A MOF-Based Metamodel for SA/RT

Joakim Isaksson, Johan Lilius, and Dragos Truscan[⋆]

Embedded Systems Laboratory, Turku Centre for Computer Science,
Lemminkäisenkatu 14A, 20520 Turku, Finland

Abstract. We present a MOF-based metamodel for the SA/RT (Structured Analysis for Real-Time Systems) design method. The metamodel provides a well-defined interpretation of the SA/RT elements, allowing the designer to create unambiguous specifications of systems. The metamodel was designed to be easily combined with UML and UML tools, in order to provide support for creating, editing and manipulating SA/RT models in UML environments. The approach allowed us, to specify and implement automated model transformations both between SA/RT models and between SA/RT and UML models.

1 Introduction

In light of increasing complexity of today's embedded systems, model-driven development has become one of the necessary solutions to handle high complexity and to ensure consistency of the specifications at different steps during the development process. Usage of models allows designers to raise the level of abstraction and to use various views to describe the system at different levels of detail, thus shifting the focus from implementation concerns to solution modeling. To fully take advantage of a model-based approach, appropriate tool support is required. Not only means to (graphically) create, edit and manipulate model elements, but also scripting facilities to support automated manipulation and consistency checking of such models are required, in order to speed up the design process and to cut down development times.

Recently, OMG started to promote the Model-Driven Architecture (MDA) initiative [1]. The main idea behind MDA is to define and use well-defined models to represent the system, and model transformations to go from requirements to specific implementations, assisted by appropriate tools. The main modeling language of MDA is, unsurprisingly, represented by the Unified Modeling Language (UML) [2], but other languages can be addressed too. MDA suggests the Meta-Object Facility (MOF) [3] as the main language for defining modeling languages. A MOF model can be seen as a meta-model for other meta-models, or, using the standard metamodeling terminology, a meta-meta-model. The strength of a MOF-based metamodel resides in the possibility to define and integrate other graphical notations into UML tools, thus allowing creating, editing and manipulating of models in a graphical fashion.

One such modeling language is represented by the Structured Analysis for Real-Time Systems (SA/RT), a graphical design notation focused on analyzing the functional

[⋆] Support for this work from the HPY and TES research foundations is gratefully acknowledged.

behavior and information flow through a system. Although one of the most frequently used methods in designing embedded applications, in recent years Structured Analysis has been largely overshadowed by UML and object-oriented methods. In our opinion, the supremacy of UML is largely due to the tool support it provides. However, we believe that structural methods provide, in certain situations, important advantages compared to object-oriented methods and that sometimes a combination of both is not only useful, but also fundamental for developing complex embedded systems [4]. We presented in [5] such a model-driven approach along with a discussion of the related work in combining object oriented and data flow views of the systems.

In this paper, we present a MOF-based metamodel for SA/RT that, by providing a well-defined interpretation of the SA/RT elements, enables the designer to create unambiguous representations of systems in a tool supported manner. The goal of this work was three fold: (1) to explore the flexibility of MOF in creating and using meta-languages; (2) to specify and implement a MOF metamodel for SA/RT that can be used either as a stand-alone tool or in combination with other MOF metamodels (e.g., UML) in a common modelling environment; (3) to investigate the support for implementing automated model transformations not only between models of the same metamodel, but also between models of distinct metamodels.

Following, the paper presents in Sect. 2 how the metamodel was defined and built starting from the SA/RT specification. Section 3 explains the metamodel implementation in our SMW tool along with the implementation decisions taken during the process. Short examples of scripts are given in Sect. 4 to illustrate how the MOF-based approach could be used to provide automation for model interrogation and manipulation, and to implement model transformations between different models. The paper ends with some concluding remarks.

2 A Metamodel for SA/RT

This section presents the SA/RT metamodel and the methodology used to develop it. One of our goals was to be able to incorporate data flow models and data flow information with UML models. We tried to keep the SA/RT metamodel as compatible with the UML metamodel as possible, although it also functions perfectly well as a stand-alone model. In practice, this means that the core framework of the metamodel is quite similar to the UML metamodel core, but with some unnecessary features removed, making the SA/RT metamodel directly connectable to the standard UML 1.4 metamodel.

2.1 The SA/RT Notation

One apparent problem when creating models using the SA/RT approach is that different tools and practitioners interpret the SA/RT notation in different ways [6], a fact which can give rise to inconsistencies and confusion. A well-defined metamodel for SA/RT could help to resolve such problems.

