VERification-oriented & component-based model Driven Engineering for real-time embedded systems

F3.1.1 – Analysis of Existing Frameworks and Definitions of Methodological Rules – V2

••

Document version no.:	2.3
Edited by:	TCF CEA TAS TRT GSY UPB RB IFX AST SCS FZI
Coordinated by	UPB

History

Document version #	Author(s)	Date	Remarks
Version 0.1	UPB	24.11.2010	Initial release
Version 0.2	TCF, CEA, TRT, UPB, RB, FZI	15.12.2010	
Version 0.3	UPB	20.12.2010	
Version 0.4	TCF, UPB	07.01.2011	Section regarding LwCC refined
Version 0.5	TCF, UPB, CEA	19.01.2011	Further refinements, sections introduction and conclusion completed
Version 0.6	TCF, CEA	21.01.2011	Class diagrams for LwCCM, additional elements for the definition of the VERDE ML.
Version 0.7	TRT, UPB, RB	25.01.2011	Section requirements added
Version 0.8	TCF, UPB	04.02.2011	Integrated some comments from internal review
Version 0.9	UPB	10.05.2011	Added some illustrations and minor modifications to section 4.
Version 0.10	ASTR	01.06.2011	Paragraph 1.3 and annex Space Domain data Model added
Version 1.0	TCF	10.06.2011	Removed the space domain annex, as it will be released in a separate document. Also put additional elements in introduction and conclusion, as well as for the modeling patterns.
Version 1.1	RB	14.06.2011	Cleanup of Section 5, Requirements Coverage
Version 2.0	UPB	15.12.2011	Final Updates
Version 2.1	TCF	15.01.2012	Review and improvements of some explanations
Version 2.2	TCF	05.09.2012	Clarify the modeling of flow ports, and explained how to use standard UML stereotypes for component types and implementations.
Version 2.3	TCF	6.8.2013	Fixed the version number and integrated all the revisions

Table of Contents

TABLE OF FIGURES

1. Introduction

This document is the main part of deliverable F3.1.1, which is the outcome of task 3.1. Its purpose is to define the part of the VERDE language that deals with the description of execution platform. The rationale for the definition of the VERDE language is to rely on existing modeling languages and define of subset of them. More specifically, UML was selected as the modeling language to choose; the VERDE language is an identification of a subset of UML profiles. The VERDE language is defined to address the modeling aspects that are required to describe the execution platforms that are involved in the project: Lightweight CCM, SCA, SystemC and AUTOSAR.

An additional document, describing the data model used for the space domain in the scope of the VERDE project, is also provided as part of the F3.1.1 delivery.

1.1 Rationale

The purpose of task 3.1 is to provide the definition of a common modelling language that can address all the concepts implemented in the targeted platform technologies (Lightweight CCM, SCA, SystemC and AUTOSAR).

Work in task 3.1 consists in analysing the selected component frameworks, considering their component model, execution model, composition strategies, deployment and configuration aspects, and verification models. This implies the analysis of the existing capabilities and the definition of rules that bring usage constraints regarding execution or communication semantics in a real time environment. The outcome of this analysis is a set of comparison criteria that are used to identify what concepts must be addressed by the Verde modelling language. A common model of computation and communication is defined and constitutes the basis to transform VERDE models characterized with NFP onto actual execution components in the other tasks of WP3.

The elaboration of the VERDE modelling language considers and evaluates the results of other collaborative projects (like ANR-Flex-eWare or Artemis-Chess), and aims at mostly relying on existing standards such as plain UML, SysML and MARTE. The expected outcome of task 3.1 is to define a modelling language that is a subset of already implemented modelling UML profiles, to ease its adoption.

Figure 1 gives an overview of the goal of this task.

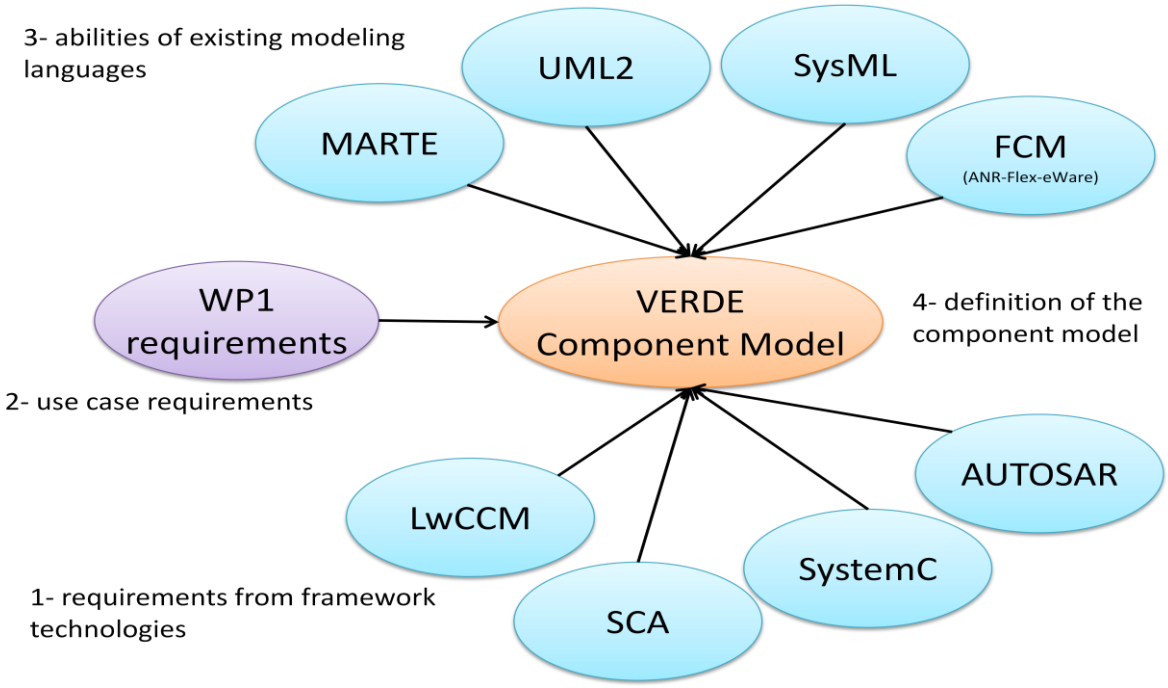


Figure 1: Overview of the standards involved in VERDE

1.2 Relation to other Work Packages and Tasks

The VERDE modelling language defined in task 3.1 is meant to be a part of the complete VERDE language that addresses application component and execution platform descriptions. That is, it defines the elements of the VERDE language that cover the component model and the component deployment information. It is the input for tasks T3.2, T3.3, T3.4 and T3.5 that deal with the transformation from the VERDE language to the existing platforms (LwCCM, SCA, SystemC and AUTOSAR).

This deliverable is related with deliverables of other work packages, especially work packages 4 and 5 that deal with other modeling aspects in project VERDE. The consistency between all these deliverables ensures that verification through analysis and the testing performed are consistent with the concrete execution platforms.

Figure 2 depicts the links to other work packages and tasks in the VERDE project.

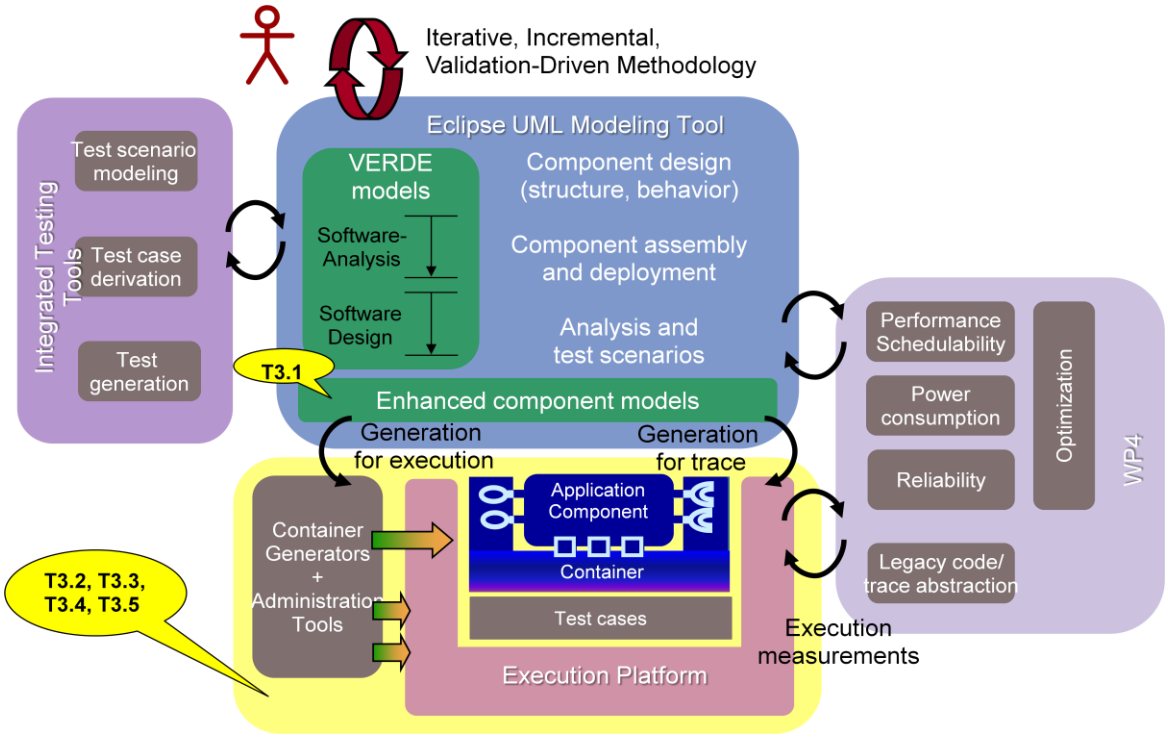


Figure 2: Project integration

1.3 Document Structure

The document is structured in two main parts: the first part describes the targeted technologies (section 2) and the languages selected as a basis for the VERDE language (section 3). The second part defines the VERDE language itself (section 4).

1.4 Annex for the Space Domain Data Model Definition

The Space Domain Data Model as used by ASTRIUM GmbH is based on the European Standard ECSS-ETM-10-23. This standard is the common denominator for the European Space Agency (ESA) and the space industry for any project developing a space system. This leads to a rather specific data model, which is due to the fact, that it is related to a clearly defined system purpose and system operating environment. As it is very specific to the space domain and the Astrium activities, this model is released in a separate document.

In order to account for this specialization, the meta-model for space domain is enclosed in a separate annex. Please note that Verde will most probably lead to further extensions to this data model, especially in the area of verification to support model derivation or case derivation. This is more than welcome and can be introduced back to the relevant ECSS-ETM-10-23 steering committee which ASTRIUM GmbH leads.

2. Standards involved in VERDE

In this section, we describe the existing technologies that were taken into account for our work. Each technology will be described.

2.1 Comparison criteria

In order to identify which concepts should be addressed by the Verde modeling language, we compared the different targeted technologies. The goal is to isolate comparison criteria for which the Verde language shall provide modeling patterns. The technologies under investigation are LwCCM, SCA, SystemC and AUTOSAR.

After studying the different technologies, we identified a set of concepts they address in various ways. We thus were able to define comparison criteria to guide us through the evaluation of the different technologies and thus to ease the comparison and identification of the key notions to cover in the VERDE modeling language. These criteria are divided into two parts, namely *component model* and *deployment information*. The component model section covers the elements that can be used to describe a software component and is further subdivided into the following parts:

- data types;
- interfaces;
- component types;
- component implementations;
- ports;
- component instances;
- interaction patterns.

The deployment information part covers modeling information related to component deployment:

- definition of execution resources;
- resource allocation;
- non-functional properties;
- relationship between runtime and components.

The following sections describe the meaning of the different criteria for each technology.

2.2 LwCCM: general concepts

LwCCM defines a component model and a way to describe component deployment. The following concepts are defined:

- Component type
- Communication port, with two possibilities: event port and operation-based port.
- Interface and Data type.

The deployment aspects cover the following elements:

- Component implementation;
- Component instance;
- Execution resources;
- Non-functional properties.

LwCCM implies a clear separation between the functional code encapsulated within components and the underlying middleware and runtime. As a consequence, the functional code has no visibility on the deployment, and thus is controlled by the runtime. Interaction with external elements is performed through communication ports: no direct communication with runtime services is possible.

2.3 LwCCM implementation in MyCCM

MyCCM is a component framework developed in Thales. It is an implementation of the LwCCM with some adaptations. The purpose of MyCCM is to provide a component framework which tailored for the particular needs of Thales operational divisions. Therefore, MyCCM is actually a collection of frameworks rather than a single product.

In the scope of the VERDE project, we use two versions of MyCCM: one for the space domain, another for software radio. The MyCCM for software radio is actually used as a basis to implement the SCA standard. In this section, we focus on the MyCCM dedicated to space domain, which targets an adaptation of LwCCM. MyCCM uses the LwCCM component model with some extensions, and does not strictly rely on the OMG D&C standard for deployment description. As depicted in Figure 3, five aspects can be identified in a MyCCM model: the data types and the component models correspond to the LwCCM component model itself. Activities and deployment topology correspond to the description of execution resources. Finally, the deployment information creates a relationship between the component model and the execution resources.

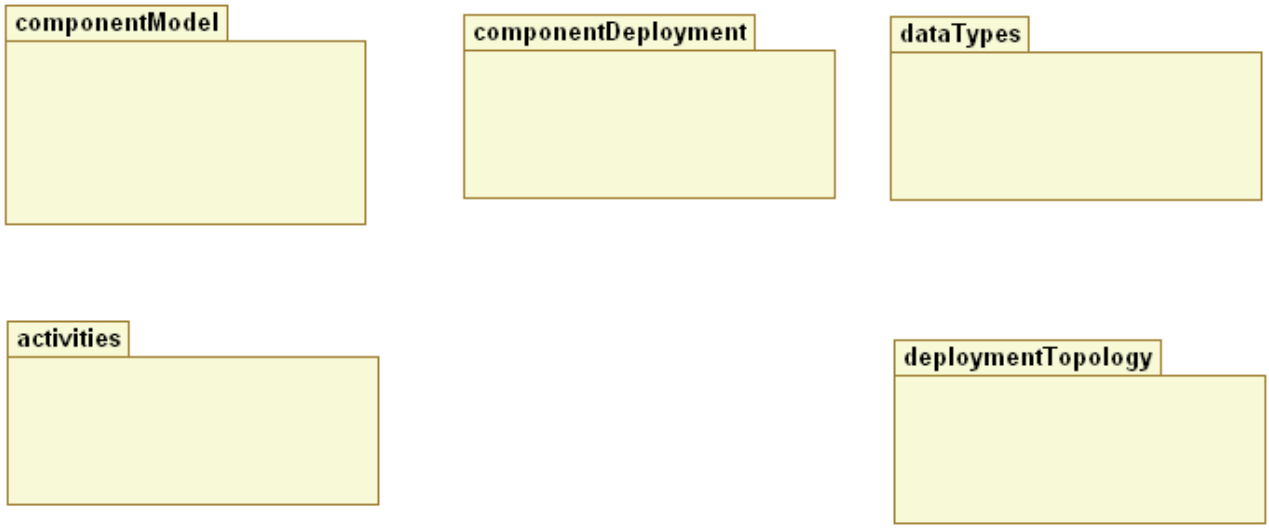


Figure 3: MyCCM model

2.3.1 Data types

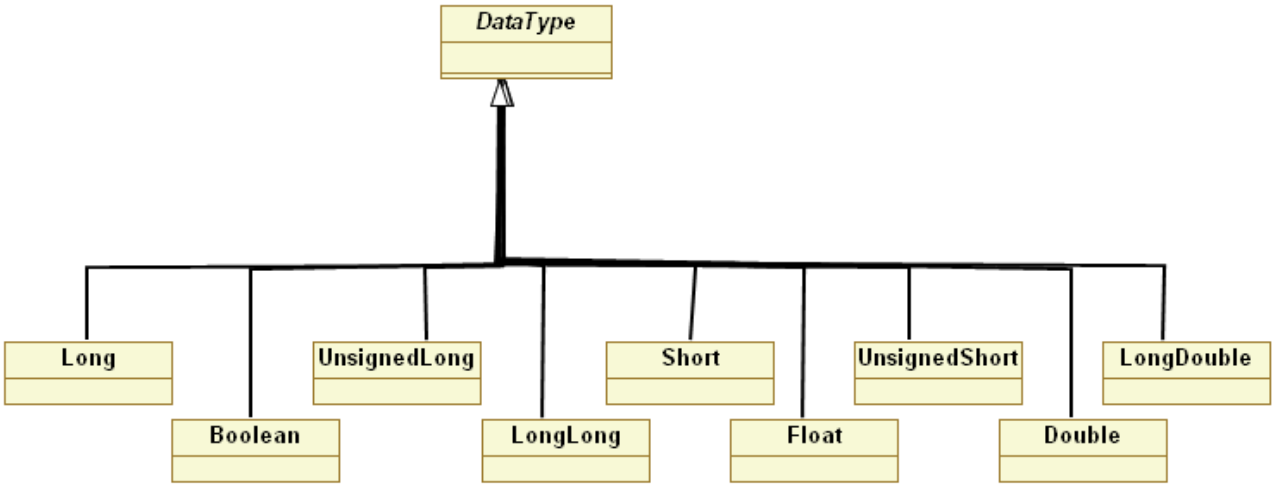


Figure 4: Data types

MyCCM supports IDL data types, as well as some specific extensions. The simple data types are integers, Booleans and float numbers (see Figure 4); there are different sizes of integers and floats. Complex data types are arrays, enumerations, structures and unions (Figure 5).

