children) at this position to RML variable X.
`rml-list`	`name="X"`	*		Bind a sequence of elements (and their children) to X.
`rml-use`	`name="X"`		*	Output the contents of RML variable X at this position.
				Matching element names or attribute values
`rml-X`	...	*		Bind element name to RML variable X.
`rml-X`	...		*	Use variable X as element name.
...	`...="rml-X"`	*		Bind attribute value to X.
...	`...="rml-X"`		*	Use X as attribute value.
...	`rml-others="X"`	*		Bind *all* attributes that are not already bound to X.
...	`rml-others="X"`		*	Use X to output attributes.
...	`rml-type="or"`	*		If this element does not match, try the next one with rml-type="or".
				Elements that add constraints
`rml-if`	`child="X"`	*		Match if X is already bound to 1 element, and occurs *somewhere* in the current sequence of elements.
`rml-if`	`nochild="X"`	*		Match if X does not occur in the current sequence.
`rml-if`	`last="true"`	*		Match if the younger sibling of this element is the last in the current sequence.
A `*` in the `A` column means the construct can appear in a rule antecedent. A `*` in the `C` column is for the consequence.				

Figure 3. All the RML constructs

Combinations of RML with other components like programming language interpreters has been applied successfully in the EU project OMEGA (IST-2001-33522, URL: http://www-omega.imag.fr) that deals with the formal verification of UML models for software. That tool for the simulation of UML models does the XML transformations with RML, and uses an external interpreter for example for floating point calculations on attributes in the XML encoding.

In the remainder of this paper, we show how RML can be used for the analysis of the enterprise architecture in the running example. RML was designed to make the definition of executable XML transformations also possible for other stakeholders than programmers. This is of particular relevance when transformations capture for instance business rules. In this way it is possible to extend the original model in the problem domain XML vocabulary with semantics for that language. Similarly, it is also possible to define rules for constraining the models with RML.

Below we show an example of RML by presenting the rule that defines the state transformation of the action of our running example, where emp and order-reg are individual names for an employee and the Order_registry, respectively. The details of this notation are discussed later in this paper.

```
emp, order-reg := Register_order_placement(
        emp, order-reg)
```

Content-preserving RML constructs have been omitted for clarity.

```
div class=rule name="Register order placement"
    div class=antecedent
        variables
            rml-Employee order=rml-OrderName
                product=rml-ProductName
            order-registry
                rml-list name=oldOrders
    div class=consequence
        variables
            rml-Employee order=rml-OrderName
                product=rml-ProductName
            order-registry
                rml-use name=oldOrders
                order name=rml-OrderName
```

This example illustrates several RML constructs which do not appear in the removal of duplicates example. In particular, it uses variables for element names and attribute values. The effect of applying this rule is that order-registry is extended with an order.

4. Static analysis

We designed our own XML vocabulary, because we could not find an adequate standard one. We base this design on a formal basis discussed in Sect. 4.1. Diagrams like the one in Fig. 2 can be viewed in an abstract way as consisting of nodes and arrows, where some of the arrows are bidirectional. In the architectural community the nodes are called concepts and the arrows are called relations. Depending on the topic of the diagram, in some cases there is an existing *standardized* XML vocabulary that can be used to provide an XML encoding of the diagram. For instance there is XMI to enable XML-based interchange of metadata between UML modeling tools. What is typically lost in such XML Metadata Interchange format are some of the visual elements: the positions of the boxes in the picture and the lengths of the lines for the arrows. An XML encoding only captures the names of the nodes and the arrows and what nodes are connected via which arrows. There can also be information in the XML encoding about attributes of the nodes and arrows, information that is not visible in the diagram but in the accompanying text in English. An example of such extra information is that a department consists of a maximum of 100 employees.

4.1. A formal basis for static analysis

The core of a symbolic model of an architecture consists of its *signature* which specifies its *name space*. The names of a signature are used to denote symbolically the structural elements of the architecture, their relationships, and their dynamics. The *nature* of each structural element is specified by a *sort* (a data type identifier), and each architectural relationship by a *relation* between sorts. Additionally, a signature includes an ordering on its sorts and its relations for the specification of a classification in terms of a generalization relation on the structural elements and the architectural relations. For example, the sort object in Figure 1 can be defined as a generalization of both the sorts Order_Registry and Product_Registry given in Figure 2, to indicate that every element in Order_Registry or Product_Registry is also an element of sort object. Also, an association between role and product is a generalization of the relation owns between Employee and Product.

