UNIVERSITY OF ZAGREB
**FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING**
AND
MÄLARDALEN UNIVERSITY
**SCHOOL OF INNOVATION, DESIGN AND ENGINEERING**

# UML PROFILE FOR SAVECCM

Ana Petričić

Zagreb, July 2008.

# Table of contents


1. Introduction ........ 1
  1.1 Overview ........ 1
  1.2 The goals of this thesis ........ 2
  1.3 Outline of the text ........ 2

2. Component – Based Software Engineering ........ 3
  2.1 An itroduction to CBSE discipline ........ 3
  2.2 Basic concepts in CBSE ........ 6
    2.2.1 Component ........ 6
      2.2.1.1 Component validation and quality ........ 8
    2.2.2 Component Interface ........ 8
      2.2.2.1 Interface specification ........ 9
      2.2.2.2 Contracts ........ 9

3. Component models ........ 12
  3.1 On component models ........ 12
  3.2 Important aspects of a component model ........ 12
  3.3 Industrial Component Models ........ 13
    3.3.1 COM, DCOM, COM+ ........ 13
    3.3.2 .NET Framework ........ 14
    3.3.3 Enterprise JavaBeans ........ 15
  3.4 Research Component Models ........ 16

4. Unified Modeling Language ........ 17
  4.1 About Unified Modelling Language ........ 17
  4.2 The UML Metamodel ........ 17
  4.3 The UML Infrastructure Architecture ........ 19
  4.4 The UML Superstructure Architecture ........ 21
  4.5 The UML Component model ........ 21
    4.5.1 Composite structures ........ 22
      4.5.1.1 InternalStructures package ........ 22
      4.5.1.2 Ports package ........ 23
      4.5.1.3 Collaborations package ........ 23
      4.5.1.4 StructuredClasses package ........ 23
      4.5.1.5 Actions package ........ 24
    4.5.2 Components package ........ 24
      4.5.2.1 Component ........ 26
      4.5.2.2 Connector ........ 29
  4.6 UML Extensibility Mechanisms ........ 30
    4.6.1 First – class extension mechanism ........ 30

4.6.2 Profile extension mechanism ........ 31
4.6.2.1 Description of classes in Profiles package ........ 33
4.6.2.2 Characteristics of profile extension mechanism ........ 34

5. SaveComp Component Model ........ 35
5.1 The SaveCCM application domain ........ 35
5.2 SaveCCM – syntax and semantics ........ 35
5.3 Architectural elements ........ 36
5.3.1 Components ........ 37
5.3.1.1 Clock and Delay ........ 38
5.3.1.2 Composite component ........ 38
5.3.2 Switches ........ 38
5.3.3 Assemblies ........ 39
5.3.4 Ports ........ 39
5.3.5 Connections ........ 39
5.4 The Cruise Control Example ........ 40

6. SaveUML profile ........ 42
6.1 Background ........ 42
6.2 The process of mapping UML to SaveCCM ........ 42
6.2.1 Specific design decisions ........ 43
6.2.1.1 Components ........ 43
6.2.1.2 Subcomponents ........ 43
6.2.1.3 Interfaces ........ 44
6.3 The SaveUML profile specification ........ 45
6.3.1 Imported libraries ........ 45
6.3.2 Profile constraints ........ 45
6.3.3 Description of stereotypes ........ 46
6.3.3.1 SaveAssembly ........ 46
6.3.3.2 SaveAttribute ........ 47
6.3.3.3 SaveBindPort ........ 47
6.3.3.4 SaveClock ........ 48
6.3.3.5 SaveCombinedInPort ........ 48
6.3.3.6 SaveCombinedOutPort ........ 50
6.3.3.7 SaveComplexConnection ........ 50
6.3.3.8 SaveComponent ........ 51
6.3.3.9 SaveComposite ........ 52
6.3.3.10 SaveConnection ........ 53
6.3.3.11 SaveDataInPort ........ 54
6.3.3.12 SaveDataOutPort ........ 54
6.3.3.13 SaveDelay ........ 55
6.3.3.14 SaveDelegation ........ 56
6.3.3.15 SaveFromComplex ........ 56