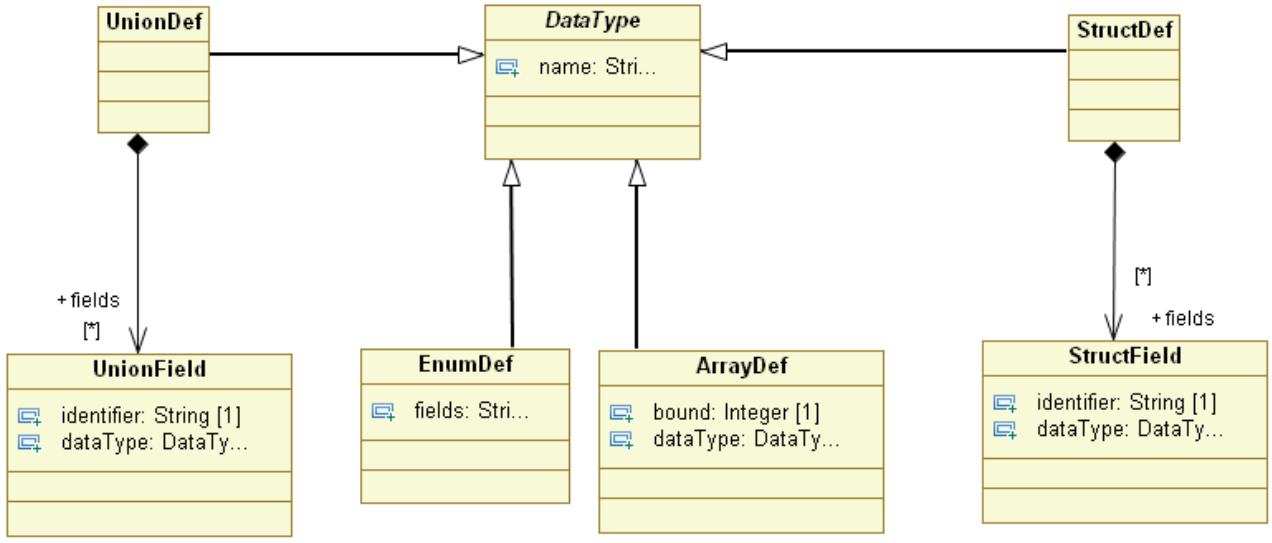


Figure 5: Complex data types

In addition, MyCCM supports constrained integers, which are integer with arbitrary value ranges associated with a unit, and constrained arrays, which are arrays that can be indexed with an enumeration instead of integers (Figure 6).

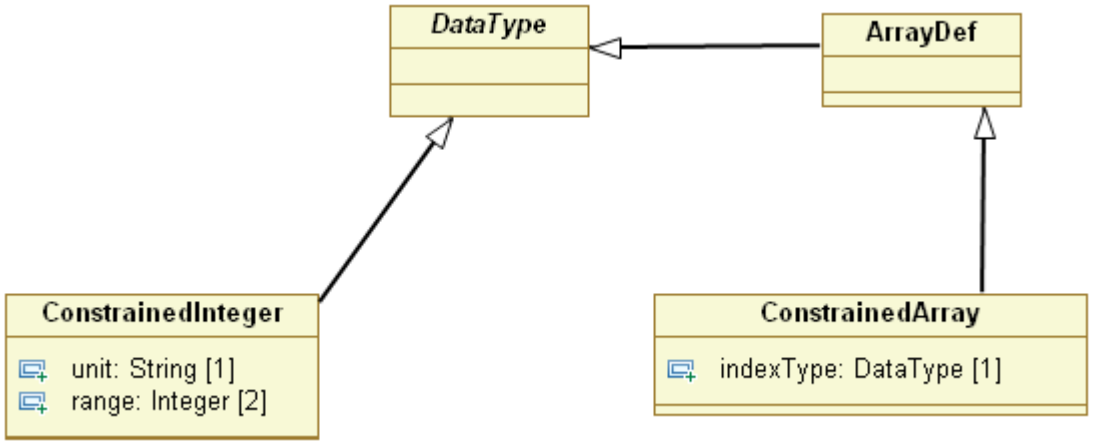


Figure 6: LwCCM Constrained data type

2.3.2 Interfaces

MyCCM supports IDL interfaces, which are sets of operations. Operations have parameters, which have types (Figure 7).

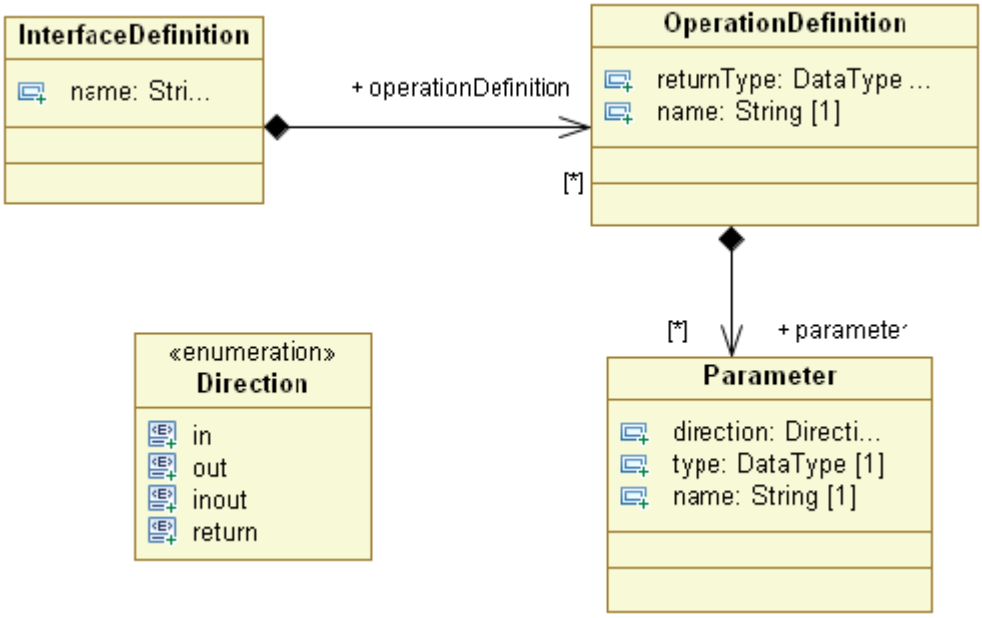


Figure 7:LwCCM Interfaces

2.3.3 Component types

Component types have ports, which are interaction points (see Figure 8).

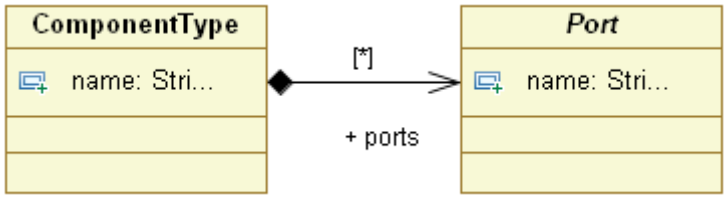


Figure 8: LwCCM component ports

2.3.4 Interaction patterns

There are several kinds of ports: event ports and interface ports. Interface ports are associated with one interface, which is either provided (facet) or required (receptacle). In addition to standard LwCCM ports, MyCCM supports the notion of extended port, which can have several interfaces as displayed in Figure 9.

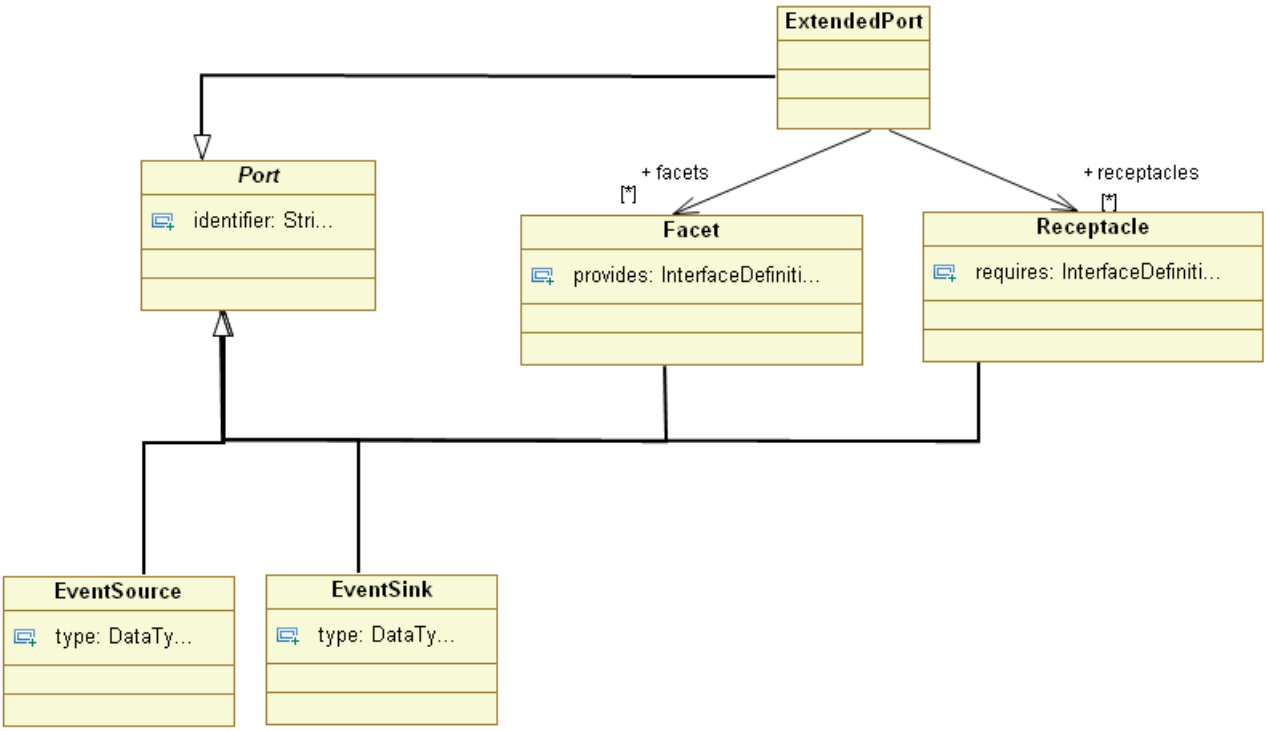


Figure 9: LwCCM extended ports

2.3.5 Execution resources

As LwCCM deals with software architecture, execution resources are operating system threads. MyCCM supports three kinds of threads: periodic, sporadic and "one-shot". One-shot threads are to be executed only once (Figure 10).

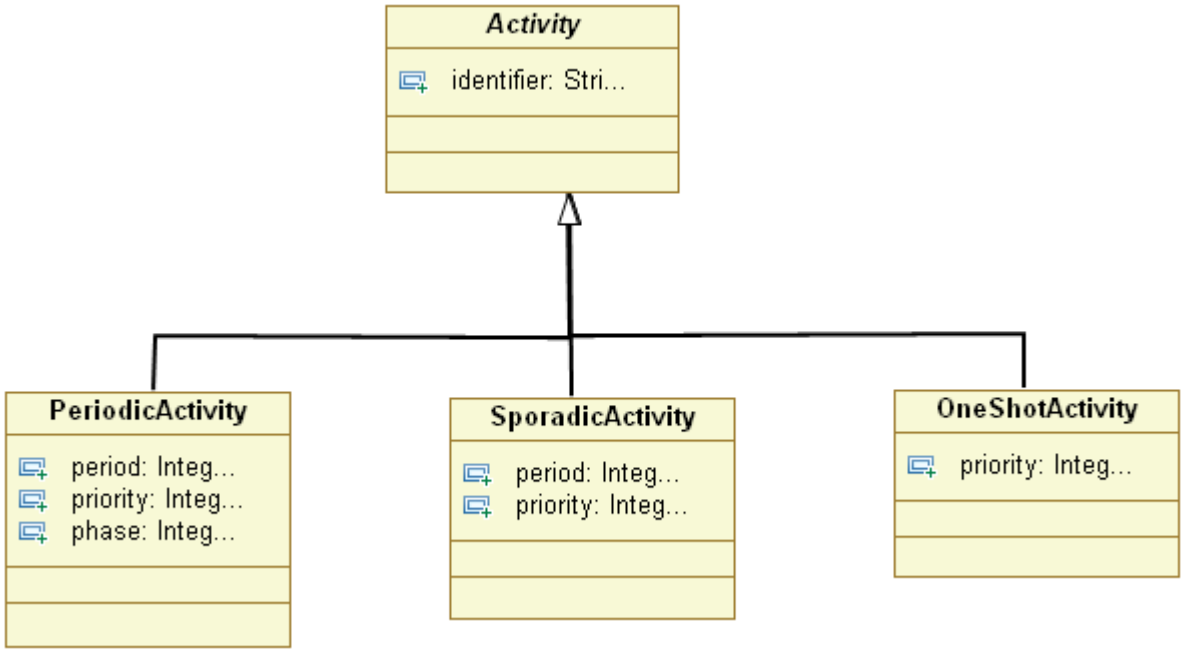


Figure 10: LwCCM execution resources, thread

Activities are associated with processes, which correspond to operating system programs. The hostname of a process is the identification of the computer on which the process will be deployed (Figure 11).

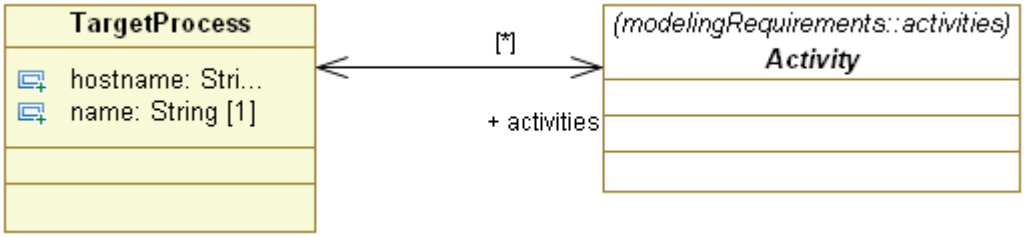


Figure 11: LwCCM execution resources, process

2.3.6 Component implementation and instances

Component types define component signatures, and do not represent actual source code. Source code is stored within component implementations. Component implementations are themselves deployed on execution resources (Figure 12).

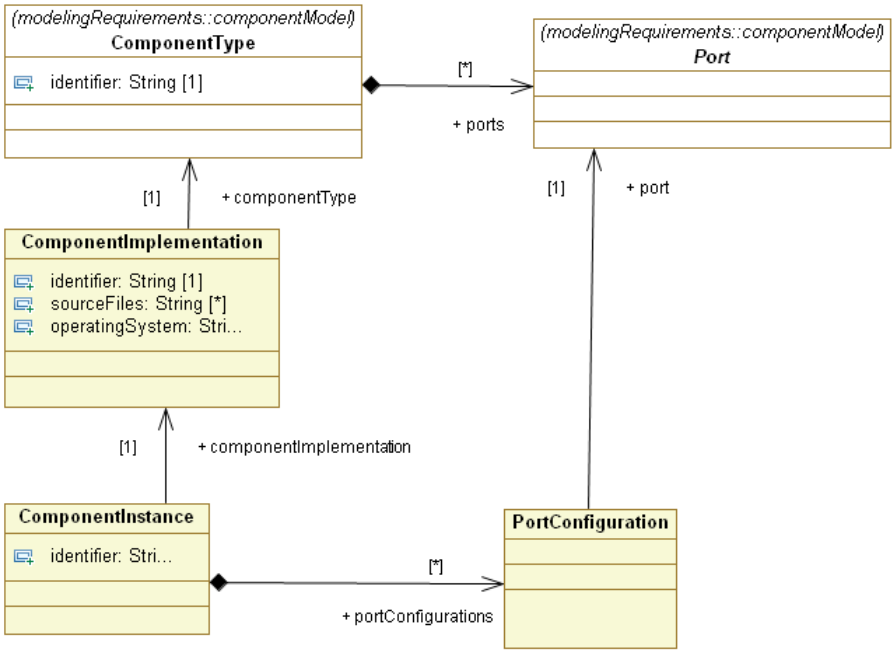


Figure 12: LwCCM component implementation

Component instances are deployments of component. The corresponding concept for ports is port configuration. A port configuration is associated with a component instance, and possibly with activities. Indeed, activities control component ports, not the component themselves (Figure 13).

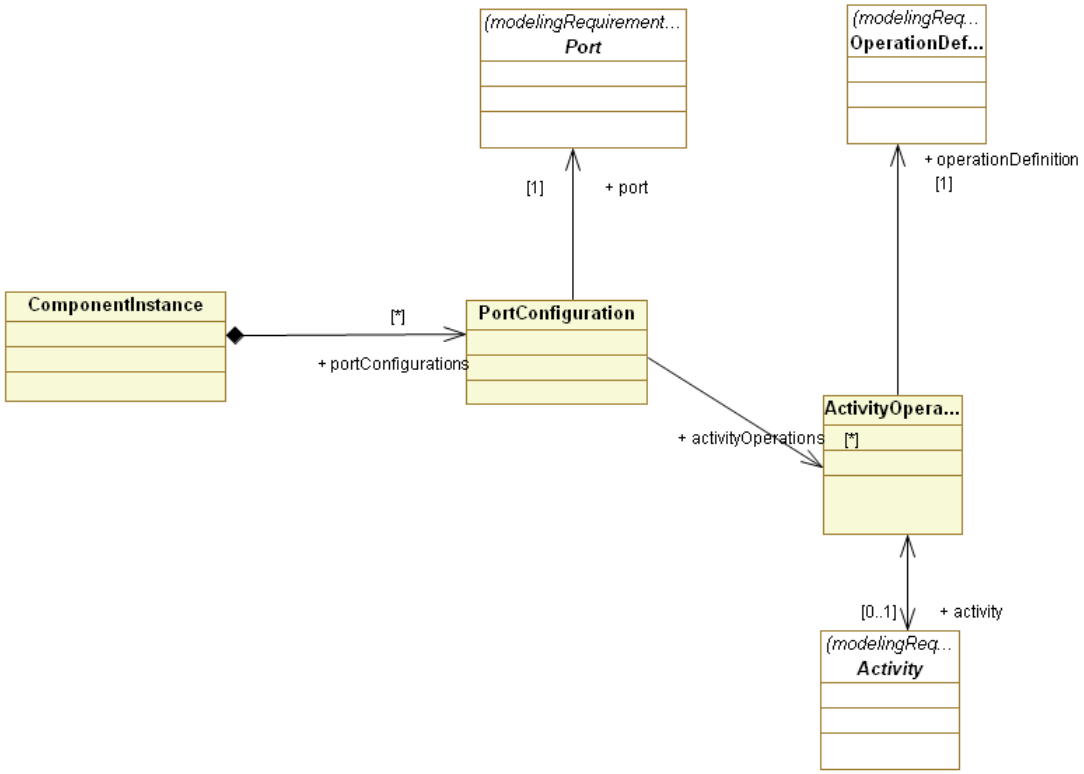


Figure 13: LwCCM component instances

2.3.7 Component connections

Ports of component instances must be connected one with another in order to create a complete architecture. Two kinds of connections are supported by MyCCM: direct connections (Figure 14) and connections through a connector (Figure 15).

Direct connections are used for interaction patterns that correspond to standard port kinds (events or interfaces), while connectors are used to implement more complex interaction patterns (i.e. consensus, deferred communications, etc.).

