



D6.1B Derivation of V&V Models from Architecture Models

Author(s): **Adrián Noguero** **ESI**
 Leire Etxeberria **MU**
 Goiuria Sagardui **MU**

Issue Date	July 2010 (m18)
Deliverable Number	D6.1-B
WP Number	WP6
Status	Released

Dissemination level	
X	PU = Public
	PP = Restricted to other programme participants (including the JU)
	RE = Restricted to a group specified by the consortium (including the JU)
	CO = Confidential, only for members of the consortium (including the JU)

Document history			
V	Date	Author	Description
<i>0.1</i>	<i>08-03-2010</i>	<i>ESI</i>	<i>Initial table of contents for the deliverable</i>
<i>0.2</i>	<i>09-07-2010</i>	<i>ESI</i>	<i>Initial contribution to the contents of the deliverable</i>
<i>0.3</i>	<i>14-07-2010</i>	<i>MU</i>	<i>Initial contribution about managing variability</i>
<i>0.4</i>	<i>26-07-2010</i>	<i>ESI</i>	<i>Section 3 completed with allocation specification. Some small formatting changes done.</i>
<i>0.5</i>	<i>30-07-2010</i>	<i>ESI</i>	<i>Completed sections 1-4 and 6.</i>
<i>0.6</i>	<i>30-07-2010</i>	<i>MU</i>	<i>Section about Managing variability completed</i>
<i>1.0</i>	<i>30-07-2010</i>	<i>MU</i>	<i>Review and some small changes done.</i>

Disclaimer

The information in this document is provided as is and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

The document reflects only the author's views and the Community is not liable for any use that may be made of the information contained therein.

Summary

The D6.1-B is a public document delivered in the context of WP6, task 6.1 with regard to the derivation of analyzable models from the architecture design models.

This document contains the exhaustive description of the eDIANA modelling methodology. Additionally, the document includes methodological guidelines to refine architecture models with non-functional information relevant to two different analysis techniques: schedulability and performance. Lastly, the document provides guidelines regarding variability management in the context of early V&V.

Contents

SUMMARY	3
CONTENTS	4
ABBREVIATIONS	5
TABLE OF FIGURES	6
TABLE OF TABLES	8
1. INTRODUCTION	9
2. METHODOLOGICAL FRAMEWORK	9
3. ARCHITECTURE MODELLING RULES	10
3.1 MODELLING EMBEDDED APPLICATIONS.....	11
3.1.1 <i>Application Components' View</i>	11
3.1.1.1 Application Architecture Modelling.....	11
3.1.1.2 Modelling Application Logic.....	14
3.1.1.3 Constraint List for the Application Components View.....	19
3.1.2 <i>System View</i>	21
3.1.2.1 Constraint List for the System View.....	24
3.2 MODELLING EMBEDDED PLATFORMS.....	25
3.2.1 <i>Constraint List for the Platform View</i>	29
3.3 CONFIGURING THE SYSTEM. THE ALLOCATION VIEW.....	29
4. ADAPTING ARCHITECTURE MODELS FOR ANALYSIS	33
4.1 NON-FUNCTIONAL PROPERTIES (NFPs).....	35
4.2 SCHEDULABILITY ANALYSIS.....	37
4.3 PERFORMANCE ANALYSIS.....	38
5. MANAGING VARIABILITY IN THE MODELS	39
5.1 SYSTEM MODELLING WITH VARIABILITY.....	40
5.2 QUALITY ASPECTS ANNOTATION.....	45
5.3 DERIVATION AND TRANSFORMATION TO ANALYSIS MODELS.....	47
5.4 ANALYSIS TAKING INTO ACCOUNT VARIABILITY.....	48
6. CONCLUSIONS	52
ACKNOWLEDGEMENTS	53
REFERENCES	53

Abbreviations

eDIANA	Embedded Systems for Energy Efficient Buildings
MARTE	Modelling and Analysis of Real-Time Embedded systems. UML profile
NFP	Non-Functional Property
LIF	Linking InterFace
LQN	Layered Queuing Network
RTOS	Real-Time Operating System
UML	Unified Modelling Language
MoCC	Model of Computation and Communication
CS	Client-Server
EDF	Earliest Deadline First
LLF	Least Laxity First
FIFO	First-In First-Out
OCL	Object Constraint Language
PEPA	Performance Evaluation Process Algebra
SPL	Software Product Line
VSL	Value Specification Language

Table of Figures

Figure 2-1 – Integration Levels.....	10
Figure 3-1 – Definition of a component and its LIFs.....	12
Figure 3-2 – Example state machine diagram	14
Figure 3-3 – Example time triggered state machine	16
Figure 3-4 – Example state machines triggered by communication events.....	17
Figure 3-5 – Delegating data tokens to UML properties.....	18
Figure 3-6 – Example state machine with a change event driven transitions...	18
Figure 3-7 – Defining an output transition with GCMInvocatingBehavior	19
Figure 3-8 – Example system using UML composite diagrams	22
Figure 3-9 – Using Papyrus to show port icons	22
Figure 3-10 – Platform Model	26
Figure 3-11 – Example platform model	27
Figure 3-12 – Modelling allocation in MARTE	30
Figure 3-13 – An allocation with a NFP constraint in MARTE defined in OCL ...	33
Figure 4-1 – Early V&V in the V-Model context	34
Figure 4-2 – The V-Model implementation in the eDIANA model-driven methodology [6]	34
Figure 4-3 – Non-functional datatypes in MARTE [1].....	36
Figure 4-4 – Specification of the worst case execution time of a behaviour	38
Figure 4-5 – Adding extra NFPs to the definition of a CommunicationMedia ...	39
Figure 5-1– Validation process in a system with variability	40
Figure 5-2– Extract of the extended feature model of the Arcade Product Line	42
Figure 5-3– Game Generalization.....	43

Figure 5-4– Movable sprites decomposition 43
Figure 5-5– The refreshment scenario 47
Figure 5-6– Formula to calculate the refreshment time 50

Table of Tables

Table 5-1: Qualitative impacts.....	42
Table 5-2: An extract of the Feature/class dependencies of Arcade Game Maker Product Line.....	44
Table 5-3: Traceability table for the worst case	49
Table 5-4: Refreshment time of products	51

1. Introduction

This document contains the results of the eDIANA project with regard to the MDE methodology developed for the eDIANA architecture. This document contains the evolved version of the methodology including the changes related to the different versions of the MARTE profile released since 2007. Namely, this version is referred to MARTE version 1.0 (document no. formal/2009-11-02), plus additional changes that will be included in version 1.1. This work refines and extends the work described in [5] with more annotations and constraints to meet the requirements of the eDIANA platform.

The structure of the document is as follows. In section 2 we will discuss the principles of the methodology. These principles have constrained the modelling artefacts used in the methodology, with regard to both UML and MARTE. These modelling artefacts will be related to the specific elements of the embedded systems under design/analysis. Section 3 will cover the concrete modelling aspects of the methodology w.r.t. embedded applications modelling, platforms modelling and allocation, specifying the constraints applied to the standard UML+MARTE models. Section 4 will discuss the extra annotations required to increment the expressivity of the models in order to use them for analysis purposes. Section 5 includes information and techniques about variability management w.r.t. non-functional properties and analysis. Lastly, section 6 draws some conclusions about this work.

2. Methodological Framework

It is not possible to understand the methodology described in this document without taking into account the architectural constraints imposed by the component-oriented approach adopted by the eDIANA consortium. This component-oriented approach defined the software components as the main artefacts for creating applications. In this context applications are created by interconnecting components.

Components are, therefore, independent application pieces with their own data and behaviour. Each component interacts with the rest of the application through their Linking InterFace (LIF). The LIF defines the data and the messages a component may send/receive and also the services a component provides/requires.

Connections between components can be established in both synchronous and asynchronous modes, enabling the definition of applications following different models of computation and communication (MoCC).

As we already said, each component contains its own behaviour and manages its own data. The behaviour of a component is defined in the top level as a state machine which defines the status of the component in each moment. State transitions are driven by external events, i.e. messages or calls received at the component's LIF, or by time.

The proposed methodology separates the application from the execution platform, and, thus, the methodology can be used at different integration levels.

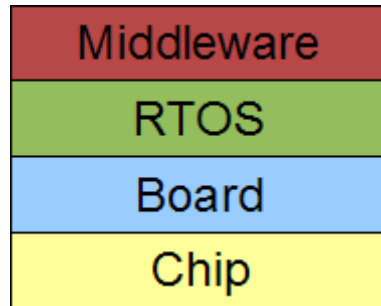


Figure 2-1 – Integration Levels

Depending on the targeted integration level on top of which the application is developed, the execution platform should be defined differently. For example, if the designer is developing an application at the RTOS level, the platform model should include platform elements of the RTOS layer, such as schedulers, threads, etc.

Another rationale for separating application and platform is component reutilization. The defined methodology should provide the designers of a means to easily reuse their application components in different applications, focusing only in the integration of these components through their LIFs. Application components will be after allocated to the execution platform forming the final system model.

3. Architecture Modelling Rules

This section should cover the usage of modelling to develop high-level architectural designs of our systems. In this previous section, a rationale for this methodology should be explained.

3.1 Modelling embedded applications

As we already discussed in this document, application models will be defined by the application components view and the systems view. Concretely, these views will define the following aspects of the models:

- Application Components view. This view includes the definition of each of the components of an application, in terms of LIFs and behaviour. LIF definitions include the MoCC in which the component can be used, as well as the messages/services used by the component. The behaviour describes how each component interacts with its LIFs and its different states.
- System view. This view defines the application system and subsystems as compositions of application components. This view includes the interacting components and their LIFs and specifies how they are connected in the context of the current application.

The selected language to represent these two views is UML (class and state machine diagrams, and composite structures diagrams respectively) plus the MARTE profile for embedded systems modelling (GCM and HLAM subprofiles).

In the following subsections we will explain how to develop correct models according to the methodology proposed in this document.

3.1.1 Application Components' View

3.1.1.1 Application Architecture Modelling

The application components view uses class diagrams and state machine diagrams of UML specification along with the GCM and HLAM profiles of the MARTE profile for embedded system to describe the structure and behaviour of the application components respectively.

The most important elements of the methodology proposed in this document are components. Each component owns a concrete behaviour that defines its functionality in the whole system. In this context, components are modelled as UML active classes stereotyped with the <<RtUnit>> stereotype of the HLAM subprofile. This stereotype provides the class the semantics of an active element with communication capabilities for interacting with other components of the application. The <<RtUnit>> stereotype allows the designer to define the behaviour of the component through the *operationalMode* property. This property should point to the state-machine defining the behaviour of the component, as it will be further discussed later.

In order to communicate with other components in the system and, consequently, orchestrate the complete application, real-time units require

specific features to manage communications. This is done via linking interfaces or LIFs. Depending on the interaction kind that will be established through it three different LIF types are defined:

- Synchronous Client-Server LIFs. These LIFs model synchronous client-server communications. In this case, clients are blocked until a response from the server is received. An explicit connection between client and server is required in this kind of communication.
- Asynchronous Client-Server LIFs. These LIFs model a client-server between two components in which the client is not blocked. Data sent is encapsulated in a signal structure.
- Dataflow LIF. Dataflow LIFs are used to model a dataflow model of computation and communication (MoCC) interaction between a number of real-time units. These LIFs just provide the components of a means to communicate raw data tokens. Senders are not blocked when transmitting data.

In the UML notation proposed in this document, LIFs are defined using UML interfaces which are implemented by real-time units. The semantics of these interfaces is refined by the <<ClientServerSpecification>> and <<FlowSpecification>> stereotypes.

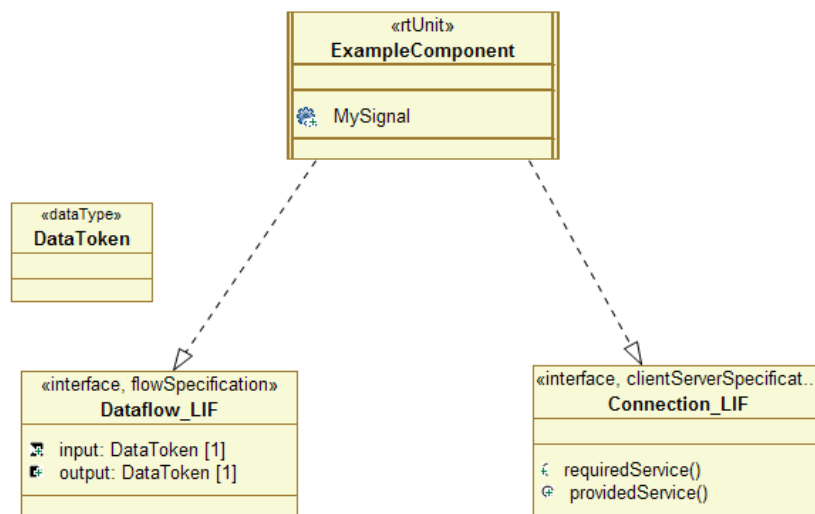


Figure 3-1 – Definition of a component and its LIFs

The <<ClientServerSpecification>> stereotype provides the LIF the semantics of a client-server interaction between two components. The interaction is specified by a set of required and provided services modelled as UML operations in the interface stereotyped with <<ClientServerFeature>> which defines the role of the component regarding each service (provided, required or both). This kind of interaction requires the client component to block until the

server component replies to its request. In order to model asynchronous client-server interactions UML Signal Receptions elements, stereotyped with <<ClientServerFeature>>, are used instead. In the later case a provided signal reception means that the component is capable of accepting (i.e. receiving) the defined signal, whereas a required signal reception means that the component will generate a signal element through the port. For the sake of consistency, a client-server specification may not include two operations with the same name or more than one reception referred to the same signal with the same direction.