6.3.3.16 SaveModel ... 57
6.3.3.17 SaveSwitch ... 58
6.3.3.18 SaveSwitchCondition ... 59
6.3.3.19 SaveToComplex ... 60
6.3.3.20 SaveTriggerInPort ... 60
6.3.3.21 SaveTriggerOutPort ... 60
6.4 Using SaveUML profile ... 61
6.4.1 UML CASE tool ... 61
6.4.2 Applying stereotypes to user model elements ... 61
6.4.3 Modelling UML models using SaveUML profile ... 63
6.4.4 Validating the model ... 65

7. SaveUML transformations ... 67
7.1 Abstract ... 67
7.2 Conceptual design ... 67
7.3 XSLT transformations ... 68
7.3.1 User model files ... 69
7.3.2 Input parameters ... 70
7.4 Transformation tool ... 71
7.4.1 System specification ... 71
7.4.2 The transformation tool architecture ... 72
7.4.3 Error handling ... 72
7.4.4 Using SaveUML transformation tool ... 73

8. Conclusion ... 78

## Definitions and acronyms

| | |
|---|---|
| API | Application Programming Interface |
| CASE | Computer Aided Software Engineering |
| CBD | Component Based Development |
| CBSE | Component Based Software Engineering |
| COM | Component Object Model |
| CORBA | Common Object Request Broker Architecture |
| DCOM | Distributed Component Object Model |
| DSML | Domain Specific Modelling Language |
| DTD | Document Type Definition |
| EJB | Enterprise JavaBeans |
| IDL | Interface Definition Language |
| MIDL | Microsoft Interface Definition Language |
| MOF | Meta Object Family |
| MS | Microsoft |
| OCL | Object Constraint Language |
| RSM | Rational Software Modeller |
| SaveCCM | SaveComp Component model |
| UML | Unified Modeling Language |
| W3C | World Wide Web Consortium |
| XML | eXtensible Markup Language |
| XMI | XML Metadata Interchange |
| XSL | eXtensible Stylesheet Language |
| XSLT | XSL Transformations |

# List of figures

Figure 2-1: The Component - based design pattern ........................................ 4
Figure 2-2: Provisions and requirements of a component .............................. 10
Figure 3-1: COM approaches in composition, (a) containment and (b) aggregation 14
Figure 4-1: OMG four – level metamodel hierarchy ........................................ 18
Figure 4-2: The Core Packages ...................................................................... 19
Figure 4-3: Sub-packages of the Classes package and their dependencies ... 20
Figure 4-4: The top-level package structure of the UML ................................ 21
Figure 4-5: A component with internal structure ............................................. 23
Figure 4-6: Dependencies of the Component packages ................................ 25
Figure 4-7: UML metaclasses included in UML 2.0 Component Model ........... 26
Figure 4-8: Mapping component interfaces at multiple levels .......................... 28
Figure 4-9: A component with packaged elements .......................................... 29
Figure 4-10: Component with two subcomponents and three connectors ...... 30
Figure 4-11: Profile example, (a) definition, (b) application ............................ 32
Figure 5-1: Graphical notation of SaveCCM .................................................... 36
Figure 5-2: Clock and Delay components, graphical presentation .................. 38
Figure 5-3: SaveCCM model of ACC system .................................................. 41
Figure 6-1: Applying stereotype – Properties view .......................................... 62
Figure 6-2: Applying stereotype – available stereotypes ................................. 62
Figure 6-3: Applying stereotype – editing tagged values ................................. 63
Figure 6-4: Adding a property to a component ................................................ 63
Figure 6-5: UML model of ACC system – Project explorer view ...................... 64
Figure 6-6: Connecting a SaveModel element to an SaveComponent model element 65
Figure 6-7: Validating model – live validation popup window .......................... 66
Figure 6-8: Validating model – batch validation error messages ..................... 66
Figure 7-1: Conceptual design of SaveUML transformations .......................... 67
Figure 7-2: SaveUML transformation tool architecture .................................... 72
Figure 7-3: The stand-alone application graphical interface ............................ 74
Figure 7-4: Using RSM plug-in, pop-up menu ................................................. 74
Figure 7-5: Using RSM plug-in, exporting UML model .................................... 75
Figure 7-6: Using RSM plug-in, warning message .......................................... 75
Figure 7-7: UML model of ACC system ........................................................... 76
Figure 7-8: Transformed ACC system model .................................................. 77

