# A Logical Viewpoint on Architectures

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 Public Consulting, Rosmalen, The Netherlands*
[4] *Technical University of Delft, The Netherlands*

## Abstract

*In this paper we introduce a logical viewpoint on architectures. The logical viewpoint 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 leads to more precise characterization of the concepts introduced in IEEE standard 1471-2000, and provides a formal approach to the design of enterprise of architectural description languages and a general mathematical foundation for the use of formal methods in enterprise architectures.*

*Additionally, we show how this logical viewpoint allows for the definition of a simple general XML language for the description of both static and dynamic aspects of an architecture. For the meta-analysis of both these aspects we introduce a new XML tool for general XML transformations based on a Rule Markup Language.*

## 1. Introduction

In this paper we consider the gap between abstract enterprise architecture descriptions and much more detailed business process models. The problem of analyzing and simulating enterprise architectures is that they are described in much more vague terms than business process models. For example, the IEEE standard 1471-2000 is based on the notion of the viewpoint of a stakeholder with a set of concerns, and it defines view, architectural description, architecture and system accordingly. However, despite the fact that this approach has led to a useful reconsideration of the concepts used in architecture, the drawback is that it does not lead to concepts which are precisely defined in a mathematical sense, and consequently it is neither very clear how to bridge the gap between architectural descriptions and business process models, nor how to incorporate the architectural concepts in tools.

In this paper we study the following two research questions.

1. How to incorporate business process models in enterprise architectures to analyze and simulate their behavior?
2. How to provide architectural tool support for enterprise architectures with business process models?

To incorporate business process models in enterprise architectures, we believe that we have to extend the IEEE architectural concepts. Consider the main IEEE 1471 standard descriptions.

**architecture:** The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.

**system:** A collection of components organized to accomplish a specific function or set of functions.

**system stakeholder:** An individual, team, or organization (or classes thereof) with interests in, or concerns relative to, a system.

**view:** A representation of a whole system from the perspective of a related set of concerns.

**viewpoint:** A specification of the conventions for constructing and using a view. A pattern or template from which to develop individual views by establishing the purposes and audience for a view and the techniques for its creation and analysis.

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

†Corresponding author. Email: `L.W.N.van.der.Torre@cwi.nl`

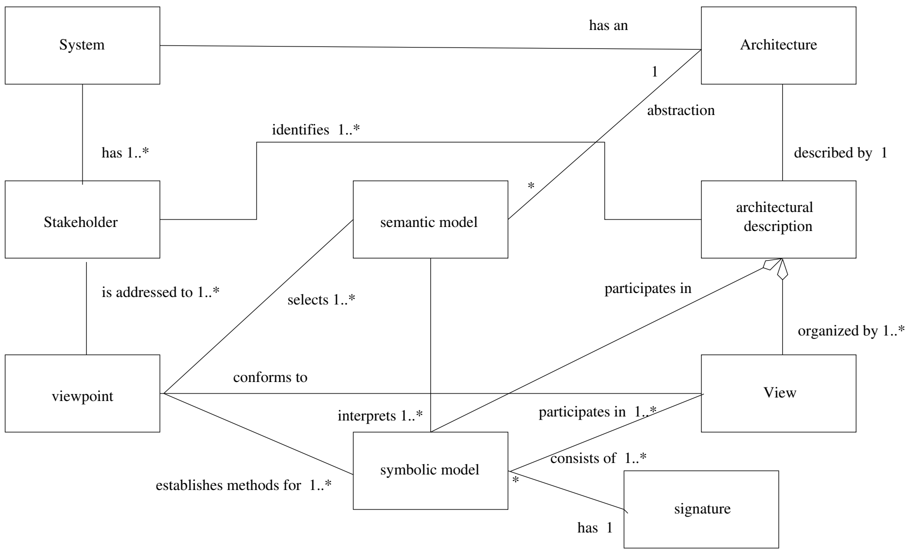


**Figure 1. Extension of IEEE with LVA**

These descriptions do not reflect the distinction between enterprise architectures and business process models. Our extension of the IEEE conceptual model is visualized in Figure 1, in which a symbolic model corresponds to the IEEE concept of model, and which contains the two new concepts semantic model and signature (we leave out IEEE 1471-2000 concepts not related to our new concepts).

