

ITEA

INFORMATION TECHNOLOGY

FOR EUROPEAN ADVANCEMENT

MARTES

Model-based Approach for Real-Time Embedded Systems development

Specification of the Models (PIM, PSM) in the MARTES Methodology

Deliverable ID:	1.5
Version:	1.0
Date:	05/09/07
Editor:	Thierry Saunier
Status:	Final
Confidentiality:	Public

This document is part of the work of the EUREKA Σ! 2023 – ITEA 04006 project MARTES.
Copyright © 2005-2006-2007 MARTES consortium.

Authors:

Author	Partner	Email
Michel Bareteau	Thales TRT	michel.barreteau@thalesgroup.com
Erwin De Kock	Philips	Erwin.de.kock@philips.com
Paul Feautrier	Inria	Paul.Feautrier@ens-lyon.fr
Marko Hännikäinen	TUT	marko.hannikainen@tut.fi
Erno Salminen	TUT	erno.salminen@tut.fi
Klaus Kronlöf	Nokia	klaus.kronlof@nokia.com
Fernando López	Telefonica I+D	fla@tid.es
Vincent Perrier	CoFluent	vincent.perrier@cofluentdesign.com
Pablo Sanchez	UC	sanchez@teisa.unican.es
Thierry Saunier	Thales DLJ	thierry.saunier@fr.thalesfgroup.com
André Turostowski	Thales DLJ	Andre.turostowski@fr.thalesgroup.com
Kari Tiensyrjä	VTT	kari.tiensyrja@vtt.fi
Jari Kreku	VTT	jari.kreku@vtt.fi
Stefan Van Baelen	K.U.Leuven	Stefan.VanBaelen@cs.kuleuven.be

Document History:

Date	Version	Editor	Description
12/02/07	0.1	Thierry Saunier	Initial outline based on D1.2 structure extended to focus on the domain part expressed with the MARTES meta-model and its UML description expressed with the MARTES UML profile (Chapter 3)
26/02/07	0.2_VTT	Kari Tiensyrjä Jari Kreku	VTT D1.5 contribution added in sub-section 3.3.3
23/03/07	0.2_TID	Fernando López	Inputs and some corrections from TI+D
28/03/07	0.3	Thierry Saunier	French consortium contribution added in section 3 and integration of VTT and TI+D contributions
25/05/07	0.4	Erno Salminen	Description of the MARTES profile.
12/06/06	0.5	Andre Turostowski	Integration of revised contributions.
20/06/07	0.6	Thierry Saunier	Integration of revised contributions
31/08/07	0.7	Stefan Van Baelen	Review of the deliverable
05/09/08	1.0	Andre Turostowski	Final issue, reviewed and corrected

Filename: MARTES_D1.5_v1.0.doc

© Copyright MARTES Consortium

TABLE OF CONTENTS

Executive Summary **8**
1. Introduction **9**
2. MARTES Co-Modeling Approach **10**
2.1 Requirement and Specification Modeling 12
2.1.1 Functional Requirements 12
2.1.2 Non-Functional Requirements and User Interface Requirements 13
2.1.3 General End-User Requirements and Business Requirements 14
2.1.4 Executable Use Case Models 14
2.1.5 Abstract Use Case 14
2.1.6 Refined Use Case 15
2.2 Application Modeling 15
2.2.1 Functional Modeling 16
2.2.2 Execution Modeling 18
2.2.3 Workload Modeling 25
2.3 Execution Platform Modeling 27
2.4 Allocation Modeling 28
2.5 Implementation Oriented Modeling 29
3. Models in the MARTES Methodology **31**
3.1 Design Rationale 31
3.1.1 Model Components 31
3.1.2 Run-time Components 33
3.1.3 How the Pieces Fit Together 33
3.2 MARTES Domain Specific Model 34
3.2.1 Domain View 34
3.2.2 Application Model 35
3.2.3 Execution Platform Model 40
3.2.4 Allocation Model 41
3.2.5 Allocated Model 43
3.2.6 Implementation Oriented Model 43
3.3 UML Profile for MARTES 44
3.3.1 Overview of the MARTES Profile 45
3.3.2 Stereotypes of the UML MARTES Profile 48
3.3.3 Performance Modeling and Evaluation 64
4. Conclusion **76**
5. Terms and Definitions **77**

© Copyright MARTES Consortium

6. Acronyms 80
7. References 83

© Copyright MARTES Consortium

LIST OF FIGURES

Figure 1: Co-modeling artifacts and phases. 11
Figure 2: Logical model for execution modeling. 18
Figure 3: Primitives of the specification. 21
Figure 4: Single specification and execution framework. 22
Figure 5: Border channels and border processes. 24
Figure 6: Example of using state machine diagrams in workload modeling. 27
Figure 7: MARTES models and transformations. 32
Figure 8: MARTES domain specific application model. 35
Figure 9: MARTES domain specific application behavioral model. 37
Figure 10: MARTES domain specific expression model. 39
Figure 11: MARTES domain specific execution platform model. 40
Figure 12: MARTES domain specific allocation model. 42
Figure 13: OMG RTES related standards. 44
Figure 14: Structure of the MARTES profile. 45
Figure 15: Example of using the MARTES profile. 47
Figure 16: Main stereotypes of the MARTES profile. 47
Figure 17: Example top-level use case requirements model. 49
Figure 18: Stereotypes for service-oriented modeling. 49
Figure 19: Example service specification model. 50
Figure 20: Example of modeling service activities. 51
Figure 21: Stereotypes for application modeling. 52
Figure 22: Example of structural application modeling using a composite diagram. 53
Figure 23: Example of behavioral application modeling with statecharts. 54
Figure 24: Stereotypes for HW platform modeling. 56
Figure 25: Example for HW platform modeling. 56
Figure 26: Stereotypes for SW platform modeling. 57
Figure 27: Example of SW platform modeling. 58
Figure 28: Stereotypes for grouping and allocation modeling. 61
Figure 29: Stereotypes for memory allocation modeling. 62
Figure 30: Example of allocation model. 63
Figure 31: Example of memory allocation model. 63
Figure 32: Stereotypes constraint definition. 64
Figure 33: MARTES performance modeling and evaluation approach. 65
Figure 34: Overview of model structures for performance modeling and evaluation. 66
Figure 35: Workload meta-model. 67
Figure 36: ExecutionPlatform meta-model 69

© Copyright MARTES Consortium

Figure 37: Component and master parameters. 70
Figure 38: Image subsystem..... 71
Figure 39: Execution platform model and router structure. 73

LIST OF TABLES

Table 1: Descriptions for application stereotypes. 37
Table 2: Descriptions for behavioral stereotypes. 38
Table 3: Descriptions for expression stereotypes. 39
Table 4: Descriptions for execution platform stereotypes. 41
Table 5: Descriptions for allocation stereotypes. 43
Table 6: Descriptions for service oriented system modeling stereotypes. 52
Table 7: Descriptions for application stereotypes. 55
Table 8: Descriptions for execution platform stereotypes. 59
Table 9: Descriptions for allocation stereotypes. 62
Table 10: Workload stereotypes. 68
Table 11: Interface for transferring load information to the execution platform model. 69
Table 12: Component layer attributes. 71
Table 13: Services of the image subsystem. 72
Table 14: Network communication attributes. 72
Table 15: Platform services. 73
Table 16: Performance simulation attributes. 75

© Copyright MARTES Consortium

Executive Summary

This document constitutes the MARTES project deliverable D1.5 “Specification of the models (PIM, PSM)” final document.

The objective is to define the notation requirements, describing the abstraction levels and models views, based on the description of generic behaviors, and the definition of a formalism for the development of Real Time Embedded Systems, according the different activities of the MDA MARTES co-modeling approach.

Chapter 2 defines the co-modeling approach. The emphasis lies on the raising of the modeling abstraction of both the application development and platform development, and on making them interoperable in order to enable efficient performance validation and evaluation at system level. The approach aims to improve design productivity by reducing complexity, to increase predictability by allowing early validation of performance, and to optimize costs and risks by providing justified information for decision making.

Chapter 3 specifies the models in the MARTES methodology focusing on the architecture and analysis & design phase. End-user centric functional and non-functional requirements are specified through different models of computation and UML encapsulations.

The MARTES co-modeling approach follows the specify-explore-refine paradigm applying the principles of the so called Y-chart model, i.e., by mapping a model of an application onto a model of a platform, and by executing the analysis of the resulting allocated model. An application is modeled at different abstraction layers, with result models corresponding to the views needed at the co-modeling phases. A platform consists of hardware, operating systems, compilers etc., abstracted at different layers that hide the unnecessary details of the underlying implementation and yet carry enough information about the layers below to prevent design iterations.

The MARTES co-modeling approach covers the following activities:

- Requirement and Specification Modeling
- Application Modeling
- Execution Platform Modeling
- Allocation Modeling
- Implementation Oriented Modeling

The models and associated mappings specified according the MDA co-modeling approach will serve as a basis for the specification of the model transformations, which will be presented in the MARTES project deliverable “D1.6 Specification of the Model Transformations (PIM, PSM)”.

© Copyright MARTES Consortium

1. Introduction

The goal of the MARTES co-modeling approach is to close the methodological and tool gap between the application functionality (UML2) and the platform architecture (SystemC2) design domains.

The purpose of this deliverable is to specify the MARTES co-modeling approach expressed via dedicated PIM/PSM Models, focusing on the architecture and analysis & design phase. These specifications lead to the definition of a new UML2 profile, and guide the convergence of various existing system-centered technologies into an integration framework.

Meta-models of basic physical architecture elements (non-functional characteristics, topological communications and dimensions at chipset, board and product levels), and underlying real-time characteristics (performance modeling, processing requirements, data streaming profiles, Real-Time Operating Systems - RTOSs) are essential to model large-scale distributed embedded systems based on heterogeneous multiprocessor architectures. We therefore specify the communication channels for data streaming applications, IP models and encapsulation abstractions (hardware, software and performance) to enable a UML2 and equivalent C, C++, Java, Ada, SystemC, or VHDL encapsulation. Those models and stereotypes will serve as the description infrastructure to build HW/SW interface of our Platform-Specific Models (PSMs) suited for each architectural element with an effective implementation, or according to legacy or measurements on existing targets.

The challenge of the MARTES co-modeling approach is to provide a common way of modeling both hardware and software aspects of a RTES in order to

- Improve communication between developers
- Enable interoperability between development tools used for specification, design, verification, code generation, etc.
- Foster the construction of models that may be used to make quantitative and partitioning predictions regarding hardware and software characteristics.

© Copyright MARTES Consortium

2. MARTES Co-Modeling Approach

The MARTES co-modeling approach aims at defining a model-based approach for system-level design of real-time embedded systems. The emphasis is on the raising of the modeling abstraction of both the application development and platform development, and on making them interoperable in order to enable efficient performance validation and evaluation at system level. The approach aims to improve design productivity by reducing complexity, to increase predictability by allowing early validation of performance, and to optimize costs and risks by providing justified information for decision-making.

The MARTES co-modeling approach will provide interoperability of the modeling methodologies, languages and tools [i] that address, on one hand, the application development domain, and, on the other hand, the platform development domain at different levels of abstraction:

- Languages and concepts provide means for modeling and expressing model artifacts.
- Methodology, as a combination of methods and design processes, define how to exploit the languages, i.e., which languages to use at which level of abstraction, how levels of abstraction relate to each other, and what are the semantic transformations needed in order to move between abstractions.
- The interoperability of the modeling methodologies is addressed by applying the principles of OMG's Model Driven Architecture (MDA) [ii] that provides a methodological structure.
- In addition to using models instead of traditional implementation descriptions in the development, the important points are the separation of the concepts of the platform independent models (PIMs) of the application functionality and the platform specific models (PSMs) that adjoins to the models the information on how the application functionality will use a specific platform. The platform model represents the different kinds of parts that make up the platform, and the services provided by the platform.

The domain of interest of the MARTES co-modeling approach is the Real-Time Embedded Systems (RTES) domain. No commonly accepted definition of both real-time and embedded terms exists, but the characteristics of the RTES domain can be outlined as follows [iii]:

- Embedded systems are generally defined as interconnected computer based devices that interact with their environment. Embedded systems design requires optimization due to constrained resources. Typical characteristics are heterogeneity, real-time, power consumption, reactivity and criticality. Moreover, they are not user-programmable in a same sense as general-purpose computers, such as PCs.
- Classes of RTES include reactive systems that control and supervise their environment, command & control systems that manage the execution of processes or objects in the physical world, and intensive data computation systems that are typical in signal processing, image processing and mobile devices, where real-time processing often requires parallel computation.

Models are the primary artifacts in a model-based development. Different models are created through model transformations (either manual or automatic) at different phases of development for different purposes. This is coarsely depicted in Figure 1.

© Copyright MARTES Consortium

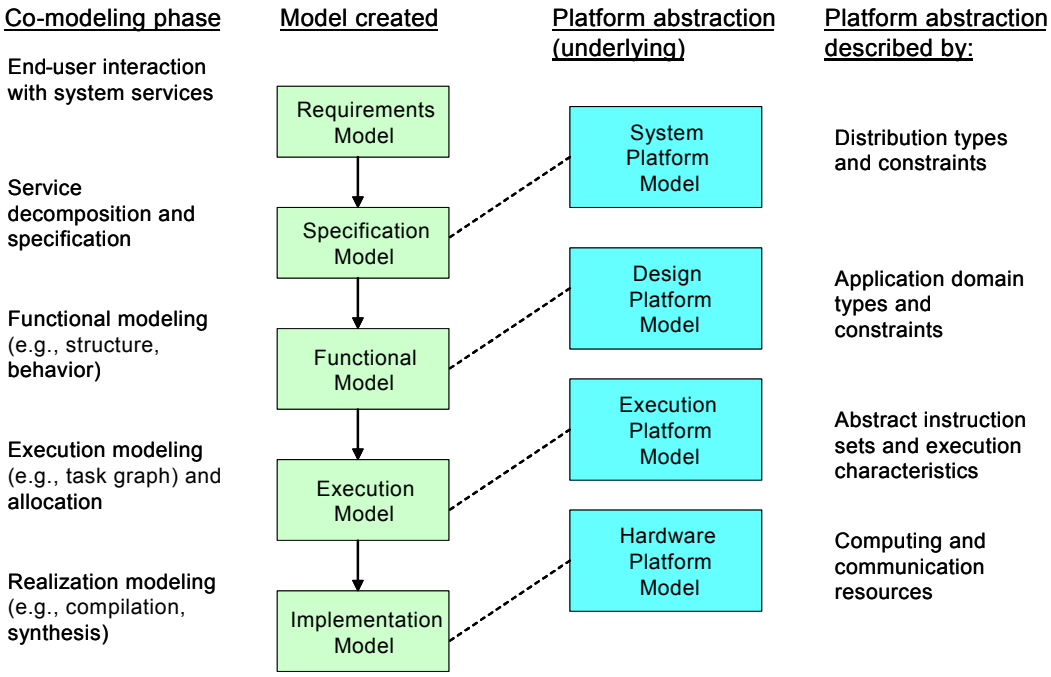


Figure 1: Co-modeling artifacts and phases.

Two main types of model transformations [i] can be distinguished:

- Refinement transformations change the level of model abstraction (column “Model created” in Figure 1). In other words, more details are added to the models when transferring to a lower level of model abstraction, and there is an obvious need to verify the correctness of the resulting model.
- Analysis transformations do not change the level of model abstraction, but are created for model analysis to provide designers information (predictions, estimates, etc.) about the characteristics of the RTES for decision making.

The MARTES co-modeling approach follows the specify-explore-refine paradigm applying the principles of the so called Y-chart model [iv], i.e., by allocating a model of application onto a model of platform and by executing the analysis of the resulting allocated model. The goal is to separate the application development from the platform development:

- An application is modeled at different abstraction layers with result models corresponding to the views needed at the co-modeling phases.
- A platform consists of hardware, operating systems, compilers, etc., abstracted at different layers that hide the unnecessary details of the underlying implementation and yet carry enough information about the layers below to prevent design iterations.

In order to emphasize explicitly platform independence, a concept of abstract (ideal) platform was proposed by Almeida [v]. The motivation for this is that by making the platform explicit at each level of abstraction, platform independence becomes a visible attribute of the model. If not made explicit, the platform will anyhow exist implicitly because of the choice of design concepts and collaboration rules. For example, the selection of a modeling language, e.g., UML, actually means a selection of certain platform-

© Copyright MARTES Consortium

related properties due to request-response and event based collaboration mechanisms. Furthermore, the approach would allow balancing of platform independence with platform-specific realization. Similar ideas were presented by Selic [vi], which also proposed hierarchical layering of platforms characterized by the services and Quality-of-Service (QoS) attributes that each platform layer provides.

2.1 Requirement and Specification Modeling

In this section, we have taken a use case driven approach to requirements modeling. Generally speaking, a use case captures a contract between the stakeholders of a system about its behavior. It describes the system behavior under various conditions as it responds to a request from one of its stakeholders, called the primary actor. The primary actor initiates an interaction with the system to accomplish some goal.

Classical use case descriptions are textual, although they can be written using graphical forms (sequence charts, flow charts, Petri nets, and so on). We believe that in some cases graphical forms, such as sequence diagrams, are a more natural form to describe a use case, because it is not limited to the language, and people can understand better graphical components that textual description. We see that textual and graphical forms are just two alternative presentations for the same thing. Ideally, a tool environment should support different presentation forms of a coherent underlying use case model.

2.1.1 Functional Requirements

Classical use case approaches are quite developed in expressing the functional aspects, so we adopt these concepts directly. Basically, we define different kinds of actors and the actions they take in the use case.

- In the textual form we first define the actors (primary actor, other external actors, subject), and then describe the actions and their ordering in the main success scenario. Actions are written in active form so that the initiating actor that takes the action, and the possible receiving actors, are explicitly named.
- In the graphical sequence diagram, vertical lines represent form actors and horizontal arrows, which extend from the initiating actor to the receiving actor, represent actions. The ordering of actions is from top to bottom. More complex ordering (concurrency, alternative) can be expressed with special notations.

Hierarchy is a useful way to hide details in a complex use case. Hierarchy can be used in both textual and graphical forms. In the action dimension, we can define composite actions that represent a sequence (or other ordering) of lower level actions. In the actor dimension, we can define composite actors in basically the same way. However, representing actor hierarchy in textual form is non-trivial, while in graphical form it is straightforward. Note that composite actions can be applied recursively to create a hierarchy of arbitrary depth.

Classically, the actions follow each other sequentially in the main success scenario. This is fine for simple use cases, but sequential definition becomes very complicated or almost impossible when we need to combine loosely coupled (almost independent and concurrent) functionalities in the same use case. In the context of use cases, concurrency must be interpreted as a requirement (or lack of thereof) imposed on the system. In our method the semantics of concurrency is defined as follows:

- Concurrent actions (input actions) initiated by the user (primary actor or any other actor) can be taken in any order and the receiving actor (typically subject) must accommodate that. In other words, the subject does not have any control of the ordering of input actions.

© Copyright MARTES Consortium

- Concurrent actions initiated by the subject (output actions) have no ordering constraints (but can have real-time constraints). In other words, the designer can choose an ordering that is easiest to implement.

The semantics of two composite actions, i.e., actions hierarchically defined by lower level actions, is that of interleaving. The ordering of low level actions belonging to the same causal sequence (or thread) is preserved, but there are no ordering constraints on actions in different threads.

As an example, consider a combined voice call and PDA function (e.g., calendar). The voice call can be described as a sequence of steps (selecting number, initiating call, conversation, closing call). Calendar usage is also sequential at high level (opening, using, closing), although can be concurrent at lower level. We can use the concurrent action to combine call and calendar to express the requirement that the user must be able to check the calendar during a call and/or that the user must be allowed to make a call while working on the calendar. Different ways of constructing the hierarchy using concurrent and sequential actions result in different end-user requirements, but the exact meaning of the requirements is always clear. This might look something like this:

1 Simultaneous Call and Calendar (Concurrent)
1.1 Call (Sequential)
1.1.1 Select Number (Primitive)
1.1.2 Initiate Call (Primitive)
1.1.3 Conversation (Sequential)
1.1.3.1 …
1.1.4 Close Call (Primitive)
1.2 Calendar Use (Sequential)
1.2.1 Open calendar (Primitive)
1.2.2 Use Calendar (Concurrent)
1.2.2.1 …
1.2.3 Close Calendar (Primitive)

We feel that it is very important to allow concurrency in order to avoid over-constraining the use case. After all, ordering of the steps is an additional requirement that has to be explicitly justified, because it limits the implementation possibilities.

2.1.2 Non-Functional Requirements and User Interface Requirements

Most non-functional requirements concern a specific part of the use cases, e.g., response time to a user request or quality of picture at a specific point. This is true for real-time requirements as well as, e.g., for user interface requirements and security requirements. We want to attach this kind of requirements explicitly to the relevant actions of the use case instead of specifying them separately from the use case. On the other hand, we also want to have a purely functional view of the use case available, since the amount of information in the use case tends to grow so big that it obscures the essential function (hierarchical definition of actions is another way to hide details).

© Copyright MARTES Consortium

Ideally, a tool would allow adding non-functional and user interface requirements to either the textual or graphical presentation of the main success scenario. For example, it is convenient to represent a response time as a line segment in the sequence diagram. Furthermore, a tool should support multiple views of the use case for different purposes. All this should be based on a coherent model so that the consistency of different views is guaranteed.

2.1.3 General End-User Requirements and Business Requirements

Some (non-functional) requirements are quite general and difficult to associate to a single use case. A typical requirement refers to the size/weight of the device (e.g., pocket-size), robustness (e.g., waterproof), battery life (although this is actually dependent on the use cases), etc. In use case descriptions, non-functional requirements are primarily intended for requirements that are specific to that use case (or part of it). Normally we do not include general requirements in the use case description.

Some technology and component choices depend almost exclusively on the business environment, and have very little to do with the actual end-user requirements. Business requirements refer to legacy technologies, industrial partners, other leading business players, etc. We omit this kind of requirements completely in our method, although we understand that it can be very important from a business perspective.

2.1.4 Executable Use Case Models

We use UML2 to create executable use case models for the purposes of clarification or demonstration that is often necessary in the negotiations with the stakeholders. In the MARTES approach, we create a separate model for each of the actors of the use case. The actions themselves are modeled as interactions between these models. Interactions are modeled as message (or signal) interchange (asynchronous communication).

In a simple use case, each actor is modeled with a single state machine. Complex use cases are often composed of simpler ones so that the model of each actor is a composition of simpler actor models. The primary actor (normally the user) is composed in a way that expresses the sequential, concurrent or alternative composition of simpler behaviors. The semantics is analogous to the corresponding composite actions of the textual use cases. In most cases, the communication between the actors in the use case is essentially synchronous, in the sense that the initiating actor normally waits for some kind of response before continuing its behavior (e.g., the user wants to see something happening on the display of the device). We model synchronous communication with two-way asynchronous communication, that is, with a request-confirmation message/signal pair.

Non-functional requirements are treated as attributes of specific elements of the model. For example, real-time requirements are refined as (allowed) state transition durations and communication latencies.

As mentioned in the previous section, in a practical business situation there are non-functional and user interface requirements that have nothing to do with the end-user needs, or even with the use case. Our method does not cover such requirements at the moment.

2.1.5 Abstract Use Case

Future mobile devices will offer users several, even tens of different applications, many of which are similar to those accessible today via networked desktops. Sets of applications may run concurrently, and some of them may interact. Their use patterns will be very different depending on the type of the user and the specific context of use. Use cases provide means to capture different usage scenarios, which are

© Copyright MARTES Consortium

composed of application functions and events triggering their execution. An abstract use case treats all actors of the use case as black boxes. We are only interested in the externally observable behavior of the actors. Furthermore, we consider only high-level actions that are meaningful to the end-user to accomplish her or his goal.

2.1.6 Refined Use Case

The use case modeling approach starts with the actor (end-user) oriented use cases that consider the system under design as a black-box, and decomposes them hierarchically to system-oriented use cases that look at the system under design as a white-box, but treat the sub-systems as black-boxes. This proceeds until the component level (with no further decomposition) is reached. Execution requires computation of the function algorithms, and transfer of data within the computation and between executions of different functions, i.e., communications. UML2 provides constructs, e.g., sequence, composite structure and class diagrams, and interface and data type definitions, that support the hierarchical approach. Sequence diagrams allow the definition of states, references, and alternative, repeated, optional, and exceptional sequences.

Use case refinement splits the end-user actions into smaller technical/internal actions of the target system. The goal is to eventually map all the actions to predefined services of the execution platform. Refinement is typically done in many steps. The number of step depends on the size and complexity of the use case. The nature of the target services (primitive vs. high level) also affects the number of refinement steps needed. Each refinement step is verified, which means ensuring that the functional behavior is preserved as well as that the non-functional constraints are satisfied. The particular technique used for verification also affects the number of refinement steps needed.

The refinement steps concern the subject actor only, while all other actor models remain the same. In these refinement steps the state machine model of the subject is split hierarchically and recursively into multiple state machines that communicate with each other. In each refinement step we must ensure that the externally observable behavior (i.e., the message communication at external interfaces) of the subject remains the same.

The goal of the refinement is to finally express the subject's behavior use cases using predefined services. Each predefined service is also modeled as a state machine with a communication interface. The fully refined subject model consists of a hierarchical structure, where the leaves (lowest level objects) are predefined services (with predefined state machine models).

A use case often involves some kind of end-to-end system behavior that may extend outside the borders of the target system (e.g., a single terminal). We have to accommodate this in our method, while, at the same time, translate the required behavior into requirements for the terminal system. For example, we need to decide what part of the functionality goes into the terminal or how to divide an end-to-end performance requirement between the devices on the network. These high-level architectural design decisions are done in the first refinement step. The second refinement step further splits the functionality allocated to the terminal, and maps it to predefined services.

2.2 Application Modeling

Real-Time Embedded System (RTES) application modeling is primarily concerned with the specification of the structure and behavior of the application functionality that still remains independent of the platform implementation on which it will actually be executed. Application modeling for Real-Time Embedded

© Copyright MARTES Consortium

Systems mostly consists of the definition of several views onto the application: Functional modeling (both structural as well as behavioral), Execution Modeling and Workload Modeling.

2.2.1 Functional Modeling

Functional modeling refers to modeling of the application-specific functions of a system. These functions can be implemented in hardware and/or software.

There are different reasons for structuring a model. Structure in a functional model typically encapsulates well-defined functions, while structure in an execution model typically encapsulates a unit of scheduling. These structures are not necessarily the same. Modeling must enable to compose components hierarchically and repetitively. The hierarchical composition consists in refining a function. The repetitive composition consists in taking into account the dimensionality of the component by expressing it in a compact way. It enables to exploit intensive data computations, e.g., data parallelism.