Since its introduction, several alternative interpretations of the SA/RT notation have been presented in the literature. To create a metamodel it is thus necessary to decide upon one interpretation of the standard. Our approach is to use the original specification by

Ward and Mellor [7] as the starting point and use it as far as possible. A second variant of the SA/RT notation was proposed by Hatley and Pirbhai [8], with the difference that it separates the dataflow and control information into two separate views, making in our opinion the models more difficult to understand. This is certainly not the only example of differences between the Hatley/Pirbhai and Ward/Mellor methodologies, for there are some minor notational differences in the definitions of the data-flow diagrams, but we think that the better clarity of the Ward/Mellor approach motivates our choice. The basic building blocks of a SA/RT system do remain the same regardless of the different notational dialects: systems are described using `data-flow diagrams`, `finite state machines` and `data dictionaries`. These are briefly described below.

Data Flow Diagram (DFD) is the main diagram used for structured analysis, modeling the data flow through the entire system along with the manipulations done to this data. A DFD consists of three main kinds of components: `transformations`, `stores`, and `flows`. A `transformation` performs some operation on the information it receives as input, after which the modified information is produced as output, while a `store` only stores the information it receives as input, eventually passing it on unmodified to another model element. `Flows` act as the glue of the system, connecting `transformations` and `stores` together and transporting the information between them. DFDs can (and should) be hierarchical, where the upmost layer is actually a context diagram describing the interfaces between the system and the outside world, and the `external entities` interacting with these interfaces. Lower layers then refine the system until the functionality of all transformations has been described.

An important and distinguishing characteristic of the DFD is the separation of data and control information, meaning that there are both control and data variants of the three main components listed above. `Control transformations` process control events, while `data transformations` process data or special control events (e.g., enabling and disabling the data transformations). Similarly, `data stores` and `data flows` only handle data, while their `control flows` counterparts only accept events. How to describe the functionality of the data transformations is not specified in detail in the literature, but typically this can be done using pseudocode or flowcharts. The functionality of `control transformations` is described using state machines. There are other rules regarding how different system components may be connected to each other, but due to space reasons we omit them here.

State Machines are used to model `control transformations`. The specification in [7] describes a rather simple state machine model, without any of the syntactic sugar found the statechart definition of UML (e.g., hierarchical and history states).

Data Dictionaries describe the data flowing through the system, where the contents and structure of each data element is described in detail. Ward and Mellor suggest a regular-expression like notation to denote this information, as the data may be refined into several smaller subsets of the original data type.

2.2 The Metamodel

The four main components in a MOF-based metamodel are: `packages`, `classes`, `associations` and `data types`. Meta-objects in the target metamodel are modeled using `classes`, while the `associations` model binary relationships between

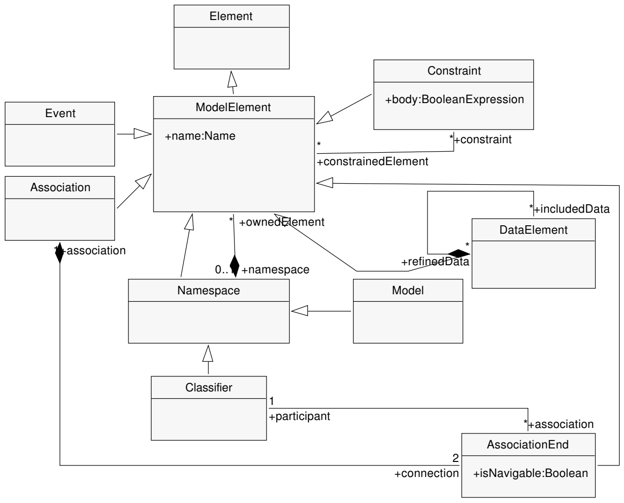


Fig. 1. The Core package of the SA/RT metamodel

these meta-objects. Data types model data such as primitive and external types. Packages are used to make the model more manageable by modularizing it. “Standard” object-oriented features are supported, i.e. classes may have attributes, inheritance and aggregation associations are included, etc. In practice, a MOF-based metamodel can be constructed in much the same way as one would construct any class diagram for an object-oriented design. Additional constraints on the metamodel can be specified through well-formedness rules which must hold in order for the model to be considered correct. The well-formedness rules which are defined for the MOF standard are expressed using the Object Constraint Language (OCL).