The ordering on sorts and relations is in general used to capture certain aspects of the *ontology* of an architecture. Other ontological aspects can be captured by the aggregation and containment relations. For technical convenience however we restrict to the generalization relation only.

Definition 1 *A signature consists of*

- *a partially ordered set of* primitive sorts*, also called the* sort hierarchy*;*
- *a partially ordered set of* relations*, where each relation is of the form $R(S_1, \ldots, S_n)$, with R the name of the n-ary relation and S_i the primitive sort of its ith argument.*

We allow *overloading* of relation names, i.e., the same name can be used for different relations. For instance, given the primitive sorts $Person$, $Boss$, and $Employee$, the relations $Responsible(Boss, Employee)$ and $Responsible(Person, Person)$ are in general two different relations with the same name.

Further information about the architecture is expressed symbolically in terms of suitable extensions of one of its signatures. Usually a signature is extended with operations for constructing complex *types* from the primitive sorts. Examples are the standard type operations like *product type* $T_1 \times T_2$ of the types T_1 and T_2, and the *function type* $T_1 \rightarrow T_2$ of all functions which require an argument of type T_1 and provide a result of type T_2. Note that a relation $R(S_1, \ldots, S_n)$ is a sub-type of $S_1 \times \cdots \times S_n$.

Given functional types, the name space of a signature can be extended with *functions* $F(T_1) : T_2$, where F specifies the name of a function of type $T_1 \rightarrow T_2$. Functions can be used to specify the *attributes* of a sort. For example, given the primitive sorts $Employee$ and $\mathbb{N}$, the function $Age(Employee) : \mathbb{N}$ is intended for specifying the age of each person. Note that *multi-valued* functions $F(T_1, \ldots, T_n) : T'_1, \ldots, T'_m$ can be specified by the functional type $T \rightarrow T'$, where T denotes the product type $T_1 \times \cdots \times T_n$ and T' denotes the product type $T_1 \times \cdots \times T_n$. In general, functions are also used to specify symbolically the dynamics of an architecture.

The next example shows the signature of the business process architecture described in Figure 2.

Example 1 *The sorts of the example described in Figure 2 and 1 are simply enumerated by*

```
role
object
product
Employee
Product
Order_Registry
Product_Registry
```

Note that we did not include processes as a sort. The subsort relation is specified in AML by the following enumeration

```
is-a
    domain name=Employee
    codomain name=Role
is-a
    domain name=Order_Registry
    codomain name=Object
is-a
```

```
        domain name=Product_Registry
        codomain name=Object
    is-a
        domain name=owns
        codomain name=assignment
```

Note that we have encoded meta-model information of an architecture as part of the signature of the architecture itself. The relation between the meta-model sorts and relations and architectural sorts and relations is expressed by the respective partial orders between sorts and relations of the signature.

In AML the owns-relation itself is specified by

```
    owns
        domain name=Employee
        codomain name=Product
```

Finally, the processes are specified in AML as functions. The types of the arguments and result values are determined as follows: A role which is assigned to a process specifies the type of both an argument and a result value of the corresponding function. Similarly, an outgoing access relation from a process to an object specifies the type of both an argument and a result value of the corresponding function. On the other hand, an incoming access relation from an object to a process only specifies the type of the corresponding argument (this captures the property of 'read-only').

```
    Register_order_placement
        domain name=Employee
        domain name=Order_Registry
        codomain name=Employee
        codomain name=Order_Registry
    Place_order_for_product
        domain name=Employee
        codomain name=Employee
    Accept_product
        domain name=Employee
        domain name=Order_Registry
        codomain name=Employee
    Register_product_acceptance
        domain name=Employee
        domain name=Product_Registry
        codomain name=Employee
        codomain name=Product_Registry
```

Note that the triggering relation is not included in our concept of a signature. In our view such a relation specifies a temporal ordering between the processes which is part of the business process language discussed below in section 5.

Interpretation of types We first define a formal interpretation of the types underlying a symbolic model.