UML define 5 kinds of behavior models: Use-Case Diagrams, State Machine Diagrams, Activity Diagrams, Interaction Diagrams (4 types: Sequence Diagrams, Communication Diagrams, Interaction Overview Diagrams, and Timing Diagrams), and Structure Diagrams (6 types: Class Diagrams, Object Diagrams, Component Diagrams, Deployment Diagrams, Composite Structure Diagrams, and Package Diagrams). Their usage can be summarized as follows:

- Use-Case Diagrams show the relationship among actors and use cases within a system. There have been presented in the previous section as the main models used during the Requirements and Specification phase.
- State Machine Diagrams are diagrams, which show a state machine, including simple states, transitions, and nested composite states. State Machines can be used to specify behavior of several model elements just as to express usage protocol.
- Activity Diagrams show the decomposition of an activity into its constituents.
- Interaction Diagrams emphasize object interactions. These include
    - Sequence Diagrams, which show object interactions arranged in time sequence. In particular, it shows the objects participating in an interaction and the sequences of messages exchanged.
    - Communication Diagrams, which show interactions organized around the parts of a composite structure, or the roles of a collaboration. Unlike a sequence diagram, a communication diagram explicitly shows the relationships among the elements. On the other hand, a communication diagram does not show time as a separate dimension, so the sequence of messages and the concurrent threads must be determined using a sequence numbering scheme.
    - Interaction Overview Diagrams, which are variations of activity diagrams, incorporating sequence diagram fragments together with flow of control constructs. They define interactions through a variant of Activity Diagrams, and are focused on the overview of the flow control.
    - Timing Diagrams, which explicitly show changes in state on a lifeline and metric time (time units). They are specifically useful in real-time applications.
- Structure diagrams support the architecture aspect, using
    - Class Diagrams,
    - Object Diagrams,
    - Component Diagrams,

© Copyright MARTES Consortium

- Deployment Diagrams,
- Composite Structure Diagrams, and
- Package Diagrams.

Modern software engineering techniques try to isolate functionality parts into software components. These software components define specific communication ports in order to exchange information with their environment (other components or other subsystems). Software components can be wired to each other (linked) using connectors into an architectural component composition. By defining Composite Structure Diagrams, UML2 provides support of modeling components using ports and connectors.

2.2.1.1 Functional Modeling in the Telecom Area

Functional modeling of services in the telecom area is usually performed by means of non-UML compliant methods. The use of UML would allow the simulation of these complex systems taking into account both HW and SW components, making possible to extract some system level conclusions from hybrid integration of components that have been characterized at a different level of detail. System level simulations are of a capital importance in telecom services development, as most of the user-oriented services that are currently being proposed are based on software intensive systems which are of a distributed nature. This approach makes necessary to integrate a number of different terminals and platforms, connected through infrastructure and ad-hoc networks, during the development phase.

Some tools have been recently introduced in the market making possible the use of UML to address the needs stated above, such as Enterprise Architect [vii]. This kind of tools allow the designer to precisely characterize the interfaces between each peer SW components running on different platforms independently of their nature, processing power, and presentation capabilities (mobile terminal, PDA, laptop, desktop computer, mainframe, etc.). User-centric distributed processing systems based on heterogeneous communications can be integrated by a number of additional subsystems (e.g., home automation, in car devices or sensor networks). This fact pushes the need to perform dynamic real-time architecture simulation, including aspects related to distributed computing, self-organization networks, minimal operating systems characterization, etc.

UML can be used to improve the field of SW component and sub-system integration, taking advantage of MDA techniques to reduce the gap towards the capture and modeling of the end-to-end characteristics of the system, both functional and extra-functional, and defining a rich component model supporting the required end-to-end analysis. These have been in the list of the most relevant remaining open issues for a long time.

UML is currently being introduced to the modeling of distributed telecom systems and service design activities. It is expected that it will provide the means to solve, to some extent, the usual reusability problem. Today, software functions are generally implemented on processors belonging to separate platforms, with little reusability or hardware independence. Existing reusable subsystems are integrated late in the design process. The telecom industry is moving towards new approaches that would allow the integration of middleware and software services acting seamlessly across several platforms, allowing the deployment of applications independently from the actual computing architecture.

2.2.1.2 Models of Computation

The RTES domain is diverse and heterogeneous, which requires careful separation of computation and communication, in addition to separation of function and architecture [viii]. Different applications usually are

© Copyright MARTES Consortium

best modeled by selecting one or more appropriate models of computation (MoC) among various alternatives, e.g.,

- Synchronous Data Flow (SDF) [ix],
- Dynamic Data Flow (DDF) [x],
- Process Network (PN) [xi],
- Discrete Event (DE) [xii],
- Communication Sequential Process (CSP) [xiii],
- Synchronous Reactive (SR) [xiv].

In data flow computation intensive applications, functional modeling consists in defining a graph of functions (e.g., Pulse compression, Doppler filtering) that describes the data streaming flow. A graph usually matches a mode or a group of modes that is a particular flow in a given context. Each function usually consumes and produces data. Arrays are suitable data structures for data streaming applications. A function may be parameterized. There are different kinds of parameters according to their periodicity of activation.

In streaming applications, YAPI [xv], which is a C++ library for modeling Kahn Process Networks (KPNs) can be used for functional modeling. A process is a function on (infinite) streams. Data flow is a special case of a KPN, in which processes can be written as repetitive functions on finite streams. KPNs and data flow approaches are very well suited for streaming applications. There is a need to extend these approaches with control constructs in order to add termination, run-time reconfiguration, and modeling of passive components (e.g., monitors).

Variant of KPNs are needed, called Communicating Regular Processes (CRPs), in which tighter constraints on the source code allow static analysis of a system. Given a CRP model, the compiler can then detect deadlocks, impose and check buffer size limits, resize arrays and schedule execution in an optimal way. CRP must also be extended with a modicum of dynamic control without sacrificing the possibility of a static analysis.

2.2.2 Execution Modeling

Execution modeling involves organization of communicating tasks as task graphs and refinement of unbounded communication channels to bounded ones. This can be done using a Task Transaction Level (TTL) interface, whose logical model is depicted in Figure 2.

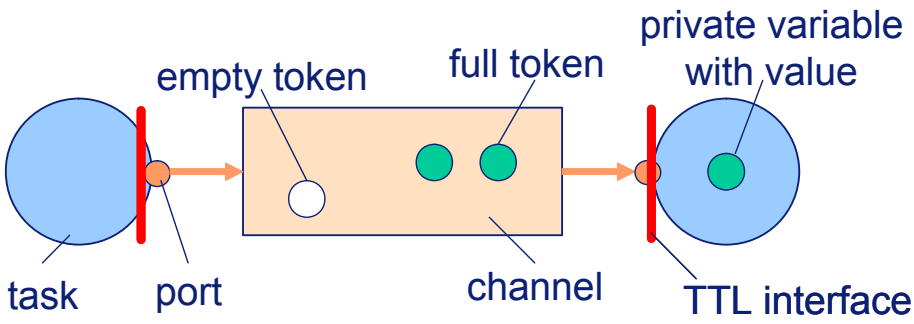


Figure 2: Logical model for execution modeling.

© Copyright MARTES Consortium

A task is a unit of scheduling which can be a process (thread), co-routine, or an actor (function). Tasks communicate with each other through interface function calls. There are different communication styles that are modeled with interface types. Each of the 7 interface types provides a set of interface functions. Each interface type is split in an output side (data production) and an input side (data consumption). The interface functions implement communication services based on typed, unidirectional, reliable, ordered, and bounded communication channels.

- The first interface type is called **Combined Blocking (CB)**. This is a message-passing interface that combines task synchronization services with data transfer services in one function. The output side provides a **write** function. The input side provides a **read** function. These functions transfer a vector of tokens to/from the communication channel and block until there is room/data available in the channel. Note that the buffer capacity of a channel may be smaller than the length of a vector. The type of a vector element must match with the type of the tokens of the communication channel.
- The second interface type is **Relative Blocking (RB)**. This is also a message-passing interface but it splits task synchronization and data transfer in three functions. The output side provides **acquireRoom, store,** and **releaseData** functions. The input side provides **acquireData, load,** and **releaseRoom** functions. The reason for splitting task synchronization and data transfer services is to decouple the granularity of task synchronization and data transfer. This allows task to perform coarse-grain synchronization with fine-grain data transfer, which is especially useful in tasks implemented in dedicated hardware IP blocks that have limited local memory. In the relative blocking interface type, the acquire functions are blocking until there is room/data available in the channel. Note that the buffer capacity of a channel cannot be smaller than the length of a vector.
- The third interface type is **Relative Non-blocking (RN),** where the acquire functions are non-blocking. They try to acquire the requested number of empty/full tokens and return a failure when they cannot. The non-blocking functions are used in actors and co-routines, whereas the blocking functions are used in processes. The data transfer functions *store* and *load* transfer a vector of tokens to/from the communication channel starting at an offset that is relative to the starting point of the acquired room/data. The implementation of data transfer functions must take care of wrap-around in the circular buffer of the communication channel. Of course, the *store* and *load* functions may only transfer data to/from the acquired room/data in the channel.
- The fourth interface type is **Direct Blocking In-order (DBI)**. This is a shared-memory interface type that provides task synchronization services. There are no data transfer services, since tasks access data via pointer de-references. The output side provides **acquireRoom** and **releaseData** functions. The input side provides **acquireData** and **releaseRoom** services. The reason for providing shared-memory communication services is to avoid data transfer overhead (data copying) in case the private tokens of a task are mapped onto the same memory as the tokens of its communication channels. The acquire functions are blocking. They return a pointer to the first empty/ full token in the channel.
- The fifth interface type is **Direct Non-blocking In-order (DNI),** where the acquire functions are non-blocking. They try to acquire an empty/full tokens and return a NULL pointer when they cannot. The shared-memory interfaces do not provide vector functions, i.e., only one token can be acquired per function call to avoid wrap-around problems in the circular buffers on the communication channels. Tasks may not assume that consecutive buffer elements are placed consecutively in memory.
- The last 2 interfaces types are **Direct Blocking Out-of-order (DBO)** and **Direct Non-blocking Out-of-order (DNO).** They are identical to the DBI and DBO interfaces, respectively, except from the fact that

© Copyright MARTES Consortium

the release functions take an additional argument indicating the token that should be released. This means that tokens can be released in a different order than that they are acquired. In all other interfaces, types tokens are released in the order that they are acquired. This is useful for large tokens in order to save memory requirements (e.g., when shuffling I, P, and B frames).

In the execution model, designers have to choose a communication style for each communication channel. This choice depends on the allocation and scheduling of tasks to processors, the communication mechanism between these processors, and the efficiency that is needed to implement the communication channel.

The language to describe the execution model can be chosen according to the needs. For a hardware/software neutral description, UML, C++ or Java may be appropriate. This allows functional simulation of the execution model. For a given hardware/software partitioning of tasks, different languages may be appropriate. SystemC may be appropriate for tasks that will be implemented in hardware. C may be appropriate for tasks that will be implemented in software. In the next section we elaborate on SystemC.

2.2.2.1 SystemC-based Execution Model

SystemC [xvi, xvii] is currently the most widely accepted proposal as a unified system-level, executable specification language. Since its release 2.0, some of its standard primitives provide a basic, but insufficient, support for modeling under more abstract Models of Computation (MoCs) (i.e., for PN MoC [x]). A key advantage of SystemC is its facilities for extension. This is reflected by the object oriented (OO) nature of the language and the extra primitives offered for that purpose, such as primitive channels and events. As such, new primitives (i.e., channels) and features, grouped in a methodological-specific library [xvi], only need to use those extension features, which keeps the SystemC kernel small, general and standardized.

In the line of raising abstraction, the standard TLM library, built over the SystemC standard library, proposes a reduced set of general and simple interfaces and some communication elements. As stated by Rose [xviii], the most important part of it is the set of interfaces, which partially restrict the semantic content of communication channels but provide homogeneity, and as such reusability to TLM prototypes for the speed-up of architecture exploration. However, a system-level specification methodology requires a set of specification guidelines and rules, plus a set of MoC primitives with their particular and complete semantic content, which can be executed by the simulation kernel. Thus, SystemC still lacks a complete and methodological support for system-level heterogeneous specification.

During the last years, the Microelectronic Engineering Group of the University of Cantabria is working in the application of SystemC to the system-level, heterogeneous specification of complex embedded systems. In its current development stage, the proposed methodology provides a systematic, fast, and reliable way to construct a specification, which can later be verified through a faster simulation. The basic technique of the methodology is abstraction. The new primitives for the support of new MoCs are built through abstraction from basic (in terms of semantic content), visible and standard SystemC primitives. Specifically, the abstraction in the time domain permits the interpretation of abstract time information (i.e., event order in untimed MoCs) from the specific time information handled by the strict-timed discrete-event (DE) simulation kernel of SystemC, which handles a double coordinate (t,δ), where δ is every SystemC evaluation-update cycle and t the SystemC time tag [xvii]. Abstraction provides additional and important advantages to the specification methodology. Firstly, elements in charge of assisting debugging and primitives for heterogeneous support (interfaces, channels, restriction checkers) are provided as classes of a library called HetSC. Thus, a decoupled extension from the standard SystemC kernel is provided. The HetSC

© Copyright MARTES Consortium

library is basically a communication library since most of its elements are new SystemC channels. Notice that, in contrast to the TLM library, the most important content of the HetSC library is the set of communication channels with their whole and specific semantic content. HetSC can be seen as a superset of TLM, since it can adopt TLM compatible interfaces and it provides a wider set of communication channels, with specific semantic content while providing the means to monitor MoC constraints. The specification methodology has proven to be able to support a single-source design methodology [xix], based on the application of methodology-specific libraries for performing system-level design steps, such as verification through simulation and performance estimation [xvi] and implementation steps, such as automatic SW generation [xx]. The proposed methodology is open and can easily integrate any new MoC that can be simulated in C++.

The specification methodology enables the writing of an executable specification. The specification exhibits at least one hierarchical level, although more levels are possible. The top-level is instanced from a *sc_main* function, which encloses the test bench and the system modules (sc_module primitive). The system specification is part of the executable specification. Another main part of the executable specification is the environment model, which is used to provide input stimulus and check the output results from the system specification. Environment and system are communicated by means of channels, which become the input/output of the system. Since SystemC version 2.1, the *sc_export* primitive has enabled the inclusion of the I/O channels inside the system module, as show in Figure 3.

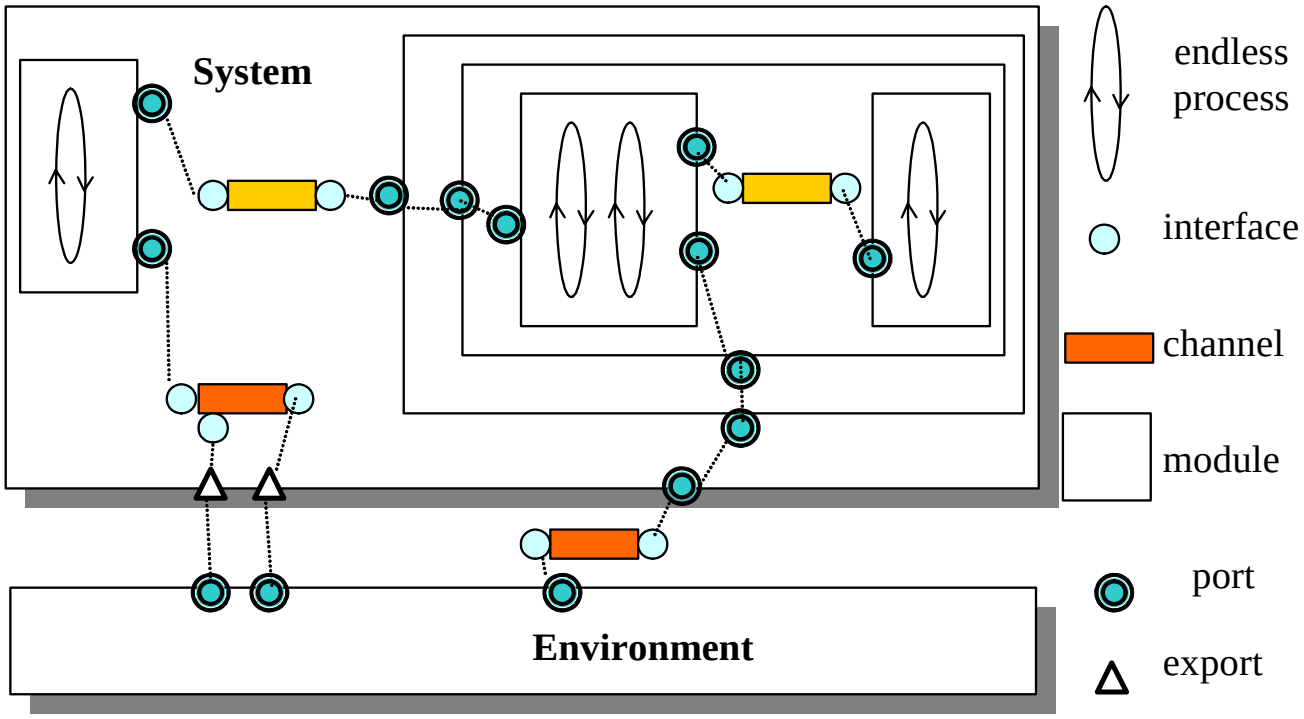


Figure 3: Primitives of the specification.

The specification methodology establishes basic rules and methodological guidelines. For the environment modules, the restrictions are, besides those imposed by SystemC, the need to have ports compatible with the channels which connect system and environment modules. Restrictions in the system specification

© Copyright MARTES Consortium

focus the task of the designer on writing concurrent functionality through C/C++ algorithms (thus, on writing computation code). Concurrency is supported since the functionality is contained by SystemC processes (*SC_THREAD* and *SC_METHOD*). There is a strict separation between communication and computation, since channels are the only way to communicate processes. The methodology makes it possible to specify communication by using (instancing and accessing) standard or library provided channels, only considering their semantics. Explicit synchronization through *wait* (except wait on time delay) and *notify* primitives is not allowed, since they cannot appear in the computation code. The use of new user channels at system level is possible, but associated implementations have to be provided for the support of system-level profiling and SW/HW generation steps [xix]. Following the IP reuse-oriented design recommendations, communication among processes of different module instances necessarily passes through ports (*sc_port*). As with threads and channels, port and module generation is static. IPs of any class can also be included as modules. The IP must have compatible ports otherwise the development of a wrapper is necessary.

Some other additional information is included for other design steps, such as performance estimation and SW generation. For example, the HW/SW partition is specified through *#define* preprocessor clauses. Additionally, some macros assist the debugging tasks of the specification. For instance, the *CH_MONITOR* macro enclosing each channel access in the specification code is used to obtain a report of the last blocking state of every process at the end of simulation. This is a helpful tool for early detection of deadlock situations at the specification level.

As presented in Figure 4, the specification methodology supports heterogeneity, offering a single specification and execution (simulation) framework, in contrast to other approaches, which are based on different simulation engines, and, commonly, on different specification frameworks.

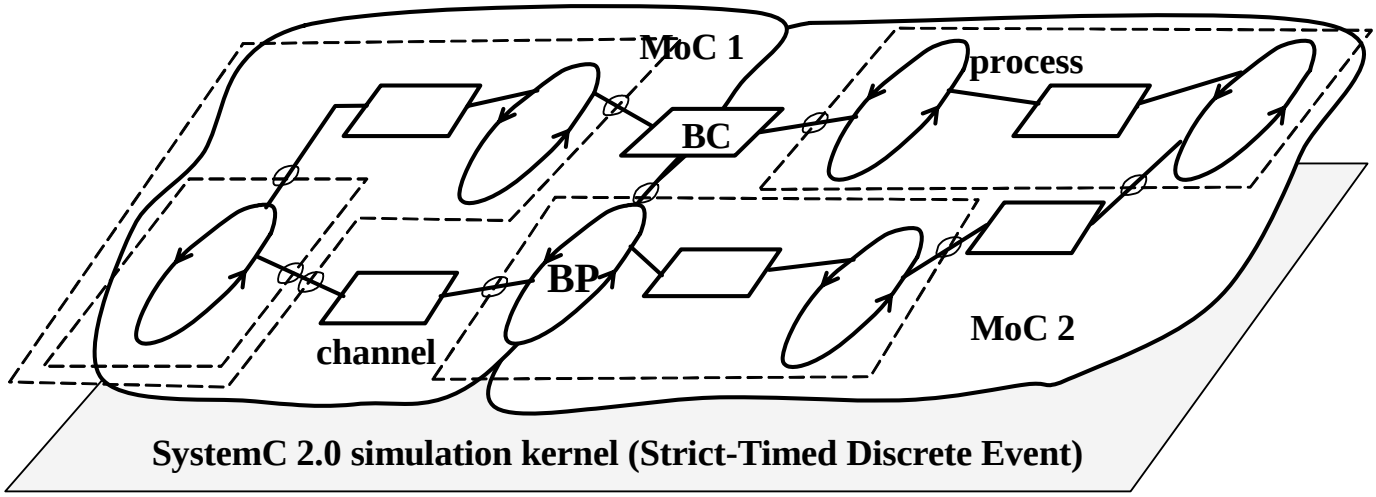


Figure 4: Single specification and execution framework.

The common specification framework comprises specification guidelines for heterogeneous specification and the public SystemC primitives (most of them provided by the HetSC library). The standard strict-time DE reference simulation kernel of the SystemC distribution constitutes the common simulation framework.

© Copyright MARTES Consortium

The new SystemC primitives provided by the HetSC library are macros, interfaces, and mostly MoC-specific channels, providing additional features or different semantics to standard channels. There are also elements transparent for the user. Such is the case of check classes, in charge of monitoring MoC constraints.

The HetSC classes are written using public (visible) SystemC elements, which explains why no kernel extension is needed. The C/C++ OO capabilities (inheritance, polymorphism) and extension primitives provided by the SystemC kernel for creating channels (*sc_primitive_channel* and *sc_channel*) and handling synchronization and detailed time (*sc_event* class, *wait*, immediate and delta *notify* methods, the evaluation-update mechanism through the *update* and *request_update* methods or the *sc_delta_count* method) are used. In this way, new primitives encapsulate abstract communication functionality, hiding and internally using standard primitives, which permit the fine control of execution of the new primitive. For instance, for the channels provided, it is possible to define their blocking and resumption conditions, transfer of data, internal storage capability, etc. However, the specifier only needs to know the semantics of the new element and the way to use it, that is, how to instance and call the channel in the specification code.

Specification guidelines and rules complement the specification methodology especially in those points where checking classes cannot warn or oblige the specifier to fulfil MoC restrictions. Additionally, MoC condition checkers can be enabled and disabled to allow some flexibility in the application of MoC rules.

The methodology focuses first on the support of abstract MoCs with well-known properties, formally studied and demonstrated. It currently supports the following MoCs: Process Networks (PN), Kahn Process Networks (KPN) [xi], Communicating Sequential Processes (CSP) [xiii], Synchronous Data flow (SDF) [xiv], and the Synchronous Reactive (SR). This support is approached in a structured way, following the MoC classification in [xxi], which considers the level of detail handled in the time domain. The classification distinguishes untimed, synchronous, and timed MoCs.

In untimed MoCs, the only time information handled is the order relationship among the events of the system. The unique time tag associated to each event fixes this order. However, this time tag is not explicitly handled at the specification level. Most of untimed MoCs supported (KPN, PN, CSP, SDF) are characterized by a homogeneous computation style, that is, a sequential algorithm, which is mapped to the SC_THREAD, the most general class of process provided by the standard SystemC. Untimed MoCs are mainly distinguished amongst themselves by their communication domain. The PN and KPN MoCs are characterized by the FIFO channel. SystemC can provide a basic coverage for these MoCs through the *sc_fifo* channel. However, their support has been improved through two new channels provided by the HetSC library, the *uc_fifo* (for PN) and the *uc_inf_fifo* (for KPN). The *uc_inf_fifo* provides an infinite size semantic, the only limitation being the host heap memory. In addition, these new channels implement checks, such as monitoring if several writers or readers access a channel instance or banning internal data introspection. While these channels illustrate the improvement of fidelity, other primitive channels of HetSC show how the lack of SystemC primitives matching the constructors of other MoCs, such as the untimed CSP and SDF MoCs, is covered. For CSP, the *uc_rv_sync*, *uc_rv_uni* and *uc_rv* rendezvous channels are provided. Each one provides a specific communication semantic based on rendezvous. That communication semantic can be approximated through two fifo channels. However, each HetSC rendezvous channel provides in a single primitive the syntactic and semantic required by rendezvous. In SDF, the *uc_arc* channel is in charge of offering syntactics, semantics and checks of this MoC. The *uc_arc* demonstrates the focus of the methodology in channels to acquire the additional information that some MoCs require, as well as to fix the execution semantics. The SDF requires providing production and

© Copyright MARTES Consortium

consumption rates of SDFG arcs. Although they could be provided as separate information or as information associated to the computation node, this information is acquired in the methodology at instance time and handled at execution time by the *uc_arc* channel. The rate figures are internally used to condition the execution semantics, i.e., the consumption rate fixes when to unblock the process where the SDF node is mapped.

As support for the Synchronous Reactive (SR) MoC, a representative synchronous MoC has been given. Synchronous MoCs impose more restrictive conditions on the event ordering. Specifically, the SR MoC imposes the perfect synchronity hypothesis, which establishes that the system reacts to input data producing output results in zero time (instantaneous reaction). Therefore, outputs are synchronous (they have the same time tags) with the inputs that provoked them. As a consequence all the activity of the system concentrates on specific points of the time axis, called slots. In the methodology, a new *uc_SR* channel is provided which enables a way to connect processes in what is called a reactive chain, where every event has the same SystemC simulation time for the same reaction. The *uc_SR* behaves in two ways, depending on the specification structure: it can behave (1) as a reactive channel (RC), in which case it is only in charge of transmitting the reaction and data without simulation time advance (only with delta time advance), or (2) as a generator channel (GC), in which case the *uc_SR* channel receives and transmits the events of the generator process, which are at the beginning of the reactive chain sourcing data at different slots. The different SystemC time tags (of type *sc_time*) distinguish the events of different slots. Two new important checks are performed in GCs which help to fulfill SR conditions. A first check obliges separation in SystemC time of different source events (generator events). A second one checks that the reaction time has not provoked a SystemC time advance.

As the underlying simulation kernel is common, heterogeneous specification requires the definition of the MoC interfaces. MoC interfaces are located in border channels and/or processes. Thus, no element external to SystemC has to be used to perform the MoC adaptation. This again maintains the homogeneity of the specification and the single simulation kernel.

As the underlying simulation kernel is common, heterogeneous specification requires the definition of the MoC interfaces. MoC interfaces are located in border channels and/or processes. Thus, no element external to SystemC has to be used to perform the MoC adaptation. This again maintains the homogeneity of the specification and the single simulation kernel.

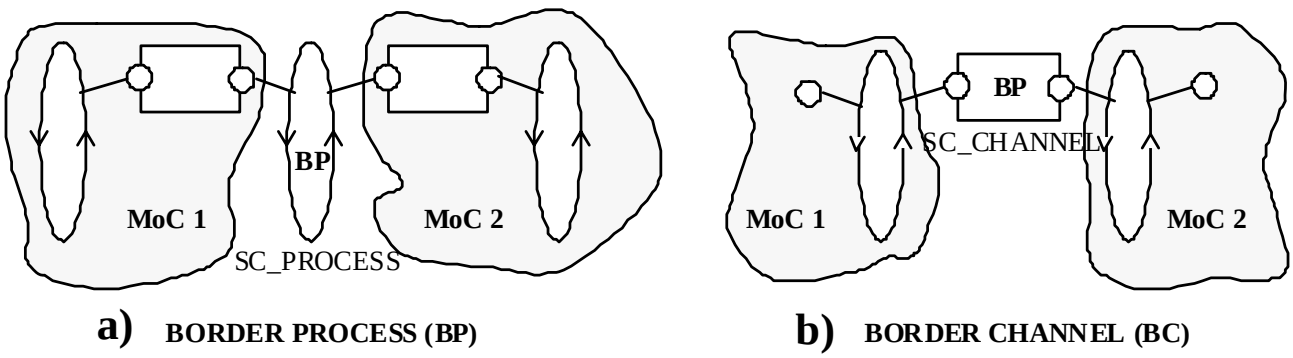


Figure 5: Border channels and border processes.

