MARTES

Model-Based Approach for Real-Time Embedded Systems development

Specification of the Model Transformations (PIM, PSM) in the MARTES methodology

Deliverable ID: 1.6
Version: 1.0
Date: 13 June 2007
Editor: Dennis Alders
Status: Final version
Confidentiality: Public
This document is part of the work of the EUREKA Σ! 2023 – ITEA 04006 project MARTES. Copyright © 2005-2006-2007 MARTES consortium.

Authors:

<table>
<thead>
<tr>
<th>Author</th>
<th>Partner</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>Erwin de Kock</td>
<td>Philips</td>
<td><a href="mailto:erwin.de.kock@philips.com">erwin.de.kock@philips.com</a></td>
</tr>
<tr>
<td>Jari Kreku</td>
<td>VTT</td>
<td><a href="mailto:jari.kreku@vtt.fi">jari.kreku@vtt.fi</a></td>
</tr>
<tr>
<td>Kari Tiensyrjä</td>
<td>VTT</td>
<td><a href="mailto:kari.tiensyrja@vtt.fi">kari.tiensyrja@vtt.fi</a></td>
</tr>
<tr>
<td>Martin Höst</td>
<td>Lund University</td>
<td><a href="mailto:martin.host@telecom.lth.se">martin.host@telecom.lth.se</a></td>
</tr>
<tr>
<td>Per Andersson</td>
<td>Lund University</td>
<td><a href="mailto:pera@cs.lth.se">pera@cs.lth.se</a></td>
</tr>
<tr>
<td>Michel Barreteau</td>
<td>Thales TRT</td>
<td><a href="mailto:michel.barreteau@thalesgroup.com">michel.barreteau@thalesgroup.com</a></td>
</tr>
<tr>
<td>Johan Devos</td>
<td>Barco</td>
<td><a href="mailto:johan.r.devos@barco.com">johan.r.devos@barco.com</a></td>
</tr>
<tr>
<td>Walter Tibboel</td>
<td>Philips</td>
<td><a href="mailto:walter.tibboel@philips.com">walter.tibboel@philips.com</a></td>
</tr>
<tr>
<td>Stefan Van Baelen</td>
<td>K.U. Leuven</td>
<td><a href="mailto:stefan.vanbaelen@cs.kuleuven.be">stefan.vanbaelen@cs.kuleuven.be</a></td>
</tr>
<tr>
<td>Aram Hovsepyan</td>
<td>K.U. Leuven</td>
<td><a href="mailto:aram.hovsepyan@cs.kuleuven.be">aram.hovsepyan@cs.kuleuven.be</a></td>
</tr>
<tr>
<td>Thierry Saunier</td>
<td>Thales DLJ</td>
<td><a href="mailto:thierry.saunier@fr.thalesgroup.com">thierry.saunier@fr.thalesgroup.com</a></td>
</tr>
<tr>
<td>Vincent Perrier</td>
<td>CoFluent</td>
<td><a href="mailto:vincent.perrier@cofluentdesign.com">vincent.perrier@cofluentdesign.com</a></td>
</tr>
<tr>
<td>Dennis Alders</td>
<td>NXP</td>
<td><a href="mailto:dennis.alders@nxp.com">dennis.alders@nxp.com</a></td>
</tr>
<tr>
<td>Fernando López</td>
<td>TI+D</td>
<td><a href="mailto:fla@tid.es">fla@tid.es</a></td>
</tr>
<tr>
<td>Natividad Martínez</td>
<td>UC3M</td>
<td><a href="mailto:nati@it.uc3m.es">nati@it.uc3m.es</a></td>
</tr>
<tr>
<td>Ralf Seepold</td>
<td>UC3M</td>
<td><a href="mailto:ralf@it.uc3m.es">ralf@it.uc3m.es</a></td>
</tr>
<tr>
<td>Julio Cano</td>
<td>UC3M</td>
<td><a href="mailto:jcano@inv.it.uc3m.es">jcano@inv.it.uc3m.es</a></td>
</tr>
<tr>
<td>Pablo Sánchez</td>
<td>UC</td>
<td><a href="mailto:sanchez@teisa.unican.es">sanchez@teisa.unican.es</a></td>
</tr>
</tbody>
</table>

Document History:

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Editor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>10/07/06</td>
<td>1.0</td>
<td>Erwin de Kock</td>
<td>D1.3 release.</td>
</tr>
<tr>
<td>16/02/07</td>
<td>0.1</td>
<td>Dennis Alders</td>
<td>Initial outline based on D1.3</td>
</tr>
<tr>
<td>26/02/07</td>
<td>0.1_BARC</td>
<td>Johan Devos R.</td>
<td>Minor change 3.11.1.3</td>
</tr>
<tr>
<td>27/02/07</td>
<td>0.1_VTT</td>
<td>Kari Tiensyrjä Jari Kreku</td>
<td>VTT updates to D1.6 added to sections 3.5.2, 3.7.2 and 3.10.2</td>
</tr>
<tr>
<td>Date</td>
<td>Version</td>
<td>Author/Contributor</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>26/03/07</td>
<td>0.1</td>
<td>Fernando López</td>
<td>TI+D updates to D1.6 added to sections 3.10.3, change to some images and general modifications of the original document.&quot;</td>
</tr>
<tr>
<td>07/03/07</td>
<td>0.2</td>
<td>Dennis Alders</td>
<td>First Version</td>
</tr>
<tr>
<td>27/04/07</td>
<td>0.28</td>
<td>Fernando López</td>
<td>TI+D and UC3M modifications and inputs, with UC revision.</td>
</tr>
<tr>
<td>30/04/07</td>
<td>0.2_VTT</td>
<td>Jari Kreku, Kari Tiensyrjä</td>
<td>VTT contribution updated.</td>
</tr>
<tr>
<td>4/05/07</td>
<td>0.3</td>
<td>Dennis Alders</td>
<td>Second Version</td>
</tr>
<tr>
<td>22/05/07</td>
<td>0.31</td>
<td>Kari Tiensyrjä</td>
<td>VTT comments on v. 0.3</td>
</tr>
<tr>
<td>05/07/07</td>
<td>0.32</td>
<td>Klaus Kronlöf</td>
<td>Nokia’s review corrections and comments</td>
</tr>
<tr>
<td>13/07/07</td>
<td>1.0</td>
<td>Dennis Alders</td>
<td>Final Version</td>
</tr>
</tbody>
</table>

Filename: MARTES_D1.6_final.doc
### TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Executive Summary</td>
<td>9</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>10</td>
</tr>
<tr>
<td>2 MARTES Models</td>
<td>12</td>
</tr>
<tr>
<td>3 MARTES Model Transformations</td>
<td>14</td>
</tr>
<tr>
<td>3.1 Transformation Characteristics</td>
<td>14</td>
</tr>
<tr>
<td>3.2 Classification of model transformations</td>
<td>15</td>
</tr>
<tr>
<td>3.2.1 Transformation Type</td>
<td>15</td>
</tr>
<tr>
<td>3.2.2 Execution Type</td>
<td>16</td>
</tr>
<tr>
<td>3.2.3 Implementation Type</td>
<td>17</td>
</tr>
<tr>
<td>3.3 MARTES model transformation scheme</td>
<td>17</td>
</tr>
<tr>
<td>3.4 Generic upsilon transformation pattern</td>
<td>19</td>
</tr>
<tr>
<td>3.5 Requirement model to application model transformations</td>
<td>21</td>
</tr>
<tr>
<td>3.5.1 UML model transformations</td>
<td>22</td>
</tr>
<tr>
<td>3.6 Requirement Model to Execution Platform Model Transformations</td>
<td>24</td>
</tr>
<tr>
<td>3.6.1 UML model transformations</td>
<td>24</td>
</tr>
<tr>
<td>3.7 Application model to application model transformations</td>
<td>24</td>
</tr>
<tr>
<td>3.7.1 UML model transformations</td>
<td>24</td>
</tr>
<tr>
<td>3.7.2 Trace model transformations</td>
<td>27</td>
</tr>
<tr>
<td>3.7.3 C/C++ model transformations</td>
<td>29</td>
</tr>
<tr>
<td>3.8 Execution platform model to execution platform model transformations</td>
<td>34</td>
</tr>
<tr>
<td>3.8.1 UML model transformations</td>
<td>34</td>
</tr>
<tr>
<td>3.9 Application-execution platform-allocation model to allocated model transformations</td>
<td>37</td>
</tr>
<tr>
<td>3.9.1 UML model transformations</td>
<td>37</td>
</tr>
<tr>
<td>3.9.2 Trace model transformations</td>
<td>42</td>
</tr>
<tr>
<td>3.10 Allocated model to allocated model transformations</td>
<td>43</td>
</tr>
<tr>
<td>3.10.1 UML model transformations</td>
<td>43</td>
</tr>
<tr>
<td>3.11 Allocated model to implementation model transformations</td>
<td>46</td>
</tr>
<tr>
<td>3.11.1 UML model transformations</td>
<td>46</td>
</tr>
<tr>
<td>3.12 Implementation model to SystemC or Java transformations</td>
<td>50</td>
</tr>
<tr>
<td>3.12.1 UML model transformations</td>
<td>50</td>
</tr>
<tr>
<td>3.12.2 Trace model transformations</td>
<td>51</td>
</tr>
<tr>
<td>3.13 Implementation model to Java transformations</td>
<td>54</td>
</tr>
<tr>
<td>3.13.1 Java model transformations</td>
<td>54</td>
</tr>
<tr>
<td>3.13.2 Java Executable</td>
<td>54</td>
</tr>
<tr>
<td>3.14 Application model to C/C++ transformations</td>
<td>55</td>
</tr>
<tr>
<td>3.14.1 UML-based UTF to PV model transformations</td>
<td>55</td>
</tr>
</tbody>
</table>
3.14.2 C/C++-based UTF to PV model transformations ........................................... 56
3.15 Application model to SystemC transformations .................................................. 60
    3.15.1 C/C++-based UTF to CA model transformations ........................................ 60
4 Conclusion ................................................................................................................ 65
5 Terms and Definitions ............................................................................................... 66
6 Acronyms .................................................................................................................... 69
7 References .................................................................................................................. 70
8 Appendix A .................................................................................................................. 71
    8.1 Producer-Consumer example in YAPI C++ ....................................................... 71
        8.1.1 pc.h ....................................................................................................... 71
        8.1.2 pc.cc ..................................................................................................... 71
        8.1.3 producer.h ............................................................................................ 72
        8.1.4 producer.cc .......................................................................................... 72
        8.1.5 consumer.h ........................................................................................... 73
        8.1.6 consumer.cc .......................................................................................... 73
    8.2 Producer-Consumer example in TTL C++ ......................................................... 74
        8.2.1 PC.h ....................................................................................................... 74
        8.2.2 PC.cc ..................................................................................................... 75
        8.2.3 PCMain.cc .............................................................................................. 75
        8.2.4 Producer.h ............................................................................................. 76
        8.2.5 Producer.cc ............................................................................................ 76
        8.2.6 ProducerMain.cc ..................................................................................... 77
        8.2.7 Consumer.h ........................................................................................... 77
        8.2.8 Consumer.cc .......................................................................................... 78
        8.2.9 ConsumerMain.cc .................................................................................. 78
9 Appendix B .................................................................................................................. 79
    9.1 CoFluent meta-models .......................................................................................... 79
        9.1.1 Application meta-model/timed functional .............................................. 79
        9.1.2 Execution platform model/timed functional .......................................... 79
        9.1.3 Allocation model/timed functional ......................................................... 80
        9.1.4 Allocated model/yimed Functional ....................................................... 81
        9.1.5 Implementation model/timed functional ............................................... 81
        9.1.6 SystemC model/timed functional ............................................................ 81
    9.2 CoFluent models transformations ......................................................................... 82
        9.2.1 Application model (to implementation model) to SystemC ................... 82
        9.2.2 Application-platform-allocation model to allocated model ..................... 82
        9.2.3 Allocated model (to implementation model) to SystemC ....................... 83

© Copyright MARTES Consortium
LIST OF FIGURES

Figure 1. Model Transformation Pattern [20]................................................................. 14
Figure 2. Process Overview.......................................................................................... 18
Figure 3. MARTES Y-chart scheme. ..................................................................... 19
Figure 4. Hardware-software Y-chart model............................................................... 19
Figure 5. The generic upsilon transformation pattern................................................. 20
Figure 6. Example of Specification of Requirements with Uses Case Model .............. 23
Figure 7. Highest-level application model for the pilot display example...................... 26
Figure 8. Producer-Consumer pattern model............................................................... 26
Figure 9. Producer-Consumer pattern instantiation model........................................... 27
Figure 10. Result of the application model to application model transformation............ 27
Figure 11. Transformation from application to workload models............................... 28
Figure 12. JPEG decoding process network................................................................... 33
Figure 13. Data parallel JPEG decoding process network............................................ 33
Figure 14. Fine-grain JPEG decoding process network.................................................. 34
Figure 15. Highest level execution platform model....................................................... 36
Figure 16. Execution platform communication pattern model....................................... 36
Figure 17. Execution platform communication pattern instantiation model.................. 36
Figure 18. Result of execution platform model to execution platform model transformation.. 37
Figure 19. Allocation model as the execution platform pattern instantiation model........ 37
Figure 20. Allocated model as the result of the application, execution platform and instantiation model to allocated model transformation................................................... 39
Figure 21. Structure of the MARTES profile............................................................... 40
Figure 22. Hierarchical structure of execution platform models..................................... 42
Figure 23. Transformation from workload to execution platform models..................... 43
Figure 24. The controller pattern................................................................................. 44
Figure 25. Controller pattern instantiation................................................................. 45
Figure 26. Low-level allocated model ...................................................................... 45
Figure 27. Allocated Model to Implementation Model transformation ......................... 47
Figure 28. Getter/Setter pattern for attributes......................................................... 48
Figure 29. Getter/Setter pattern for navigable associations with multiplicity one............. 49
Figure 30. Add/Remove methods for associations with multiplicity larger than one........... 49
Figure 31. The resulting implementation model.......................................................... 50
Figure 32. Transformation of UML workload to TLM SystemC.................................... 52
Figure 33. Position of behavioral synthesis in the design flow...................................... 60
Figure 34. Behavioral synthesis input and output......................................................... 61
Figure 35. RT level protocol of the signals................................................................ 63
Figure 36. Interface synthesis. ........................................................................................................... 63
Figure 37. CoFluent allocated model example. .................................................................................. 81

LIST OF TABLES
Table 1. Main transformation relations in the Allocated to Implementation transform. ................. 48
Table 2. Example of platform layer services. .................................................................................... 52
Table 3. Example of sub-system layer services. ............................................................................. 53
Table 4. Low-level interface for transferring load information from workloads to the execution platform model. ................................................................................................. 53
Table 5. RB-in from IMC to signals. .................................................................................................. 62
Executive Summary

This document constitutes the MARTES project deliverable D1.6 “Specification of the model transformations (PIM, PSM)” version 1.