On the other hand, the <<FlowSpecification>> stereotype modifies the semantics of UML interfaces by including the idea of dataflow interfaces, through which data tokens are transferred from a component to the other. This kind of connection is data centric, instead of behaviour centric, and the connections are defined by UML properties stereotyped with the <<FlowProperty>> stereotype. Each property defines a data token that a component may produce or consume. These properties have to be typed with a UML Datatype element that defines the data token itself. The data token may also have non-functional properties associated to it in order to define its size, communication time, etc. Non-functional properties will be covered in section 4. The direction (in, out or inout) of each data token is defined in their <<FlowProperty>> stereotype. For the sake of consistency, flow specifications must not include two properties with the same type and direction.

It is required that the real-time unit implements a UML port stereotyped with <<ClientServerPort>> or <<FlowPort>>, for each LIF implemented by the component. These ports will be used in the system view to interconnect the components. Depending on the nature of the ports the following modelling constraints should be applied:

- A client-server port should be typed with the client-server specification defining it, non-atomic and feature based. The *featuresSpec* property defined in <<ClientServerPort>> should point to the client-server specification element described before. The kind of the port must be consistent with the client-server features defined in the specification. A client-server port should always be defined as behavioural, since an event on these ports will trigger an event in the component.
- A flow port should always be typed with the flow specification that defines it and marked as non-atomic. Furthermore, the direction of the port should be consistent with the individual directions of the flow properties defined in the specification of the port. Flow ports can be either set as behavioural or non-behavioural depending on the communication paradigm used in the communications. A behavioural port implies a data event to be triggered to the component (i.e. a "push" communications paradigm), while a non-behavioural port

makes the data token to be stored at the component to be used afterwards (i.e. a "pull" communications paradigm).

3.1.1.2 Modelling Application Logic

Once we have defined the interfaces and the structural features of the component we need to define its behaviour. This is defined by a state machine diagram in the first level, linked to the real-time unit by its classifier behaviour and by the *operationalMode* property defined in the <<RtUnit>> stereotype.

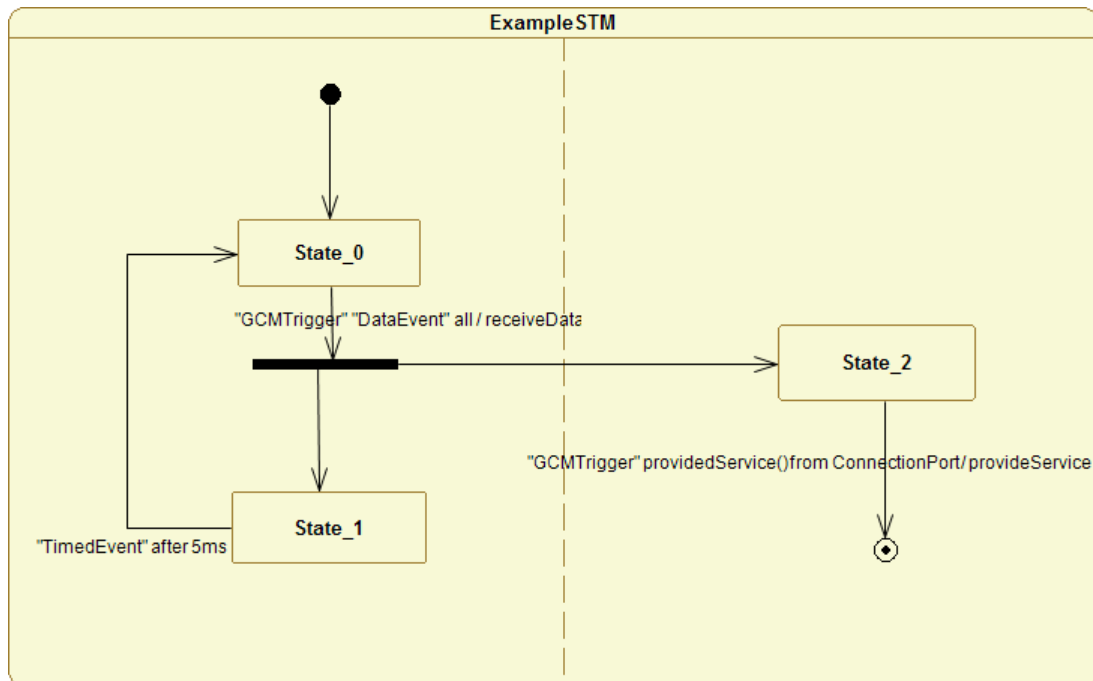


Figure 3-2 – Example state machine diagram

The latter figure depicts a state machine definition example. A component will be defined by a set of states and transitions between them. In the context of the methodology proposed in this document the following assumptions are made:

- While in a state a component will stop its execution until a transition is triggered (i.e. an event occurs).
- A transition may be triggered by:
 - An event received in a port.
 - Time.
 - A change introduced by the user interface (e.g. a button has been pushed).

- Immediately.
- In principle a state may have any number of outgoing transitions; however the following restrictions must be respected:
 - A state must have at least 1 outgoing transition.
 - If a state has an immediate outgoing transition, this transition should be the only outgoing transition of a state.
 - A state may have only one outgoing transition triggered by time; although it may have other outgoing transitions triggered by message events or change events.
 - A state may have many event triggered transitions, as long as these triggers refer to different events.
- A component may also define any number of concurrent/parallel execution threads by using UML regions and UML fork pseudostates as depicted in the figure. Different regions represent different execution threads. A transition between different regions must always have a fork pseudostate as origin.
- It is important to note that pseudostates are not states, and therefore, the component cannot stop its execution on them. As a consequence all the outgoing transitions of a pseudostate (e.g. initial state, fork state...) must always be triggered automatically.

Triggers are related to events. Basically, the methodology proposed in this document includes three different types of events:

- Time Events
- Communication Events, that is:
 - Call Events (for synchronous client-server interactions)
 - Signal Events (for asynchronous client-server interactions)
 - Any Receive Events (for asynchronous dataflow interactions)
- Change Events

TimeEvent elements are used to specify time driven transitions. In this case, the "after" keyword will be used to specify the time elapsed between the instant when the component entered the current state and the instant in which the time transition is triggered. Figure 3-3 depicts a very simple time driven state machine with a unique state which exemplifies the semantics of the time

event. In this example, the "runOperation" behaviour is executed every 5ms after the component is initialized; thus, the execution of "runOperation" should take less than 5ms to complete or the component would become unstable. Note that the "runOperation" behaviour is considered to start executing after the component has completed the transition from the source state to the target state; in other words, a transition is considered to be immediate. TimeEvents related to this kind of triggers should be defined as relative.

TimeEvents should be applied the <<TimedEvent>> stereotype; however in this version of the methodology properties defined in it are not being used.

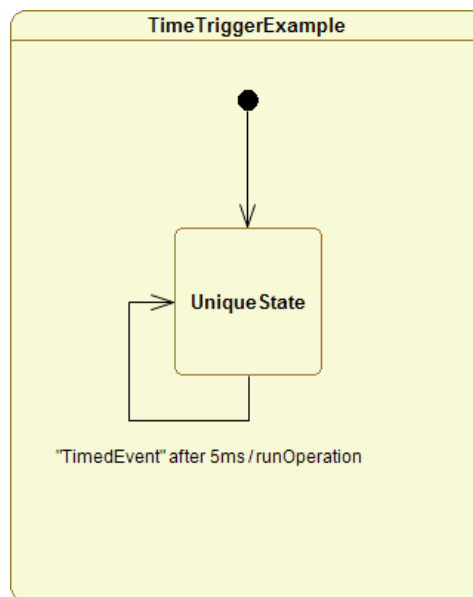


Figure 3-3 – Example time triggered state machine

Communication events, on the other hand, model the reception of messages at the ports of a component. Since different communication types have been defined, also different modelling methods are required (see Figure 3-4).

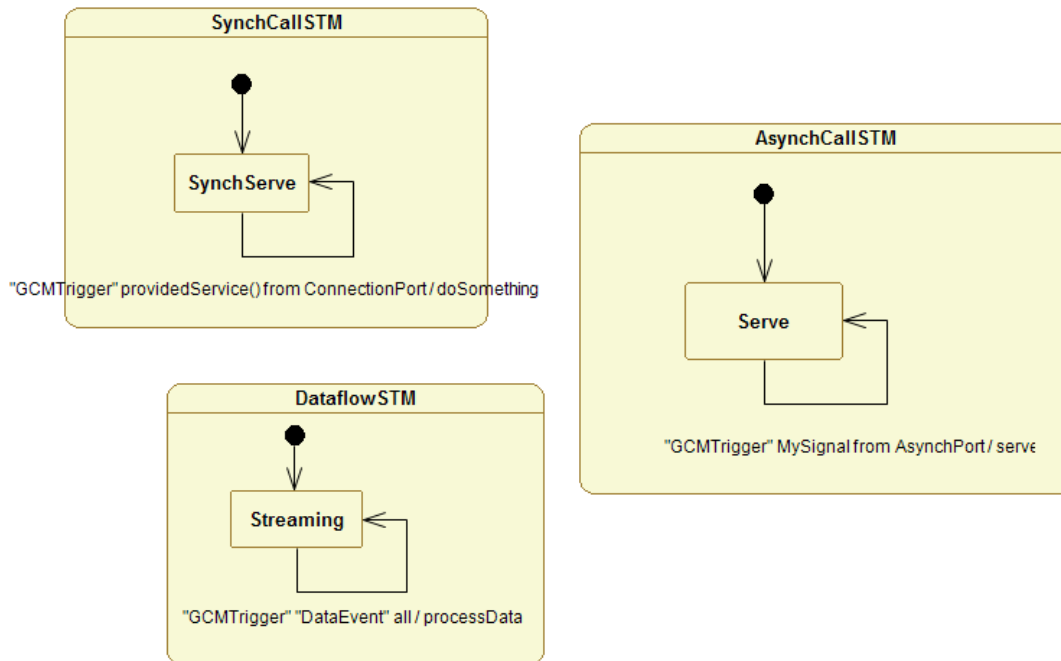


Figure 3-4 – Example state machines triggered by communication events

Triggers related to communication events are dependent of other components in the application. This is modelled by stereotyping the UML Trigger element with `<<GCMTrigger>>`. Triggers related to communication events need to define the port at which the trigger has been originated. The `<<GCMTrigger>>` stereotype enables to further define the exact feature of the specification at which the trigger has been originated. Note that the feature to which the trigger is related must be an input (i.e. "in" or "inout" for flow communications and "provided" or "proreq" for client-server communications).

Client-server communications are modelled using Call Event and Signal Events for synchronous and asynchronous communications respectively. Dataflow communications are modelled using UML AnyReceiveEvents stereotyped with `<<DataEvent>>`. This stereotype enables the definition of the type of data token sent through the port. It is also possible that some components don't need to handle the received data token exactly when it is originated (e.g. a "pull" communication); in such cases, flow ports are directly connected to UML properties in the component that share the same type as the incoming data token. This can be easily specified using a composite structures diagram, as depicted in Figure 3-5:

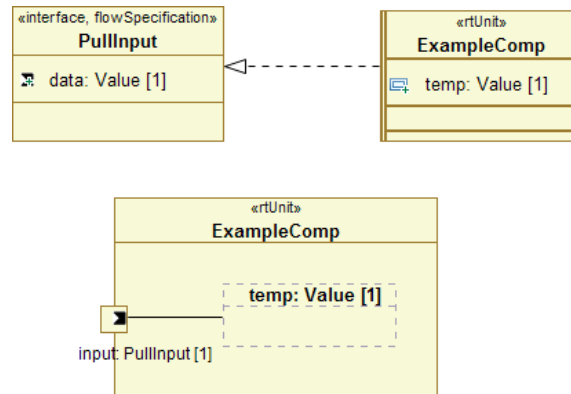


Figure 3-5 – Delegating data tokens to UML properties

Lastly, change events are used to model user driven transitions (e.g. a value is introduced by the user, a button is pushed, etc.). Change events are triggered whenever the boolean expression specifying it changes from *false* to *true*. For the sake of applicability, this methodology proposes the use of a method returning a boolean. This method will be called for checking the change, and in the code generation phase, it will contain the actual logic in the concrete programming language. The keyword “*when*” is used to specify a change event in the diagram. Figure 3-6 shows an example state machine where a change event are generated:

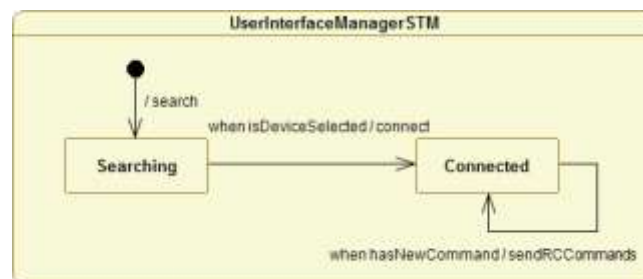


Figure 3-6 – Example state machine with a change event driven transitions

Transitions may also define a behaviour that will be executed as soon as the component changes its state. Currently, the specification of this behaviour can be defined using either:

- UML OpaqueBehavior. This opaque element allows the designer to specify a behaviour using any language.
- UML Activities. Activities allow the designer to specify a behaviour using UML Actions. How these activities should be defined is out of the scope of this document, so it will not be covered here.