© Copyright MARTES Consortium

A Border Process (BP), as depicted in Figure 5a, is a process accessing channels belonging to different MoC parts, where the adaptation is explicitly written inside the BP functionality. The border process is a highly flexible mechanism that can be used by the specifier to mix two or more MoCs. A Border Channel (BC), as depicted in Figure 5b, is a SystemC channel where the MoC interface is concentrated. The processes that communicate through this channel belong to different MoCs, with their own properties and characteristics. The adaptation is hidden and performed within channel implementation, but the specifier implicitly knows its semantic. The basic principle of the BC semantic is that on each MoC side, it is seen as the channel associated to the MoC. This is not always strictly possible since the intersection of the assumptions and restrictions corresponding to the MoCs involved in the interface might lead to incompatibilities, solved through specific criteria. BCs avoid the rewriting of the adaptation code, and allow a clear and explicit separation among MoC parts. They also enable the direct connection of modules specified with different MoCs (component-MoC match) presenting ports with different access interfaces, which is interesting for the exchange of intellectual property (IP) blocks. Transactors can be considered as a particular case of BC (usually untimed), however, the methodology considers more cases involving other MoCs.

From the MoC classification based on the time domain, a MoC interface taxonomy which considers untimed-untimed, untimed-timed, untimed-synchronous, etc., MoC interfaces, is derived. Up to now, untimed-untimed and untimed-SR (a specific case of untimed-synchronous) MoC interfaces have been dealt with. MARTES developed a semantical framework for defining different MoCs in UML in order to support the refinement of the original UML model into SystemC, ensuring the preservation of the computational model specified.

2.2.3 Workload Modeling

Workload models are created from functional and execution models of the application to facilitate fast, light effort performance evaluation at system level. Workload models present resource requirements of functional and execution models as computing and communication loads that exercise platform resources. Computing and communication loads are obtained from functional and execution models through estimation techniques.

Workload models illustrate the load the application causes to the physical architecture when executed. They are non-functional in the sense that they do not perform the calculations or operations of the original application, e.g., workload models of a video encoder application are not able to encode video. Workload models enable performance evaluation already in the early phases of the design process, because they do not require that the modeled applications are finalized. The use of workload models enhances simulation speed as the functionality is not simulated and typically they can be easily modified to quickly evaluate various versions of use cases.

Workload models have a hierarchical structure. Starting from the bottom, we have ***load primitives,*** such as *abstract instructions*, which are used to characterize the load. For example, these abstract instructions could be *read* and *write* for modeling memory accesses and *execute* for modeling data processing. The physical architecture model used in conjunction with the workload models has to provide an interface, through which the load primitives are transferred during simulation.

An ordered set of *abstract instructions* based on their dependences, is called a ***basic block***. Basic blocks and their application-side control, e.g., branches and loops, form a ***function***. A ***process*** consists of one or more functions and their control, and is the basic unit for the operating system scheduler in the physical architecture model. Finally, an ***application*** comprises of one or more processes and the control between

© Copyright MARTES Consortium

them. It may be necessary to add further layers on top of the application layer to model complex distributed workloads, e.g., a ***distributed application*** and/or a ***networked application*** layer that are typical in telecom systems.

There are a number of methods for obtaining the load information for the workload models from different sources.

- In the ***analytical method***, the number of memory accessing and data processing operations is estimated from the algorithm description or other suitable source. This is most suitable for simple DSP-like applications (e.g., static data flow models), where the control flows are simple enough so that such estimations can be made with reasonable accuracy. However, often it is very difficult to get a realistic distribution for the operations with real-life applications.
- Another alternative is ***source code based modeling***, which relies on a compiler tool chain for producing the required information for the workload models or for mapping execution models on SystemC TLM or ISS architecture models. The source code of the application, or of a very similar application, has to be available, and it has to be mature enough so that it can be compiled at least in a workstation environment, if not in the target architecture. In this method, a profiler is used to obtain control information for the workload models, and to limit the modeling work to most loading functions. The compiler front-end can be used to produce the load primitives for the models.
- The third method for obtaining the load information is ***trace or measurement based approach***. In this case, an existing execution trace or measured load information of a similar application in a resembling architecture is used as input. Multiple independent traces or measurements can be combined to model complex use cases. This is a rapid way for producing workload models but also a bit limited due to the prerequisites.

Workload models can be implemented in UML using, for example, class diagrams for the model hierarchy, and state machine diagrams or even text diagrams for the control and load primitives. Figure 6 shows an example how state machine diagrams have been used in workload models of MPEG-4 intra-frame execution. The model simulates processing of macroblocks one by one by first calling a separate DCT model and then performing quantization in a state machine subdiagram.

After the workload models have been created, they must be allocated to the physical architecture model. In this phase, e.g., the host processing unit and memory block for each workload is selected. Finally, the workload models are combined with the physical architecture model for performance co-simulation, possibly transforming the workload models to the same language as the physical architecture models first with, e.g., SystemC code generation.

To support performance evaluation at system level, the development framework would have to be able to provide the means for handling of non-functional and resource constraints, allowing the designer to take into account during the design phase of the criteria yielding to conflicting dependability, helping to understand the trade-offs between average performance and predictability.

© Copyright MARTES Consortium

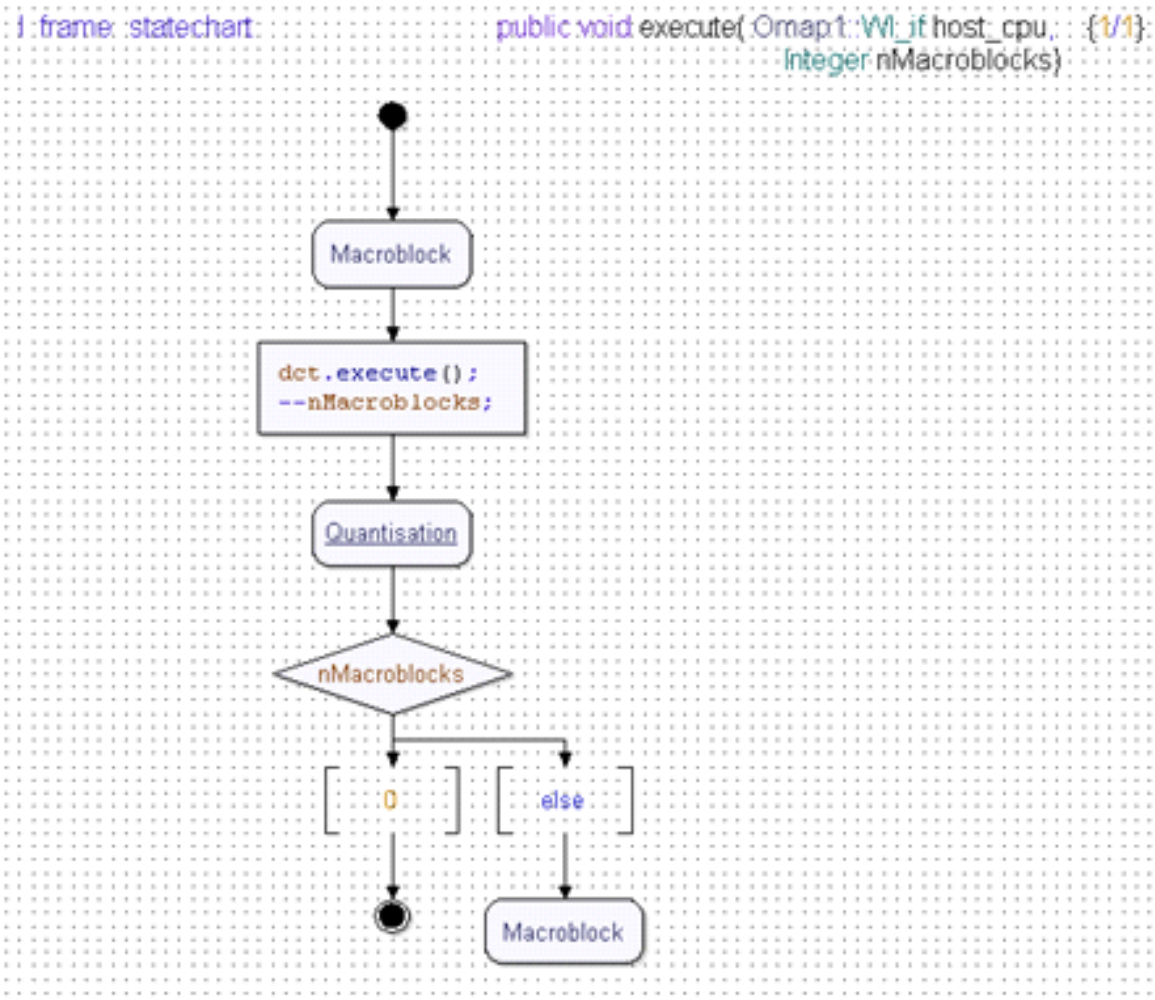


Figure 6: Example of using state machine diagrams in workload modeling.

2.3 Execution Platform Modeling

The Execution platform modeling comprises the description of hardware and platform software (middleware) components and interconnections that are useful for the chosen purpose (e.g., performance simulation, code generation). Similar to application modeling, platform modeling must consider hierarchical and repetitive structures to exploit topology and parallelism. It is described at the same level of abstraction as the application, and contains also timing information along with structural and behavioral aspects.

The models are not used as a true execution engine, where an ISS is used for software and RTL VHDL code is used for hardware. Instead, execution platform models provide platform-related information to application designers, such as what function a platform can support and what is the performance impact. One should be aware that, in the design of an embedded system, the execution platform is not as well defined. This is obvious when the target is an ASIC, or, to a lesser degree, an FPGA. It is still true for a SoC platform, where, even if the core processor and the interconnections are fixed, many other parameters

© Copyright MARTES Consortium

are not yet defined, such as the selection of IPs or the amount of memory are under control of the designer. Hence, the execution platform modeling process must allow for architecture exploration.

Architecture exploration is a tedious and error prone process, which must be helped as much as possible by the chosen methodology. One of the key factors is incremental modeling: a small change in a part of the design should not entail extensive modification of all the configurations.

Another point is that, in many cases, one infers the architecture from a higher-level specification, instead of deciding it a priori and then trying to fit the application. This is for instance the case for ASIC design, where the synthesis tool infers the architecture from its RTL description. This is also true from SoC design, where one infers the architecture (i.e., the selection of IPs) from a functional description of the application.

The interconnections between components should match with the execution model if the components are application-specific components. If two application-specific components are connected by a communication channel in the execution model, and these components and the channel are implemented in dedicated hardware, then the transactions in the execution model should also be visible in the execution platform model. General purpose hardware uses another level of transactions (e.g., processor loads and stores) typically covered by AXI and similar protocols. The same holds for general-purpose (infrastructure) software and application-specific software.

The execution platform models consist of layered architecture models of the platforms and layered service models of the platforms.

- The bottom layer is the ***component layer***, which includes the *processing units* (e.g., dedicated hardware), *storage units (*e.g. RAM), *interconnection units (*e.g., bus) and *network structure*.
- One layer up is the ***HW architecture layer***, which describes the *structure* and *interconnects* of the system (Usage of the *component layer*).
- Another layer up is the ***platform-architecture layer***, which includes the *platform software*, e.g. the operating system.
- Higher layers that describe more complicated system, e.g., ***distributed computers*** and ***networked platforms***, could be built on top of low-level layers, but these are not detailed further in this document.

The layered ***service models*** are abstraction views of the architecture models. They describe the *platform behaviors* and related *attributes*, e.g., performance, but hide other details. High-level services are built on low-level services, and they can also use the services at the same level. The way to link the architecture models and the service models, is by defining services in the service models, and implementing the services in the architecture models. Such separation speeds up the design space exploration process, because replacing components or platforms by others could be easily done as long as they implement the same services.

The execution platform model is close to the performance simulation model. Some simulation objects (that have no match with hardware components) can be added to the latter in order to drive the simulation. Different kinds of behaviors can be attached to these components (e.g., an access policy to arbiter a bus) for performance simulation.

2.4 Allocation Modeling

Allocation modeling describes the association matching applicative functions onto infrastructure resources. Allocation combines the application model and an execution platform model. The result is called an allocated model. This matching can be done at any level of the hierarchy and with multiple objects (use of

© Copyright MARTES Consortium

repetitive structures). Allocation usually defines both distribution and scheduling. Distribution is the spatial allocation of computation to processing elements, of data structures to memories, and of data transfers to communication resources. Scheduling is the temporal ordering of the activities (computations, value storage or communication) allocated to each resource. Both spatial and temporal allocations have to be coherent to ensure a correct (deadlock free) execution of the application.

Various scheduling models can be associated to processing, storage or communication components of the execution platforms. For example, abstract RTOS models for a software processor component, parallel execution for a hardware processor component, bus models for a communication link, point-to-point models for a communication link, router models for a communication link, and n-port shared-memory models for a storage component

Along with this distribution and scheduling, allocation models can also be used to define the allocation of non functional properties to application or hardware architecture components, such as execution time of a functional component on a given hardware architecture component, or send/receive time of a functional communication on a given physical link.

In general, the allocation models of time-related non-functional properties of application components (defined as durations of computations and communications) are defined by the scheduling model of the physical component they are allocated to. For computations, processing elements of the execution platform introduce a speed-up/down ratio. For communications, send and receive latencies are added to functional communications to represent additional time needed to get across physical links. This can get more complex when the concurrency of communications on links and link interfaces are taken into account.

Not time-related non-functional properties can be considered generally as performance properties and encompass memory footprint, cost, power consumption, silicon area, etc. While some of those properties are static and defined initially for a component (e.g., cost), others are dynamic and depend on computations and communications taking place at the considered time.

For system level performance evaluation, workload models are mapped onto architecture models to generate a transaction level performance simulation model. They can be simple (absolute value, analytical / mathematical formulae) or more complex (behavior). Then, they are translated into requests (e.g., a 100 words memory read) that drive the performance simulation (stimuli).

As the execution platform model is a static model offering a set of computation and communication resources, the dynamic nature of the allocated model resulting from the allocation of the application model to the execution platform model comes from the application itself. The workload model results directly from the combination of the application and allocation models.

2.5 Implementation Oriented Modeling

Implementation oriented modeling encompasses the transformation of components of the allocated model to target a specific exploitation of the model: simulation, SW code generation, HW code generation, etc. As the allocated model (representing a generic set of communicating functions allocated to a generic set of communicating execution resources) is transformed into a domain-specific model, the generic components constituting the allocated model get transformed into implementation-level elements using the API offered by the targeted implementation (e.g., SystemC, C for a RTOS, VHDL, Java). For instance, as a result of this abstraction layer, considering a mobile service development environment, the implementation oriented modeling solves the transformations needed for Java code generation. The modeling translation should

© Copyright MARTES Consortium

meet the constrains, specifications and behavior given by the application, execution platform and allocation models (e.g., OSGi framework, MIDP profile, SIP protocol).

The implementation oriented modeling exploits the resulting allocation modeling. Distribution and scheduling information are used to generate code according to a given execution model. Granularity of the performance simulation code (e.g., TLM) can be coarser than the target one.

© Copyright MARTES Consortium

3. Models in the MARTES Methodology

3.1 Design Rationale

3.1.1 Model Components

Model components are bits of models as opposed to bits of executable code. Tools reuse model components in various ways. For example, based on the application engineer's specifications expressed via a Domain Specific Language (DSL), tools select model components, configure them in particular ways, and compile them. The compiled model components can (but don't have to) execute on top of run-time components.

Platform-independent models allow the designer to focus on the functionality and behavior of the application without having to worry about the implementation details. The MDA guide [ii] says *"A very common technique for achieving platform independence is to target a system model for a technology-neutral virtual machine. A virtual machine is defined as a set of parts and services (communications, scheduling, naming, etc.), which are defined independently of any specific platform and which are realized in platform-specific ways on different platforms. A virtual machine is a platform, and such a model is specific to that platform. But that model is platform independent with respect to the class of different platforms on which that virtual machine has been implemented. This is because such models are unaffected by the underlying platform and, hence, fully conforms to the criterion of platform independence".*

Platform-independence is therefore a relative, and thus not an independent concept. It is not meaningful to assert that a model is independent without a specifying what kinds of platforms the model is independent of. The so-called Y-chart model pattern may be applied several times during the design process in order to be the basis of applying MDA in MARTES. Due to the fact that platform-independence is a relative concept, we consider that in MARTES, we have several platforms according the OMG definition, and so several PIMs and PSMs. A model may thus be platform-independent according a relative platform (a PIM) and at the same time platform-specific for another, more detailed relative platform (a PSM).

For the purposes of this document, *platform-independent* means independent of information formatting technologies, such as C++, Java, Ada, SystemC, VHDL, etc. The MARTES definition of platform is thus the set of formatting technologies that provide a coherent set of functionality through interfaces and specified usage patterns that any subsystem depending on this platform can use without concern for the details of how the functionality provided by this platform is implemented.

This chapter defines the models for the MARTES development. As shown in Figure 7, they are embodied in five models:

- The MARTES Application Model,
- The MARTES Execution Platform Model,
- The MARTES Allocation Model,
- The MARTES Allocated Model
- The MARTES Implementation Oriented Model (SystemC, C, C++, Java, Ada, …).

© Copyright MARTES Consortium

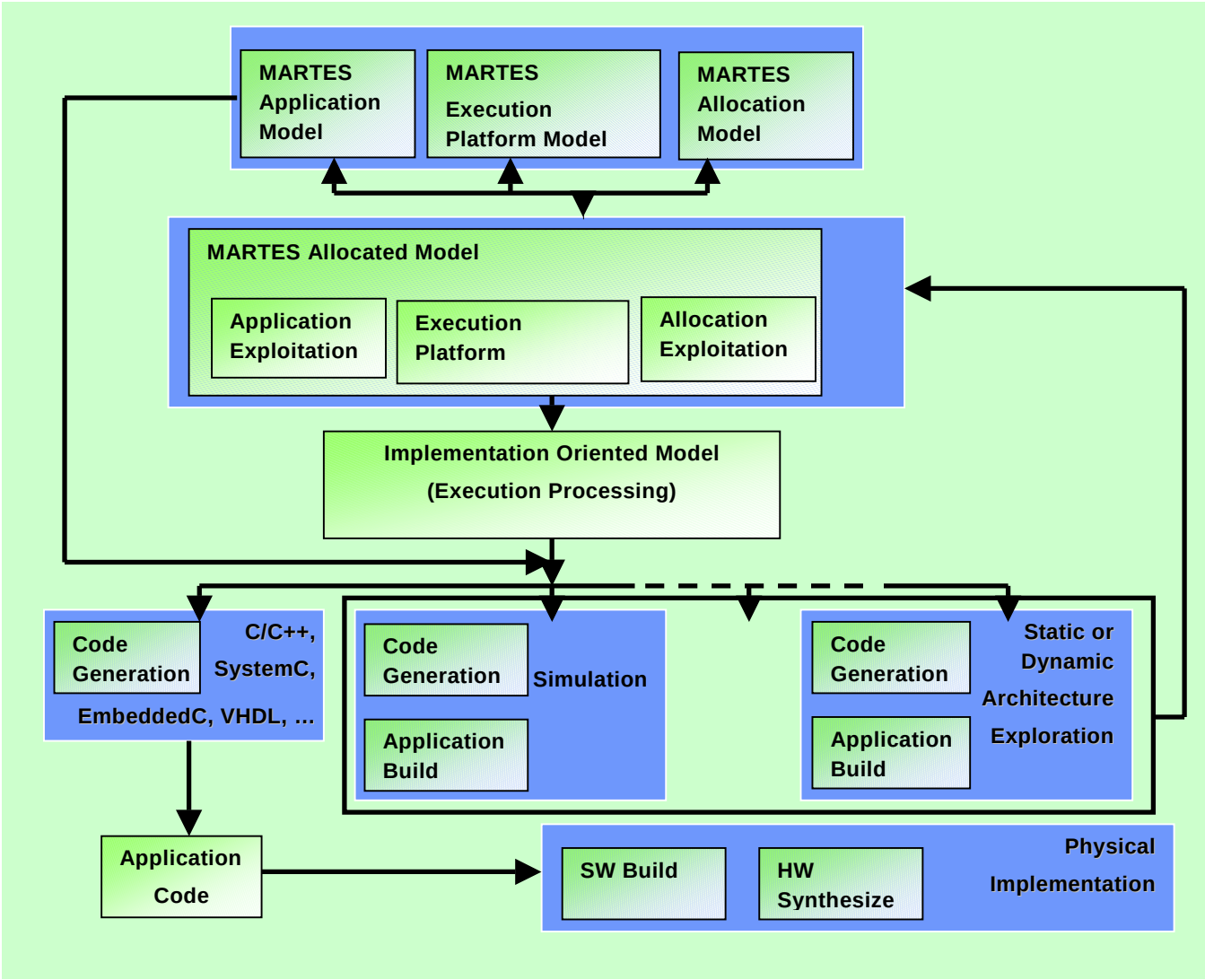


Figure 7: MARTES models and transformations.

3.1.1.1 MARTES Application Model

The MARTES Application Model defines types that MARTES applications must implement. Tools that process specifications, expressed via the UML profile, use the PIM to generate application skeletons or possibly to generate complete applications in some cases. For example, an application engineer marks an element of an application model with certain stereotype of the profile. Then, a tool that processes the model can produce an element that has all features explicitly defined by the application engineer and also the features defined by the associated type in the PIM. Note that, for each concept, there is both a stereotype in the profile and a type in the model. The semantics of the stereotype are that the stereotyped element must implement the type defined in the model.

© Copyright MARTES Consortium

3.1.1.2 MARTES Execution Platform Model

The MARTES Execution Platform Model defines the types that a MARTES execution platform must implement. It includes an abstraction of the hardware architecture specification.

3.1.1.3 MARTES Allocation Model

The MARTES Allocation Model defines the types that a MARTES allocation model must implement. The defined types include the types defined by the MARTES Application Model and MARTES Execution Platform. MARTES Allocation Model contains information related to both application and hardware components. Note that at this level of abstraction, models may not be specific to the platform view of the system.

3.1.1.4 MARTES Allocated Model

The MARTES Allocated Model defines the types that a MARTES Allocated Model must implement. The allocated model also includes information of application and execution platform components. Rules defined in the allocation model are followed. At this level of abstraction, information about the specific technology that is used in the realization of the model on a specific platform is included.

3.1.1.5 MARTES Implementation Oriented Model

The MARTES Implementation Oriented Model defines C, C++, Java, Ada, and/or SystemC abstracted types that MARTES must implement in order to run in platform environment. The Implementation Oriented Model is derived from the MARTES Allocated Model.

3.1.2 Run-time Components

Although run-time components are often critical core assets, this specification does not provide run-time components to support MARTES. In fact, OMG standardizes languages, transformations, and types, but not implementations as such. Tools that implement the languages, transformations, and types defined in this specification will probably provide run-time components that support the implementation.

3.1.3 How the Pieces Fit Together

External clients invoke application APIs and the application invokes infrastructure APIs. Standardized interfaces for MARTES applications make it possible to port external clients from one MARTES infrastructure to another. Standardized interfaces for MARTES applications make it possible for a MARTES infrastructure to support multiple MARTES applications, and to move MARTES applications from one MARTES infrastructure to another.

The fixed types defined in the MARTES Applications Model and the MARTES Execution Platform Model map to fixed C, C++, Java, Ada, and SystemC interfaces for MARTES derived Models, used for simulation or code generation for example. MARTES applications support additional interfaces, corresponding to the features that the application engineer defines in the application model, that is, the model that the application engineer expresses in terms of the UML profile.

A domotic scenario could be taken as an example of application development following the MARTES co-modeling approach. Domotic services provide for home environment control allowing different user interfaces, such as mobile phones or PDAs. The application development process would start with a complete service and platform UML2 model, including behavior definition, SW/HW characteristics, and constrains related to the mobile environment (i.e., memory limits or java security). The required OSGi services and its methods for managing them are described, as well as service providers and requesters

© Copyright MARTES Consortium

(bundles). All of these application peculiarities are expressed via stereotypes given by the corresponding UML profile. This information leads to the so called PIM model in the MARTES modeling approach, which is completely technology independent and is the same regardless of the technology it sits on. This PIM model is then transformed into a PSM model. This PSM model is specific to the particular technology, e.g., J2EE or J2ME in this case. Finally, the Java application code is generated by means of a code generation tool. That is to say, instead of writing the application by hand based on the UML model, a tool generates the majority of the code from the UML model. The SW generation tool generates all software logic needed, meeting the constrains, specifications, and behavior given by the application and hardware models.

To provide a common way of modelling both hardware and software aspects of a RTES, in order to improve communication between developers and enable interoperability between domain development tools used, a Domain Specific Model (DSM) has been defined. The MARTES UML profile described the UML extensions required to support the concepts defined in the Domain Specific Model.

3.2 MARTES Domain Specific Model

3.2.1 Domain View

The MARTES Domain Specific Model extensions are used to define the structure and parameters of an application and execution platform as well as their allocation. Correspondingly, the system design is divided into three parts: application description, execution platform description, and allocation. Both the application and execution platform descriptions can be designed independently of each other. The MARTES Domain Specific Model mainly concerns the structure of an application and execution platform. The application is seen as a set of active components with an internal behavior. The platform is seen as a set of component too.

The MARTES Domain Specific Model classifies different application and platform components by defining various composed elements and strict rules how to use them. The objective is to enhance the support of external domain tools for automatic analyzing and profiling a description of an embedded system. The classification also assigns defined parameters to proper components.

© Copyright MARTES Consortium

3.2.2 Application Model

Figure 8 and Table 1 describe the MARTES Domain Specific Model dedicated to the Application Model.