The goal of this document is to identify model transformations that are used in design processes for embedded systems. The models cover a broad scope ranging from high-level customer and functional requirements, high-level system performance, executable functional specifications, down to cycle-accurate implementations. The model transformations describe small design steps that may be automated by tools. They do not describe a single design process; rather they can be used as building blocks for many possible design processes.

Chapter 1 provides an introduction to model transformation. It contains a recipe to describe model transformations and introduces the model transformation processes.

Chapter 2 revisits the MARTES models. It summarizes the models, identifies abstractions levels, and briefly describes the modeling languages and techniques that are in scope.

Chapter 3 describes the MARTES model transformations. We identified the following model transformation classes:

1. requirement model to application model transformations,
2. application model to application model transformations,
3. execution platform model to execution platform model transformations,
4. application-platform-allocation model to allocated model transformations,
5. allocated model to implementation model transformations,
6. implementation model to SystemC and Java transformations,
7. application model to C/C++ and Java transformations, and
8. application model to SystemC transformations.

The first class of transformations deals with requirements engineering and tries to formally capture requirements. The next four classes of transformations are typical transformations to explore the design space. The emphasis is on exploration of the application, execution platform, and allocation and scheduling space. The last three classes of transformations relate to further implementation of a suitable point in the design space in a System Transaction Level Modeling (TLM) or Instruction Set Simulation (ISS) environment. Here, hardware is modeled in SystemC and software is modeled in C, C++ or Java.

The model transformations described in this document will serve as a basis for the description of detailed views of models and transformations in the MARTES project deliverable D2.1.
1 Introduction

The purpose of this document is to specify the model transformations in the MARTES methodology. MARTES model transformations are transformations from one or more MARTES models into one or more other MARTES models. These models are defined in Deliverable D1.5 [1] “Specification of the Models in the MARTES methodology”. A specification of a model transformation is a specification of

- input models to the transformation,
- output models of the transformation,
- design decisions that have to be taken to transform input to output,
- design constraints that have to be satisfied to obtain feasible outputs, and optionally
- design criteria that quantify the quality of feasible outputs.

This document does not specify the MARTES instances of model transformations, although it contains instances of model transformations for illustration purposes. The MARTES instances of the model transformations will be derived from Deliverable D3.2 “Case study specifications” [2]. Furthermore, this document does not specify algorithms to solve model transformations, i.e., to find feasible or optimal instances of output models and design decisions for given instances of input models.

In Model Driven Engineering, model transformations are complementary to models: model is the static dimension and model transformation is the dynamic dimension. Model transformation is the visible user effect transforming a source model into a target model. This complementary goes further when we consider that a model transformation is also a model.

The three main stakes of the model transformation are:

- to improve the software engineering productivity by automating recurrent activities and raise the level of abstraction of the software specification,
- to capitalize transformation expertise, and
- to apply these stakes to the model transformation engineering itself, since model transformation development is a software activity and has itself the same stakes.

We can define a model transformation by the consecutive application of several transformation steps that are the bases of a transformation process.

A transformation can be decomposed into three steps:

- **Query step.** Definition in the original model of the part that will be concerned by the transformation. We have to make a selection.
- **View step.** Get a given selection from an initial model, which is a model taken from another one, called a view. Views can be used in several application contexts but the purpose of a view is always to see the useful part of a model at a given time to avoid useless information that is contained in the whole model that can be huge.
- **Transformation step.** Application of the transformation and production of a new element that will be a part of the whole output model of the transformation.
A transformation rule is nothing else than a function with outputs produced from inputs, a clear purpose and a description.

Rules should be decomposed into atomic rules. For any decomposition, there is a logical assembly where every link should have a justification and every rule should have a clear effect.

Composition and reuse of links between rules define an organization. The architecture, stating for this organization, is a key point to have pertinent (efficient) and evolving model transformation models. For instance:

- The set of rules associated to a transformation should be segmented into subsets to avoid large and monolithic sets of rules.
- Rule components should be created for abstraction and reuse with defined contracts on their interfaces.
- Generic transformations should be created to reduce maintenance costs.
- Interoperability for tool-independence should be adopted.
- Rule strategies should be developed.
- Concerns should be separated.
- Model transformation should be classified into several families according the kind of transformation, particular problem and/or associated solutions (same meta-model for the input and output models, different meta-model fully or partially disjoined or included…)

Traceability is essential for iterative model transformations. A trace link should be created or updated manually by the user or automatically during the model transformation. A trace link is not only a link between a source and a target; it provides the context:

- Source and Target identifiers. An identifier shall be unique to avoid ambiguity to find a model element.
- Update date. This field should be useful for the existence synchronization model transformation to delete old elements.
- Mapping role. In a meta-model, a same class may play many roles in a mapping specification; a role is required to unambiguously identify which role a meta-model class in a mapping plays.

Traceability issues will be treated in the MARTES Deliverable D1.7 "Specification of the Model Driven Engineering Process".
2 MARTES Models

Here we summarize the key elements of Deliverable 1.5 “Specification of the models (PIM, PSM) in the MARTES methodology [1]. There we identify five models:

- the MARTES Requirements Model
- the MARTES Application Model,
- the MARTES Execution Platform Model,
- the MARTES Allocation Model,
- the MARTES Allocated Model, and
- the MARTES Implementation-Oriented Model.

The inspiration for these models comes from the Y-chart introduced by Kienhuis et al. [3]. Furthermore we identify the following modeling styles:

- Informal models,
- Trace-based models,
- Un-Timed Functional models (UTF),
- Timed Functional models (TF),
- Programmer View models (PV),
- Programmer View with Time models (PVT), and
- Cycle Accurate models (CA).

We use here the term modeling styles rather than the more conventional abstraction levels, since in our terminology the abstraction level is defined by the domain concepts used in the model, not the style used for modeling. For example, UTF (or TF) can describe a wide variety of things at different abstraction levels starting from end-user observable behavior (use case) down to some details of system internal behavior.

We adopt the informal models from the UML community. We adopt the trace-based modeling abstraction level from Lieverse et al. [4] and the other SystemC TLM abstraction levels from Grotker et al. [5]. In MARTES we use the five models at each of the abstraction levels. Each abstraction level has the following characteristics.

- Informal models are not executable. Typically they describe or model stakeholder requirements and constraints, for instance, in English text or in UML. In the requirements phase, it is possible to define through models the functional requirements of the system through use case diagrams, and through formal methods to describe the scenarios of those use cases or functions. Going further, it is possible to describe use cases using sequence diagrams or even create executable models of the use cases, but such models fall into the styles of trace-based models or UTF/TF.

- Trace-based application models are not executable. Rather traces model application behavior (for a given data set). The corresponding execution platform, allocation, allocated, and implementation-oriented model operate on traces rather than on a descriptive application model. Trace transformations transform application trace elements into execution platform primitives. For instance Lieverse et al. [6] transform
read and write function calls into check data, load, and signal room primitives and check room, store, and signal data primitives, respectively. The motivation of the transformation is to better match application event with execution platform primitives in order to obtain more accurate performance analyses. Pimentel and Erbas [7] use an integer data-flow model for application event transformation.

- UTF application models are executable but they do not explicitly describe entities (e.g. threads and variables) that can be allocated to execution platform resources (e.g. processors and memories). There is no time present in the models. The corresponding execution platform, allocation, allocated, and implementation-oriented models are not based on memory maps.

- TF models are executable. They contain timing annotations that represent timing constraints or timing estimates. The timing annotations can be the result of allocating application entities to execution platform entities (e.g. as in the CoFluent approach).

- PV models focus on modeling memory maps. Application models define the entities that have to be mapped. Execution platform models define the target resources. Allocation models define the design decisions that need to be taken in order to map applications onto execution platforms. Allocated models define the design decisions that have been taken to map applications onto execution platforms. Implementation-oriented platforms describe the implementation of applications on execution platforms.

- PVT models focus on modeling (cycle-approximate) time for given memory maps. Whereas PV models focus on allocation of activities to resources, PVT models focus on sequencing of activities on allocated resources. Therefore, it is possible to illustrate the implementation of a PVT model by looking at a practical timed design as well as inter-module arbitration policies.

- CA models focus on bit and cycle accuracy. Cycle Accurate models are required to perform architecture verification, performance analysis, architectural exploration, tuning of application code and power estimation. PV models suffice the needs of driver development but to perform architectural exploration, power estimation, etc. CA models are required.

In the next chapter we present the MARTES transformations between these models.
3 MARTES Model Transformations

Model transformations, also called mappings, are used to transform one or more input specifications into one or more output specifications.

3.1 Transformation Characteristics

Model transformation represents the process of converting one or more models into an output model of the same system [18]. Mappings and relations are defined as specializations of a transformation. Contrary to a mapping, which is defined as a unidirectional transformation, a relation is presented as bidirectional. Note that the result of one transformation is another model.

Model transformation languages are used to specify a model transformation. These model transformation languages are defined in a metamodel level and establish the relationship between source metamodel elements and target metamodel elements, as presented in Figure 1, which is an evolution of pattern Bezivin [27].

![Activity Diagram of Model Transformation](image)

**Figure 1. Model Transformation Pattern [20].**

As it can be seen in Figure 1, some information must to be available for the Transformation Engine to perform a model transformation. Basically, this kind of information comes from the
Source Metamodel and the Target Metamodel primitives that we are using to describe both, the model from which we are about to perform the transformation, and the model into which we want to translate the first one. This type of information can be patterns, templates or UML profiles. Another additional information can be used to guide the transformation resulting from analyzing and mixing all the information mentioned above, different transformation relations, between the Source Model and the Target model primitives are established. The Transformation Rules are defined as a result of this process.

TI+D and UC3M use XSLT Rules as Transformation Rules, XSLT Language as Transformation Language, XMI Language as Source & Target Language, and XML will be the MetaMetaModel.

In the case of the University of Cantabria (UC), the mapping rules will be implemented in an external program that uses XML as source language and SystemC as target language.

3.2 Classification of model transformations

Several types of transformations can be identified based on the way their input and output domains are related:

- A **vertical refinement transformation** is a transformation whereby a model of a given level of abstraction is transformed into a model of a lower level of abstraction. Such transformation is called a refinement since it refines a certain model into a model containing more details.

- A **vertical abstraction transformation** is a transformation whereby a model of a given level of abstraction is transformed into a model of a higher level of abstraction. Mappings that transform code into models are known as "reverse engineering" tools.

- A **horizontal transformation** does not change the level of abstraction, but changes the structure of the input specification or weaves multiple specifications into a single output specification (aggregation mapping).

- Some transformations consist of both a horizontal transformation, adapting the structure of the specification, and a vertical transformation, changing the level of abstraction (e.g. compilers first build an abstract syntax tree before generating code). From the outside, these transformations appear to be a single vertical refinement transformation. Such complex combined transformations are called **oblique transformations**.

The qualities of a transformation are its safety - is the target model a faithful implementation of the source model? - and its efficiency - is the complexity of the target model equivalent to the complexity of the source model?

Also, we have to consider the following criteria in the transformations:

- Type
- Execution type
- Implementation type.

3.2.1 Transformation Type

In a MDD process the transformations can be of the type
• CIM¹ to PIM, named requirement model to application model transformation
• PIM to PIM, application model to application model transformation or application-execution platform- allocation model to allocated model.
• PIM to PSM, named allocated model to implementation model transformation
• PSM to PSM, implementation model to implementation model transformation
• PSM to Code, implementation model to Java transformation.

A CIM (Requirements Model) can be defined into another CIM, also PIM and PSM can be made too. We have not considered the reverse transformation in this project.

TI+D and UC3M concentrate their works in the PIM to PSM transformation, called Allocated Model to Implementation Model transformation within MARTES methodology context. UC concentrates its works in the PIM to code transformations (mapping rules that generate PIM trace models) and PSM to Code (implementation model to SystemC transformation).

### 3.2.2 Execution Type

There are many methods for transformations:

• The transformation may be done by hand. This is the less safe method, and efficiency is not guaranteed either.

• A more realistic approach is to map components by hand, and to use an automatic or semi-automatic tool for linking the target components isomorphically to the source structure. An example of this approach is the use of libraries (BLAS, STL, etc) in ordinary programming. In this approach, the needed tools are (1) a language for expressing the structure of the source model, and (2) a linker for constructing the target model. The method is safe as soon as the component models are proved equivalent. However, it might be sub-optimal because optimizations that would cut across components borders are forbidden.

• At the other end of the spectrum, one may contemplate fully automatic transformation. This is the usual solution in ordinary programming: modern compilers are safe and efficient enough that it is not considered cost effective to improve their output by hand. The situation is less clear for hardware synthesis: hand crafting a circuit or part of a circuit is still used for critical jobs, like the implementation of a general-purpose processor. In the case of embedded systems, the critical issue is that of parallelization. Parallelism is necessary first because embedded systems have to deal with the physical world, which is parallel. Second, some embedded systems (or some stages of embedded systems) need processing power beyond what a single processor can provide, for instance for signal processing. Lastly, parallelism is often the key for power reduction. Parallelizing compilers do exist, but are limited to simple, regular programs. It seems that communicating process networks strike the right balance between manual parallelization among processes and automatic parallelization inside processes.