**Semantic model.** The missing concept in the IEEE 1471-2000 to bridge the gap between enterprise architectures and business process models is the notion of a semantic model, which *interprets* symbolic models.

**Signature of an architecture.** Moreover, each symbolic model has a signature, which contains besides the usual concepts and relations (including special relations like is-a) also functions. The functions play a crucial role in our process models, as some of them are interpreted by *actions* in the semantic model.

Finally, in contrast to IEEE 1471-2000 we distinguish between the conceptualization of an architecture and its visualization (though this is not visualized in Figure 1).

Concerning tool support, our logical viewpoint provides the formal foundations for the use of XML as a representation language for the signature of an architecture, and more generally as a representation language for symbolic as well as semantic models. In this paper we use AML instead of XML, which is equivalent with XML, but designed to be better readable for humans. Roughly, in AML the end tags and angle brackets are replaced by indentation principles.

Moreover, we promote the use of the Rule Markup Language or RML as a language to describe model transformations and thus actions. As explained in detail in this paper, actions are interpreted as functions, and can thus be described by their input/output behavior, which can be described by transformation rules. RML consists of a small 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.

We do not develop one particular approach to formalize architectural descriptions, such as existing object-oriented languages, algebras, or (description) logics. What we propose, on the other hand, is the much more abstract logical way to look at architectural descriptions, which can be instantiated in many ways.

The motivation of our study is to bridge the gap between abstract IEEE 1471-2000 descriptions and the detailed ArchiMate language for modeling enterprisae architectures [9]

The layout of this paper is as follows. In Section 2 we introduce a running example to explain our definitions. In Section 3 we explain the signature, the distinction between symbolic and semantic model, and the actions. In Section 6 we discuss tool support, XML, AML and RML.

## 2 Archimate: a running example

Archimate is an enterprise architecture modeling language [9, 10]. It provides through a meta-model, 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 [3] but it has also a UMLflavor, introducing concepts like interfaces, services, roles and collaborations.

In the remainder of this paper, we will consider as running example, the enterprise architecture of a small company, called *ArchiSell*, modeled using the Archimate language. 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 to an owner, responsible for selling the product.

To describe this enterprise we will need the ArchiMate meta-concepts and their relationships as presented in Figure 2. In particular, we will 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.

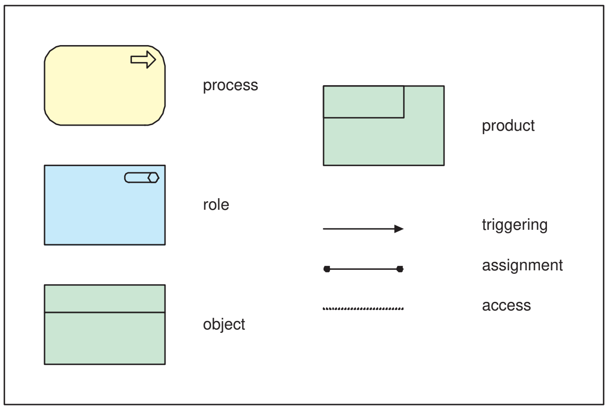


**Figure 2. Some concepts and relations**

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 processes describes the temporal relations between them. When executed, processes may need to access data, whose representation is here called object.

We will specifically look at the *business process architecture* for ordering products, depicted in Figure 3.

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.

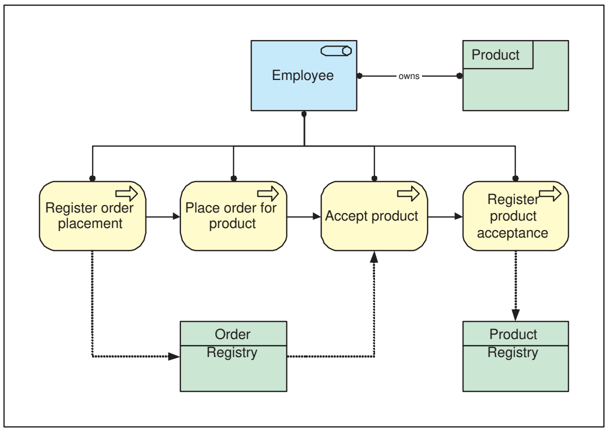


**Figure 3. A Business Process Architecture**

- 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).