# 1. Introduction

## 1.1 Overview

Due to the increasing complexity of information technology (IT) – based software application systems and intensive competition in the marketplace, the main challenge for software developers today is to cope with complexity and to adapt quickly to changes. One key to the solution of these problems is reusability. The rapidly emerging approach called component-based development (CBD) establishes the idea of reuse. In CBD, software systems are built by assembling components already developed and prepared for integration.

Various component models have been created to facilitate the design of component – based systems, usually for some specific domain. One of such models is the SaveComp Component Model (SaveCCM), a research component model intended for embedded control applications in vehicular systems. SaveCCM is a simple model in which flexibility is limited to facilitate analysis of real – time and dependability.

Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software system. During the past 10 years, it became a *de facto* standard for visual design of software systems. It is widely accepted due its expressiveness and a variety of application domains.

While the UML already provides standards for the design of component - based systems in general through a UML 2.0 component model, it does not provide everything necessary for the design of systems based on specific implementation architectures, in particular, time-critical and resource-critical systems. UML is lacking in some key areas that are of particular concern to real-time and embedded system designers and developers such as a quantifiable notion of time and resources. In particular, it does not explicitly capture the semantics expressible in the SaveCCM model.

There is therefore a significant motivation for ensuring that the UML can be used to describe SaveCCM-based systems. The UML was designed to be extensible, however, and provides standard extension mechanisms for defining new semantic constructs. These mechanisms can be used to define constructs describing SaveCCM model elements. One of the UML extensibility mechanisms are UML profiles. They specialize or refine the UML for a specific purpose. A profile does not add any basic concepts. Instead, it specializes existing concepts and defines conventions for applying them. The UML Profile for SaveCCM can be used to develop models describing systems from SaveCCM application domain.

Additionally there is a need to join those two modelling languages and to allow transformation of existing UML models to SaveCCM model and reverse.

## 1.2 The goals of this thesis

The first goal of the thesis is to provide mapping from UML to SaveCCM domain by creating a UML profile for expressing the semantics of SaveCCM model and thus to allow modelling of SaveCCM models using UML tools. The profile should fulfil the following requirements:

- Define a standard approach for modelling of component – based real-time systems using SaveCCM semantics.
- To define complete, accurate and unambiguous representations of SaveCCM architectural elements and their semantics for modelling purposes.
- Support all practical requirements commonly encountered in the design of systems from SaveCCM application domain.
- Define well-formedness rules, expressed as Constraints written in the Object Constraint Language (OCL) for validation purposes.

The second goal of the thesis if to design a transformations form existing UML models to SaveCCM models and reverse. A prototype of the tool performing those transformations should be developed.

It is important to emphasize that this thesis is a continuation and an upgrade of the SaveUML project [SP web]. The work done during this project was taken as the basis for accomplishing the goals mentioned above.

## 1.3 Outline of the text

The first part of this document elaborates a theoretical background of the problem. Section 2 describes the component – based approach in software engineering. Section 3 explains the term of "component model" and gives an overview of existing industrial component – based technologies. The UML is described in section 4, with a brief analysis of UML 2.0 component model. A research component model – SaveCCM is analysed in section 5.

The second part of the document provides a solution for achieving the thesis goals. Section 6 specifies the SaveUML profile, a UML profile which maps the UML component model to SaveCCM. model. SaveUML transformations are described in section 7 together with a transformation tool prototype.

To emphasize important terms *italic* font style will be used.

# 2. Component – Based Software Engineering

This section introduces the new and increasing approach in software engineering, the Component – Based Software Engineering.

## 2.1 An itroduction to CBSE discipline

The field of component-based software engineering (CBSE) is concerned with the development of software applications and systems by assembling pre-existing smaller pieces, which are termed *software components*. A consequence of this approach is that software components may be independently developed, reused in many various applications or systems, or can be replaced within those systems without any side effect. To assemble components, a proprietary code that connects the components is usually needed (this code is often called a .*glue code*).