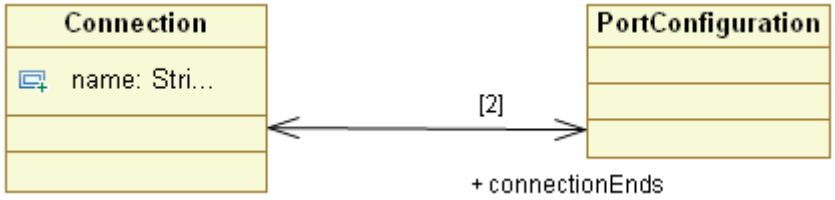


Figure 14: LwCCM direct connection

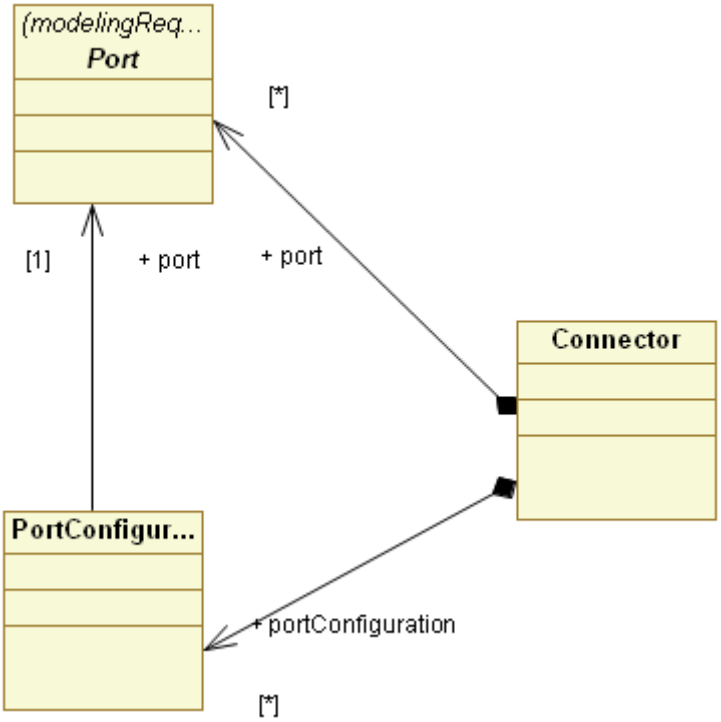


Figure 15: LwCCM connection through a connector

2.3.8 Summary

Component model

Component types, component implementations and component instances correspond to the standard LwCCM approach. Specifically, component types correspond exactly correspond to the standard IDL3 definition of LwCCM.

Data types are the standard IDL data types with some restrictions and some extensions. Only fixed-size data types are allowed; thus, strings, any types are forbidden. Complex data types such as sequences, unions and structures are allowed. As the targeted language is Ada, data types are extended to manage some specificities of Ada: union elements are associated with numbers, integers (short, long, etc.) can be associated with ranges of values.

The two kinds of ports (facets and event sinks) can be driven by threads to have asynchronous communications. This has an impact on interface specifications: operations must be declared as synchronous (standard IDL semantics) or asynchronous (which implies specific processing by the framework and a different code mapping).

Deployment information

MyCCM for space explicitly defines the execution resources as system threads, with priority, nature (periodic or sporadic), period if needed and memory stack size. This information is non-functional properties that must be attached to thread declarations.

Threads control the execution of component ports. They can used be associated with either input event ports (event sinks) or operations of provided interface ports (facets). Threads are to be associated with processes, i.e. memory partitions. Memory partitions are to be associated with computation nodes, i.e. processors. The space domain use case is very specific, as the targeted platform only has two computation nodes: the satellite and the ground station on Earth. Therefore, there is no need for non-functional properties to specify network addresses.

2.4 LwCCM implementation in eC3M

eC3M (see also www.ec3m.net) is a component based modeling / middleware approach that is inspired by CCM. The component model is based on the Flex-eWare component model (FCM), i.e. a UML [6] model applying the profiles FCM (see section 3.3) and MARTE. The latter is the OMG standard for embedded and real-time systems. As a deviation from standard CCM, IDL based descriptions are replaced by model elements: UML components (classes) owning a set of ports and optionally inheriting from others. The XML deployment descriptors of CCM are replaced by a set of UML instance specifications which allow for the configuration of attributes and the allocation to nodes.

The transition from the model towards executable code is done by a sequence of model transformations. These transformations include

The “connector” pattern is the replacement of a connection by a component that is responsible for the realization of the interaction. Thus, a connection becomes a first class model element which posses a type and an implementation. Since the interaction component needs to be adapted to the context (e.g. the port types), is must be instantiation from a template within a model library.

The “container” pattern, is the enclosing of a component executor by a container which may provide an additional set of services, either by intercepting port communication (enabling thus for instance services such as tracing or mutual exclusion) or providing additional services (used for instance for reflective data).

The distribution of model elements towards a model that represents the elements required on a certain deployment node.

Finally, a model to code transformation uses the standard code generation facilities of Papyrus, in particular for C++. An Eclipse CDT project is generated and configured for each node.

An overview of the approach is shown in the Figure 16.

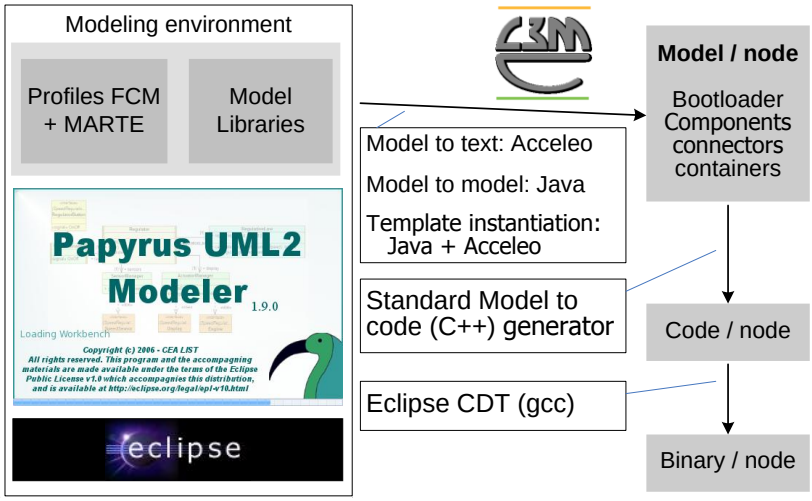


Figure 16: The eC3M toolchain

2.5 SCA

Here we will explain SCA and especially provide details for the different comparison criteria. This will be connected with deliverable F3.3.1.

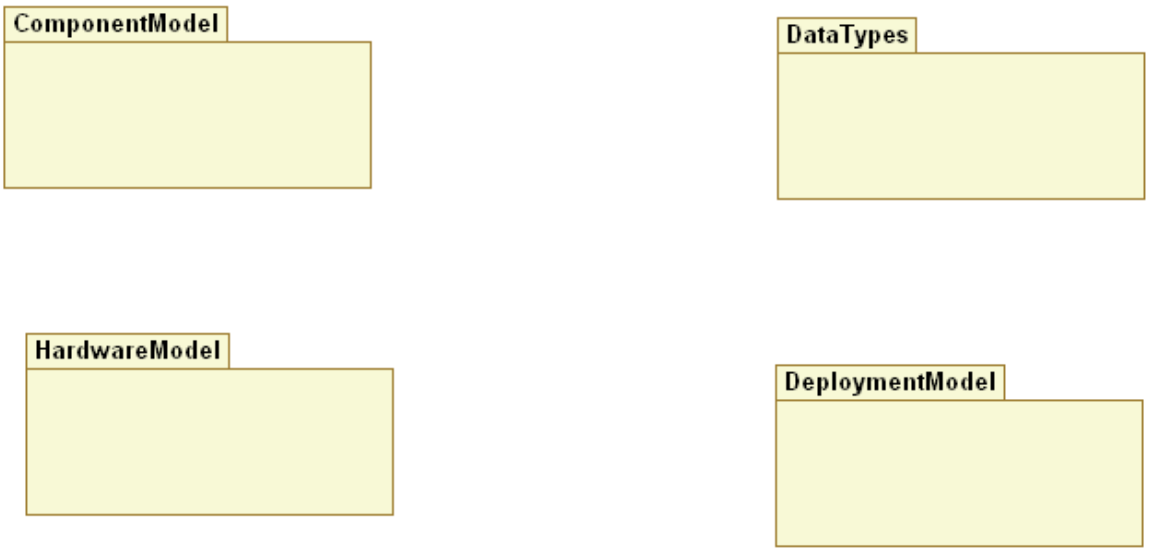


Figure 17 SCA metamodel Packages overview

2.5.1 Component types

In SCA model a component (Resource) contains a set of ports (inheriting from Port, LifeCycle, Testable, PortSupplier). All this classes are designed for activation and configuration and connection. SCA Resources cannot be nested. They are controlled by the AssemblyFactory.

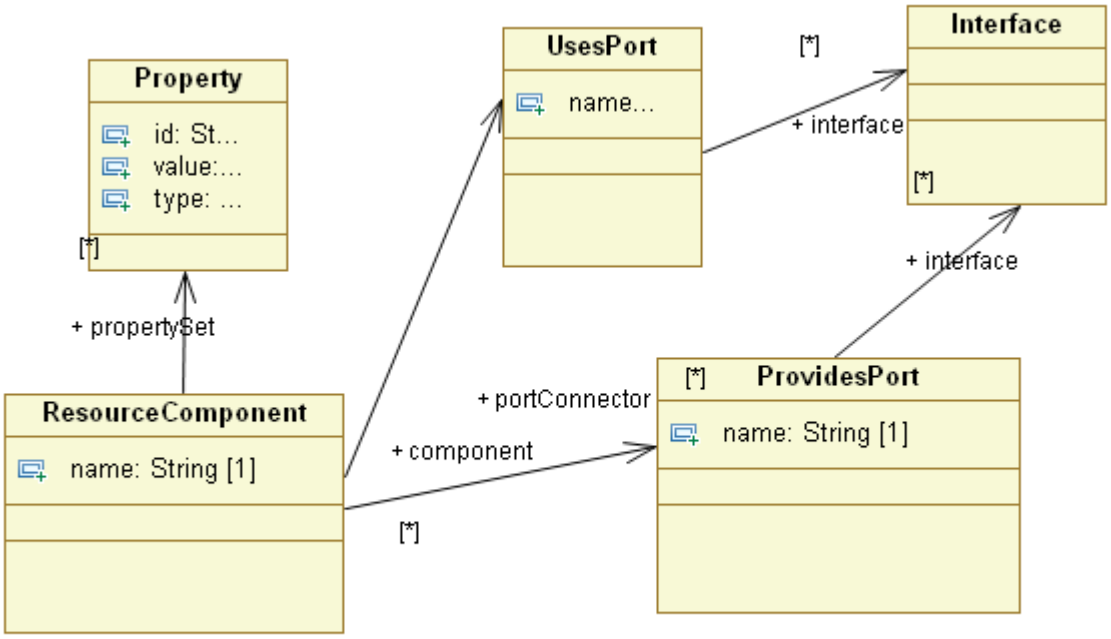


Figure 18 component types metamodel

2.5.2 Ports

Like any component model, they are either required or provided interfaces. "getPort" method has to be implemented to provide specific interaction patterns, that permits to obtain required and provided port.

2.5.3 Interfaces

This feature permits to realize declaration of operations (C-like functions).

2.5.4 Component implementations

It allows hosting code provided by user (stored in a so-called artifact). It could contain tasks, and synchronization mechanisms.

2.5.5 Component instances

They are a way to use the same component implementation with different values thanks to factory.

2.5.6 Interaction patterns

In SCA two ways of communications are proposed: synchronous operation calls and events (for asynchronous call). The Resources could communicate through CORBA.

2.5.7 Data type

The set of types corresponds to OMG IDL, with some limitations: integer with defined size, float, fixed point values, structures, enumerations, sequence.

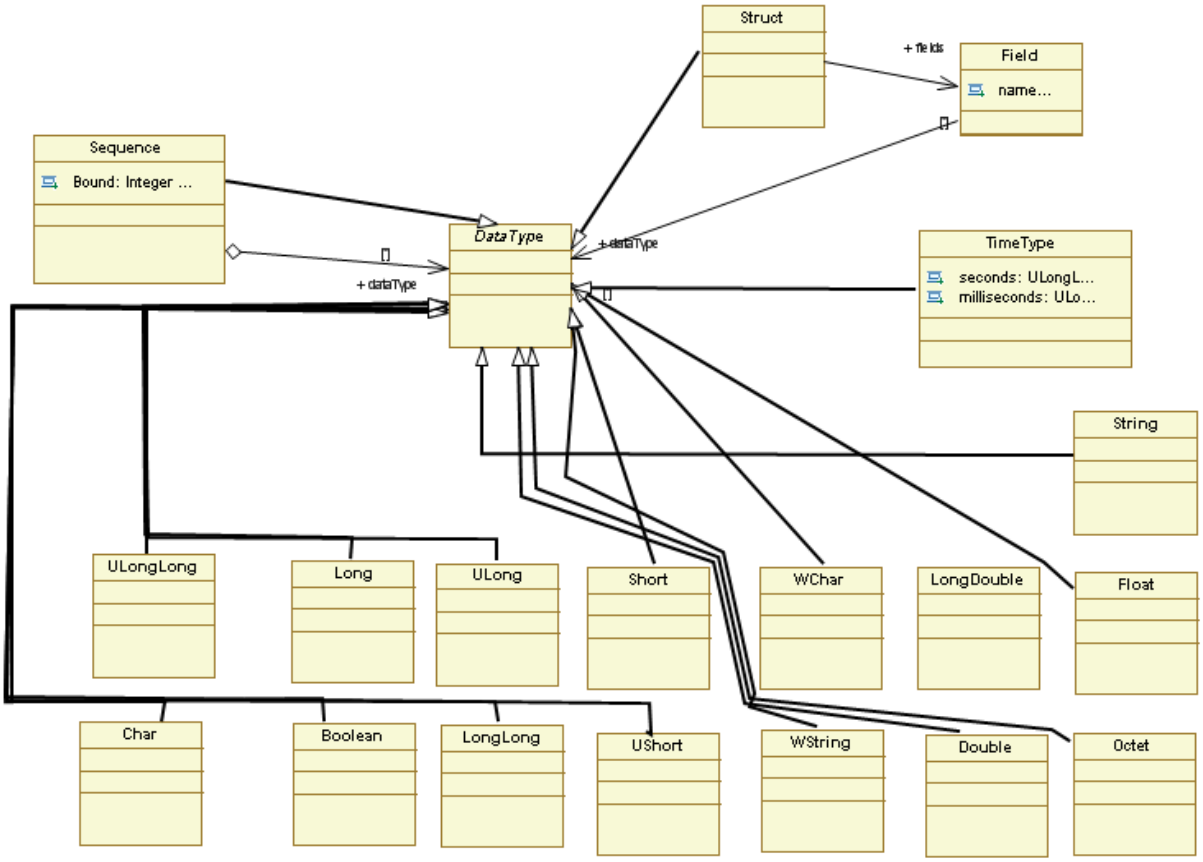


Figure 19 Datatypes metamodel

2.5.8 Deployment allocation

SCA mainly provides deployment facilities. SCA Core Framework provides some means to manage the assembly of components, application life cycle, and devices.

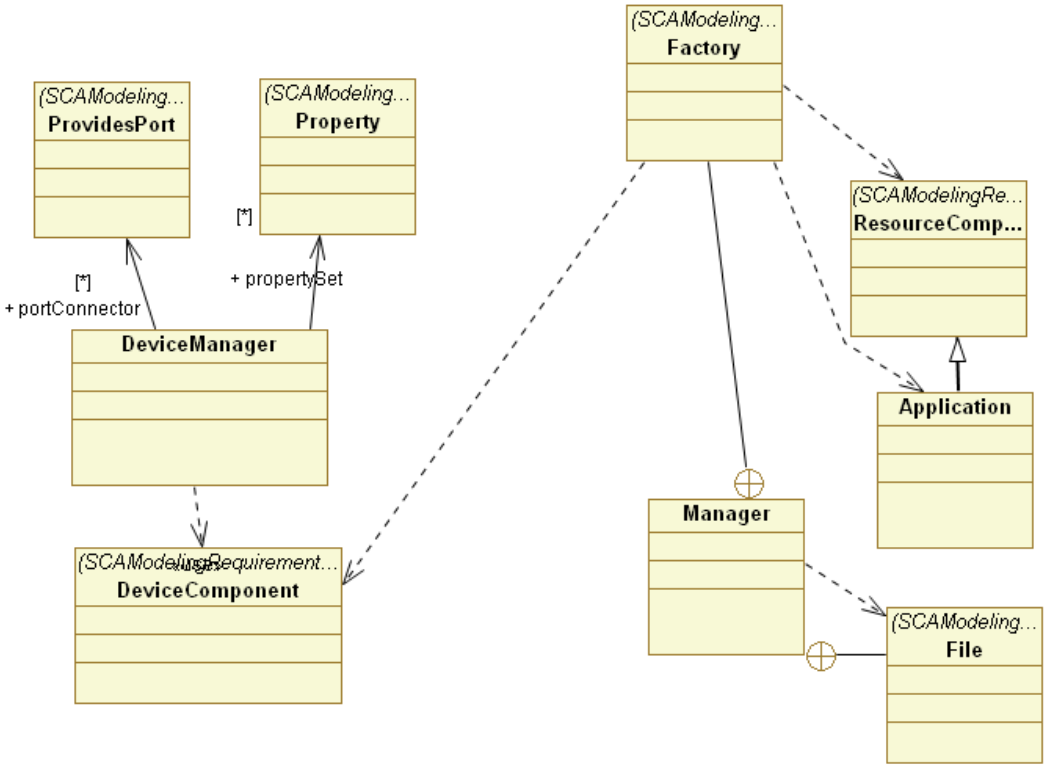


Figure 20 SCA deployment Metamodel

2.5.9 Execution resources

This feature is not directly present in SCA. The execution semantic has to be provided by the component designer. Supposedly POSIX (CORBA is optional).

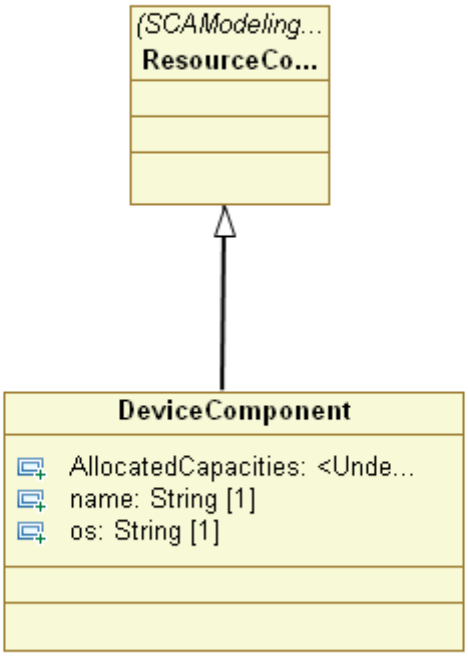


Figure 21 SCA Hardware Metamodel

2.5.10 Non-functional properties

Tasks behavior, synchronization, and relation to operating system are not directly defined in the SCA. Runtime has to be provided by the user, and the XML configuration file has to be designed.