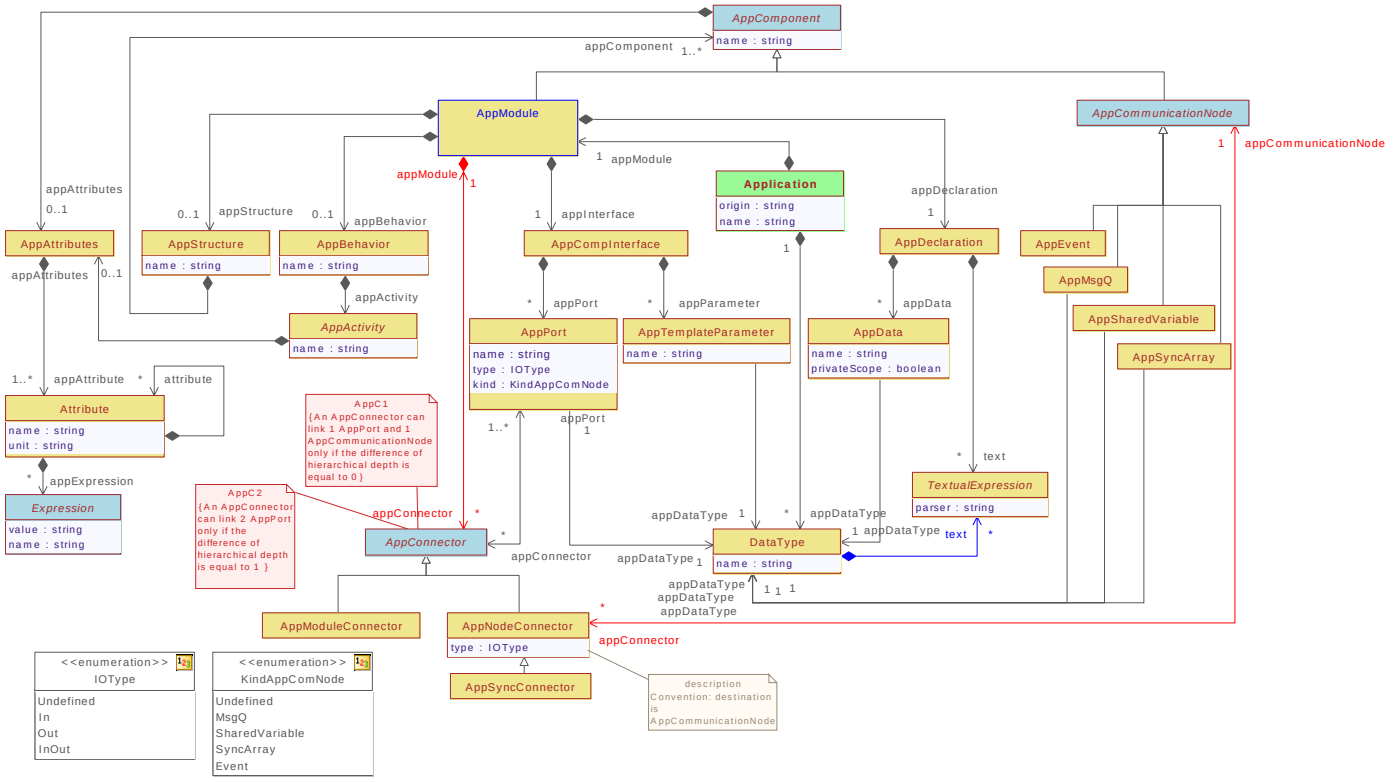


Figure 8: MARTES domain specific application model.

MARTES Domain Specific Application Model elements	Description
Application	Application model container. An application is composed of a set of components.
AppComponent	Component of the application model.
AppModule	Active behavioral or structural element of the application. All modules of an application are connected (communicating) and concurrent (i.e., run in parallel). Structural modules include other sub-modules in a hierarchical way. Leaf modules of the hierarchical tree are behavioral modules.
AppCommunicationNode	Communicating element of the application model (inter-module connection).

© Copyright MARTES Consortium

AppMessageQueue	Synchronous data exchange (n-to-n relation) of mailbox type: modules can send messages to and receive messages from a message queue. Temporal dependency (two-way synchronization) is defined by the message queue's capacity (sender blocked when full, receiver blocked when empty) and access policy.
AppSharedVariable	Asynchronous data exchange (n-to-n relation): modules can write and read data stored in a shared variable. There is no temporal dependency (no synchronizations) between readers and writers.
AppSyncArray	Data array on which producer and consumers are synchronized.
AppEvent	Synchronization event between modules (no data exchanges) with a defined policy.
AppActivity	Behaviors are defined as a set of linked activities, including operations (computations) and inputs/outputs (communication actions), according to control flow operators (sequence, alternative, iteration, parallelism). The way activities are composed/linked defines the active module's behavior.
AppPort	Input or output port of a module allowing for inter-module communications and synchronizations (data, event, and message flows).
AppConnector	Connection between a port of a module and a communication element.
AppBehavior	Behavior associated to an active (as opposed to structural) module of the application. A behavior organizes computations and communication actions (inputs and outputs) for delivering a specific function.
AppCompInterface	Interface of a module (its visible part from the outside, i.e., other components) defining its connection points (type of communication: message queue, shared variable, event, direction in or out, associated data type) and parameterization variables (for tuning internal variables from the outside).
AppTemplateParameter	Parameterization variable (tuner) allowing for setting values of internal parameters influencing the behavior or characteristics of a module from the outside.
AppData	Local data (internal variables) of a module.
AppAttributes	Structural element that includes attribute definitions.
AppAttribute	Attribute of a component, containing behavioral or time-related characteristic.
AppStructure	Structural element. Organized set of components or sub-components (hierarchical tree).
AppModuleConnector	Connector used by a module to manage a connection with a communication element.
AppNodeConnector	Connector used by a communication element to manage a connection with a module.
AppSyncConnector	Specialization of a AppNodeConnector for synchrone communications.

© Copyright MARTES Consortium

AppDeclaration	Declaration of variables or textual expressions.

Table 1: Descriptions for application stereotypes.

Figure 9 and Table 2 describe the Domain Specific Application Model elements dedicated to behavioral modeling.

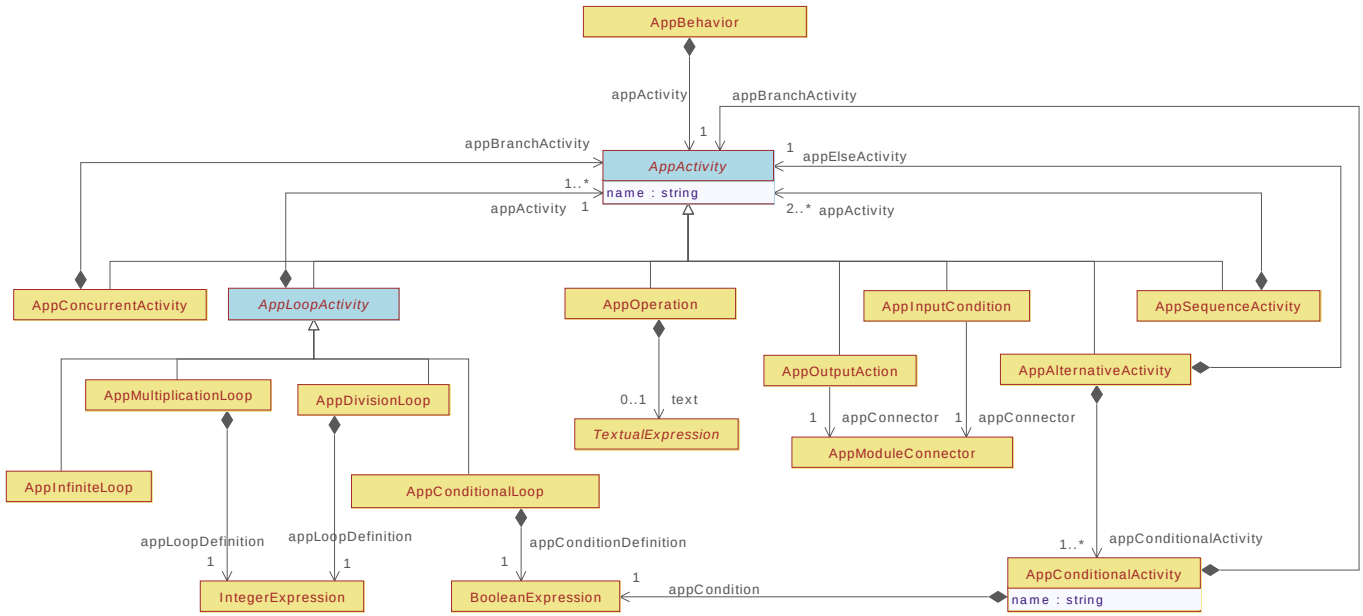


Figure 9: MARTES domain specific application behavioral model.

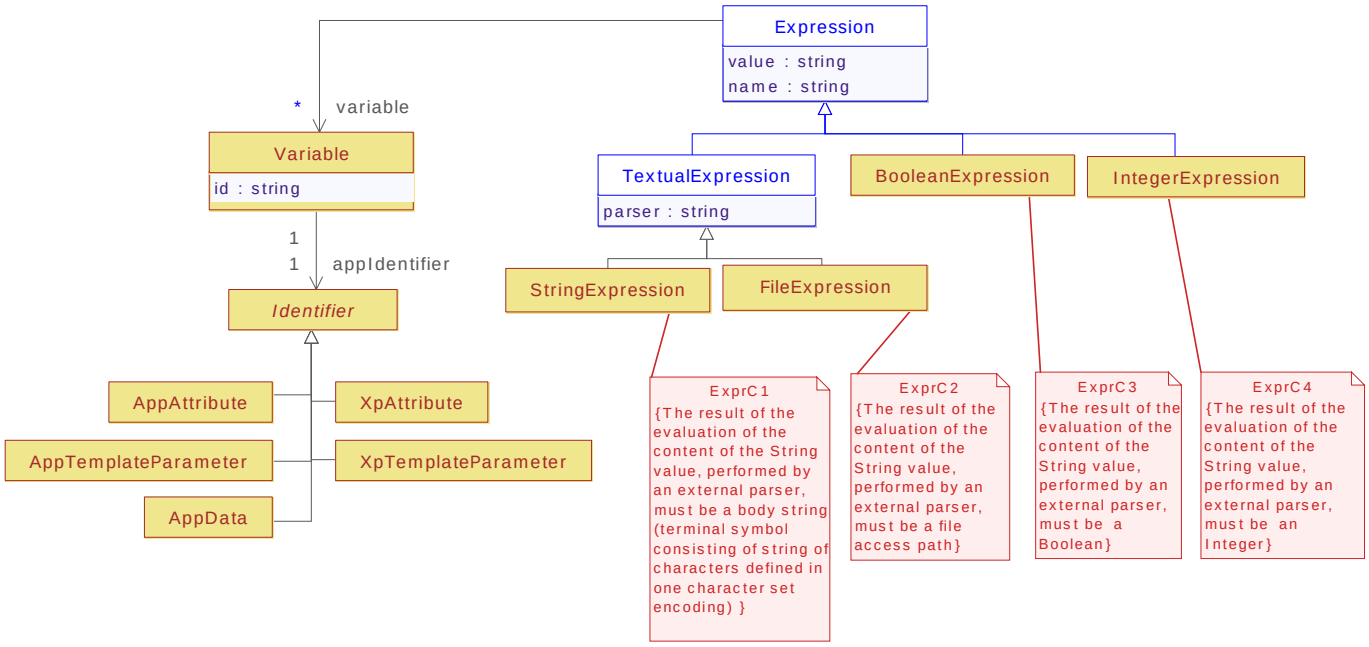
MARTES Domain Specific Application Behavioral Model elements	Description
AppBehavior	Behavior associated to an active (as opposed to structural) module of the application. A behavior organizes computations and communication actions (inputs and outputs) for delivering a specific function.
AppActivity	Behaviors are defined as a set of linked activities, including operations (computations) and inputs/outputs (communication actions) according to control flow operators (sequence, alternative, iteration, parallelism). The way activities are composed/linked defines the active module's behavior.
AppOperation	Activity defining a computation (data/control processing).

© Copyright MARTES Consortium

AppLoopActivity	Activity expressing the iteration (WHILE/FOR operators) of other activities.
AppAlternativeActivity	Activity expressing an alternative (OR operator) between several other activities (branches).
AppConcurrentActivity	Activity expressing the parallelism (AND operator) of other activities.
AppConditionalActivity	Activity in an alternative branch.
AppSequenceActivity	Activity defining a sequence of other activities.
AppConditionalLoop	Loop in activity while condition is true ('WHILE (condition)').
AppMultiplicationLoop	Loop N times in activity ('FOR (i=0; i<N; i++)').
AppDivisionLoop	Loop 1 out of N times in activity.
AppInfiniteLoop	Loop infinitely in activity ('WHILE (true)').
AppInputCondition	Activity defining an input communication action (waiting for receiving a message or an event, reading a shared variable).
AppOutputAction	Activity defining an output communication action (sending a message or an event, reading a shared variable).

Table 2: Descriptions for behavioral stereotypes.

Figure 10 and Table 3 describe the Domain Specific Model elements dedicated to expression modeling.



© Copyright MARTES Consortium

Figure 10: MARTES domain specific expression model.

MARTES Domain Specific Expression Model elements	Description
Expression	Mathematical expression applying various operators to arguments (or operands).
TextualExpression	Expression, for which the result of the evaluation must be text (string or file expression)
StringExpression	Textual Expression, for which the result of the evaluation must be a body string.
FileExpression	Textual Expression, for which the result of the evaluation must be a file access path.
BooleanExpression	Expression, for which the result of the evaluation must be a Boolean.
IntegerExpression	Expression, for which the result of the evaluation must be an Integer.
Variable	Variable used as argument of an expression.
Identifier	Identify the variable.

Table 3: Descriptions for expression stereotypes.

© Copyright MARTES Consortium

3.2.3 Execution Platform Model

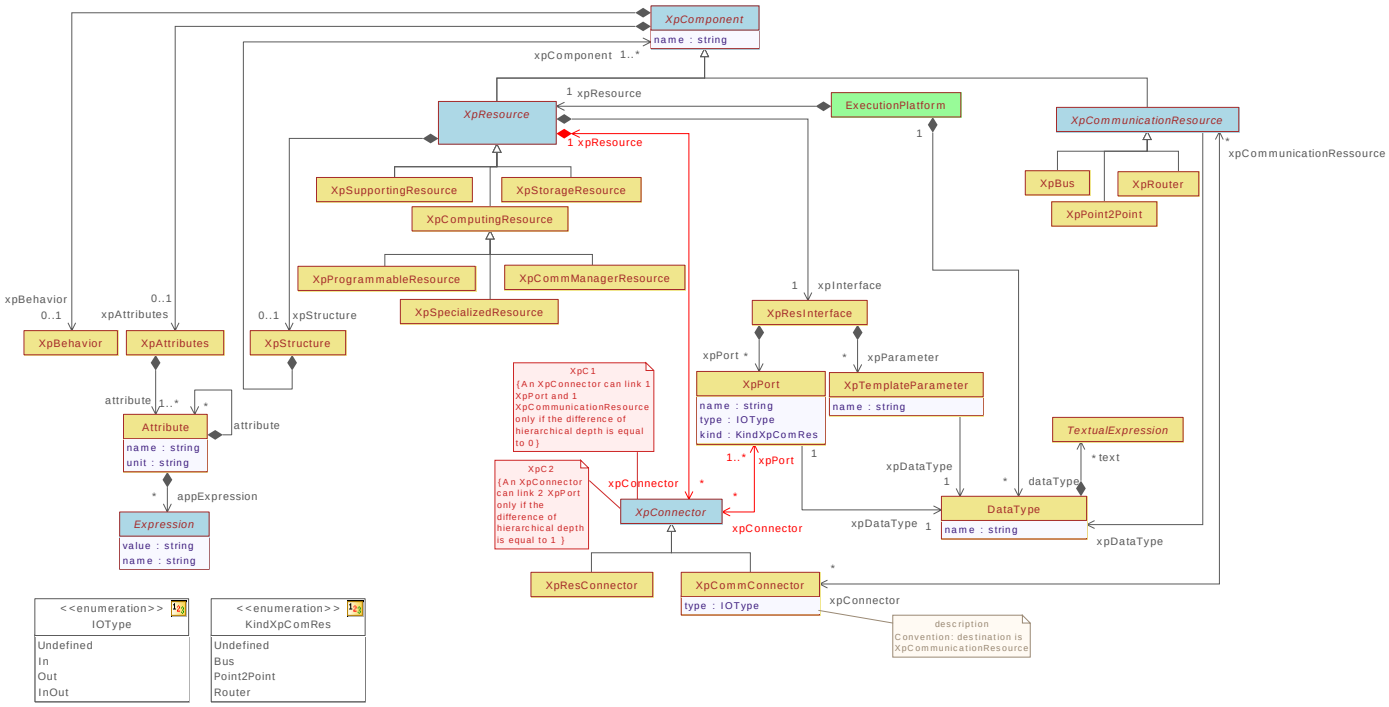


Figure 11: MARTES domain specific execution platform model.

Figure 11 and Table 4 describe the Domain Specific Model elements dedicated to execution platform modeling.

MARTES Domain Specific Execution Platform Model elements	Description
ExecutionPlatform	Execution platform architecture model container. A platform is composed of a set of components.
XpComponent	Component of the execution platform model.
XpResource	Execution (computation, supporting, or storage) resource or structural element of the platform. All execution resources of a platform are connected through communication resources. Structural resources include other sub-resources in a hierarchical way. Leaf resources of the hierarchical tree are execution resources.
XpComputingResource	Computing engine (hardware or software processor).
XpSpecializedResource	Application-specific processor (ASIC, ASSP, etc.).
XpProgrammableResource	Programmable processor (e.g., CPU, MCU for software, FPGA for hardware).

© Copyright MARTES Consortium

XpStorageResource	Storage resource (memory shared between processors).
XpCommunicationResource	Physical link or interconnect for inter-processor communications.
XpSupportingResource	Any resource that does not fit with other resource definitions (i.e., sensors).
XpPort	I/O port of physical component allowing for inter-resource communications.
XpResInterface	Interface of physical component (its visible part from the outside, i.e., other components) defining its I/O connection points (direction in or out, associated data type) and parameterization variables (for tuning internal parameters from the outside).
XpConnector	Connection between a port of an execution resource and a communication element.
XpBehavior	Behavior associated to a resource of the platform.
XpResConnector	Connector used for linking processors at different levels in the hierarchy: port-to-port.
XpCommConnector	Connector used for linking a processor to a physical communication element: port-to-link.
XpBus	Bus (e.g., VME, AMBA).
XpRouter	Router (e.g., network switch).
XpPoint2Point	Point-to-point link (e.g., RS232, USB).
XpTemplateParameter	Parameterization variable (tuner) allowing for setting values of internal parameters influencing the behavior or characteristics of a resource from the outside.
XpCommManagerResource	Communication manager resource (e.g., bus controller, interrupt controller, DMA controller, bridge).
XpStructure	Structure associated to a resource of the platform.
XpAttributes	Structural element that includes attribute definitions.
XpAttribute	Attribute of a component: behavioral or performance characteristic.

Table 4: Descriptions for execution platform stereotypes.

3.2.4 Allocation Model

Figure 12 and Table 4 describe the Domain Specific Model elements dedicated to allocation modelling.

© Copyright MARTES Consortium

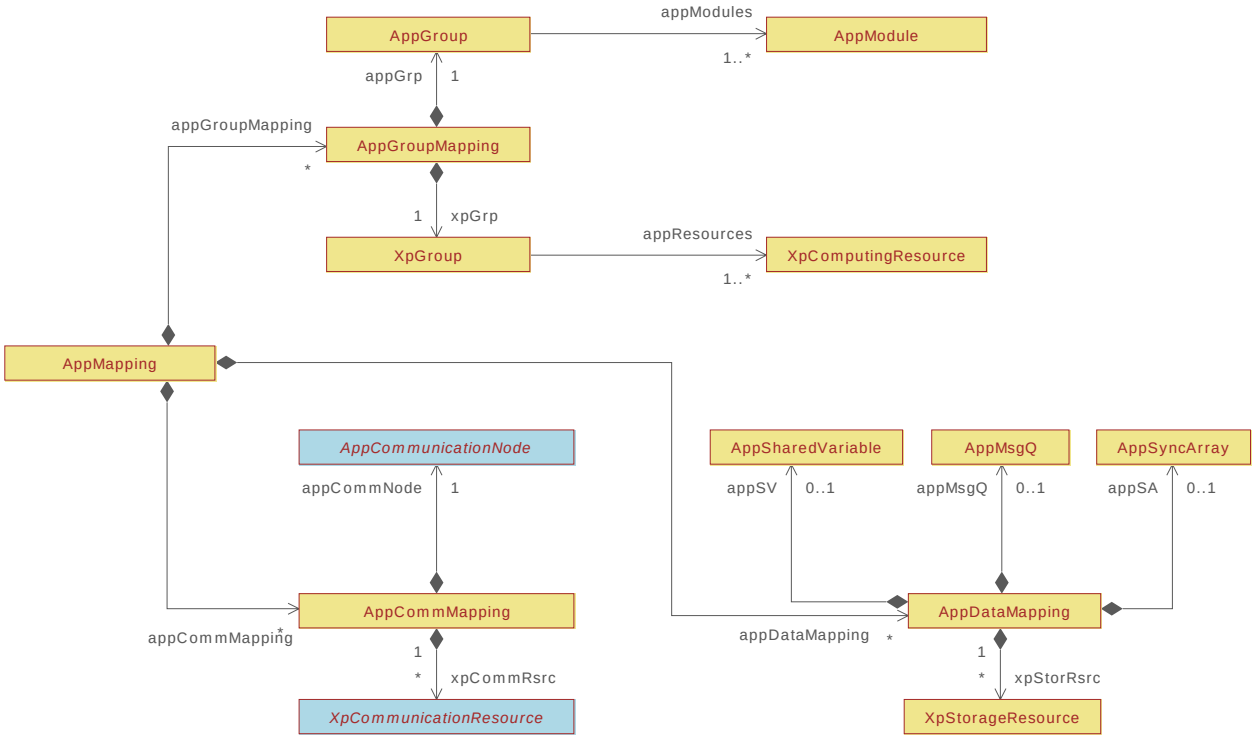


Figure 12: MARTES domain specific allocation model.

© Copyright MARTES Consortium

MARTES Domain Specific Allocation Model elements	Description
AppMapping	Structural element for allocations, which includes AppCommMappings and AppGroupMappings.
AppGroup	Group of application modules.
AppGroupMapping	Allocation of application modules to platform resources.
XpGroup	Group of platform resources.
AppCommMapping	Allocation of application communication nodes to platform communication resources.
AppDataMapping	Allocation of application data communication channels (shared variables, message queues, sync arrays) to platform storage resources.

Table 5: Descriptions for allocation stereotypes.

3.2.5 Allocated Model

The allocated model is the result of the combination of three models: the application model, the execution platform model, and the allocation model, which describes how application and physical elements can be combined. This combination captures design choices of the system designer who decides which components of the execution platform “hosts” which component of the application model. Then, when allocation decisions are taken, the allocation model provides rules for defining the structure, behavior, and properties of the allocated model.

Composite structure diagrams are used to describe the connections between components of the application (functions and communications channels) and execution platform (computing resources, storage elements and communication links) models. Active components of the application model represent the concurrent processes (computing elements) and communication channels (communication elements) of the application, while other components are only structural, and therefore have no behavior defined. Components can be hierarchically organized using class diagrams and composite structure diagrams.

The allocated model is described in a composite structure diagram representing the mapping of application components to execution platform components following rules defined in the allocation model. Resulting from the allocation, new components can appear in the allocated model, such as interfaces necessary for communications on physical links, and abstract RTOS necessary for scheduling tasks on a SW processor. New properties of components may also be required, as the allocation of functional components to physical elements implies a change of the nature of the functional components. For example, a function mapped to a SW processor becomes a RTOS task with its priority and stack size, and a link interface on a SW processor can be a SW driver or a HW DMA channel with their own properties, etc.

3.2.6 Implementation Oriented Model

Each implementation oriented model is specific to its domain and its purpose. There are as many models as targeted implementations. Two examples can be considered:

- A model for SystemC simulation
- A model for SW code generation (Java, C, C++, Ada, etc.)

© Copyright MARTES Consortium

3.3 UML Profile for MARTES

The UML Profile for MARTES defines a set of stereotypes for extending UML, as well as design practices to describe applications, process allocation, and platforms. It is especially targeted to automate system implementation using a UML2 description. A UML profile classifies different application components by defining various stereotypes and strict rules how to use them. The classification assigns defined parameters to proper components. The objective is also to enhance the support of external tools for automatic analyzing, profiling, and modifying the UML2 description of an embedded system.

The MARTES profile is directly related to certain OMG specifications. The relation of various RTES-related UML profiles is illustrated in Figure 13. The most obvious of these is the UML2 Superstructure specification, which is the basis for any UML profile. In addition, the MARTES profile is related to the following other OMG specifications:

- The UML profile for Systems Engineering (SysML), which deals with many of the same areas, such as the modeling of platforms and their constituent elements (hardware resources), and the allocation of software to platforms (i.e., deployment). In areas where there is conceptual overlap, the MARTES profile either reuses the corresponding SysML stereotypes, or defines elements that are conceptually and terminologically aligned with SysML.
- The UML for SoC (UML4SoC) profile and UML for SystemC profile. These specifications define a UML2 profile for System-on-a-Chip design by adding SoC description capabilities to UML.
- The UML for MARTE profile. MARTE is an in-progress OMG RFP specification which adds capabilities for modeling Real-Time and Embedded Systems (RTES), and for analyzing schedulability and performance properties of UML specifications. This new profile is intended to replace the existing UML

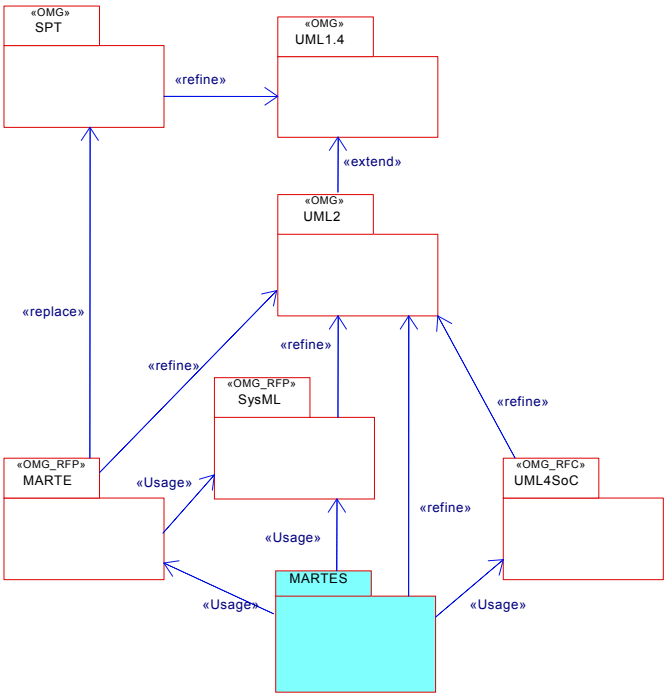


Figure 13: OMG RTES related standards.

© Copyright MARTES Consortium

Profile for Schedulability, Performance and Time (SPT). Many partners of the MARTES Project are involved in the proMarte consortium which is in charge to finalize a proposition for this upcoming standard for RTES modeling with UML.

The structure of the UML Profile for MARTES is presented in Figure 14 using the Martes Domain Specific Model terminology. An application is composed of application components, which are instantiated as application processes. Next, application processes are grouped into process groups. Correspondingly, an execution platform is composed of platform components, which are instantiated as platform component instances. Finally, process groups are mapped to platform component instances using an allocation mapping.