¹ CIM is the Computational Independent Model and we use that to refer to the Requirements Model.
• Still another approach, which is actively pursued at INRIA, consists of a manual mapping followed by automatic or semi-automatic verification. The rationale here is that most problems in efficient model transformations are NP-complete. Hence, calculating the transformed model may have exponential complexity, but checking the result as given by a human designer has only polynomial complexity. Tools for this approach range from several kinds of model checkers to proof assistants like Coq or PVS.

3.2.3 Implementation Type

Transformation Rules can be implemented in:

• general programming language as Java, C++, php,…
• graph transformation language as AGG [20] and VIATRA [21]
• languages for transformations such as ATL [22], QVT [23] or XSLT.

3.3 MARTES model transformation scheme

The main characteristic of the MARTES model transformation process is that it is a semi-automatic, model-driven and transformation-based process. This process is defined in Figure 2.
Figure 2. Process Overview.

Figure 3 shows the general MARTES Y-chart transformation and mapping scheme. This scheme may be applied iteratively.

- The MARTES Requirements Model is mapped into (or used as a constraints model for) the MARTES Application Model describing the Software System on the one hand, and the MARTES Execution Platform Model describing the Hardware System on the other hand.

- A MARTES Allocation Model is constructed in order to associate the applicative functions onto the infrastructure resources, hereby following the so-called Y-chart model [3] as presented in Figure 4.

- The MARTES Application Model, Execution Platform Model, and Allocation Model are aggregated into the MARTES Allocated Model.

- The MARTES Allocated Model is transformed into the MARTES Implementation-oriented Model.

- The MARTES Implementation Model is used as a base for code generation towards C, SystemC, C++ and Java.

- The MARTES Application Model can be used in order to directly generate Simulation Models or code, without constructing the MARTES Allocation Model, Allocated Model or Implementation-Oriented Model.
3.4 Generic upsilon transformation pattern

In this section we describe the generic upsilon transformation (GUT) pattern developed by K.U. Leuven, where each model transformation step in a transformation chain can be defined
using an upsilon transformation, consisting of a source model, a pattern model and a pattern instantiation.

Four main types of model transformation are model refinement, model weaving, model extraction (reverse engineering resulting in a model creation from source code) and model refactoring. Our focus for applying the generic upsilon transformation pattern will be mainly on model refinement and model weaving, although the proposed approach could also be used for model refactoring.

Independent from the domain of the underlying software application, most model transformations usually follow a certain pattern. If a given model transformation does not follow any pattern it is specific to one input model and cannot be reused which limits its usability. Thus a model transformation can be seen as a pattern application. By pattern we do not only mean the commonly accepted definition of a design pattern. It could be a software pattern (e.g. EJB pattern), hardware pattern (e.g. communication pattern) or other kind of pattern (e.g. mapping of a software system to hardware components).

Typically these patterns are realized within the model transformator whereby the pattern mapping is realized implicitly, because it is hidden within the transformation step. An alternative to patterns "hidden" in a transformator is to model them explicitly. A generic model transformator could take as an input the pattern model, the original model and the mapping information and produce the same result as a specific model transformation.

K.U.Leuven developed an approach where each model transformation is realized as a pattern application represented by an upsilon transformation pattern. The left branch of the upsilon represents the original model that we would like to transform. The right branch is a generic definition of the pattern to be applied. Along with the two branches we also provide a pattern instantiation description that instructs the model transformator how particular elements from the original model map on the pattern, as presented in Figure 5. Elements that are not present in the pattern instantiation description are just copied. Elements that should be removed during the transformation must be labeled as such within the pattern (see further).

![Figure 5. The generic upsilon transformation pattern.](image)
In order to allow a flexible mapping of the original model to the pattern we have developed a UML profile, which the model transformator can use in order to generate a correctly mapped resulting model. This profile is empirically constructed from a number of case studies and can be easily extended in order to enrich the transformation process when needed.

<<merge>>

This stereotype is used to denote a merge between a single entity from the original model and a single entity from the pattern. The merge will result into a single merged element for each 2 merged entities. When applied on classes for example, the model transformator combines the two classes along with their attributes and methods.

<<remove>>

This stereotype is used to denote the removal of a link between two entities. Usually the link does not just disappear as such, but other entities or connectors in the pattern replace its functionality.

<<delegate>>

This stereotype is used to denote a mapping between an entity from the original model and an element from the pattern. The model transformator creates an association between each of the involved classes. Certain functionality is assigned to the delegated class however this information is specified at a lower level.

<<hardware>>

This stereotype is used to denote a class representing a hardware entity. Usually a firmware code should be created from this stereotype.

<<process>>

Practically all software components that run on a hardware capable of doing computations are tagged with this stereotype.

<<controller>>

Classes, which are denoted by the controller stereotype, are usually lower level interfaces to software and hardware components

The main advantage of this approach lies in its flexibility. Existing patterns could be modified; new patterns could be introduced without any knowledge of the specific model transformation language. The pattern mapping is explicit and easier to understand than the internal stereotype binding. Model transformations could be interchanged without any compatibility between the transformation languages.

The generalized upsilon transformations (GUTs) will be applied and illustrated within a number of transformation steps that are defined in the next sections.

3.5 Requirement model to application model transformations

One of the first steps for designing an application is to figure out what are going to be the main functionalities, so the proper tool to express this kind of aspects is the use case.

A use case is a technique used in order to capture potentials requirements for a new system; in our case this system means a service. In general, a use case represents the interaction among the different elements needed in order to compose a service. These elements used
belong to several different domains, and today, the most part of services are based on distributed architectures over data communications.

### 3.5.1 UML model transformations

Both divisions of Barco (Barco Avionics & Barco Command and Control) participating in the MARTES project are active in the market of safety critical, real-time embedded systems. More specifically, Barco is often confronted with a demand for compliance towards the Do-178B standard. This standard describes the boundary conditions imposed on a software development lifecycle process as to ensure that the resulting software product is of the necessary level of dependability to be safely deployed onboard an aircraft. Two cornerstones within this standard are requirements development and traceability/refinement of these requirements throughout the different software development lifecycle artifacts.

However, the current practices employed in Barco are not completely satisfactory. Customers often provide system requirements, which are vague, not formally specified and incomplete/contradicting in some areas. The necessary preconditions on how such a set of system requirements can be consciously and methodically be transformed into a coherent, complete and formal set of detailed requirements are not fully understood. Barco wants to explore how this requirements formalization process can be enhanced with the end goal of creating a precise requirements model within UML.

From Telefónica I+D and UC3M point of view, we can explain the UML Model Transformation with the following example. Let’s imagine we want to establish a communication using the Session Initiation Protocol (SIP), then the user which desires to establish such communication, must register in the system as active user. After registering, the user must send a communication invitation to another user which should be registered as active user; this user should accept or deny the invitation finishing the communication establishment process.

The use case diagram is represented by the Figure 6: It defines the actors of the use case and their relations. Often this is complemented by some kind of behavior description, such as natural language text sequence diagram, to define the sequencing of the (inter)actions of the use case.
In many cases the designer already knows how he is going to implement certain functionality, so the information reflected in the course of basic events is often a description of the steps needed to accomplish this. In general, it would be enough to specify a reference protocol. However, one should not forget why it is useful to specify use cases; it is to ease the communication between the stakeholders, knowing that not all of them are interested in implementation details that are common for the designer (for instance: usability, service launching, portability, module reusability, etc.)

As we can see, the diagram functionally represents the requirements of the application model. In this example we have divided the communication establishment "problem", in several sub problems. This provides us the level of abstraction to reason about the functionality aspects of the service. The communication is established after the user registers himself at the SIP server. He can exchange information with others by sending an invitation, but this is implicit in the conversation concept.

It is necessary to specify how to carry out those transformations between the requirement model and the application model. Our starting point is a UML use case diagram (remember it means functionality diagram). The only thing we have to do is to add meaning in the UML diagram application model terms: “what are the relationships among the different components of the system”. Due to this aspect the transformation can only be realized manually. For instance, in the SIP example, a two process bidirectional communication could be the application model.
3.6 Requirement Model to Execution Platform Model Transformations

As is described above, the requirement model is expressed through UML use case diagrams and associated behavior descriptions. An Execution Platform Model specifies the hardware requirements of the system. This provides one the expressiveness to decide how those processes at the application model are going to be implemented in the final physical implementation. For example if a process or a thread is used, the way the communication is implemented via a message queue or by shared memory, etc.

3.6.1 UML model transformations

Due to the diversification of hardware terminals, and the huge amount of technologies and architectures present in services development, it is necessary at do this transformation manually. For example each device could integrate a different architecture, and it will be very difficult to reach a good approximation for almost all of them.

3.7 Application model to application model transformations

3.7.1 UML model transformations

The purpose of Application model to Application model transformations is to refine and optimize the Application model achieved previously. An Application Component refinement may be required in order to achieve a complete model. One may even introduce new components that were not present in the original application model.

As a result of this transformation we will obtain an output UML 2 model that represents a lower-level Application Model, expressing enriched details of the application under development. Note that both, source and output models, must be Application MetaModel compliant.

Some information must to be available for the Transformation Engine to perform this model transformation process:

- The metamodel definitions (Application MetaModel) used to describe the Application Model.
- Transformation rules. Three different transformation can be considered, depending on the way these rules are established and the way we apply them to the source model: automatic, semi-automatic or manual.
- The source model itself that must be seen as an instance of the Application MetaModel. It is a UML 2 model that represents a high-level Application Model.

Besides this kind of transformations can be done following an automatic or semi-automatic procedure, TI+D and UC3M approach do no consider any automatic o semi-automatic transformation at this point of the process. This way, any Application Model enrichment transformation will be handmade.

The issues in UML-based application model transformations are:

- model refinement: multiple iterations of PIM-to-PIM and PSM-to-PSM transformations,
- traceability, and
PIM to PSM mapping, using configuration parameters.

The purpose of application model to application model transformations is to refine the application model by introducing more detail and/or applying software design patterns on the application model. The specification of the application model to application model transformation is as follows:

- The input model to the transformation is a UML 2 model that represents a high-level application model.
- The output model of the transformation is a UML 2 model that represents a lower-level application model containing more detail.
- The design decision that has to be taken to transform input to output is the amount of detail that is added to the model and/or the software design pattern that is introduced into the model.
- There are no general design constraints that have to be satisfied in order to obtain feasible outputs, although each transformation can have certain specific design constraints.
- There are no general design criteria that quantify the quality of feasible outputs, although each transformation can have certain specific design criteria.

To illustrate the transformation in more detail, we show for the producer-consumer example how the input model, the transformation and the resulting output model could look like. As the system into which the producer-consumer example will be applied, we have chosen an aircraft display system.

On any aircraft there are hundreds of different sensors that collect information on the airplane position, speed, distance from near objects etc. The pilot cabin has a display screen where the relevant information is shown. Even though the number of displays is usually more than one we assume that there is only one display screen that shows all the information. The sensors might send raw data to the displays, but usually send it to a central processing unit called a data filter where the information is processed and sent to the display.

We assume that there are general-purpose hardware boards on which all the sensors, display and data filter are implemented. All the information is sent between the boards via a communication bus. To ensure the correct message delivery a data-allocation protocol is implemented.

We will start the example from a high-level MARTES application and execution platform model. In this and the following sections, we will refine them following the MARTES Y-chart scheme and applying the generic epsilon transformation pattern consecutively until we reach the MARTES implementation-oriented model which is the platform specific model for a C++ platform. Notice this example could be implemented using a platform specific model for a Java platform but is not shown in this document since the used methodology is the same but considering the differences among both languages (both are object-oriented...)

On the highest abstraction level we would like to have a model that does not have to deal with any lower-level details such as hardware information, memory allocation etc. Thus the highest-level application model presented in Figure 7 contains only the sensors, pilot display and data filter entities. The sensors might send raw data to the display, but usually send it to
a central processing unit called a data filter where the information is processed and sent to the display. Associations denote the communication between the three.

Figure 7. Highest-level application model for the pilot display example.

We identify the necessity of introducing the producer-consumer software pattern in this model and thus we would like to apply this pattern on the application model. Figure 8 represents the producer-consumer pattern, whereby a producer produces some data and passes it to a number of consumers that consume it. An association class “Communication” represents this sending of data. Figure 9 presents the mapping of the original application model to the pattern. Sensor only produces data, display consumes it and data filter consumes the signal from the sensors and produces a signal for the display. The <<delegate>> stereotype is used to map each class from the Application Model to an entity from the pattern. As the communication between the original entities should now be realized by the Communication class in Producer-Consumer pattern instead of using a direct association, we have used the <<remove>> stereotype on the associations in the Application Model in order to replace the direct associations by the Communication class.

Notice that the producer-consumer pattern has been applied three times: for the Aircraft Sensor to Pilot Display channel, for the Aircraft Sensor to Data Filter Channel and a third time for the Data Filter to the Pilot Display channel.

Notice that we use the generic upsilon transformation (GUT) pattern to realize this application model to application model transformation.

Figure 8. Producer-Consumer pattern model.
3.7.2 Trace model transformations

In the Performance Modeling and Evaluation approach, the application is modeled using the service-oriented approach described in [26]. The application model is transformed to a layered, hierarchical workload model as depicted in Figure 11.
The purpose of workload models is to enable performance evaluation in early phases of the design process, since creating the models does not require the applications to be finalized.

The transformation from the application model to the higher layers of the workload model is currently done manually. The transformation is based on the simulation of the use case and functional architecture models of the application. The traces of the event and message transactions are transformed to control structures of the higher layers (main, application and process level) of the workload model. In the service-oriented application modeling, the application sub-systems are modeled in terms of services provided by the execution platform, which cause most of the loads to the lower layers (function, basic block and abstract instruction levels) of the workload model. The loads of the functions of application sub-systems are also modeled and added to the lower layers of the workload model.

The workload models describe the computing and communication loads applications cause to the resources of the execution platform. The workload models are presented in UML as described in detail in the MARTES deliverable report D1.5 [1] and summarized in the sequel.

Workload models have a hierarchical structure, where main workload model $W$ divides into application workloads $A_i$ for different processing units of the physical architecture model.

Each application workload $A_i$ is constructed of one or more processes $P_i$. 