2.5.11 Relationship between runtime and components

No specific runtime is associated and defined in the specification. The user has to define its own runtime.

2.6 SystemC

This section will provide information and details for the SystemC component model with regard to the comparison criteria defined in section 2.1. Besides of the component model itself, this will also embrace composition strategies, deployment, and execution semantics.

SystemC is a C++ library which integrates concepts of concurrency, parallelism, modularity, and separation between communication and computation into the standard C++ programming language [3][4]. The following sections will abstract away the absolute main C++ concepts and focus on the component characteristics of SystemC.

2.6.1 Component Types

SystemC designs are basically composed by an interconnection of several components. This basic component structure is a module, in SystemC syntax called *sc_module*. A module can be hierarchical which means that it can contain other modules. A special kind of module, which is used to represent communication

behavior, is called *sc_channel* (see also section 2.6.6). Channels usually implement methods for communication purposes which are defined in interfaces (please refer to section 2.6.3). As SystemC is a C++ library, modules are C++ classes which inherit from the overall superclass *sc_object* (see Figure 22). Therefore, they can contain member variables, attributes, and member functions.

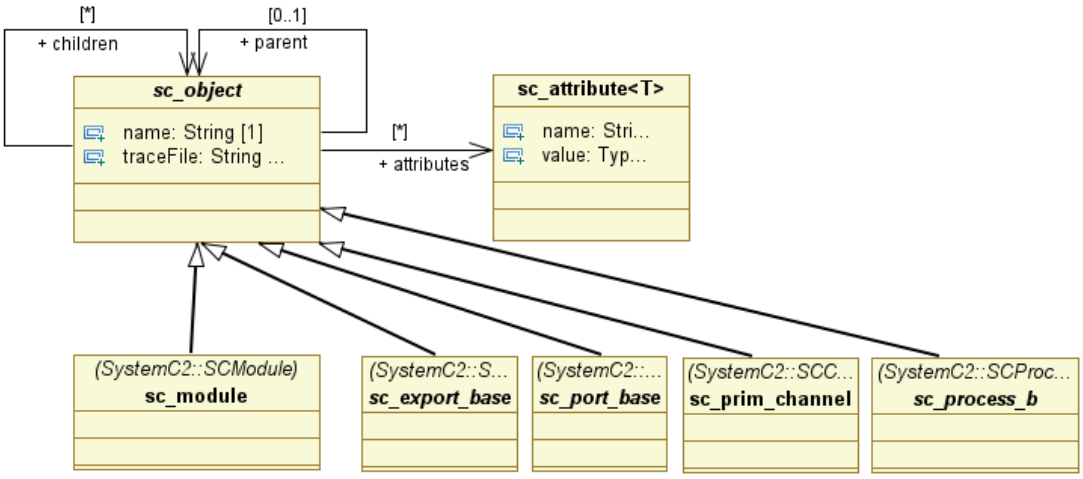


Figure 22: SystemC Module Hierarchy

Furthermore, modules are considered as the basic component entity in the component model of SystemC. Therefore they own ports for communication with other modules and SystemC processes (see Figure 23 for SystemC processes please refer to section 2.6.4) which are registered to the SystemC simulation kernel.

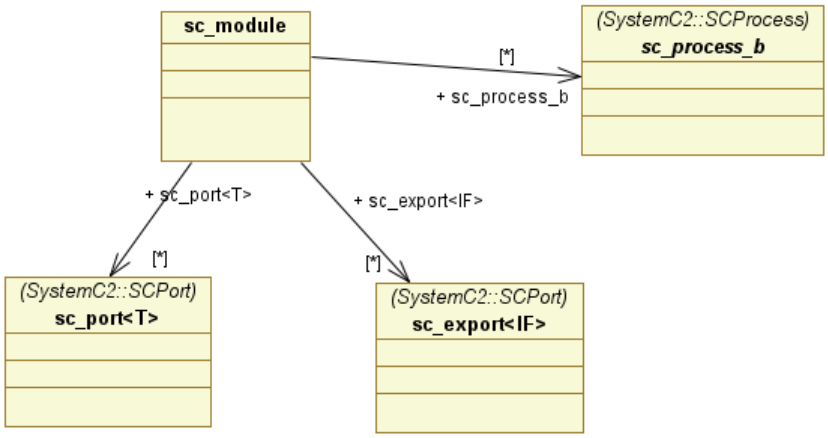


Figure 23: SystemC Module may contain ports, exports and processes

2.6.2 Ports

To underline the characteristics of a component model, a SystemC module can contain multiple ports (*sc_port*) which are a means for communicating with other modules. Ports refer to a SystemC interface which means that the functions which are defined in that interface can be called by the port (see Figure 24). The implementation of those interface functions is done in a *sc_channel*.

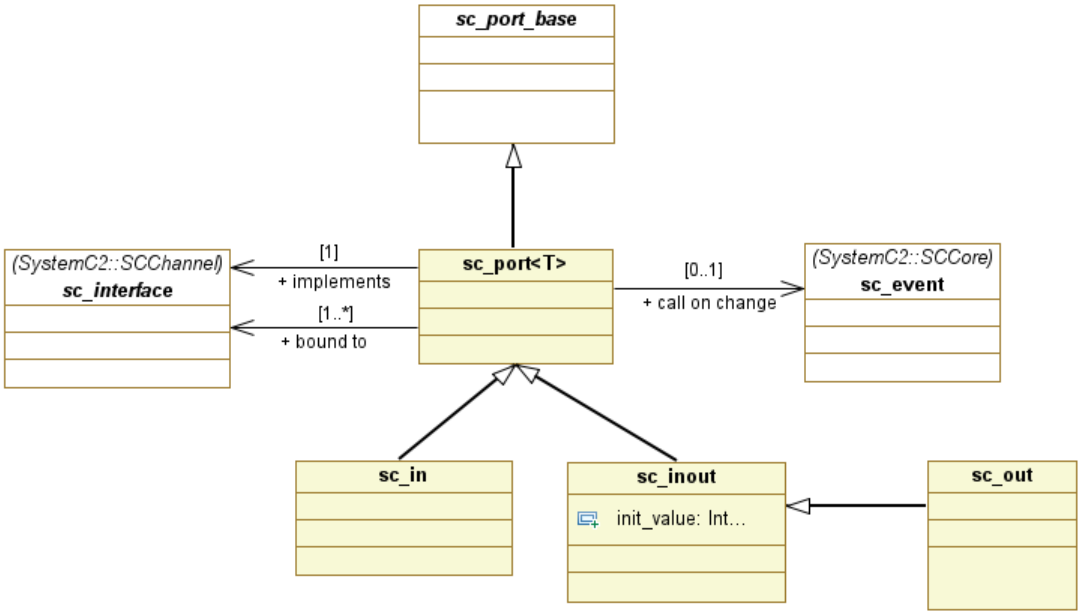


Figure 24: SystemC Ports enable communication between components utilizing interfaces

In SystemC, there exists also the concept of providing interface functions to another module through a *sc_export* element. This is in contrast to a port where the interface functions are called actively. The sc_export construct is mainly used in hierarchical designs where a module or a channel respectively provides some functions to its hierarchical module in which it is contained (see Figure 25).

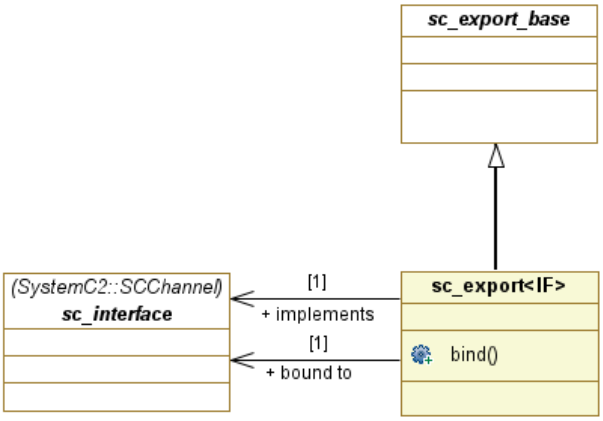


Figure 25: Providing Interface Functions in SystemC

Other SystemC elements for communication purposes are attributes (*sc_attribute*). They provide a direct access to variables which in fact are public C++ class members. In practice attributes are usually not used because they somehow contradict the object-oriented manner of C++ programming language by public access functions (getter/setter functions) and the modularity of components by port communication.

2.6.3 Interfaces

In general, interfaces in SystemC are native C++ interfaces which inherit from class *sc_interface*. They define the signature of functions without providing an implementation. As shown in Figure 26, SystemC ports refer to those interfaces indicating that the appropriate port is able to call the interface methods. The function implementations of interfaces are contained in a sc_channel.

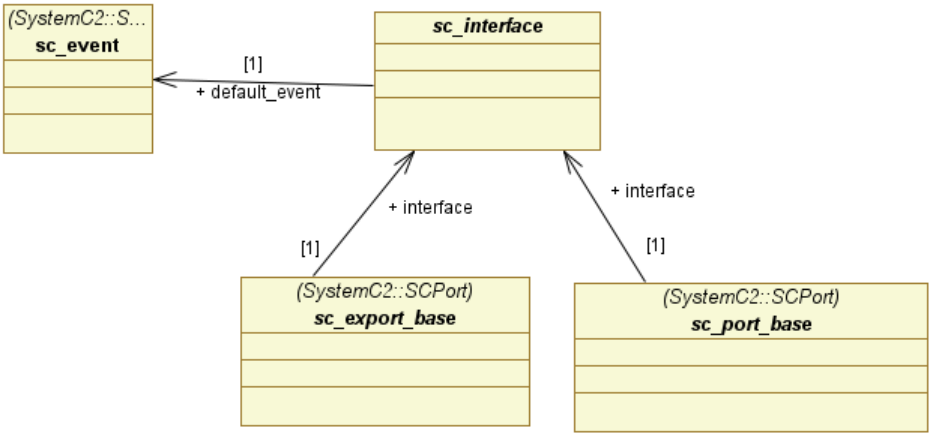


Figure 26: SystemC Interfaces provide the signature of functions for ports and exports

2.6.4 Component Implementations

The behavior of a SystemC module is defined by its processes. As depicted in Figure 27, SystemC processes are distinguished between *sc_method* and *sc_thread*.

Methods in SystemC are functions which cannot be interrupted and therefore consume no time (in the sense of simulated time). They are triggered by any event in their sensitivity list, e.g. receiving data at an input port or a simple event notification. This sensitivity list is statically defined for each method.

As opposed to methods, threads can be interrupted by calling a wait-operation. They are usually queued in the queue of active threads within the simulation kernel when the simulation of the design starts. Therefore, they usually contain at least one endless loop which is blocked at some point until a specific event occurs (dynamic sensitivity). During their execution they have to consume time either by calling a wait-operation with a certain time value (e.g. *sc_wait(10,SC_MS)*) or an execution inside of a clocked design. This means for example that a thread is activated each time a clock has its rising edge.

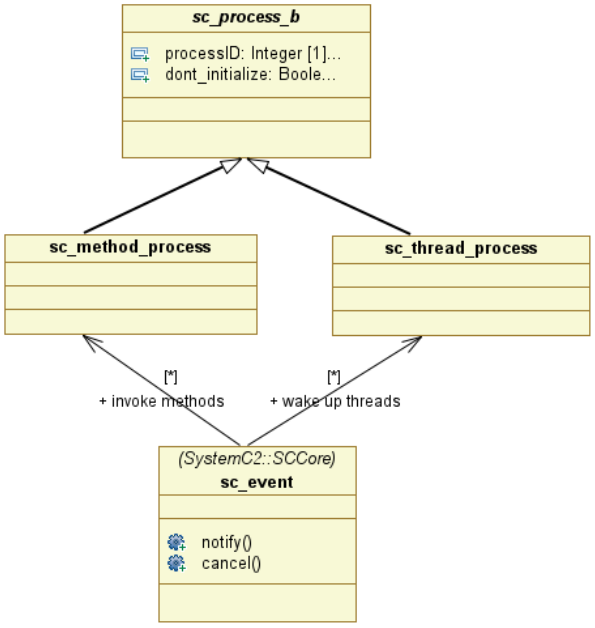


Figure 27: The behavior of SystemC Components is implemented in processes

By means of threads the concurrent behavior of SystemC is realized. Each time a thread is interrupted by a blocking call to the wait-operation the thread queue in the simulation kernel is arranged in execution order

and the next thread is activated. This jump to the next (runnable) thread might include a jump on the simulated time line.

2.6.5 Component Instances

Comparable to all C++ classes SystemC modules are instantiated by using the *new*-operator of the C++ programming language. Calling this new-operator executes the appropriate constructor method which allocates memory space for the newly created class instance. It is also possible to have several constructors in parallel which differ in their signature, e.g. different number of parameters or different types of parameters. Developers have to take care that instance memory space is released when the instance is deleted. Otherwise a memory leak might occur in the design.

2.6.6 Interaction Patterns

In SystemC, interaction between modules and their behavior respectively happens by means of communications via signals or channels (see Figure 28). Communications over signals are assignments of values to signal ports (*sc_in*, *sc_out*, *sc_inout*) using read- and write-methods. Signal ports indicate the communication direction. The super class of SystemC signals is called *sc_prim_channel* which embraces also mutual exclusion concepts (*sc_mutex* and *sc_semaphore*) and a basic first-come-first-serve buffered communication concept (*sc_fifo*). Transferred data types are passed as a template parameter in the channel specification.

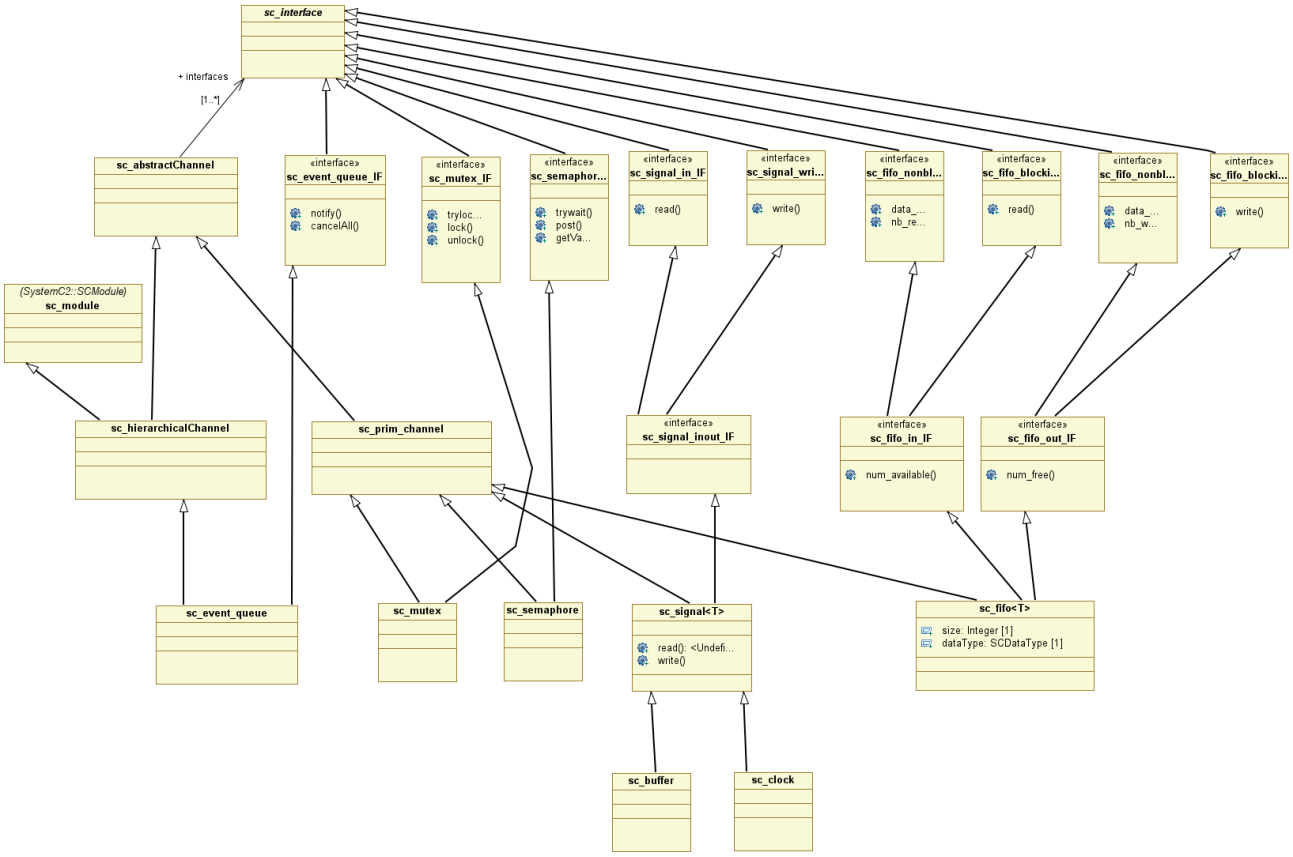


Figure 28: SystemC Interactions

Hierarchical channels may also contain a non-trivial behavior, e.g. thread execution. For a detailed description of System execution resources please refer to section 2.6.4. Channels implement a communication interface which means that they usually contain functions to send data to a channel, to elaborate and forward the data regarding a specific communication protocol, and to receive data from a channel.

However, it is not possible to explicitly model interaction patterns in SystemC. Instead, the behavior of an interaction (e.g. blocking or non-blocking communication patterns) is implicitly implemented in the communication functions inside a sc_channel and especially depending on the context of the function call inside the SystemC process. For example, a call to a send-function doesn't specify if the channel is actively waiting on an acknowledge event of the receiver side of the communication.

For synchronization purposes SystemC has the notion of events. Executed processes can be interrupted by waiting for a specific event or notify a processes which is actually waiting for an event. As an event notification is only discovered if a process is currently waiting for it, they are queued in an event queue.

2.6.7 Data Types

All C/C++ data types which are defined in the ANSI standard can be used within SystemC designs. Additionally, the SystemC standard defines data types which are more hardware-related (see Figure 29). SystemC allows the specification of the bitwidth of most standard C/C++ data types e.g. a 16 bit unsigned integer would result in a variable definition of *sc_uint<16>*. Furthermore, SystemC defines logic- and bit vector data types which are usually used to build complex entities in bit representation. Examples of those complex bit entities are frames of a specific communication protocol which are sent over the network as a bit stream and can logically be separated into frame fields e.g. destination address, user data, checksum.