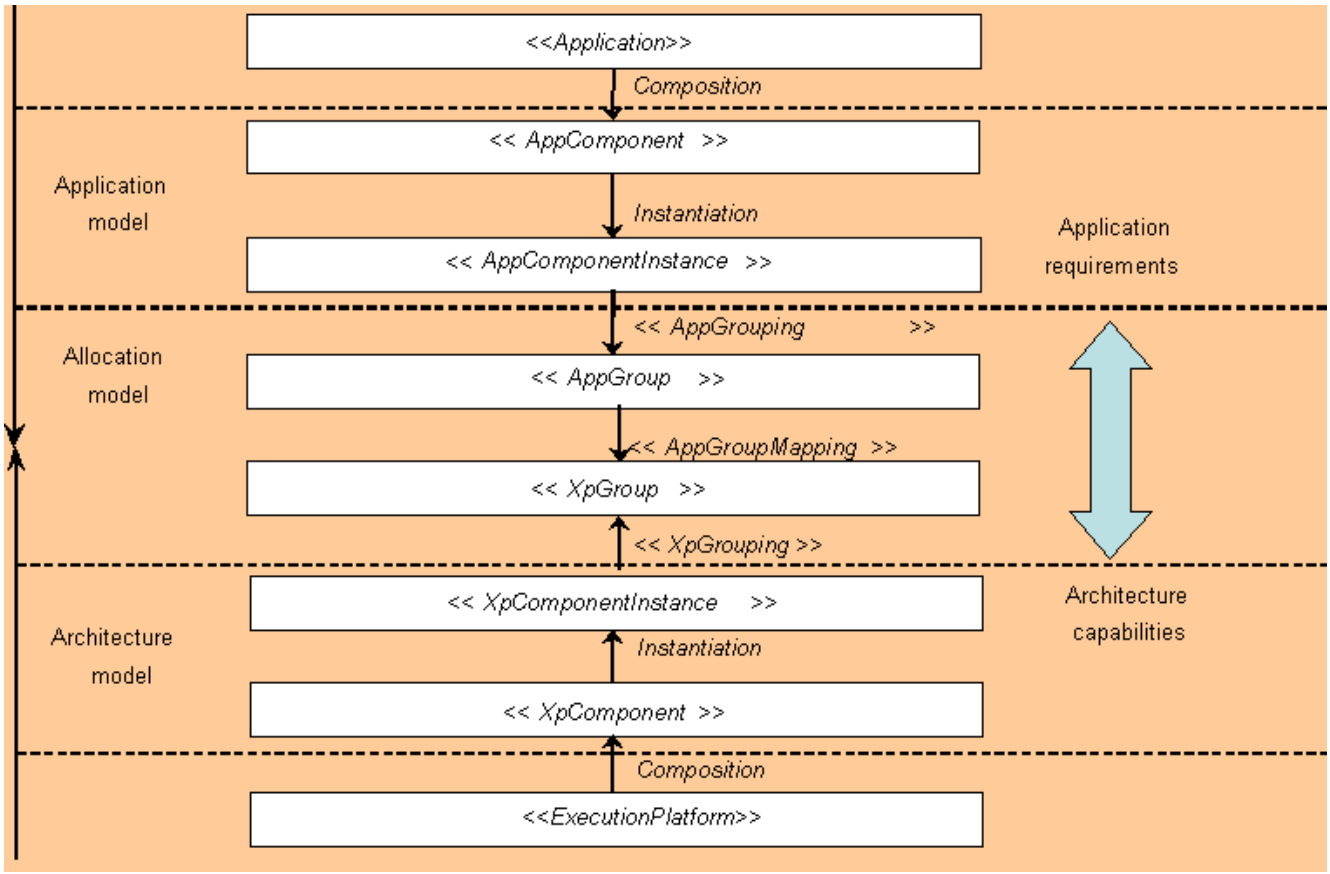


Figure 14: Structure of the MARTES profile.

3.3.1 Overview of the MARTES Profile

3.3.1.1 The MARTES Profile

The MARTES Profile extends the UML2 metamodel to enable the modeling of the structure and parameters of the application and platform, as well as the process allocation. Correspondingly, the system description is divided into three parts: application, platform, and allocation. The platform description is further divided

© Copyright MARTES Consortium

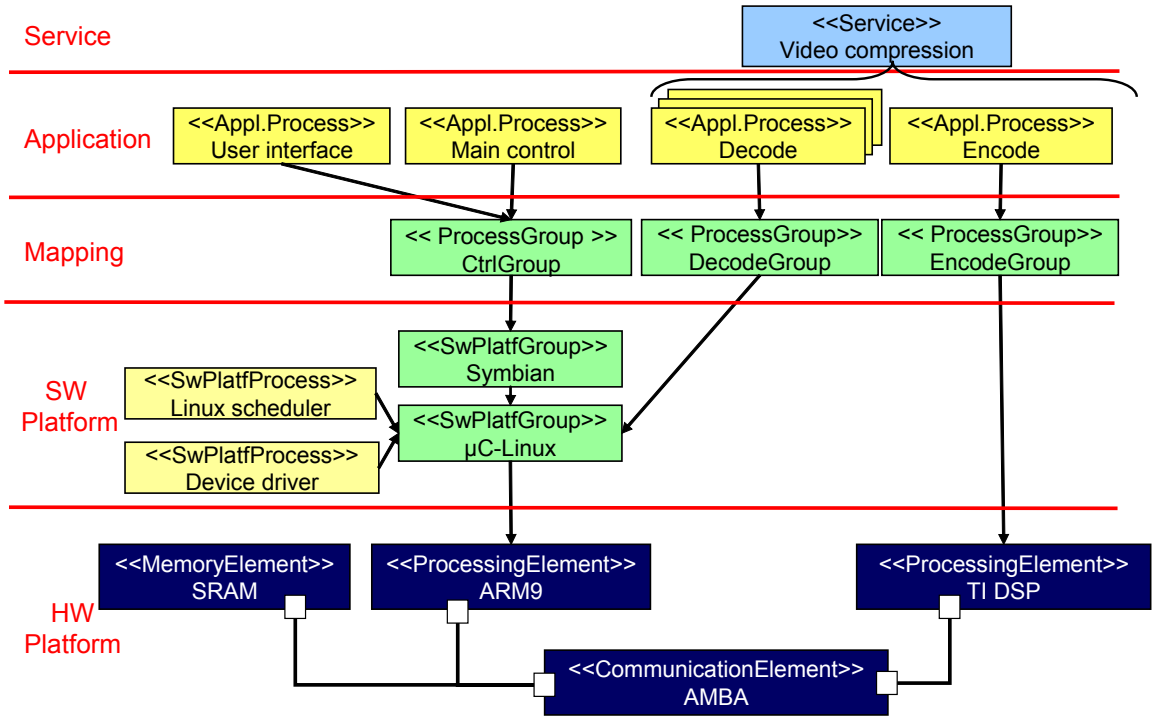
into hardware and software platforms. Specification of non-functional requirements is also possible. The designer can set limiting conditions for the system, for example maximum area. The design automation tools consider these requirements while constructing and optimizing the system. The realized values can be back-annotated to original UML models so that the designer can view how well they were met.

The following discussion shows all stereotypes of the MARTES Profile. It must be stressed that not all of them are necessary for efficient embedded system design. Nevertheless, the presented features *allow* very detailed modeling in UML but do *not necessitate* it.

3.3.1.2 MARTES Profile Description

The basic idea is so-called *Y-chart* system design: the application and platform models are designed independent of each other, and combined in an explicit allocation phase. The allocation of an application defines which Processing Elements (PE) executes certain application functionalities. In this way, exactly the same application description can be mapped onto various platforms, or its mapping on a certain platform can be explored efficiently. The application and platform can be composed of parameterized components taken from component libraries.

Figure 15 shows a simplified example of how the MARTES Profile can be used. However, the real system model will be actually defined using several separate models in order to handle the complexity. Basically, each horizontal slice in the figure is a separate model in UML. The purpose of the figure is to offer a high-level view of the Profile, and not to act as a specification. The upper part shows the behavioral model of the system (*application*). The execution platform is shown on the bottom. The connectors and groups in the middle define the mapping. The behavioral modeling starts with service-oriented system models. These models capture the use cases and high-level services that are needed to implement them. The service functionality is refined and described in the application modeling phase. The application processes are clustered into 3 groups in the example.

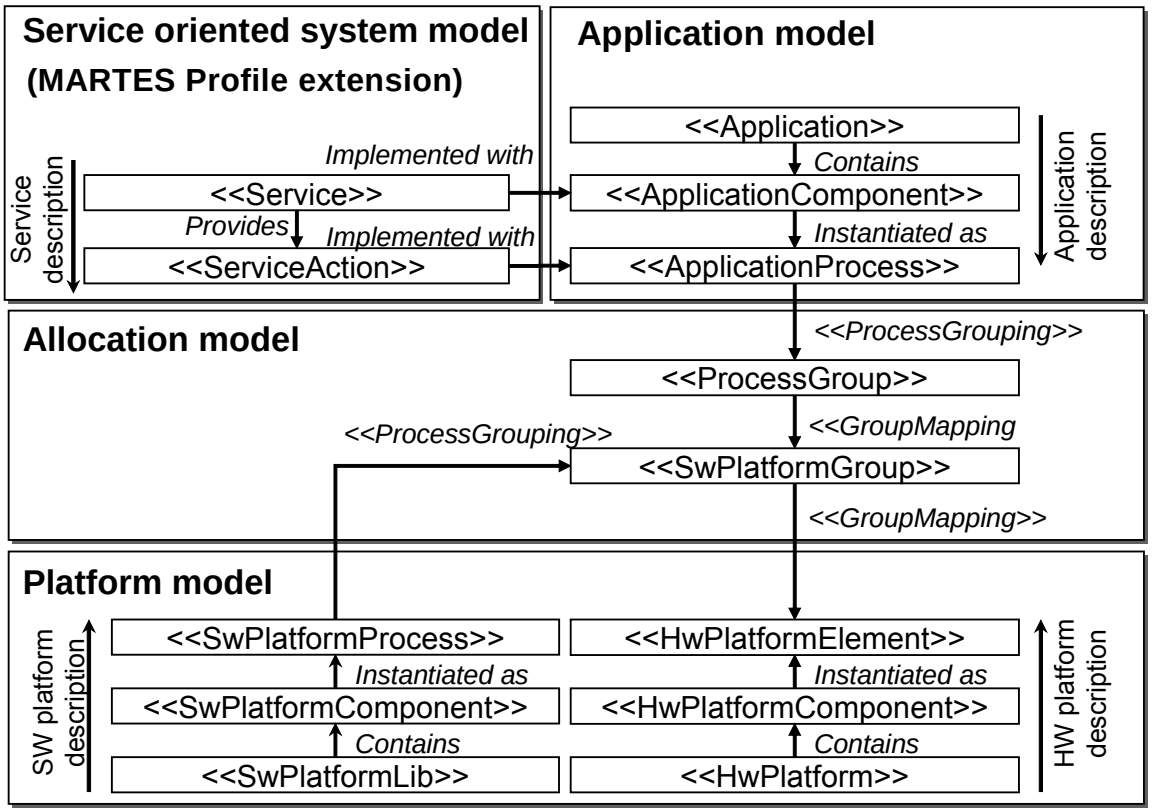


© Copyright MARTES Consortium

Figure 15: Example of using the MARTES profile.

The example HW platform has 2 CPUs, 1 memory unit and 1 bus. The ARM9 CPU has a SW platform that constitutes of 2 operating systems (OS). An operating system includes a scheduler and, in this case, few device drivers, as illustrated for uC-Linux. The lower OS, uC-Linux, offers real-time guarantees. It executes another OS in one thread and process group *decode* in another thread. Symbian OS, on the other hand, schedules the main control and user interface. The DSP executes only one encode process and, therefore, does not need any scheduler. Some process groups include only single process for clarity. In reality, there are likely several processes in every group.

The allocation model selects where the application process groups are executed. The leftmost group, *CtrlGroup*, which has two processes, runs on top of the Symbian OS on an ARM9 CPU. This means that Symbian schedules between *UserInterface* and *MainControl* processes. The middle group, *DecodeGroup*, runs on the same CPU but on top of µC-Linux. µC-Linux schedules between Symbian and all the processes in the DecodeGroup. The DSP executes only one process, hence, no scheduler is needed. However, a separate group is defined also in such case, in order to perform allocation in a consistent manner in every case.



The MARTES Profile focuses on describing the structure of the application and platform, and does not extend the behavioral modeling capabilities of UML. The basic behavioral description styles are naturally allowed, and contemporary design tools can generate executable code from some of them. To handle the complexity, design capture is performed in several steps, each showing only those components that are necessary at that point. Hence, not all stereotypes are accessible in all phases. For example, application

© Copyright MARTES Consortium

processes (or tasks) are present in the application description and its grouping, but not in the platform description nor allocation phases.

All the main stereotypes and their relations are depicted in Figure 16. Their details and design steps are discussed in the following sections.

3.3.2 Stereotypes of the UML MARTES Profile

3.3.2.1 Package Service

Service refers to a high-level capability of a device, for example WLAN access, encryption, or video encoding. Services communicate via messages. Services are aimed for reuse, and this necessitates their minimal dependence on the platform. Total independence, however, is sometimes impossible. For example, a service offering WLAN access requires that the HW platform includes appropriate radio, but does not assume anything about the type of CPUs needed. One or several services are needed to implement a single use case, but the same service can be used in multiple use cases. At first, the designer refines the use cases by identifying the needed services and their connections. One service can have multiple *action*s, for example, WLAN offers transmission and receptions actions. All actions of a service are not necessarily needed in every use case.

A service model may include a functional description, an API specification, and platform requirements. A functional description is optional in the first phase, as it can be described in the application modeling phase. At run-time, the service actions are called by *application nodes* or other services. Service oriented system modeling can be divided into three phases: *Use case requirements modeling, service specification modeling, and service activity modeling*.

- The use case requirements model is used to capture high-level use cases of the system with respect to its environment (user or other devices). This is performed with use case diagrams and possibly textual notations.
- The service specification model is used to define the high-level services that are required to implement the use cases. The service specification model also defines communication relationships between different services in the system. The communication can be both messages and data channels. The service specification model is described using activity diagrams, in which services appear as action nodes and their communication with activity edges.
- The *service activity model* is used to describe the co-operation of different services in system *activities,* whereas the service specification model defines the existence of the services as well as their static communication relationships. Activities are defined as flows of actions provided by the services. In an activity diagram, the *ServiceActions* are connected together with activity edges. The execution semantics of activities is based on a token flow model. Tokens flow from one action node to another via activity edges. There are separate tokens for control and data. An activity partition contains all the actions performed by one service. A system use cases can be defined as a single activity or as a set of multiple activities. Service actions are later refined with statechart descriptions. In addition, the activities and their relation to each other can be captured with interaction overview diagrams.

3.3.2.1.1 Profile Diagrams

Figure 17 presents an example of a top-level use case requirements model in a use case diagram. In the figure, the system (a video player) has six main use cases and two actors. One actor is the system user, and the other actor is a remote system that exchanges video data with the system.

© Copyright MARTES Consortium

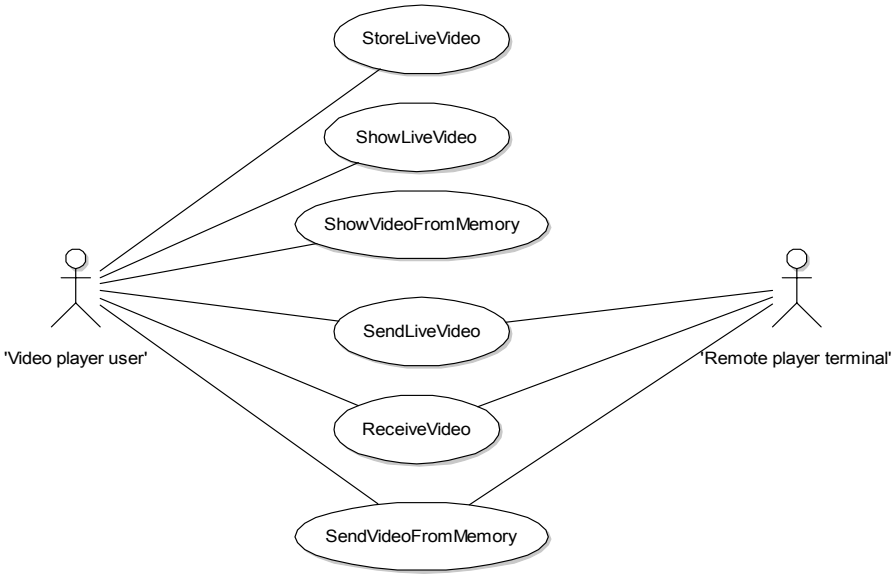


Figure 17: Example top-level use case requirements model.

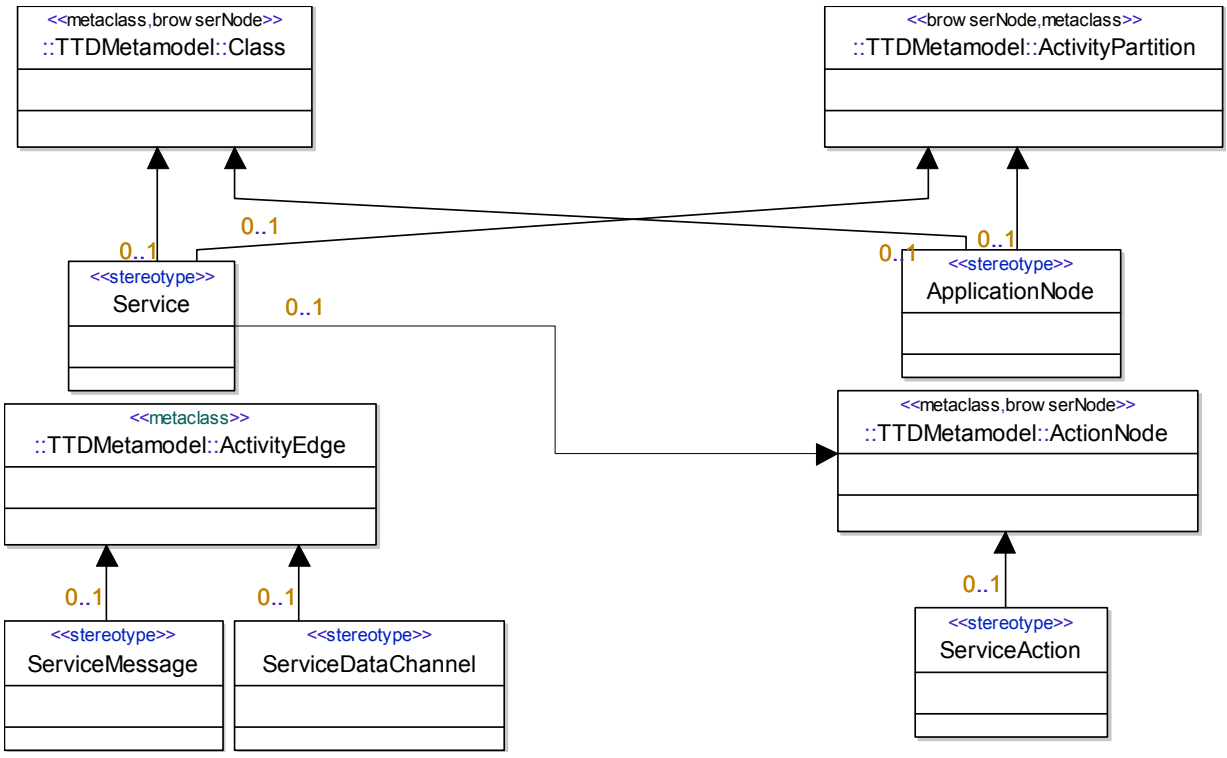


Figure 18: Stereotypes for service-oriented modeling.

© Copyright MARTES Consortium

Figure 18 presents the stereotypes used in service modeling. *Service*s are used to denote classes that implement a high-level capability of a system. The *Service* stereotype is given for the class itself, an action node in the service specification model (see Figure 19), and an activity partition in the activity diagram (see Figure 20). An *application node* is a similar kind of a high-level capability as a service, but it does not provide any service actions for other application nodes or services. Thus, they are usually application-specific control entities that call services to perform system activities. *ServiceMessage* and *ServiceDataChannel* stereotypes are used in activity edges between services in the service specification model.

Figure 19 presents an example of a service specification model. Altogether, 9 services and one application co-operate to implement the use cases. In addition, the data and message channels between services have been defined.

Figure 20 presents an example of modeling service activities. In the example, a single activity refining the "SendLiveVideo" use case is defined. The activity is implemented using a co-operation of four services: Camera, VideoEncoder, Tutmac_protocol, and radio. Each service is represented with a large horizontal rectangle. The activity itself is defined as a consecutive execution of four service actions, namely CaptureFrame, EncodeFrame, SendFrame, and SendRadioFrame. These actions are to be refined with application processes (statecharts) inside the class defining the service. The diagram may contain multiple actions for each service as well as branching and joining of edges. However, they are omitted from the example figure for clarity.

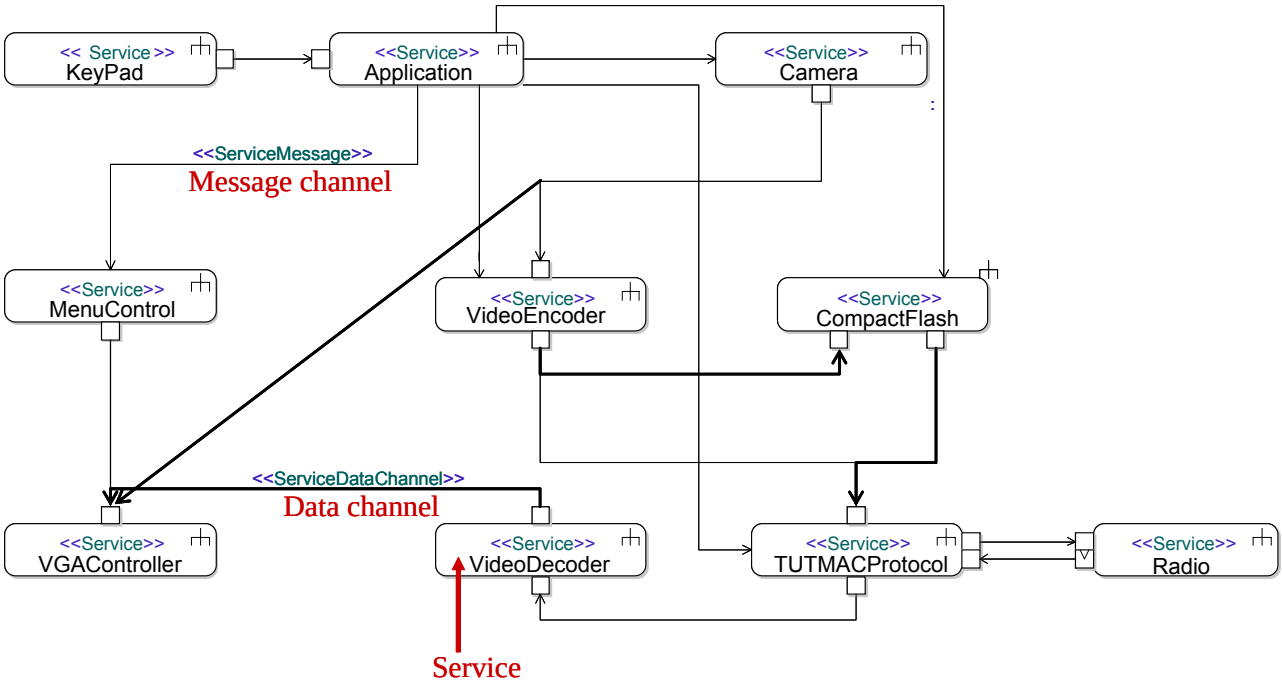


Figure 19: Example service specification model.

© Copyright MARTES Consortium

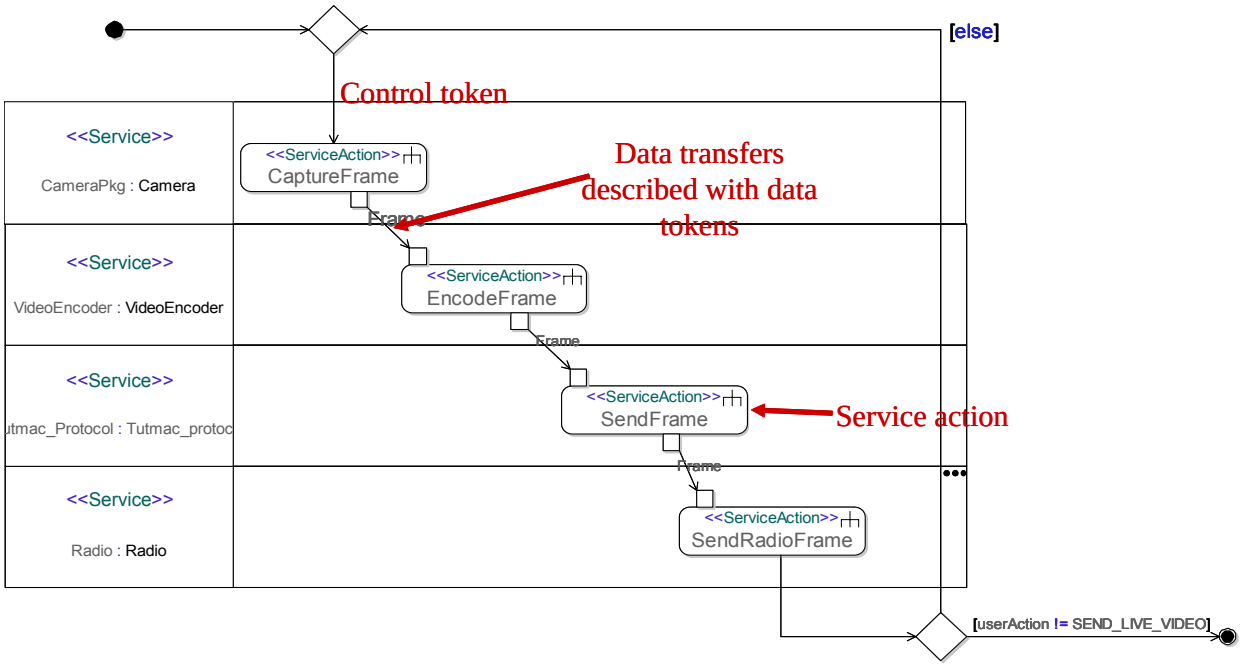


Figure 20: Example of modeling service activities.

3.3.2.1.2 Profile Element Description

The stereotypes needed for service oriented system modeling are listed in **Erreur ! Source du renvoi introuvable.**

Stereotype	Role	Generalization
Service	High-level capability of the system	Class, Activity Partition
ServiceAction	An action provided by a service.	Action node
ServiceMessage	Defines message relationship between services	Activity edge
ServiceDataChannel	Defines data channel relationship between services	Activity edge
ApplicationNode	High-level capability of the system that does not provide any service actions, but uses ones to perform its task.	Class, Activity Partition

© Copyright MARTES Consortium

Table 6: Descriptions for service oriented system modeling stereotypes.

3.3.2.2 Package Application

The application is seen as a set of active classes with an internal behavior. The application stereotypes focus on modeling the structure and parameters of an application. An application is described as a process graph, and processes communicate with signals that may carry arbitrarily complex data types. The functional description of a service contains one or multiple application processes. The profile does not restrict the behavioral modeling, and by default, it utilizes standard UML2 concepts for this. The behavior of the application components can be described, for instance, as standard UML or as external C code.

Signal passing is used to maximize the decoupling of processes, in contrast to communication via function calls. For this reason, the application processes do not have any “methods” (apart from basic constructors) when they are realized on a SoC. However, their actions depend on the type of the signal they receive.