## 3. Systems and architectures

Following IEEE 1471-2000, every system has an architecture. Our logical perspective abstracts from pragmatics, like design principles. Therefore an architecture is the structure and dynamics of a system consisting of its components and their relationships.

The architecture of a system is purely conceptual and different from particular symbolic descriptions of that architecture. An architectural description consists of several symbolic models (also called model in [6]) and other pragmatic information. Examples of the latter are the architectural rationale. In the next sections we focus on the logical nature of these symbolic models which involves their syntax and semantics.

### 3.1. The signature of an architecture

The very 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*, 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 2 can be defined as a generalization of both the sorts Order_Registry and Product_Registry given in Figure 3, to indicate that every element of 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 $i$th 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*

$$T_1 \to T_2$$

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 $T_1 \to T_2$. 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 $T \to T'$, 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 3. It is written in AML, a human-understandable notation for generating XML documents. AML and the corresponding tool-support will be discussed in Section 6.

**Example 1** *The sorts of the example described in Figure 3 are simply enumerated in AML by*

```
Role
Object
Employee
Product
product
Order_Registry
Product_Registry
```

*Note that we did not include processes as a sort (in our logical view explained above, processes are modeled as functions). The subsort relation is specified in AML by the following enumeration*

```
is-a
    domain name=Employee
    codomain name=Role
is-a
    domain name=Product
    codomain name=product
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=association
```

*Note that we have encoded meta-model information of an architectural description 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. For example, the sort* Product *in Figure 3 is modeled as a subsort of the meta-concept* product *in Figure 2.*

*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 4.1.*

The recommendation IEEE 1471-2000 [6] emphasizes that views on an architecture should be seen from the perspective of a viewpoint of a stakeholder, that has several concerns. In our logical characterization, a viewpoint is essentially a partial transformation over signatures, and a view is a visualization of the result of the transformation, given a visualization.

Summarizing, 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 be easily formalized in XML for storage and communication purpose, and can be integrated as an independent module with other tools including, e.g., graphics for visualization. In the following sections we define the formal *semantics* of a symbolic model of an architecture. Such a semantics provides a formal basis for the development and application of tools for the *logical analysis* of the dynamics of an architecture.

### 3.2. Interpretation of Types

In this section 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 $T_1 \rightarrow T_2$ as the set

$$I(T_1) \rightarrow I(T_2)$$

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 $T_1 \rightarrow T_2$ by

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

The function type $T_1 \rightarrow T_2$ 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 $I(T_1 \rightarrow T_2)$ is indeed a subset of $I(T_1' \rightarrow T_2')$.

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.

## 4. Semantic models

In our logical perspective, a semantic model is a formal *abstraction* of the architecture of a system. The logical perspective presented until now, only concerned the symbolic representation of an architecture by means of its signature. Next we show how to obtain a formal model of a system as a semantic interpretation of the symbolic model of its architectural description.

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 $\Sigma$ 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

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

**Example 2** *For our running example we introduce the following semantic model. In this model we have only two products p1 and p2. This is specified in AML by*

```
Product
    p1
    p2
```

*In order to model the processing of orders and products individuals of the sort Employee have a product attribute and an order attribute. These attributes indicate the order and product the employee is managing. These attributes can also be viewed as providing an* interface *to the environment consisting of the clients and suppliers. Both the order of a client and the product of a supplier will be stored by an employee (not necessarily the same employee). In our model individuals of the sort Employee are fully characterized by these attributes. Therefore in our model the sort Employee contains four elements, as described in AML by*

```
Employee
      e1 order=p1 product=p1
      e2 order=p1 product=p2
      e3 order=p2 product=p1
      e4 order=p2 product=p2
```

*In our simple model both the Order_Registry and Product_Registry can contain only information about one of the two products p1 and p2 (in section 6 we discuss how to model an Order_Registry as a finite list of orders). Consequently, we can identify in this simple model the interpretation of these sorts with that of Product:*

```
Order_Registry
     p1
     p2

Product_Registry
     p1
     p2
```