Definition 2 *An* interpretation I *of the types of a signature assigns to each primitive sort S a set $I(S)$ of* individuals *of sort S which respects the subsort ordering: if S_1 is a subsort of S_2 then $I(S_1)$ is a* subset *of $I(S_2)$.*

Any primitive sort is interpreted by a subset of a universe which is given by the union of the interpretation of all primitive sorts. The hierarchy between primitive sorts is expressed by the subset relation.

An interpretation I of the primitive sorts of a signature of an architecture can be inductively extended to an interpretation of more complex types. For example, an interpretation of the product type $T_1 \times T_2$ is given by the Cartesian product $I(T_1) \times I(T_2)$ of the sets $I(T_1)$ and $I(T_2)$. The interpretation of the function type $T_1 \to T_2$ as the set $I(T_1) \to I(T_2)$ of all functions from $I(T)1)$ to $I(T_2)$, however, does not take into account the *contra-variant* nature of the function space. For example, since the sort $\mathbb{N}$ of natural numbers is a sub-sort of the real numbers $\mathbb{R}$, a function from $\mathbb{R}$ to $\mathbb{R}$ dividing a real number by 2 is also a function from $\mathbb{N}$ to $\mathbb{R}$, but, clearly, the set of all functions from $I(\mathbb{R})$ to $I(\mathbb{R})$ is *not* a subset of the set of functions from $I(\mathbb{N})$ to $I(\mathbb{R})$.

Therefore, given the universe $\mathbb{U}$ defined as the union of all the interpretations of the primitive sorts, we define the interpretation of the function type $T_1 \to T_2$ by

$$I(T_1 \to T_2) = \{f \in \mathbb{U} \to \mathbb{U} \mid f(I(T_1)) \subseteq I(T_2)\}.$$

The function type $T_1 \to T_2$ thus denotes the set of all functions from the universe to itself such that the image of $I(T_1)$ is contained in $I(T_2)$. Note that if T_1' is a subtype of T_1 and T_2 is a subtype of T_2' then $I(T_1 \to T_2)$ is indeed a subset of $I(T_1' \to T_2')$.

In general, there can be a large number of different interpretations for a signature. This reflects the intuition that there are many possible architectures that fit a specific architectural description. In fact, a signature of an architecture basically only specifies the basic concepts by means of which the architecture is described.

Semantic models The semantic model of a system involves its concrete components and their concrete relationships which may change in time because of the dynamic behavior of a system. To refer to the concrete situation of a system we have to extend its signature with names for referring to the individuals of the types and relations. For a symbolic model, we denote by $n : T$ a name n which ranges over individuals of type T.

Given a symbolic model of an architecture extended with individual names and an interpretation I of its types, we define a semantic model Σ as a function which provides the following interpretation of the name space of the symbolic model covering its relations, functions, and individuals.

Relations For each relation $R(S_1, \ldots, S_n)$ we have a relation

$$\Sigma(R) \subseteq I(S_1 \times \cdots \times S_n)$$

respecting the ordering between relations, meaning that if R_1 is a sub-relation of R_2 then $\Sigma(R_1)$ is a subset of $\Sigma(R_2)$.

Functions For each symbolic function $F(T_1) : T_2$ we have a function

$$\Sigma(F) \in I(T_1 \to T_2).$$

Variables For each individual name $n : S$ we have an element

$$\Sigma(n) \in I(S).$$

4.2. XML for static analysis

In this section we describe the methodology we follow to design an XML vocabulary for diagrams like in Fig. 2 and 1. In general we will model every node in the diagram with an XML element. Figure 1 is a legenda, a collection of unconnected concepts and relation names with their visual representation. Only the concepts are given XML elements, not the relation names. For the concepts (rectangles and rounded rectangles) in Fig. 1 and 1 we design XML elements with that name. The lines in Fig. 2, and other relations that are mentioned in the accompanying text, will be modeled with XML elements with the name of the relation, and these elements will have `domain` and `codomain` children that contain cross-references to the elements that participate in the relation. This way it is possible to define n to m relations by taking n `domain` elements and m `codomain` elements. A designer could choose to take other names for `domain` and `codomain`, like `from` and `to`, but the methodology remains the same.