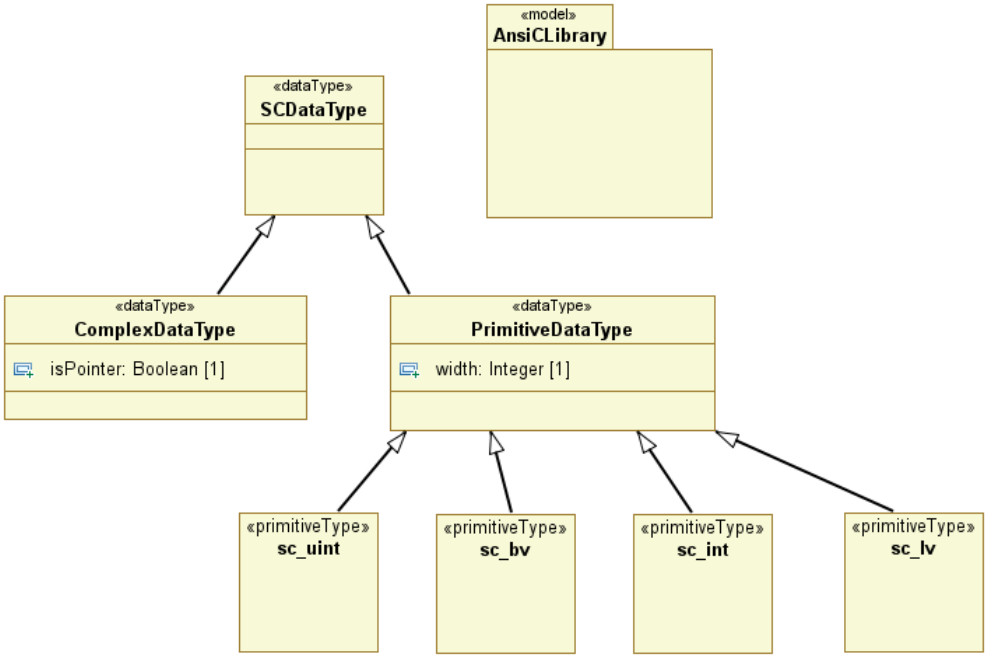


Figure 29: SystemC extends C/C++ with hardware-specific Data Types like Logic or Bit Vectors

SystemC enables modeling on various levels of abstraction. For a higher abstraction level, communication data is usually modeled as a whole transaction abstracting away multiple communications of simple data types. The transactions encapsulate these multiple data types into a complex data type which is implemented in a C struct or C++ class. In order to speed up the communication time, even pointers on complex data types can be transmitted. This is possible because the simulation is executed within the same address space, so pointers to data structures are valid across all entities of a SystemC simulation.

2.6.8 Deployment Allocation

There is no direct deployment allocation because SystemC does not differentiate between hardware and software execution. There is rather a separation of concerns (e.g. communication and computation) and a difference in levels of abstraction. Modeling hardware components imposes the use of signal communication and clock sensitivity, or appropriate techniques to abstract away low-level signaling. Composition of SystemC modules by binding ports to channels or ports to exports (see Figure 30) is done right after module instantiation and before the simulation start.

Software implemented in C++ can natively be executed inside a SystemC process. Both timing information reflecting the execution of this software on hardware and synchronization can be added using the techniques described in section 2.6.6. There is a common view on hardware and software in SystemC which means that it is used to model a whole system and simulate its behavior.

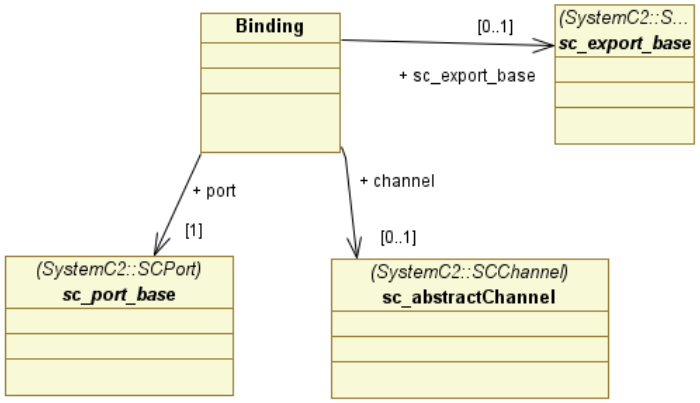


Figure 30: SystemC Composition is done by binding channels (e.g. signals) to (ex)ports

2.6.9 Execution Resources

There is a notion of system threads in SystemC which has already been described in section 2.6.4. Additionally, it is possible to create processes dynamically (during runtime of the simulation). This is done via a call to *sc_spawn*-function which needs to know the name of the function which is supposed to be spawned and optional parameters. Note that a sc_spawn-function must necessarily be called inside a sc_thread. Then, the spawned process is queued in the queue of runnable threads within the simulation kernel.

2.6.10 Non-functional Properties

Non-functional properties are not an integrated part of SystemC. There exists the notion of time inside a SystemC execution reflecting the elaboration of SystemC processes from a simulation kernel point of view.

2.6.11 Relationship between runtime and components

As the SystemC simulation kernel controls the execution of all registered threads there is a direct communication between the components (behavior) and the runtime.

2.7 AUTOSAR

AUTOSAR (AUTomotive Open System ARchitecture) describes a methodology for automotive software development. AUTOSAR [1] distinguishes between 3 software layers on the highest level of abstraction running on top of the ECU (Electronic Control Unit) hardware (see Figure 31). These layers are: the application layer, the runtime environment and the basic software. The key feature of AUTOSAR is the clear separation between application software and infrastructure (Basic Software). This leads to a platform independent software component development and enables the support of re-usability. Following, the chosen subset of the AUTOSAR software component template will be presented based on the comparison criteria presented in section 2.1. Hereby the main focus is on the application software layer[2].

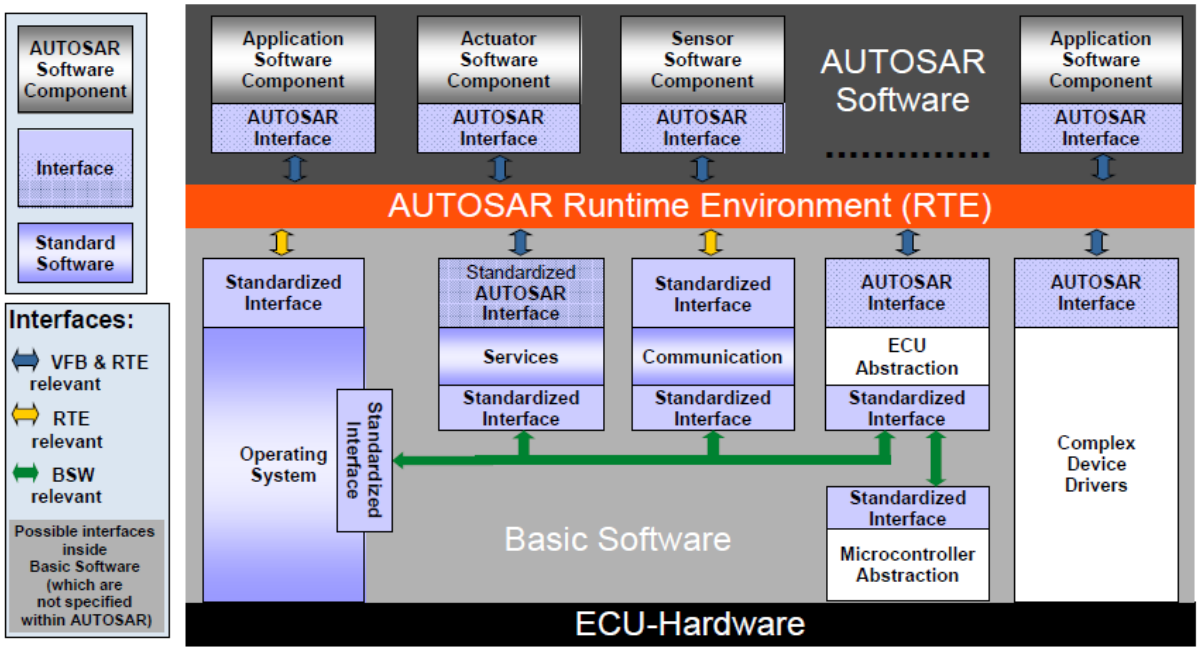


Figure 31: AUTOSAR software layers

2.7.1 Data types

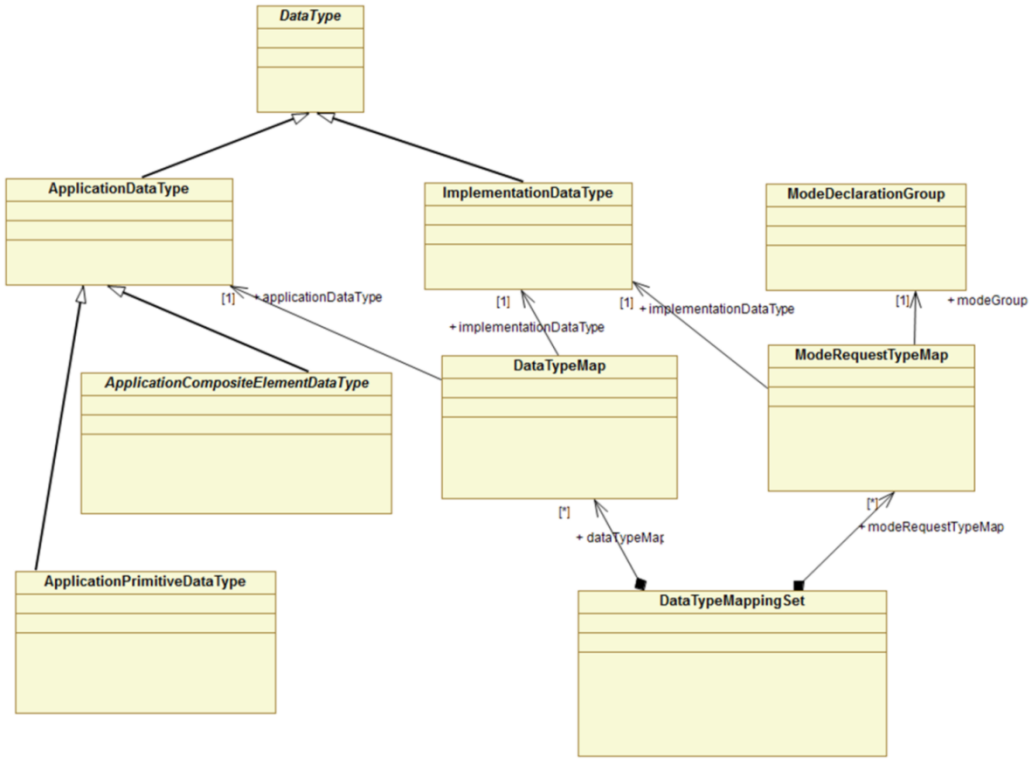


Figure 32: AUTOSAR Data types

AUTOSAR provides a set of predefined primitive and complex data types. There are two levels of abstraction: application and implementation level. At the application level common interface definition languages typically specify their data types by combining predefined primitive data types to form various user define types or structures, whereas at the implementation level the mapping of data types and data structures to bits and bytes is of primary concern. Primitive data types comprise types like integer signed and unsigned in 8, 16 and 32 bit version and float in 32 and 64 bit version. Additionally types like Boolean, Char and Strings also exist.

Composite data types can be for instance Array types, which are always associated with exactly one data type for all its elements or Record types, which contain an ordered set of record elements.

2.7.2 Interfaces

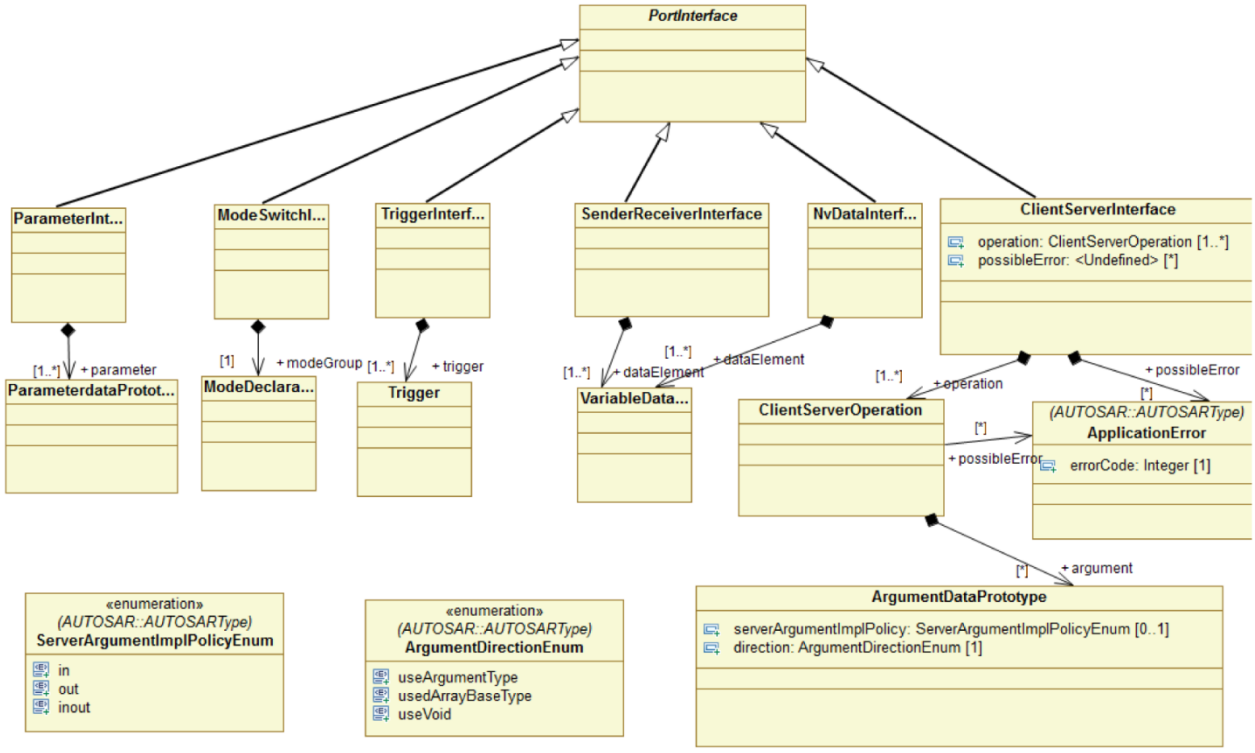


Figure 33: AUTOSAR Interfaces

An AUTOSAR interface defines the information exchanged between software components and/or basic software (BSW) modules. The description is independent of a specific programming language, ECU or network technology. AUTOSAR Interfaces are used in defining ports (see section 2.7.3) of SWC and/or BSW modules. Through these ports SWC and/or BSW can communicate with each other by sending, receiving data or invoking services. Furthermore AUTOSAR distinguishes three kind of interfaces Client/Server (C/S), where clients can execute operations on the server, Sender/Receiver (S/R) where primitive or composite data types are exchanged with several relations between senders and receivers (e.g.: 1:m and n:1) and Calibration through which calibration data can be requested.

2.7.3 Ports

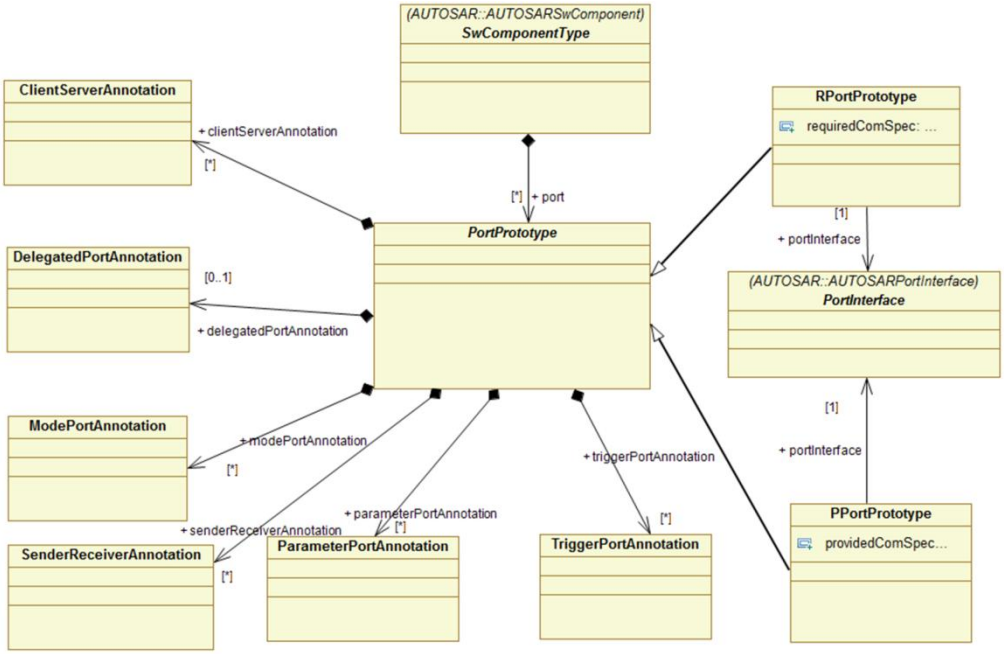


Figure 34: AUTOSAR Ports

AUTOSAR ports are the interaction points of software components. Hereby there are two types of ports, which are always bounded to an AUTOSAR interface: PPorts (provide ports) and RPorts (require ports). Additionally, ports are categorized in three different areas namely application software ports, calibration ports and service ports. Application software ports are used for the communication in the application layer through sender/receiver port or client/server port. Calibration ports are used to exchange calibration parameters via calibration provider or require port. Service ports are used by service modules of basic software to provide functions to other software components (sender/receiver port, client/server port).

2.7.4 Component types

An AUTOSAR software component (SWC) is a structural element that uses ports to communicate with the environment. Communication is done via ports (PPort, RPort). Furthermore, AUTOSAR distinguishes three kind of SWC namely atomic SWC, composition and sensor/actuator SWC. An atomic SWC encapsulates parts of the functionality of the application. They are called atomic because they cannot be distributed over several ECUs. Compositions are composed of several atomic SWCs. They are structural and used for abstraction purposes, therefore they are only used at the modelling level. Sensor/Actuator SWC are special software components which encapsulate the dependencies of application specific sensors and actuators.

