

**SINTEF Telecom and Informatics**

Address: N-7465 Trondheim
NORWAY

Location Trondheim:
O.S. Bragstads plass, Gløshaugen

Location Oslo:

Forskningsveien 1

Telephone: +47 73 59 30 00

Fax: +47 73 59 43 02

Enterprise No.: NO 948 007 029 MVA

SINTEF REPORT

TITLE

Using UML for architectural design of SDL systems.

AUTHOR(S)

Jacqueline Floch

CLIENT(S)

SINTEF Telecom and Informatics

REPORT NO. STF40 A00009	CLASSIFICATION open	CLIENTS REF.	
CLASS. THIS PAGE open	ISBN 82-14-01927-3	PROJECT NO. 402856.03	NO. OF PAGES/APPENDICES 33
ELECTRONIC FILE CODE UML_ArchDesign.doc		PROJECT MANAGER (NAME, SIGN.) Richard Sanders	CHECKED BY (NAME, SIGN.) Richard Sanders
FILE CODE	DATE 2000-02-01	APPROVED BY (NAME, POSITION, SIGN.) Eldfrid Ø. Øvstedal	

ABSTRACT

Describing the architectural design (i.e. the mapping from a logical functional model to a concrete system) is a general problem in software system engineering. While notations have been defined to describe abstract models and implementations, little work has been done in order to define a notation for architectural design. Solutions are proposed in UML for describing the system software structure and the deployment of software components on a hardware platform. We believe that these UML solutions are not yet mature and that they are incomplete. In our work, we have clarified and extended them in order to model the architectural design.

Our starting points are the SISU methodology and the SOON notation, and the architecture design in TIME. Our work has thus focused on the transition from an *SDL specification* to a concrete system. However, the solutions that we propose are general and may be applied when using other notations for modelling the abstract system, e.g. UML itself.

In this document, we first introduce the architectural design concepts and present the SOON notation. Then we describe the implementation concepts defined in UML and present how these concepts can, together with the UML extension mechanisms, be used for architectural design. We discuss some shortcomings in UML, and also comment on achievements in the Telelogic Tau tools.

KEYWORDS	ENGLISH	NORWEGIAN
GROUP 1	Computer science	Datateknikk
GROUP 2	System model	Systemmodell
SELECTED BY AUTHOR	Architectural design	Arkitekturdesign
	UML	UML
	SDL	SDL

TABLE OF CONTENTS

1	INTRODUCTION	4
2	ARCHITECTURAL DESIGN	5
3	SOON.....	7
3.1	HARDWARE STRUCTURE	7
3.2	SOFTWARE STRUCTURE.....	9
3.3	DISCUSSION.....	10
4	IMPLEMENTATION CONCEPTS AND EXTENSIONS MECHANISMS IN UML.....	11
4.1	IMPLEMENTATION CONCEPTS IN UML.....	11
4.1.1	<i>Components.....</i>	<i>11</i>
4.1.2	<i>Component Diagram.....</i>	<i>11</i>
4.1.3	<i>Node</i>	<i>13</i>
4.1.4	<i>Deployment Diagram.....</i>	<i>13</i>
4.2	EXTENSION MECHANISMS.....	15
4.2.1	<i>Tagged values.....</i>	<i>15</i>
4.2.2	<i>Constraints</i>	<i>17</i>
4.2.3	<i>Stereotypes</i>	<i>18</i>
5	GUIDELINES FOR USING UML FOR ARCHITECTURAL DESIGN.....	24
5.1	MAPPING SOON-UML	24
5.1.1	<i>Overall structure of the hardware system</i>	<i>24</i>
5.1.2	<i>Overall structure of the software system.....</i>	<i>26</i>
5.1.3	<i>Combined software/hardware diagrams</i>	<i>27</i>
5.1.4	<i>Reference to SDL entities</i>	<i>27</i>
5.2	CODE GENERATION INFORMATION.....	27
5.2.1	<i>Design information related to the SDL application</i>	<i>27</i>
5.2.2	<i>Design information related to the execution target.....</i>	<i>27</i>
5.2.3	<i>Design information independent of any model.....</i>	<i>28</i>
5.2.4	<i>Design information related to the system simulation</i>	<i>29</i>
5.3	DYNAMIC SYSTEM RECONFIGURATION	29
6	TELELOGIC DEPLOYMENT EDITOR.....	30
6.1	NODES AND COMPONENTS	31
6.2	THREADS	31
6.3	OBJECTS	31
6.4	ASSOCIATION	31
6.5	COMPOSITE AGGREGATION	32
6.6	CONCLUSION	32
7	REFERENCES	33

LIST OF FIGURES

FIGURE 1: SOON HARDWARE STRUCTURE SYMBOLS	7
FIGURE 2: AN EXAMPLE OF AN OVERALL HARDWARE STRUCTURE IN SOON.....	8
FIGURE 3: SOON SOFTWARE STRUCTURE SYMBOLS.....	9
FIGURE 4: AN EXAMPLE OF AN OVERALL SOFTWARE STRUCTURE IN SOON.....	10
FIGURE 5: AN EXAMPLE OF A COMPONENT DIAGRAM IN UML	12
FIGURE 6: AN EXAMPLE OF A DEPLOYMENT DIAGRAM IN UML.....	14
FIGURE 7: AN EXAMPLE OF A HARDWARE DIAGRAM TYPE IN UML	14
FIGURE 8: USING AN EXTENSION OF UML FOR HARDWARE/SOFTWARE DIAGRAM	15
FIGURE 9: AN EXAMPLE SHOWING THE USE OF IMPLEMENTATION REFERENCE (UML EXTENSION)	15
FIGURE 10: USING ATTRIBUTES FOR REPRESENTING IMPLEMENTATION INFORMATION IN A TYPE DIAGRAM (UML EXTENSION).....	17
FIGURE 11: USING ATTRIBUTES FOR REPRESENTING IMPLEMENTATION INFORMATION IN AN INSTANCE DIAGRAM.....	17
FIGURE 12: USING TAGGED VALUES IN AN IMPLEMENTATION DIAGRAM.....	17
FIGURE 13: USING CONSTRAINTS IN AN IMPLEMENTATION DIAGRAM	18
FIGURE 14: STEREOTYPE DEFINITION.....	19
FIGURE 15: USING STEREOTYPES	20
FIGURE 16: USING STEREOTYPES FOR A LIST OF ELEMENTS	20
FIGURE 17: USING THE STEREOTYPES <<PROCESS>> AND <<THREAD>>	21
FIGURE 18: USING THE STEREOTYPE <<FILE>>	21
FIGURE 19: USING THE STEREOTYPE <<BECOME>> FOR SHOWING MIGRATION OF OBJECTS.....	22
FIGURE 20: USING SEQUENCE DIAGRAMS WITH COMPONENT INSTANCES.	22
FIGURE 21: SHOWING CREATION OF COMPONENTS IN DEPLOYMENT DIAGRAMS	23
FIGURE 22: SHOWING CALL BETWEEN COMPONENTS IN DEPLOYMENT DIAGRAMS.....	23
FIGURE 23: MAPPING FROM SOON TO UML (HARDWARE SYMBOLS).....	24
FIGURE 24: MAPPING FROM SOON TO UML (SOFTWARE SYMBOLS).....	26
FIGURE 25: INTERFACE TO THE ENVIRONMENT.....	28
FIGURE 26: DESCRIPTION OF AN INTERFACE TO THE ENVIRONMENT.	28
FIGURE 27: INFORMATION ABOUT THE REALISATION OF AN INTERFACE TO THE ENVIRONMENT	28
FIGURE 28: SYSTEM BUILDING INFORMATION USING <<DERIVE>> AND TAGGED VALUES.....	28
FIGURE 29: SYSTEM BUILDING INFORMATION USING A SUBTYPE STEREOTYPE OF <<DERIVE>>	29
FIGURE 30: COLLABORATION DIAGRAM WITH NODE AND COMPONENT INSTANCES	29
FIGURE 31: ENTITIES DEFINED IN THE DP EDITOR.	30

1 Introduction

The purpose of this activity is to investigate how UML can be used to model:

- the mapping from a logical functional model to a concrete system, and
- the system properties that are not represented in the logical model or the so-called "non-functional" properties.

This is called the architectural design.

Describing the architectural design is a general problem in software system engineering. While notations¹ have been defined to describe abstract models and implementations, little work has been done to define a notation for architectural design. SOON² ([3]) has been proposed for the representation of the hardware and software structures and the mappings from the SDL system to the concrete system. Although SOON satisfies our needs for architectural design description, SOON has not been widely used. There exists no tool that supports the notation. Solutions are also proposed in UML [5] for describing the system software structure and the deployment of software components on a hardware platform. We believe that the UML solutions are not yet mature and that they are incomplete. In our work, we have clarified and extended them in order to model the architectural design.

Our starting points are the SISU methodology and the SOON notation [3], and the architecture design in TIME [6]. Our work will thus focus on the transition from an *SDL specification* to a concrete system. However, the solutions that we propose are general and may be applied when using other notations for modelling the abstract system, e.g. UML itself.

In this document, we first introduce the architectural design concepts and present the SOON notation. Then we describe the implementation concepts defined in UML and present how these concepts can, together with the UML extension mechanisms, be used for architectural design.

Our work is also related to Z.109 [7]. The Z.109 recommendation focuses on the transition from domain analysis with UML to application design with SDL. UML stereotypes have been defined to represent SDL entities in UML. We have looked at how the work done in Z.109 can be taken into account, so that the transition from SDL to UML can be unified with the transition from UML to SDL as described in Z.109.

The Telelogic Deployment Editor (Telelogic Tau SDL Suite 3.6) can be used to describe how an SDL system will be implemented in a run-time environment. We have studied this new tool and found out that its capabilities are very restricted with respect to the approach of the SISU methodology and TIME.

¹ For example, SDL or UML may be used to describe the logical functional models. C++ or Java may be used for implementations.

² SISU object oriented notation

2 Architectural design

There exist several differences between SDL systems and real systems. While an SDL system consists in logical active entities and logical channels, a real system is composed of physical computers, software processes, buses, communication networks. [3] distinguishes between:

- *fundamental* differences that lie in the nature of the components and their "imperfections". They encompass processing time, errors developed by physical components and noise, physical distribution and limitation of resources.
- *conceptual* differences that lie in the way components function. They encompass concurrency, communication modes (e.g. stream vs. message), synchronisation, data abstractions.

The purpose of the architectural design³ is to describe the decisions done in order to produce the implementation (concrete or real system) from the SDL system and document how the gap or differences between the SDL system and the real system are handled. The architectural design description describes the concrete system and relates it to the abstract SDL system. The architectural design is an abstraction of a concrete system representing [6]:

- the overall structure of the hardware identifying at least all physical nodes and interconnections needed to implement the abstract system,
- the overall structure of the software identifying at least all software nodes, software communications and relations needed to implement the abstract system (in terms of processes, procedures, and data). Additional aspects include process priorities, queues and semaphores, interrupt handlers, timers, watchdogs, etc.

Although [3] considers physical distribution, it deals on distribution in general, but does not take into account middlewares for distributed processing environments (DPEs) and the differences between the SDL abstraction and the concepts introduced in DPEs. For example, DPEs may provide support for dynamic system reconfiguration (e.g. migration). As DPEs are becoming very popular, and are expected to become a main technology for communication networks (e.g. active networks or network management using distributed mobile agents), it should be possible to describe the implementation of an SDL system on a DPE platform.