Section 4.1 shows examples for the various XML elements in the model. The complete XML model for static analysis for the example consists of a `businessprocess` element with as children elements the examples in Sect. 4.1.

All the concepts and relations from Fig. 2 and 1 and the explanatory text have been put into XML. The disadvantage of storing meta-information in an XML encoding, like in this case with `is-a` relations, is that the encoding risks to become too big and chaotic. The chaos can be improved upon with extra elements, for instance by putting the meta concepts (`process`, `role`, `object` and `product`) in a containing element called `meta`, but this still does not solve the size problem. If analysis is not using the meta information, then it can be omitted, or stored in an external file for future reference. In the above model this method would remove all the `is-a` relations and the four meta elements.

Our XML encoding does not make much use of the possibilities to use hierarchy between elements in XML itself. An example of using more XML hierarchy would be:

```
businessprocess
    role
        Employee
    object
        Order_Registry
        Product_Registry
    product
        Product
    process
        Register_order_placement
            domain name=Employee
            domain name=Order_Registry
            codomain name=Employee
            codomain name=Order_Registry
        Place_order_for_product
            domain name=Employee
            codomain name=Employee
        Accept_product
            domain name=Employee
            domain name=Order_Registry
            codomain name=Employee
        Register_product_acceptance
            domain name=Employee
            domain name=Product_Registry
            codomain name=Employee
            codomain name=Product_Registry
    owns
        domain name=Employee
        codomain name=Product
```

which is a more efficient encoding for the example, but our experience shows that it is generally a good idea to be cautious when using XML hierarchy. With this last encoding it will be more difficult for example to put the meta information in a separate file. And there are several kinds of relations in a model, like generalization, composition and association, that can be expressed with hierarchy in XML, but once we have chosen to use hierarchy in XML for generalization it will not readily be possible to use XML hierarchy also for composition relations when we want to add those later. In the case of modeling generalization there is also the problem of modeling what is known as "multiple-inheritance" in computer science: it is not generally possible to model a generalization of two concepts with XML hierarchy alone because an XML element only has one parent element. If generalization is very important and interesting for the analysis you have in mind then modeling it with XML hierarchy could possibly work out very well, but in our methodology we start out using as little XML hierarchy as possible.

XML individuals for semantic models So far we have only put sorts and relations into XML, but not individuals of sorts, necessary for semantic models. Putting the individuals into XML can be useful for several types of analysis, especially for analysing dynamics. In our methodology we can model individuals of a sort as XML children of the sort element, with all attributes that are needed as can be inferred from the text description of an architecture. The name of the children element is free to choose, but there could be a naming convention such that it is clear what sort an individual belongs to. For example, adding two individuals of sort `Employee` can be modeled with:

```
businessprocess
    ...
    Employee
        e1 order=Product product=p1
        e2 order=Product product=p2
    ...
```

where the `e1` and `e2` elements are `Employee` individuals and their `order` and `product` attributes have been added because the textual description of the architecture said that an employee has an order in mind and that an employee is handling a product. There is only one `Product` sort in our example, so the `order` attribute looks redundant, but we may want to add more products later.

Another approach is to put all the XML elements for sort individuals inside a `variables` element, and in that case it would be a good idea to give the individuals an attribute that designates their sort, like in

```
businessprocess
    ...
    Employee
    ...
    variables
        e1 sort=Employee order=Product product=p1
        e2 sort=Employee order=Product product=p2
        ...
```

where we see the use of an extra `sort` attribute. Of course another name than `variables` is possible. And of course their are many different approaches altogether, but with the two described here we have good experiences.

Examples of static analysis An example of static analysis is to analyse whether all name attributes of domain and codomain elements in the functions are defined as XML element names, and to do type checking if that is considered useful. Another example is to check if all the `is-a` relations are anti-symmetric. Yet another example is *impact analysis*.

To perform the static analysis there are many tools in the industry that can be used that are capable of parsing XML. These tools can be used to turn the XML in a graphical representation, or they can do things like counting the number of employees or adding their salary attributes. The RML tools can also be used. The RML tools are designed for *transformations* of XML to XML so they are more targeted at dynamic analysis, but it is very well possible to define transformations of XML that rearrange the input: for example displaying a list of employee elements. Due to a lack of space we can not already show examples of such RML transformations here, we refer to Sect. 5.2 for RML examples.

