Advances in the automation of model driven software engineering for hard real-time systems with Ada and the UML Profile for MARTE

Julio L. Medina and Alejandro Pérez Ruiz

Departamento de Electrónica y Computadores, Universidad de Cantabria, 39005-Santander, SPAIN
{julio.medina, alejandro.perezruiz}@unican.es

Abstract

The traditional application of model based development techniques in the design of real-time systems comprises usually the generation of code from both structural and behavioral models. This work describes recent advances in a tool-aided methodology that enables the assembly and transformation of such design intended models into schedulability analysis models that match the corresponding automatically generated implementation code. Both, the analysis models and the code are generated by means of model transformations from the high-level architectural formalisms provided by the UML Profile for MARTE standard. As a novelty the Ada code generator uses not only the typical information provided in structural models, which is used to create the skeleton of the classes and procedures, but also its activities, whose scenario based behavioral information is used to fill the code inside the procedures and functions there contained. From the perspective of the real-time practitioner, the use of activities instead of state machines helps significantly to keep in tune the two fundamental views of the system: the implementation code, and its corresponding schedulability analysis model.

1. Introduction[1]

Model-based software development is progressively taking momentum in industry as one of the most promising software engineering approaches. It helps to create and keep assets of many kinds along the development process. It facilitates the separation of concerns, increasing the process efficiency, and finally empowering the quality of software.

For real-time applications, a model-based methodology can also help to simplify the process of building the temporal behavior analysis models. These models constitute the basis of the real-time design and the schedulability analysis validation processes. With that purpose, the designer must generate, in synchrony with the models used to generate the application's code, an additional parameterizable model, suitable for the timing validation of the system resulting out of the composition of its constituent parts. The analysis model for each part abstracts the timing behavior of all the actions it performs, and includes all the scheduling, synchronization and execution resources information that is necessary to predict the real-time qualities of the applications in which such part might be integrated. In the approach here presented, these analysis models are automatically derived from high level design models annotated with a minimum set of real-time features taken from the requirements of the application in which they are to be used. Following the generation of the application's code as a composition of the code of its constituent parts, the complete real-time analysis model of the application can also be automatically generated from the composition of the set of real-time sub-models that form it.

A discussion of such process used for the design of the real-time characteristics in a strict component-based development methodology may be found in [1].

The research effort that this paper presents considers the definition of schedulability analysis models as part of the chain of tools and techniques used in a model driven engineering approach. At this abstraction level, the concrete modeling paradigm used to conceive and elaborate the system is not specified, but for practical purposes we assume it is able to be expressed in UML[16]. This is a general purpose modeling language but we will use with it its standardized extensions for Modeling and Analysis of Real-Time and Embedded systems, namely the UML Profile for MARTE [13].

The very basic use of model based development techniques, not only in the design of real-time systems but in the software domain in general, comprises usually the generation of code from structural models like class diagrams. With those automations an initial set of skeletons of the classes and structural packages that form an application is usually easy to obtain. Also some form of reverse engineering is available through the usage of specially formatted "comments" placed as textual marks surrounding the space

1. This work has been funded by the European Union under contract FP7/NoE/214373 (ArtistDesign); and by the Spanish Government under grant TIN2011-28567-C03-02 (HI-PARTES). This work reflects only the author's views; the EU is not liable for any use that may be made of the information contained herein.

for the real code. The final implementation code is then inserted (usually typed by hand) between the marks managed by the code generators. A further refinement that generates both, specifications and bodies, in the modeling side, are code generators that use state machines for modeling the behavior of the classes. This mechanism uses the operations of a class as messages handlers that trigger the events between states. That way the messages from other objects can interact with the automaton of the class, though in a non-predictable order. Then, this kind of code generators is not consistent with the required wosrt case scenario-based description of activities used for schedulability analysis. For this reason a different approach to the code generation is necessary if we want to keep both models in tune.

The mechanism for code generation that may be used to fill the code inside the marks of the structural skeletons is the use of the behavioral models given for each operation of the class. These models are usually made for explanatory or documentation purposes, but they are well suited for specification. For this labor the proper modeling elements are activity diagrams. The formalization of the "code" inside actions may be either the standardized action language [17] of the OMG, or specific annotations in the target language with the actions to be performed.