The architectural design description is an important document for the system developers and system users, and also possibly for tools. If we model formally the architectural design aspects, the description can be, for example, interpreted by a code generator tool. In our work on automatic code generation from SDL, we give detailed examples of design information elements that may be relevant during code generation. [4] classifies this information as follows:

- Design information related to the SDL application. This includes signal priorities, "SDL timer units", variable properties (e.g. static vs. dynamic allocation), optimization (e.g. memory usage, performance of execution), response time requirements, security requirements, type definition expansion (e.g. inheritance may be expanded).
- Design information related to the execution target. This includes distribution of the software into files, software process (e.g. Unix processes), target programming code (e.g. C++, java), external code import, interface with the environment.
- Design information independent of any model. This includes the identification of the SDL code generator that is used, and other tools used to produce the system implementation (e.g. compilers).
- Design information related to the system simulation. This includes description of noise (e.g. loss of signals, routing errors, and data corruption in signals).

³ Also called Implementation Design in [1].

The classification and examples proposed in [4] complement the description in [3]. [4] focuses on detailed properties rather than overall HW and SW architecture. Both documents and their results are relevant for the work presented in this document.

3 SOON

In [3], SOON (SISU object oriented notation) is proposed for the representation of the hardware and software structures and the mappings from the SDL system to the concrete system. Our approach based on UML should at least enable us to describe similar elements. We present here a summary of SOON for implementation design and some examples. More detailed information can be found in [3]. Later in this document, we will try to represent the same information and examples using UML.

3.1 Hardware Structure

Figure 1 presents the different SOON hardware symbols and their meanings. The symbols are used in hardware diagrams that describe the overall structure of the hardware system and the mapping from SDL to the real system.

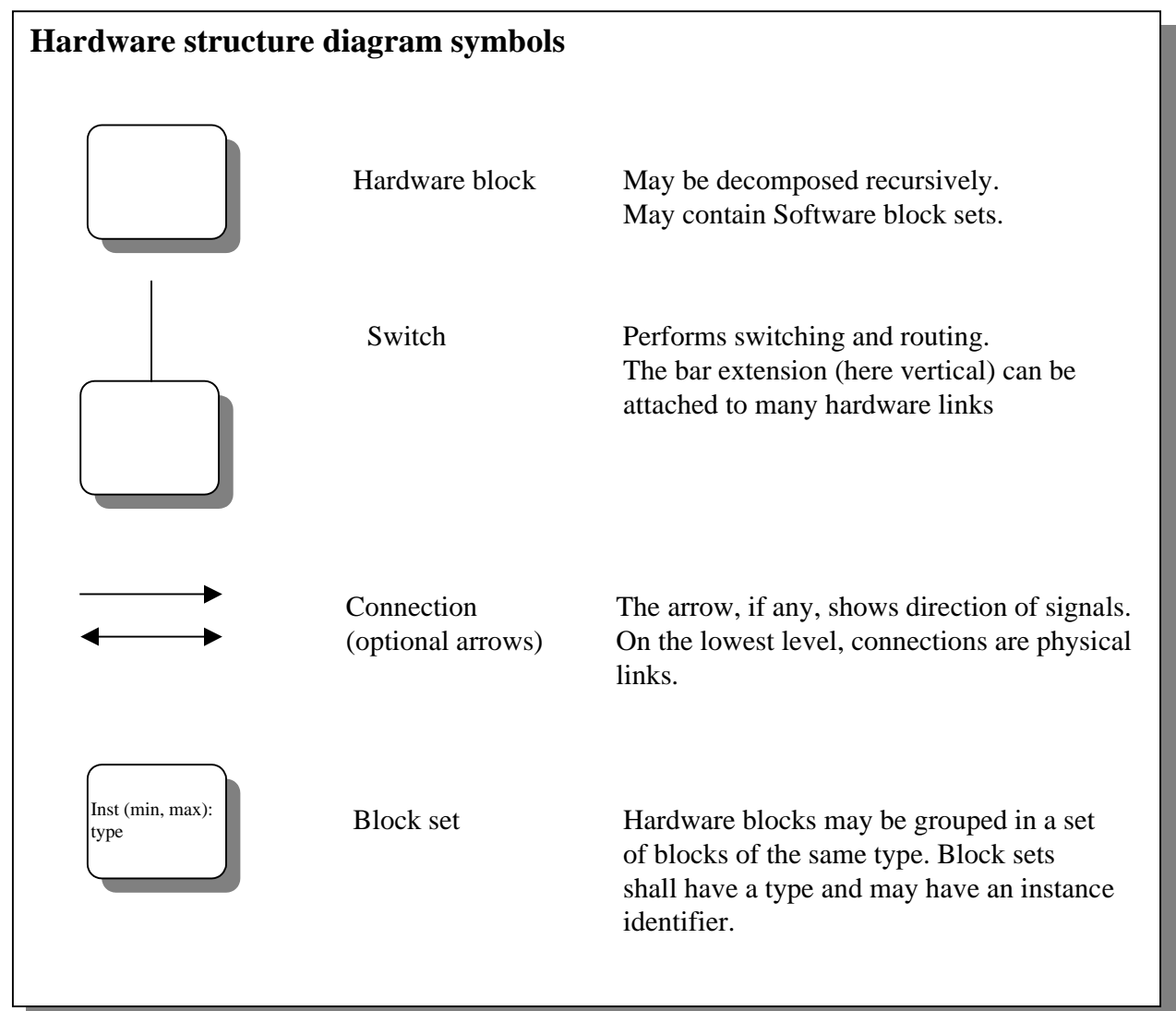


Figure 1: SOON hardware structure symbols

In order to save space in the hardware diagrams, it is possible to refer to an SDL entity. This is done using a reference symbol:

<i> = reference text

Different types of references are defined:

- implement : the referred SDL entity is *realised* by the hardware entity.
- execute : the referred software entity is *loaded* on the hardware entity.

Note that in the first reference type indicates a difference of abstraction between the entities related by the reference relation.

Figure 2 presents an example of the overall structure of the hardware of the system "Access Control". In this diagram, references to SDL entities are made using the "implement" reference relation.

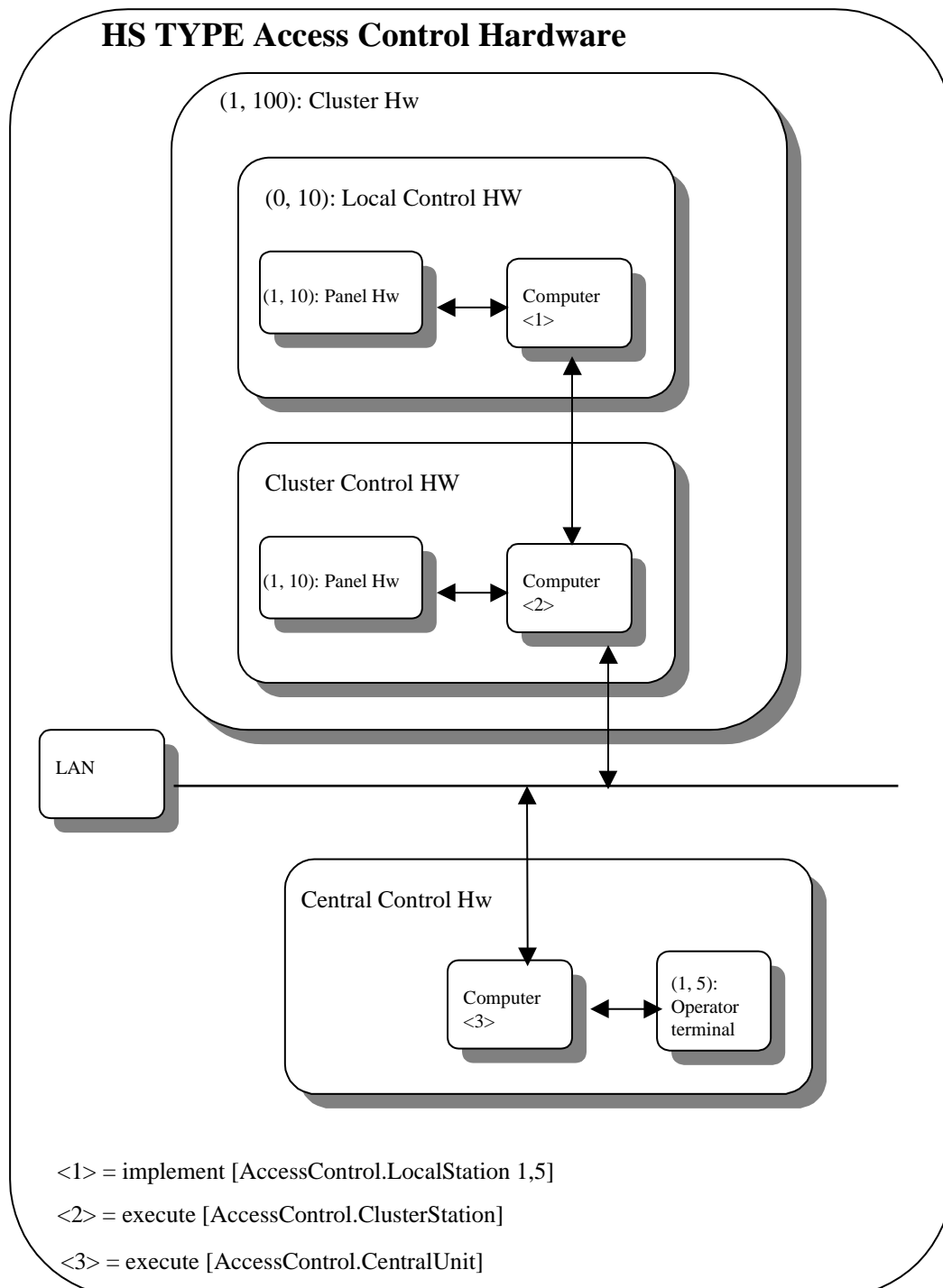


Figure 2: An example of an overall hardware structure in SOON

3.2 Software Structure

Figure 3 presents the different software symbols and their meanings. The symbols are used in software diagrams that describe the overall structure of the software system and the mapping from SDL to the software system. They are also used in combined hardware/software diagrams.