5. Dynamic analysis

5.1. A formal basis for dynamic analysis

We can model the dynamic behavior of a model of an architecture with a state-machine [3]. The transitions in the state-machine correspond with RML rules or recipes.

State machine semantics The sort individuals are coordinated by means of state machines. These state machines consist of *transitions* of the form

$$l \xrightarrow{[g]/a} l'$$

where l is the *entry* location and l' is the *exit* location of the transition. Furthermore, g denotes its boolean *guard* and a its *action*.

The boolean guard of a transition is a boolean expression that consists of the usual integer values and string values but also of RML-variables from the rule or recipe that is captured by the transition. For evaluating the guard these RML-variables will be assigned a value by the RML matching algorithm with the XML encoding of the model as input.

An action involves a call to the RML tools executing an RML rule or recipe on the model. For the action in the transition we generally use the name of the file the rule or recipe is stored in.

In the following we use *class* for sort and we use *object* for individual, because these names are more usual when describing state-machines, e.g. in UML.

In order to formally define the *operational* semantics of state machines in architectures we assume for each class c of a given architecture a set O_c of *references* to objects in class c. In XML such references can be modeled by means of `id` attributes with unique values, and cross-reference attributes. In case class c extends c' (according to the architecture) we have that O_c is a subset of $O_{c'}$. (For classes which are not related by the inheritance hierarchy these sets are assumed to be disjoint.)

Definition 3 *An* object diagram *of a given architecture with classes $c_1, \ldots, c_n$ can be specified mathematically by functions σ_c, for $c \in \{c_1, \ldots, c_n\}$, which specify for each object in class c existing in the object diagram the values of its attributes, i.e., $\sigma_c(o.A)$ denotes the value of attribute A of the object o, i.e., it denotes an object reference in $O_{c'}$, where c' is the (static) type of the attribute A (defined in the class c in the architecture).*

Often we omit the information about the class and write simply $\sigma(o.A)$. Control information of each object o in an object-diagram is given by $\sigma(o.L)$, assuming for each class an attribute L which is used to refer to the current location of the state machine of o.

Given an architecture consisting of a finite set of classes $c_1, \ldots, c_n$ and a state machine, we define its behavior in terms of a *transition relation* on the object diagram.

This transition relation is defined parametric in the semantics of the application operations.

More specifically, we assume for each action a involving an RML rule or recipe a *labeled* transition relation $\sigma \xrightarrow{a} \sigma'$ which specifies σ' as a possible result of the execution of the call a on σ.

Such a labeled transition describes the *observable* effect on the architecture of the execution of the corresponding call by the RML tools. As a special case we assume for each *guard* g a *labeled* transition relation $\sigma \xrightarrow{g} b$

where b denotes a boolean value which indicates the result of the evaluation.

Definition 4 *Formally, given an architecture and the semantic interpretations of the RML rules and recipes, we have a transition $\sigma \rightarrow \sigma'$ from the object-diagram σ to the object-diagram σ' if the following holds: there exists an object o and a transition*

$$l \xrightarrow{[g]/a} l'$$

in its state machine such that

Location $\sigma(o.L) = l$ *and* $\sigma'(o.L) = l'$*;*

Guard $\sigma \xrightarrow{g} true$*;*

Action *in case of a call a involving an RML rule or recipe we have*

$$\sigma \xrightarrow{a} \sigma'.$$

The first clause above describes the flow of control. The second clause states that the guard evaluates to true (without side-effects). A call to an RML rule or recipe is described in terms of a corresponding labeled transition which models the execution of the call by the underlying RML tools. Note that the execution of a transition of a state-machine is atomic. However, more fine-grained modes of execution can be introduced in a straightforward manner.

5.2. XML+RML for dynamic analysis

In our methodology we start with writing out scenarios. Scenarios consist of sequences of semantic models, called *scenes*, connected by functions, called *transitions*. We use the words scene and transition or transformation when discussing XML encodings. An example is an employee who registers an order in the order registry: the source-scene of the transition contains an employee with an order and an order registry, the target-scene contains the employee and the order registry with the order added. When we have collected enough examples of transitions, we define the RML rules that define the XML transformations from scene to scene. We could also try to define the RML rules without collecting scenes first, but using scenes has proven to be useful in practice and the scenes also provide a testbed to try the rules on, and later versions of rules. From source- and target-scene to an RML rule often does not involve much more than replacing literal strings with RML variables. The resulting set of RML rules can be used as actions in state-machines to define the behavior of an architecture. If a particular transition is too complex for 1 rule then a sequence of possibly iterating rules can be collected in an RML recipe, and the recipe can then be used as the action in the transition of a state-machine.