![Diagram](https://via.placeholder.com/150)

*Figure 11. Transformation from application to workload models.*
The processes are comprised of function workloads $F_i$.

Function workload models are basically control flow graphs, where nodes are basic blocks and arcs are branches. Basic blocks are ordered sets of load primitives, e.g. abstract instructions, which are used for load characterization. For example, these abstract instructions could be read and write for modeling memory accesses and execute for modeling data processing.

There are a number of methods for obtaining the load information for the workload models from different sources. In the analytical method, the number of memory accessing and data processing operations is estimated from the algorithm description or other source. This is most suitable for simple DSP like applications (e.g. static data flow models), where the control flows are simple enough so that such estimations can be made with reasonable accuracy. However, it is often very difficult to get a realistic distribution for the operations with real-life applications.

Another alternative is source code based method, which relies on a compiler tool chain for producing the required information for the workload models. The source code of the application or of a very similar application has to be available and it has to be mature enough that it can be compiled at least in a workstation environment if not in the target architecture. In this method, a profiler is used to obtain control information for the workload models and to limit the modeling work to the most loading functions. The compiler front-end can be used to produce the load primitives for the models.

The third approach for obtaining the load information is trace or measurement based method. In this case, an existing execution trace or measured load information of a similar application in a resembling architecture is used as input. Multiple independent traces or measurements can be combined to model complex use cases. This is a rapid way for producing workload models but also a bit limited due to the prerequisites.

The workload models use interface functions that are provided by the layered hierarchical execution platform models. Specifically, interfaces are needed in the subsequent performance simulation for:

- Transferring load information from the workload model side to the execution platform side (load functions)
- Controlling the execution platform model from the workload models (control functions)
- Utilizing services provided by the execution platform (service functions).

A more detailed description of the interface functions is presented in Section 3.12.2.

### 3.7.3 C/C++ model transformations

Here we consider transformations to optimize the resource requirements in terms of required operations. The assumption is that designers have a rough idea of the performance and cost of calls to platform services such as communication, synchronization, and scheduling. The purpose and relevance of the transformations is to optimize applications that are written as process networks with respect to cost of using platform services. We consider four transformations:

- YAPI-to-TTL transformation,
• interface transformation,
• data parallelism transformation, and
• task merging transformation.

We discuss these transformations in the following sections.

3.7.3.1 YAPI-to-TTL Transformation

The purpose of the YAPI-to-TTL transformation is to reflect decisions on buffer sizes of communication channels in application code. Next to that the purpose is to change the syntax to describe process networks from YAPI to TTL in order to distinguish between the roles of application development and application mapping. The purpose of YAPI is to support application development, while the purpose of TTL is to support application mapping. To this end, TTL provides more and lower-level primitives compared to YAPI. The specification of the YAPI-to-TTL transformation is as follows.

• The input model to the transformation is a process network described in C++ using the YAPI C++ software library.
• The output model of the transformation is a process network described in C++ using the TTL C++ software library.
• The design decision that has to be taken to transform input to output is the buffer size, i.e., the number of buffer elements, of each communication channel.
• The design constraint that has to be satisfied to obtain feasible output is that the buffer sizes are sufficiently large to avoid deadlock.
• The design criterion that quantifies the quality of feasible output is the sum of the buffer sizes. The goal is to minimize this sum.

To explain the transformation in more detail we make use of a small producer-consumer application. First we present the input to the transformation, which is the YAPI program for this application. The complete source code of this program is listed in Appendix 8.1. Here we only list the functionality of the producer and consumer processes (see previous page).

Second we present the output of the transformation, which is the TTL program for this application. The complete source code of this program is listed in Appendix 8.2. Note that there is no transformation of functionality. Only interfaces and interconnections have changed syntactically.

```c++
14 void Producer::main()
15 {
16   write(p, n);
17   for (int i=0; i<n; i++)
18     write(p, i);
19 }
20
21 }
22 }
```

```c++
16 void Consumer::main()
17 {
18   int n;
19   read(p, n);
20   for (int i=0; i<n; i++)
21     {
22       int j;
23       read(p, j);
24       assert(i==j);
25     }
26   }
27 }
```
Third we present the design decisions that are input for the transformation. These are described in the file ‘PC.ttl’:

```
01 MAPPING PC
02 {
03    TRANSPORT
04    {
05        a["256"]
06        {
07            TTL::Channel<int>
08        }
09    }
10    NET
11    {
12        fifo -> a
13    }
14    MAPPING Consumer : c
15    {
16    }
17    MAPPING Producer : p
18    {
19    }
20 }
```

At line 05 a TTL channel called ‘a’ is introduced that contains a buffer of 256 elements. The elements are of type integer. At Line 12 the YAPI channel called ‘fifo’ is mapped onto TTL channel ‘a’. The rest of the file is not relevant at this moment.

Finally, the design constraints and design criteria are not made explicit.

### 3.7.3.2 Interface Transformation

The purpose of interface transformation is to optimize the interaction between processes and communication channels in process networks. To this end, TTL provides several interface types, which have been presented in MARTES Deliverable D1.5. The specification of the interface transformation is as follows:

- The input model to the transformation is a process network described in C++ using the TTL C++ software library.
- The output model of the transformation is a process network described in C++ using the TTL C++ software library.
- The design decision that has to be taken to transform input to output is the interface type and the interface protocol for each port of each task.
- The design constraint that has to be satisfied to obtain feasible output is that the interface types of ports that are connected by a communication channel are equal.
- The design criteria that quantify the quality of feasible output are the sum of buffer sizes, the sum of memory requirements, and the number of interface function calls.

To explain the transformation in more detail, we show how the design decisions and the resulting output look like for the producer-consumer example. For the design decisions Line 07 of the file ‘PC.ttl’ changes into

```
07        TTL::Channel<int> USING RbOut, RbIn
```
This specifies that the communication channel ‘a’ must be transformed into a communication channel with relative blocking output and input interfaces. In the original line combined blocking interfaces were assumed. As a result Lines 32 and 44 of the file ‘Producer.h’ and Line 14 of the file ‘Producer.cc’ change from Cb to Rb. Similarly, Lines 31 and 40 of the file ‘Consumer.h’ and Line 18 of the file ‘Consumer.cc’ change.

Also the main functions of the producer and consumer change into:

```
15 void Producer::main()
16 {
17    reAcquireRoom(p, 1);
18    store(p, 0, n);
19    releaseData(p, 1);
20    for (int i = 0; i < n; i++)
21    {
22       reAcquireRoom(p, 1);
23       store(p, 0, i);
24       releaseData(p, 1);
25    }
26 }
```

```
18 void Consumer::main()
19 {
20    int n;
21    reAcquireData(p, 1);
22    load(p, 0, n);
23    releaseRoom(p, 1);
24    for (int i = 0; i < n; i++)
25    {
26       int j;
27       reAcquireData(p, 1);
28       load(p, 0, j);
29       releaseRoom(p, 1);
30       assert(i == j);
31    }
32 }
```

The above-mentioned transformation does not have any direct advantages. The advantages become clear if the calls to the synchronization functions in the loop bodies are moved outside the loop bodies. This reduces the number of synchronization actions. The resulting code looks as follows:

```
15 void Producer::main()
16 {
17    reAcquireRoom(p, n+1);
18    store(p, 0, n);
19    for (int i = 0; i < n; i++)
20    {
21       store(p, i+1, i);
22    }
23    releaseData(p, n+1);
24 }
```

```
18 void Consumer::main()
19 {
20    int n;
21    reAcquireData(p, 1);
22    load(p, 0, n);
23    reAcquireData(p, n+1);
24    for (int i = 0; i < n; i++)
25    {
26       int j;
27       load(p, i+1, j);
28       assert(i == j);
29    }
30    releaseRoom(p, n+1);
31 }
```

### 3.7.3.3 Data Parallelism Transformation

The purpose of the data parallelism transformation is to distribute the processing of a data stream over multiple instances of the same task. The benefit of this transformation is that the throughput of the process network increases provided that the task instances can be executed in parallel. The specification of the transformation is as follows.

- The input model to the transformation is a process network described in C++ using the TTL C++ software library.
The output model of the transformation is a process network described in C++ using the TTL C++ software library.

The design decision that has to be taken to transform input to output is the number of instances of each task and the data to be processed by each instance.

The design constraint that has to be satisfied to obtain feasible output is the interleaving of the output of all task instances is equal to the output of the original single task instance.

The design criteria that quantify the quality of feasible output are the number of task instances and the amount of data to be processed by each task instance. The goal is to distribute the workload between the different task instances in such a way that the each task instances satisfies its throughput constraints.

As an example of the transformation we show JPEG decoding process networks before transformation in Figure 12 and after transformation in Figure 13. The variable length decoded data stream is input for two-dimensional inverse discrete cosine transformation and rastering (i.e. pixel reordering to scan rate order). This data stream consists of Y, U, and V data that can be processed independently of each other.

In order to perform rastering, image properties such width and height are needed. These properties need to be communicated to all instances of the rastering task.

3.7.3.4 Task Merging Transformation

The purpose of the task merging transformation is to optimize the number of application threads (assuming that each task is mapped into one thread). This is important because data communication between threads consumes resources. The amount of resources spent on communication should be compensated by a gain in performance due to parallel execution of
threads. Hence, a thread should perform a sufficiently large amount of computation compared to its communication. The specification of the transformation is as follows.

- The input model to the transformation is a process network described in C++ using the TTL C++ software library.
- The output model of the transformation is a process network described in C++ using the TTL C++ software library.
- The design decision that has to be taken to transform input to output is the partitioning of tasks into disjoint subsets in which each subset forms a new task.
- The design constraint that has to be satisfied to obtain feasible output is that task merging does not introduce deadlock in a process network.
- The design criteria that quantify the quality of feasible output are the number of subsets in the partition, the number of communication channels between the subsets, and the number of calls of each interface function. The goal is to distribute the workload between the sets in the task partition such that each set satisfies its computation and communication constraints.

As an example of task merging we compare Figure 14 before merging and Figure 12 after merging.

![Figure 14. Fine-grain JPEG decoding process network.](image)

Task merging can be implemented in different ways. One way is to merge code into one single piece of code. Another way is to implement a run-time scheduler inside the new task that schedules the original tasks that have been combined in the new task.

### 3.8 Execution platform model to execution platform model transformations

#### 3.8.1 UML model transformations

The purpose of the Execution Platform Model to Execution Platform model transformation is similar to the Application model to Application model transformations explained before. Once
again, this transformation allows to refine and optimize the Execution Platform model achieved previously.

As a result of this transformation we will obtain an output UML 2 model that represents a lower-level Execution Platform Model, which expresses enriched details of the original Execution Platform Model under development. Again, note that both, source and output models, must be Execution Platform MetaModel compliant.

In order to be able to achieve this transformation, some basic information must to be available for the Transformation Engine to perform this model transformation process:

- The metamodel definitions (Execution Platform MetaModel) used to describe the Execution Platform Model.
- Transformation rules. Three different transformation can be considered, depending on the way these rules are established and the way we apply them to the source model: automatic, semi-automatic or manual.
- The source model itself, that must be seen as an instance of the Execution Platform MetaModel. It is a model that represents a high-level Execution Platform Model.

In a similar way to the Application model to Application Model transformation, and besides this kind of transformations can be done following an automatic or semi-automatic procedure, TI+D and UC3M approach do no consider any automatic or semi-automatic transformation at this point of the process. This way, any Execution Platform Model enrichment transformation will be handmade.

Although an execution platform model is mostly developed using bottom-up composition and abstraction, it could be possible in theory to develop the execution platform model using top-down refinement of the execution platform model. To illustrate the applicability of the Generic Upsilon Transformation (GUT) pattern for execution platform transformation, we will present in this section a (possibly hypothetical) top-down execution platform model to execution platform model transformation.

The purpose of execution platform model to execution platform model transformations is to refine the execution platform model by introducing more detail on the execution platform model. The specification of the execution platform model to execution platform model transformation is as follows:

- The input model to the transformation is a UML 2 model that represents a high-level execution platform model.
- The output model of the transformation is a UML 2 model that represents a lower-level execution platform model.
- The design decision that has to be taken to transform input to output is the amount of detail that is added to the model.
- There are no general design constraints that have to be satisfied in order to obtain feasible outputs, although each transformation can have certain specific design constraints.
- There are no general design criteria that quantify the quality of feasible outputs, although each transformation can have certain specific design criteria.
To illustrate the transformation in more detail, we show for the producer-consumer example how the input model, the transformation and the resulting output model could look like. As presented in Figure 15, the hardware architecture of the Aircraft Data System is modeled separately as a set of communicating boards and a number of processes running on each board.

![Figure 15. Highest level execution platform model.](image)

We have chosen the bus pattern to realize the communication between the boards. As presented in Figure 16, the bus pattern represents a single Bus that has many BusElements connected to it. Following the GUT approach to realize this mapping, we add the mapping of the hardware architecture model to the pattern, as presented in Figure 17.

![Figure 16. Execution platform communication pattern model.](image)

![Figure 17. Execution platform communication pattern instantiation model.](image)
During the mapping we have used the <<merge>> stereotype in order to denote that the association class CommunicationChannel should be merged with the Bus, while Board is merged with BusElement. The result is depicted on Figure 18.

3.9 Application-execution platform-allocation model to allocated model transformations

3.9.1 UML model transformations

This transformation represents the assignment of Application Components and processes to Platform Components accordingly to Allocation Model rules. So three models are needed here as source models for the transformation:

- Application Model, including the Application Components and processes to allocate Platform Components.
- Execution Platform model, representing the platform components of the system and their connections.
- Allocation model, indicating the way application components and platform components can be allocated.

The following is the process to generate the allocated model:

a) Every Application Process Group contains several Application Processes with are instances of Application Component. Application Components are selected for every Application Process Group to be allocated.

b) Application Component Groups are allocated by hand to Platform components.

As a result of this transformation we will obtain an UML 2 model as output, that represents the Allocated Model. Where Application Components are associated to Platform Components.
This can be done following a semiautomatic procedure where restrictions given by the Allocation Model are checked. Automatic procedure may not be feasible to be implemented here since models do not provide enough information to determine the desired allocation.