- UML StateMachines. State machines allow the designers to establish a hierarchy of state machines. This allows the definition of substates, inside a state.

In a transition a component may also send a message to or require a service from another component. This will be specified by applying the <<GCMInvocatingBehavior>> stereotype to the behavioural element defining the behaviour of the component. This stereotype allows designers to define which interactions are triggered inside a behavioural element, without inspecting its internals.

For example, the following figure shows how to define output interactions in a UML OpaqueBehavior using Papyrus. It is important to note that, if a <<FlowProperty>> or a <<ClientServerFeature>> is referenced by this stereotype, the <<FlowPort>> or <<ClientServerPort>> containing it must also be referenced.

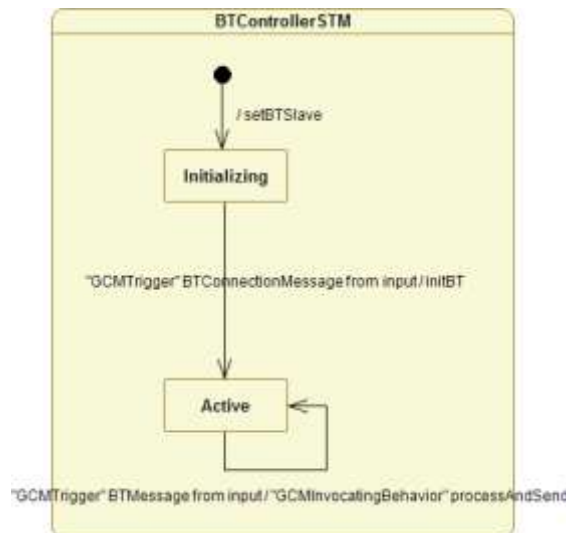


Figure 3-7 – Defining an output transition with GCMInvocatingBehavior

This process should be followed for each component in the application. Components defined this way can be used to compose a repository of application components enabling reuse at design time.

3.1.1.3 Constraint List for the Application Components View

This section will include a list of all the constraints applicable to models compliant with the methodology presented in this document regarding the application components view. The constraints are formalised using OCL.

Name
Active classes must be stereotyped <<RtUnit>> and viceversa. This will be considered a component.

The classifier behavior of a component must be set to a state machine.
The operationalMode of a <<RtUnit>> must point to the same state machine as the classifier behavior.
LIF specifications are defined as UML interfaces stereotyped with either <<FlowSpecification>> or <<ClientServerSpecification>>
Flow LIFs may not define any signal receptions or operations.
Properties defined in Flow LIFs must be stereotyped with <<FlowProperty>>
Properties defined in Flow LIFs must be typed with Primitive Types, Data Types or Enumerations.
Flow LIFs must not include two properties with the same type and the same direction.
Flow Ports must have a direction coherent with its defined Properties
CS LIFs may not define any properties.
CS LIFs may only define either Operations or Receptions
Receptions and Operations defined in CS LIFs must be stereotyped with <<ClientServerFeature>>
Receptions defined in CS LIFs must be referred to Signals
CS LIFs must not include two operations with the same name.
CS LIFs must not include two receptions with the same signal and the same kind
ClientServerPorts must be of a kind coherent with its defined Features
For each LIF implemented by a component, that component must own a port.
Each port typed with a <<FlowSpecification>> must be stereotyped with <<FlowPort>> and viceversa.
Each port typed with a <<ClientServerSpecification>> must be stereotyped with <<ClientServerPort>> and viceversa.
Flow ports must be non-atomic
CS ports should be behavioral.
CS ports should be feature based
A State must have at least an outgoing transition, unless it is a FinalState.
States may have a single automatically triggered transition.
States may only have a single transition triggered by time.
Events triggering the outgoing transitions of a state must be different.
Outgoing transitions in States must always end in the same region.
Initial pseudostates must have a unique automatic transition
Fork Pseudostates must have a single outgoing Transition targeting a State in their same Region
Outgoing transitions in a Fork Pseudostate must target States in different Regions.
Outgoing Transitions in Fork Pseudostates must be automatically triggered
TimeEvents related to Triggers must begin with the "after" keyword
TimeEvents related to Triggers should be declared relative
TimeEvents should be stereotyped <<TimedEvent>>
Triggers related to communication events must be stereotyped <<GCMTrigger>>
GCMTriggers must define the port at which the interaction is happening
GCMTriggers must define the feature of the port at which the interaction is happening
Features defined in a GCMTrigger must be owned by the Specification typing the Port at which the interaction is happening
If the event triggering a communications transition is a CallEvent, the GCMTrigger must point an Operation feature and viceversa
If the event triggering a communications transition is a SignalEvent, the GCMTrigger must point a Reception feature and viceversa
If the event triggering a communications transition is a AnyReceiveEvent, the GCMTrigger must point a Property feature and viceversa
AnyReceiveEvents related to triggers must be stereotyped <<DataEvent>>.
The classifier property of a DataEvent should refer to the type of the flow property referred in the GCMTrigger

Triggers related to Flow LIFs must refer to in or inout properties
Triggers related to CS LIFs must refer to provided or proreq features
ChangeEvents related to Triggers must refer to Operations returning Booleans in the containing component
For each Feature referred by a GCMInvocatingBehavior effect its related Port must be also referred*
Ports referred by a GCMInvocatingBehavior must be owned by the same component containing it*
Flow features referred by a GCMInvocatingBehavior must be declared as "out" or "inout"*
CS features referred by a GCMInvocatingBehavior must be declared as "required" or "proreq"*
Ports referred by a GCMInvocatingBehavior must be FlowPorts or CSPort*
Non-behavioral FlowPorts must be connected to a property that shares type with an "in" or "inout" flow property in it or a behavioral port of the same type through a Delegation Connector

3.1.2 System View

The objective of the system view is to define subsystems and applications from repository components, whose definition has already been covered in the previous section.

As we already said, the methodology proposed in this document envisages applications as a set of components that interact with each other to achieve a goal. The system view defines the interconnections between instances of these already defined components. The components, running the behaviour defined in them, will start sending and receiving messages and orchestrating the application as a whole.

This view enables the designers to define subsystems. These subsystems are also composed of different components and subsystems, and it also enables the designers to define a hierarchy of components.

Regarding the methodological aspects, this document proposes the use of UML composite diagrams to define systems and subsystems. Figure 3-8 depicts an example subsystem with two instances of the same component.

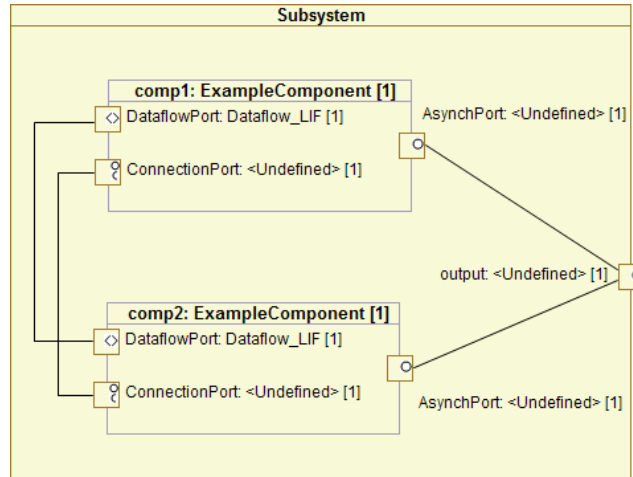


Figure 3-8 – Example system using UML composite diagrams

Both systems and subsystems are modelled as UML classes stereotyped with <<GaWorkloadBehavior>>. For each instance of a component in a system a UML property element is created. The type of these properties should be the <<RtUnit>> classes defining the components that we already created in the previous step or subsystem classes, such as the one depicted in Figure 3-8. Each property in a composite structures diagram should show all the ports owned by the typing class, as shown in the figure. Subsystems will also include a set of non-behavioural external ports, defined as we already discussed in the previous section. These ports will define the LIF of the subsystem. Thus, a subsystem can be seen as a complex component, internally composed of other components.

TIP: In Papyrus you can make the tool show the icons of the ports using the 'Appearance' tab in the Properties view, as shown in Figure 3-9.

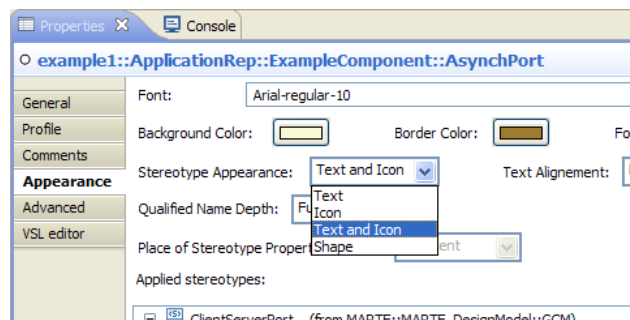


Figure 3-9 – Using Papyrus to show port icons

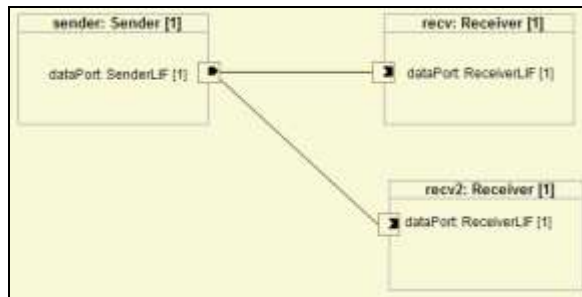
After instantiating all the components required for the (sub)system, we will start creating UML connectors to link the ports of the components. A link between two ports establishes a communications path between them. How this

communications will be implemented is not modelled here, since it is considered a service provided by the underlying software/hardware platform.

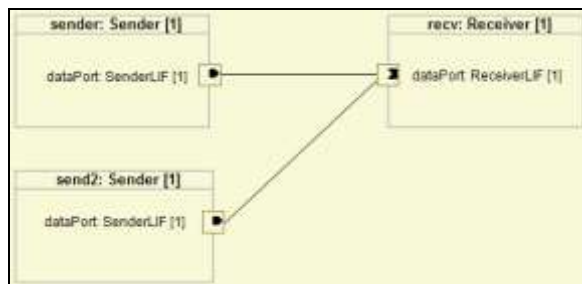
Connector creating communication paths between components need to be assembly type, since the message sent by the source node is handled by the target component. On the other hand, connectors between component ports and external ports (in the case of subsystems) must always be defined as delegation type. A delegation connection means that a message arriving to the port is deferred to all the ports linked by a delegation connector.

A port may be connected to several other ports. The semantics of these multiple connections are as follows:

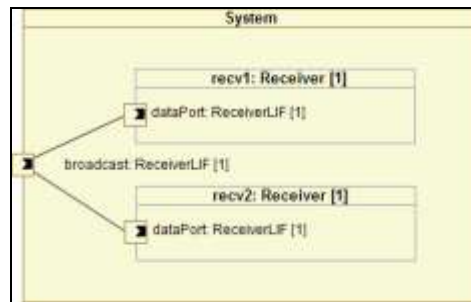
- An output port connected to several inputs means that a broadcast message is being sent. This configuration is not valid in case of synchronous client-server interactions.



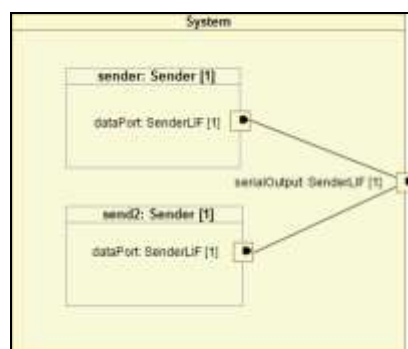
- An input port connected to several outputs means that a component will interact with other components without knowing which component sent the message.