In an ideal world, assembling of components would be simple and smooth. However, in real world, CBSE is still a new discipline, and there are many problems to solve, and supporting technologies and tools to develop. That is why assembling of components is still a complex and often difficult process.

The widespread development and reuse of software components is regarded by many as one of the next biggest phenomena for software development. Reusing high-quality software components in software development has the potential for drastically improving the quality and development productivity of component-based software. Building systems based on reusable components is not a new idea. It has been proven to be a very effective cost-reduction approach.

Component-based systems are a result of adopting a component-based design strategy. By design strategy, it is considered something very close to architectural style – a high level *design pattern* described by the types of components in a system and their patterns of interaction [Bass 98]. Software component technology includes the concepts that support and reflect this design pattern as it is depicted graphically in Figure 2-1. This reflection is due to the fact that software component technology does not only exist in development process but it is also the part of deployed system.

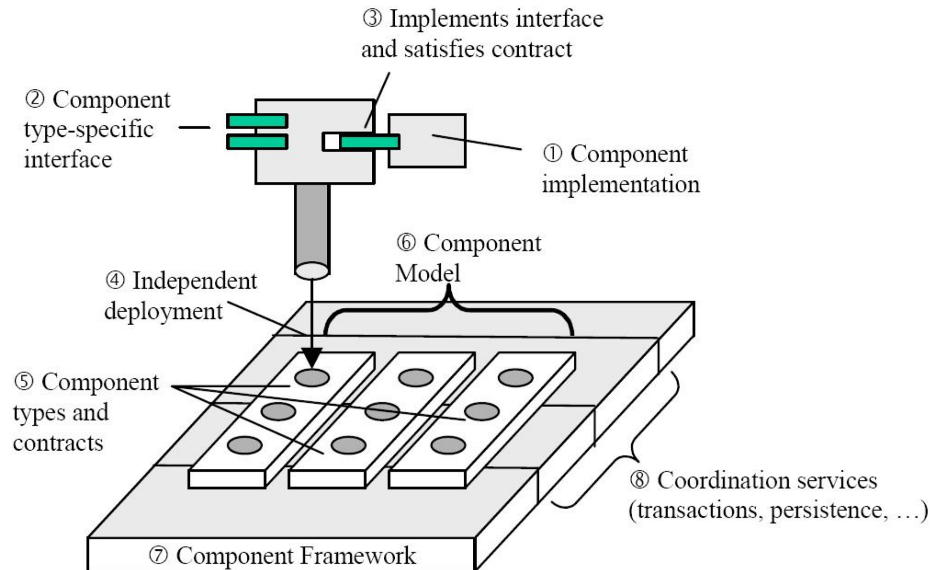


Figure 2-1: The Component - based design pattern

This pattern can be found in commercial software component technologies such as Sun Microsystems' Enterprise JavaBeans, Microsoft's' COM+ and many others.

In Component Based Development (CBD) a component (①) is a software implementation that can be executed on a physical or logical device. A component implements one or more interfaces that are imposed upon it (②), this means that the component satisfies certain obligations that are often described as a *contract* (③). Those contractual obligations ensure that independently developed components follow certain rules so that components interact in predictable ways, and can be deployed into standard build-time and run-time environments (④). A component-based system uses a small number of distinct component types, each of which plays a specialized role in a system (⑤). A *component model* (⑥) consists of component types, their interfaces, and additionally, a specification of the allowed patterns of interaction among those component types. A component framework (⑦) provides runtime services (⑧) to support the component model. In many cases, component frameworks are similar to special-purpose operating systems, although they operate at much higher level of abstraction.

CBD is a relatively new approach in software engineering, it is recognized as a very powerful approach and is able to improve significantly the development of software. Some of the most important benefits of CBD are:

- *Independent extensions*. A problem that often appears, and which causes software to become no longer adaptable in a cost – effective way is lack of flexibility. Components are units of extension and a component model defines how extensions are made. The component model and

framework ensure that extensions do not have unexpected interactions, thus extensions may be independently developed and deployed.