*The interpretation of the processes of our running example in this model are specified in AML by means of matrices of input/output pairs. For example, in the following we illustrate two such input/output pairs belonging to the interpretation of Register_order_placement: it replaces the product stored in the Order_Registry by the product stored in the order of the employee:*

```
matrix function=Register_order_placement
    input
        e1 order=p1 product=p1
        p1
    output
        e1 order=p1 product=p1
        p1
    input
        e1 order=p1 product=p2
        p2
    output
        e1 order=p1 product=p2
        p2
```

*The other processes are formally described in a similar manner. Because of space limitations we restrict to a informal description of their interpretations.*

*The function Place_order_for_product does not affect the information stored in an employee (in more refined models this function may in fact describe an update which records information about the supplier involved).*

*The function Accept_product simply checks whether the product managed by an employee is stored in the Order_registry. We model this check as a* partial *function which contains only those input/output pairs for which the product stored in the Order_registry coincides with the product managed by the employee. Note that the product managed by the employee results from the delivery of a supplier and that the order managed by an employee may have changed after it has been stored in the Order_registry.*

*The function Register_product_acceptance simply stores the product managed by the employee in the Product_registry.*

*Finally, in order to refer to the elements of the different sorts we introduce individual names emp:Employee, order-reg: Order_Registry, and product-reg: Product_Registry. A semantic model assigns individuals to these names, for example, such an assignment is specified in AML simply by*

```
emp = e1 order=p1 product=p1
order-reg= p1
product-reg= p2
```

*Note that this assignment describes an employee which manages an order of product p1 and a delivery of product* *p1, an Order_registry which registers an order of product p1, and a Product_registry which registers the acceptance of a product p2.*

### 4.1. Dynamics of a system

The dynamics of a concrete system with an architectural description given by its signature can be specified in different ways. Below we distinguish two different use of functions to describe the dynamics of a system: one where functions are seen as primitive actions that change the state of a system, and another where functions are seen as data transformers.

In the first case, we define the *action of a function* $F(S){:}T$ by an assignment of the form

$$n{:}= F(m)$$

where $n{:}T$ and $m{:}S$ are names ranging over the types $T$ and $S$, respectively. The execution of such an action in a semantic model $\Sigma$ *assigns* to the name $n$ the return value of

$$\Sigma(F)(\Sigma(m))$$

which denotes the result of applying the function $\Sigma(F) \in I(S \to T)$ to the element $\Sigma(m) \in I(S)$. Note that actions transform semantic models (i.e. the state of a system) but not the interpretation of a signature (i.e. the structural information of a system).

**Example 3** *Given the interpretation of the individual names e and or of the example 2, the execution of the action*

e,or:=Register_order_placement(e,or)

*results in the new semantic model* $\Sigma'$ *such that* $\Sigma'(\mathrm{or}) = \mathrm{p1}$.

Given this concept of an action as a transformation of semantic models, we can define more complex *processes* by combining actions, that is, we can define operations on actions determining the order of their execution. For example, we can define the sequential composition $n{:}= F(m);n'{:}= G(m')$ of two actions $n{:}= F(m)$ and $n'{:}= G(m')$ as the composition their transformation of semantic models. Other operations on actions include case structure, loops, parallel composition, and synchronization.

**Example 4** *Given the above sorts Product and Employee, and a function name Produce of type Employee* $\times$ *Product* $\to$ *Product, we can define a* pipe-line *by*

p1:=Produce(e1,p1);p2:=Produce(e2,p1),

*where e1 and e2 denote individual employees and p1 and p2 denote some products.*

The above interpretation of functions as actions forms a formal basis for the introduction of process algebras and corresponding analysis techniques in business process modeling. A process algebra [5] is a structured approach for constructing complex processes out of actions. Alternatively, we can use functions to specify the *data-flow* in a system illustrating how data is processed in terms of inputs and outputs. In this view a multi-valued function

$$F(T_1, \ldots, T_n){:}T'_1, \ldots, T'_m$$

is interpreted as an *asynchronous* process transforming data as follows. It has an *input channel* for each of its arguments; when on each input channel data, i.e., an element of the corresponding type, has arrived it *outputs* the result values on corresponding *output channels*. Such processes can be connected via their channels in a *data-flow network* [11] pictorially represented by a Data Flow Diagram [4]. Because of space limitations we omit the formal details.