- An external input port deferring to several input ports inside the subsystem means that the message is deferred to all the ports simultaneously. This configuration is not valid for synchronous client-server communications.



- An external output port fed by several output ports means that the output port will serialize the output messages generated internally.



It is important to note that in order to make the model consistent there are some constraints that must be met:

- Ports at the end points of a connector must be of the same type (i.e. dataflow, synchronous client-server or asynchronous client-server).
- The communicating features in both ends must be coherent (e.g. at least one property in a dataflow communication must be of the same type in both ends and with different directions).
- There shouldn't be two connectors between the same two ports.
- All the ports in a system or subsystem must be connected.

3.1.2.1 Constraint List for the System View

Name
Systems and subsystems must be <<GaWorkloadBehavior>> stereotyped non-active classes
(Sub)systems should not have any Operation
The types of properties defined in a (sub)system must be RtUnits or other subsystem
No Connectors can be defined between the same two elements*
Connectors between internal ports in a system should be Assembly Connectors
Connectors between external ports and internal features should be Delegation Connectors
Connectors between Ports must link either FlowPorts or ClientServerPorts
Connected internal input FlowPorts must connected to output FlowPorts and viceversa
Connected internal input ClientServerPorts must connected to output ClientServerPorts and

viceversa
Connected FlowPorts must be conjugated
Connected Synchronous CS Ports must be conjugated
Connected Asynchronous CS Port must be conjugated
One-to-Many connections are not allowed for internal synchronous client-server interactions
One-to-Many connections are not allowed for external synchronous client-server interactions
All ports in a system or subsystem model must connected to another by a Connector
The types of properties defined in a (sub)system must be RtUnits or other subsystem

3.2 Modelling embedded platforms

Regarding verification and validation aspects, embedded system platforms, both hardware and software, become as important as the concrete applications running on top of them. The methodology proposed in this document provides some guidelines regarding high level platform modelling for embedded systems. This methodology envisages a single platform view to describe embedded software and hardware platforms.

In order to define this methodology, a high level model for embedded platforms was created. This model defines the relationships between the different elements of an embedded system platform, hardware and software, as well as a set of important parameters required for performing analyses on them. Next figure depicts this model.

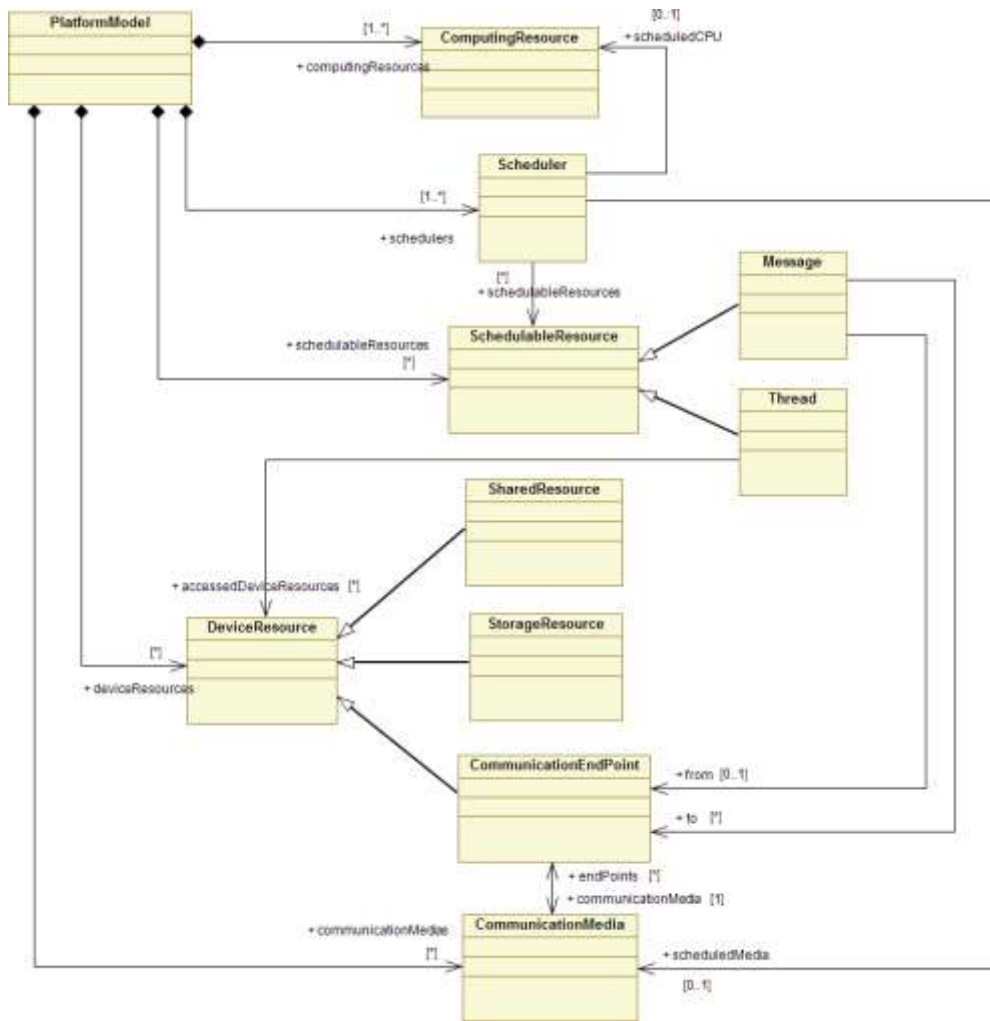


Figure 3-10 – Platform Model

The latter figure specifies the syntax of the proposed platform models; however, in the methodology described in this document we do not aim at creating a new metamodel, but to define some guidelines to use the MARTE profile. Nevertheless, the latter figure serves as a starting point, and will help the reader to better understand the rules and constraints that will be defined later in this document.

The platform view will be constructed using the UML composite structures diagram along with the MARTE GRM profile. The GRM profile enables the designer to include in the platform model many different hardware and software elements that will support the deployment of the embedded applications modelled in the previous step. An embedded platform is always modelled as a UML class, stereotyped with <<GaResourcesPlatform>>, that contains a set of properties and connectors that represent the resources contained in it and the interactions between them. UML connectors are used to

link some of these resources to explicitly define which resources may access others.

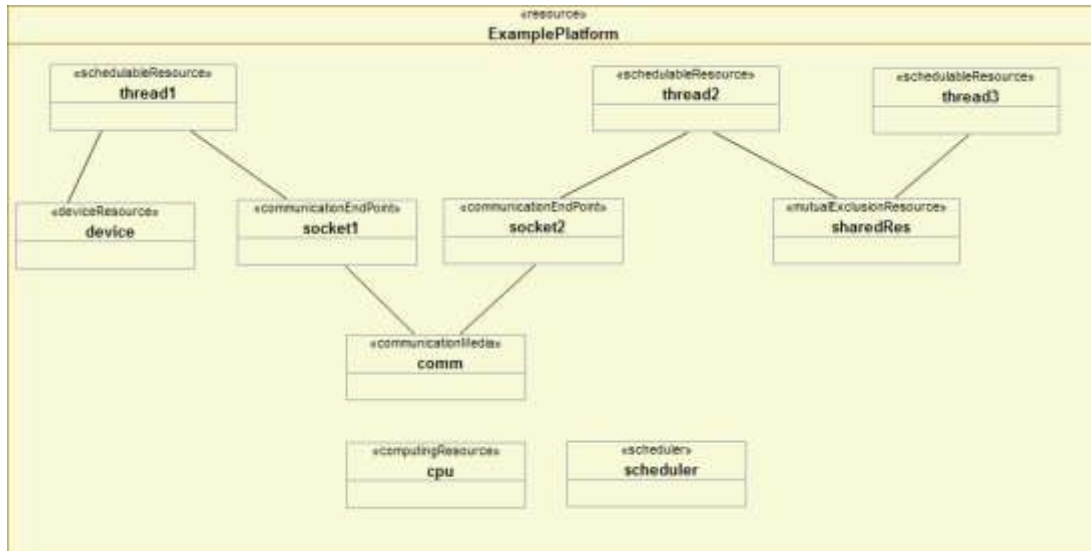


Figure 3-11 – Example platform model

As shown in Figure 3-11, the GRM profile enables the definition of different resource types:

- **Computing Resource.** A computing resource models any CPU or core in a processor capable of executing the application code. A platform model must include at least a computing resource to be considered valid. Computing resources must be stereotyped with <<ComputingResource>> and they must always define their speed factor value as a ratio between their speed and the speed of the reference resource of the platform (e.g. if a embedded system has two processors, one running at 1GHz, and another one running at 2GHz, the speed factor of the second one would be 2, taking the first one as reference). Additionally, a computing resource should be governed by a Scheduler.
- **Scheduler.** A scheduler is a very important platform element in real-time applications, since it schedules the use of computing resources and communication media. A scheduler must always define a number of schedulable resources that it will manage, the resource it will rule (i.e. a computing resource or a communication media) and the scheduling policy being used. Additionally a scheduler may define a number of shared resources being governed by it and a host (i.e. the computing resource at which the scheduler is executed).

- **Schedulable Resource.** A schedulable resource represents either an execution thread or message type. A schedulable resource must always refer to the scheduler governing it and it must define its scheduling parameters. The scheduling parameters must be compatible with the scheduling policy selected by the scheduler resource. Depending on the processing resource managed by the host scheduler, would it be a computing resource or a communication media, a schedulable resource will be seen as a thread or a message type respectively. Schedulable resources representing threads can be connected to device resources accessed by them, namely any library or device requiring a driver library, such as an external storage resource, a communication driver or a mutex. Schedulable resources representing messages, on the other hand, cannot be connected to any other resource.
- **Device Resource.** A device resource models any kind of resource external to the embedded system that is accessed by it, like an actuator, a sensor, etc. Three elements are subtypes of this Class: Mutual Exclusion Resource, Storage Resource and Communication End Points.
- **Mutual Exclusion Resource.** A mutual exclusion resource models a protected shared resource used by several threads to synchronize their access to shared data. It must define the scheduling policy managing it. These elements can be connected to [0..*] threads that may access it.
- **Storage Resource.** A storage resource models an external storage media for storing data, such as an SD card or a data base. A Storage Resource must define in its properties the size of the smallest storable element in bytes (using the *elementSize* property), plus the number of elements present in the storage unit (using the *resMult* property).
- **Communication End Point.** A communication end point models a driver that provides access to a communication media (i.e. Bluetooth, TCP/IP, USB, etc.). Communication end point must be linked to a single communication media, and they may define the maximum packet size that can be transmitted.
- **Communication Media.** A communication media models a network or a communications infrastructure capable of transmitting messages between threads. Communication media must be linked to their communication end points. Regarding specific properties, communication media must define the maximum packet size in bytes, the transmission mode and its capacity. Additionally, a communication media may also define a scheduler.

3.2.1 Constraint List for the Platform View

<i>Name</i>
Platforms should be non-active
Platforms must not define any Operations
Platforms must not define any Ports
Platforms must define at least a Computing Resource
Computing Resources must define their SpeedFactor
Computing Resources must define their mainScheduler
Schedulers must define if they are preemptable
Schedulers must define their scheduling policy correctly
A Scheduler must govern either a Computing Resource or a Communication Media
Schedulable resources must have a host
Schedulable Resources must have the correct scheduling parameters set*
Storage Resources must define the element size and the number of elements that it can store
Mutual Exclusion Resources must define the protocol followed for preemption management
A Mutual Exclusion Resource with a priority ceiling protocol must specify the priority ceiling
Communication End Points may define a maximum packet size
Communication Media must define their packet size, transmission mode and capacity.
Device Resources must be linked with a Connector to at least one Schedulable Resource
Device Resources must not be linked with a Connector to anything but Schedulable Resources
Storage Resources must be linked with a Connector to at least one Schedulable Resource
Storage Resources must not be linked with a Connector to anything but Schedulable Resources
Communication End Points must be linked with a Connector to at least one Schedulable Resource
Communication End Points must not be linked with a Connector to anything but Schedulable Resources or a Communication Media
Mutual Exclusion Resources must be linked with a Connector to at least one Schedulable Resource
Mutual Exclusion Resources must not be linked with a Connector to anything but Schedulable Resources
Communication End Points must be linked to a single Communication Media
Communication Media must only be linked to Communication End Points
Computing Resources must not be linked
Schedulers must not be linked
Connectors must not be duplicated

3.3 Configuring the system. The allocation view.

Applications can be deployed on hardware/software platforms in many different ways, having each of them different outcomes and results. Thus, how application components are mapped on top of platform elements becomes very important for the architectural analysis. In some way, this allocation of application components on platforms can be understood as a configuration of the whole embedded system.