We now demonstrate our methodology applied to the ”Register order placement” process in the running example.

The XML contain a `businessprocess` element as shown before containing the sorts and relations from the symbolic model and a `variables` element where we keep the sort individuals. To save space we only show the `variables` section from now on.

A first scene consists of an employee and an order registry:

```
variables
    e1 sort=Employee order=Product
    order-registry
        Product
        Product
```

The XML element with the name `e1` corresponds to an *emp:Employee* variable in Sect. 5.1 and the XML element with the name `order-registry`, with its children, corresponds to a *or:Order_Registry* variable. These variables are parameters of a function *Register_order_placement* like in Sect. 5.1.

From this scene, the register order placement process leads to another scene:

```
variables
    e1 sort=Employee order=None
    order-registry
        Product
        Product
        Product
```

where the order attribute Employee is now None and the order for a Product has been added to the registry.

To produce a simplistic RML rule based on only these two scenes, we define

```
div class=rule name="Register order placement"
    div class=antecedent
        variables
            e1 sort=Employee order=Product
            order-registry
                Product
                Product
    div class=consequence
        variables
            e1 sort=Employee order=None
            order-registry
                Product
                Product
                Product
```

as the first version of the RML rule we want to develop for the process. To create this rule we simple copied the first scene in the antecedent of the rule, and we copied the second scene in the consequence.

This RML rule works, but only for employee elements with the name `e1`, and only for products of type `Product` as value of the order attribute of the employee. There could be other products e.g. `Product2` in the symbolic model and such products as value of the order attribute will not work. And the rule would only work when there are exactly 2 Products already in the registry where we want the rule to work with any number in the registry already. We can see these other possibilities by looking at other possible source scenarios we collected around this process.

```
variables
    e1 sort=Employee order=Product
    e2 sort=Employee order=Product2
    order-registry
        Product
        Product
```

```
variables
    e2 sort=Employee order=Product2
    order-registry
        Product
```

To make the rule work also on these other scenarios, we change the relevant literal strings in the rule into RML variables, according to table 3, leading to the second version of the rule:

```
div class=rule name="Register order placement"
    div class=antecedent
        variables
            rml-Employee sort=Employee
                         order=rml-P
            order-registry
                rml-list name=OldOrders
    div class=consequence
        variables
            rml-Employee sort=Employee
                         order=rml-P
            order-registry
                rml-use name=OldOrders
                Product
```

This rule is much better, but still not finished. This rule only works if there is exactly 1 employee sort individual defined and exactly 1 order-registry. But there could be other things defined in the variables section around the employee elements (we assume that an order-registry is always last in the variables section). If there are, the rule will not work since the first element does not match the pattern for an employee element as defined, or the second element is not an order-registry element. To copy such other elements in the variables section we change the rule,

```
div class=rule name="Register order placement"
    div class=antecedent
        variables
            rml-list name=Pre
            rml-Employee sort=Employee
                         order=rml-P
            rml-list name=Post
            order-registry
                rml-list name=OldOrders
    div class=consequence
        variables
            rml-use name=Pre
            rml-Employee sort=Employee
                         order=rml-P
            rml-use name=Post
            order-registry
                rml-use name=OldOrders
                Product
```

putting everything before the employee we want to match in RML variable `Pre` and putting everything after it, except the last element that must be `order-registry`, in `Post`.

A final addition to the rule is needed because an employee pattern in the rule now has a `sort` and a `order` attribute, but could very well have other attributes we want to keep in the output. This is done by adding an attribute `rml-others=Others` to the `rml-Employee` elements in the antecedent and in the consequence.