- *Reduced time to market*. The uniform component abstractions and the availability of components drastically reduce time it takes to design and to develop a system. Design time is reduced because key architectural decisions have been made and are embedded in the component model and framework.
- *Improved predictability. Since* component models express design rules that are enforced over all components deployed in a component-based system, then various global properties can be designed into the component model, so that properties such as security, scalability, execution time and so forth can be predicted for the system.
- And many other advantages such as more effective management of complexity, increased productivity, improved quality and a greater degree of consistency

However, there are also some disadvantages and factors that can discourage the application of component – based development approach [Crnković 02]:

- *Time and effort required for development of components*.
- *Unclear and ambiguous requirements*. One of the major problems of software development comes from unclear, ambiguous, incomplete, and insufficient requirements specifications. Reusable components are intentend to be used in various applications, some of which may yet be unknown and the requirements of which cannot be predicted. This makes it more difficult to identify the requirements of the component properly.
- *Conflict between usability and reusability*. Component must be sufficiently general, scalable, and adaptable in order to be widely reusable. However this demand results with component to become more complex (and thus more complicated to use) and more demanding of computing resources (and thus more expensive to use).
- *Component maintenance costs*. Although application maintenance costs can be lowered, component maintenance costs can be very high since the component must respond to the different requirements of different applications running in different environments and with different reliability requirements.
- *Reliability and sensitivity to changes*. Because components and applications have separate life cycles and different kinds of requirements, there is some risk that a component will not completely satisfy the particular requirements of certain applications or that it may have characteristics not known to the application developer. When

introducing changes at the application level (changes such as the updating of an operating system, the updating of other components, and changes in the application), developers face the risk that the change introduced will cause a system failure.

## 2.2 Basic concepts in CBSE

The main principle in CBSE is a concept of the component. Other terms, such as interface, contract, framework, and pattern, are thereby also related to component-based software development. This section gives an overview of the concept of a component in the component – based software engineering. Second part of this section deals with the concepts of *interface* and *contract*, and discusses the issues related to those concepts.

### 2.2.1 Component

There are many different understandings of the concept of a software component among engineers, managers and teachers. Here are some of the important definitions of software component given by the experts. One of the earliest definitions is given by Gready Booch [Booch 93]:

*A reusable software component is a logically cohesive, loosely coupled module that denotes a single abstraction.*

The definition above captures the idea of a component as an encapsulated software module consisting of closely related elements. Later, Clement Szyperski presented his well-known definition of a software component at the 1996 European Conference on Object-Oriented Programming [Szyperski 99]:

*A software component is a unit of composition with contractually specified interfaces and context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*

This definition is well accepted in the CBSE community because it emphasizes the major properties of software components that can not be found in traditional software modules, such as context independence, composition, deployment, and contracted interfaces. The definition of a component expressed in Unified Modeling Language (UML) [UMLs 07] is:

*A component represents a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces.*

A software component considered in a context of a component – based systems must have the following characteristics [Gao 03]:

- *Identity*. A component must be uniquely identifiable among its environment. Modern component technology uses a well – defined

naming scheme to ensure this demand (CORBA, EJB, and Microsoft DCOM).

- *Modularity and encapsulation*. Each component encapsulates a set of closely related elements and implements appropriate logic to perform a specific task.
- *Independent delivery*. Software component must be delivered as independent part which can be integrated into the system and can be replaced under certain conditions. Each component must play an independent role in a system.
- *Contract – based interface*. A software component has its interface which determines a contract between the client of an interface and a provider of an implementation for the interface. A contract specifies what is expected from the client in order to use the interface to access a component service function. It also defines what kind of services and implementations have to be provided to meet the service contract. This implicates that a component delivers its provided services only if its clients access the provided component interface in a right way.
- *Reusability*. Reusability of software components is the fundamental notion of CBSE. Unlike a conventional software module which usually has a very limited reuse scope, a software component provides multiple-level granularities for reuse on a large scope. The reusable elements of a software component include its analysis specification, component design and design patterns, source code, and executables. In addition, component deployment mechanisms and test support information (such as test cases and test scripts) are also a reusable items.