Our SA/RT metamodel is divided into six packages. The Core package contains fundamental metamodel elements needed by the other packages, while the Dataflow, ActivityGraph and StateMachine packages describe the actual diagrams of an SA/RT model. Additionally, we use the Auxiliary and Expressions packages to reduce the size of the Core package.

Since we wanted to retain the possibility to incorporate data flow modeling capabilities with UML, the Core package (Fig. 1) is essentially a subset of the UML core model. The main differences are the addition of the DataElement class, which represents the different types of data flowing through the system and which may in turn be subsets of other data types, and the simplification or omission of some constructs such as the Event class, which does not need the associated signature it has in the corresponding UML metamodel, as events do not carry any data values in SA/RT.

The Dataflow package (Fig. 2) defines the elements that may be included in a DFD. There are essentially three main classes and their subclasses that provide all the model elements allowed.

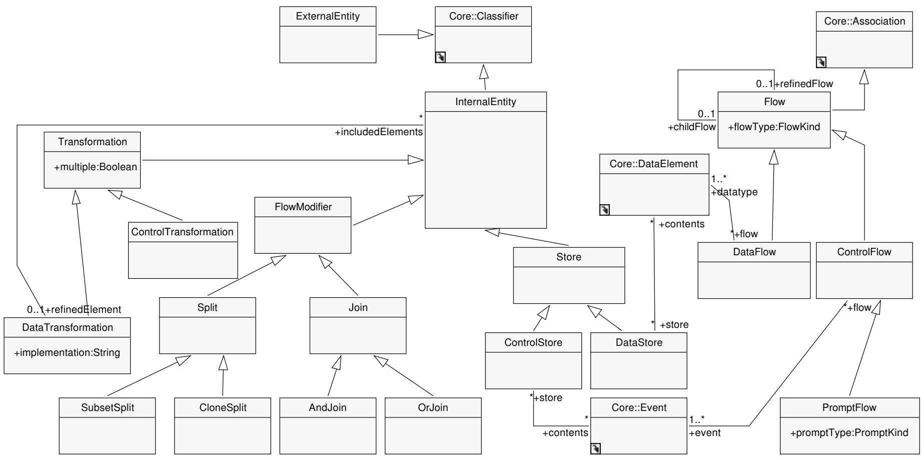


Fig. 2. The Dataflow package of the SA/RT metamodel

- The `ExternalEntity` class represents entities outside the system boundaries. Interacting through the system-level interfaces, these entities can provide input to the system or receive output from the system.
- The `Flow` class defines the flows between the transformations and stores in the DFD. The subclasses `DataFlow` and `ControlFlow` define flows for transporting data and events, respectively. The `PromptFlow` class models the special control flows enabling and disabling data transformations. The `flowKind` attribute specifies whether the flow is continuous or discrete, while the `refinedFlow`-`childFlow` association represents the connection between a flow in a higher-level DFD and the corresponding refined flow in the lower-level DFD.
- The `InternalEntity` class is used to define the `Transformation`, `Store` and `FlowModifier` elements, from which the elements of the DFDs are defined.
  - The subclasses of `Transformation` define `DataTransformations` and `ControlTransformations`. Their internal representation is specified by the `StateMachine` and `ActivityGraph` packages, respectively.
  - `FlowModifier` subclasses are used to model the connection points where the contents of a flow can be modified. `Split` either splits a flow into a number of identical copies (`CloneSplit`) or into a number of subsets (`SubsetSplit`) of the original flow. `Join` either joins the contents of several flows into one flow (`AndJoin`) or creates one flow containing one of the original flows (`OrJoin`).
  - `ControlStore` and `DataStore` model event and data repositories.

The Ward/Mellor book defines a very simplistic state machine structure for modeling control transformations, but there is of course nothing that prevents us from using some other, more complex, state machine model. Because of this, and because of our metamodel's compatibility with the UML core metamodel, we instead chose to use the statechart metamodel of UML as the basis for our `StateMachine` package, which defines the state machine part of the metamodel. Similarly, since the SA/RT specification does not impose a certain approach and because we wanted to employ a graphical approach, we decided to use, beside plain text, activity diagrams to represent the internals of the data transformations. We defined them in the `ActivityGraph` package. As these metamodels are essentially identical to the UML `stateChart` and UML `activityGraph` metamodels, we omit them here. We also omit the presentation of the `Auxiliary` and `Expressions` packages, but they can be found in [9].