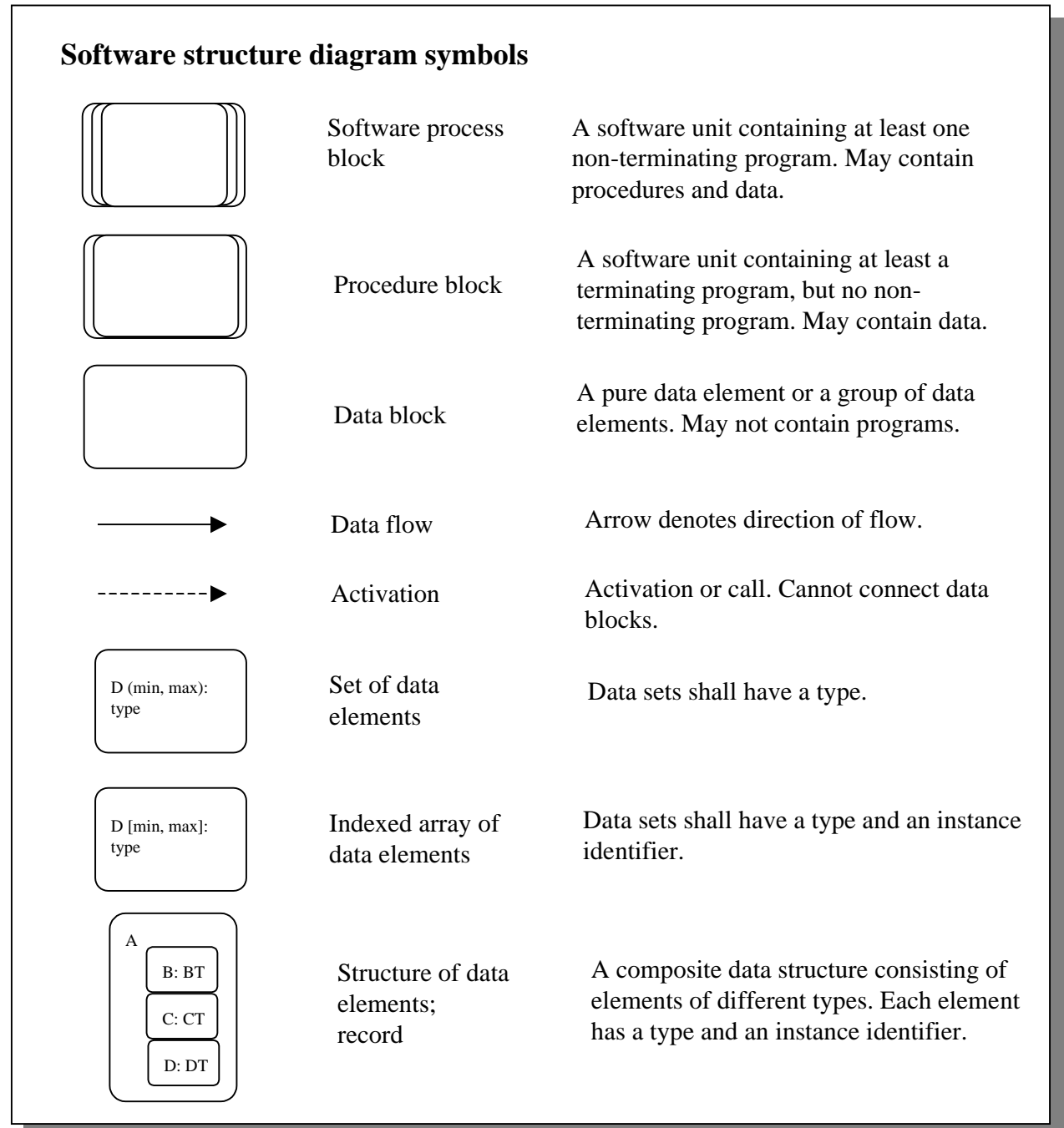


Figure 3: SOON software structure symbols

As in hardware diagrams, it is possible to refer to entities from other diagrams, either SDL diagrams or hardware diagrams. The following references are defined:

implement	:	the referred SDL entity is <i>realised</i> by the software entity.
implemented_by	:	the software entity or SDL entity is realised by the specified referred entity.
executed_by	:	the software entity is loaded on the specified referred entity.

Figure 4 presents an example of the overall software structure for the local unit in the system "Access Control". The diagram also shows the interface with the input-output hardware units.

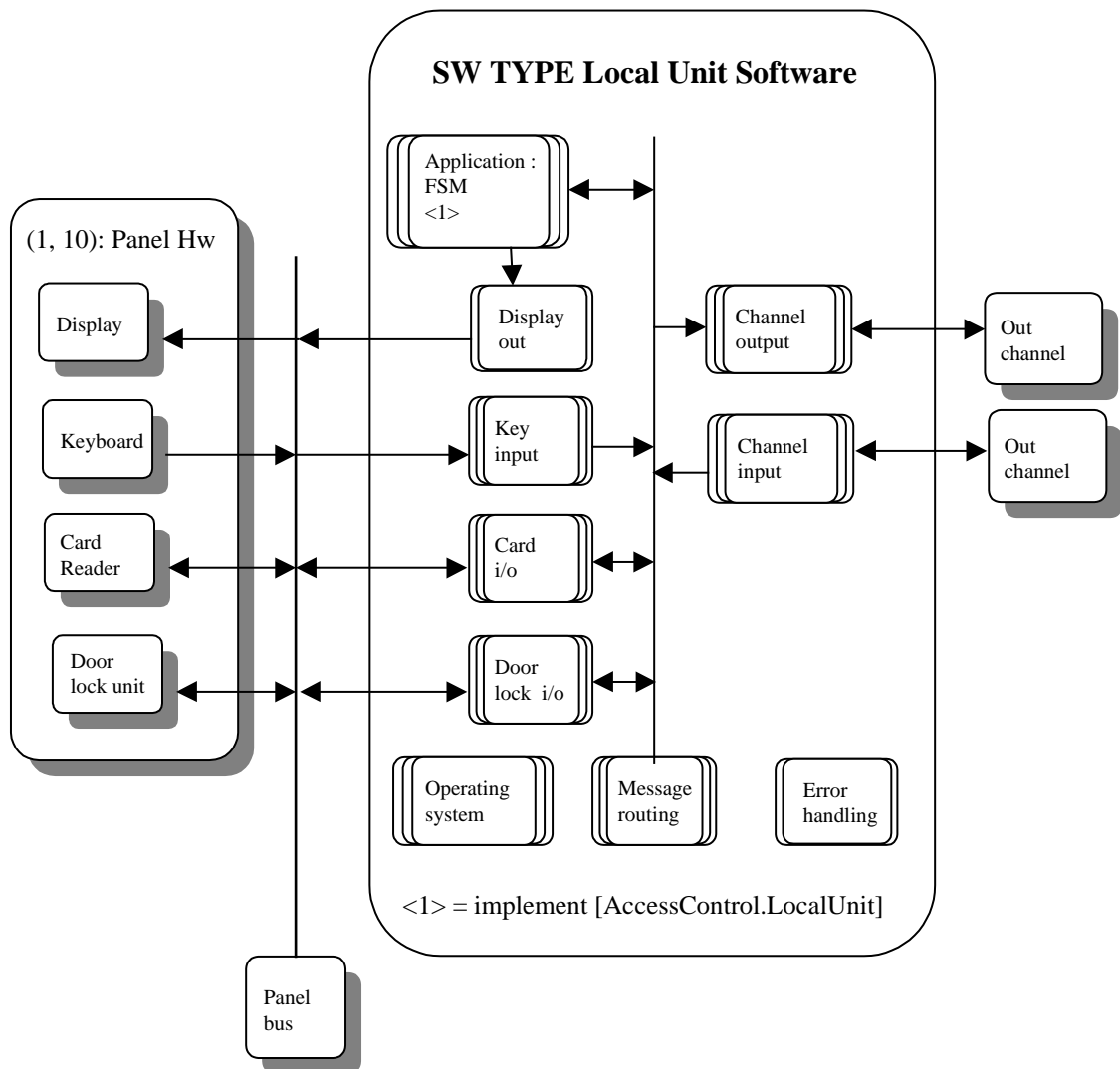


Figure 4: An example of an overall software structure in SOON

3.3 Discussion

From this short introduction to SOON, we observe that the notation focuses on the overall hardware and software structures of the concrete system. In addition to this structural information we also wish to describe other non-functional properties of the system, for example information that may be relevant for the code generator (see section 2 on page 5).

4 Implementation concepts and extensions mechanisms in UML

UML [5] is a general-purpose visual modelling language that can be used to specify and document almost any property of a system. In addition to the basic UML elements, UML introduces extension mechanisms that enable one to customise and extend model elements. These extension mechanisms give UML an (almost) unlimited power of expression, however with the main drawback of reducing the universality of the models. An extensive use of these mechanisms may lead to models that are not portable among tools, and that are difficult to understand even by UML experts.

In this section, we investigate how the basic UML elements and the extension mechanisms can be used in order to represent the architectural design information. We believe that SDL users should avoid developing their own UML extensions, but rather agree on common extensions for architectural design. This section proposes an approach for a common solution.

Before presenting a complete approach, we first discuss the UML concepts that we will use. A *main drawback of UML is the lack of formality of the language specification*. Some concepts in the specification (especially the concepts related to the implementation diagrams) are defined in a vague manner. Some concepts are not completely defined in the UML Semantics (as they should), but rather mentioned in the UML Notation Guide.

4.1 Implementation concepts in UML

UML defines components and nodes that represent software and hardware units respectively. Two types of diagrams, the component diagram and the deployment diagram are used to show the overall software structure and the run-time environment. Note that *UML focuses on deployment, meaning a configuration of run-time systems, rather than on overall hardware structure*. UML sets restrictions on the diagrams with respect to types and instances; these restrictions make it difficult to model hardware/software types as it is done in SOON.

4.1.1 Components

A component represents a physical piece of implementation of a system, including code (source, binary, executable) or equivalent items such as scripts or files. A component may conform to and provide a set of interfaces. Thus a component is a piece of software or a code file. We will use components in order to represent software processes, procedures or data blocks.

A Component is a child of Classifier (UML metamodel). The UML specification *explicitly* explains that a Component, as Classifier, may have its own features such as attributes and operations, and realise Interface. The UML specification does not mention that relationships (described using Association) may be defined between a component and other classifiers. According to the metamodel and the UML semantics, this is possible. Thus we may, for example, specify that a component is composed of other components or that a component realise classes.

4.1.2 Component Diagram

The component diagram is defined in the UML Notation Guide. The purpose of this diagram is to show the dependencies among software components. The Notation Guide does not explicitly state that the component diagram can be used to describe composite components or other relations among components. On another hand, when presenting the deployment diagram (see section 4.1.4 below), the UML specification gives examples where some component instances consist of objects (or class instances). We believe that composition can also be represented in a component diagram (in a type form - eventually graphically nested form - showing components that consist of other components or classes).

A UML component diagram has only a type form, not an instance form. To show component instances, a deployment diagram is used. The deployment diagram may possibly be a degenerate⁴ deployment diagram i.e. one without nodes. We wonder why the UML specification has introduced such bizarre rules. *Why does not UML align the concepts of the implementation diagrams with those of the class and object diagrams* (i.e. allowing type and instance diagrams for components)?

Figure 5 shows how components and component diagrams can be used to model the overall software structure of Local Unit. This example will be refined later in the document in order to match the SOON example of section 3. Here, we have focused on composition: the component "Local Unit Software" is composed of the components "Application", "Display", etc. Composition is a form of aggregation with strong ownership and coincident lifetime of part with the whole (section "3.47 Composition" in the UML Notation Guide). When lifetimes do not coincide, open aggregation should be used instead.

Note that several notations are defined in UML in order to represent composition (i.e. attributes, associations with filled diamonds, nested elements). Any of them could be used. We have chosen the nested graphical representation that it is close to SOON. However, *we are aware that most UML tools do not support the nested representation*.

When further details about a component have to be specified, it may be wise to split the information to several diagrams, and possibly to use the attribute notation. The example does not show any dependencies between the components. The UML specification and the UML textbooks give many examples that illustrate dependencies.

The nested elements may have a role name within the composition (e.g. "os", "err"). The multiplicity is shown in the upper right corner of the nested component symbol. Here it is "1" for each nested component. A range (i.e. min, max) or a set of values may also be specified. In section "3.43 Multiplicity" of the UML specification [5], the meaning of an unspecified multiplicity is not explicitly described. However section "3.47 Composition" explains that: when omitted, the default multiplicity is many (many may also be represented using "*").