This paper presents some advances in the methodology proposed and reports as a relevant contribution the definition and implementation of a new kind of code generator. It does not only generate the classical skeletons from UML classes and operations, but also fills the bodies of those operations with code generated from the interpretation of UML activities. The activities are graphically described using activity diagrams.

The paper is organized as follows: Section 2 presents a global view of the approach and situates the research efforts undertaken in its perspective. It also makes a brief summary of the challenges, and presents some related efforts as well as the basis of the modeling languages used for it. Section 3 summarizes the concrete rules for the code generation. It describes and identifies the intermediate formalisms in the modeling language for the generation of the implementation code and points out the technologies used for its automation. Section 4 presents a usage example that assesses the code generation features and illustrates the available results. Finally some conclusions and the definition of our next steps in the completion of the envisioned model driven engineering approach.

2. The approach

The approach that supports the efforts here described uses UML as modeling language. The UML standard extensions proposed by MARTE [13] for the modeling and analysis of real-time and embedded systems are used with it. It complements UML to enable the specification of the necessary real-time features in the models. A synthetic view of the approach is shown schematically in Figure 1.

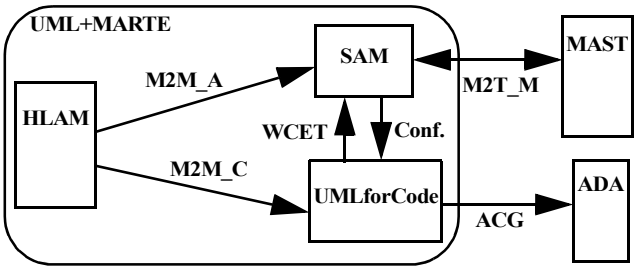

Figure 1. Models & transformations used in this approach

The initial model used to describe the application and its real-time features is constructed using the MARTE extensions for high level application modeling (HLAM). From this formalisms, two model-to-model (M2M) transformations are used. One, **M2M_A**, is used to create the UML representation of the analysis model. This transformation is used to create a model for each real-time situation under analysis together with the model of the processing resources, and the workload to consider. For this model the schedulability analysis modeling capabilities of MARTE (SAM) are used. The other, **M2M_C**, is used to generate an intermediate model useful for the code generation. In this methodology the target implementation language is Ada and the intermediate model, called **UMLforCode** in Figure 1, is a typical UML object oriented generic model that comprises structural as well as behavioral information. The behaviors of the operations in this model are expressed by means of activity diagrams.

The model-to-text transformation, denoted as **M2T_M** in Figure 1, is needed to generate the final schedulability analysis models in this approach, and it is part of our previous work [2]. An eclipse based tool [15] is provided for the generation of analysis models, the invocation of the analysis tools, and the retrieval of results back into the modeling analysis context. The tool then converts SAM models into the formalisms used by MAST [12] and then recovers its results back into the UML+MARTE model.

This paper presents the advances achieved in the techniques and tools used to generate the Ada implementation code from the UMLforCode object oriented generic model. This is a model-to-text transformation, called **ACG** (standing for Ada Code Generation) in Figure 1. The code implemented out of the combination of M2M_C and ACG is consistent from the execution semantics point of view with the analysis models generated out of the combination of M2M_A and M2T_M. Instrumented versions of the code will serve to meassure actual execution times (**WCET**) for

the SAM model. Once the analysis is performed, scheduling results are back annotated to the SAM models. These real-time configuration data include priorities (or relative deadlines) for the concurrent units, and priority ceilings (or preemption levels) for shared resources. Then, these data, denoted as **Conf.** in Figure 1, is used as part of the configuration information in the UMLforCode generation model.

2.1. The need for a new code generation technique

Following previous efforts that have studied the design of real-time systems using object oriented formalisms, we observe that most of them include the specification of the concurrency using structural models, usually at the design-for-implementation level. These dual structural-behavioral formalisms are made in the aim that this will help to realize schedulability analysis with the simple tasking model in mind and basic RMA techniques later on. Unfortunatly the complexity of the mechanisms used to generate the code makes this assumption not realistic, such as in ROOM [3] [4], Octopus/UML [8], ACCORD/UML [10] [11], Comet [7], or the design model extremely constrained and monolithic such as in HRT-HOOD [5], OO-HARTS [6].