Properties listed above, evince that a software component merges two distinct perspectives: component as an implementation and component as an architectural abstraction; components are therefore architectural implementations. Viewed as implementations, components can be deployed, and assembled into larger (sub)systems. Viewed as architectural abstractions, components express design rules that impose a standard coordination model on all components. These design rules take the form of a component model, or a set of standards and conventions to which components must conform.

The most important feature of a component is the separation of its interfaces from its implementation. What distinguishes the concepts in CBSE from concepts in many other programming languages are requirements of integration of a component into an application. Component integration and deployment should be independent of the component development life cycle and there should be no need to recompile the application when updating with a new component.

A software component should have the following elements to ensure it is well defined [Crnković 02]:

- A set of interfaces provided to, or required from the environment.
- An executable code, which can be coupled to the code of other components via interfaces.

To improve component quality, especially to avoid problems and risks that arise in CBD (see disadvantages of CBD), the component specification can additionaly contain [Crnković 02]:

- The specification of nonfunctional characteristics, which are provided and required.
- The validation code, which confirms a proposed connection to another component.
- Additional information, which includes documents related to the fulfilling of specification requirements, design information, and use cases.

#### 2.2.1.1 Component validation and quality

In component-based software engineering, application systems are constructed by reusing and integrating components. This suggests that the quality of reusable software components has a serious impact and ripple effect on software systems that adopt them. Therefore, testing and quality assurance of reusable components become extremely important to the success of component-based software development because a single defect of a component may affect many component-based systems in projects, product lines, and organizations. This raises some serious questions and concerns to software engineering researchers and industry practitioners.

Testing of the components should focus on the consistency of the component specification with the user specification and customization testing [Gao 03].

Component users usually do not have the access to the source code, therefore black-box (also known as functional testing) testing methods are adopted in this context.

For component developers, more efforts should be conducted to ensure the reusability, interoperability, and other related quality features, which are generally overlooked by traditional unit level software testing techniques. White box testing (also known -as structural testing or glass-box testing) is used in this case.

### 2.2.2 Component Interface

The ability to integrate components into assemblies, to develop components independently, to develop a market of components, depends fundamentally on the notion of component interfaces. The concept of interface is basic and familiar,

however the importance and criticality of interfaces in CBSE exposes limitations of conventional methods of interface specification

#### 2.2.2.1 Interface specification

Interface abstraction is a mechanism to control dependencies that arise between modules in an application or a system. In a programming language, an application programming interface (API) is a specification of module properties which the clients of the module can depend on. Respectively, the clients should not depend upon the properties that are not specified by the API. All modern programming languages support some form of interface specification, also some independent interface specification languages have been developed (for example Object Management Group's Interface Definition Language [IDL web]).

API "hides" the implementation of a module, the idea is that information hiding makes systems substitutable. But, it turns out that this theory has its weaknesses as it depends upon APIs hiding properties that clients shouldn't depend on. The API can only hide the properties about which it can "speak", and programming languages are only equipped to speak about a narrow range of properties. All other properties can "leak" through the interface abstraction. Usually APIs can only express functional properties (such as signature of a service a module provides), but are not able to express extra-functional properties (such as accuracy, availability, safety, execution time, latency, reliability, robustness, performance etc.). Extra-functional properties are often called a *quality attributes*, or when associated with a particular service, *quality of service*. As APIs cannot describe these properties, they cannot hide them. Therefore, modules may come to depend upon some of these properties, thus reducing the probability that one module can be substituted with another.

Attempts have been made to extend APIs in various modern languages to make them more expressive for extra-functional properties, however this is still a matter of a speculation.

#### 2.2.2.2 Contracts

Interfaces were described in previous section as specifying dependencies from components that implement services to the clients that use these services. The client has some assumptions about the services that a component provides and it depends on those assumptions.

A client and a component can also be considered as co-dependent; a client depends upon a component to provide service in a certain way, but a component also depends upon a client to access those services and use them in a certain way (for example to provide necessary arguments within certain bounds). This arranged and strictly defined interaction is described as a contract between a

client and a component. A contract enables a more accurate specification of a component's behaviour.