Since the same application model may be allocated differently on a platform model, this methodology proposes that different allocation alternatives are stored in different models. This will enable the analysis of the different

allocation alternatives separately to define which possible configuration leads the system to a higher level of performance, better schedulability slack, lower power consumption, etc.

The MARTE profile encourages the use of the Alloc subprofile to model this allocation of application components on top of platforms. Allocation establishes a relation between N application component and 1 platform elements, for example, N connectors between components and a communication media, a region in the state machine of a component and a thread, etc. These N to 1 relations are modelled in MARTE through many individual 1 to 1 allocation dependencies. It is important to note that a single application element may not be allocated to two different platform elements.

The selected representation is a UML abstraction dependency enriched with the MARTE <<Allocate>> stereotype as shown in the figure:

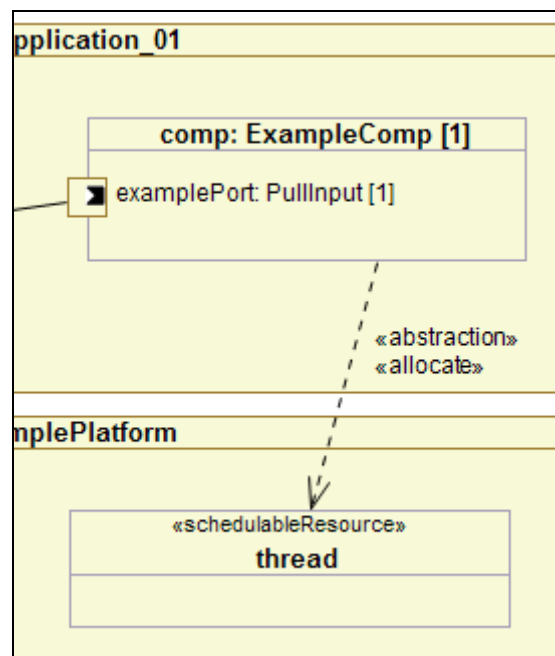


Figure 3-12 – Modelling allocation in MARTE

In a generic scenario, abstractions can link any two UML elements in a model to define that there is a refinement semantic between them. The MARTE profile enriches this relation to define the allocation dependency between application elements and platform elements; however, in this methodology we will constrain this generic scenario for the sake of applicability. Thus, an allocation dependency may not be defined between any two elements in the model; instead, they may only be defined between some application-platform element pairs. Additionally, for the allocation to be complete, all the required application elements must be allocated to their respective platform pairs, else analysing the

models would be impossible. The following table defines all the legal application-platform allocation pairs:

<i>Application element</i>	<i>Platform elements</i>	<i>Allocation description</i>
Components	Schedulable resource	A component may be allocated to a schedulable resource if and only if its internal state machine does not contain any parallel regions. This allocation makes the execution of the behaviour of the component to be done inside the given scheduling thread. Multi-region components must allocate each of their regions separately.
Regions (in state machines)	Schedulable resource	In components with parallel state machines, regions can be allocated to independent schedulable threads. If this should be done, then all the behaviours triggered in that region would be executed in the given scheduling thread.
Ports	Communication End Points	Ports must be linked to communication end points; however this may happen if and only if the connector to which the port is linked is allocated to a communication media. If this should be the case the allocation would define communication API linked to the port utilization.
Connectors	Communication Media or Mutual Exclusion Resources	Connectors model the communications between components. Depending on the nature of this communications, allocation may be done differently. If the two components are linked through a communication media, such as a network (or using the internal protocol stack)

		connectors can be allocated on communication media elements. In this case a communication through the connector implies messages to be sent through the network. Components inside the same machine (e.g. allocated in different threads) may also communicate internally using shared memory resources. In this case a connector would be allocated to a shared resource. In this case, a communication would imply the locking and unlocking of the shared resource.
Behaviours (in transitions)	Device or Storage resources	Some behaviour elements in a state machine may be allocated to external libraries used by the application (e.g. a driver of an external device, a library with a specific algorithm, an external SD card, etc.). In these cases a behaviour can be allocated to a device resource or a storage resource. Such an allocation will imply that the allocated behaviour will use the driver of selected resource during its whole execution.

Additionally from the semantics described in the latter table, an allocation dependency may introduce a non-functional constraint on a non-functional property defined in either the application or the platform elements affected by it. Non-functional constraints are modelled in MARTE using the regular UML Constraint elements stereotyped with <<NfpConstraint>>. They are intended to modify the values of the non-functional properties of the system that may change during the allocation process; for example, the worst case execution time of a behaviour will definitely change depending on the computing resource running it. MARTE allows an NFP constraint to be defined required, offered or contract depending on the type. This information is also very important for the

analysis phase. In the latter example of the worst case execution time, the associated NFP constraint should be defined as a contract, as it is derived from the allocation. The specification of an NFP constraint can be done in any language; however, it is highly desirable that the selected language can be formally checked (e.g. OCL).

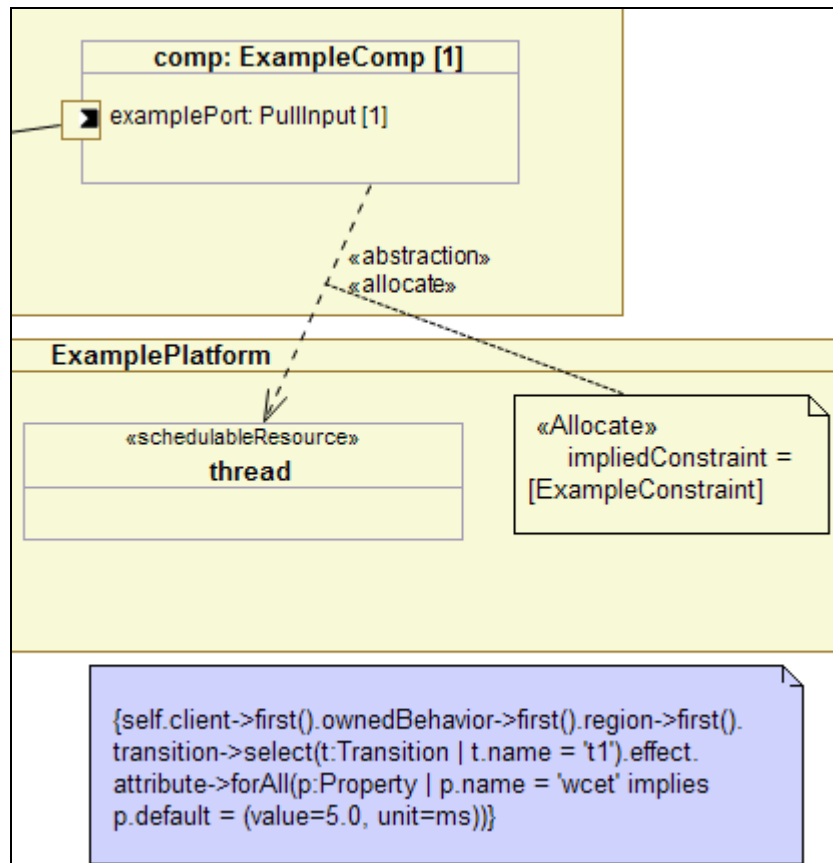


Figure 3-13 – An allocation with a NFP constraint in MARTE defined in OCL

4. Adapting Architecture Models for Analysis

One of the main goals of the methodology described in this document is to enable the early verification and validation of the design models that will make it through to the following development steps in the global eDIANA development process framework. Additionally, the models should be ideally valid for different verification and validation techniques.

The early V&V phase will take as input the provided design models, and will analyze them to obtain a first approximation of their performance and/or timing figures. In a generic V-model (shown in Figure 4-1) development framework, early V&V would take place right after design has been completed and the results for the analysis would provide feedback for both designers and developers in the further steps of the development process.

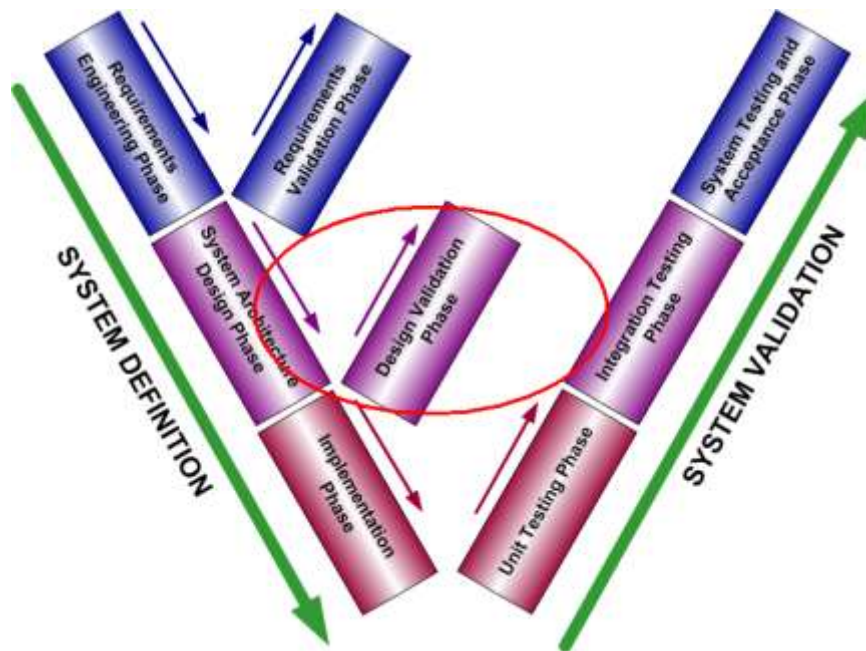


Figure 4-1 – Early V&V in the V-Model context

The latter early V&V phase has been implemented in the eDIANA model driven methodology, described in D2.1-A [6], as described in the following figure. This document, as the model-driven methodology of eDIANA, separates the design process into three steps: application design, platform definition and system allocation/configuration. As shown in Figure 4-2, after the system allocation phase, the early V&V analyses take place.

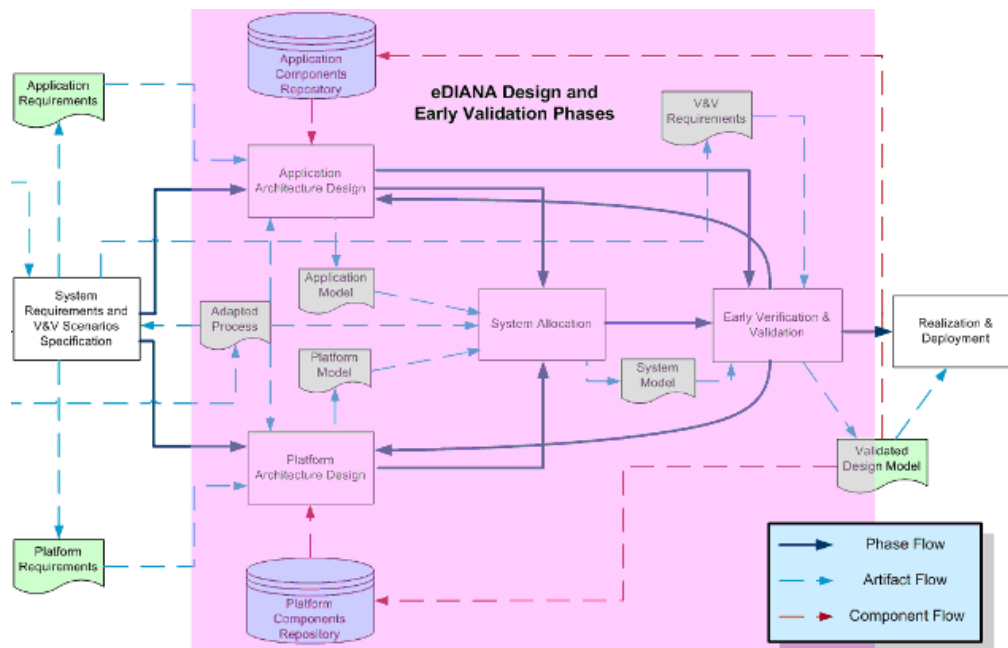


Figure 4-2 – The V-Model implementation in the eDIANA model-driven methodology [6]

However, in order to successfully perform V&V analyses on the design models it is mandatory to extend the design models with non-functional annotations; for example, the execution times of the behaviours, failure probabilities, etc. In this section we will describe how the design models generated following the guidelines in section 3 can be extended in order to enable early schedulability analysis and early performance analysis. This methodology can be extended following similar guidelines for other analysis types; however, this is out of the scope of this document, and it is defined as future work.

4.1 Non-Functional Properties (NFPs)

The main artifacts provided by MARTE to include analysis information in the models are non-functional properties (NFPs). NFPs are modelled as regular UML properties annotated with the <<Nfp>> stereotype. Additionally, the MARTE standard library includes many complex datatypes to model non-functional aspects, such as length, data sizes, time or even prices (see Figure 4-3).