Being a syncretism of all those mentioned, and in order to ease the application of simple schedulability analysis techniques, the high level application modeling constructs in MARTE (see its HLAM section in [13]) also facilitate the use of structural models for the specification of the concurrency. But the interactions between them (including distribution) may take complex patterns that require a richer model for the analysis. The offset based analysis techniques scale better to deal with this scenarios than the basic tasking model. HLAM proposes two basic building blocks, the real-time unit: *RtUnit* and the passive protected unit: *PpUnit*. As for the behaviors in them (the code inside the marks), due to its natural complexity it is usually not just passive linear code that can be modeled as a computation time; instead they include delays, and interactions among objects and nodes, mostly when they become formed out of a composition of distributed operations (behavioral models). In these cases a state machine is not directly transformable into an analysis model.

From the analysis perspective, the models that are required to apply the modern offset based analysis techniques, are fundamentally scenarios. A scenario is an expression of the (worst case) expected or observable manifestation of the design intents (coded behaviors). This is the basis for coping with complexity that distinguishes RMA schedulability analysis techniques from those other strategies based on timed automata or synchronous languages.

As a modeling language for this domain, the scheduling analysis modeling section of MARTE (SAM) is also able to express that kind of scenario models, and then it is an adequate formalism to feed the corresponding analysis tools. Unfortunately these scenarios are not necessarily part of the initial specification of the system behavior. They are a means to express: the expected stimuli, the high level expected workload, and the end-to-end timing requirements, but they are usually not the basic data used for design intent or code generation drawn by the designers.

The creation of these (usually worst case) analysis oriented scenarios in tune with the final code is actually the main duty and a high responsibility of the real-time practitioner. In order to help in this labor the automation tools need the model used for code generation to have the behaviors of its operations expressed as scenarios. For this reason the adequate input models for the generation of the code inside the operations in the UMLforCode model are UML activities. Then the tool that fills the code for the procedures and functions associated to the classes retrieves it from activity diagrams.

The use of scenarios has an additional benefit. This method helps to support the design of applications in terms of composable parts, which are closer in granularity to the concept of real-time objects than to the fully CBSE interpretation of components. In a fully component-based approach, the creation of the analysis models would have to be made as a combination of both, structural elements plus their deployment. In a model-driven approach, this later strong form of composability is in a higher level of abstraction, but still may benefit of the approach here described in order to assess a variety of non-functional properties, in our case of course the assesment of its timing properties by means of schedulability analysis.

3. The UMLforCode (meta)model

The purpose of having this intermediate model is basically to have a UML object oriented representation of the system that allows us to have the behaviors expressed in a way as close as possible to the way it is expressed its schedulability analysis model. Also this model must serve to implement the system in potentially different target programming languages. As a starting point for its practical implementation we have considered Ada as the target language.

In this section we describe the elements of UML that have been selected for the creation of these models, and the way they are used to generate Ada code. Instead of using a full metamodel, or a reduced version of the UML metamodel to formalize this description, we prefer to present it by identifying the capacities of the object oriented modeling/programming that are supported.

The technologies used for this automation are those provided by PapyrusUML as graphical tool, the UML2 plug-in

as model repository, and the Acceleo plug-in for the extraction of text from the UML2 models. As in marte2mast (M2T_M) [15], also here a number of Java functions have been necessary to implement the code generation.

3.1. Structural elements

The structural object oriented elements currently supported are Classes, Packages, and Interfaces. They are modeled in Class Diagrams.

- Packages may contain classes and have dependencies among them. Dependencies are implemented by means of **`with`** clauses between the ada **`package`** construct.
- Classes are the basic building blocks of code in an object oriented language. In Ada they are implemented by means of what Ada calls **`tagged types`**. These constructs support the inheritance and polymorphism, and hold in a natural way the UML concepts of object property (attribute) and operation (method). Static attributes and methods are declared out of the tagged type, so to keep them together they need to be hold by a wrapping Ada package in which the tagged type is also defined. This mechanism allows us to implement in Ada also the dependencies between classes and the visibility (accessibility) restrictions of the properties and operations. The inheritance and the realization of interfaces are implemented natively by Ada in the tagged types definition. Figure 2 shows how the Ada wrapping package visibility scopes match the visibility of the class members.

Figure 2. Class members visibility in the wrapping package

- Interfaces are directly implemented by using the corresponding Ada concept, which supports the definition of object methods. Static methods and attributes (including constants) are implemented like in classes by generating the corresponding code in the wrapping package. Interface object attributes are not supported.