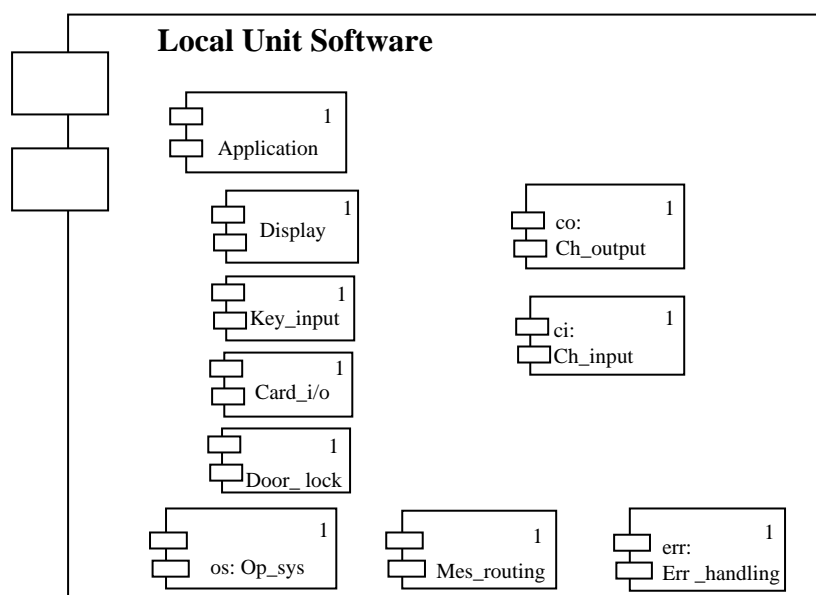


Figure 5: An example of a component diagram in UML

⁴ "degenerate" is the term used by UML specification in section "3.94 Component Diagram".

In this figure, we do not distinguish between different types of software elements as SOON does. We will add this information later.

4.1.3 Node

A node represents a run-time computational resource, generally having at least memory and often processing capability, where components can be deployed. Thus a node is a piece of hardware. We will use nodes in order to represent hardware blocks.

A `Node` is a child of `Classifier` (UML metamodel). However as distinct from the presentation of a `Component`, the UML specification does not explicitly present that a `Node` may, as `Classifier`, have its own features such as attributes and operations, and realise `Interfaces`. According to the metamodel and the UML semantics, this is allowed. It should also be possible to define relationships (using `Association`) between a node and other classifiers. *The question is: which diagrams should be used to represent the properties of node types?* (see section 4.1.4).

Although the examples given in the UML Notation Guide only illustrates node instances, the notation guide explains that a node can be represented as type or as instances. We believe that node types should be used to specify node properties. For example, we may want to specify node properties such as memory size, CPU frequency, etc. We also want to be able to describe relationships between a node and other classifiers. For example, we may want to specify that a node is composed of other nodes or that a node realises some classes.

4.1.4 Deployment Diagram

The deployment diagram is defined in the UML Notation Guide. The purpose of the diagram is to show the configuration of run-time processing elements and the software components that live on them. The deployment diagram represents run-time manifestations of code units (i.e. instances of software components), and not for example source code. Components that do not exist as run-time entities are shown on component diagrams.

As presented above, the UML notation guide allows the representation of node types. It is not clear, from reading the UML specification, which diagram should be used to model node types. The UML specification only illustrates the deployment diagram using an instance form. In their UML textbook [1], Booch & al. give some examples where the deployment diagram is used to show the overall hardware structure of a system in a type form.

A major restriction in UML is that component types are not allowed in deployment diagrams. *This prevents us from modelling the overall hardware structure in a type form as it can be done in SOON* (showing both hardware blocks properties and relations between hardware block types and software component types).

Figure 6 shows how nodes and deployment diagrams can be used to model the run-time configuration for the sub-system Central Control presented using SOON in section 3. The figure illustrates that the instance of the node type "Central Control" is composed of the node instance "Computer" and 2 instances of the node "Operator terminal". As the diagram has an instance form, we are not allowed to show a variable multiplicity (the diagram shows links between instances, not associations). Moreover each instance of node or component type must be represented. We have not specified the instance names of the nodes.

The figure shows a link between the node instance "Computer" and the two instances of "Operator terminal". This link represents a physical connection among the nodes. The direction of signalling is not shown (as it was done in SOON). We will use UML extension mechanisms to represent

directions of signals.

The figure is not a pure hardware diagram, as it also shows that a component instance is deployed on the instance of "Computer".

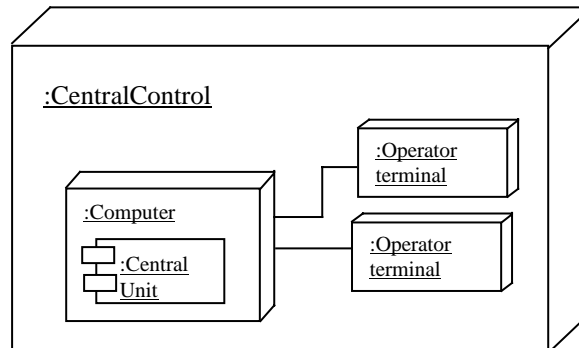


Figure 6: An example of a deployment diagram in UML

4.1.4.1 Representing hardware structure type

A hardware diagram type that shows the hardware structure of "Central Control" is illustrated in the next figure. Here the multiplicity of the composition between the node type "Central Control" and the node type "Operator terminal" is indicated using a range. No component is shown as component type; this is not allowed in deployment diagrams.

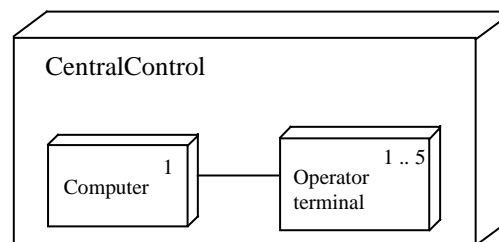


Figure 7: An example of a hardware diagram type in UML

4.1.4.2 Representing hardware/software structure type (extension to UML)

In order to represent a hybrid software/hardware structure, *we propose an extension of the UML deployment diagram*. This extension allows component types to be represented in a deployment diagram. We wish to model two main relations between node and component:

- the "execute" relation means that a hardware node is able to execute a software component. *The "execute" relation can be represented using composition.*
- the "implement" relation means that a software component (or a class) is realised by a hardware node. *The "implement" relation is represented using the UML stereotype of abstraction <<realize>> (Abstraction is a child of Dependency in the UML metamodel). Stereotypes are described in section 4.2.3.*

Figure 8 shows the hardware structure of "Central Control" where an "execute" reference from the node "Computer" to the software component "CentralUnit" is made.

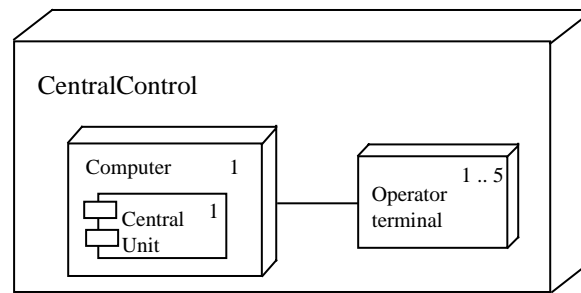


Figure 8: Using an extension of UML for hardware/software diagram

The next figure shows the hardware structure of "LocalControl" where an "implement" reference (using "realize") to the class "CentralUnit" is made. An icon is defined in UML for the stereotype "realize" (see section 4.2.3.3 on page 20).

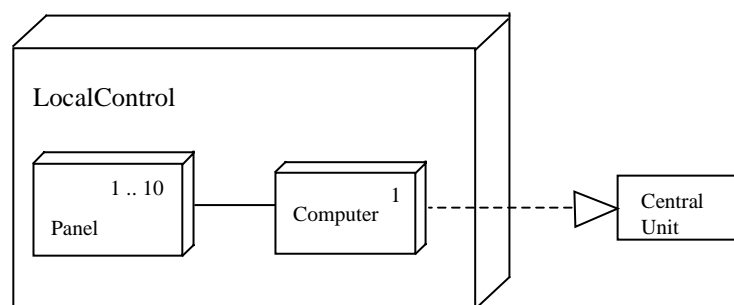


Figure 9: An example showing the use of implementation reference (UML extension)

4.2 Extension mechanisms

The basic model elements defined in the UML specification can be customised and extended with new semantics. Three extension concepts are defined in UML: stereotypes, tagged values and constraints. Stereotypes, tagged values and constraints apply to model elements and enable new kinds of elements to be added to the modeler's repertoire. They represent extensions to the modelling language.

4.2.1 Tagged values

A tagged value enables arbitrary information to be attached to any model element. A tagged value is expressed by a pair (name, value). The interpretation of the tagged value is beyond the scope of UML; it is determined by the user or a tool convention. The UML specification gives some examples of possible use, such as code generation options, model management information.

The UML Notation Guide explains that properties can be attached to an element using attributes, associations and tagged values. Attributes and tagged values are actually quite similar concepts. A major question is when should we use tagged values rather than attributes. The following aspects may be considered when choosing whether attributes or tagged values should be used:

- We can think about tagged value as a metadata because its value applies to an element, not to its instances.
- Tagged values are not interpreted by UML, so the value can be expressed using any syntax.
- It is usual to consider that information that is not part of the "Application Domain" should not be described by an attribute. This can be discussed. It is also usual to model several views of a system as for example a functional view, an implementation view, or a management view. Although implementation information does not necessarily belong to the "Application Domain", we may wish to represent this information using attributes in an implementation view. The examples given in the UML specification are not convincing. For example, the

tagged value (author, "Joe") could also be defined using an attribute of type string. Whether or not this information in this example is relevant in the model, can be discussed as the information belongs to the model management view.

- A tagged value does not change dynamically while the value of an attribute usually may be modified after the instance of the classifier is created. Note however that an attribute may be attached the property "frozen" meaning that its value cannot be altered after instantiation and initialisation of its values
- The information expressed using tagged values may be interpreted by special tools such as code generators. Another way to represent such type of information is to define model use conventions, attribute naming conventions, or attribute stereotypes (see section 4.2.3) for representing the information to be interpreted by the tools. For example, we may define that all the attributes defined in implementation views are to be interpreted by the building tools. We may also define an attribute name "language" for representing the implementation language selected to generate code for a software component.
- It is not clear, from reading the UML specification, whether or not tagged values are inheritable features. In the description of inheritance in section "2.5.4 Semantics" of the UML specification [5], the inheritable features of classifiers are explicitly listed. These include features (attributes, operations and methods) and participation in associations. Constraints (see section 4.2.2) are also mentioned as inheritable features for any model element. Neither the metamodel class diagrams nor the well-formedness rules make it clear that `Constraint` and `TaggedValue` should follow different rules as for as inheritance is concerned. We also note that section "2.6 Extension mechanisms" mentions that tagged vales are inheritable for stereotypes (see section 4.2.3).

As the purpose of the UML implementation diagrams is to describe the hardware and software units and their properties, we represent any information relevant for the implementation in these diagrams. We use attributes to describe information that properties of instances of the physical entities in the model while we will use tagged values to qualify elements (i.e. group elements that share some characteristic). Although the UML specification mentions that tagged values can be used to describe information specific to some tools, e.g. code generators, the code generation information should be described using attributes when this information varies for the different instances of a (user-defined) model element. We will see that tagged values are particularly of interest when describing stereotypes (see 4.2.3).

Figure 10 shows how attributes can be used in order to express implementation information. The node and component elements are elements that represent implementation entities; their properties are described using attributes. The attributes "language" and "priority" are not set to any default value in this diagram. A default value is defined for the attribute "OS". All the implementation properties described in this figure are of interest for code generation. It is necessary to define implementation attribute names that the code generator can recognise. Note how the realisation of "Validation" by "CentralUnit" is represented in the figure. The realisation association indicates a difference of abstraction between the entities.