The idea of interface contract is most closely linked to the work of Bertrand Meyer and the developments inspired by that work [Meyer 92][Meyer 97].

Interface contract is a metaphor and has several meanings:

- Contracts are established between two or more parties. This is the generalization of above mentioned co-dependency to component based systems with multiple components which collaborate.
- Parties often negotiate the details of the contract before they become signatories.
- Contracts define normative and measurable behaviours for all parties (component certification).
- Contracts can not be changed unless the changes are agreed by all parties. This is very important for establishing markets for components.

According to Meyer [Meyer 92], a contract lists the global constraints that the component will maintain (the invariant). For each operation within the component, a contract also lists the constraints that need to be met by the client (the precondition) and those the component promises to establish in return (the postcondition). Those three constraints prescribe components behaviour.

It is convenient to think of an interface as union of *provisions* and *requirements.* "*Provisions*" is a set of services provided by a component. On the other hand, "*requirements*" is a set of services which component requires to be provided by its environment. In addition, in context of contracts, requirements part may also involve the demand for appropriate and accurate use of the provisions (requirement that client component has to supply all necessary parameters and with legal values).This is depicted in Figure 2-2 for two components C1 and C2.

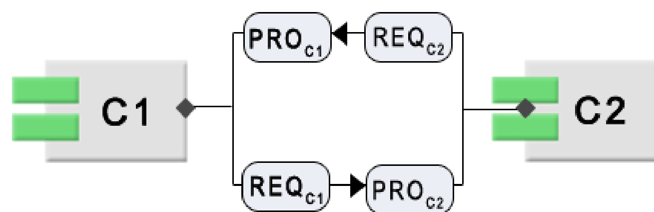


Figure 2-2: Provisions and requirements of a component

In this illustration, C1 obligates that it will implement provisions (labelled as $PRO_{C1}$), but to do so it needs C2 to implement the requirements (labelled as

$REQ_{C2}$). The same demand exists for the opposite direction on C2 provisions and C1 requirements (labelled as $PRO_{C2}$, $REQ_{C1}$ respectively). For C1 and C2 to interact, C1 must provide what C2 requires, and C2 must provide what C1 requires.

Contracts shift the focus from specification of components to specification of patterns of interactions and the obligations common to all of participants in these interactions. The total behaviour of a component may be quite complex because it may participate in many contracts.

# 3. Component models

This section provides an explanation of what a component model is and what are the main characteristics of every component model. Section 3.3 gives an overview of different component technologies currently available in industry. Section 3.4 explains the importance of research component models.

## 3.1 On component models

A component model specifies the standards and conventions imposed on developers of components. It defines the ways to construct components and regulates the ways to integrate and assemble components. It supports component interactions, composition, and assembly. In addition, a component model also defines the mechanisms for component customization, packaging, and deployment. Compliance with a component model is one of the properties that distinguish components (when they are used) from other forms of packaged software.

For each component model there usually exists a *component framework* which implements the services that support or enforce a component model.

## 3.2 Important aspects of a component model

Some important aspects of a component model are:

- *Component types*. A component's type may be defined according to the interfaces it implements. For example if a component implements three different interfaces X, Y and Z, then it is of type X, Y and Z. Different component types can play different roles in systems. It means that a component that implements X, Y and Z is polymorphic – it can play the role of an X, Y, or Z at different times. This approach is an important aspect of components found in Microsoft COM and Sun Java technologies.
- *Interaction schemes*. Component models usually specify which communication protocols are used, and how qualities of service (such as security and transactions) are achieved. The component model may describe how components interact with each other, or how they interact with the component framework. Interaction schemes may be common across all component types or unique for a particular component types.
- *Resource binding*. The process of composing components consists of binding a component to one or more resources. A resource can be a service provided by a framework or by some other component in that

framework. A component model describes how and when components bind to these resources.

## 3.3 Industrial Component Models

This section provides an overview of the most important existing industrial component models, MS COM (Section 3.2.1), .NET (Section 3.2.2) and Enterprise JavaBeans (Section 3.2.3).