Next we present some limitations of the tool, and modeling constraints for UMLforCode models. The tool is able to detect them and warn the user about their occurrence:

1. For members of a Class (properties and operations) the visibility clause `package` will not be enforced by the Ada language. They will be public and consistently renamed with the prefix **`package_`** .
2. Other visibility clauses (`public`, `protected`, and `private`) are supported as indicated in Figure 2.
3. Attributes need to have a name and a type.
4. Operations need to have a name and a type. Also each parameter needs: a name, the direction of assignment (**`in`**, **`out`**, or **`inout`**), and a type.
5. Classes and Interfaces need to have `public` visibility.
6. Nested classes are not supported.
7. Multiple inheritance is not supported in Ada. Interfaces realization is suggested to overcome this issue.

In order to handle inheritance, classes contained inside packages, and do so respecting the visibility defined by the modeler, three possible solutions where studied: (a) use the containing package directly as the wrapper, (b) use the class wrapping package as a child package of the container, (c) use the containing-contained relationship only as a mechanism to define the name of the wrapping packages for the class. The chosen solution was (c).

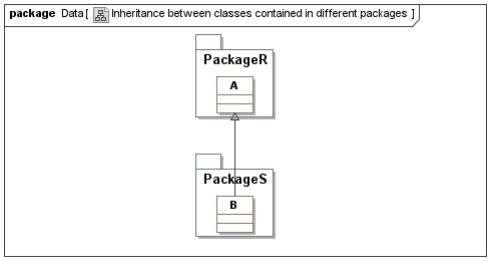

Figure 3. Inheritance among classes in different packages

To see this, consider the example in Figure 3. The wrapping package for class B will be denominated PackageS_B, correspondingly, the wrapping package for class A will be denominated PackageR_A. The fully qualified name of class B is PackageS_B.B and inherits from PackageR_A.A.

3.2. Behavioral models

Following the structure of SAM models described in previous research efforts [2] [14], MARTE provides concepts to organize the analysis models using three main categories: the platform resources, the elements describing the logical behavior of the system constituent parts, and finally the real-time situations (scenarios) to be analysed.

Scenarios are expressed usually by the annotation of SaSteps (SaCommStep, ResourceUsage or GaScenario) in sequence charts or activity diagrams. In marte2mast [15], scenarios may also be constructed from the lists of steps that are implicit in the chain of internal sub steps of a SaStep. These are expressed using the sub-usages list, hence using a structural element of the MARTE profile. This helps the tools to extract the analysis model in a more efficient way. But to express the high level end-to-end flows scenarios, sequence charts or activity diagrams are used instead.

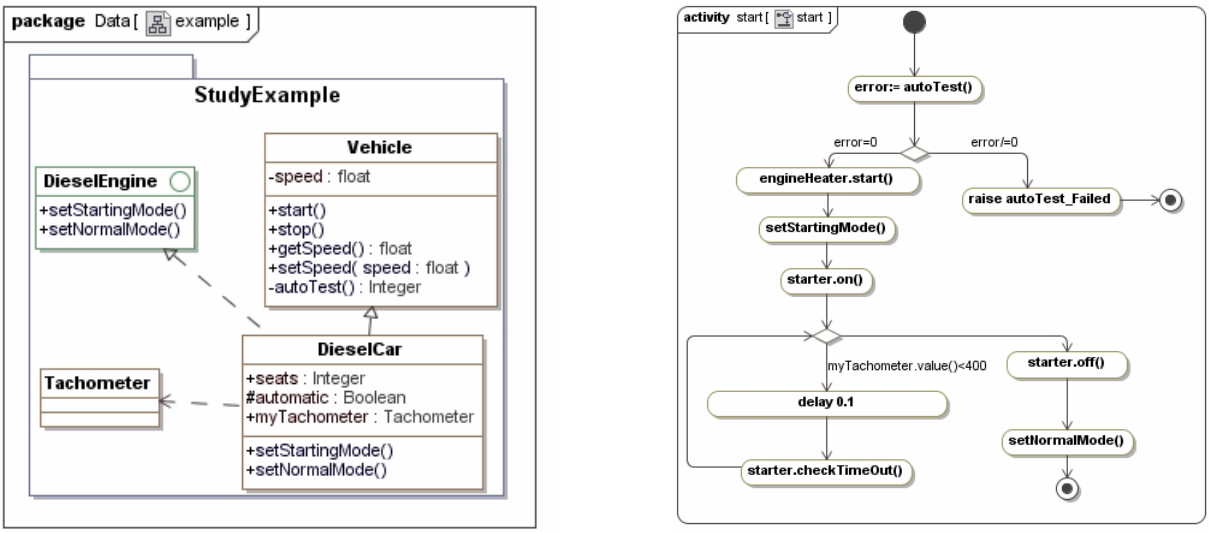

Figure 4. Structural and behavioral models used in the study example.