In addition to the graphical metamodel, additional constraints have to be enforced on the model. For example, one such rule states that data sinks are forbidden, i.e. *a data transformation must have both inputs and outputs*. Some or even most of these constraints could also have been specified in the graphical metamodel, at the cost of more cluttering the metamodel diagram. When developing a metamodel this is of course a tradeoff which must be evaluated on a case-per-case basis. The implementation of the well-formedness rules will be discussed in more detail in the next section.

3 Tool Support for the Metamodel

To really benefit of a SA/RT metamodel, tool support is required. We used the Software Modeling (SMW) toolkit [10] to automatically generate the metamodel, to create a SA/RT profile and, by using the scripting facilities of the tool, to provide automated querying and manipulation of SA/RT models.

3.1 The SMW Toolkit

SMW is built upon the OMG's MOF and UML standards, allowing editing, storage and manipulation of metamodels. The tool uses the Python language [11] to describe the elements of a model, each element being represented by a Python class. This fact provides the basic mechanism for accessing and executing queries over given models, as well as implementing transformations of the model elements in an OCL-like style. In addition, SMW allows the creation and usage of user defined profiles, based on the MOF standard.

3.2 SA/RT Metamodel Generation

When developing the metamodel we employed a graphical approach. The Python implementation of a specific metamodel can be generated simply by giving to the SMW metamodel generator the metamodel file as input as a UML class diagram saved in XMI [12] format. Therefore, by drawing the SA/RT metamodel class diagram directly into SMW we were able to automatically generate the metamodel.

The SMW toolkit enforces some constraints automatically (e.g., the multiplicity of the model elements) and adds checks for these in the metamodel file, but more specific constraints still have to be defined by hand in a separate file and supplied as input to the metamodel generator. The constraints were extracted from the informal specification of SA/RT and implemented as well-formedness rules coded directly in Python, using OCL-like constructs. The approach is similar to the way the UML metamodel is defined. Some examples of constraints, and their implementation in Python, are shown below. In total 20 such well-formedness rules were implemented to constrain the dataflow part of the SA/RT models. A complete list of these rules can be found in [9].

```
def wfrDataStore1(self):
 "A DataStore may only be connected to DataFlows"
 return(self.association.forAll(lambda ae:
    ae.association.oclIsKindOf(DataFlow)))

def wfrFlow1(self):
 "Flows may not have the same source and target elements"
 return(self.connection.forAll(lambda ae1,ae2:
    not(ae1.participant==ae2.participant)))
```

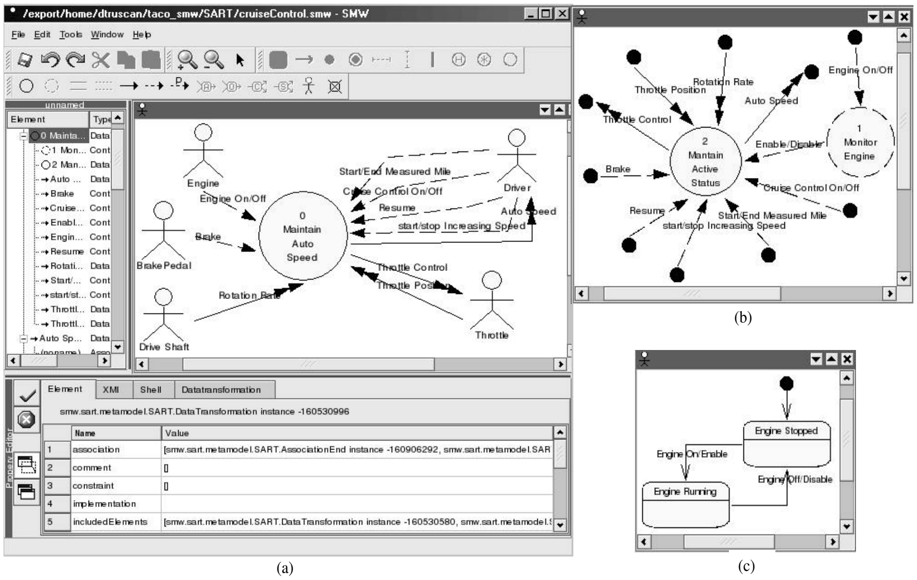
3.3 The SA/RT Profile

To be able to create, edit and manipulate models in a graphical environment, a SA/RT profile has been implemented in SMW. Two possible approaches existed to represent the graphical elements: to use the existing UML notations or to implement a new editor in SMW featuring the actual SA/RT notations. We followed the second approach. While some work was needed to implement the editor functionality for the SA/RT profile, the only change which had to be made to the metamodel was the inclusion, in the `Auxiliary` package of the metamodel, of a `PresentationElement` class describing the physical views of the `ModelElement`.