Now that we have defined this rule, we can define the first transition of the state-machine for this business process. To do this in XML we add a `statemachine` element to the `businessprocess` element, and with this first transition it looks like:

```
statemachine
    transition id=t1
        source state=start
        target state=state_1
        action
            implementation
                """Register order placement"""
```

When we have modeled the whole running example, there will be 4 `transitions` in the state machine, for the 4 processes in Fig. 1. A transition does not have to consist of an action alone, there can also be a guard with an guard-expression containing the usual things like string values and integers, but also RML variable names from the RML rule in the action. The guard-expression can be for example a

Java expression that can be evaluated by a Java interpreter, or it can be an OCL expression, or anything else suitable. The purpose of such a guard-expression is to constrain the applicability of the RML rule. For example to add the constraint that only orders of sort `Product2` or `Product3` may be added, a guard is added to the `t1` transition, resulting in:

```
state machine
    transition id=t1
        source state=start
        target state=state_1
        guard
            implementation
"""P == 'Product2' or P == 'Product3'"""
        action
            implementation
                """Register order placement"""
```

6. Summary and outlook

The techniques proposed in this paper enforce architects to think about the relation between their architectures and the real world. Static analysis techniques allow them to think about structural issues, like cardinality and “is-a” relationships. With dynamic analysis techniques, they can make small simulations the processes or other behavioural descriptions they propose. All these techniques improve the understanding of their own creations.

In this paper we have introduced a XML tool for static and dynamic analysis of enterprise architectures. We have shown how it transforms XML data and how it can be used to simulate business processes. A summary of the methodology we follow:

1. Create a symbolic model, see Sect. 4.2.
2. Collect scenes (semantic models) around transitions (functions).
3. Create RML rules using copy and paste from scenes.
4. Replace strings by RML variables in the RML rules where needed.
5. Create state-machines with the RML rules as actions in the state-machine transitions.

Acknowledgements This paper results from the ArchiMate project (`http://archimate.telin.nl`), a research initiative that aims to provide concepts and techniques to support architects in the visualization, and analysis of integrated architectures. The ArchiMate consortium consists of ABN AMRO, Stichting Pensioenfonds ABP, the Dutch Tax and Customs Administration, Ordina, Telematica Institute, CWI, University of Nijmegen, and LIACS.

References

[1] *The Description Logic Handbook: Theory, Implementation and Applications*, Cambridge University Press, 2003.

[2] F.S. de Boer, M.M. Bonsangue, J. Jacob, A. Stam, and L. van der Torre. A Logical Viewpoint on Architectures. In *Proceedings of the 8th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2004)*, IEEE Computer Society Press, 2004.

[3] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language*, Addison-Wesley, 1999.

[4] P. Chen. The Entity-Relationship Model–Toward a Unified View of Data. *ACM Transactions on Database Systems*, Volume 1(1):9 - 36, 1976.

[5] H. Eertink, W. Janssen, P. Oude Luttighuis, W. Teeuw, and C. Vissers. A business process design language. In *Proceedings of the 1st World Congress on Formal Methods*, 1999.

[6] C. Gane and T. Sarson. *Structured Systems Analysis: Tools and Techniques*. Prentice Hall, Englewood Cliffs, 1979.

[7] C.A.R. Hoare. *Communicating Sequential Processes*, Prentice Hall, 1985.

[8] IEEE Computer Society. IEEE Std 1471-2000: IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, Oct. 9, 2000.

[9] J. Jacob. The ASCII Markup Language (AML) whitepaper. Available at `http://homepages.cwi.nl/~jacob/aml`.

[10] J. Jacob. The Rule Markup Language: a tutorial. Available at `http://homepages.cwi.nl/~jacob/rml`.

[11] H. Jonkers. et al.. Towards a Language for Coherent Enterprise Architecture Description. In M. Steen and B.R. Bryant (eds.), *Proceedings 7th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2003)*, IEEE Computer Society Press, 2003.

[12] H. Jonkers et al.. Concepts for modelling enterprise architectures. *International Journal of Cooperative Information Systems*, 2004.

[13] G. Kahn. The semantics of a simple language for parallel programming. In *IFIP 7th Congress*. North Holland, Amsterdam, 1974.

[14] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

[15] World Wide Web Consortium. Extensible Markup Language (XML). Available at`http://www.w3.org/XML/`.

[16] J.A. Zachman, A Framework for Information Systems Architecture, IBM Systems Journal, Vol. 26, No. 3, 1987.