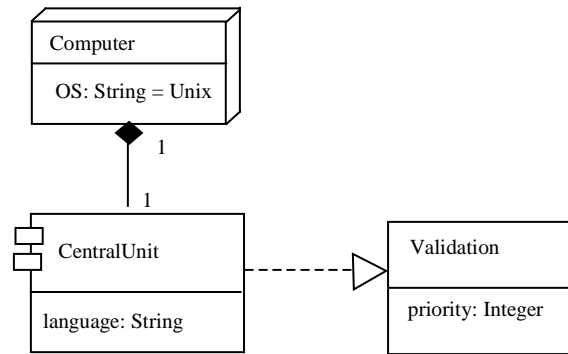


Figure 10: Using attributes for representing implementation information in a type diagram (UML extension)

The system type of Figure 10 may be instantiated as shown on Figure 11.

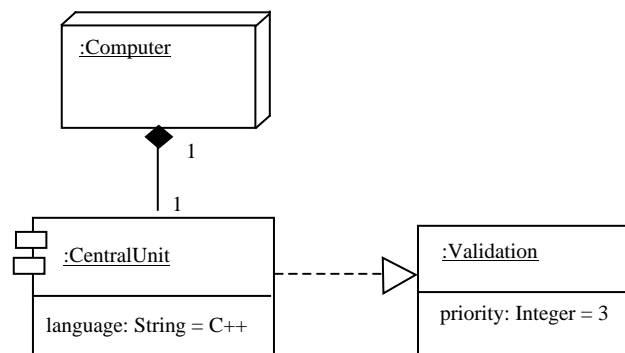


Figure 11: Using attributes for representing implementation information in an instance diagram

We have found it difficult to propose good examples showing the use of tagged values without defining stereotypes (see 4.2.3). Figure 12 is given for illustrating the notation. Tagged values are specified within braces **{ }**. Conventions for naming the tags and their potential values may be defined in order to enable their interpretation by a tool. The syntax for specifying the value has to be defined outside of UML.

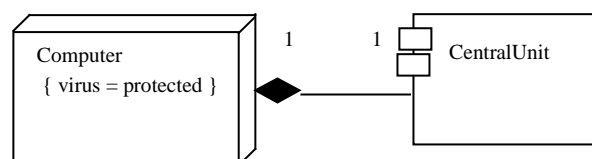


Figure 12: Using tagged values in an implementation diagram

4.2.2 Constraints

The UML constraint concept allows a constraint (or semantic restriction) to be specified for a model element or a group of model elements. The constraint can be written as an expression in a constraint language chosen by the user (or accepted by the tools used by the user). The UML specification suggests the formal language OCL (Object Constraint Language) for doing this.

In the architectural design description, we need to express constraints (or requirements) on the concrete system, e.g. optimisation aspects, response time requirements, security requirements. UML constraints can be used for that purpose. We do not propose any language for specifying

constraints. As for tagged values, in the case the information is to be interpreted by tools, the language to be used must be determined by the tool.

Figure 13 shows how UML constraints can be used. Here constraints are attached to single model elements (a component, a node or a connection). It is possible to attach a constraint to several elements (i.e. representing constraints between elements). It is also possible to refer to attributes of the model elements the constraint is attached to. The UML specification gives some examples.

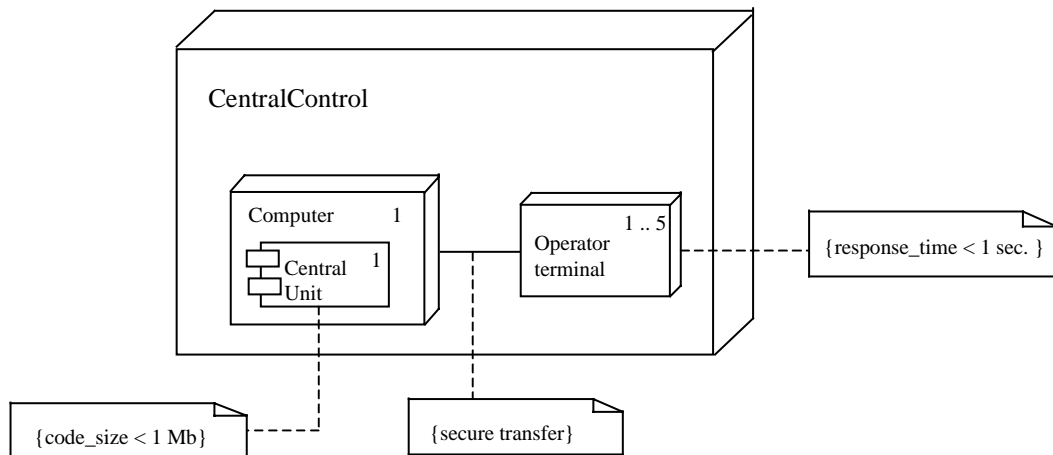


Figure 13: Using constraints in an implementation diagram

4.2.3 Stereotypes

The stereotype concept provides a way of classifying (marking) elements in order to add some new meaning or usage to the basic element of the same kind. The stereotype refers to a base class (model element) in the UML metamodel such as Class, Association, etc. The instances of a stereotype have the same structure as the instances of the model element the stereotype refers to (e.g. attributes, associations, and operations). The stereotype may specify additional constraints and required tagged values that apply to instances.

In the metamodel, `Stereotype` is a subtype of `GeneralizableElement`. TaggedValues and Constraints attached to a `Stereotype` apply to `ModelElements` classified by that `Stereotype`.

If a stereotype is a subtype of another stereotype, the subtype inherits the constraints and tagged values from the supertype. The stereotype subtype must apply to the same kind of base class as the supertype.

The UML specification defines itself several stereotypes; one of them, the "realize" stereotype, was used in Figure 6 on page 14.

The general presentation of a stereotype is to use the symbol of the basic model element where the name of the stereotype is added within guillemets `<< >>`. For example, the class stereotype `<<process>>` is defined in Z.109 [7] in order to represent an SDL process in UML. A graphic icon can also be associated to the stereotype. The stereotype icon may be used in addition to the icon of the basic model element or instead of this icon. The Telelogic deployment editor defines an SDL like process symbol for the `<<process>>` stereotype; the process symbol is used in the editor instead of the UML class symbol for showing process stereotypes.

4.2.3.1 Declaration of stereotypes

The classification of stereotypes themselves can be shown on a class diagram. This is a metamodel diagram that usually is not developed by the system developer but by the methodology or tool provider.

The UML specification does not clearly specify how the properties of the stereotype can be defined. How can tagged values, constraints on stereotypes be expressed? How can an icon be associated to a stereotype? We believe that it is important to specify the exact semantic of the stereotypes and that the definition should be provided to the methodology or tool provider, and be available to the system developer.

We have tried to define the SOON software structure concepts in UML (note that a process concept is also defined in UML - see section 4.2.3.3). Figure 14 illustrates the definition of software process, procedure and data. In this figure, guillemets (<< >>) are added to the keyword "stereotype", but not to the names of the stereotypes. This notation is used in [2]. We have attached constraints to the stereotypes, but are not sure whether or not this is permitted by UML. The description of the icons to be used for these stereotypes is not given. We may use the same icons as defined in SOON.

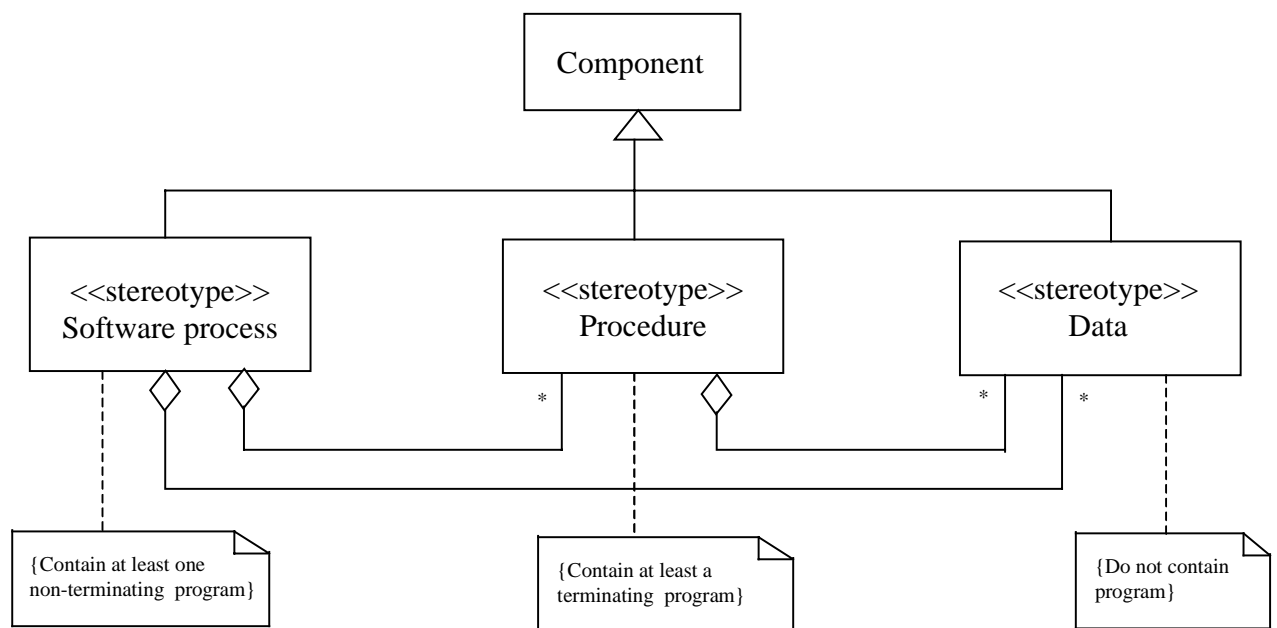


Figure 14: Stereotype definition

4.2.3.2 Using stereotypes

Assuming that we have defined the "connection" stereotype (connection between hardware blocks) and the "process" stereotype (representing an SDL process), we can extend the "Local Unit" hardware diagram. This is shown in Figure 15. In this figure, we also use the "OS" attribute representing an implementation property and a constraint.

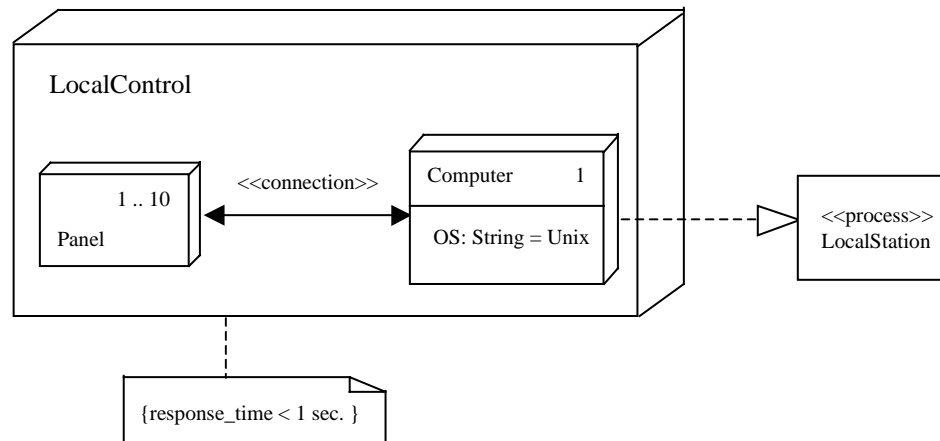


Figure 15: Using stereotypes

Until now, we have shown stereotypes attached to UML classifiers (e.g. component, class, and association). Stereotypes can be defined for any UML model elements. Thus we may also attach stereotypes to attributes. Section "3.24.1 List Compartment" of the UML Notation Guide describes how a property (stereotype or visibility rule) can be attached to a group of attributes or operations. A stereotype specified in front of a list of elements applies to all succeeding elements until another property string appears. As shown in Figure 16, we can use this mechanism in order to express the properties of variables of an SDL process.