A screen shot of the resulting profile is presented in Fig. 3 (a). The system shown represents the top-level model (i.e. context diagram) of the Cruise Control system described by Ward and Mellor. The property editor for the currently selected element (i.e. `'0-Maintain-AutoSpeed'`) can be seen at the bottom of the screen. A refinement of the top-level diagram is presented in Fig. 3 (b), where the outer flows (from e.g., the context- or some other higher-level diagram) have been connected to the inner, refined transformations. The state machine corresponding to the `'1-MonitorEngine'` control transformation is presented in Fig. 3 (c).

One slight problem due to following the graphical construction approach was that although we used well-defined rules to specify those characteristics of the metamodel which are not defined graphically (i.e. in the XMI file), we still needed to implement those rules again in the SMW editor. For instance, although one of our well-formedness rules specifies that a `join` or a `split` must not be connected to both data and event flows, we still have to specify this rule explicitly in the part of the editor which handles connections between elements. That is, it is not sufficient to check if the well-formedness rule holds after the erroneous element already has been added to the diagram, but rather the user should be prevented to add the incorrect element at all.

Of course, it is also the case that some well-formedness rules cannot be applied to models which are under development. An example of this situation is the well-formedness rule stating that a non-abstract `DataTransformation`, i.e. a data transformation which is not refined by any other transformations, may only be connected to `DataFlows` and `PromptFlows`, not to regular `ControlFlows`. When designing a model it would be unpractical to apply this restriction, as we are likely to have some



Element | XMI | Shell | Datatransformation

smw.sart.metamodel.SART.DataTransformation instance -160530996

	Name	Value
1	association	[smw.sart.metamodel.SART.AssociationEnd instance -160906292, smw.sart.metamodel.SAR
2	comment	[]
3	constraint	[]
4	implementation	
5	includedElements	[smw.sart.metamodel.SART.DataTransformation instance -160530580, smw.sart.metamodel.S

Fig. 3. Top-level DFD (a), its refined DFD (b) and the state machine of the *Monitor Engine* control transformation (c)

unrefined `DataTransformations` in our unfinished model which later are going to be refined, and thus become abstract. Figure 3 (a) illustrates this case, as here we have not yet added any refining transformations for the `'0-MaintainAutoSpeed'` transformation, but we still have control flows connected to it. Applying the well-formednes rule while designing the model would mean that we need to define all transformations completely before we can add flows between them. In this case it is thus more reasonable to only check that the condition holds for the complete model. The SMW editor features a shell through which the model can be inspected and also modified using Python scripts. Executing one single command in the shell window, all well-formedness rules on the metamodel are recursively checked.

4 Model Manipulation

Using the scripting facilities of the SMW tool, Python scripts could be implemented to navigate, query and manipulate SA/RT models. Next we present couple of examples of such scripts. For instance, the following script returns the name of all data flows in a model that are inputs for a data transformation element with a given name. Here, the function `getAllParts()` recursively returns all the elements of a given model.

```
dfdModel.getAllParts().select(lambda df:
    df.oclIsKindof(DataFlow) and
    dfdModel.getAllParts().select(lambda dt:
        dt.oclIsKindOf(DataTransformation) and
        dt.name==someName and
        df.connection[1].participant==dt)).name
```

Similar scripts can be used to modify or transform the SA/RT models created. One such example can be, for instance, when two output data flows with identical names originate in the same data transformation and have distinct target data transformations (sink). In this case a fork (`clone`) flow is added to replace the two flows.

```
dfdModel.getAllParts().select(lambda source:
   source.oclIsKindOf(DataTransformation) and
   dfdModel.getAllParts().select(lambda firstFlow:
       firstFlow.oclIsKindOf(DataFlow) and
       firstFlow.connection[0].participant==source~and
       dfdModel.getAllParts().select(lambda secondFlow:
          secondFlow.oclIsKindOf(DataFlow) and
          secondFlow.connection[0].participant==source and
          firstFlow.name==secondFlow.name and
          addAClone(firstFlow, secondFlow, source))))
```