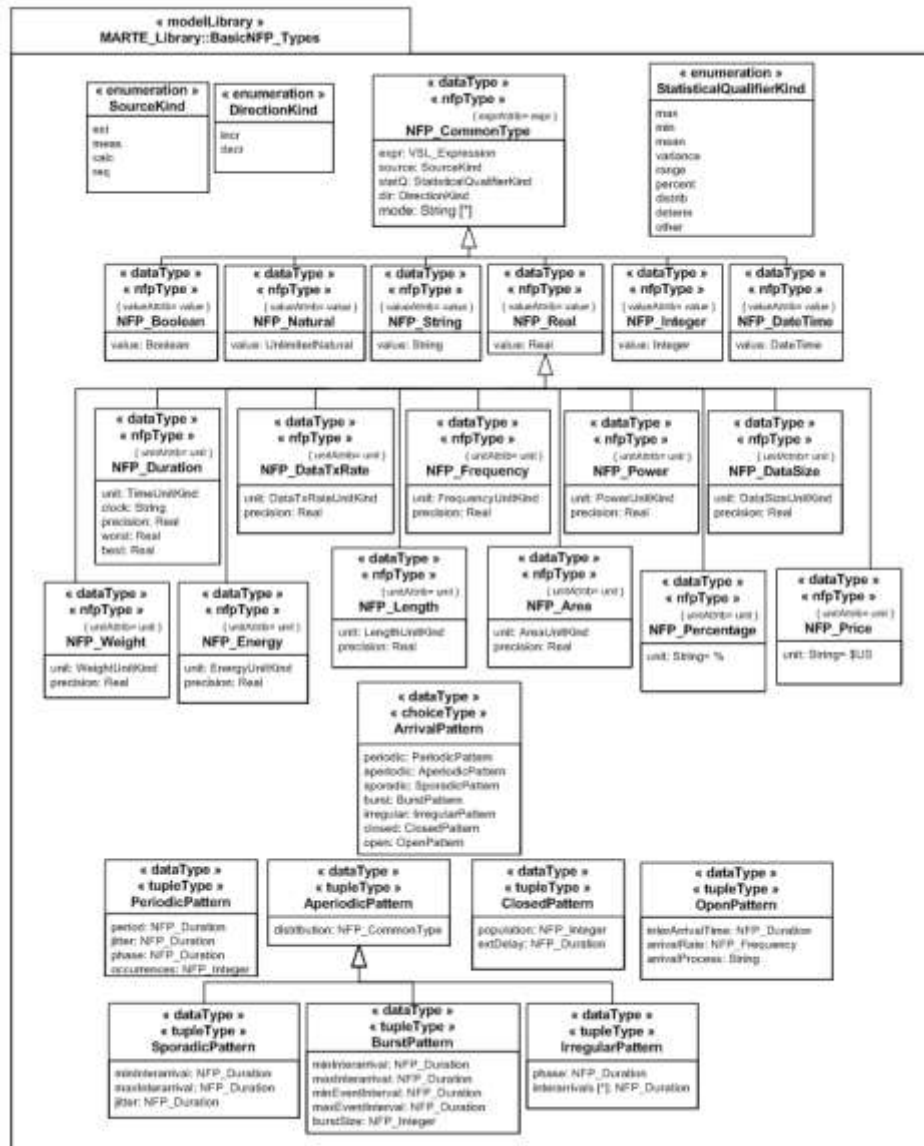


Figure 4-3 – Non-functional datatypes in MARTE [1]

As NFPs are subtypes of classis UML properties, every structured classifier in the model may be enriched with NFPs. In particular, for the methodology described in this document, the following structured classifier list is envisaged:

- Systems and subsystems. Application systems and subsystems can be refined with non-functional information; for example, the failure probability of a subsystem or the total development cost.
- Components. Components, as active classes, may also be refined with NFPs. Examples of possible NFPs for components are failure rates, memory consumption, throughput, etc.

- Behaviours. All UML behaviours (i.e. activities, state machines, opaque behaviours...) are subclasses of structured classifier and, therefore, they can be enriched with non-functional information. Examples of NFPs applicable to behaviours are execution times (worst, best, average).
- Signals, Primitive Types and Datatypes. These are also candidate elements for NFPs specification. The clearest example of an NFP applicable to these elements is their data size.
- Platform elements. In some cases we may want to refine the properties of the standard platform elements provided by MARTE with extra NFPs. In these cases, we can create platform component classes that will type the properties in the platform models.

Other NFPs and application elements are also possible; however, they won't be covered in this document as they are out of the scope of eDIANA.

4.2 Schedulability Analysis

One of the main analysis targets of eDIANA is the early evaluation of the schedulability of the systems. Schedulability refers to the way processes and threads are assigned to run on the available CPUs ensuring that the timing requirements of each of the executing tasks will be met.

Operating Systems include a special process called scheduler who is in charge of organizing the execution of the different processes and threads in the system according to a particular scheduling policy. Thus, it is the selected scheduling policy one of the most critical aspects of schedulability. We already discussed in this document that MARTE includes specific annotations to represent schedulers and their scheduling policies. More specifically, MARTE natively supports: Fixed Priority, EDF, LLF, TimeTable, FIFO and Round Robin scheduling. Additionally, MARTE supports the definition of user-specific scheduling policies through the use of special constructs.

It is important to note that in order to use these scheduling policies, the threads and processes of the system must be described with a set of parameters as required by the scheduling policy of the system. MARTE also supports the modelling of the different threads and processes available in the system and their scheduling parameters.

Then, NFPs are the only information missing in the models for schedulability, namely timing information. In the previous section we already discussed that execution times were one of the most important NFPs to be included in behaviours. Also it is a requirement to include the maximum deadline time before which the result of a task is valid. Indeed, through the inclusion these times in the behaviours of the models, these become ready to be analyzed with

different scheduling algorithms. Figure 4-4 depicts a behaviour including non-functional timing information:

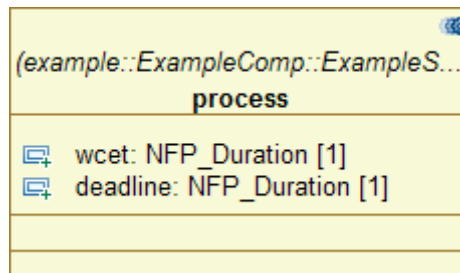


Figure 4-4 – Specification of the worst case execution time of a behaviour

Other important aspects of schedulability are shared resources. The access to shared resources may lead the system to timing failure due to inter-task blocking. How these resources behave depends on the specific policy implemented by the resource. MARTE supports the definition of many different shared resource policies.

4.3 Performance Analysis

Performance evaluation means a process of estimating through using performance models in quantitative terms what would the performance properties of the system being designed when implemented, whereas after implementation it is more of measuring the values of properties (the latter is not addressed in the sequel).

Performance evaluation in the context of real-time embedded systems development tries to provide insight to three main issues:

- Responsiveness: Is the system capable of producing responses to user (external) service requests in defined response times or at defined throughput?
- Resource adequacy/utilization: Does the system have resources and their capacity enough for the currently planned applications? How efficiently are the resources utilized?
- Scalability: Does the system facilitate extensions/reductions and scale up/down resources and their capacity enough to accommodate future applications/ changes in applications?

Performance evaluation methods can be classified to three main classes: analytical methods, simulation methods and monitoring methods [7].

Analytical performance modelling is typical in early phases of design and methods are based on mathematical models of the workload and the system

architecture. Markov chains, queuing models and Petri-nets are typical examples of analytical modelling techniques. Analyses are normally based on solving the equations, but also simulation is used as a supporting tool.

In the performance simulation, the execution of a workload with a model of execution platform is simulated. The workload modelling can be based on several alternatives, e.g. executable programs (real application or benchmark programs), execution traces of programs and stochastic models. The execution platform modelling can be based on e.g. abstract resource capacity models or virtual platform models where instruction-set simulators are used to simulate programmable.

Monitoring / measurement based approaches need working prototypes of hardware. The prototype is instrumented to gather performance information during the execution of the software.

Thus, in order to apply any of these techniques to the design models it is a prerequisite to include performance specific NFP information. Standard MARTE stereotypes already include many non-functional fields for performance analysis (e.g. the CommunicationMedia stereotype, used for modelling networks and buses, allows specifying the transmission time of the packets and the blocking time). Additionally, other NFPs can be used following the methods already discussed (see Figure 4-5).

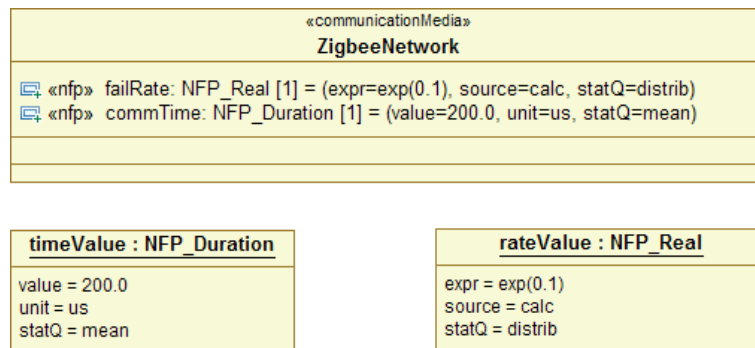


Figure 4-5 – Adding extra NFPs to the definition of a CommunicationMedia

5. Managing Variability in the Models

When a software product line or a system with variability is being modelled and analyzed (such in the case of eDIANA), variability must be managed to be able to model and perform the analysis. In the next figure, it is shown the process for analyzing a system with variability.

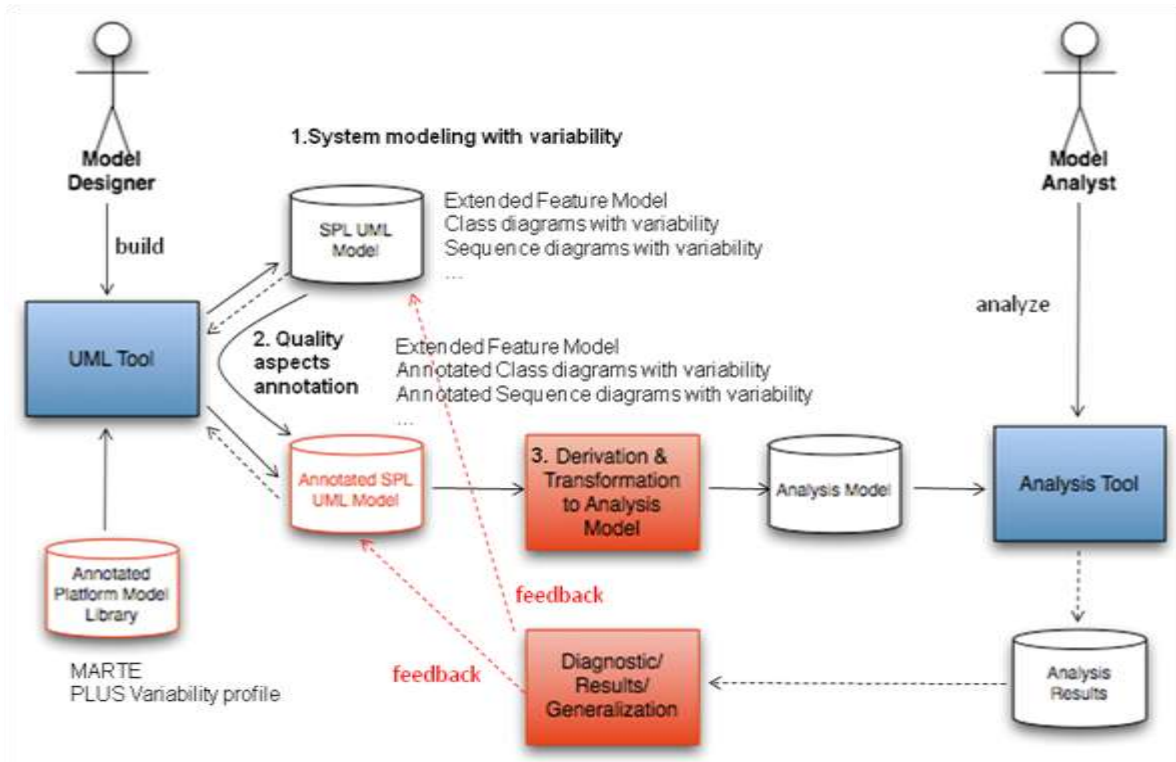


Figure 5-1– Validation process in a system with variability

The process proposed in MARTE’s specification [1] has been extended to address variability; the following steps must be performed:

5.1 System modelling with variability

This step consists on modelling the system or family of systems with variability. As a reminder, variability is understood as both functional (variation of functionalities) and quality variability (variability on quality requirements: different priority levels of performance and timeliness requirements depending on the product). These aspects are crucial to ensure the correctness by design of embedded systems. Different causes for this variability can be considered:

- Variability for addressing a family of products with different functionalities or design (software product lines). Examples of software product lines may include a family of products with different functionalities for high end, low end and mid-range markets. Or a family of products with same functionalities but deployed in different platforms.
- Variability as different design alternatives. During design, different design alternatives (at high level and also at lower level) are usually considered. Addressing this variability allows

performing analysis of the impact of each alternative on quality requirements, as well as, trade-off analysis.