The elements that are currently used for the generation of the code inside class operations (bodies of the methods) are activities described by means of activity diagrams. The concrete modeling elements used in the diagrams are:

Initial nodes / Control Flow /Guards / Decision nodes / Opaque actions / Final nodes

These elements may be considered intuitively as corresponding to the basic assembly instructions for an Eckert-Mauchlyin architecture (also called Von Neuman architecture). With them the tool is able to extract out of the diagrams: regular **sentences**, **invocations,** simple **while loops** as well as **if-then-elseif-else** conditional branches (see the example in Figure 4). These basic constructs are the minimum required to describe scenarios, but they are sufficient for the general case; specially considering that, in the scope of the full approach, UMLforCode models are to be automatically generated by M2M_C from HLAM structural and behavioral models.

From its implementation point of view, in order to make code from activities, ACG has required much more than the basic automations provided by Acceleo. In particular due to the difficulties to handle variables inside the Acceleo scripts. The use of Java code inside the scripts, and the UML2 Java library created by the eclipse community, have been necessary to overcome this problem.

4. Study example

In order to show the capabilities of this tool we propose as an example the structural and behavioral models shown in Figure 4. They represent a very simplified extract of a car class and the activity model of an operation to start it.

Two pieces of code extracted from that model are shown next: the specification, and an extract of the implementation body of the `DieselCar` class. The implementation body shows the code for the function `start`.

The obtained Ada specification is:

```
with StudyExample_Vehicle.Vehicle;
with StudyExample_Tachometer.Tachometer;
package StudyExample_DieselCar is

    type Public_Part is abstract  new StudyExample_Vehicle.Vehicle
    and StudyExample_DieselEngine.DieselEngine  with  record
            seats : Integer;
            myTachometer : StudyExample_Tachometer.Tachometer;
    end record;

    -- Complete_type
    type DieselCar is new Public_Part with private;

    -- Public methods:
    procedure startIgnition (Self : DieselCar'Class );
    procedure setNormalMode (Self : DieselCar'Class);
    procedure setStartingMode (Self : DieselCar'Class);
    overriding procedure start (Self : DieselCar'Class);
    overriding procedure stop (Self : DieselCar'Class);

private
     -- Protected attributes
    type DieselCar is new Public_Part with record
            automatic : Boolean;
    end record;

end StudyExample_DieselCar;
```

The Ada body obtained for the start function inside DieselCar Class is:

```
package body  StudyExample_DieselCar is

    --Methods

    overriding procedure start (Self : DieselCar'Class) is
    begin
         error:=autoTest();
         if error/=0 then
            raise Autotest_Failed;
         elsif(error=0) then
            engineHeater.start();
            setStartingMode();
            starter.On();
            while myTachometer.value()<400 loop
                delay(0.1);
                starter.checkTimeOut();
            end loop;
            starter.off();
            setNormalMode();
         end if;
```

```
    end start;
    --Methods

    procedure setStartingMode (Self : DieselCar'Class) is
    begin
         --  (AP) Generated: replace with real body!
        pragma   Compile_Time_Warning   (True,   "setStartingMode
unimplemented");
        raise Program_Error;
        return setStartingMode (Self);
    end startIgnition;

    ...

end StudyExample_DieselCar;
```

5. Conclusions and future work

This work has presented the recent advances in a tool-aided methodology that enables the assembly and transformation of high level design intended UML models into schedulability analysis models that match the corresponding automatically generated implementation code. Both, the analysis models and the code are generated by means of model transformation from the high-level architectural formalisms provided by the UML Profile for MARTE standard. As a novelty this paper presents a new kind of Ada code generator that generates not only the skeleton of the classes, but also the code inside the procedures and functions there contained. It uses activity diagrams to fill them.