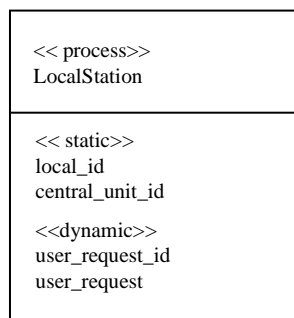


Figure 16: Using stereotypes for a list of elements

4.2.3.3 Stereotypes defined in UML

UML defines several stereotypes for the basic model elements. We have found out that some of them are relevant for architectural design. We present them here.

`<<process>>` and `<<thread>>` are stereotypes defined for `Classifier`. The `<<process>>` stereotype specifies that the classifier represents a heavyweight flow of control. The `<<thread>>` stereotype specifies that the classifier represents a lightweight flow of control. In [1], the authors relate these stereotypes to active classes and objects. An active object owns a process and thread, and can initiate control activity. An active class is a class whose instances are active objects. *According to the UML semantics, these stereotypes can be used for any classifier.* It may not make any sense for interfaces or nodes, but the concept may be applied to components. We represent SOON software processes using components of the stereotype `<<process>>`⁵. SOON has not defined the concept of thread, but we agree that it is a useful concept.

We must be careful however when using the stereotype `<<process>>`, as we also would like to use the stereotypes defined in Z.109 [7]. Z.109 defines a stereotype `<<process>>` in order to represent

⁵ Using the SDT deployment editor, a component is composed of threads that themselves are composed of classes. The thread level may be unspecified; in that case it is interpreted as having one thread for each component.

SDL processes; it is true that SDL processes have their own flow of control, so that a `<<process>>` stereotype defined using Z.109 semantics is also a `<<process>>` stereotype as defined by UML, but the inverse is not true. We will use the Z.109 `<<process>>` stereotype for class to indicate a reference to an SDL process; when there may be ambiguity, i.e. in the case the Z.109 stereotype may be interpreted as the UML stereotype, we will use the stereotype `<<SDL-process>>`. Figure 17 illustrates the use of the process and thread stereotypes. The component "CentralUnit" is a process that may consist of one or up to 1000 thread components "Cluster-unit".

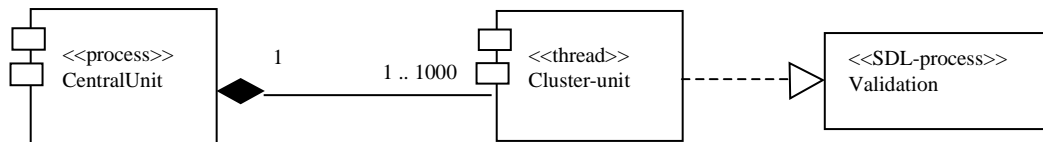


Figure 17: Using the stereotypes `<<process>>` and `<<thread>>`

`<<realize>>` is a stereotype defined for Abstraction (a child of Dependency, itself a child of Association). An abstraction is a dependency relationship that relates two elements or set of elements that represent the same concept at different levels of abstraction. The stereotype `<<realize>>` specifies a relationship between a specification model element (s) and a model element (s) that implements it. This stereotype can be used to show that a hardware or software entity realises a class (e.g. an SDL process). We have already used this stereotype in several of the examples earlier in this document. Figure 17 shows that the component "Cluster-unit" realises the SDL process "Validation". An icon is defined in UML notation for this relation. Note that, in the example given in "3.95 Deployment Diagram" in the UML specification [5], a composition relation is used between component instances and class instances. The objects represented in that example should be instances of implementation classes (i.e. there should not be any difference of abstraction between the component and composite elements).

`<<executable>>` and `<<file>>` are stereotypes defined for Component. The `<<file>>` stereotype denotes a document containing a program. The `<<executable>>` stereotype denotes a program that may run on a node. Both stereotypes are useful for system building. As shown in Figure 18, we can use the stereotype `<<file>>` to show information about files containing code derived from SDL.

`<<derive>>` is a stereotype defined for Abstraction. The stereotype `<<derive>>` specifies a derivation relationship among model elements. This stereotype can be used to show that one entity can be computed from another. Figure 18 shows that the file "validation.c" is derived from an SDL process "Validation".

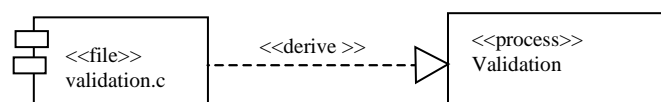


Figure 18: Using the stereotype `<<file>>`

`<<become>>` is a stereotype of Flow. Flow is a child of Relationship (Relationship is an abstract class in the metamodel that is parent of Association, Dependency, Flow and Generalization). A flow is a directed relationship between a source object (s) and a target object (s). The stereotype `<<become>>` specifies a flow relationship between the same instance at different points of time. It can be used to show the migration of components from node to node, or

objects from component to component [2]. As for notation, the keyword `<<become>>` is used on a dashed arrow (see "3.95 Deployment" in UML specification [5]). Figure 19 illustrates the use of this stereotype. The object "TrafficTeller" migrates from "MT" to "MO".

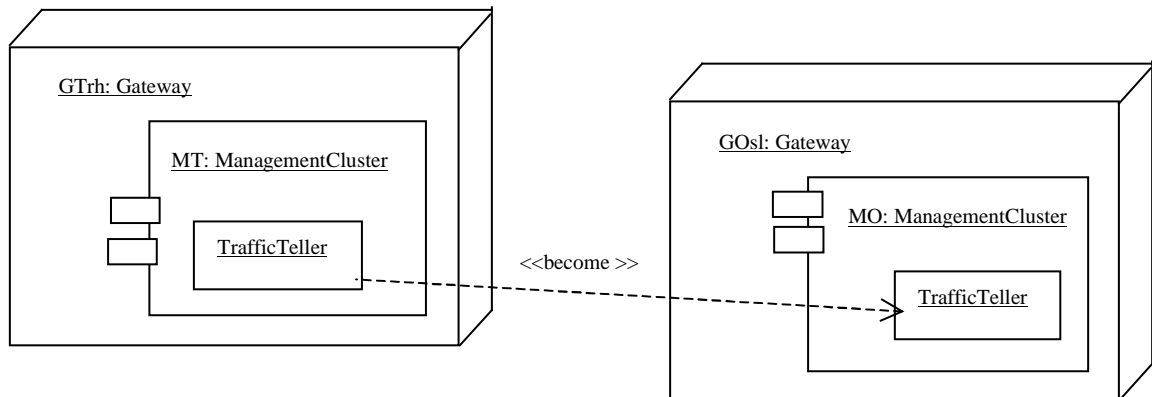


Figure 19: Using the stereotype `<<become>>` for showing migration of objects

`<<create>>` is a stereotype defined for `BehavioralFeature` (i.e. a dynamic feature of a model element such as an operation or method). The stereotype `<<create>>` specifies that the designated feature creates an instance of the classifier to which the feature is attached. According to the UML semantics, all classifiers may have behavioural features, so it should be possible to use this stereotype for any classifier. Using the UML Notation, the stereotype is only used in sequence diagrams. Sequence and collaboration diagrams are used to show a pattern of interactions between instances or more precisely role instances. *The examples given in the UML specification [5] and in the UML textbook of Booch & al [1] show only role instances for classes. However, according to the specification it should be allowed to show collaboration between role instances for any classifier.* In order to represent creation of component instances, we may:

- use the role of component instances in sequence and collaboration diagrams. An example using a sequence diagram is given in Figure 20.
- extend the deployment diagram use in order to be able to show creation of component instances in the diagrams. An example is given in Figure 21.

The second example gives more information as node instances are also presented.

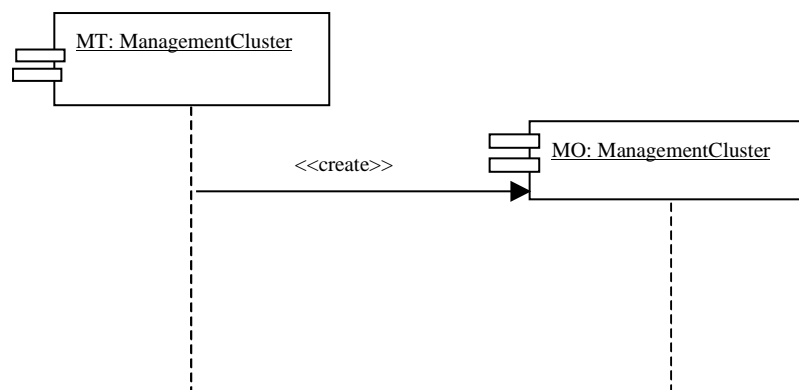


Figure 20: Using Sequence diagrams with component instances.

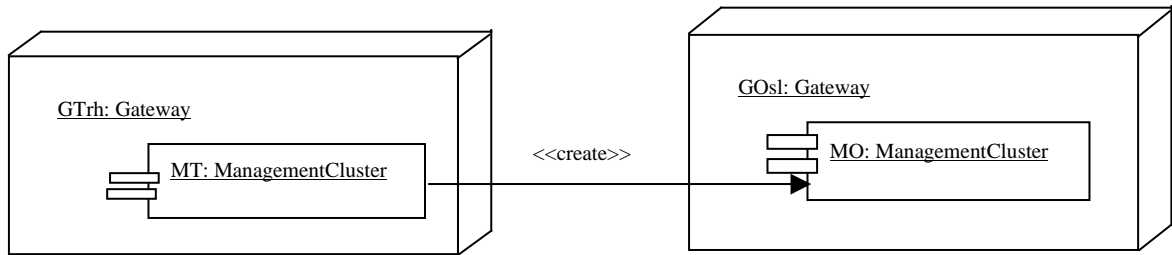


Figure 21: Showing creation of components in deployment diagrams

<<call>> is a stereotype defined for Usage. Usage is a child of Dependency. A usage is a dependency in which a client requires the presence of a supplier. The stereotype <<call>> specifies that an operation in the source class invokes an operation in the target class. We will use this stereotype in order to show activation or call relations between software components. In UML this stereotype is used in sequence diagrams or collaboration diagrams. As for the previous stereotype (<<create>>), we may use component instances in sequence and collaboration diagrams, or we may prefer to use this stereotype in deployment diagrams as shown in Figure 22.

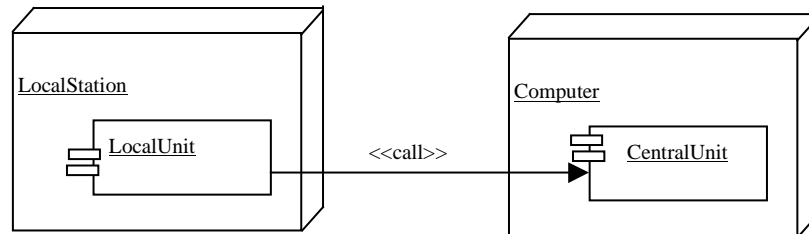


Figure 22: Showing call between components in deployment diagrams

5 Guidelines for using UML for Architectural Design

This section summarises the discussions of the previous section and presents shortly our approach to UML for architectural design.