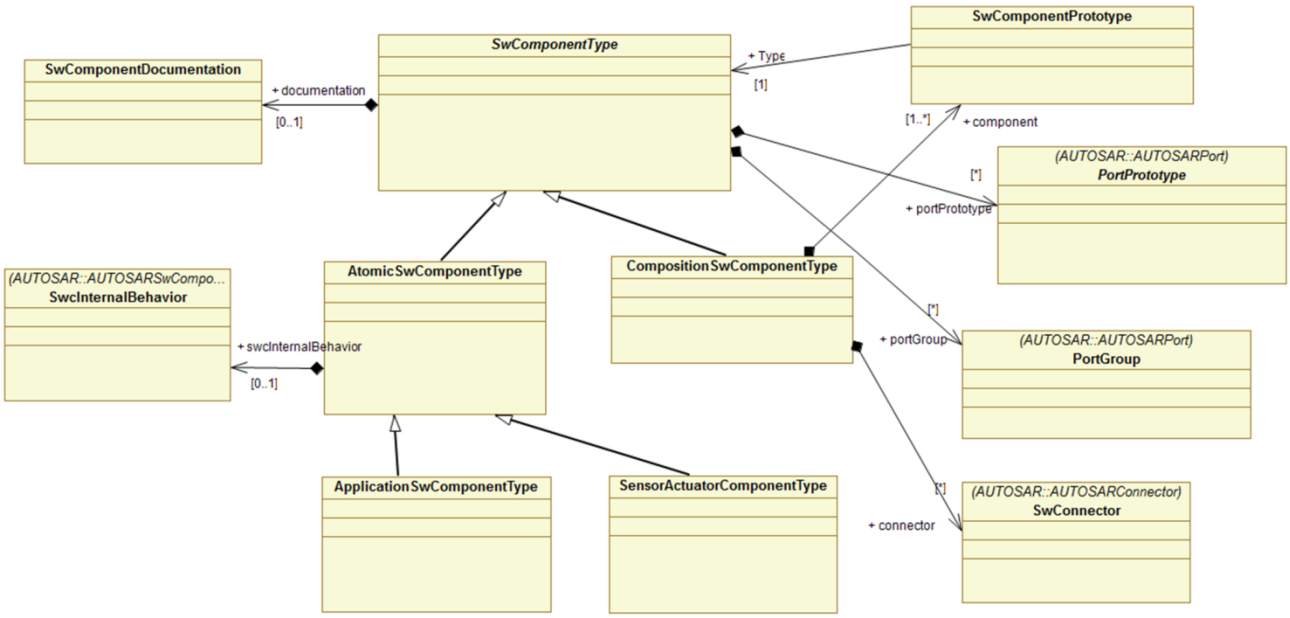


Figure 35: AUTOSAR Component types

2.7.5 Component implementation

The implementation of an AUTOSAR Software Component is independent from the underlying infrastructure. This means for example the type of ECU on which the AUTOSAR Software Component is mapped or the location of the other AUTOSAR Software Components with which the software component interacts. The behaviour of a SWC can be implemented by means of the so-called Runnables. A runnable is a code sequence in an Atomic SWC. It can be triggered either by a timing- or a data-related event. Furthermore runnables can implement server-operations, send or receive data through ports or even communicate with each other via the so-called “Interrunnable-Variables”. Basically there are two categories of runnables category 1 (Cat1) and category 2 (Cat2) (see Figure 36). Runnables of the category 1 can be subdivided into two further classes. They differentiate themselves by their runtime duration: Cat1A (short), Cat1B (finite) and Cat2 (infinite).

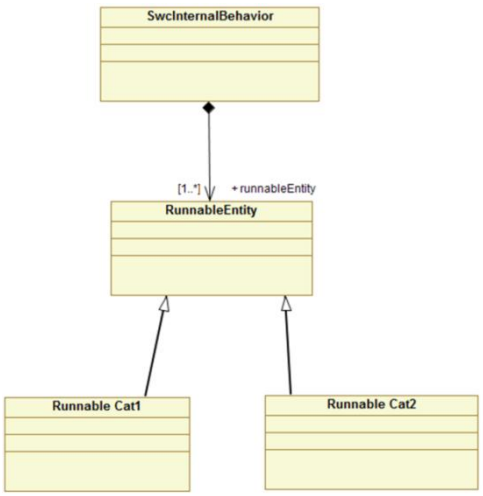


Figure 36: AUTOSAR Runnables

2.7.6 Interaction pattern (Virtual Functional Bus)

The central concept of AUTOSAR is the Virtual Functional Bus (VFB). The VFB abstracts the communication layer by encapsulating the underlying architecture (e.g.: CAN, FlexRay) of the Electronic Control Unit (ECU).

The concrete implementation of the VFB on an ECU is the runtime environment (RTE). At the modeling level, connections between the ports of the SWCs are modeled by means of the so-called connectors.

AUTOSAR provides two kind of communication pattern: Client-Server and Sender-Receiver. Moreover only ports with compatible interfaces can be connected with each other. During the Client-Server communication pattern, the server provides a service(s) that can be used by many clients (n:1).

On the other hand the Sender-Receiver communication pattern supports the multicast scenario (1:n) which is very efficient since the bandwidth on the senders side does not grow with the number of receivers. Additionally the multiple senders and one receiver scenario is also supported. Not only is the multiplicity of the communication relation coded at the VFB level but also the behavior of data transmission. Moreover Sender-Receiver communication can be either explicit or implicit:

- Explicit: data is sent or received explicitly by means of an API call. On the receivers side there is a buffer in form of a queue. Whereby the queue size influences the communication. For queue size = 1, the data semantic is “Last is Best” is applied. This is suitable for systems where only the last value is relevant, whereas for queue size > 1 the “FIFO” semantic is applied.
- Implicit: Data transmission or reception is not triggered directly by a communication call but is implicitly executed by the Runtime Environment. Data is forwarded after the sender´s runnable has executed and provided to the receiver before it starts its execution

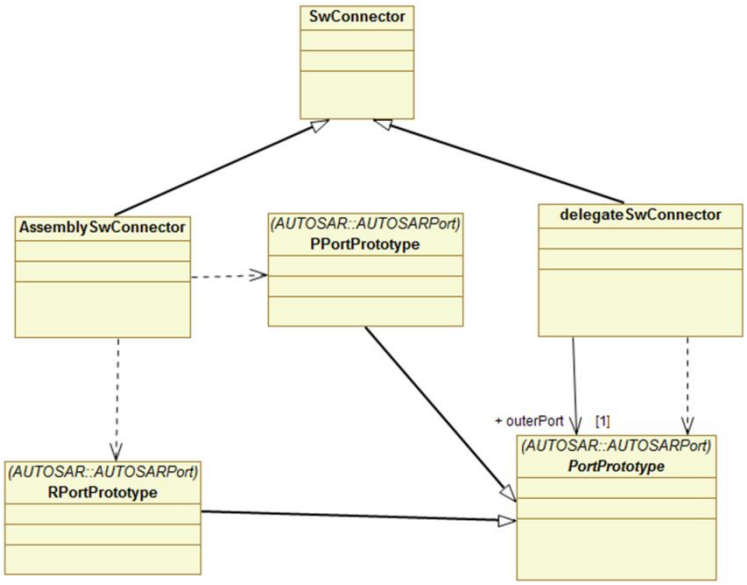


Figure 37: Interaction pattern, Connectors

2.7.7 Deployment allocation

Deployment is also a very important aspect in AUTOSAR. The AUTOSAR architecture distinguishes on the highest abstraction level between three layers as depicted in Figure 31: Application, Runtime Environment (RTE) and Basic Software (BSW) which run on a microcontroller. The Basic Software is further divided in the layers: Services, ECU Abstraction, Microcontroller Abstraction and Complex Drivers. The Basic Software Layers are further divided into functional groups. Examples of Services are system, memory and communication Services.

As afore mentioned, the application layer comprises two kind of SWC namely application software components and Sensor/Actuator components (hardware dependent). The Basic Software abstracts from the ECU-hardware and the Runtime Environment enables communication between software components and/or basic software.

2.7.8 Execution resources

Basic Software runs on top of ECU-hardware and is subdivided into further layers which are: Services layer (e.g. OS, Memory, Diagnosis and comm. functions), ECU abstraction layer, Microcontroller abstraction layer and Complex device drivers.

2.7.9 Non-functional properties (Timing)

AUTOSAR also does provide means to model Non-functional properties like timing. Timing related constraints can be formulated at the different level of abstraction: VFB-, ECU- and SWC-level by means of Timing event and Timing Event chains. There are two different interpretation alternatives while dealing with timing constraints. It can be either a restriction for the timing behavior of the system (e.g. minimum or maximum latency bound for a certain event sequence) or a guarantee for the timing behavior of the system (e.g. timing event is guaranteed to occur periodically with a certain maximum variation). A time event is an abstract representation of a specific system behavior, which can be observed at runtime. A

Time event chain describes the causal order for a set of functionality dependent timing event. Each event has a well defined stimulus and response, which describe its start and end point and can be hierarchically decomposed into an arbitrary number of sub-chains (event chain segment)

2.8 Summary

This chapter listed component models and deployment information of the different execution platforms used within VERDE, and this eventually leads to the definition of the VERDE modeling language. The abstraction towards the VERDE modeling language is a powerful approach. For example, within an initial system model it is not necessary to decide, whether a certain component is later on implemented in software (e.g. as an AUTOSAR component) or in hardware (modeled in SystemC). The approach based on the VERDE modelling language facilitates the development of complex and scalable systems, where deployment decisions can be done as late as possible, with cost criteria in mind. Based on the commonalities of the different execution platforms, if considered from a sufficiently abstract point of view, it can already now be seen that it is possible to define a common modelling approach across the specific execution platforms, which supports efficient code generation towards the execution platforms.

On the other hand, the VERDE language is not supposed to be a proprietary language; it is a subset of UML profiles, and hence can be understood and used by all project partners. In particular it is straight forward to recognize the link between a component in any of the execution platforms and a component in the VERDE language; the same is true for composition strategies. This chapter was the major groundwork for the definition of the VERDE language. The next chapter will investigate the potential UML profiles to be considered for the definition of the VERDE language.

3. Existing UML profiles

In this section, we will describe the different modeling languages that we will choose, in the scope of the comparison criteria.

3.1 MARTE

MARTE [5] is a profile for the UML2 [6] language dedicated to the definition of real-time embedded systems. MARTE consists of a set of sub-profiles dedicated for different aspects, globally divided into foundations, design, analysis and annexes. The foundation part defines general concepts such as non-functional properties and time. The modeling part contains concepts that are useful for modeling, e.g. the component model (GCM) already mentioned and high level application modeling (HLAM) concepts dealing for instance with time properties of service invocations. The modeling part also contains for instance the possibility to characterize the properties of hardware resource, e.g. bandwidth/jitter of busses. In the context of components, it is a useful to annotate target platforms (deployment) with these properties in order to enable timing analysis. The annex of MARTE standardizes common non-functional characteristics, for instance durations, frequencies or arrival patterns.
An interesting feature of the MARTE approach is that its generic component model (GCM) is a basis to model components for existing technologies, such as CCM and other component-based approaches, since the UML base model (structured classes with ports and internal composition) is shared by almost all component models. Some of the interaction types supported by GCM are linked with a specific way to treat invocations, as defined in the MARTE section high-level application modeling (HLAM). In the sequel, we give a short introduction to GCM and (a part of) HLAM.

3.1.1 GCM

The UML component model comprises the possibility that components have ports through which they can interact. UML ports are typed by a classifier. A port exposes a set of provided interfaces and a set of required interfaces. UML supports composite structures (hierarchal components): a component can own structural properties (also denoted as ``parts'') which are typed by another component1. It is possible to connect the ports of properties by means of a UML connector.

MARTE GCM extends the UML component model by adding two specializations of ports: client/server ports and flow ports. The former corresponds to “normal” method invocations. It can expose one or more provided or required interfaces as in standard UML.

In a client/server interaction, there is typically a single connection attached to a client port, since the semantics of a port which is connected with more than one provided interface is not defined (at least in case of return parameters2). Server ports have no restrictions on the number of connections.

A flowport enables a data-flow oriented communication, i.e. a communication in which a sender may publish a certain value and a set of consumers receive it. The type of this value may either be given directly as the port type. If data needs to be send and received through the same port or multiple data types are involved, the port is typed with a so-called flow-specification. The latter is a specific UML interface which contains only properties. Each of these has an additional parameter denoting the direction of the data flow (in, out or inout). We will examine a flow-specification in the context of an example presented below. Unlike a message port, a producing flow-port may be connected with multiple ports that consume the data. Data consumption is done either via polling or pushing, i.e. triggering the execution of a behavior.

A third interaction kind is the emission and reception of signals. This possibility is not done by means of a different kind of port. Instead, an interface may contain a reception (referencing a signal) in addition to

[1] To be precise: UML is more generic, properties have a Type which is a generic superclass for many meta-model elements including Interface, Class and Component

[2] In the CCM model, these ports are called multiplex receptacles. But the calling implementation has to select a single reference from the set of connections and make a single call

operations. If the signal is in a provided interface, it corresponds to signal reception; if it is in a required interface, it corresponds to signal emission (note that the UML term Reception is misleading, since it applies only to the consumer role).

It is possible to type both flow-ports and message ports with an interface containing a signal. Since the semantics is identical in both cases, we assume in the context of this paper that only flow-ports are typed with a signal.

3.1.2 HLAM

The high level application modeling (HLAM) chapter of MARTE allows the specification of temporal properties of calls. This comprises in particular the modeling of active objects (active components), i.e. of entities that provide their own thread of control. Invocations of these objects make use of an additional specification that defines real-time characteristics (RtFeature), such as a ready-time and deadline. The RtFeature also contains an arrival pattern that allows for specifying periodic invocations. Component or classes may be stereotyped with two stereotypes that specify a different concurrency behavior: protected passive («PpUnit») and reactive/real-time («RtUnit») objects. Both own real-time services, i.e. specializations of services which have addition real-time properties.

Protected passive units are executed by the calling thread and have to specify a concurrency policy. There are three concurrency policies: the first authorizes concurrent access, the second only sequential access assuming that the caller will respect the access policy. The last policy, named «blocked», blocks concurrent requests on the level of the passive unit. Only this policy requires specific execution support.

Reactive or real-time («RtUnit») components are executed by their own thread (Schedulable resource). Different policies control thread pool behavior and scheduling policies for these objects. In particular the ConcurrencyKind attribute of a RtService is evaluated to enable one writer, n-reader synchronization protocols.

An RtFeature may be applied to a port (among other elements), it is thus possible to specify the temporal behavior of a GCM port. The subset of real-time features that are useful (in the sense that they are respected by code generation) depend on the port kind. For instance, a miss-rate is useful for a data reception, but not for an invocation.

3.1.3 Non functional properties (NFPs)

The modeling of non-functional properties is an important part of MARTE. This Information may be used for code-generation as well as validation and verification purposes, for instance for performance prediction and resource usage evaluation.

The NFP sub-profile is based on the QoS Profile. It offers possibilities to model (physical) values by means of a magnitude and a unit. This allows modelling for instance energy, data size and duration.

There are four main stereotypes: Nfp, Nfptype, NfpConstraint and Unit. It is important to note that specific units are not part of the profile itself but defined in a model-library containing nfpTypes such as time-units (ms, us), power, frequency and duration. The available units are therefore extensible in order to accommodate the needs of a specific domain.

The Value Specification Language (VSL) allows the specification of NFP types. It is basically a grammar enabling a textual specification of values for NFP types (the underlying meta-model is based on UML DataTypes and ValueSpecifications). For instance durations are specified by a tuple of magnitude and unit, as in (4, ms) (a value of type NFP_Duration which is part of the MARTE model library). These expressions are composable, as in case of the ArrivalPattern type which is also defined in the standard MARTE model-library:

Periodic (period=(2.0,ms), jitter=(30,us)) specifies periodic events with a period of 2 ms and a jitter of 30 us.

The use of non-functional properties is a basic mechanism within MARTE which is combinable with other UML based modeling approaches. For instance, it is well possible to annotate a model using NFPs values while using a different component model.

Since MARTE is a UML2 profile, it inherits its ability to describe dynamic behavior using for example statecharts. This means that MARTE provides an interesting approach to describing both the architecture of the application and (part of) its dynamic behavior in a single modeling tool.

3.2 SysML

This paragraph introduces the SysML Block concept and ports which can be seen as the component concept in the language.

3.2.1 SysML Block

A Block is a modular unit that describes the structure of a system or element. It may include both structural and behavioral features, such as properties and operations, which represent the state of the system and behavior that the system may exhibit. Some of these properties may hold parts of a system, which can also be described by blocks. A block may include a structure of connectors between its properties to indicate how its parts or other properties relate to one another.

SysML blocks are based on UML classes as extended by UML composite structures. Some capabilities available for UML classes, such as more specialized forms of associations, have been excluded from SysML blocks to simplify the language.

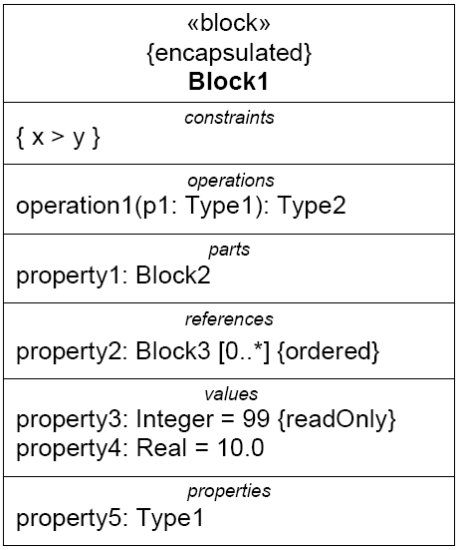


Figure 38: Block Definition Diagram

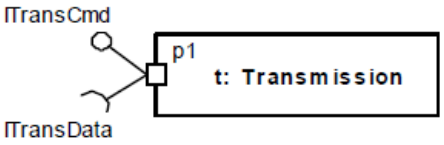
The Block Definition Diagram in SysML defines features of blocks and relationships between blocks such as associations, generalizations, and dependencies. It captures the definition of blocks in terms of properties and operations, and relationships such as a system hierarchy or a system classification tree. The Internal Block Diagram in SysML captures the internal structure of a block in terms of properties and connectors between properties. A block can include properties to specify its values, parts, and references to other blocks. Ports are a special class of property used to specify allowable types of interactions between blocks.

3.2.2 SysML ports

In SysML there are 3 kinds of ports: Standard, Flow and ItemFlow Ports.

Standard Ports

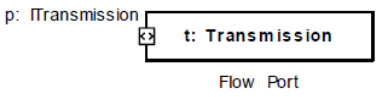
A standard port specifies the service the owning block provides (offers) to its environment as well as the services that the owning block expects (requires) of its environment. The specification of the services is achieved by typing the standard port by the provided and/or required interfaces. In general standard ports are used in the context of service-oriented components and/or architectures, either when specifying software components or applying a service-based approach to system specification. Standard ports typically contain operations that specify bidirectional flow of data, so they are typically used in the context of peer-to-peer synchronous request/reply communications. A special case of a service is signal reception, which signifies a one-way communication of signal instances, where the handling of the request is asynchronous.