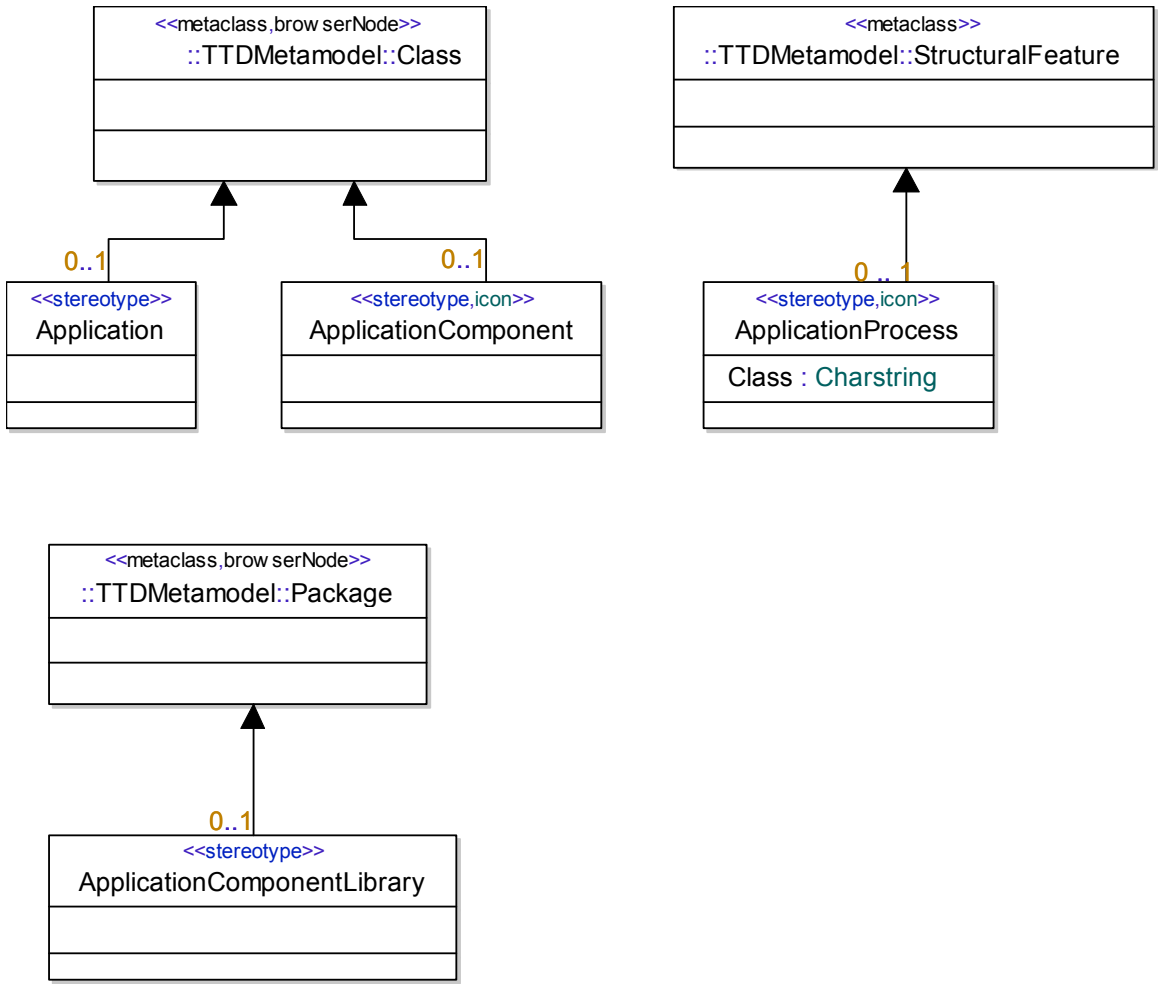


Figure 21: Stereotypes for application modeling.

© Copyright MARTES Consortium

3.3.2.2.1 Profile Diagrams

Figure 21 shows the stereotypes needed for application modeling. The *application library* stereotype defines a collection of available application components for reusing existing applications. The stereotype *application* is attached to the top-level class of the functional description, which defines the structure of the application process network by instantiating *application components* as *application processes*. The application model may contain hierarchical structures, but the *application process* stereotype is applied only on parts on the lowest level of the hierarchy.

An example of composite structure diagram used for application model is depicted in Figure 22. It consists of 10 entities. Some of them are application processes (for example, *Keypad* or *VgaController*), whereas some are hierarchical having several processes (for example, *VideoEncoder* and *Tutmac_Protocol*). In both cases, the parts have one or multiple ports that used for communication. The arrows between ports define which parts interact and the direction of the flow.

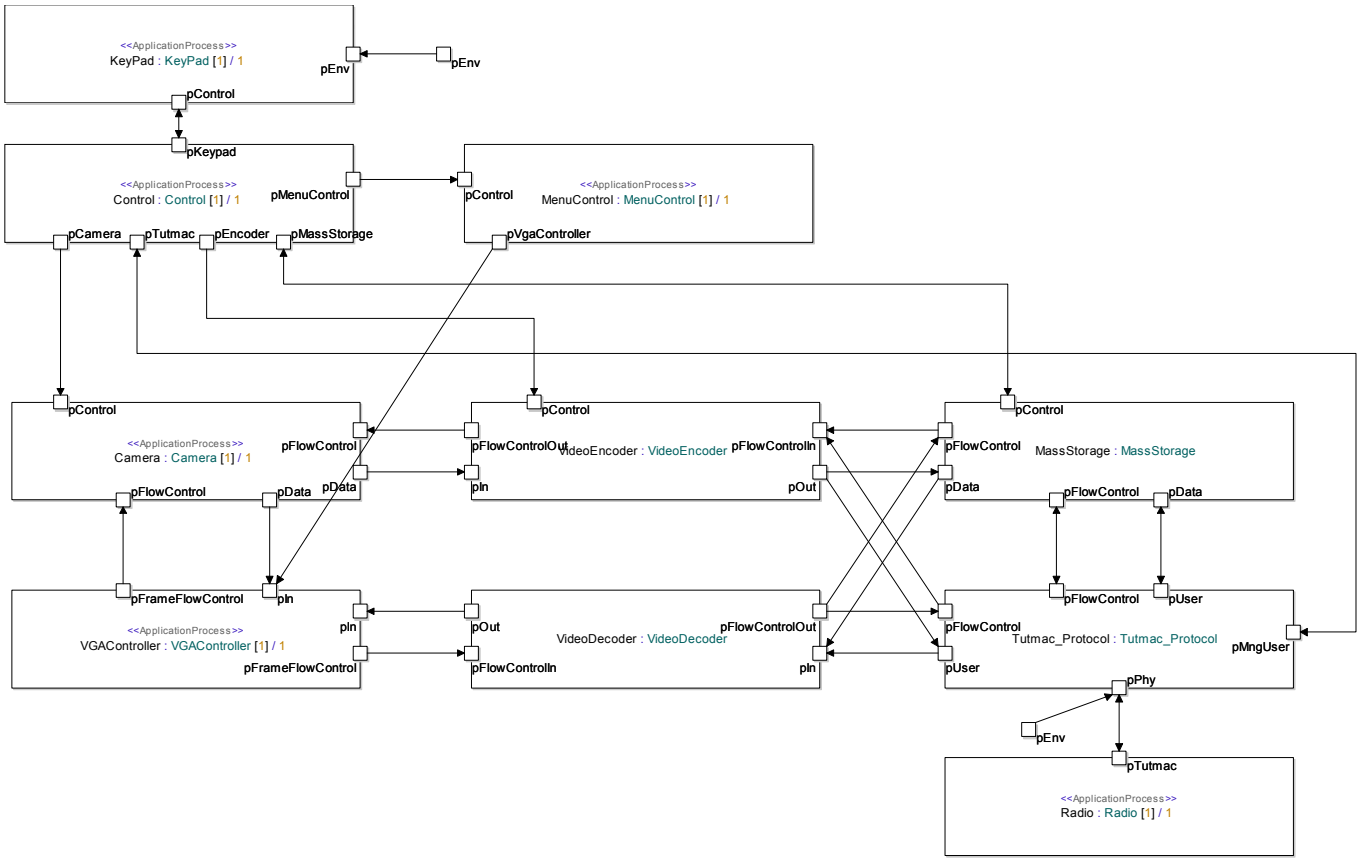


Figure 22: Example of structural application modeling using a composite diagram.

Figure 23 shows an example of behavioral modeling using statecharts. The black dot on top denotes initialization node. Other elements are states (e.g., *idle* and *waitack*), signal reception and sending, actions, assignment statements, and conditional branching. For example, the commercial Telelogic Tau 2 tool can generate C code from such models. It is also possible to call external C functions from statechart

© Copyright MARTES Consortium

description, which allows the reuse of legacy code. By wrapping the code calls in UML, seamless interaction between application processes is allowed, no matter how their behavior is captured (pure UML, external C functions, etc.).

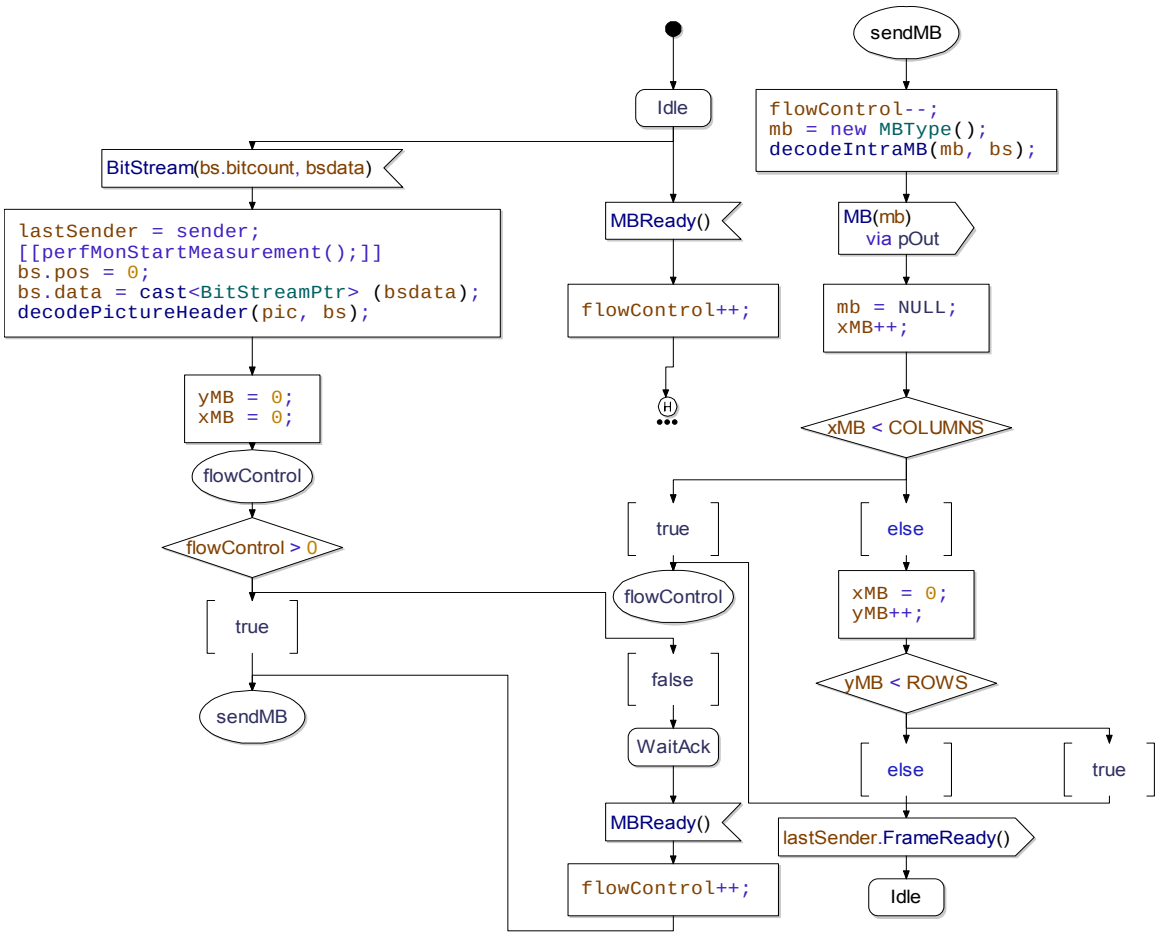


Figure 23: Example of behavioral application modeling with statecharts.

3.3.2.2.2 Profile Elements Description

The stereotypes needed for application modeling are listed in Table 7

© Copyright MARTES Consortium

Stereotype	Role	Generalization
Application	Top-level class of the application.	Class
ApplicationComponent	Stereotype for a class describing application functionality	Class
ApplicationProcess	Instantiated application component	Part
ApplicationLibrary	A package containing a set of application components	Package

Table 7: Descriptions for application stereotypes.

3.3.2.3 Package Execution Platform

The execution platform model includes all the information (capabilities and limitations) required to the accurate high-level hardware/software co-simulation or model development towards implementation. The platform model consists of the *hardware* and *software platform* instances. They are combined in separate allocation phase (described in section 0).

This model describes the execution platform as a composition of the various SW and HW components defined in a library. The software platform means the SW that is distinct from the application, for example, operating system (OS) and device drivers. In the HW platform model, some components (processor, FIFO, memory, ...) are instantiated, deployed, and interconnected to depict a consistent representation of the HW platform. The HW platform can be, for example, a single chip with numerous IP cores, a number of chips connected via onboard bus, or a motherboard communicating with one or several daughterboards via one or several connectors.

3.3.2.3.1 Profile Diagrams

The platform elements can be represented by a structure diagram, in which the platform components are represented using UML2 component representation. They are interconnected via links between the ports.

Hardware Platform

The HW platform instance is an abstract presentation of HW components and their interconnections. As shown in Figure 24, a hardware platform contains all the *HW platform components* available for the selected platform. When HW components are instantiated as parts, their stereotype becomes *element*, for example, *processing element* (PE). *Component* defines which attributes are present in the metaclass, whereas *element* defines values for those that are common for all instances. Separate stereotypes *component* and *element* are needed to allow instance-specific attribute values, for example, size in kilobytes in case of a memory component.

Hardware components are further divided into three subcategories: *communication components*, *memory components* and *processing components*, which all are inherited from the base stereotype *hardware platform component*. The communication elements interconnect other types of elements, e.g., a bus

© Copyright MARTES Consortium

segment or mesh. Memory elements allow mapping of various code actions. The generic stereotype *HW platform element* is used for those that do not belong to any of the aforementioned 3 subcategories.

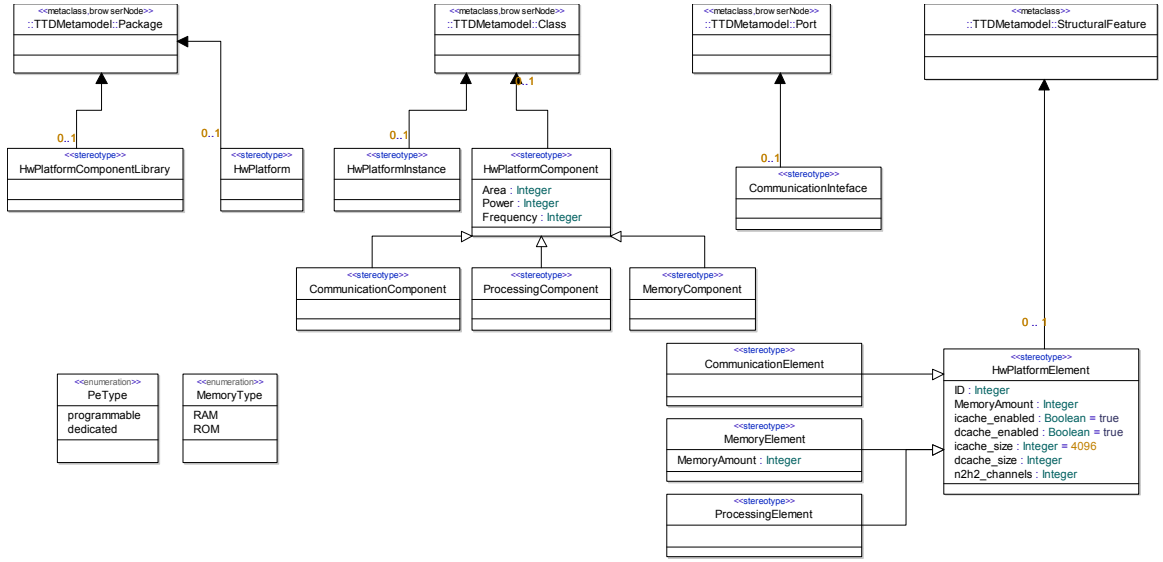


Figure 24: Stereotypes for HW platform modeling.

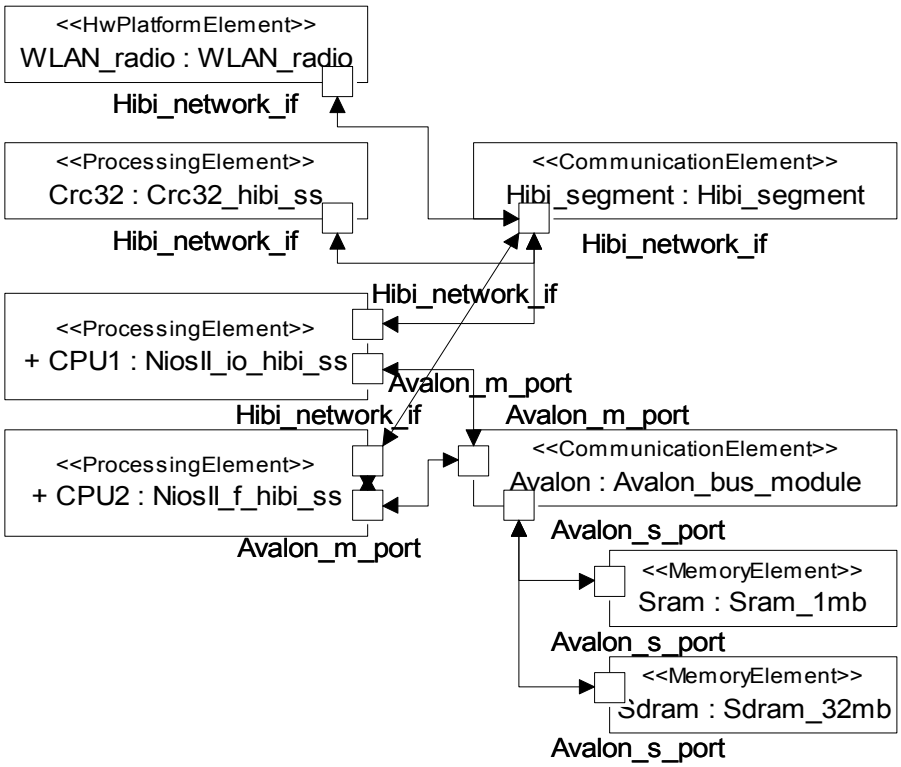


Figure 25: Example for HW platform modeling.

© Copyright MARTES Consortium

A PE represents an entity that can execute application processes. A PE can be a CPU, DSP, or hardwired accelerator. General-purpose PEs, such as CPUs, can also execute SW platform functionality if needed. SW platform will be introduced further on in more detail.

An example of a hardware platform instance is given in Figure 25. It has two Nios II processors, a hardware accelerator for 32-bit Cyclic Redundancy Check (CRC), interfaces to external Wireless Local Area Network (WLAN) radio and external memories. There are 3 PEs, 2 communication elements, and 2 memories. The WLAN_radio is modeled as (generic) *HWPlatformElement*. The components are connected using an Avalon bus module and a custom HIBI communication segment.

PEs can be composed of other components and internal structure is described in another view. The hierarchical modeling of HW components promotes component re-use and helps managing the design complexity. It also simplifies reusing larger entities such as subsystems with several processors. In this way, the designer has very detailed control over PE parameters, for example, the amount and type of local memories. However, this is not necessary in all cases. In many cases, the internal structure can be defined only in implementation model, such as VHDL, and all necessary settings are given to the high-level UML model as attributes. Furthermore, an automated design space exploration tool included can take care of

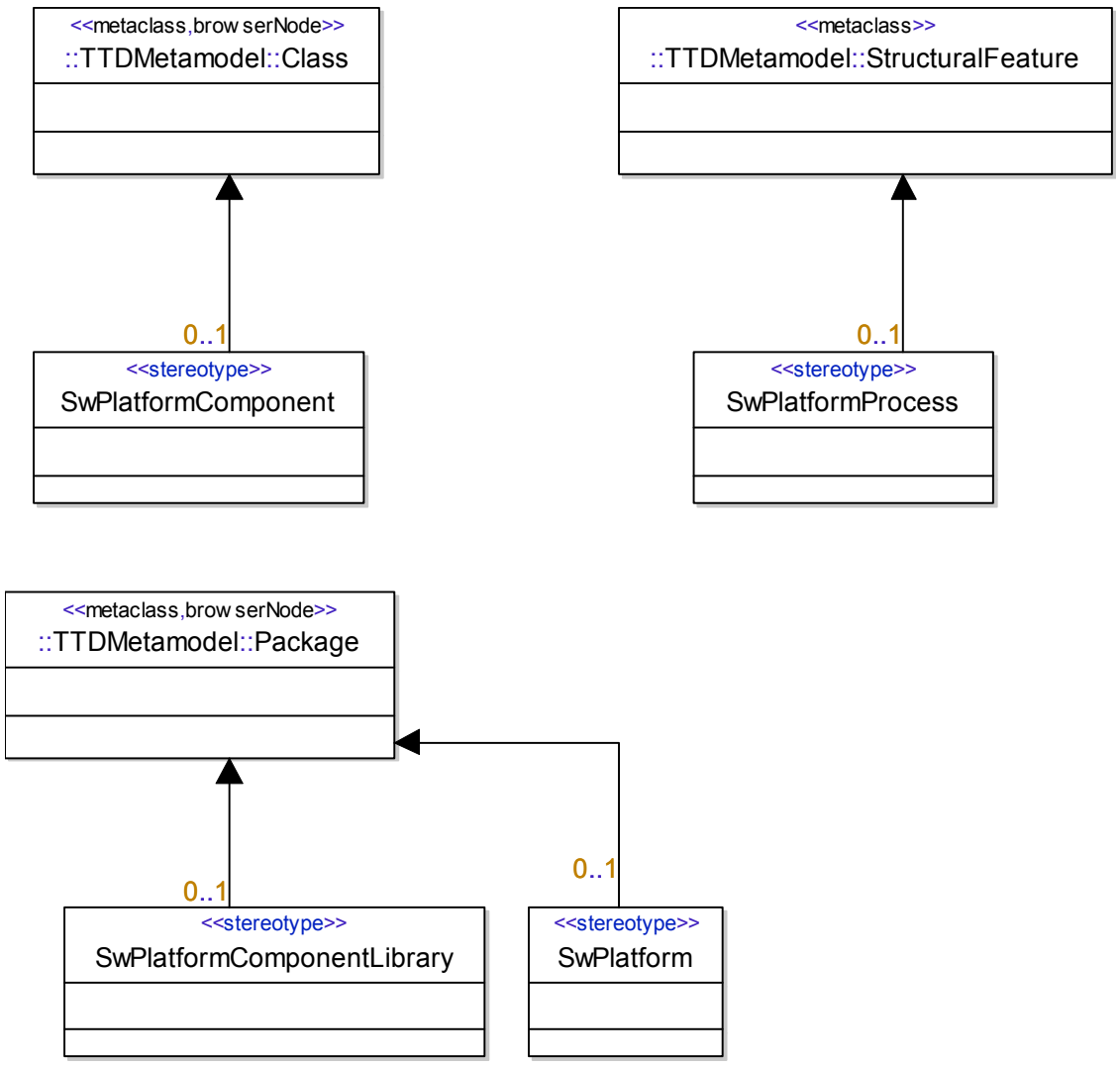


Figure 26: Stereotypes for SW platform modeling.

© Copyright MARTES Consortium

selecting appropriate parameter values. Hence, system designer can, in most cases, ignore the exact structure of the PE when using it. Third-party components are usually delivered in a format that does not reveal the internal details.

Software platform

In the software platform model, the software platform can be composed and parameterized according to the applications requirements. A SW platform includes functionality on processors which cannot be included into the application, such as RTOS scheduler, device drivers, middleware components, and the HW abstraction layer.

The stereotypes for the SW platform are shown in Figure 26. The software platform contains a collection of available *component*s which are instantiated as *SW platform process*es. A *SW platform process group* is defined by clustering software platform processes together (more information will be given in the allocation description in section 0). Separate stereotypes are used for SW platform processes and application processes in order to clarify allocation models. Different group stereotypes are also used for application and software platform process groups. This is necessary to allow mapping of multiple instances of software platform groups to different processors, whereas application groups are always mapped only once.

Figure 27 shows an example of grouping of a software platform. In this example, the software platform group is composed of four components: a scheduler, support for UML signal passing, and two device drivers.

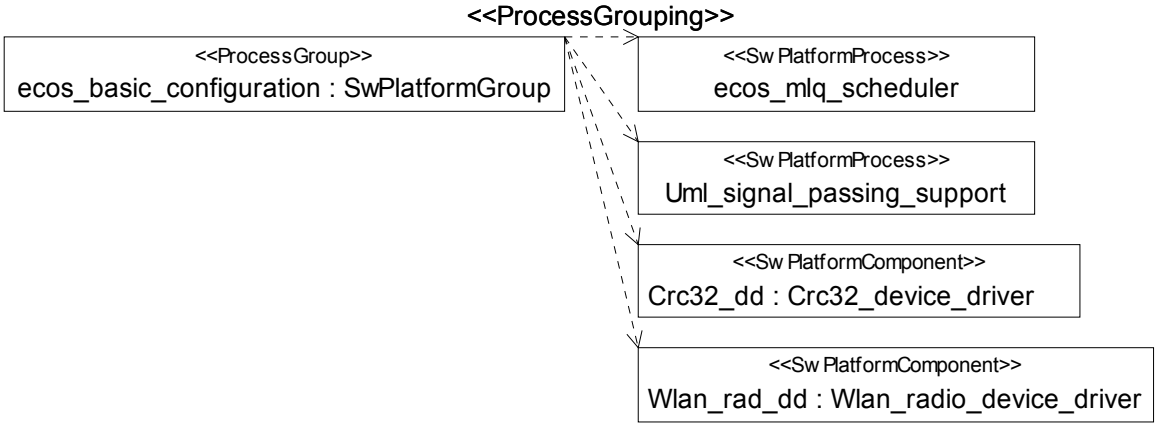


Figure 27: Example of SW platform modeling.

3.3.2.3.2 Profile Elements Description

The stereotypes needed for execution platform modeling are listed in Table 8.

© Copyright MARTES Consortium

Stereotype	Role	Generalization
HwPlatformLibrary	Includes all available HW components	Package
HwPlatform	Subset of platform library	Package
HwPlatformInstance	Instance of previous	Class
HwPlatformComponent	Representation of a HW Component	Class
HwPlatformElement	Instance of previous	Part
ProcessingComponent	A specific kind of HW Component	Class
ProcessingElement	Instance of previous	Part
CommunicationComponent	Connects other HW components	Class
CommunicationElement	Instance of previous	Part
MemoryComponent	A specific kind of HW Component	Class
MemoryElement	Instance of previous	Part
SwPlatformLib	Includes all available SW platform components	Package
SwPlatform	Subset of platform library	Package
SwPlatformProcessGroup	Clustering of SW processes	Part
SwPlatformComponent	Element of a SW platform	Class
SwPlatformProcess	Instance of SW component	Part

Table 8: Descriptions for execution platform stereotypes.

3.3.2.4 Package Allocation

An allocation binds application and execution platforms together. It defines the location where application and SW platform processes are executed. In addition, it defines to which memory components the distinct memory sections (code vs. data) are located.