## 5 Design support

In this section we discuss the support that can be offered by our logical perspective to describe the evolution of a system. In particular we will briefly describe the role of logical languages and design action in the design of an architecture.

### 5.1. Logical languages

Logical extension of a signature consists in considering types as predicate symbols that can be combined into more complex formulas by means of logical operators like conjunction and disjunction. The resulting logical language can be used to constraint the set of semantic models under consideration. There are several logical languages that can be used as logical extensions of a signature, and a more detailed description of them is beyond the scope of this paper. We just mention here description logics [1] as formalism for constraining semantic models and for reasoning about architecture. They are tailored towards a representation of architecture in terms of concepts and relationships between them. A description logic system consists of the following components:

1. a description language to construct complex description from simple ones;
2. a specification formalism to make statements about how concepts and relations are related each other (TBox) or to make assertions about individuals (ABox)
3. a reasoning procedure.

The advantage of using description logics is that they can be formulated in terms of diagrams, called the Entity-Relationships Diagrams (ERD) [2]. Basically they illustrate the logical structure of a system in terms of concepts and their relationships.

Temporal logics [12] are specially tailored towards the specification of the dynamic aspects of a systems. They consist of some atomic predicates on the semantic models together with the propositional connectives and some temporal operators like next (X), until (U), some time in the future (F), and always in the future (G). In our view, a temporal logic is interpreted in terms of sequences of semantics models generated by the actions of the symbolic model. For example the formula

```
emp.order = p1
implies
(emp.order = p1 U order_reg = p1)
```

specifies that if employee emp has received an order for product p1, then eventually the order will be register and until then the employee cannot process any different order.

### 5.2. Design actions

A design action is a transformation between symbolic models. It contains some additional non-logical information that can used to describe the evolution of the system. Examples are actions for adding sorts or relations, for deleting them, or for renaming them. Design actions can be realized by means of rules (for example expressed in RML) that have as antecedent a set of parameter and as consequence a description of the change. When the parameters are collected the rule can fire resulting in a new symbolic model as described in consequence of the rule.

## 6. Tool support

In this section we discuss how our logical perspective provides a formal basis for the integration of XML based tools for the semantic analysis of architectures.

The Extensible Markup Language (XML) [13] is a universal format for documents containing structured information so that 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 the 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) [7] used 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. The example below shows an AML document in which syntactic elements like the end tags and the angle brackets are replaced by appropriate indententation.

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

In the next sub-section we describe 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.

### 6.1. The Rule Markup Language

RML stands for Rule Markup Language. 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. The set of RML constructs is concise and shown in Table 4.

Rules defined in RML consist of an antecedent and a consequence. The antecedent defines a pattern and variables in the pattern. Without the RML constructs for variables this pattern would consist only of elements from the chosen XML vocabulary. The pattern in the antecedent is matched against the input XML. The variables specified with RML constructs are much like the wild-card patterns like `*` and `+` and `?` as used in well known tools like `grep`, but the RML variables also have a *name* that is used to remember the matching input. Things that can be stored in RML variables are element names, element attributes, whole elements (including the children), and lists of elements.

If the matching of the pattern in the antecedent succeeds then the variables are bound to parts of the input XML and they can be used in the consequence of an RML rule to produce output XML. When one of the RML tools applies a rule to an input then by default the part of the input that matched the antecedent is replaced by the output defined in the consequence of the rule; the input surrounding the matched part is kept intact.

Below we show an example of RML by presenting the rule that defines the state transformation of the action