This mapping relates stereotypes in the MARTES UML profile (Application, Execution Platform and Allocation types) to the types defined within the Allocated Model. Thus, given an element in an allocation model (which depends on Application and Execution Platform models) expressed via the UML profile, the mapping defines how to refine that element into a more detailed (platform-independent) allocated model. The Allocation model leverages the types defined in the MARTES Allocated Model.

Note that the allocation and allocated model expressed via the profile is itself platform independent, based on our relative definition of platform independence taking into account the C++, C, Java and SystemC target platforms. Thus there are two platform-independent levels of abstraction that are relevant here, although one is at a higher level of abstraction than the other; that is, the model that the system engineer creates via the profile is at a higher level of abstraction than the refined application that a tool produces using the Allocation Model-to-Allocated Model mapping.

3.9.1.1 High-Level allocated model
The purpose of application model, execution platform model and allocation model to allocated model transformations is to map the application model onto the execution platform model by using the information of the allocation model. The result of this mapping is expressed in the allocated model. The specification of the application model, execution platform model and allocation model to allocated model transformation is as follows:

- The input models to the transformation are 3 UML 2 models that represent an application model, an execution platform model and an allocation model (defining the mapping of the application model onto the platform model).
- The output model of the transformation is a UML 2 model that represents an allocated model, containing more detail or being more optimized.
- There are no design decisions that have to be taken to transform input to output, since the decision information is already contained in the allocation model.
- There are no general design constraints that have to be satisfied in order to obtain feasible outputs, although each transformation can have certain specific design constraints expressed in the allocation model.
- There are no general design criteria that quantify the quality of feasible outputs, although each transformation can have certain specific design criteria expressed in the allocation model.

To illustrate the transformation in more detail, we show an example on how the input models, the transformation and the resulting output model could look like for the producer-consumer example, following the GUT approach.

Once the application model (Figure 10) and execution platform model (Figure 18) have been refined up to the right level of abstraction we need to map the software onto the hardware in order to obtain the allocated model.
Obviously we need to map Producer and Consumer to a CommunicatingProcess element. As UML does not offer methods to be associated we have mapped the methods using the <<in>> and <<out>> stereotypes. The Communication between Producer and Consumer is removed by using the <<remove>> stereotype. This functionality is implicitly delegated to the. Figure 19 presents a possible mapping between the two. The allocated model in Figure 20 shows the result of the mapping where Sensor, Display and DataFilter communicate with each other using the producer-consumer pattern. Each Producer and Consumer is implemented on a hardware board and boards communicate via a single bus.
3.9.1.2 Low-level allocated model

The application is considered at a functional level. The architecture model at this level can be seen as a physical description (depending on the configuration from how it will be used) of the execution platform (that is hierarchy and topology between processing units, memories and busses). It is not yet executable because there is no link yet with the application to feed it.

The allocation model first includes information about a global mapping of application components to execution platform ones. As a matter of fact, the first design step consists in merging application and execution platform models (task parallelism) by defining sorts of allocation clusters (let’s call them segments for instance, or ProcessGroup as defined in D1.5 and once again shown in Figure 21) in order to associate one or several application components (tasks) to one or several execution platform resources. Here computations are allowed to processors, and as input and output working memories of each core functions are known, basic memory mapping is also known. To this end one can graphically create these logical clusters, then drag and drop both functional tasks and hardware resources to them. Hierarchical segments may be created.

Figure 21. Structure of the MARTES profile.

Allocating several tasks to the same segment means that all the resources of the same segment will be in charge of executing all these tasks. Allocating a hardware resource to several segment means that this resource will be considered as a replicated logical resource even if it is the same physical one. This resource is supposed to be managed by an operating system.

A design constraint assumes that all the execution platform resources that have been selected in the same segment must be of the same type (e.g. same DSP or same GPP).
The allocation model is now structured according to segments that include data structures (arrays) and applicative functions (tasks) from the application model, hardware resources from the execution platform model and memory maps that have been introduced thanks to the application / execution platform association.

Once the allocation model has been refined using spatial transformations (w.r.t. the functional scheduling), the next design step consists in considering temporal transformations to refine memory usage. First this can be done through the fusion transformation.

Indeed at coarse grain level, the overall scheduling of each segment is usually based on a logic of FIFOs, with blocking reads and (in general) non-blocking writes. Human involvement is needed when the segment has several inputs that can be processed individually, e.g. in order to fix priorities. At fine-grain level, scheduling is based on an affine model. Parallel execution of activities, typically computations and communications, is a key issue for performance, so that the execution of tasks within a fusion is generally arranged in pipeline.

When two tasks have the same delay, the constraints of precedence due to produced / consumed arrays apply, while the order of execution is a priori left free when delays are different. The overall order has a strong impact on hardware resource utilization (avoiding key resource waiting for available data), as well as on latency, and memory occupation.

Indeed implicitly, the natural way to schedule a sequence of tasks in a segment is to execute each of them completely, one after the other. Fusion consists in creating an upper level task, which executes sequentially its daughter tasks in all iterations, each with a reduced iteration space. In other words, the iteration spaces of all low level tasks participating to the fusion must be partitioned into a common number of tiles, such that an iteration space can be put in factor to all of them. A condition is that if task A and task B participate in the fusion, and task A produces an array which is an input for task B at any iteration of the upper level iteration space must be included into the sub-array produced by Task A during the same iteration; or, as an extension, into the union of sub-arrays produced by Task A during all iterations up to including the current one. While methods exist to compute automatically fusions, fusion is prepared here by the designer, who decides which tasks must participate and prepares the appropriate tiling for each of them. The tool has to verify the correctness of the proposed fusion, and when accepted, creates a macro-task with its iteration space and all the fused tasks as daughters.

Another additional refinement consists in graphically adding synchronization barriers to these subtasks and permuting them while respecting the scheduling dependencies. So we get a software pipelining view that shows tasks scheduled over time.

All these transformations yield the allocated model.

3.9.1.3 Step 1 of the UML to SystemC model transformation

Within the project, Lund University develops a tool for supporting translation from UML to SystemC. The tool is built as an add-in to the Telelogic UML tool. Though it is possible to define a set of mapping rules from a pure UML model directly into SystemC code, it would give the engineer little influence of the mapping and most likely a less satisfactory result. Instead we divide the mapping in three steps. During initial system modeling a pure UML model is used. During the system development the original UML model is transformed into an allocated model. Lund University will not define this transformation, but rely on the other
partners within the MARTES project for this part of the design flow. The two steps that Lund University contributes to are described in Sections 3.11.1.1 and 3.12.1.1. All models involved in the above mentioned steps are available and editable. This makes it possible for the engineer to have full control over the relevant details for the system under development and have the tool manage all remaining details.

3.9.2 Trace model transformations

In addition to the workload models described in Section 3.7.2, execution platform and allocation models serve as inputs for the transformation to the allocated model. This transformation is performed by manually defining the mapping between the workload and execution platform models.

The execution platform models consist of layered architecture models of the platforms and layered service models of the platforms as depicted in Figure 22.

**Figure 22. Hierarchical structure of execution platform models.**

The bottom layer is the component layer, one layer up is the sub-system layer, which describes the structure and interconnects of the sub-system. The next layer up is the platform-architecture layer, which includes the network interconnects and the platform software, e.g. the operating system.

The layered service models are abstraction views of the architecture models. They describe the platform behaviors and related attributes, e.g. performance, but hide other details. High-level services are built on low-level services, and they can also use the services at the same level.

The way to link the architecture models and the service models is by defining services in the service models and implementing the services in the architecture models.
For doing the Workload-Platform-Allocation, a skeleton model of the execution platform in UML is used as depicted in Figure 23. This model presents the same structural information as the SystemC execution platform model.

![Figure 23. Transformation from workload to execution platform models.](image)

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

### 3.10 Allocated model to allocated model transformations

#### 3.10.1 UML model transformations

The purpose of Allocated model to Allocated model transformations is to refine and optimize the Allocated Model obtained using the Application-Execution Platform-Allocation model to Allocated Model transformations, before transforming it into an implementation model. For instance software and hardware controllers that implement some lower level functionality could be added. The specification of the Allocated model to Allocated model is as follows:

- The input model to the transformation is a UML 2 model that represent a high-level Allocated Model.
• The output model of the transformation is a UML 2 model that represents a lower-level Allocated Model.
• The design decision that has to be taken to transform input to output is the amount of detail that is added to the model or the amount of optimization that has to be performed onto the high-level Allocated Model.
• There are no general design constraints that have to be satisfied in order to obtain feasible outputs, although each transformation can have certain specific design constraints.
• There are no general design criteria that quantify the quality of feasible outputs, although each transformation can have certain specific design criteria.

To illustrate the transformation in more detail, we show an example on how the input model, the transformation and the resulting output model could look like for the producer-consumer example, following the GUT approach.

For instance we would like to introduce producer and consumer controllers that implement methods such as memory allocation and the actual data interchange. The original producer and consumer become a higher level entities which are only concerned with the production and consumption of some data.

Therefore we have developed the Controller Pattern that is presented in Figure 24, which consists of four entities: An Element, its Environment, an ElementController and an interface that it should realize.

![Figure 24. The controller pattern.](image)

The pattern mapping in Figure 25 is described using a UML notation for pattern instantiation where we instantiate Producer as an Element, Board as its Environment and add an interface method which should be present in the newly created ProducerController. The association between Producer and Board is removed by using <<remove>> stereotype on the connector between them. The functionality of this connector is actually substituted by two newly created associations.
The same pattern with different parameters is applied on Consumer and Board elements. The final result of the pattern instantiation is shown in Figure 26.

3.10.1.1 Dependencies transformations
Application Components, like libraries, system functionality, etc. may be depend on other components. These components may not be specified in the Application Model, so a new
transformation may be needed to include them. The elements needed in these transformations are the following:

- The Allocated Model, so that it is known which platform is every Application Component associated to.
- The Application Component library, where dependencies between Application Components are indicated.

The process that can be followed to accomplish this transformation is as follows:

- For every Application Component find dependencies for the associated platform. This dependencies can be found using the Application Component Interfaces (Ports, Services, etc.) looking for new components that feed the needed information for other components.
- Include the new Components in the model and associate them with the dependent components.
- Repeat the process until all component dependencies are matched.

If dependencies cannot be matched the process fails. This occurs when no component can be found to match a certain dependency.

As a result of this transformation we will obtain an output UML 2 model, that represents a refined Allocation Model with new Application Components. TI+D and UC3M approach consider only a semi-automatic transformation at this point of the process. In case that several components can be selected to solve a dependency human interaction is needed to select the correct one.

### 3.11 Allocated model to implementation model transformations

#### 3.11.1 UML model transformations

At this point of the process, our goal is to place our application in a specific implementation environment. To achieve this implementation model, we will transform a refined Allocated Model, which is platform independent, into the corresponding Implementation model, resulting in a Platform Specific Model. TI+D and UC3M approach defines a completely automatic transformation that decreases the level of abstraction, mapping the Allocated Model semantics into the new platform specific features.

Obviously, as long as the target model is platform specific, different allocated model to implementation model transformation must be defined, depending on the target platform like OSGi/Java, J2ME and J2EE, etc.

As a result of this transformation we will obtain an output UML 2 model, that represents an implementation model, expressing details of the specific implementation environment in which the application will be executed. So that, some information must to be available for the Transformation Engine to perform the model transformation process.

- The source and target metamodels definitions (Allocated MetaModel and Implementation MetaModel) used to describe both, the model from which we are about to perform the transformation, and the model into which we want to translate the first one.
Transformation rules. Based on the source and target metamodel, different transformation relations are established. TI+D and UC3M approach packed all of these transformation rules and describes them using the XSLT language. Basically, these rules define how the Allocated Model concepts are transformed into the Implementation model.

The source model itself must be seen as an instance of the Allocated metamodel. It is an UML 2 model that represents a low-level allocated model. The diagram in Figure 27 shows all the data needed during this process.

![Figure 27. Allocated Model to Implementation Model transformation](image)

Note the design decision to transform input to output is the amount of detail added to the model, as well as the model adaptations needed to target a specific implementation environment. There are no general design constraints that have to be satisfied in order to obtain feasible outputs, although each transformation can have certain specific design constraints and must be coherent with the target metamodel (Implementation Model). There are no general design criteria that quantify the quality of feasible outputs, although each transformation can have certain specific design criteria.

Summing up, once the allocated model is at the right level of abstraction we would like to obtain the implementation model, which is the lowest level PSM that could be used to generate implementation code. The relevant details that should be added during this transformation are usually properties that express all the application semantics and the navigable associations between different classes, but from the point of view of our specific target platform environment. For instance, working on a Java/OSGi scenario, each application description component, defined at the Allocated level, will become in a Implementation component that includes all the main peculiarities related to the target technology (bundle attributes, OSGiBundleActivator, OSGIServiceInterface, etc.). This new enriched model contains all the information that allows us to generate a Java code implementation, as required by the OSGi services and its methods for managing them.
Table 1 shows the main mapping relations defined for a Allocated Model to Implementation Model transformation using a Java/OSGi Implementation Metamodel.

Table 1. Main transformation relations in the Allocated to Implementation transform.

<table>
<thead>
<tr>
<th>Allocated Model</th>
<th>Implementation Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Description</td>
<td>OSGi System</td>
</tr>
<tr>
<td>ComponentRoot</td>
<td>Bundle</td>
</tr>
<tr>
<td>Component → Attributes</td>
<td>JavaPackage</td>
</tr>
<tr>
<td>ComponentRoot → Structure</td>
<td>JavaClass Implementation</td>
</tr>
<tr>
<td>ComponentRoot → Attribute</td>
<td>OSGi Manifest</td>
</tr>
<tr>
<td>ComponentRoot → ActivityDef</td>
<td>Method</td>
</tr>
<tr>
<td>Interface</td>
<td>JavaClass Interface</td>
</tr>
<tr>
<td>Relation</td>
<td>Socket</td>
</tr>
<tr>
<td>Interface → Port</td>
<td>Input &amp; Output Port</td>
</tr>
<tr>
<td>Component</td>
<td>Database</td>
</tr>
</tbody>
</table>

To illustrate the transformation in more detail, we show an example on how the input model, the transformation and the resulting output model could look like for the producer-consumer example. Notice that we use the generic upsilon chart transformation (GUT) pattern to realize the allocated model to allocated model transformation.