The necessity of this way of generating code lays in the fact that the UML+MARTE schedulability analysis specific models are described by means of scenarios. The creation of this (usually worst case) analysis oriented scenarios is actually the main duty of the real-time practitioner. Then, in order to automate the consistency between the code structure and the analysis model, both need to be expressed as scenarios, instead of state machines behaviors. From the real-time and embedded systems research community perspective, this effort constitutes another step to get the effective exploitation of the capabilities of the available analysis and verification techniques, which despite the efforts in dissemination, have not yet reached an audience large enough to reward the many years of work in the field. The modelling strategy and tools proposed in this work are another step in this direction.

References

[1] López P., Drake J.M., and Medina J.L., Enabling Model-Driven Schedulability Analysis in the Development of Distributed Component-Based Real-Time Applications. In Proceedings of 35th Euromicro Conference on Software Engineering and Advanced Applications, Component-based Software Engineering Track, Patras, Greece, August 2009, IEEE, ISBN 978-0-7695-3784-9, pp. 109-112.

[2] J. Medina and A. Garcia Cuesta. Model-Based Analysis and Design of Real-Time Distributed Systems with Ada and the UML Profile for MARTE. In Proc. of the 16th International Conference on Reliable Software Technologies-AdaEurope 2011, LNCS 6652, pp 89-102, ISSN 0302-9743

[3] Bran Selic, Garth Gullekson, and Paul T. Ward. Real-time Object Oriented Modeling. ISBN 0-471-59917-4, John Wiley & Sons, Inc., USA, 1994

[4] Bran Selic and Jim Rumbaugh. Using UML for Modeling Complex Real-Time Systems. Rational white papers, http://www.rational.com/products/whitepapers/UML-rt.pdf, March 1998

[5] Alan Burns, Andy Wellings. HRT-HOOD, a structured design method for hard real-time ADA systems. ISBN 0 444 82164 3. Elsevier, Amsterdam, 1995

[6] Mazzini S., D'Alessandro M., Di Natale M., Domenici A., Lipari G. and Vardanega T. HRT-UML: taking HRT-HOOD into UML. In Proceedings of 8th Conference on Reliable Software Technologies Ada Europe, 2003

[7] Hassan Gomaa. Designing Concurrent, Distributed and Real-Time Aplications with UML. ISBN 0-201-65793-7, Addison-Wesley, USA, 2000

[8] E. Domiczi, R. Farfarakis and J. Ziegler. Octopus Supplement Volume 1. Nokia Research Center. http://www-nrc.nokia.com/octopus/supplement/index.html, 1999

[9] Laila Kabous. An Object Oriented Design Methodology for Hard Real Time Systems: The OOHARTS Approach. Doctoral Theses, School Carl von Ossietzky, Universität Oldenburg. 2002

[10] F. Terrier, G. Fouquier, D. Bras, L. Rioux, P. Vanuxeem and A. Lanusse. A Real Time Object Model. Presented in TOOLS Europe'96. Paris, France. Prentice Hall, 1996

[11] A. Lanusse, S. Gerard and F. Terrier. Real-Time Modeling with UML: The ACCORD Approach. In Selected papers from the First International Workshop on The Unified Modeling Language "UML"'98: Beyond the Notation. Mulhouse, France, June 3-4, 1998. Pp. 319-335. ISBN:3-540-66252-9. Springer-Verlag London, UK 1998.

[12] M. González Harbour, J.J. Gutiérrez, J.C.Palencia and J.M.Drake, MAST: Modeling and Analysis Suite for Real-Time Applications, in Proc. of the *Euromicro Conference on Real-Time Systems*, June 2001.

[13] Object Management Group, UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, version 1.1, OMG doc. formal/2011-06-02, 2011.

[14] J.L.Medina, M.González Harbour and J.M. Drake, Mast Real-Time: A Graphic UML Tool for Modeling Object-Oriented Real-Time Systems, in Proc of the *22nd IEEE Real-Time System Symposium (RTSS 2001)*, pp 245-256, 2001.

[15] http://mast.unican.es/umlmast/marte2mast

[16] Object Management Group. Unified Modeling Language version 2.4.1, OMG document formal/2011-08-06, 2011

[17] Object Management Group. Action Language for Foundational UML (Alf), Concrete Syntax for a UML Action Language. OMG document ptc/2010-10-05, 2010.