So, variability in hardware, variability in functionalities, variability in design... must be modelled. For modelling this variability the extended feature model is used (see D6.1.A deliverable) and variability is modelled in design models using Gomaa's UML profile [2].

In order to illustrate the methodology for managing some extracts of an example based on the *Arcade game maker pedagogical product line* developed by SEI (Software Engineering Institute) [8] are presented. The product line consists of three different games (Brickle, Pong and Bowling). All the products have in common the use of a set of sprites (movable and stationary), graphical animation and a set of rules about the interaction of game pieces. However, the type of sprites (puck, paddle, bowling ball, brick...) and the number is variant depending on the game as well as rules, behaviour of sprites and environment. Some of the mandatory quality requirements of the example are the following:

- Performance: The action of the game must be fast enough to seem continuous.
- Usability: The game must be attractive for the player.

To illustrate better the methodology, a new game, the possibility of using two different graphics images for sprites, optional sound feedback and two types of devices have been added.

- The added game is *SpaceWar*, a known arcade game. The specific movable sprites for this game are spacecraft (player's one and enemy's spacecrafts) and bullets shoot from the spacecrafts.
- The graphical icons or images that are used for sprites can be chosen from two different sizes which have lower or higher resolution.
- The sound support is an optional functionality to provide a sound feedback whenever there is a collision of sprites.
- Two alternatives devices are available to deploy the game: mobile (Nokia S60 CPU ARM-9) or a PC (Pentium IV).

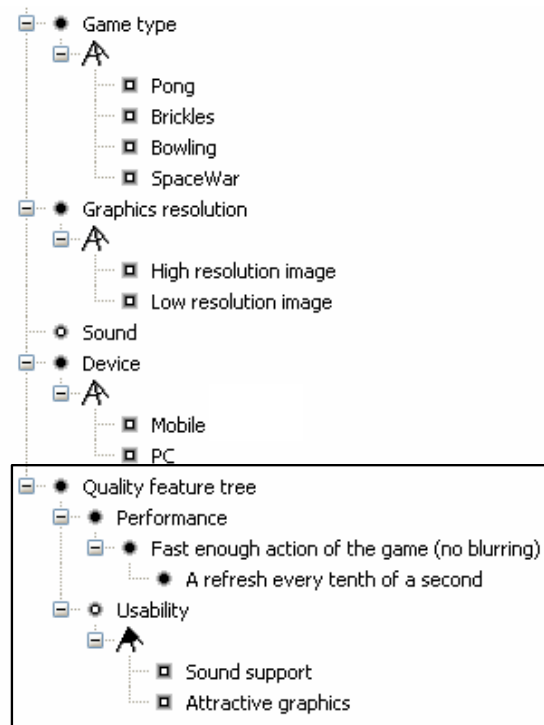


Figure 5-2– Extract of the extended feature model of the Arcade Product Line

In the Figure 5-2, an extract of the extended feature model is shown and some qualitative impacts are listed in the Table 5-1. *SpaceWar* have more movable sprites than the rest of the games, so the refreshment will be more difficult. In the same sense, *Bowling* and *Brickles* have more sprites than *Pong*.

Table 5-1: Qualitative impacts

Feature	Impact	Quality feature
<i>Brickles</i>	-	<i>A refresh every tenth of a second</i>
<i>Bowling</i>	-	<i>A refresh every tenth of a second</i>
<i>SpaceWar</i>	--	<i>A refresh every tenth of a second</i>
<i>High resolution image</i>	+	<i>Attractive graphics</i>
<i>Sound</i>	++	<i>Sound Support</i>
<i>Mobile</i>	--	<i>A refresh every tenth of a second</i>

Variability has been added in the models using the Gomaa’s profile, as example, in the next view the specialization of Game by four classes is shown.

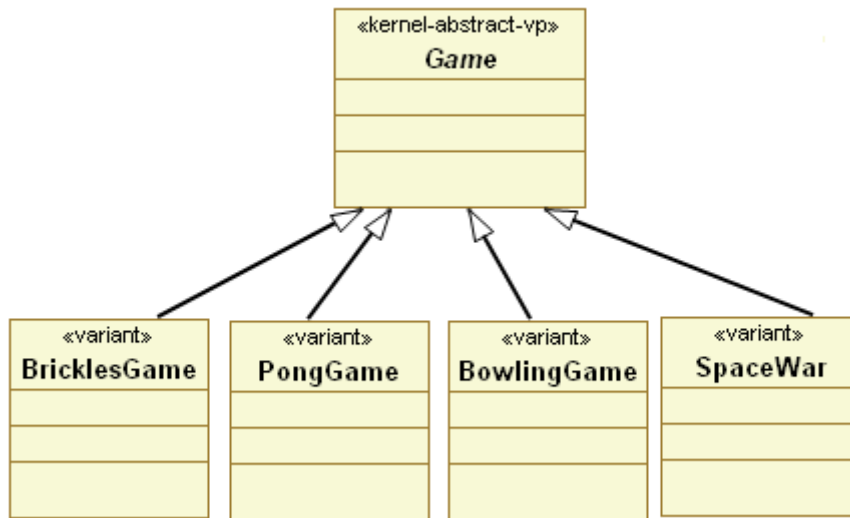


Figure 5-3– Game Generalization

Or the next view that shows the classes that specialize MovableSprite (see following figure).

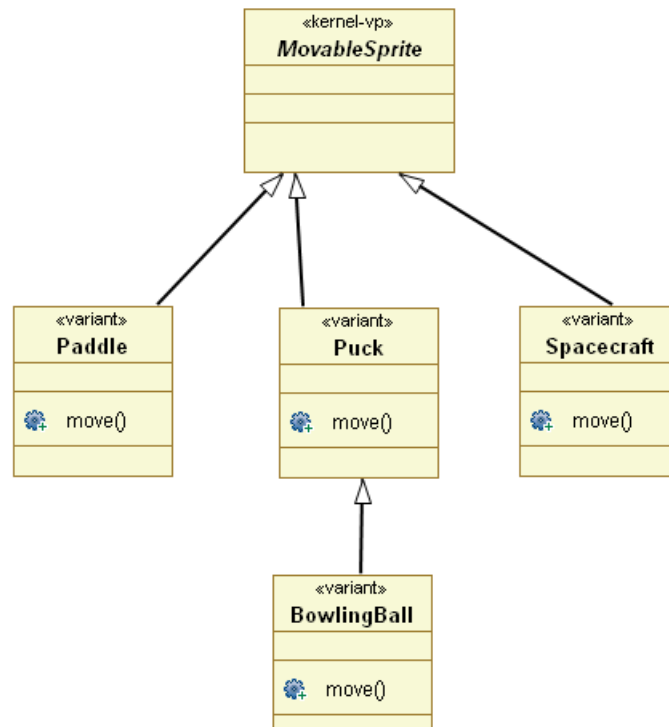


Figure 5-4– Movable sprites decomposition

The traceability among features and classes is maintained using a feature/class dependencies table as proposed by Goma [2].

Table 5-2: An extract of the Feature/class dependencies of Arcade Game Maker Product Line

Feature Name	Feature Category	Class Name	Class Category	Class Parameter
Game Type	common	Game	Kernel-abstract- vp	
		EventHandlerInterface	Kernel	
		GameBoardInterface	Kernel	
		ScoreBoardInterface	Kernel	
		SpeedControllInterface	Kernel	
		GameBoard	Kernel	
Brickles	alternative	BricklesGame	Variant	
Pong	alternative	PongGame	Variant	
Bowling	alternative	BowlingGame	Variant	
SpaceWar	alternative	SpaceWarGame	Variant	
Graphics' resolution	parameterized	Icon	Kernel-param- vp	Resolution
Device	common	Device	Kernel-abstract- vp	
Mobile	alternative	Mobile	Variant	
PC	alternative	Mobile	Variant	
Sprites	common	Sprite	Kernel	
Movable	common	MovableSprite	Kernel-abstract- vp	
Paddle	alternative	Paddle	Variant	
Puck	alternative	Puck	Variant	
BowlingBall	alternative	BowlingBall	Variant	
SpaceCraft	alternative	SpaceCraft	Variant	
...				

Variability must be added in all the models of the architecture (in all the views: application component view, system view and platform view).

5.2 Quality aspects annotation

MARTE provides facilities to annotate models with information required to perform specific analysis. Especially, MARTE focuses on performance and schedulability analysis. However, MARTE does not provide explicit means of modelling variability in performance and schedulability quality attributes.

Depending on the analysis aim, models are annotated in a different way, that is, different stereotypes are used for different purposes. In case we want to analyze the performance of a modelled system, PaRunTInstance stereotype will be annotated for example. PaRunTInstance stereotype provides an explicit connection between a locality or role in a behaviour definition (a lifeline or swimlane) and a run time instantiation of a process, and optionally defines properties of the process. SaSchedObs stereotype will be used in schedulability analysis. SaSchedObs provides prediction about scheduling metrics such as overlaps, the maximum number of suspensions caused by shared resources or the blocking time caused by the used shared resources.

Variability must be taken into account during annotation:

- Variability in analysis: different analysis may be required depending on the product. So to apply different stereotypes may be necessary in order to be able to perform different analysis: performance, schedulability...
- Variability in values:
 - Parameters are used to manage variability in analysis context
 - Variability in stereotype attributes is specified using variables and Value Specification Language (VSL) of MARTE
 - And impacts are specified using OCL

For instance, in the Arcade Product Line Game Maker product line, time aspects has been annotated using MARTE taking into account variability aspects, in order to be able to make a performance analysis. A sequence diagram of the most critical scenario (the refreshment): “Every tenth of a second a refresh must be done, the game to be seen as continuous to the user” has been modelled (see Figure 5-5). During that refreshment, all the movable sprites must be computed to their new location, collisions among sprites must be checked and handled and all the sprites must be painted.

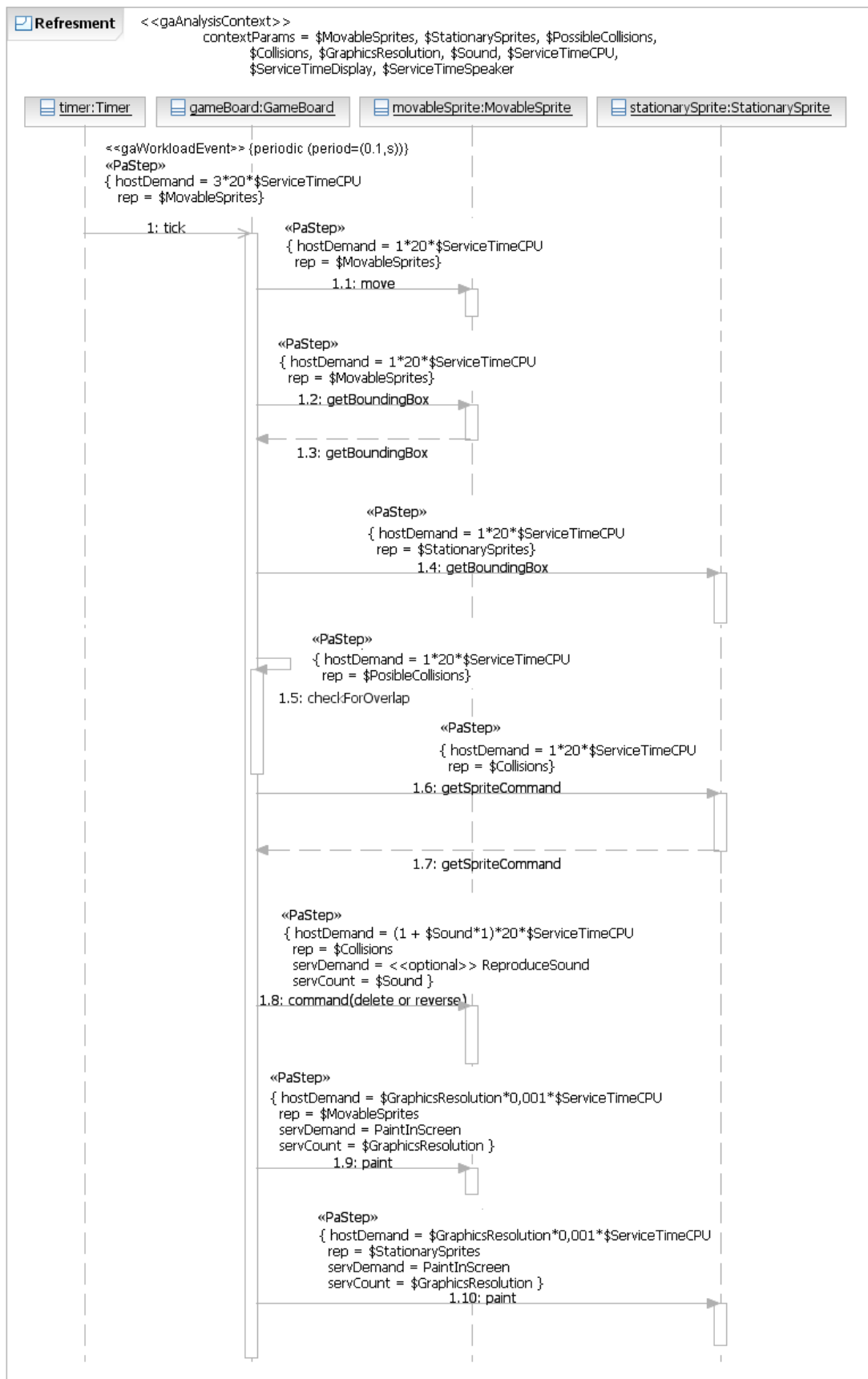


Figure 5-5– The refreshment scenario

The AnalysisContext corresponds to the scope of a study or evaluation. It combines the system represented by its behaviour (BehaviorScenarios) and its resources, with one or more workloads. It has a set of parameters which are used in expressions which define system input parameters, and which define the range of variation of cases which may arise within the study [1]. Those context parameters can be attached to the analysis context and this assists in identifying parameters that may be varied over cases in the analysis or over products in a product line.