A block can call operations and\or send signals through its behavioral ports that have required interfaces. A block must implement all the operations specified in its behavioral ports provided interfaces. Also, a block must react to all the signals specified in its behavioral ports provided interfaces. Non-behavioral ports delegate operations and signals to/from their internal parts over internal connectors between the non-behavioral ports and the internal parts.

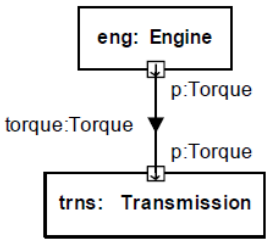
Flow Ports

A flow port specifies the input and output items that may flow between a block and its environment. Flow ports are interaction points through which data, material, or energy can enter or leave the owning block. The specification of what can flow is achieved by typing the flow port with a specification of things that flow. This can include typing an atomic flow port with a single type representing the items that flow in or out, or typing a non-atomic flow port with a flow specification which lists multiple items that flow. A block representing an automatic transmission in a car could have an atomic flow port that specifies “Torque” as an input and another atomic flow port that specifies “Torque” as an output. A more complex flow port could specify a set of signals and/or properties that flow in and out of the flow port. In general, flow ports are intended to be used for asynchronous, broadcast, or send-and-forget interactions.



Items Flows

Item flows represent the things that flow between blocks and/or parts and across associations or connectors. Whereas flow ports specify what “can” flow in or out of a block, item flows specify what “does” flow between blocks and/or parts in a particular usage context. This important distinction enables blocks to be interconnected in different ways depending on its usage context.



Item Flow with an Item Property

3.3 FCM

The Flex-eWare Component Model (FCM) is a common meta-model derived from Fractal and CCM that has been defined in the scope of the ANR project Flex-eWare (www.flex-eware.org). It is a generic model that is extensible via model libraries, in particular it is possible to specify new ports, connectors (types & implementations) and container services in a library. The extensible elements include:

- Port: ports are characterized by a *kind* which defines whether the port is used as a data producer in a dataflow interaction of a client in a client/server interaction. From a programming language viewport, a port is represented by the interfaces it provides (for external use) or requires (internal use) from others. These are automatically derived from the port kind and type. Extensible port kinds have been standardized in the context of the DDS for CCM specification.
- Containers: As CCM, FCM supports enclosing a component by a container. A set of container services manipulate interactions or provide additional services.
- Connectors: FCM does not provide a fixed set of interaction patterns; rather, it provides a means to specify new connection point types (aka connectors) that define a specific communication pattern. The realization of the communication pattern is left to the connector implementation. A connector type captures a communication pattern while a connector implementation is a specific realization of a communication pattern.

With these extension mechanisms, it is for instance possible to express MARTE GCM within FCM by means of a dedicated MARTE library, as proposed in [7] and implemented by eC3M.

FCM models are typically enriched with non-functional properties specified by means of MARTE's value-specification-language (typically specifying the values of standard MARTE NFP types).

A UML profile that corresponds to the concepts of the FCM meta-model has been defined. We consider this profile in the scope of the VERDE project.

4. VERDE model

In this section, we define the VERDE Modeling Language (VERDE ML). This language must address all the concerns that were identified and explained in section 2. In order to avoid redefining yet another modeling language, we choose to rely on existing UML standards and profiles described in section 3. In particular, we rely as much as possible on MARTE; the other profiles are used for aspects that are not covered by MARTE. Our goal is to define the VERDE modeling language as close as possible to the MARTE profile.

The VERDE ML defines a set of modeling patterns that match the different aspects of the different technologies involved in VERDE (LwCCM, SCA, SystemC and AUTOSAR). These modeling patterns are to be used together with other patterns corresponding to other concerns (e.g. analysis, testing).

The following modeling patterns are to be used as support for the VERDE methodology described in document F2.2.2. They provide modeling solutions to address the modeling requirements identified in section

5. The patterns either use UML class diagrams and composite structure diagrams and/or their SysML equivalents.

4.1 Data Types

Data types used for the VERDE language are the MARTE data types, plus some additional definitions. In this section, we describe the types that are required with respect to the comparison criteria of section 2.1. However, all MARTE data types, including those that are not mentioned in this section, are part of the VERDE ML.

4.1.1 Basic Data Types

Basic data types are predefined and shipped in a model library, so that they can be referenced by models. The Verde ML mostly relies on the types defined in the MARTE library.

Integer

The integer type is defined in the MARTE library, in **MARTE_Library::MARTE_PrimitiveTypes::Integer**. This type does not specify data size.

Float

The float type is defined in the MARTE library, in **MARTE_Library::MARTE_PrimitiveTypes::Real**. This type does not specify data size.

Boolean

The Boolean type is defined in the MARTE library, in **MARTE_Library::MARTE_PrimitiveTypes::Boolean**.

Clock tick

The clock tick type is used to model discrete time, especially in hardware architectures. For this type, the VERDE ML uses **MARTE_Library::BasicNFP_Types::NFP_Duration**. The attribute unit represents the measurement unit. (e.g. MARTE unit “tick”).

Logic state

Logic states are used for hardware systems, and consist of four values: high (1), low (0), undefined (X) and high impedance (Z). The MARTE library does not provide such kind of type. Therefore, it is defined in the VERDE library by an enumeration: **VERDE_Library::PrimitiveTypes::LogicState**.

4.1.2 Parameterized Data Types

Integer Range

Integer range must be defined using a UML DataType and by applying stereotype *BoundedSubtype* from the MARTE profile. The tag definition baseType of stereotype *BoundedSubtype* must reference the MARTE integer. The tag definitions minValue, maxValue, isMinOpen, isMaxOpen define the range.

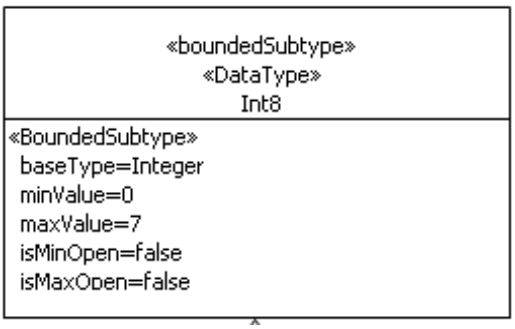


Figure 39: Integer datatype of 8 bit length

Float Range

Float range must be defined using a UML DataType and by applying stereotype *BoundedSubtype* from the MARTE profile. The tag definition baseType of stereotype *BoundedSubtype* must reference the MARTE real. The fields minValue, maxValue, isMinOpen, isMaxOpen define the range.

4.1.3 Complex Data Types

Alias

Alias types in Verde ML are data types that inherit another data type without adding any information. Thus, they are defined using a UML DataType and an inheritance link to the data type to be aliased. No additional information is allowed.

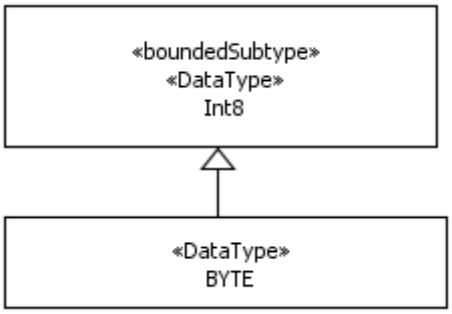


Figure 40: An alias of a predefined datatype

Enumeration

Enumerations in the Verde ML are plain UML enumerations.

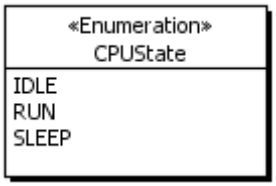


Figure 41: CPU states as enumeration literals

Data Structure

Data structures are modeled using UML DataType with the MARTE stereotype *TupleType*, from MARTE::MARTE_Annexes::VSL::DataTypes. The field tupleAttrib of *TupleType* is the list of the elements included in the structure. The elements are UML properties with simple data types and should be properties of the UML Datatype.

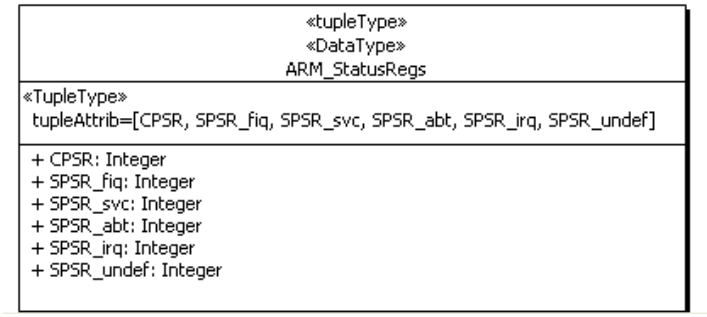


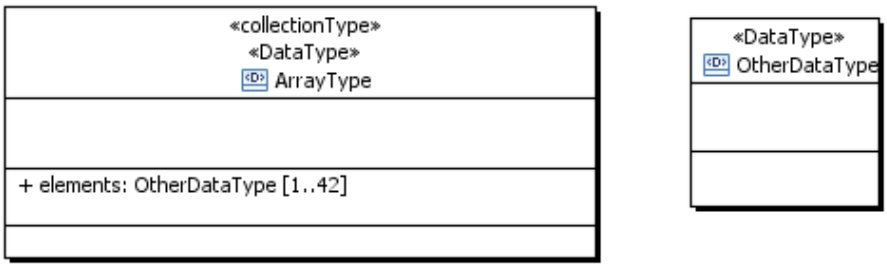
Figure 42: ARM 32-bit status registers

Union

Unions are modeled using UML DataType with the MARTE stereotype *ChoiceType* from MARTE::MARTE_Annexes::VSL::DataTypes. The field choiceAttrib of *ChoiceType* is the list of the elements the union. These elements are UML properties, that is, subelements of the union DataType that represent the union members.

Array

Arrays are modeled using UML DataType with the MARTE stereotype *CollectionType* from MARTE::MARTE_Annexes::VSL::DataTypes. The value of field collectionAttrib of *CollectionType* is a UML property (that is, a subelement of the DataType) correspond to the data type stored in the array, with an arity that indicates the size of the array.



4.2 Component Model

The VERDE component model defines components and ports (with the interfaces to be associated with these ports, if needed). It thus covers the definition of the architectural aspect of the software application. All these definitions correspond to declarations, and thus are made in class diagrams. These declarations are to be instanticated in the deployment (section 4.4).

4.2.1 Interfaces

Verde ML entirely relies on UML for the definition of interfaces. One should use a UML interface with UML operations. In class diagrams a realization and usage relationship denote the dependency to UML components as shown in Figure 43.

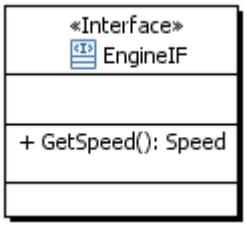


Figure 43: Interface modeling using UML class diagram

4.2.2 Component types

A component type represents the definition of the component as seen from outside the component. It therefore covers the definition of the functional interactions that the component is supposed to provide or require, that is, the declaration of interaction ports (see below in section 4.2.4).

Component types are UML components identified using the standard UML stereotype *Specification* applied on UML components. No additional information is required. Only component types should have ports.

4.2.3 Component Implementations

A component implementation represents the internal structure of a component: how the functional interactions defined by the component type are actually implemented. It typically contains algorithms or state machines. The specification of the component behaviour encapsulated in a component implementation is not in the scope of the structural part of the Verde ML, and therefore not described in this deliverable. Information about that can be found in deliverables of work packages 4 and 5, but Verde does not force users into using a specific way of modelling component behaviours.

Component implementations are UML components identified using the standard UML stereotype *Implement*. No additional information is required. The relationship between a component implementation and the corresponding component type is inheritance.

Using inheritance between component implementation and component type allows the preservation of ports, which are defined at the component type level. No port should be defined in component implementations.

It is also allowed to stereotype a UML component with both stereotypes *Specification* and *Implement*. This allows a simplification of the architecture model if there is only one implementation for a given component type.

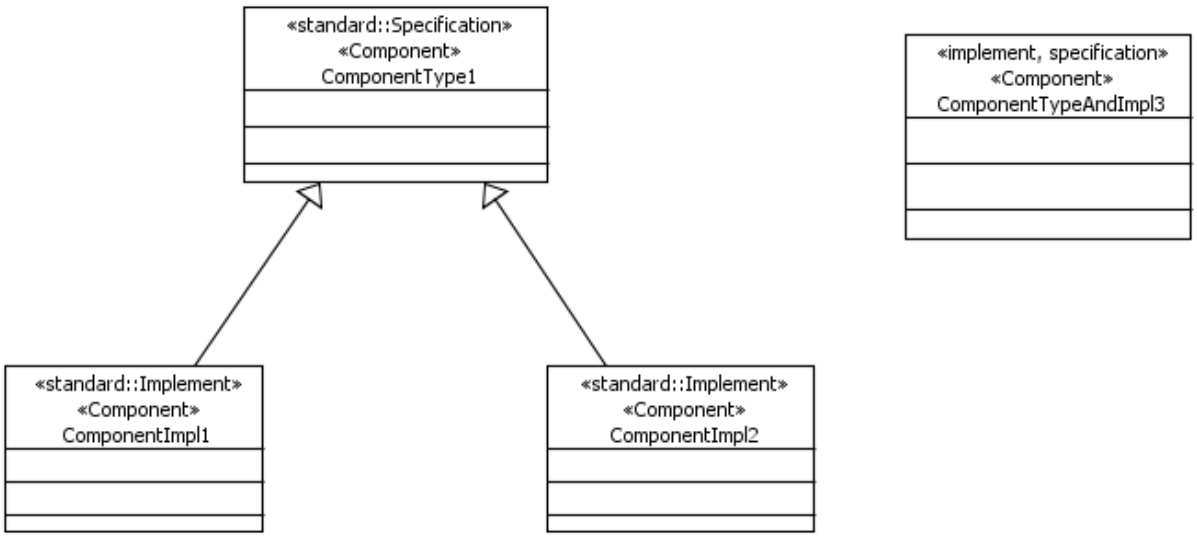


Figure 44: Example of componentdeclarations

4.2.4 Ports

Ports represent interaction points that are associated with components. The behaviours implemented in a component only communicate with the component outside through ports.

Ports in Verde ML are UML ports (see the UML standard, section on composite structures) on which MARTE stereotypes are applied. Here we describe the two communication mechanisms that are used for Verde: operation calls and message passing. MARTE provides other stereotypes, that could be used as well to model other communication mechanisms.

Operation calls

Operation calls are modelled using an interface (see section 4.2.1) that is used between a server and a client. The client component type has a port that requires the interface; the server component type has a port that provides the same interface.

Communications based on service calls are modelled by ports with stereotype *ClientServerPort* from MARTE. A MARTE client server port can provide and/or require several interfaces. In order to specify interfaces that are provided and required, some attributes of the stereotype should be filled in as shown partly in Figure 45.

The attribute /specificationKind must identify the interface-based usage of the port. The attribute kind indicates whether the port provides and/or requires interfaces. The attributes reqInterface and provInterface list the interfaces that are required or provided. Their semantic should be consistent with the value of field kind.

As the interfaces are specified using properties of the MARTE stereotype, no type should be associated with the port itself.

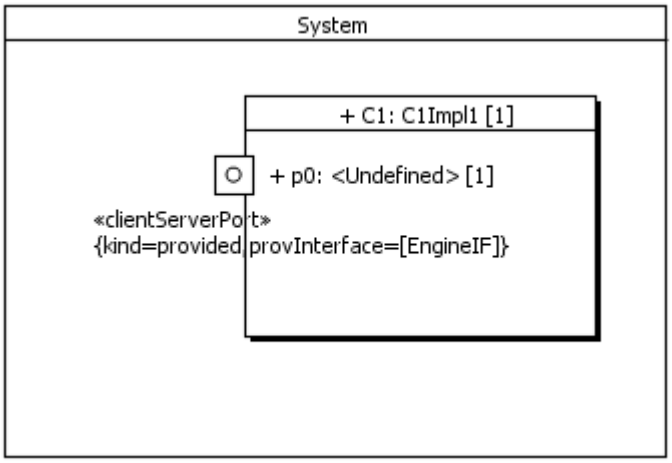


Figure 45: MARTE port that provides an interface in a component instance

Messages

Message passing corresponds to a data type sent by a producer and received by one or several consumers. The producer component type has a port that sends the data; the consumer component types have ports that receive the data. The notion of data covers many things, like signals, events (signal and actual data), etc.

Communications based on message passing are modelled by ports with stereotype *FlowPort* from MARTE. The direction tag definition of *FlowPort* must be defined accordingly. The type of data transmitted through the port is to be specified by a UML signal or a UML dataType associated with the UML port. A UML signal should be preferably used to model message passing, as a signal carries data as well as the notion of emission or reception. A UML DataType should be used to model data sent without notification to the receivers.

4.2.5 Specific Interaction Patterns

The modeling of specific interaction patterns (e.g. synchronous/asynchronous calls of CORBA, AUTOSAR-COM communication, but also simple fifos) should rely on the concept of connectors.

One solution for the VERDE ML is to rely on FCM connectors. A FCM connector is a specific kind of component identified by FCM *ConnectorType* and *ConnectorImpl* and thus not to be confused with UML connectors. A FCM connector shares all properties of FCM components, i.e. connectors have ports and there is a separation between type and implementation.

Another solution is to pre-define connectors and port kinds in a specific UML profile. In this situation, the VERDE ML is then combined with this additional UML profile.

4.3 Execution Platform Topology

An execution topology model covers the hardware execution platform as well as software execution resources and their mapping. Verde ML relies mainly on the MARTE Detailed Resource Model (DRM) to describe all these elements.

4.3.1 Computation Nodes

Computation nodes, i.e., the hardware platform, are modelled using SysML blocks or plain UML classes refined by MARTE stereotypes. For this, the VERDE ML relies on MARTE's Hardware Resource Model (HRM) and supports the definition of different abstraction levels.

SysML blocks stereotyped by *HwComputingResource* correspond to abstract active processing resources. In addition, more precise MARTE stereotypes like *HwMemory*, *HwStoragemanager*, *HwCommunicationResource*, *HwTimingResource* and *HwDevice* are used to describe the complete platform model as shown in Figure 46.

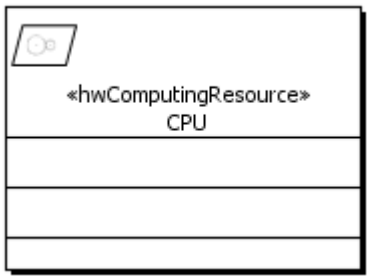


Figure 46: Computation node declaration

At a more detailed level one may use more precise MARTE stereotypes to refine these abstract hardware resources (e.g. *HwProcessor*, *HwASIC*, or *HwPLD* for specializing HwComputingResources). But this specific information is not required regarding the component modeling.