Once the allocated model is at the right level of abstraction we would like to obtain the implementation model, which is the lowest level PSM that could be used to generate implementation code. The relevant details that should be added during this transformation are usually property getters and setters, derived properties created by the navigable associations between different classes, getter and setters of the derived properties. For uniformity reasons and in order to automate the model transformations, we have represented these details also as a pattern. We have modeled two patterns that generate the getters and setters in the model. The first pattern presented in Figure 28 generates getters/setters for class attributes.

![Figure 28. Getter/Setter pattern for attributes.](image)

The second pattern in Figure 29 generates derived properties and getters/setters for navigable association between two classes. In addition we have modeled a pattern in Figure
30 that generates derived properties along with add/remove methods. The latter is used for navigable associations with target multiplicity greater than 1.

All the three patterns are implicitly applied on all attributes and navigable associations. The resulting implementation model is shown on Figure 31.

Figure 29. Getter/Setter pattern for navigable associations with multiplicity one.

Figure 30. Add/Remove methods for associations with multiplicity larger than one.

3.11.1.1 Step 2 of the UML to SystemC model transformations

This transformation is one step in the SystemC code generation process. During this transformation the allocated model is transformed into a new UML model, which only includes UML constructs that have direct representations in SystemC. This means that constructs such as classes are included but not constructs such as state machines. A state machine is mapped to a class with attributes containing the state and methods implementing its behavior. A complete list of UML constructs which are removed during this transformation is beyond the scope of this document, but will be published elsewhere. The details of the mapping from UML only concepts to SystemC concepts will also be published elsewhere.

In addition to removing UML only concepts we also make all relations in the model explicit. When a class is made active in UML it implies that the class will have its own thread of execution. In SystemC this is realized using sc_thread or sc_method, which implies that the class is an instance of the SystemC class sc_module. During this transformation all such implicit relations are made explicit. For example, we add a generalization relation to the SystemC class sc_module} to all active UML classes. In the first version of the tool the resulting model will be an un-timed functional model.
3.12 Implementation model to SystemC or Java transformations

3.12.1 UML model transformations

The last transformation is devoted to code generation (configuration, target or SystemC simulation code for instance). It relies on the implementation-oriented model. The remaining work mainly consists in spanning the sequence of segments and replacing data references by their addresses following the memory allocation policy (e.g. sharing the same memory zone by several arrays, allocating to each array several memory zones in terms in its life time) and generating the underlying program using the chosen language (e.g. C, MPI, SystemC). The resulting binary code will be executable (after compilation and linking) on the target.

Regarding SystemC performance simulation, inputs are both a behavioral model of the target machine (that is strongly derived from the execution platform model with additional simulation objects to drive it) and a simulation scenario (that is composed of all the design...
transformations graphically given by the user). Performance simulation greatly helps to find interactively the appropriate ordering, by showing the expected timing of scheduled activities and their simultaneity. Memory occupancy resulting from a scheduling scheme can also be displayed. In comparison with the other transformations described in Section 3.9.1, this one is not yet implemented but also will remain internal to the tool.

3.12.1.1 Step 3 of the UML to SystemC model transformations
In this transformation the UML model resulting from the transformation described above is transformed into a corresponding SystemC code. This transformation is a one to one correspondence between the UML model and the resulting SystemC code, i.e. this is a pretty print of the UML model. This step is implemented using the existing C++ code generator from Telelogic and thus reuse its support for scope rules, header-file inclusion and make file generation without any modifications. If the generated code is to be read by humans it is desirable to use the common SystemC macros when applicable. This requires a slight customisation of the syntax of the generated code. For example we can then generate `SC_MODULE(MyModule){…}` instead of `class MyModule:public sc_module{…}`.

3.12.2 Trace model transformations
The purpose of workload model transformation from UML to SystemC is to enable performance simulation in SystemC by combining the transformed workload models with a SystemC-based execution platform model as depicted in Figure 32. Since SystemC is basically a C++ library, one method to perform this kind of transformation is to use the C++ code generator of the UML tool, e.g. Telelogic Tau. The SystemC generator developed by Telelogic and Lund University in the MARTES project provides automation to this transformation.

The transformation from a UML-based workload model to a SystemC-based workload model is performed for allocated models created as the result of the transformation described in Section 3.9.2, i.e. it has been selected, where each part of the workload model will be executed on the execution platform model.

Interfaces between the workload and execution platform model are provided by the execution platform models and used by the workload models for transferring load information, controlling the execution platform model, and utilizing services. At the platform (Table 2) and sub-system layers (Table 3) the service interfaces are domain and application specific, while the low-level interface is generic (Table 4).
Figure 32. Transformation of UML workload to TLM SystemC.

Table 2. Example of platform layer services.

<table>
<thead>
<tr>
<th>Service</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>video_playback(x, y, z)</td>
<td>Play a video with resolution rate (x,y) and digital zoom factor z. The video is taken from the memory sub-system, decoded in the image sub-system and sent to the display sub-system for viewing. The whole sequence is controlled by the general purpose sub-system.</td>
</tr>
<tr>
<td>send_command(dst, type)</td>
<td>Send a command with type to a target (dst).</td>
</tr>
<tr>
<td>data_transfer(src, dst, length)</td>
<td>Move the length of data from src to dst.</td>
</tr>
</tbody>
</table>
Table 3. Example of sub-system layer services.

<table>
<thead>
<tr>
<th>Service</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>preprocess($x, y, mx, my, z$)</td>
<td>Preprocessing for ($x, y$) resolution video with ($mx, my$) macroblocks and digital zoom factor $z$.</td>
</tr>
<tr>
<td>decode($x, y, mx, my, b, m, c$)</td>
<td>Decode ($x, y$) resolution video using macroblocks of ($mx, my$) pixels and $b$ bits per pixel. $m$ old macroblocks will be analysed in motion estimation per each new macroblock. The compression ratio of the encoded video is $c$.</td>
</tr>
<tr>
<td>encode($x, y, mx, my, b, m, c$)</td>
<td>Encode ($x, y$) resolution video using macroblocks of ($mx, my$) pixels and $b$ bits per pixel. $m$ old macroblocks will be analysed in motion estimation per each new macroblock. The compression ratio of the encoded video is $c$.</td>
</tr>
<tr>
<td>postprocess($x, y, mx, my, d$)</td>
<td>Postprocessing for ($x, y$) resolution video with ($mx, my$) macroblocks and final colour depth $d$.</td>
</tr>
<tr>
<td>dma_transfer(source, target, words, word_size)</td>
<td>Read $words \times \text{word_size}$ bits of memory from source address and write it to target address.</td>
</tr>
</tbody>
</table>

Table 4. Low-level interface for transferring load information from workloads to the execution platform model.

<table>
<thead>
<tr>
<th>Interface function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(address, words, word_size)</td>
<td>Read $words \times \text{word_size}$ bits from address.</td>
</tr>
<tr>
<td>write(address, words, word_size)</td>
<td>Write $words \times \text{word_size}$ bits to address.</td>
</tr>
<tr>
<td>execute(n)</td>
<td>Execute $n$ data processing or control instructions</td>
</tr>
<tr>
<td>set_pc(address)</td>
<td>Set program counter to address.</td>
</tr>
</tbody>
</table>

Transformation from the UML-based workload model to a SystemC-based workload model requires the interfaces to be described in UML so that the workload models are able to use them and the code generator of the UML tool can output correct SystemC code. The transformation is performed from Telelogic Tau using the SystemC generator developed by Lund University. The output model of the transformation is SystemC transaction level presentation of the allocated UML workload model.

The SystemC performance simulation model at transaction level abstraction is obtained by linking and compiling the workload model with the execution platform model that both are presented in SystemC.

The resulting SystemC models are simulated in the SystemC simulation environment to obtain performance data of the system.
3.13 Implementation model to Java transformations

3.13.1 Java model transformations

In a final step, an Implementation Model to Java code transformation is carried out. The Implementation Model components are mapped to the corresponding Java Classes, creating the whole application. This way, our approach achieves its main goal of generating Java Code from a high level application definition. The code generated is compatible with the popular Java development environment Eclipse and is defined in customizable templates so that the user can modify to optimize it.

3.13.2 Java Executable

In theory, it is not possible to create a Java executable for a non-specific platform, because Java is a multiplatform language whereas the executable files depend on the target platform. Nevertheless, several solutions can be used in order to cope with this problem. A first solution would involve the use of an specific platform, but this is obviously not valid in this context. A different solution would be to implement a script that allows an easier execution depending on the platform. However, if we want to achieve a optimal solution that permits a smart transformation between the implementation model and the executable, without taking into account the platform used, the use of a JAR file turns out to be mandatory.

A JAR file is a file format based on the popular ZIP file format, which is used for aggregating different files into one, making it executable. A JAR file is essentially a ZIP file that contains an optional META-INF directory. These files can be created in several ways and can be named using any legal name on a particular platform.

Programs intended to run on a JVM must be compiled into a standardized portable binary format, which typically comes in the form of .class files. A program may consist of many classes in different files. For easier distribution of large programs, multiple class files may be packaged together in a .jar file (short for Java archive). The JVM runtime executes .class or .jar files, emulating the JVM instruction set by interpreting it.

It is necessary to create a JAR manifest file to use this executable. The manifest file consists of a main section followed by a list of sections for individual JAR file entries, ant it includes information related to security, configuration and the required attributes for packages or files contained in the JAR file. Therefore, note that the creation of this file becomes a very important step in the implementation model transformation process. We must ensure that any relevant information is included in this file. The main section of the manifest contains the security and configuration information about the JAR file itself, as well as the application or extension that this JAR file is part of. It also defines main attributes that apply to every individual manifest entry. The individual sections define various attributes for packages of files contained in this JAR file.

The main attributes included in the manifest file can be classified in different groups:

- General main attributes: manifest-version, created-by,…
- Attribute defined for stand-alone applications: main class
- Attributes defined by applets: extension-list, <extension>extension-name, …
• Attribute defined for extension identification: extension-name
• Attributes defined for extension and package versioning and sealing information

There also exists per-entry attributes (applied only to the individual JAR file entry to which the manifest entry is associated with).

Security is also an important factor to take into account when the transformation process is faced. In this sense, a signed JAR file can be used if necessary. A signed JAR file is exactly the same as the original JAR file, except for its manifest, which is updated, and two additional files that are added to the META-INF directory: a signature file and a signature block file. The signature is always validated when the manifest is first parsed.

We also need to define a JAR index. This information is used to optimize the class searching process of class loaders for network applications. Once the class loader finds an index file in a particular jar file, it always trusts the information listed here. To prevent adding too much space overhead to the application, and to speed up the construction of the in-memory hash table, the index file is kept as small as possible.

Finally, we should include the service provider configuration files. A service is a well-known set of interfaces and classes. A service provider is a specific implementation of a service. The classes in a provider typically implement the interfaces and subclasses that classes define within the service itself. Service providers may be installed in an implementation of the Java platform in the form of extensions, that is, jar files placed into any of the usual extension directories. Providers may also be made available by adding them to the applet or application class path, or by some other platform-specific means.

A service is represented by an abstract class. A provider of a given service contains one or more concrete classes, that extend this service class with data and code specific to the provider. Typically, this provider class will not be the entire provider itself, but rather a proxy that contains enough information to decide whether the provider is able to satisfy a particular request, along with the code that can create the actual provider on demand. The details of provider classes tend to be highly service-specific; no single class or interface could possibly unify them, so no such class has been defined. The only requirement enforced here is that provider classes must have a zero-argument constructor so that they may be instantiated during lookup.

3.14 Application model to C/C++ transformations

3.14.1 UML-based UTF to PV model transformations

UTF (Un-Timed Functional) and TF (Timed Functional) to PV (Programmer View). Within these transformations, we need to concentrate on 3 important domain constraints:
• Coding standards/rules for safety critical software (e.g. MISRA-C)
• Preserve traceability
• Platform definition & constraints

3.14.1.1 Coding standards for safety critical software

© Copyright MARTES Consortium
For both Avionics and Command & Control displays, coding is restricted to a number of rules that must ensure the reliability of the software. The reference in our domain is MISRA-C, a standard originally introduced by the automotive industry. Even though this standard is designed for the C language, a lot of the rules are applicable to C++ as well. Another guide for C++ design is the update Effective C++ 3rd Edition (Scott Meyer, published by Addison Wesley). Finally, for embedded applications, it is not recommended to use the full flavored C++. The Embedded C++ subset describes the consequences of a number of C++ features. As such, one can decide whether or not to use these (depending on memory footprint, performance issues, and timing predictability). We need to make sure that the transformation from Application Model to C/C++ does not produce any violations to the mentioned above.

3.14.1.2 Traceability
Due to the desired compliance to the DO-178B standard, we need to have full traceability of requirements up to source code level. As such, the requirements that have been preserved throughout the preceding model transformations must also be included in the generated C/C++ code. A possible way to achieve this is by adding the requirement tags and descriptions as source comments and by linking configuration control operations (check-in or check-out) to requirement tags.

3.14.1.3 Platform definition
The model transformations are controlled by a set of parameters that specify:
   - The HW configuration (memory map, physical registers, component type/availability). This definition lists the target specifications with respect to memory layout, register definition, component availability or address. Introducing this definition at this very last transformation in the modeling process provides additional flexibility in the exact platform definition parameters, and decreases the interdependence between software and hardware design, and as such the impact and risk of changes.
   - Code generation parameters. Depending on the usage of the generated code, one can enable of disable parts of the model/code. Possible reasons are:
      - profiling code
      - optimizations (memory footprint, performance)
      - robustness settings (system exception handling)
      - distribution parameters over different processes and/or CPU’s

3.14.2 C/C++-based UTF to PV model transformations
Here we assume that the application model is available in TTL C++ and that a hardware and software partitioning of the application model is given. The transformation is called TTL-to-PV refinement and is specified as follows.
   - The input model to the transformation is a process network described in C++ using the TTL C++ software library.
   - The output model of the transformation is a hardware/software partitioned process network. The hardware partition of the process network is described in SystemC using a TLM style to describe the TTL interfaces of hardware modules. The software
partition of the process network is described in C using the TTL C application programmers’ interface to describe TTL interfaces of software modules.