© Copyright MARTES Consortium

In the first phase of an application allocation, the application processes are clustered to form *process groups* based on several criteria, such as shared resources, heavy interaction, or real-time requirements. Processes within one group will be executed on a single processing element, yet several groups can be mapped onto a single processing element. Grouping is an efficient way to reduce the vast design space.

An allocation also defines how the SW platform and HW platform interact, for example, which PEs have an operating system. Again, processes are clustered together into SW platform process groups. One SW platform process group constitutes a single SW platform instance that is allocated to a PE. The same SW platform group can be allocated onto multiple PEs, unlike the groups of application processes. This means that each PE executes its own *copy* of a single SW platform.

One of main purposes of the SW platform is to schedule the application processes, and to allow seamless communication between them. Therefore, the application process groups are, in most cases, allocated on top of a SW platform. Is it also possible to allocate one SW platform on top another, for example, a regular OS running as one thread of real-time OS. Each SW platform performs scheduling between those groups that are allocated to it. This is a very intuitive description style, and an example is given later.

One could also consider allocating several groups directly to a PE (in the HW model). However, several problems arise when there is no SW platform at all, or if there are multiple parallel ones. In both cases, there is no direct way to decide how to perform the scheduling. Therefore, this option was discarded.

There are, however, cases where the SW platform is omitted from a PE. The most obvious is with HW accelerators that cannot perform general-purpose processing. Another example could be when the application codes are hand-written (from scratch or reused legacy code, possibly in assembly). This is efficient when PE executes only single process. If there are several processes, a SW platform is needed.

3.3.2.4.1 Profile Diagrams

The stereotype relations for allocation modeling are shown in Table 9. There are two kinds of groups: one for the application, and another for the SW Platform. Separate stereotypes are needed due to clear semantical differences. A process group of an application can be executed in one place only (directly on a PE, or on top of a SW platform). On the other hand, SwPlatformProcessGroup can be mapped to several places, which creates separate copies of it. Designer can define distinct configurations of the SW platform, for example, basic eCos, eCos+TCP/IP, or uCLinux, and allocate those several times. In contrast, the application mapping is aimed to *distribute* the execution, and not to *replicate* it.

However, the same grouping stereotype is used in both cases for grouping. This is true also for group allocation (called GroupMapping in the MARTES Profile). A designer can assign priorities to application groups in case they are implemented as OS threads.

© Copyright MARTES Consortium

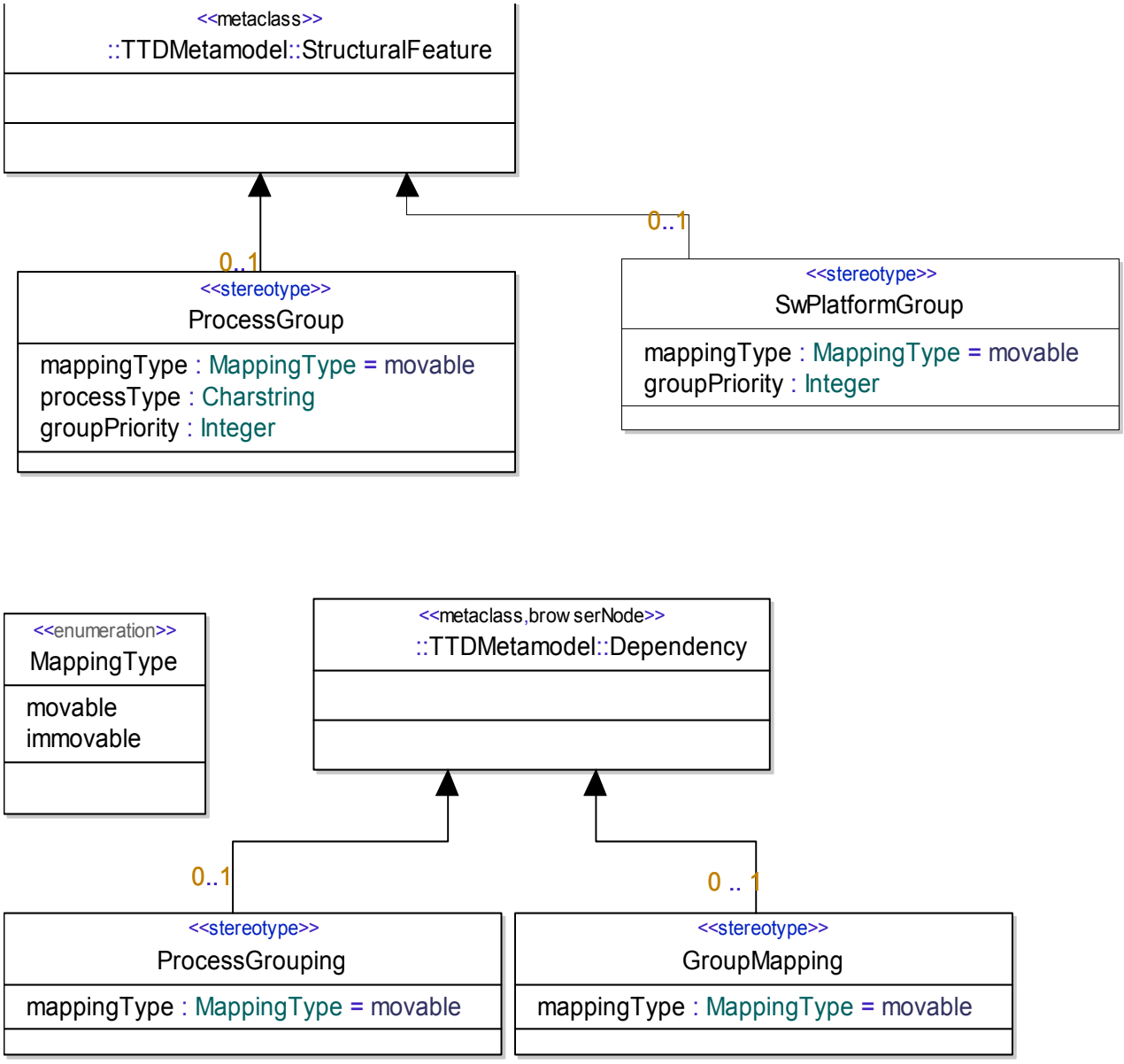


Figure 28: Stereotypes for grouping and allocation modeling.

Allowing all application processes to be moved freely during the exploration, creates impractically large number of possible allocations. However, it can be reduced by orders of magnitude, if the designer is able to come up to with initial grouping and allocation, and fixing certain properties. Groups allow restricting the design space (allowed moves) during the exploration. Grouping and mapping have specific attributes for this purpose. Attribute *movable* defines if a process can be moved to another group or if a group can be moved to another place. In addition, the group can be defined as *immutable*, which means that its contents cannot be modified. This setting overrides those of the processes inside the group. Furthermore, a PE (and

© Copyright MARTES Consortium

a SWPlatformProcessGroup) have a similar *mutable/immutable* attribute. The meaning is whether the groups can be moved in or out. This overrides the settings of a single group.

The stereotypes for memory allocation are shown in Figure 29. There are three different types of memory sections: *code* (read-only), *data* (read-and-modify, initialized) and *Bss* (read-and-modify, uninitialized).

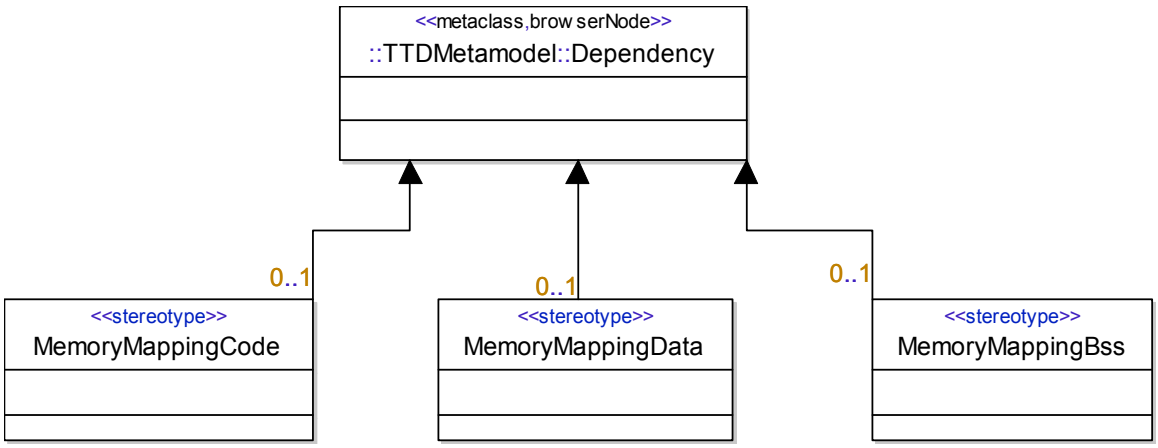


Figure 29: Stereotypes for memory allocation modeling.

3.3.2.4.2 Profile Elements Description

The stereotypes for allocation modeling are listed in Table 9.

Stereotype	Role	Generalization
ProcessGroup	A group of application processes	Struct. feature
ProcessGrouping	Attaches a process (either application or SW platform) to a group	Depencency
GroupMapping	Maps a group onto a PE (or onto other group)	Dependency

Table 9: Descriptions for allocation stereotypes.

An example of group mapping is given in Figure 30. Three application process groups are mapped to Pes. Both Pes have only a single SW platform. They use different SW platform configurations which both are based on eCos. An example of memory allocation is shown in Figure 31. The memory sections of one CPU are mapped to two memory banks.

© Copyright MARTES Consortium

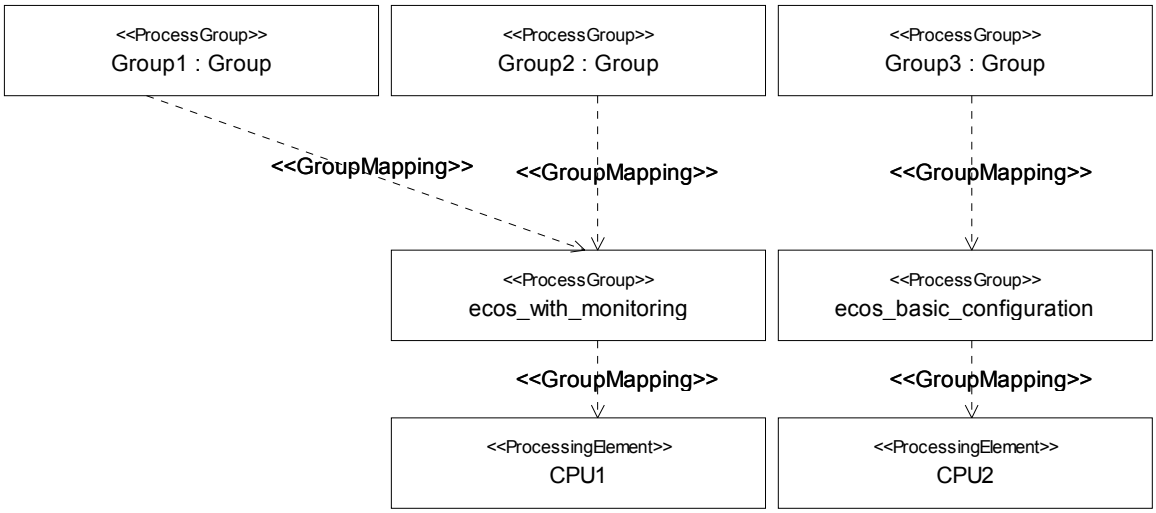


Figure 30: Example of allocation model.

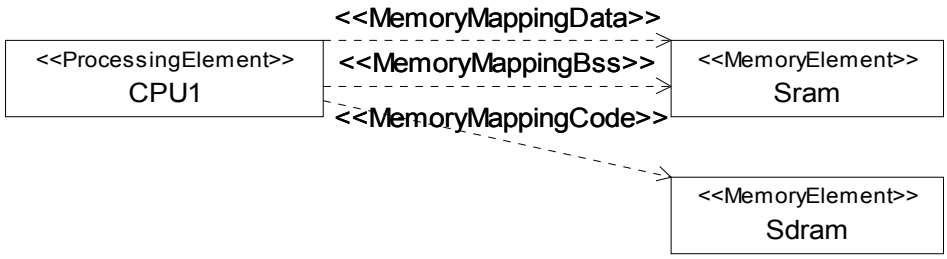


Figure 31: Example of memory allocation model.

3.3.2.5 Package Expression

Expressions are used for setting non-functional restrictions.

3.3.2.5.1 Profile Diagrams

The utilized stereotypes are shown in Figure 32. Currently, all of them are related to timing. A designer selects points from the functional description (application) for timing analysis. All actions between the start and the end of the sequence are considered as a whole. For example, a designer can set requirements (‘*req*’) for certain state transition sequences. The measured (‘*msr*’) times can be back-annotated to the original UML model which allows comparison between requirements and realization.

© Copyright MARTES Consortium

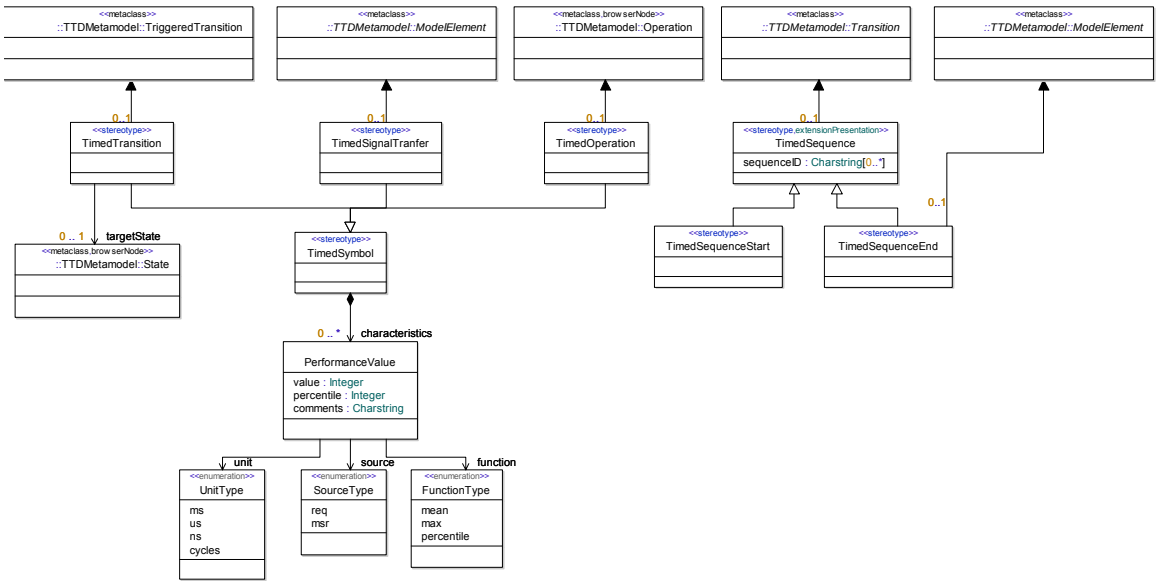


Figure 32: Stereotypes constraint definition.

3.3.3 Performance Modeling and Evaluation

The Performance Modeling and Evaluation approach exploits the UML Profile for MARTES, but provides performance specialization through defining specific stereotypes and/or attributes for workload modeling and execution platform modeling. In addition, specific service interfaces are defined for linking hierarchical workload models with hierarchical execution platform models.

At the back-end, the transaction-level SystemC performance simulation model of the system is achieved by linking the SystemC representation of the workload model, generated from the UML representation of the workload model, with the SystemC representation of the execution platform model.

The Performance Modeling and Evaluation approach is depicted in Figure 33. The approach follows the specify-explore-refine paradigm of the Y-chart model. The workload model is an abstracted model of the application that is allocated and mapped onto a model of the execution platform for executing the transaction-level performance simulation of the allocated model. The goal is to separate the issues related to the application development from those of the platform development, while keeping them interoperable [xxii].

© Copyright MARTES Consortium

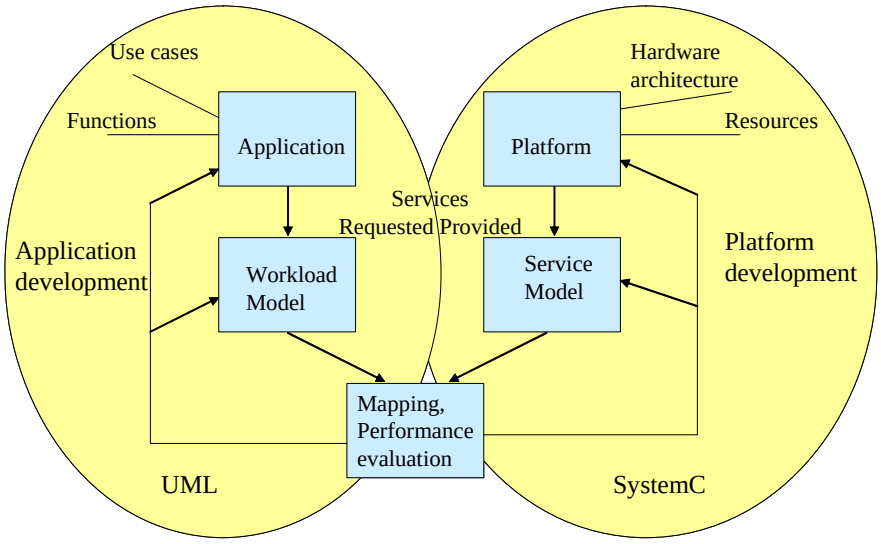


Figure 33: MARTES performance modeling and evaluation approach.

3.3.3.1 Models Overview

The layered, hierarchical models for the workload and platform modeling [xxiii] are presented in more detail in Figure 34, which also shows the linkage between the workload and application models. The application modeling is divided in abstract and refined use cases, and functional architecture models that are described in terms of services that the application requires from the platform when executed. The workload model is structured into layers, reflecting the hierarchy needed in the performance simulations. The arrows depict the main sources and routes of load information for workload models. The application and workload models are presented in UML.

The execution platform model is also hierarchical. The topmost layer, called platform, is composed of a number of sub-systems needed for the (envisaged) set of applications. The sub-systems are interconnected using communication network (e.g., switches and/or routers). The sub-systems themselves are computers composed of basic hardware (e.g., processor, memory, accelerator, and bus) and software (e.g., OS, driver, and service task) components. Each sub-system offers domain-specific services, as well as more generic services for running application processes implemented as embedded software tasks. The performance simulation models of the execution platform are presented in SystemC.

© Copyright MARTES Consortium

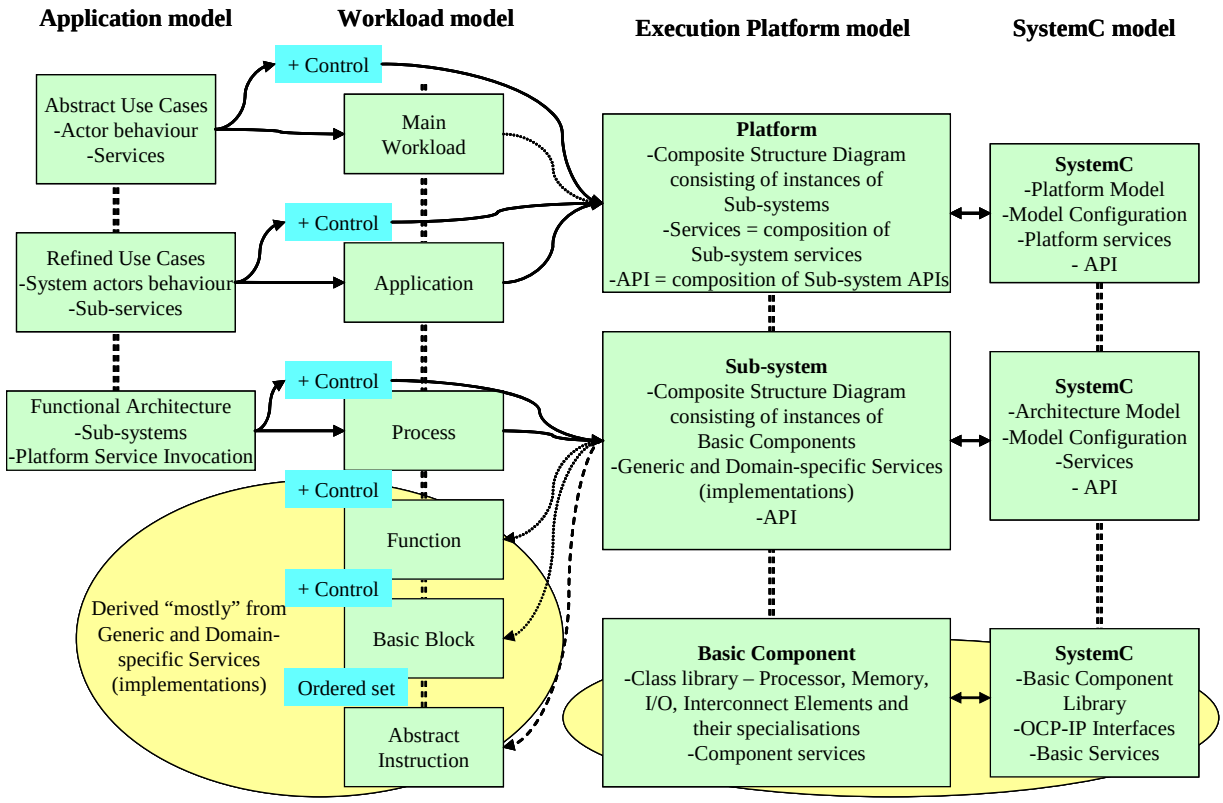


Figure 34: Overview of model structures for performance modeling and evaluation.

3.3.3.2 Application Model

The application model is created following the service-oriented modeling approach described in Chapter 2. At the Requirements and Specification modeling, it uses standard UML2 modeling techniques, i.e., use case diagrams, state machine diagrams, sequence diagrams, and composite structure diagrams. At the functional modeling of an application, the Package Application stereotypes of the UML Profile for MARTES are used.

3.3.3.3 Workload Model

As explained in section 2.2.3, the purpose of the workload model is to present computing and communication resource loads that applications and services used by the applications cause to the execution platform resources. Workload models are created from functional and execution models of the application in order to facilitate fast, light effort performance evaluation at system level. Workload models present resource requirements of functional and execution models, in terms of computing and communication loads that exercise platform resources. Computing and communication loads are obtained from functional and execution models through estimation techniques. Figure 35 shows the workload meta-model that is divided hierarchically into main, application, process, function, basic block, and abstract instruction layers.

© Copyright MARTES Consortium

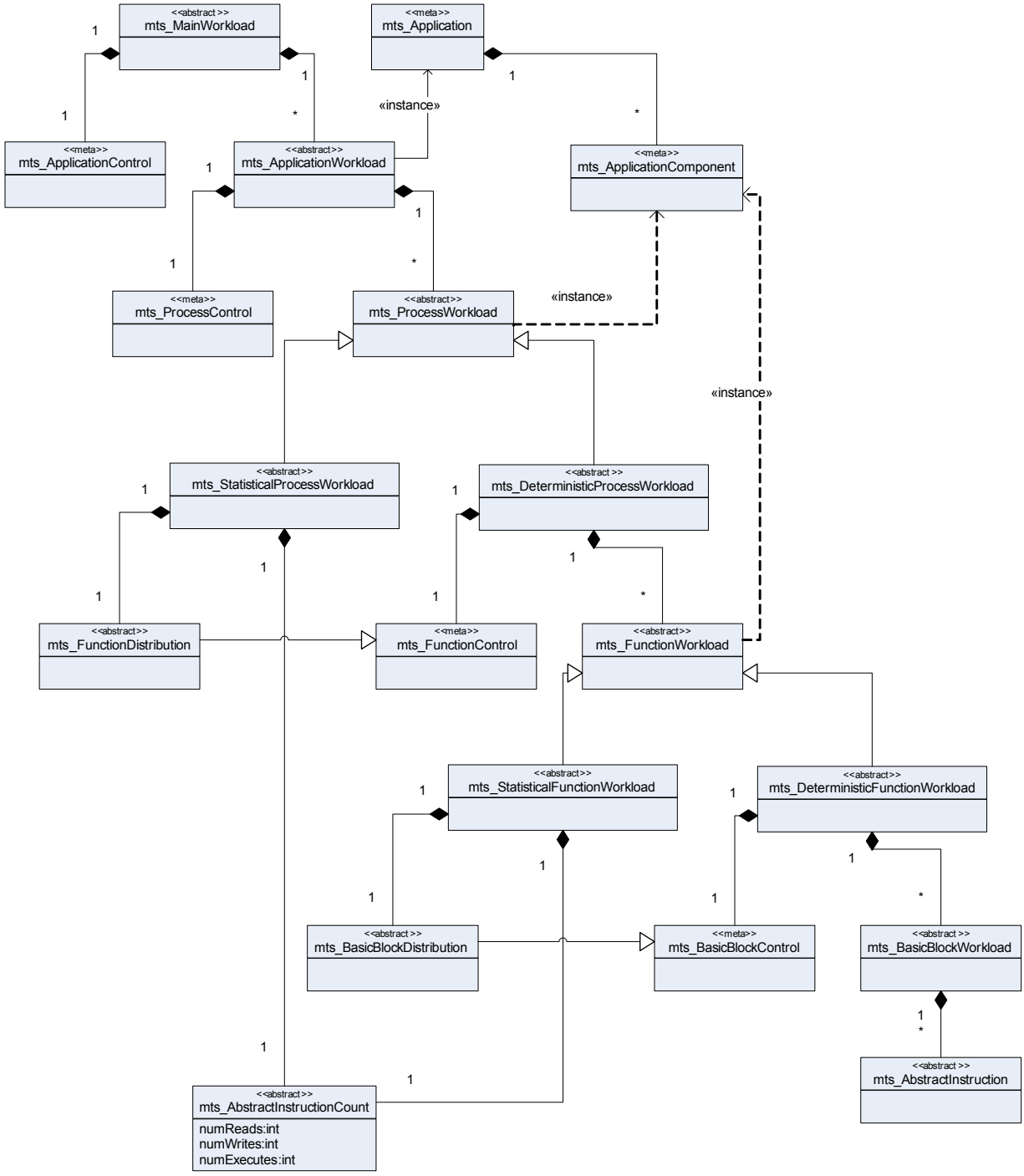


Figure 35: Workload meta-model.

The workload stereotypes are listed in Table 10. They concern the structure and load primitives of the workload model, which represents the computing and communication loads that the modeled applications cause to the underlying execution platform. The control is modeled using standard UML2 techniques, e.g., state machine diagrams.

© Copyright MARTES Consortium

Workload stereotype name	Description
MainWorkload	Main workload model W divides into application workloads Ai: $W = \{C_A, A_1, A_2, \ldots, A_n\}$
ApplicationControl	C_A denotes the common control between the workloads
ApplicationWorkload	Each application workload Ai is constructed of one or more processes Pi: $A_i = \{C_P, P_1, P_2, \ldots, P_n\}$
ProcessControl	C_P corresponds to the control between the processes
ProcessWorkload	Processes are comprised of function workloads F_i: $P_i = \{C_F, F_1, F_2, \ldots, F_n\}$,
StatisticalProcessWorkload	Statistical process workload describes the total number of different types of load primitives
FunctionDistribution	Control of statistical distribution for the primitives
AbstractInstructionCount	Number of load primitives
DeterministicProcessWorkload	Deterministic process workload
FunctionControl	CF is control and describes the relations of the functions, e.g. branches and loops
FunctionWorkload	Control flow graphs $F_i = (V, G)$, where nodes $v_k \in V$ are basic blocks, and arcs $g_k \in G$ are branches.
StatisticalFunctionWorkload	Statistical function workload
BasicBlockDistribution	Control of statistical distribution for the primitives
DeterministicFunctionWorkload	Deterministic function workload
BasicBlockControl	Control of load primitives
BasicBlockWorkload	Ordered set of load primitives
AbstractInstruction	Load primitive for load characterization

Table 10: Workload stereotypes.