5.1 Mapping SOON-UML

5.1.1 Overall structure of the hardware system

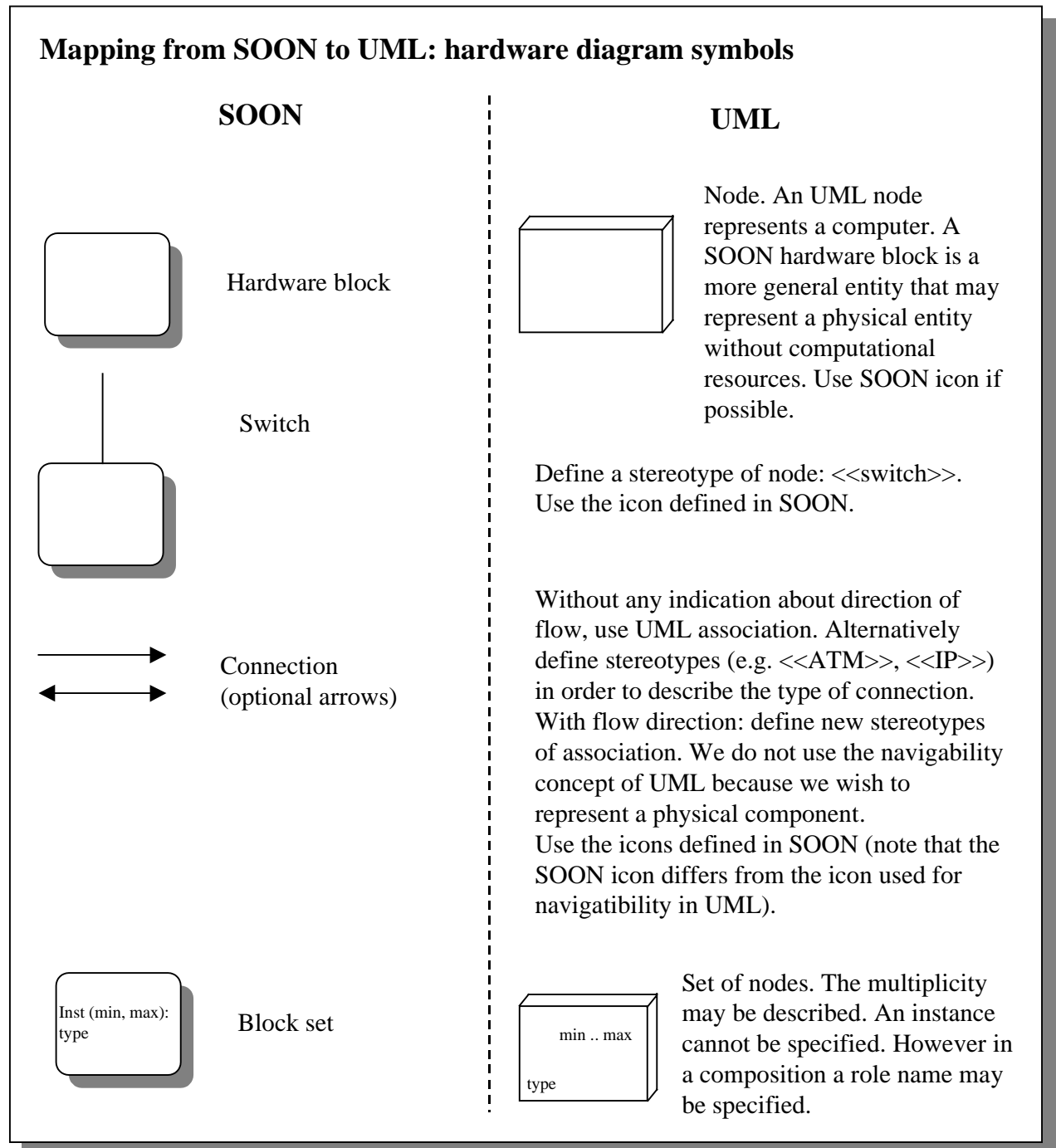


Figure 23: Mapping from SOON to UML (hardware symbols)

The UML node does not map directly the SOON hardware block. A SOON hardware block is more general than a UML node in that it can represent any hardware entity capable of sending and receiving signals. UML nodes must contain memory and have processing power, and thus cannot

represent electronic or mechatronic entities. A model element should be defined in UML that represents a hardware entity. A node could be defined as a subtype of this element. It would then be possible to define other kinds (or subtypes) of hardware entities.

The structure of the hardware is shown in a **deployment diagram**. We use deployment diagrams for showing either types or instances. We use node composition.

The references *implement* and *execute* are respectively mapped to the abstraction stereotype <<realize>> and composition between nodes and components or classes.

5.1.2 Overall structure of the software system

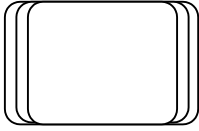


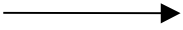
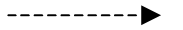
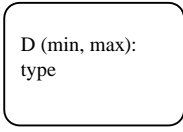
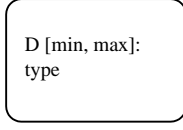
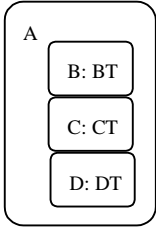
Mapping from SOON to UML: software diagram symbols		
	SOON	UML
	Software process block	UML defined stereotype <<process>> applied to Component. The icon defined in SOON may be used.
	Procedure block	Define a stereotype of Component: <<procedure>>. Use SOON icon if possible.
	Data block	Define a stereotype of Component <<data>>, or use Class. Use SOON icon if possible. The UML stereotype <<table>> for Component may also be used.
	Data flow	Use an UML association with navigability or define a stereotype of Association <<dataflow>>. In the last case, the icons defined in SOON may be used. Note that the icons are identical to the connection icons defined for hardware.
	Activation	Use the stereotype call of usage. In SOON, calls between data blocks are not allowed. This can be expressed by defining a constraint on the stereotype <<data>>.
	Set of data elements	Use a set of component stereotypes <<data>> or of class (role name and multiplicity may be described).
	Indexed array of data elements	Define a new stereotype <<indexed data>> of Component or Class. Use a set of the stereotypes.
	Structure of data elements; record	Use composition of Components and Classes (role name and multiplicity may be described).

Figure 24: Mapping from SOON to UML (software symbols)

The structure of the software is shown either in a **component diagram** (for component types) or in a **degenerate deployment diagram** (for component instances). We use composition between components and between components and classes.

The references *implement* is mapped to the abstraction stereotype <<realize>>. Composition may be used when there is no difference of abstraction between the component and composite elements. We have not defined any mapping for *implemented_by* as nodes are not represented in component diagrams. We have not defined any mapping for *executed_by* as the inverse relation is preferably shown on deployment diagrams.

5.1.3 Combined software/hardware diagrams

In order to represent a hybrid software/hardware structure, *we propose an extension of the UML deployment diagram*. This extension allows **component types** to be represented on a deployment diagram. See an example in Figure 8.

5.1.4 Reference to SDL entities

SDL entities are represented by stereotyped classes. We use the stereotypes used in Z.109 [7] i.e. <<system>>, <<block>>, and <<process>>. Note that we use the UML stereotype <<process>> for Components (i.e. component with heavy flow of control) and the Z.109 stereotype for classes (i.e. class representing an SDL process). We believe that Z.109 should have defined <<SDL_process>> in order to avoid confusion with UML stereotypes (and then also <<SDL_system>>, <<SDL_block>>).

A **qualifier** (attribute or tagged value) is attached to each class. The qualifier has the same semantics as in SDL and represents the hierarchical structure from the system or package level to the defining context.

Other stereotypes defined in Z.109 may be used in relation with code generation, e.g. <<signal>>.

5.2 Code generation information

We attach code generation information (or properties) to the elements described on the component and deployment diagrams. Different code generators may require different properties to be described.

5.2.1 Design information related to the SDL application

The SDL entities are defined as stereotyped classes.

UML constraints are used to describe optimisation aspects, response time requirements and security requirements. We have not defined any constraint language. When using ProgGen, the language defined in [4] may be used.

Properties related to entity types and instances (e.g. priority, SDL timer units) are represented using attributes. General properties related to types of entities (e.g. type definition expansion) are represented using stereotypes (constraints or tagged values may be attached to stereotypes). Properties may be related to some properties of an entity type (e.g. process variable properties); in that case they are represented using stereotypes.

5.2.2 Design information related to the execution target

This information is represented by components and nodes, and by the properties of these nodes and components (i.e. using attributes or tagged values).

Interfaces to the environment are described using UML interfaces. A component can be shown to realise an interface. Examples are given in the next figures. In these figures, the component "CentralUnit" realises the interface "ComProtocol". In the first figure, a simple reference to the interface is done.

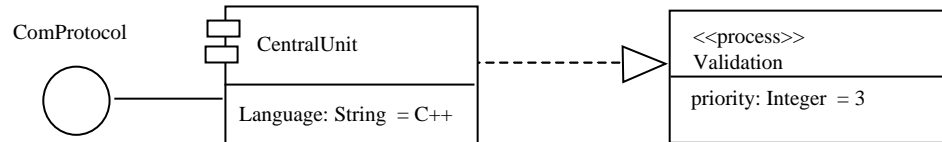


Figure 25: Interface to the environment

In the figure below, the interface is further detailed: the operation signatures are specified.

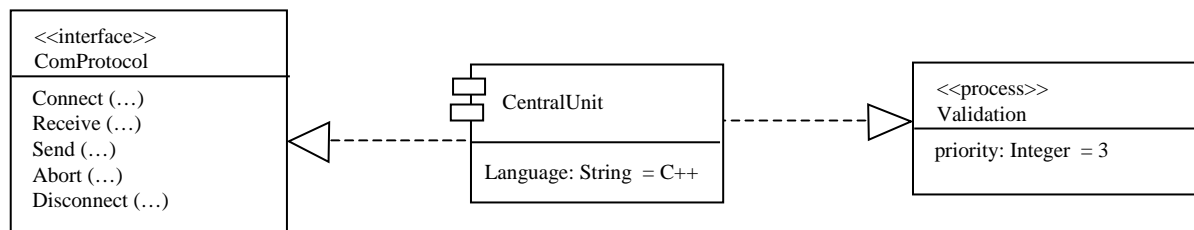


Figure 26: Description of an interface to the environment.

It is also possible to indicate which file contains the software that realises the interface using a dependency relation, as shown below.

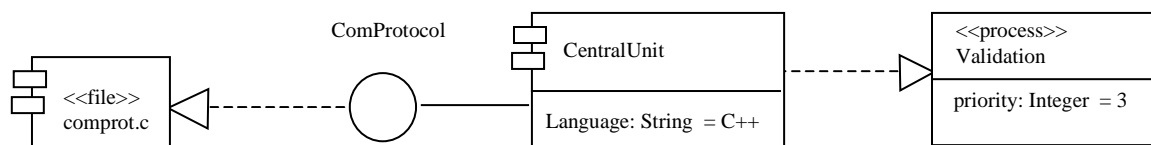


Figure 27: Information about the realisation of an interface to the environment

5.2.3 Design information independent of any model

Here we may want to indicate the type of tool that is used to produce the implementation (e.g. code generator, compiler). This tool information can either be described in a generic manner using stereotypes (e.g. all components attached the stereotype `<<java>>` are compiled using a Java compiler) or for specific components using the `<<derive>>` stereotype defined for Abstraction.