In this case, variability is modelled using those parameters: MovableSprites (the number of movable sprites), StationarySprites (the number of stationary sprites), PossibleCollisions (the number of possible collisions depending on all the combinations of sprites), Collisions (the estimated number of real collisions), Sound (whether sound feedback will be supported or not), GraphicsResolution that determine the resolution of the images of sprites (800ppp, 1200 ppp...) and service times for executing 1 Kinstr. in the CPU or for drawing to the screen and for reproducing sound in the speaker is also variant depending on the device (ServiceTimeCPU, ServiceTimeDisplay and ServiceTimeSpeaker).

5.3 Derivation and transformation to Analysis Models

Two substeps are distinguished:

- Product derivation
- Transformation to analysis models

Product derivation consists on constructing individual products using a subset of the shared software artefacts [3] and during this process the variation points (expect those whose binding time is later on: runtime...) are bounded, an option is selected for each one. As a consequence the corresponding models for each product (without variability) are obtained. Those models are single system's models and can be the input for transforming into analysis models.

The analyzable design models will have variability. If the analysis tool and transformation engine is able to support this variability, it will be possible to perform an analysis taking into account this variability in the tool in an easy way. Otherwise (the most common case), the design models of individual products (or groups of products) must be derived and once models without variability are obtained, they can be transformed into analysis models.

UML and MARTE do not provide concrete techniques and tools for performance or schedulability evaluation, they establish a framework or lingua franca for non-functional requirement evaluation purposes of UML models. This way, many different techniques and tools can be used for evaluation [9].

For this reason, the analyzable design model specified with UML and MARTE must be transformed into technique specific analysis models such as queuing models, layered queuing models (LQN), stochastic Petri nets, stochastic process algebra models, simulation models, etc.

Different tools exist that allow performing performance or schedulability analysis such as MAST and there also bridges and plug-ins that help to transform MARTE models into analysis models such as the bridge between MARTE for RSA 7.0 to RapidRMA to perform scheduling analysis of MARTE models or the MARTE Plug-ins for scheduling analysis (RMA style) with Cheddar on UML/MARTE models [4] or UML/PEPA Bridge for Eclipse 3.2/3.3 and RSA that allows extracting performance models specified in PEPA from UML activity diagrams and sequence diagrams annotated with MARTE [9][10][11]. Those PEPA models can be analyzed using the PEPA plug-in, a software tool that supports the stochastic process algebra PEPA [12].

5.4 Analysis taking into account variability

The process and steps above can be used in order to analyze all the products of a product line (if all products are derived and analyzed). However, variability analysis step can also be performed as a way to reduce validation efforts and not analyze all the products or as a way to compare different alternatives in order to select the best option.

The validation of all the products of a line or a the validation of a very variable system may be very expensive. Thus, ways of reducing the analysis efforts can be necessary but assuring that all the products of the line will achieve the required quality attribute levels. Two different strategies can be used:

- Impacts modelled specifying the influences among variability and quality attributes can guide a selection of a subset of representative products or designs to analyze and extrapolate the results to all the products. For instance, the evaluation of products in the limits helps to assure that mandatory scenarios are met in all the products.
- Another strategy is to create a generic analysis model with variability that helps to evaluate the line reducing efforts, as several products can be evaluated together with that model. There are mechanisms that can help to manage variability; for instance for analysis models that can have variables that take different values; it is possible to define the analysis cases outside the UML specification by defining groups of values of variables, with one group of values for each case. These may be expressed in a table with a column for each case and a row for each variable.

The results of analysis taking into account variability, helps to check whether all the quality requirements (mandatory, optional or alternative) are met and decide whether redesigning is necessary; For instance, the evaluation of products in

the limits helps to assure that mandatory scenarios are met in all the products. While analyzing the results, it must be possible to assure that the range of values for the selected quality attributes can be reached in the products of the line.

But analysis taking into account variability can be especially relevant when there is variability in design or in platforms; to help to compare and select the most adequate option regarding performance or other quality attributes (comparing the analysis results). In this case, the comparison of results will be the focus and the products to analyze will be selected thinking on that. For instance, analysis can help to get the worst execution time for each alternative platform and this way select the best one.

In the illustrative example, for the context parameters: MovableSprites (the number of movable sprites), StationarySprites (the number of stationary sprites), PossibleCollisions (the number of possible collisions depending on all the combinations of sprites), Collisions (the estimated number of real collisions), GraphicsResolution, Sound and Service times; a traceability table has been defined with the values of those parameters for each game and variability aspects in the worst case.

$$PossibleCollisions = \binom{MovableSprites}{2} + MovableSprites * StationarySprites$$

Table 5-3: Traceability table for the worst case

Feature	Variant	Collisions	StationarySprites	MovableSprites	GraphicsResolution	Sound	ServiceTimeCPU	ServiceTimeDisplay	ServiceTimeSpeaker
Game type	Pong	1	4	3					
	Bowling	12	16	2					
	Brickles	3	36	4					
	SpaceWar	32	4	32					
Graphics Resolution	High				1200ppp				
	Low				800ppp				
Sound	Yes					1			

	No					0			
Device	CPU						0,00001	0,002	0,002
	Mobile						0,00612	0,1	0,1

Starting from the sequence diagram (part of the analyzable design model specified using MARTE) of the most critical scenario (the refreshment): “Every tenth of a second a refresh must be done, the game to be seen as continuous to the user”, it is possible to get a generic performance model with variability that can be analyzed. The time for refreshment value is a function of the parameters specified in the table and a formula (see Figure 5-6) has been got that is applicable for every product with the parameters that change depending on the products. Applying the formula, the time that is need for refreshment is got.

n_m : Number of movable sprites = \$MovableSprites
 n_s : Number of stationary sprites = \$StationarySprites
 n_c : Number of collisions = \$Collisions
 r : \$GraphicsResolution
 s : \$Sound
 c : \$ServiceTimeCPU
 d : \$ServiceTimeDisplay
 p : \$ServiceTimeSpeaker

$$f = \left[80n_m + 20(n_m + n_s) + 20 \left[\binom{n_m}{2} + n_m n_s \right] + 40 + 20s n_c + 0,001r(n_m + n_s) \right] c + 0,001(n_m + n_s) d + p n_c$$

Figure 5-6– Formula to calculate the refreshment time

This way, the evaluation of different products is very easy; the only thing to do is to apply the formula with the corresponding values in the variables (see Table 5-4).

Especially, the limit products can be evaluated: the product with the highest refresh time. This way, the fulfilment of mandatory scenarios is checked. In the case of the new game: *Spacewar* in a mobile, the time is higher that 100 milliseconds so it does not meet the scenario “Refresh every tenth of a second”. The formula is got using the highest number of movable sprites and the highest number of possible collisions for that game (the worst case). As this game has considerably more sprites than the rest, it cannot be assured that it will finish all the processing steps, in the interval of 100 milliseconds.

Table 5-4: Refreshment time of products

Game type	Graphics Resolution	Sound	Device	Refreshment Time
Pong	High	Yes	CPU	0,009498
Pong	High	Yes	Mobile	4,680908
Pong	High	No	CPU	0,007298
Pong	High	No	Mobile	4,458508
Pong	Low	Yes	CPU	0,00947
Pong	Low	Yes	Mobile	4,663772
Pong	Low	No	CPU	0,00727
Pong	Low	No	Mobile	4,441372
Bowling	High	Yes	CPU	0,043252
Bowling	High	Yes	Mobile	12,961992
Bowling	High	No	CPU	0,016852
Bowling	High	No	Mobile	10,293192
Bowling	Low	Yes	CPU	0,04318
Bowling	Low	Yes	Mobile	12,917928
Bowling	Low	No	CPU	0,01678
Bowling	Low	No	Mobile	10,249128
Brickles	High	Yes	CPU	0,04956
Brickles	High	Yes	Mobile	26,91376
Brickles	High	No	CPU	0,04296
Brickles	High	No	Mobile	26,24656
Brickles	Low	Yes	CPU	0,0494
Brickles	Low	Yes	Mobile	26,81584
Brickles	Low	No	CPU	0,0428
Brickles	Low	No	Mobile	21,87664
SpaceWar	High	Yes	CPU	0,241304
SpaceWar	High	Yes	Mobile	111,669584
SpaceWar	High	No	CPU	0,170904
SpaceWar	High	No	Mobile	104,552784

SpaceWar	Low	Yes	CPU	0,24116
SpaceWar	Low	Yes	Mobile	111,581456
SpaceWar	Low	No	CPU	0,17076
SpaceWar	Low	No	Mobile	104,464656

6. Conclusions

This document presents the detailed methodology for embedded systems design, variability management and early V&V proposed for eDIANA. This methodology extends and refines the methodology proposed in D2.1-A and has been develop taking into account the results of D6.1-A.

Currently, the methodology supports design and early schedulability and performance evaluation, along with variability management capacities; however, it is envisaged that this methodology could grow to support further V&V techniques, such as power consumption analysis, safety analysis, deadlock detection, etc.

Regarding usability, the methodology described in this document in currently being developed as the main prototype of WP6, and it will be released in D6.1-C.

Acknowledgements

The eDIANA Consortium would like to acknowledge the financial support of the European Commission and National Public Authorities from Spain, Netherlands, Germany, Finland and Italy under the ARTEMIS Joint Technology Initiative.

References

- [1] OMG, A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems, Version 1.0, OMG Adopted Specification: ptc/2009-11-02, 2009
- [2] Hassan Gomaa. Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. Addison Wesley, 2004.
- [3] Deelstra, S., Sinnema, M., Bosch, J.: Experiences in Software Product Families: Problems and Issues During Product Derivation, In Nord, R.L.(ed): Software Product lines, Third International Conference, SPLC 3, Proceedings, LNCS 3154 Springer (2004) 165-182
- [4] The official OMG MARTE Web site: MARTE Related Tools, <http://www.omgmarTE.org/Tools.htm>
- [5] Eila Ovaska, András Balogh, Sergio Campos, Adrian Noguero, András Pataricza, Kari Tiensyrjä & Josetxo Vicedo. "Model and Quality Driven Embedded Systems Engineering", VTT publications 705, 2009.
- [6] eDIANA Consortium, "D2.1-A Model Driven Engineering methodology for architecture realisation", eDIANA, May 2009.
- [7] Jain, R. 1991. The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modelling. John Wiley & Sons, Inc., 1991, 685 p.
- [8] SEI. Arcade Game Maker Pedagogical Product Line. Web page: <http://www.sei.cmu.edu/productlines/ppl/index.html>, 2007.
- [9] Mirco Tribastone, Stephen Gilmore, Automatic Extraction of PEPA Performance Models from UML Activity Diagrams Annotated with the MARTE Profile, Proceedings of the 7th international workshop on Software and performance, WOSP'08, 67-78 , ACM, 2008
- [10] Mirco Tribastone, Stephen Gilmore, Automatic Translation of UML Sequence Diagrams into PEPA Models, QEST, Proceedings of the 2008 Fifth International Conference on Quantitative Evaluation of Systems, 205-214, IEEE Computer Society, 2008
- [11] The UML/PEPA Bridge, Web page, <http://homepages.inf.ed.ac.uk/mtribast/uml/index.html>
- [12] The PEPA Plug-in Project, Web page, <http://www.dcs.ed.ac.uk/pepa/tools/plugin/>