4.3.2 Execution Resources and Mutual Exclusion Resources

In the Verde modelling approach execution resources should be explicitly modelled, so that scheduling analysis is possible. It is of course possible not to model them, if the targeted execution platform does not have this notion: for example, a middleware-based platform that would manage execution resources by itself.

Software execution resources (i.e. operating system threads) are modelled using UML components with stereotype *SwSchedulableResource* from the MARTE Software Resource Model (SRM) as shown **Erreur ! Source du renvoi introuvable.**. Two attributes must be filled in: type and schedParams. Attribute type indicates the periodicity of the schedulable resource (periodic, aperiodic, etc.); it is described in section 14.1.5.19 of the MARTE standard. Attribute schedParams is a list of supported scheduling configurations (fixed priority, earliest deadline first, etc.); see section 10.3.2.14 of the MARTE standard.

Software executions resources must be executed within a virtual address space (i.e. in a process context). Virtual address spaces are modelled by UML components with stereotype *MemoryPartition*.

Mutual exclusion resources can be used to synchronize mutual access to shared data. Such resources are modeled by UML components with stereotype *SwMutualExclusionResource*. The attribute mechanism specifies the kind of resource (e.g. semaphore or mutex). The attribute isIntraMemoryPartitionInteraction determines if the resource is accessible from different memory partitions or not.

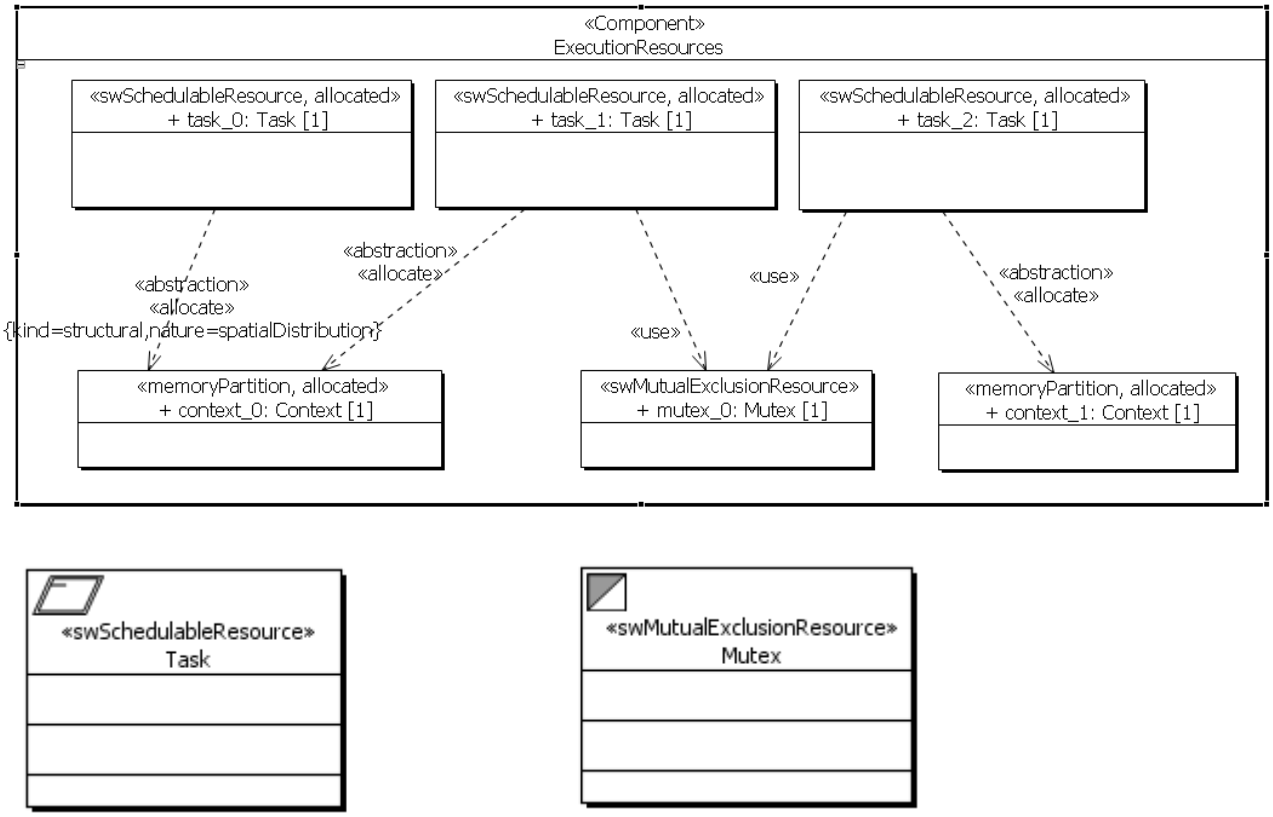


Figure 47: Execution resource modeled using Composite Structure Diagram

4.3.3 Allocation of Software Execution Resources

The allocations of software execution resources are modeled using UML abstractions with stereotype *allocate*. For example in **Erreur ! Source du renvoi introuvable.**, schedulable resources are bound to memory partitions using this construction. The same thing applies for the allocation of memory partitions to processing resources of the hardware nodes as in Figure 48. UML abstractions must be used in composite structure diagrams, thus connecting UML properties that represent instances of execution resources, computation nodes, etc.

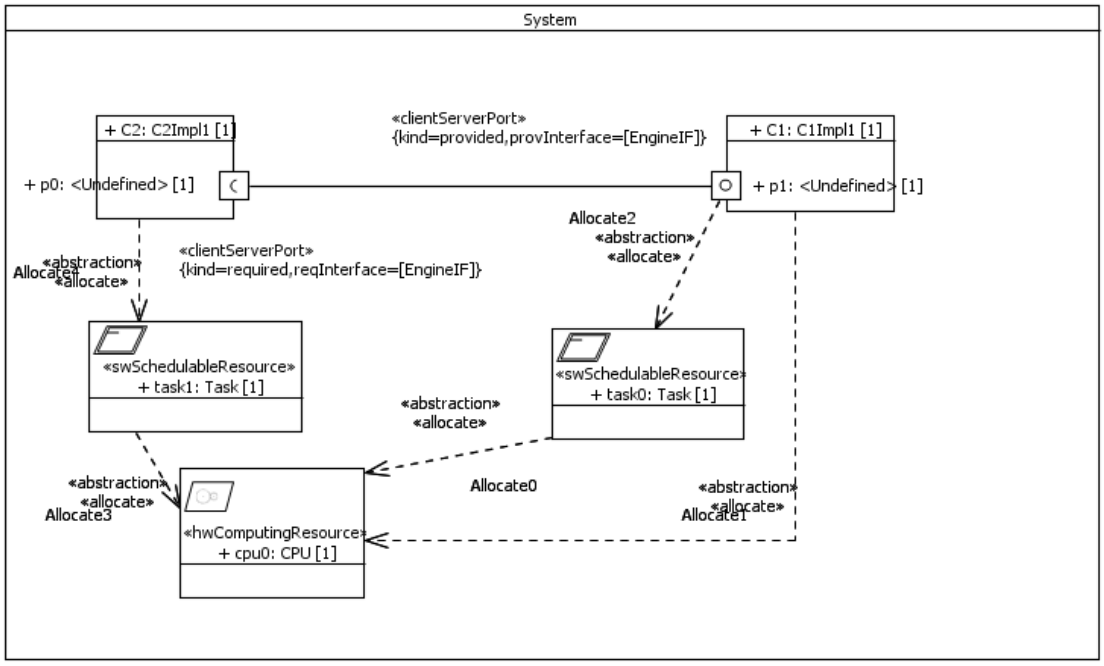


Figure 48: Resource allocation on hardware execution platform, allocation of components on execution resources, and component port connections

4.4 Component Deployment

4.4.1 Architecture Breakdown

An architectural breakdown of a system consists in decomposing system elements into sub-elements. This can apply to either functional elements (i.e. components) and/or hardware execution topology elements (i.e. computation nodes). Several situations can be considered, depending on what we want to describe.

In the case of the definition of the system that nest several elements (subsystems, hardware topology elements, etc.), one should use SysML blocks modelled with SysML Block Definition and Internal Block Diagrams. For further refinement/description of the hardware topology elements (for example, the internal structure of a computation node as shown in Figure 46), one simply has to add the stereotypes from MARTE described in 4.3.1.

In the case of a functional component that is actually decomposed into several subcomponents, one has to use a component implementation derived from a component type. An example is shown in Figure 44. In this situation, composite structure diagrams will be used. Note that a component type itself cannot contain subcomponents, as it only defines the component boundary, i.e., the functional interface.

4.4.2 Top-level global system

System architecture is defined as a global system which is decomposed into different systems with different functionalities, thus creating a tree structure. As a consequence, a system architecture model is an architecture breakdown of a top-level system. Tools will process this top-level system, which is thus typically a SysML block, a plain UML component or possibly a UML class. In the case of a UML component, the component should not be stereotyped like a component type or a component implementation, as it is not to be considered as a functional component. These constructions must be placed outside any UML package; that is, they must be direct subelements of the UML model root. The rationale for this is to ease the

processing by tools. If the designer wants to have his root component declared within a UML package, he just has to create additional top-level components that simply contain the actual root component.

There can be several top-level systems in a UML model, representing alternative deployment plans.

4.4.3 Component instance

Component instances like C1 and C2 in Figure 48 are modelled by UML properties that are associated with a component implementation. This is done in composite structure diagrams.

Please note that only a property of the specific system composite corresponds directly to an instance, since the system class is instantiated exactly once. If the property belongs to another composite, it only represents a set of potential instances, since the composite might be instantiated not at all or several times within the system. E.g. consider a component representing points and a composite that defines the bounding box of objects by means of two points, i.e. two properties “a” and “b” typed as points. A system might very well contain a set of bounding box components and “a” and “b” do not correspond to a specific instance. Yet, in many cases, composite classes exist exactly once within the system and the simplification above holds. If not (and if it is required to distinguish the instances, e.g. since they require different configuration and/or allocation) it is an option to represent instances by “real” UML instances, i.e. UML instance specifications. These may be generated from the hierarchical structure defined by the composites and the contained properties.

4.4.4 Component Allocation

The allocation of components relies on UML abstractions with stereotype *allocate* from MARTE. Several ways of allocating components can be considered:

- allocating operations of component ports on execution resources, in case of client server ports,
- allocating component ports on execution resources,
- allocating component instances on execution resources by MARTE allocation (see Figure 48),
- allocating component instances directly to memory spaces.

Execution resources as described in paragraph 0 are schedulable resources, i.e., operation system threads/tasks.

In all situations, one has to create a UML abstraction and set the UML property that represents the instance of the execution resource as the target of the abstraction.

In the case of an operation, several elements must be set as the sources of the abstraction: the component port, the operation (defined in a UML interface) and the UML property that corresponds to the component instance (or an instance specification, see explications in section 4.4.3). It is mandatory to include the component instance in the sources of the abstraction, as UML ports are associated with component declarations, not with component instances. Forgetting to specify the component instance would then lead to allocate the port of all component instances to the specified execution resource. This situation only applies for client server ports that provide at least one interface.

In the case of a port, one has to set the UML port and the UML property as sources of the abstraction. For a client server port, this will mean that all of the operations provided by the port will be controlled by the execution resource. For a flow port, this will mean that the reception of messages in this port is controlled by the execution resource.

In the case of a whole component allocated to an execution resource, one simply has to set the UML property of the component as the source of the abstraction. This is the most convenient situation from an editor point of view, as it is straightforward: one only needs to draw an abstraction arrow from the UML property of the component instance to the UML property of the execution resource instance. In this situation, all ports of the components will be controlled by the execution resource, i.e., users are not allowed to allocate component instances to more than one execution resource.

The allocation of a component instance directly to a memory partition is to be used if a given component instance is passive (i.e. not driven by any thread), or if the definition of execution resources is performed in a separate process. This allocation type should be also applied for mutual exclusion resources.

4.4.5 Port Connections

Ports of components are connected using UML connectors. This is done in composite structure diagrams.

4.4.6 Connector Deployment

If a dedicated model of the hardware (without managing middelware) is used in the model, UML connectors in the functional software component model (in the following called “logical connector”) have to be deployed on UML connectors in the hardware model (in the following called “physical connector”). This is done in a UML class digram. Since a logical connector usually connects software components which are deployed on different hardware components and which are not directly connected in the hardware model, a logical connector is usually refined by multiple physical connectors. Therefore, a dependency relation is used between first physical connector (starting from the hardware component on which the software component is deployed) and logical connector. For the following physical connectors on the path from the source component to the target component (on which the target software component of the logiocal connector is deployed) in the hardware model, dependency branches are used to refer to same dependency. This modleing methodology is displayed in Figure 49.

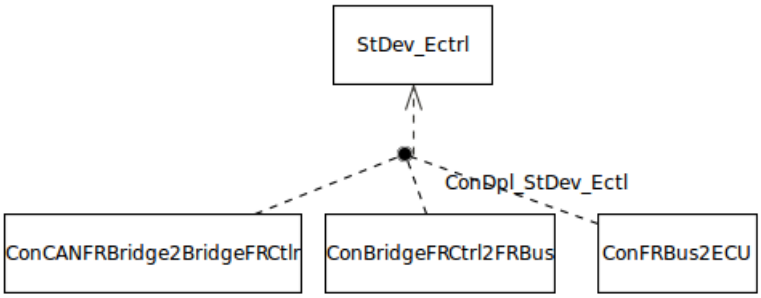


Figure 49: Connector Deployment in Class Diagram

Note that the order of dependency branches reflects the connector path in the hardware model between source and target hardware component.

5. Requirements coverage

5.1 WP3 relevant requirements

Blocker

Number	Description	Status
#50	VERDE shall support (closed) parameterizable (sub)systems (black-boxes) with clear interfaces (e.g. TLM-2.0)	Closed
#89	Handling of multiple cycle time	Closed
#90	Handling of hybrid combinations of models and code	Closed
#128	Component scheduling strategies	Closed
#196	VERDE shall allow component and code reuse (inheritance, multiple instances, parameterization, etc.)	Closed

#410	VERDE shall support partitioning (hardware / software and digital / analog)	Closed

Critical

#45	Component reuse - enable inheritance, multiple instances, parametrization	Closed
#46	Component refinement - real code shall be considered the final step of component refinement	Closed
#75	Clear definition of interfaces	Closed
#82	Debugging on Target	Closed
#91	VERDE shall provide a clear definition of execution semantics	Assigned
#123	Clear execution semantics closed	Closed
#143	VERDE shall offer component / task - how to give an active behaviour to a component	Closed
#249	VERDE shall be based on component model architecture	Closed
#250	VERDE components shall be able to provide/use interfaces	Closed
#251	VERDE components shall be able to emit/consume events	Closed
#252	VERDE components shall be able to contain attributes if necessary	Closed
#253	VERDE interfaces shall be able to provide prototypes of provided/used functions if necessary	Closed
#254	VERDE interfaces shall be able to provide attributes if necessary	Closed
#255	VERDE shall be able to specify visibility of interfaces attributes	Closed
#256	VERDE events shall contain attributes if necessary	Closed
#257	VERDE shall be able to describe implementations of components (i.e. : language, functionality level, simulated or full implementation)	Assigned
#258	VERDE shall be able to design deployment model of component implementations instances	Closed
#259	VERDE shall be able to specify links between components implementations instances	Closed
#265	VERDE shall provide enhanced data types (i.e. : range, precision, size)	Assigned
#267	VERDE shall be able to specify real time aspect of components	Closed
#421	VERDE shall support the AUTOSAR methodology and software	Closed

architecture.

Major

#20	VERDE shall offer No prerequisites in the golden code (Which means application code, not container code from component modeling) design language	Closed
#51	VERDE shall support hierarchical parameters	Closed
#70	Static and dynamic parameters	Assigned
#146	Capability of definition of generic components	Assigned
#214	Integration of / in external instruction set simulators	Reopened
#374	VERDE shall be able to describe implementations of components (i.e. : language, functionality level, simulated or full implementation)	Closed
#425	VERDE shall be able to describe implementations of components (i.e. : language, functionality level, simulated or full implementation)	Closed
#439	Linking and tracing between diagrams, models and execution	Assigned
#463	VERDE shall offer NFP properties validation at early stage (on host) -- WP3 aspects	Assigned

Minor

None.

5.2 Coverage

The VERDE modeling language presented in the deliverable covers 28 of the 36 requirements addressed to this task. In the stage especially structural requirements are covered by the common component model defined in this document:

#50 VERDE shall support (closed) parametrizable (sub)systems (black-boxes) with clear interfaces (e.g. TLM-2.0) and

#196 VERDE shall allow component and code reuse (inheritance, multiple instances, parameterization, etc.)

However, more dynamic requirements are still subject of investigation and will be addressed throughout the remaining part of the project:

#70 Static and dynamic parameters,

#214 Integration of / in external instruction set simulators

Moreover the following requirements will also be addressed after receiving feedback from the different use cases of the work package 1:

#91 VERDE shall provide a clear definition of execution semantics

#257 VERDE shall be able to describe implementations of components (i.e.: language, functionality level, simulated or full implementation)

#265 VERDE shall provide enhanced data types (i.e.: range, precision, size)

#146 Capability of definition of generic components

#439 Linking and tracing between diagrams, models and execution platform

#463 VERDE shall offer NFP properties validation at early stage (on host) -- WP3 aspects.

6. Conclusion

This document has presented a comparison of the different targeted platform technologies in the VERDE project, based on a set of criteria. These criteria are the relevant concepts that are found in each technology. The existing modeling languages, based on UML, have been compared against these criteria in order to define what elements to use in each UML profile.

Finally, the VERDE language for the execution platform has been defined by designing modeling patterns for each comparison criterion. This set of modeling patterns forms the VERDE modeling language.

7. References

[1] AUTOSAR 4.0 Specification on www.autosar.de

[2] AUTOSAR 4.0 Software Component Template http://www.autosar.org/download/R4.0/AUTOSAR_TPS_SoftwareComponentTemplate.pdf

[3] SystemC 2.2 Specification on http://www.systemc.org

[4] Standard SystemC Language Reference Manual http://standard.ieee.org/findstds/standard/1666-2005.html

[5] Object Management Group (OMG), UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems, Version 1.0, OMG document ptc/2009-11-02, 2010

[6] Object Management Group (OMG), Unified Modeling Language: Superstructure, Version 2.3, OMG Document formal/2010-05-05, 2010.

[7] A. Radermacher, A. Cuccuru, S. Gerard and F. Terrier. Generating Execution Infrastructures for Component-oriented Specifications with a Model Driven Toolchain – A case study for MARTE's GCM and real-time annotation Eighth International Conference on Generative Programming and Component Engineering (GPCE'09), pp 127-136, ACM press, 2009.

[8] Software Communication Architecture (SCA), version 2.2.2, specifications can be downloaded from http://sca.jpeojtrs.mil/sca.asp