When the `<<derive>>` stereotype is used, we may also wish to describe the tool that is used. This may be done either by using constraint (or tagged value) attached to the derivation, or by defining new subtype stereotypes of `<<derive>>`. In Figure 28 a constraint has been used.

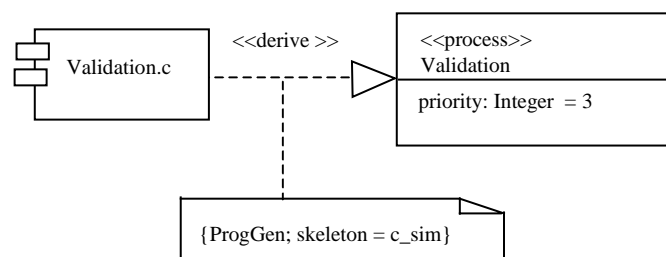


Figure 28: System building information using `<<derive>>` and tagged values

In Figure 29, we have used a new stereotype `<<derive_with_pg>>`. A tagged value is attached to the derivation in order to specify more details about the derivation.

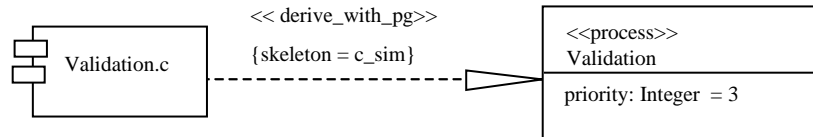


Figure 29: System building information using a subtype stereotype of <<derive>>

5.2.4 Design information related to the system simulation

This information is related to the system simulation e.g. loss of signals, errors. Specific simulation may be defined by simulation tools and require tool-dependent properties to be described. Constraints and tagged values are probably a good means for describing the quality of a connection or the error rate.

5.3 Dynamic system reconfiguration

Until now, we have mainly shown static system structures. In emerging technologies such as active networks or distributed mobile agent systems, the migration of components is a basic mechanism. UML defines the stereotype <<become>> for Flow that enables one to describe an entity at different points of time. An example was given in Figure 19 (on page 22) that illustrates the migration of an object from one node to another node using this stereotype.

Note that the diagram in this example has an instance form. In section 4.1.4.2, we proposed an extended use of the deployment diagram that enables to show hardware/software structure in a type form. We can use this form to show migration between node types in a more general way.

We may also wish to show creation of main system entities in a diagram showing the overall system structure using the UML defined stereotype <<create>>. As we discussed in section 4.2.3.3, sequence or collaboration diagrams may be used, or deployment diagrams may be extended. In addition to creation, we may wish some important steps in a dynamic reconfiguration sequence on a diagram. In that case a collaboration diagram is well suited. Figure 30 shows an example. The diagram is actually a representation close to both collaboration and deployment diagrams. Sequence numbers are used to order to show that creation of the component is done before the migration takes place.

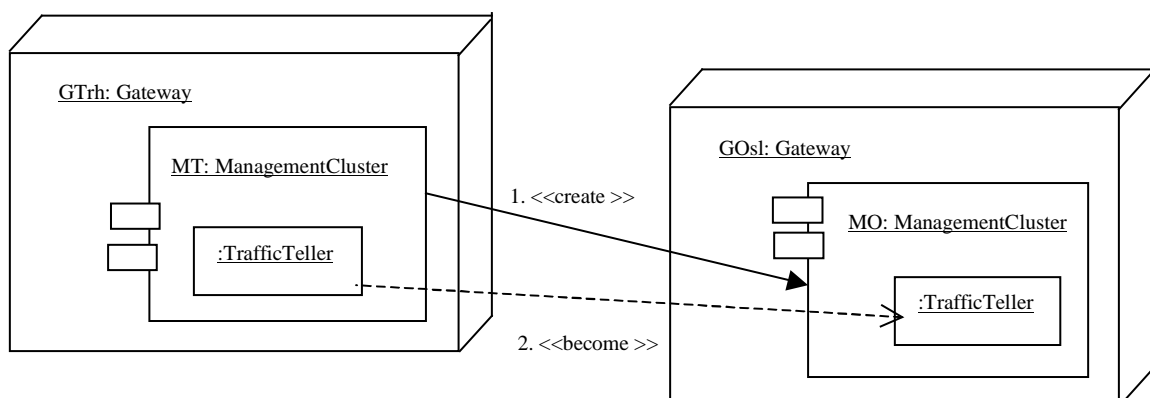


Figure 30: Collaboration diagram with node and component instances

6 Telelogic Deployment Editor

The SDT deployment editor (or DP editor) [8] is a new tool in Telelogic Tau SDL Suite 3.6. The tool allows the developer to describe graphically the configuration of the running system, or more precisely to describe the partitioning of the SDL entities in the run-time environment. Build-scripts can be generated automatically based on the partitioning information of the deployment description.

The DP editor is based on a subset of UML concepts, and does not make it possible to completely describe the overall hardware and software architecture. The focus here is the distribution of the SDL entities on nodes and on concurrency aspects.

The DP editor support four symbols representing the following entities:

- the Node represents a run-time physical object (i.e. a UML node instance),
- the Component represents a physical, replaceable part of a system that packages implementation (i.e. some kind of UML component instance with a restricted semantic),
- the Thread represents an OS thread (i.e. a UML stereotype thread; the DP editor documentation does not specify the kind of Classifier the stereotype is applied to),
- the Object represents an SDL entity, i.e. a system, a block or a process.

Two types of relations can be defined:

- An Association shows a communication channel (physical link) between nodes.
- A Composite Aggregation can be specified between a node and components, between a component and threads or objects, and between a thread and objects.

It is not possible to describe associations between other entities than nodes.

The DP editor restricts the kind of information (or properties) that can be attached to each type an entity. More details are given later in this section.

The following figure shows how entities of the deployment diagram can be related:

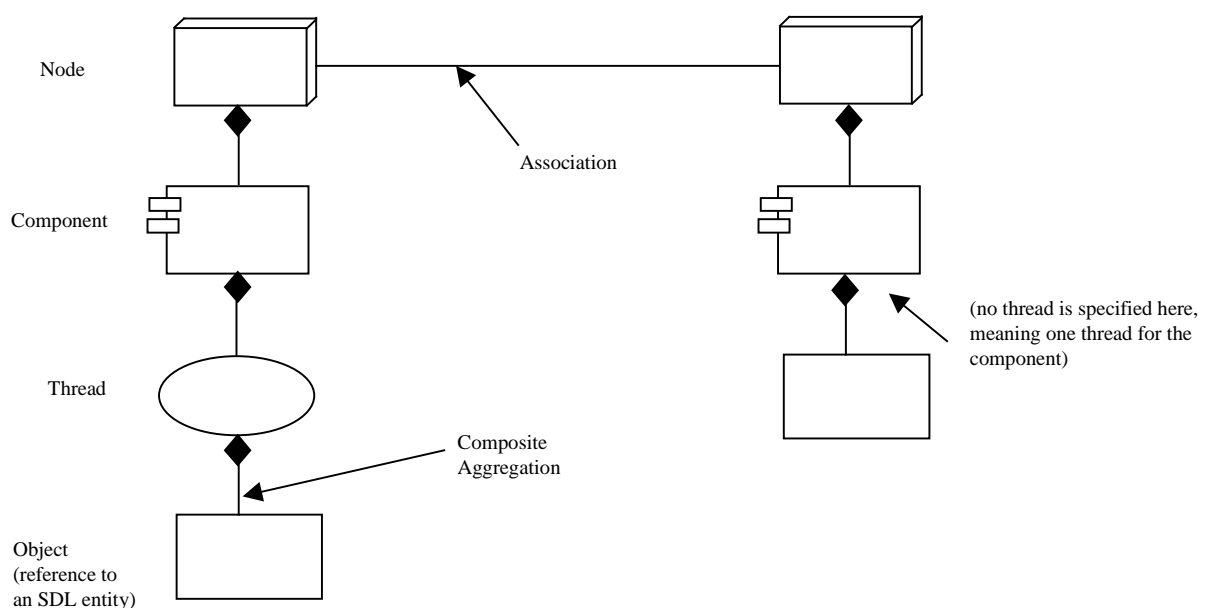


Figure 31: Entities defined in the DP editor.

The DP editor does not support the graphical nested form. Composition is shown using composition lines.

6.1 Nodes and components

The following characteristics can be specified for nodes and components:

- a name
- a stereotype
- "properties" (they are represented as UML tagged values on the diagram)
- "build info"

The DP editor does not explain how the tool interprets the stereotype (except for <<external>>) and the properties. Neither does the documentation specify what the build information expresses and the syntax that should be used.

It is not possible to specify attributes or any other relation than composition or association in the restricted manner described previously. Thus realisation or derivation cannot be shown.

During build script generation, each component results in a partition. The stereotype <<external>> on a component means that the symbol and its sub-tree are not included in the build script. If no nodes are specified, this is interpreted as resulting in one implicit node for all components.

6.2 Threads

It is not possible to specify any property for threads. The thread level may be unspecified; in that case it is interpreted as having one thread for each component.

Due to build script generation limitations, many threads belonging to the same components will not affect the build script. This is interpreted as one thread for each component.

6.3 Objects

The following characteristics can be specified for objects:

- a name
- a stereotype that represents the type of SDL entity i.e. system, block or process.
- "properties" (they are represented as UML tagged values on the diagram)
- a qualifier that represents the entity path in the SDL hierarchical structure from the system top level.

It is not allowed to specify new attributes to objects or any relation between objects.

The stereotype <<external>> on an object means that the symbol is not included in the build script.

6.4 Association

They represent connections between nodes. The following characteristics can be specified:

- a name
- a protocol
- a direction
- the type of encoding (e.g. BER)

The information is purely descriptive. Associations are ignored during script generation.

6.5 Composite Aggregation

Only the multiplicity of the composition can be specified. The multiplicity information is purely descriptive and is ignored during script generation.

6.6 Conclusion

The DP editor is a step in the right direction for architectural design with UML and constitutes an improvement for SDL users (compared with the earlier textual notations previously supported by SDT). However the DP editor focuses on the deployment of the SDL entities on computers, and provides limited support for architectural design. The set of properties of the physical entities that can be represented is limited. The DP editor does not make use of the full UML semantics for classifiers. We recommend Telelogic to add support for features suggested in this report.

7 References

- [1] Booch, Grady, Rumbaugh, James & Jacobson, Ivar. The Unified Modeling Language User Guide. Addison Wesley, 1998. ISBN 0-201-57168-4.
- [2] Booch, Grady, Rumbaugh, James & Jacobson, Ivar. The Unified Modeling Language Reference Manual. Addison Wesley, 1999. ISBN 0-201-30998-X.
- [3] Bræk, Rolv & Haugen, Øystein. Engineering Real Time Systems. Prentice Hall, 1993. ISBN 0-13-034448-6.
- [4] Johansen, Ulrik. SEP sluttrapport: Designstyrt oversetter fra SDL. August 1999. SINTEF rapport STF 40 A99040. (in Norwegian).
- [5] OMG Unified Modeling Language Specification. Version 1.3, June 1999.
- [6] TIME: The Integrated Method. Information available at <http://www.informatics.sintef.no/projects/time/> (valid URL in December 1999)
- [7] Z.109 - SDL Combined with UML (SDL_{with}UML_{for}SDL). ITU-T Recommendation. Draft document, November 1999.
- [8] On-line documentation for SDT deployment editor (or DP editor). Telelogic Tau SDL Suite 3.6.