- The design decisions that have to be taken to transform input to output is the binding of software tasks to processors and the binding of variables to memory locations (e.g. the binding of communication channel buffers and administrations to memory locations in order to implement task communication).
- There are no additional design constraints that have to be satisfied in order to obtain feasible outputs since this is a purely syntactic transformation.
- There are no additional design criteria that quantify the quality of feasible since this is a purely syntactic transformation.

To illustrate the transformation from TTL C++ to TTL C we use a sample rate converter (SRC). The SRC has been programmed as a process. In C++ a process is a class. The process has two inputs: one for audio samples and one for the input sampling frequency. The process has one output for audio samples. The input and output ports are member variables of the class. Furthermore, the SRC uses member variables to store, for instance, the incoming samples that are needed to compute the values of the outgoing samples. Each instance of the SRC process gets its own set of member values. Finally, the SRC uses static member variables to store, for instance, coefficient tables. All instances of the SRC process share one such variable. A code snippet is given below.

```c++
01 class SR_Converter : public Process {
02 public:
03   SR_Converter(const Id& n,
04     CbIn<__fixed>& f_inp_vals,
05     CbIn<int>& f_fscode,
06     CbOut<__fixed>& f_src_vals
07   );
08   void main(
09     ...
10     read(ip_inp_vals, buffer, SRC_CIRCULAR_BUFFER_SIZE);
11     ...
12     write(op_src_vals, sum_buffer, OUTPUT_BUFFER_SIZE);
13     ...
14   );
15   ...
16 private:
17   Port< CbIn<__fixed> > ip_inp_vals;
18   Port< CbIn<int> > ip_fscode;
19   Port< CbOut<__fixed> > op_src_vals;
20   ...
21   __fixed buffer[SRC_CIRCULAR_BUFFER_SIZE];
22   __fixed sum_buffer[OUTPUT_BUFFER_SIZE];
23   ...
24   static YROM_DATA __coef __Y interp_H_small[INTERP_H_LENGTH_SMALL];
25   static YROM_DATA __coef __Y interp_H_big[INTERP_H_LENGTH_BIG];
26   ...
27   );
```
In C a TTL task is a struct. The struct contains fields that correspond to the member variables of a process. The member functions become global functions. The static member variables become global variables. The corresponding code snippet for the SRC is given below.

```c
typedef struct {
   ttlCbInPortip_inp_vals;
   ttlCbInPortip_fscode;
   ttlCbOutPortop_src_vals;
   ...
   __fixed buffer[441];
   __fixed sum_buffer[24];
   ...
} TTLCSR_ConverterStruct, *TTLCSR_Converter;
```

The first step in the translation is performed by the gcc preprocessor. In this step substitution of macros takes place. Hence, the macros SRC_CIRCULAR_BUFFER_SIZE, OUTPUT_BUFFER_SIZE, INTERP_H_LENGTH_SMALL, and INTERP_H_LENGTH_BIG are substituted with their corresponding values 441, 24, 24, and 3087. Subsequently, tools process the preprocessed files. They construct the struct TTLCSR_ConverterStruct containing fields for the ports and the old member variables. Furthermore, they construct global functions for all member functions and global variables for all static member variables. The global functions and variables are prefixed with the class name to make them unique. The global functions access the fields of different instances of the SRC struct through a pointer thisHandle that takes over the role of the this pointer in C++. The main function is rewritten such that it accesses the fields of the struct through this pointer. The main function also uses the global functions and global variables. The translation from C++ to C only supports the translation of YAPI and TTL concepts. It does not support a generic translation from C++ to C. We assume that the functionality and data types of the processes are written in C. For instance, the translation does not support the translation of generic C++ classes into C structs for variables that are communicated over channels.

To illustrate the construction of process networks in TTL C we revert back to the producer-consumer example of Section 3.7.3.1. In C, process network construction is performed as shown in the following code examples. First we define the interface of PC in the file ‘PC.h’:
The implementation of PC is given in the file ‘PC.c’:

```c
#include "PC.h"

PC pc;

void PCSetup(int channel_size, int run_length)
{
    ttlChannelSetup(&pc.c1, 0, sizeof(int), channel_size);
    ProducerSetup(&pc.p, 0, &pc.c1, run_length);
    ConsumerSetup(&pc.c, 1, &pc.c1);
}
```

Finally, we create an instance of PC and start it in the file ‘main.c’:

```c
#include "network.h"
#include "PC.h"
#include <stdio.h>

int main()
{
    PCSetup(256, 1000);
    ttlNetworkRun();
    return 0;
}
```

More information on the used transformation tools can be found in documents written by Alders and Popp [8] [9].

The final binding of tasks to processors requires an additional argument in the task setup functions. Details can be found in [10]. The core compilers perform the binding of variables to memory locations.
3.15 Application model to SystemC transformations

3.15.1 C/C++-based UTF to CA model transformations

3.15.1.1 TTL-to-CA Refinement
The purpose of this transformation is to transform the application functionality that must be implemented in hardware from C++ to SystemC.

- The input model to the transformation is a task described in C++ using the TTL C++ software library.
- The output model of the transformation is a hardware coprocessor at the SystemC CA level containing the task functionality and its TTL interfaces.

Next sections explain the integration of a behavioral synthesis tool in our system-level design flow in order to carry out such transformation. Two commercial behavioral synthesis tools [14] and [15] have been evaluated. Section 3.15.1.2 describes the behavioral synthesis source input and the design output. Section 3.15.1.3 gives detailed information about the interface synthesis technology, by explaining a TTL port example. Interface synthesis, is a typical behavioral synthesis term for tool support required to convert Interface Method Call (IMC) interfaces into signal-level interfaces.

3.15.1.2 Behavioral synthesis input and output
The positioning of a behavioral synthesis tool is between the architecture and the implementation level, as depicted in Figure 33. At the architecture level the co-processor is an application model described in TTL C++ with IMC interfaces. This model serves as the functional input model for the behavioral synthesis tool. The output of the tool is a co-processor at RT level with signal-level interfaces.

![Figure 33. Position of behavioral synthesis in the design flow.](image-url)
As shown in Figure 34, mature behavioral synthesis tools support the separation of application model from platform model, which fits perfectly in this methodology. Design exploration can be performed by refinement of the application model and the platform model.

![Figure 34. Behavioral synthesis input and output.](image)

Refinement of the application model could be e.g. rewriting loop structures and adding labels that refer to tool specific optimization pragmas. The refined task is co-simulated with the other tasks of the single functional reference to secure the functional correctness. If during one co-processor task refinement other tasks have to be adapted, then this functional change will be directly reflected to all HW and SW designers, because they all share the single functional reference. This maintains the consistency between the single functional reference and all the implemented blocks, which removes ambiguity and minimizes integration problems.

The platform model in Figure 34 contains tool specific pragmas to optimize the behavioral synthesis results. Optimization examples are e.g.: loop unrolling, pipelining, controlling the amount parallelism in the data path, mapping arrays into ram or into registers, mapping multiple arrays in one array.

The second part of the platform model is the Interface Synthesis port (IS-port). The IS-port converts the IMC interface of the application model, into a signal-level interface for the RT level model (see Figure 34). Both sides of the IS-port have the same TTL interface; only the abstraction level differs. The IS-port is tool specific and TTL specific. The platform design team has to create and support the IS-ports.

### 3.15.1.3 Interface synthesis example

In this section, we present in details a TTL interface at two abstraction levels. We choose as example the TTL RB (Relative Blocking) interface type, which separates the synchronization and data transfer, using blocking semantics for the acquire synchronization operations, and has scalar and vector support. Example code using the RB type can be found in Section 3.7.3.2. Section 3.15.1.3.1 discusses the IMC-level interface, Section 3.15.1.3.2 discusses the corresponding signal-level interface and Section 3.15.1.3.3 explains the IS-port that is required to bridge these two levels.

#### 3.15.1.3.1 RB-in modeled at IMC-level

The RB-in interface has the following IMC primitives:

- `reAcquireData(p, n)`. Re-acquire n full, acquired or unacquired tokens from the channel connected to port `p`.
- `load(p, d, v)`. Copy the value of the token with distance `d` to the oldest acquired token into value `v`.

© Copyright MARTES Consortium
- load(p, d, v, n). Copy the value of the tokens with distance d, ..., d+n-1 to the oldest acquired token into vector v of length n.
- releaseRoom(p, n). Release the n oldest acquired tokens to the channel connected to port p.

Before a consumer task can obtain data (load) from the channel, it first has to acquire the data tokens. Typically the consumer performs multiple loads per acquired data block. The reserved data space is released after the releaseRoom call.

The order of the TTL calls is relevant, e.g. the reAcquireData has to occur before the load. In general a behavioral synthesis tool can change the order of operations to improve the performance results. The tool is not aware of a relation between reAcquireData and load and might change their order, which leads to erroneous behavior. Both tools evaluated are able to keep the order of the operations.

3.15.1.3.2 RB-in modeled at signal-level

At the implementation level an IMC has to be converted into a bit-true and cycle-true protocol.

<table>
<thead>
<tr>
<th>Table 5. RB-in from IMC to signals.</th>
</tr>
</thead>
<tbody>
<tr>
<td>TTL RB-in IMC</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>reAcquireData(p, n)</td>
</tr>
<tr>
<td>load(p, d, v)</td>
</tr>
<tr>
<td>load(p, d, v, n)</td>
</tr>
<tr>
<td>releaseRoom(p, n)</td>
</tr>
</tbody>
</table>

Table 5 shows the mapping of the IMC arguments to the hardware signals. The type signal informs which RB-in IMC is currently active. The tokens signal carries the ‘n’ argument and the distance signal carries the ‘d’ argument. The ‘p’ argument of an IMC is the specific port instance used by the IMC (a task can have multiple RB-in port instances). At the RT level this is managed by wires connected to a specific port instance. The data signal is only used by the load calls. It can carry a single token value (scalar load) or a vector of tokens, one at the time (vector load). The direction (in, out) of the signals is from the task’s perspective.

Figure 35 shows two examples of the clock cycle true protocol. The figure shows two scalar load transactions (type=1, tokens=1). The value in the data waveform refers to the data token of the specific distance.

A valid/accept handshake pair controls the synchronization of the value transfers on the request and data signals, see Figure 35. The valid pulse indicates that the type, tokens, and distance signals contain proper values. The accept pulse indicates that the requested token value is on the data signal. In the example the signals grouped by out (in) are driven by the HW co-processor (HW shell). On the left (right) hand side a fast (slow) response of the HW shell is depicted.
As mentioned above, before a load operation can be performed data has to be available in the channel. Testing the availability of the data is done by the reAcquireData primitive. If data is not available this primitive will block. At the IMC-level, this means that the function call does not return until the requested amount of tokens is in the channel. At the signal level, the accept pulse indicates the data availability. During the period the reAcquireData primitive is blocked, only the interface between the co-processor and the HW shell is blocked; the platform infrastructure (bus) remains available for other devices.

3.15.1.3.3 IS-port

In the abstract TTL models, each TTL port declaration is templated by the token data type (see the upper part of Figure 36).

In case the application model is used as behavioral synthesis input, the port declaration is replaced by the IS-port declaration. The IS-port has two template arguments: a data type used inside the task, and a corresponding type of the RTL data signal, used outside the task. The IS-port performs the type casting if the two types differ. The latter is a SystemC bit restricted data type. The former can be a SystemC type or a simple C data type. The evaluated behavioral synthesis tools are capable of handling IS-ports with multiple template arguments.
As discussed above, the port declaration in the TTL task at architecture level differs from the IS-port declaration at implementation level (see the pi declarations in Figure 36). Still the TTL calls in the model are exactly the same (see the load IMC in Figure 36). The amount of architecture specific details that has to be added in the TTL model is minimized, which eliminates inconsistencies between the reference and its implementation.
4 Conclusion

Within the MARTES methodology we propose model transformations for models that describe applications, execution platforms, allocations, and implementations of embedded systems in different languages, using different modeling styles and at different levels of abstraction. The levels of abstractions cover system-level models, such as requirements models, down to implementation-oriented models, such as cycle accurate models. The languages include UML, C, C++, Java and SystemC.

The purpose of model transformation is to automate design processes as much as possible in order to avoid human errors and to speed up the design process. Furthermore, model transformation enforces that design decisions are made explicit by means of transformation rules. This helps traceability in design processes.