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

| element | attribute | A | C | meaning |
|---|---|---|---|---|
| | | | | Elements that designate rules |
| `div` | `class="rule"` | | | |
| `div` | `class="antecedent" context="yes"` | | | |
| `div` | `class="consequence"` | | | |
| | | | | 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 4. All the RML constructs**

of our running example (emp and order-reg are individual names for an employee and the Order_registry, respectively). 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
            or
                orders
                    rml-list name=oldOrders
    div class=consequence
        variables
            rml-Employee order=rml-OrderName
                product=rml-ProductName
            or
                orders
                    rml-use name=oldOrders
                    order name=rml-OrderName
```

In the `antecedent` of the rule the matching algorithm first looks for an element with name `variables` which contains that part of the AML representation of the semantic model discussed 2 that stores the values of the names emp (of sort Employee) and order-reg (of sort Register_order) If that is found it looks for children of that element: one child with an `order` and `product` attribute (an employee), and one child with the name `r1` (the order registry). The algorithm binds the employee name emp to RML variable `Employee` and it binds the values of the `order` and `product` attributes to `OrderName` and `ProductName` respectively. The list of old orders, a list of XML elements that are the children of the `orders` child of the `r1` order registry, is bound to RML variable `oldOrders`. In the consequence of the rule the variables are reused in the output and an order element with the correct name is appended to the `oldOrders` list. Note that by means of this RML rule we have extended the semantic model of our running example to an interpretation of the sort Order_registry of *unbounded* capacity.

We see here that in a straightforward way, thanks to the wild-card matching technique used in RML, 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 Style-sheet 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. This because XML transformations usually involve the creation of links between elements by means of cross-referencing attributes, or the reordering, the addition or removal of elements, but does typically not include things like integer arithmetric and floating point calculations.

RML does not define, need, or use another language, it only adds a few constructs to the XML vocabulary used, like the wild-card pattern matching. 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.

### 6.2. RML as a tool for architectural description

As illustrated above, with RML a formal definition can be given of the dynamics of the basic actions of an architecture in terms XML transformations. This allows for a formal use of process algebras [5] in the modeling and analysis of business processes. In fact, the use of RML allows the formal definition of one own's business process constructs on top of the semantic description of the basic actions.

As a simple example, the execution of an action a by the process P=a.b that specifies a temporal order between a and b (namely, first a and than b), can be described in a process algebra by a *transition* of the form (we abstract from the state)

```
a.b -> b
```

As a transformation in RML this transition can be specified by the following rule:

```
div class=rule
    div class=antecedent
        process name=rml-P
            prefixes
                rml-A
                rml-B
    div class=consequence
        process name=rml-P
            prefixes
                rml-B
```

The removal of the `a` prefix is easily specified in such an RML rule as the removal of an element from a list of children elements.

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 the RML tool will have to be combined with another tool that can do the desired calculation. For modeling business architectures a transformation that can not be expressed with XML+RML alone is rather uncommon, but they may occur when the user is interested in a simulation of a model. We have applied combinations of RML with other components like programming language interpreters 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.

## 7. Summary and outlook

In this paper we consider the relation between enterprise architectures and much more detailed business process models. The missing link to bridge the gap between the two worlds is the notion of a semantic model in the IEEE 1471-2000 standard [6]. We show how semantic models can be distinguished from the models used within the standard, which we call symbolic model. This distinction provides a formal basis for the introduction of a formal definition and analysis of business processes. Moreover, we extend the IEEE standard with the notion of the signature, which serves as the basis of the enterprise architecture description, as well as the semantic model.

Semantic models are at the center of our logical perspective on enterprise architectures which integrates both static and dynamic aspects. The framework we have developed allows the integration of various models for business processes, ranging from process algebras to data-flow networks.

Furthermore, we have introduced a XML tool for the transformation of XML data and showed how it can be used to simulate business processes.

There is a rich literature of business processes. However, as far as we know, our logical perspective is a first attempt to a formal integration of such processes in enterprise architectures. We believe that our logical framework (plus tool support) also provides an promising basis for the further design and development of business process languages and corresponding tools.

**Acknowledgments** 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. Thanks to the anonymous referees for their constructive comments.

## References

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

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

[3] 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.

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

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

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

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

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

[9] H. Jonkers, R. van Buuren, F. Arbab, F.S. de Boer, M.M. Bonsangue, H. Bosma, H. ter Doest, L. Groenewegen, J. Guillen-Scholten, S. Hoppenbrouwers, M. Jacob, W. Janssen, M.Lankhorst, D. van Leeuwen, E. Proper, A. Stam, L. van der Torre, and G. Veldhuijzen van Zanten. 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)*, pages 28-39, IEEE Computer Society Press, 2003.

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

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

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

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