3.3.3.4 Execution Platform Model

The Execution Platform model, as depicted in Figure 36, is an abstracted hierarchical representation of the actual platform architecture. The Package Execution Platform stereotypes of the UML Profile for MARTES are used for modeling, but performance related specializations are introduced. Interfaces for linking workload models to execution platform models are also included.

© Copyright MARTES Consortium

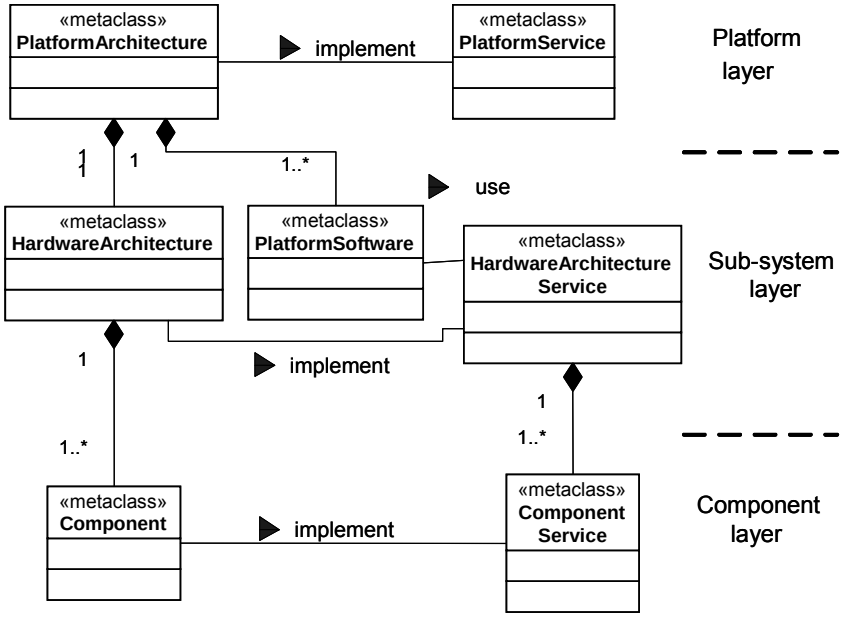


Figure 36: ExecutionPlatform meta-model

3.3.3.4.1 Component Layer

Processor models and all derived models of the component layer provide a low-level interface, through which workload models can send load primitives to the execution platform model. The functions of the interface are described in Table 11. Subsystem and platform level services are built upon these interfaces.

Interface function	Description
read(*address*, *words*, *word size*)	Read *words* * *word_size* bits from address.
write(*address*, *words*, *word_size*)	Write *words* * *word_size* bits to address.
execute(*n*)	Execute *n* data processing or control instructions.
set_pc(*address*)	Set program counter to *address*.

Table 11: Interface for transferring load information to the execution platform model.

Figure 37 and Table 12 lists the component layer performance attributes.

© Copyright MARTES Consortium

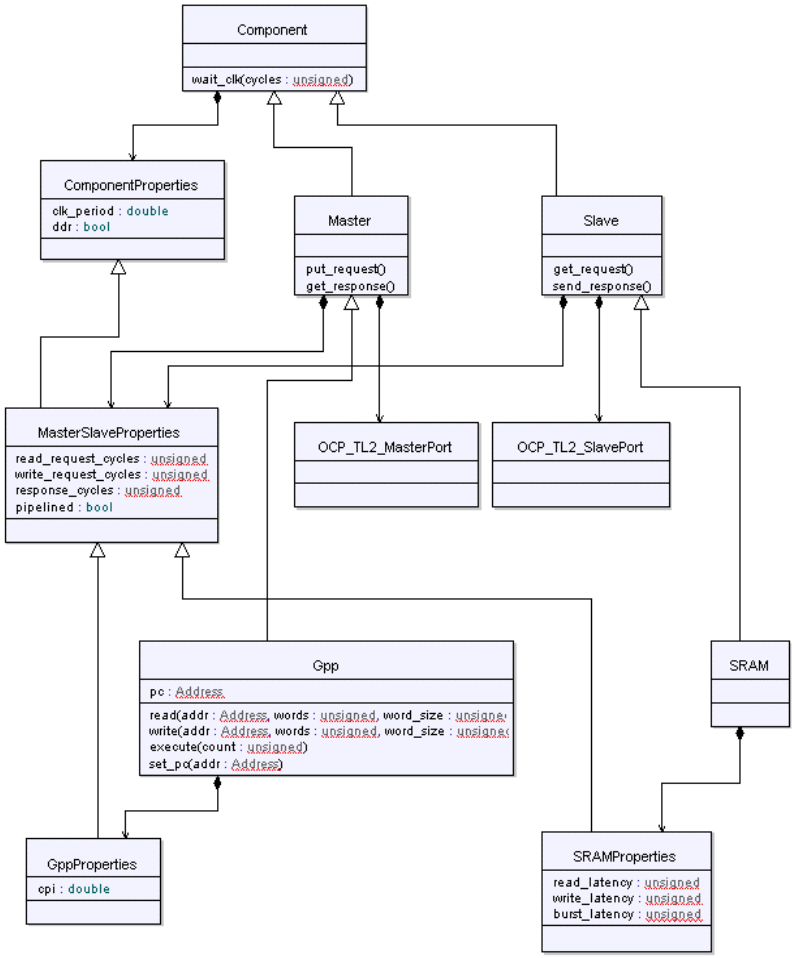


Figure 37: Component and master parameters.

Attribute name	Description
Component:	
ClkPeriod	Length of the clock cycle
Ddr	Operation in double data rate mode or not
Master-Slave:	
ReadRequestCycles	Number of clock cycles required for sending or receiving a read request from / to the interconnect
WriteRequestCycles	Number of clock cycles for a write request
ResponseCycles	Number of clock cycles for a response
Pipelined	This parameter denotes, if the master or slave is able to process requests and responses in parallel

© Copyright MARTES Consortium

General Purpose Processor:	
Cpi	Cycles per instruction
SRAM:	
ReadLatency	Number of clock cycles required for read operation
WriteLatency	Number of clock cycles required for write operation
BurstLatency	Number of clock cycles required for burst operation

Table 12: Component layer attributes.

3.3.3.4.2 Subsystem Layer

Subsystems are domain-specific collections of components, which are linked to each other, and subsystem-level services. The subsystem-level services are built on top of the component services, which are utilized via the interface functions listed in the component layer description. Standard UML2 techniques, e.g., composite structure diagrams, are used to represent subsystems.

As an example, an Image subsystem (see Figure 38) is built around the video accelerator, which provides hardware accelerated compression and decompression, pre-processing and post-processing services, as listed in Table 13. The services provided by the subsystem are controlled by a simple general purpose processor (GPP) based on the ARM v4 architecture. There is also some SRAM memory for the GPP, a video accelerator, and a DMA controller for offloading large data transfers between subsystems. The communication with other subsystems is performed through a specific interface component, namely NetworkIF.

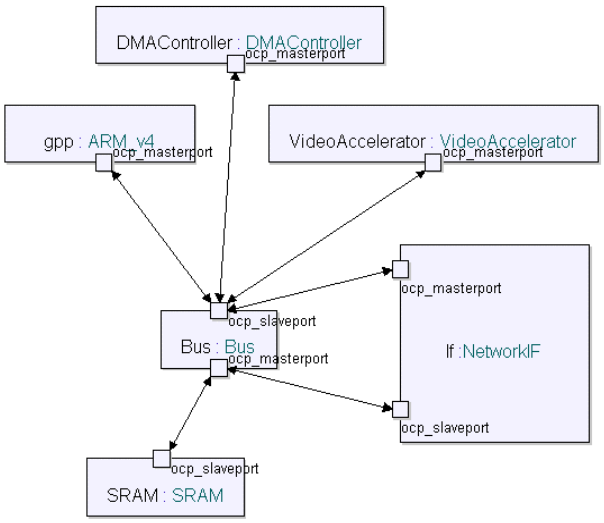


Figure 38: Image subsystem.

© Copyright MARTES Consortium

Service	Description
preprocess(*x*, *y*, *mx*, *my*, *z*)	Preprocessing for (*x*, *y*) resolution video with (*mx*, *my*) macroblocks and digital zoom factor *z*.
decode(*x*, *y*, *mx*, *my*, *b*, *m*, *c*)	Decode (*x*, *y*) resolution video using macroblocks of (*mx*, *my*) pixels and *b* bits per pixel. *m* old macroblocks will be analyzed in motion estimation per each new macroblock. The compression ratio of the encoded video is *c*.
encode(*x*, *y*, *mx*, *my*, *b*, *m*, *c*)	Encode (*x*, *y*) resolution video using macroblocks of (*mx*, *my*) pixels and *b* bits per pixel. *m* old macroblocks will be analyzed in motion estimation per each new macroblock. The compression ratio of the encoded video is *c*.
postprocess(*x*, *y*, *mx*, *my*, *d*)	Post processing for (*x*, *y*) resolution video with (*mx*, *my*) macroblocks and final color depth *d*.
dma_transfer(*source*, *target*, *words*, *word_size*)	Read *words* * *word_size* bits of memory from *source* address and write it to *target* address.

Table 13: Services of the image subsystem.

3.3.3.4.3 Platform Layer

The platform layer represents the set of subsystems, each implementing a number of subsystem services that are interconnected through a network type communication facility, which implements the communication services. The communication latency is characterized by the request delay, the response delay, and the processing time of the OCP interface (see Table 14).

Attribute name	Description
Communication Latency:	
RequestDelay	Transfer delay of request packet from initiator to network router
ResponseDelay	Transfer delay of response packet from target to network router
InterfaceProcessingTime	Network router arbitration time

Table 14: Network communication attributes.

An example of the execution platform model and the router structure are shown in Figure 39.

© Copyright MARTES Consortium

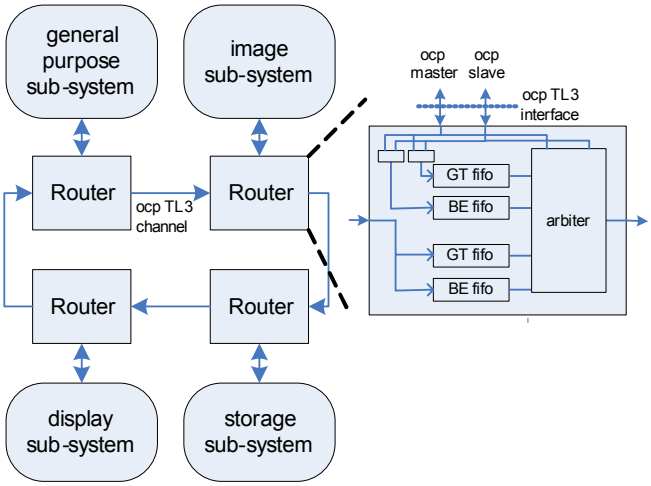


Figure 39: Execution platform model and router structure.

The platform services declare system-level functions. A platform service is "implemented" by a single subsystem or by a set of interacting subsystems. For example, the *video_playback()* platform service consists of the *preprocess()*, *decode()* and *postprocess()* services from the image subsystem, the *execute()* service from the general purpose subsystem, and the *mem_copy()* service from the memory subsystem. The communication services provided by the network are presented as the OCP interface methods at the router access ports. In most cases, they are directly used when implementing the subsystem services, and then become invisible from the application point of view. For the example of Figure 39, the implemented platform services are described in Table 15

Service	Description
video_playback(*x*, *y*, *z*)	Play a video with resolution rate (*x*,*y*) and digital zoom factor *z*. The video is taken from the memory sub-system, decoded in the image sub-system and sent to the display sub-system for viewing. The whole sequence is controlled by the general purpose sub-system.
send_command(*dst*, *type*)	Send a command with *type* to a target (*dst*).
data_transfer(*src*, *dst*, *length*)	Move the *length* of data from *src* to *dst*.

Table 15: Platform services.

3.3.3.5 Allocation and Allocated Model

For the allocation of the workload model to the execution platform model, a skeleton model of the execution platform is presented in UML. This model presents the same structural information as the SystemC execution platform model.

© Copyright MARTES Consortium

The allocation is defined by providing mapping associations from the workload model to the execution platform model. At the subsystem level, the processes and subservice functions are allocated to the processors of the subsystem.

3.3.3.6 Performance Simulation Model

The transformation from a UML-based workload model to a SystemC-based workload model is performed for allocated models created as the result of the transformation, i.e., it has been selected, where each part of the workload model will be executed on the execution platform model. Interfaces between the workload and execution platform model are provided by the execution platform models, and used by the workload models for transferring load information, controlling the execution platform model, and utilizing services. The output model of the transformation is a SystemC transaction level presentation of the allocated UML workload model.

The SystemC performance simulation model at the transaction level abstraction is obtained by linking and compiling the workload model with the execution platform model, both presented in SystemC. The performance simulation model is instrumented to collect performance data during the simulation. The attributes of the performance simulation are listed in Table 16.

Attribute name	Description
Network Utilization:	
NumberPackets	Total number of packets transferred
NumberData	Total amount of data transferred
NumberHops	Average number of hops per data
DataTransferTime	Average data transfer time
Processor Utilization:	
Idle	Percentage of time the processor is idle
MemoryAccess	Percentage of time the processor is accessing memory
DataProcessing	Percentage of time the processor is processing data
CacheHit/Miss	Number of cache hits / misses
NumberReads/Writes	Number of reads and writes initiated
Interconnect Utilization:	
NumberReads/Writes	Number of reads and writes transferred
NumberRequests/Responses	Number of requests and responses transferred

© Copyright MARTES Consortium

Memory Utilization:	
ReadWrite	Percentage of time the memory is either read or written
TotalNumberReads/Writes	Total number of reads and writes
InitiatorNumberReads/Writes	Number of reads and writes from a specific initiator
ReadWriteAccessTimes	Average access time for reads and writes
Software Execution:	
SWTaskExecutionTime	Time required to execute a SW task
SWTaskExecutionRate	Frequency with which SW task can handle data frames

Table 16: Performance simulation attributes.

© Copyright MARTES Consortium

4. Conclusion

Within the MARTES methodology, we propose an approach for which the model is the federative element for both the inter-component associations, and for the allocation and deployment of the application on an execution platform. The interest of a model in RTES design is that it allows early simulation and validation of an application with regard to its execution platform characterization, before any implementation has been done. Such approach reduces the development costs, and allows to test several scenarios and study the underlying performances.

The MARTES model defines a basis on which it is possible to model quality of service and performance characteristics, and to verify the platform / application adequation by defining non-functional constraints.

The designer will be able to incrementally enhance and exploit this model using model transformations, in order to produce an allocated model and an implementation model, which is specific to its domain and its purpose, including SystemC simulation and software code generation (Java, C, C++, Ada, etc.).

© Copyright MARTES Consortium

5. Terms and Definitions

Name	Description	Source
Abstract Platform	An abstract platform defines an acceptable, or, to some extent, ideal platform from an application developer's point of view. It is an abstraction of infrastructure characteristics, assumed for models of an application at some point of (the platform-independent phase of) the design process.	J.P. Almeida, R. Dijkman, M. van Sinderen, L.F. Pires, On the Notion of Abstract Platform in MDA Development
Horizontal Transformation	A transformation in which both the source and the target artifacts are at the same level of abstraction. It changes the structure of the input specifications, or weaves multiple specifications into a single output specification. In practice, this means that the source and target metamodel could, but must not be the same.	J. Greenfield, K Short, Software Factories
Mapping	Specification of a mechanism for transforming the elements of a model conforming to a particular metamodel into elements of another model that conforms to another (possibly the same) metamodel.	OMG MDA guide v1.0.1
Metadata	Data that represents models. For example, a UML model, a CORBA object model expressed in IDL, and a relational database schema expressed using CWM.	OMG MDA guide v1.0.1
Metamodel	A model of models.	OMG MDA guide v1.0.1
Meta-Object Facility (MOF)	An OMG standard, closely related to UML, that enables metadata management and modeling language definition.	OMG MDA guide v1.0.1
Methodology	The union of methods, notations, guidelines, strategies, and processes for a specific application domain.	K.U.Leuven
Model	A formal specification of the function, structure and/or behavior of an application or system.	OMG MDA guide v1.0.1

© Copyright MARTES Consortium

Name	Description	Source
Model Driven Architecture (MDA)	A specific approach defined by the OMG toward model driven development (MDD) that separates the specification of functionality from the specification of the implementation of that functionality on a specific technology platform. The focus of MDA is primarily the provision of standard specifications for useful technologies in the MDD space. Examples are the UML (modeling), MOF (metamodeling), transformations (QVT), platform specific annotations (set of UML profiles), etc.	OMG MDA guide v1.0.1
Model Driven Development (MDD)	A software development paradigm that promotes the use of models at different levels of abstraction, and transformations between them to derive a concrete application implementation. A specific interpretation of this notion is worked out by OMG's MDA.	K.U.Leuven
MDD Infrastructure	The collection of metamodels, platforms, transformations, languages, tools, etc., organized in a structured way, that support the development of a specific application or group of related applications (product line).	M. Voelter, J.Bettin, Patterns for Model-Driven Software-Development
Model transformation	The process of converting one model to another model of the same system. (Notice that this is not necessarily restricted to a single source model, nor a single target model)	OMG MDA guide v1.0.1 (Note: K.U.Leuven)
Oblique Transformation	A combination of a horizontal and a vertical transformation, which can appear from the outside as a single transformation	J. Greenfield, K Short, Software Factories
Platform	A set of subsystems/technologies that provide a coherent set of functionality through interfaces and specified usage patterns, that any subsystem depending on the platform can use, without concern for the details of how the functionality provided by the platform is implemented.	OMG MDA guide v1.0.1
Platform Independent model (PIM)	A model of a subsystem that contains no information specific to the platform, or the technology that is used to realize it.	OMG MDA guide v1.0.1
Platform Model	A set of technical concepts, representing the different kinds of parts that make up a platform, and the services provided by that platform.	OMG MDA guide v1.0.1

© Copyright MARTES Consortium

Name	Description	Source
Platform Specific Model	A model of a subsystem that includes information about the specific technology that is used in the realization of it on a specific platform, and, hence, possibly contains elements that are specific to the platform.	OMG MDA guide v1.0.1
Process	Definition of order of tasks and activities to be performed in order to obtain certain specific results. A process is usually divided in a number of (semi-ordered) phases.	I. Jacobson, G. Booch, J. Rumbaugh, The Unified Software Development Process
Process Activity	A concrete description of low level actions that need to be executed within a process phase.	I. Jacobson, G. Booch, J. Rumbaugh, The Unified Software Development Process
Process Phase	A distinct part within a process, consisting of several (semi-ordered) activities.	I. Jacobson, G. Booch, J. Rumbaugh, The Unified Software Development Process
Technology Domain	The domain in which we use the vocabulary of a certain technology platform which offers standard solutions for some problems. This is much related to execution platforms, such as J2EE or .NET, but also SQL, which represents a relational storage domain.	K.U.Leuven
Transformation	A model operation that takes one or more models as input, and returns one or more models as output. The operation maps elements from the source model elements to the target model elements.	J. Greenfield, K Short, Software Factories
Transformation Chain	If a sequence of more than one transformation is applied to an input model, the configuration of these transformations, including their sequence, possible branches and other information, is called a transformation chain.	B. Vanhoof, Y. Berbers, Breaking Up The Transformatoin Chain
Vertical Transformation	A transformation in which the source elements are at a different level of abstraction than the target elements. A transformation that lowers the level of abstraction is a refinement, whereas a transformation that raises the level of abstraction is an abstraction. In practice this usually means that the source metamodel is different from the target metamodel	Software Factories (J. Greenfield, K Short)

© Copyright MARTES Consortium

6. Acronyms

Abbreviation	Definition
AMBA	Advanced Microcontroller Bus Architecture
ARM	Advanced/Acorn RISC Machine
ASIC	Application Specific Integrated Circuit
ASSP	Application Specific Signal Processor
AXI	Automated X-ray Inspection
BC	Border Channel
BP	Border Process
CB	Combined Blocking
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CRP	Communicating Regular Process
CSP	Communication Sequential Process
Ctrl	Control
DBI	Direct Blocking In-order
DBO	Direct Blocking Out-of-order
DCT	Discrete Cosine Transform
DDF	Dynamic Data Flow
DE	Discrete Event
DMA	Direct Memory Access
DNI	Direct Non-blocking In-order
DNO	Direct Non-blocking Out-of-order
DSL	Domain Specific Language
DSM	Domain Specific Model
DSP	Digital Signal Processor
FIFO	First-In, First-Out
FPGA	Field Programmable Gate Array
GC	Generator Channel
GPP	General Purpose Processor
HetSC	Heterogeneous specifications in SystemC
HIBI	Highly Birefringent Fiber

© Copyright MARTES Consortium

HW	Hardware
I/O	Input/Output
IP	Intellectual Property
ISS	Instruction Set Simulator
KPN	Kahn Process Network
MARTE	UML Profile for Modeling and Analysis of Real-Time Embedded systems
MARTES	Model-based Approach for Real-Time Embedded System development
uC	Micro Controller
MCU	Micro Controller Unit
MDA	Model-Driven Architecture
MDD	Model-Driven Development
MIDP	Mobile Information Device Protocol
MoC	Model of Computation
MPEG	Moving Picture Experts Group
OMG	Object Management Group
OO	Object-Oriented
OS	Operating System
OSGi	Open Services Gateway initiative
PC	Personal Computer
PDA	Personal Digital Assistant
PE	Processing Element
PIM	Platform Independent Model
PN	Process Network
PSM	Platform Specific Model
QoS	Quality-of-Service
RAM	Random Access Memory
RB	Relative Blocking
RC	Reactive Channel
RFC	Request For Comments
RFP	Request For Proposal
RISC	Reduced Instruction Set Computer

© Copyright MARTES Consortium

RN	Relative Non-blocking
RTES	Real-Time Embedded System
RTL	Register Transfer Level
RTOS	Real-Time Operating System
SDF	Synchronous Data Flow
SDFG	Synchronous Data Flow Graph
SIP	Session Initiation Protocol
SoC	System-on-a-Chip
SPT	UML profile for Schedulability, Performance and Time
SR	Synchronous Reactive
SRAM	Static Random Access Memory
SW	Software
SysML	Systems Modeling Language
TI	Texas Instruments
TLM	Transaction Level Modeling
TTL	Task Transaction Level
UML	Unified Modeling Language, OMG Standard
UML2	Unified Modeling Language, version 2
UML4SoC	UML Profile for System-on-a-Chip SW/HW Co-Design
VHDL	VHSIC (Very High Speed Integrated Circuit) Hardware Description Language, IEEE 1076 Standard
WLAN	Wireless Local Area Network

© Copyright MARTES Consortium

7. References

[i] Oliver I., Applying UML and MDA to Real Systems Design. Proceedings of DATE'05 Conference. IEEE Computer Society Press, 2005.

[ii] MDA Guide version 1.0.1. Document number: omg/2003-06-01. 12th June 2003, http://www.omg.org.

[iii] A UML profile for MARTE, Initial Submission version 0.44, 14th November 2005, ProMARTE partners, http://www.promarte.org, 115 p.

[iv] Balarin F., Sentovich E., Chiodo M., Giusto P., Hsieh H., Tabbara B., Jurecska A., Lavagno L., Passerone C., Suzuki K., and Sangiovanni-Vincentelli A., Hardware-Software Co-Design of Embedded Systems - The POLIS Approach. Kluwer Academic Publishers, 1997. 297 p.

[v] Almeida J.P., Dijkman R., van Sinderen M., and Pires L.F., On the Notion of Abstract Platform in MDA Development. Proceedings of the 8th IEEE International Enterprise Distributed Object Computing Conference (EDOC'04), 2004.

[vi] Selic B., On Software Platforms, their Modeling with UML 2 and Platform-Independent Design. (ISORC'05), 2005.

[vii] http://www.sparxsystems.com.au.

[viii] Keutzer, K., Malik, S., Newton, A.R., Rabaey, J.M., and Sangiovanni-Vincentelli, A., System-level design: orthogonolization of concerns and platform-based design, IEEE Trans. Comput. Aided Des. Integr. Circuits Syst., 2000, 19(12), pp. 1523–1543.

[ix] Lee E.A., and Messerschmitt D.G., Synchronous data flow, Proceedings of the IEEE, Vol. 75, No. 9, September 1987.

[x] Lee E.A., and Parks T.M., Dataflow Process Networks, Proceedings of the IEEE, Vol. 83, No. 5, May 1995.

[xi] Kahn G.., The Semantics of A Simple Language for Parallel Programming", Proc. IFIP 74, North-Holland, 1974.

[xii] Eker J., Janneck, J.W., Lee E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., and Xiong, Y., Taming heterogeneity - the Ptolemy approach, Proc. IEEE, 2003, 91(1), pp. 127–144.

[xiii] Hoare C.A.R., Communicating Sequential Processes, Communications of the ACM, Vol. 21, No. 8, August 1978.

[xiv] Edwards S.A., The Specification and Execution of Heterogeneous Synchronous Reactive Systems, PhD thesis, Dept. EECS, University of California, Berkeley, May, 1997.

[xv] YAPI, http://sourceforge.net/projects/y-api/.

[xvi] Posadas H., Herrera F., Sánchez P., Villar E., and Blasco F., System-Level Performance Analysis in SystemC. Proc. of DATE'04, Paris, Feb., 2004.

[xvii] OSCI. Draft Standard SystemC Language Reference Manual. May, 2005, http://www.systemc.org.

[xviii] Rose A., Swan S., Pierce J., Fernández J.M., Transaction Level Modeling in SystemC, http://www.systemc.org.

[xix] Posadas H., Herrera F., Fernández V., Sánchez P., and Villar E., Single Source Design Environment for Embedded Systems based on SystemC. Journal on Design Automation for Embedded Systems. Vol. 9. Number 4. pp.293-312. Springer. December, 2004.

[xx] Fernández V., Herrera F., Sánchez P., and Villar E., Embedded Software Generation From SystemC For Platform Based Design, SystemC: Methodologies and Applications. Ed. W.Mueller, W. Rosenstiel, J.Ruf. Kluwer Academic Publishers. March 2003. ISBN 1-4020-7479-4.

[xxi] Jantsch A., Modeling Embedded Systems and SoC's. Morgan Kaufman Publishers. Elsevier Science. 2004.

[xxii] Kreku J., Eteläperä M., and Soininen J.-P., Exploitation of UML2-Based Platform Service Model and SystemC Workload Simulation in MPEG-4 Partitioning. International Symposium on System-on-Chip 2005, pp. 167–170.

[xxiii] Kreku J., Qu Y., Soininen J.-P., Tiensyrjä K. Layered UML Workload and SystemC Platform Models for Performance Simulation. Proceedings of the FDL'06 conference September 19-22 2006, Darmstadt, Germany. ECSI. pp.223-228.

© Copyright MARTES Consortium