In the same way, model transformations could be implemented not only between models of the same metamodel, but also between models of different metamodels (e.g., SA/RT and UML). Following, a partial example of a script that transforms a class diagram (i.e. `umlModel`) into a DFD (i.e. `dfdModel`) is presented. In there, each `actor` of the UML model is transformed into an `external entity` and added to the SA/RT model (lines 1-3). Similarly, the ≪`control`≫ and ≪`interface`≫ classes are transformed into corresponding `data transformations` and added to the DFD model (lines 5-8).

```
0. for el in umlModel.ownedElement:
1.    if el.oclIsKindOf(UML.Actor):
2.       ee=SART.ExternalEntity(name=el.name)
3.       dfdModel.ownedElement.append(ee)
4.    if el.oclIsKindOf(UML.Class):
5.       if el.stereotype[0].name=="interface" or
6.         el.stereotype[0].name=="control":
7.             dt=SART.DataTransformation(name=el.name)
8.             topEl.ownedElement.append(dt)
```

More details on these model transformations and how we used them to support a model-driven process were given in [5].

5 Conclusions

We presented a MOF-based meta-model for SA/RT. We described how the metamodel was constructed starting from the SA/RT specification and how it was implemented in the SMW tool. Using the metamodel, we were able to provide tool support for our model-driven approach presented in [5]. It allowed us to create and manipulate system models using both the graphical, as well as the scripting facilities of the SMW tool. In the same time, using OCL to enforce constraints of the SA/RT metamodel, gave us the possibility to verify and ensure the consistency of the models created.

Having the MOF standard as a common meta-meta-model for both SA/RT and UML meta-models, enabled us to easily combine and integrate the object-oriented and structured methods to represent views of the system under design at different steps of the development process and at different levels of abstraction. Moreover, it made possible to graphically create and manipulate, simultaneously and inside the same development framework, UML and SA/RT models of the system under design. In addition, benefiting of the scripting facilities of the tool and of the OCL-like constructions in Python, scripts to support model transformations between SA/RT models or between SA/RT and UML models could be implemented. They provided us the basic means for automation, thus speeding up considerably the development time and reducing the error-prone sources during the transition from one step to another.

The implementation of the SA/RT metamodel proved to be an important opportunity in experiencing the concepts of metamodeling and model-driven development, and in the same time, it served as a case study for the SMW toolkit, enabling us to evaluate the features and detect the shortcomings of SMW as a metamodeling tool.

A similar approach can be followed up to design other MOF-based metamodels in order to provide tool support for already existing notations or to integrate UML (or SA/RT) with other/new metamodels. This would result in an enriched set of tools available for the system designer. For instance, to benefit of existing UML profiles (e.g., UML/RT profile) one can use the approach to provide new descriptive features into SA/RT, like concurrency and timing information.

References

1. OMG: (Model Driven Architecture (MDA)) Doc. ormsc/2001-07-01 At `www.omg.org`.
2. OMG: Unified Modeling Language Specification, ver. 1.5. (Doc. formal/2003-03-01)
3. OMG: Meta-Object Facility (MOF). (Doc. formal/01-11-02 At `www.omg.org`)
4. Fernandes, J.M., Lilius, J.: Functional and Object-Oriented Modeling of Embedded Software. In: Proceedings of the Intl. Conf. on the Engineering of Computer Based Systems (ECBS'04), Brno, Czech Rep. (2004)
5. Tool Support for DFD-UML Model-based Transformations. In: Proceedings of the Intl. Conf. on the Engineering of Computer Based Systems (ECBS'04), Brno, Czech Rep. (2004)
6. Baresi, L., Pezzè, M.: Toward Formalizing Structured Analysis. ACM Transactions on Software Engineering and Methodology **7** (1998) 80–107
7. Ward, P.T., Mellor, S.J.: Structured Development for Real-Time Systems. Prentice Hall/Yourdon Press (1985) Published in 3 volumes.
8. Hatley, D.J., Pirbhai, I.A.: Strategies for Real-Time System Specification. Dorset House Publishing Co., New York, USA (1987)
9. Isaksson, J., Truscan, D., Lilius, J.: A MOF-based Metamodel for SA/RT. Technical Report 555, Turku Centre for Computer Science (2003)
10. Porres, I.: A Toolkit for Manipulating UML Models. Software and Systems Modeling, Springer-Verlag **2** (2003) 262–277
11. (Python Programming Language) `http://www.python.org` .
12. OMG: XML Metadata Interchange (XMI) spec. (Doc. formal/00-11-02. At `www.omg.org`)