The proposed model transformations do not constitute a complete and automated design process from requirements to implementation. Rather they identify opportunities to support design automation in design processes with human designers in the loop.
## 5 Terms and Definitions

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Comment / Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract Platform</td>
<td>An abstract platform defines an acceptable or, to some extent, ideal platform from an application developer’s point of view; it is an abstraction of infrastructure characteristics assumed for models of an application at some point of (the platform-independent phase of) the design process.</td>
<td>J.P. Almeida, R. Dijkman, M. van Sinderen, L.F. Pires, On the Notion of Abstract Platform in MDA Development</td>
</tr>
<tr>
<td>Horizontal Transformation</td>
<td>A transformation in which both the source and the target artifacts are at the same level of abstraction. It changes the structure of the input specifications, or weaves multiple specifications into a single output specification. In practice this means that the source and target metamodel could but must not be the same.</td>
<td>J. Greenfield, K Short, Software Factories</td>
</tr>
<tr>
<td>Mapping</td>
<td>Specification of a mechanism for transforming the elements of a model conforming to a particular metamodel into elements of another model that conforms to another (possibly the same) metamodel.</td>
<td>OMG MDA guide v1.0.1</td>
</tr>
<tr>
<td>Metadata</td>
<td>Data that represents models. For example, a UML model; a CORBA object model expressed in IDL; and a relational database schema expressed using CWM.</td>
<td>OMG MDA guide v1.0.1</td>
</tr>
<tr>
<td>Metamodel</td>
<td>A model of models.</td>
<td>OMG MDA guide v1.0.1</td>
</tr>
<tr>
<td>Meta-Object Facility (MOF)</td>
<td>An OMG standard, closely related to UML, that enables metadata management and modeling language definition.</td>
<td>OMG MDA guide v1.0.1</td>
</tr>
<tr>
<td>Methodology</td>
<td>The union of methods, notations, guidelines, strategies, processes for a specific application domain.</td>
<td>K.U. Leuven</td>
</tr>
<tr>
<td>Model</td>
<td>A formal specification of the function, structure and/or behavior of an application or system.</td>
<td>OMG MDA guide v1.0.1</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Comment / Source</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------------------------------------</td>
</tr>
<tr>
<td>Model Driven Architecture (MDA)</td>
<td>A specific approach defined by the OMG toward model driven development (MDD) that separates the specification of functionality from the specification of the implementation of that functionality on a specific technology platform. The focus of MDA is primarily the provision of standard specifications for useful technologies in the MDD space. Examples are the UML (modeling), MOF (metamodelling), transformations (QVT), platform specific annotations (set of UML profiles), etc.</td>
<td>OMG MDA guide v1.0.1</td>
</tr>
<tr>
<td>Model Driven Development (MDD)</td>
<td>A software development paradigm that promotes the use of models at different levels of abstraction and transformations between them to derive a concrete application implementation. A specific interpretation of this notion is worked out by OMG's MDA.</td>
<td>K.U.Leuven</td>
</tr>
<tr>
<td>MDD Infrastructure</td>
<td>The collection of metamodels, platforms, transformations, languages, tools, etc., organized in a structured way, that support the development of a specific application or group of related applications (product line).</td>
<td>M. Voelter, J. Bettin, Patterns for Model-Driven Software-Development</td>
</tr>
<tr>
<td>Model transformation</td>
<td>The process of converting one model to another model of the same system.</td>
<td>OMG MDA guide v1.0.1</td>
</tr>
<tr>
<td>Oblique transformation</td>
<td>A combination of a horizontal and a vertical transformation, which can appear from the outside as a single transformation</td>
<td>J. Greenfield, K. Short, Software Factories</td>
</tr>
<tr>
<td>Platform</td>
<td>A set of subsystems/technologies that provide a coherent set of functionality through interfaces and specified usage patterns that any subsystem that depends on the platform can use without concern for the details of how the functionality provided by the platform is implemented.</td>
<td>OMG MDA guide v1.0.1</td>
</tr>
<tr>
<td>Platform Independent model (PIM)</td>
<td>A model of a subsystem that contains no information specific to the platform, or the technology that is used to realize it.</td>
<td>OMG MDA guide v1.0.1</td>
</tr>
<tr>
<td>Platform model</td>
<td>A set of technical concepts, representing the different kinds of parts that make up a platform and the services provided by that platform.</td>
<td>OMG MDA guide v1.0.1</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>CA</td>
<td>Cycle Accurate</td>
</tr>
<tr>
<td>PIM</td>
<td>Platform Independent Model</td>
</tr>
<tr>
<td>PSM</td>
<td>Platform Specific Model</td>
</tr>
<tr>
<td>PV</td>
<td>Programmer View</td>
</tr>
<tr>
<td>PVT</td>
<td>Programmer View with Time</td>
</tr>
<tr>
<td>TF</td>
<td>Timed Functional</td>
</tr>
<tr>
<td>TLM</td>
<td>Transaction Level Modeling</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>UTF</td>
<td>Un-Timed Functional</td>
</tr>
</tbody>
</table>
7 References

8 Appendix A

8.1 Producer-Consumer example in YAPI C++

8.1.1 pc.h

```cpp
#include "yapi.h"
#include "producer.h"
#include "consumer.h"

class PC : public ProcessNetwork {
    PC(Id n, int length);
    const char* type() const;
private:
    Fifo<int> fifo;
    Producer p;
    Consumer c;
};
```

8.1.2 pc.cc

```cpp
#include "pc.h"

PC::PC(Id n, int length) : ProcessNetwork(n),
    fifo( id("fifo")),
    p( id("p"), fifo, length),
    c( id("c"), fifo) {
}

const char* PC::type() const {
    return "PC";
}

int main(int argc, char *argv[]) {
    RTE rte;
    PC pc(id("pc"), 1000);
    rte.start(pc);
    printWorkload(pc);
    printDotty(pc);
    return 0;
}
```
8.1.3 producer.h

```c
#include "yapi.h"

class Producer : public Process
{
public:
    Producer(Id n, Out<int>& o, int length);

    const char* type() const;
    void main();

private:
    OutPort<int> p;
    int n;
};
```

8.1.4 producer.cc

```c
#include "producer.h"

Producer::Producer(Id n, Out<int>& o, int length) :
    Process(n),
    p( id("p"), o),
    n(length)
{ }

const char* Producer::type() const
{
    return "Producer";
}

void Producer::main()
{
    write(p, n);
    for (int i=0; i<n; i++)
    {
        write(p, i);
    }
}
8.1.5 consumer.h

01 #ifndef CONSUMER_H
02 #define CONSUMER_H
03
04 #include "yapi.h"
05
06 class Consumer : public Process
07 {
08     public:
09         Consumer(Id n, In<int>& i);
10         const char* type() const;
11         void main();
12
13     private:
14         InPort<int> p;
15     };
16 }
17
18 #endif

8.1.6 consumer.cc

01 #include "consumer.h"
02 #include <iostream>
03
04 using namespace std;
05
06 Consumer::Consumer(Id n, In<int>& i) :
07     Process(n),
08     p( id("in"), i)
09 { }
10
11 const char* Consumer::type() const
12 {  
13     return "Consumer";
14 }
15
16 void Consumer::main()
17 {
18     int n;
19     read(p, n);
20     for (int i=0; i<n; i++)
21     {
22         int j;
23         read(p, j);
24         assert(i==j);
25         printf("Value i=%d, j=%d\n", i, j);
26     }
27 }
8.2 Producer-Consumer example in TTL C++

8.2.1 PC.h

The first eight lines with comments are omitted.

```cpp
09 #ifndef PC_INCLUDED
10 #define PC_INCLUDED
11
12 #include "channel.h"
13 #include "network.h"
14 #include "ttl_use_namespace.h"
15
16 #include "ttlTypes.h"
17 #include "cb_channel.h"
18 #include "rb_channel.h"
19 #include "rn_channel.h"
20 #include "dbi_channel.h"
21 #include "dbo_channel.h"
22 #include "dno_channel.h"
23 #include "dni_channel.h"
24 #include "cb_in_port.h"
25 #include "cb_out_port.h"
26
27 #include "Consumer.h"
28 #include "Producer.h"
29
30 class PC: public Network
31 {
32    public:
33        // constructor
34        PC
35           (Id n,
36              int length);
37
38        // extra public member function declarations
39        const char* type() const;
40
41    private:
42        // input ports
43        int length_param;
44
45        // channels
46        Channel<int> a;
47
48        // processes and networks
49        Consumer i0_Consumer;
50        Producer i0_Producer;
51    }
52 #endif // PC_INCLUDED
```
8.2.2 PC.cc

```c++
#include "PC.h"

PC::PC (Id n, int length) :
    Network(n),
    // ports
    length_param(length),
    // channels
    a(id("a"), 256),
    // processes and networks
    i0_Consumer
    (id("i0_Consumer"),
        a
    ),
    i0_Producer
    (id("i0_Producer"),
        length_param,
        a
    )
{} 
```

8.2.3 PCMain.cc

```c++
#include "PC.h"

const char* PC::type() const
{
    return "PC";
}

int main(int argc, char* argv[])
{
    PC pc(id("pc"), 1000);
    run(pc);
    printf("%s", "The end.");
    return 0;
}
```
8.2.4 Producer.h

```cpp
09 #ifndef Producer_INCLUDED
10 #define Producer_INCLUDED
12
13 // pre class items
14 //using namespace ttl;
15
16 #include "cb_in_port.h"
17 #include "cb_out_port.h"
18 #include "process.h"
19 #include "ttl_use_namespace.h"
20
21 #include "ttlTypes.h"
22 #include "cb_out_port.h"
23
24
25 class Producer: public Process
26 {
27   public:
28     // constructor
29     Producer
30        (Id n,
31         int length,
32         CbOut<int>& o);
33
34     // extra public member function declarations
35     const char* type() const;
36     void main();
37
38   private:
39     // input ports
40     int n;
41
42     // output ports
43     Port< CbOut<int> > p;
44 };
45
46 #endif // Producer_INCLUDED
```

8.2.5 Producer.cc

```cpp
09 #include "Producer.h"
10
11 Producer::Producer
12    (Id n,
13     int length,
14     CbOut<int>& o)
15 :
16   Process(n),
17   // ports
18   n(length),
19   p(id("p"), o)
20 {}
8.2.6 ProducerMain.cc

09 #include "Producer.h"
10
11 const char* Producer::type() const 
12 { 
13    return "Producer";
14 }
15 void Producer::main()
16 {
17    write(p, n);
18    for (int i = 0; i < n; i++)
19    {
20       write(p, i);
21    }
22 }

8.2.7 Consumer.h

09 #ifndef Consumer_INCLUDED
10 #define Consumer_INCLUDED
11
12 // pre class items
13 //using namespace ttl;
14
16 #include "cb_in_port.h"
17 #include "cb_out_port.h"
18 #include "process.h"
19 #include "ttl_use_namespace.h"
20 #include "ttlTypes.h"
21 #include "cb_in_port.h"
22
25 class Consumer: public Process 
26 { 
27     public:
28         // constructor 
29         Consumer 
30         (Id n, 
31             CbIn<int>& i);
32 
33         // extra public member function declarations 
34         const char* type() const;
35         void main();
36 
37     private:
38         // input ports
39         Port< CbIn<int> > p;
40 };
41
43 #endif // Consumer_INCLUDED
8.2.8 Consumer.cc

```cpp
#include <assert.h>
#include <iostream>

#include "Consumer.h"

Consumer::Consumer (Id n, CbIn<int>& i):
    Process(n), // ports
    p(id("p"), i)
{}
```

8.2.9 ConsumerMain.cc

```cpp
#include <assert.h>
#include <iostream>
#include "Consumer.h"

using namespace std;
const char* Consumer::type() const
{
    return "Consumer";
}

void Consumer::main()
{
    int n;
    read(p, n);
    for (int i = 0; i < n; i++)
    {
        int j;
        read(p, j);
        assert(i == j);
        printf("Value i=%d, j=%d\n", i, j);
    }
}
```
9 Appendix B

9.1 CoFluent meta-models

9.1.1 Application meta-model/timed functional

CoFluent application models are described at the TF (Timed Functional) abstraction level (per MARTES definitions) as a network of concurrent processes called functions communicating through functional relations:

- Communicating functions follow a timed behavior describing sequence, conditional, iterative or parallel executions of operations (= sequential algorithm/data processing) and inputs/outputs (asynchronous & synchronous communications) characterized by their durations.
- Functional relations (inter-function communications) are characterized by read/write durations and can be of different types: message queue (time dependency + data exchange), event (time dependency only), shared variable (data exchange only).

<table>
<thead>
<tr>
<th>CoFluent Application Meta-Model</th>
<th>CoFluent Application Example Model</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Diagram" /></td>
<td><img src="image" alt="Diagram" /></td>
</tr>
</tbody>
</table>

9.1.2 Execution platform model/timed functional

CoFluent execution platform models are described at the TF (Timed-Functional) abstraction level (per MARTES definitions), more precisely at a message-passing transactional level, as processors and memories communicating through nodes:

- Communicating processors & shared memories are characterized by performance properties.
- Physical links (inter-processor communications) or communication nodes are characterized by send/receive durations and performance properties and can be of different types: bus, router, point-to-point.
9.1.3 Allocation model/timed functional

CoFluent allocation models are described at the TF (Timed Functional) abstraction level (per MARTES definitions):

- Computations: processors execute functions.
- Communications: communication nodes route/transport message queues, events and shared variables; shared memories store shared variables.
9.1.4 Allocated model/yimed Functional

CoFluent allocated models are described at the TF (Timed Functional) abstraction level (per MARTES definition), more precisely at a message-passing transactional level:

- Computations: functions executed on software processor are software tasks scheduled by an abstract RTOS characterized by scheduling durations; functions executed on hardware processor are hardware modules running in parallel.
- Communications: interfaces provide adaptation layers (drivers, protocol stacks, glue logic, etc.) between intra-processor and inter-processor communications on physical links (nodes) and are characterized by various access and execution durations.

![CoFluent allocated model example](image)

9.1.5 Implementation model/timed functional

CoFluent implementation models are described at the TF (Timed Functional) abstraction level (per MARTES definitions), more precisely at a message-passing transactional level, in C/C++. Then, the CoFluent implementation meta-model is simply the C/C++ syntactic and semantic rules defined in the ANSI standard.

9.1.6 SystemC model/timed functional

CoFluent SystemC models are described at the TF (Timed Functional) abstraction level (per MARTES definitions), more precisely at a message-passing transactional level, as a set of C++ classes and functions based on the CoFluent SystemC simulation library, itself based on the SystemC 2.0.1 API defined in the OSCI standard.
9.2 CoFluent models transformations

9.2.1 Application model (to implementation model) to SystemC

This transformation is called SystemC application generation within CoFluent Studio and consists in generating a C++ representation of the application model based on classes defined in the CoFluent Studio SystemC simulation library. Each structural and behavioral element of the model is translated into its corresponding C++ class that inherits from generic (meta-model level) classes, which themselves based on the SystemC API.

9.2.2 Application-platform-allocation model to allocated model

This transformation is called architecture mapping within CoFluent Studio and is achieved in a first step by interactively dragging and dropping functions of the application model to processors of the platform model and selecting routing options for functional relations to communication nodes and shared memories. Mapping results define the allocation model, which serves as a basis for the generation of the allocated model in a second step.
9.2.3 Allocated model (to implementation model) to SystemC

This transformation is called SystemC architecture generation within CoFluent Studio and consists in generating a C++ representation of the allocated model based on classes defined in the CoFluent Studio SystemC simulation library. Each structural, behavioral and physical element of the model is translated into its corresponding C++ class that inherits from generic (meta-model level) classes, which themselves based on the SystemC API.