### 3.3.1 COM, DCOM, COM+

Component Object Model (COM) [MSCOM web] is developed by Microsoft and is intended to run software components within different processes located on the same node. Its distributed version, Distributed COM (DCOM), extends functionality of COM to run components distributed over the network. The last COM version, COM+, adds new features to COM; Microsoft Transaction Server – for using transactions, Message Queue Server – for asynchronous invocations and other service for improving performance and security. COM is not restricted to any platform, however the mainly supported platform is Microsoft Windows.

COM object is a piece of binary code, thus COM components may be written in any programming language on condition that compiler is able to compile it into a binary file with an internal structure including virtual tables and function calling conventions specified in COM specification.

COM uses Object Remote Procedure Call [ORPC web] for method invocation and to specify components Microsoft IDL (MIDL) may be used (MIDL is not directly used by COM, it is used to pre-generate source code with MIDL compiler).

COM component model only allows specifying provisions of a component in form of provided interfaces. Requirements must be obtained programmatically from a source code. COM additionally allows specifying attributes of components which serve to configure components, values of attributes may be specified at a compile time or at run time.

When an interface is defined, it should not be changed, new methods should not be added and existing methods should not be modified or removed. This restriction is not enforced but is highly recommended, a new functionality should be added by creating a new interface instead of modifying the existing one. The reason for this rule is the fact that in CBSE, different versions of a component interface may cause problems. COM solves this threat by freezing the current interface and creating a new one on every modification, thus supporting multiple interfaces for the different versions of an interface. This, however, leads to an increase in the number of new interfaces, which in practice describe the same component [Crnković 02].

COM supports two composition techniques:

- *containment*: an owning component declares a part or all provided interfaces of subcomponents. Reimplementation of these interfaces is a call (delegation) to subcomponent whose interfaces are implemented.
- *aggregation*: the owning component can expose the interface of the subcomponent (as it implemented them it self). When a client is obtaining an interface from the owning component in order to invoke subcomponent's services, subcomponent's interface may be returned and a client works directly with a subcomponent (no delegation is needed).

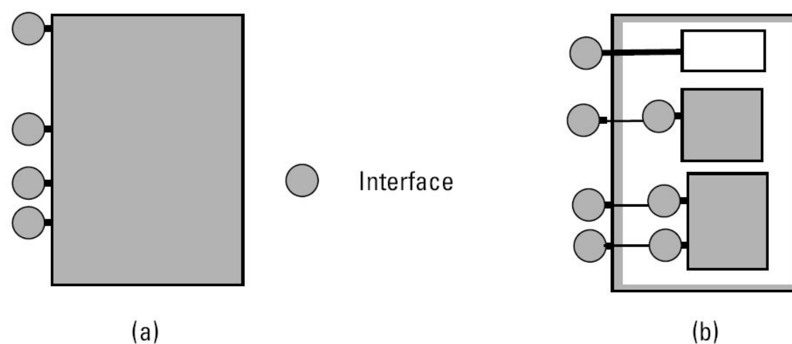


Figure 3-1: COM approaches in composition, (a) containment and (b) aggregation

These two approaches are depicted in Figure 3-1 (taken from [Crnković 02]).

In order to use COM component it must be registered in a registry database where a Global Unique Identifier (which is a 128 – bit key) is used to identify COM components. COM components are shared within a system, therefore any application can use any of the registered components. A big drawback of shared components is that a replacement of a COM component with a never version may cause incompatibility problems since the component may be used in more than one application.

### 3.3.2 .NET Framework

*.NET Framework* [MS.NET web] is a component model developed by Microsoft and it brings a completely new approach against COM in process of creating and deploying components. Unlike COM, which relies on binary interoperability, .NET relies on language interoperability and introspection. In order to achieve language interoperability .NET uses Microsoft Intermediate Language which is an interpreted byte code very similar to Java byte code. Its interpreter is a Common Language Runtime, an equivalent of Java Virtual Machine.

Components in .NET are called *Assemblies*. Assembly consists of compiled classes and a manifest file which is a component descriptor and contains all published data types and dependencies. Manifest file holds definitions of provisions of .NET components, while requirements are not explicitly defined, they must be obtained programmatically from a source code.

The .NET model is not hierarchical and composition of components is not supported. It may be achieved only explicitly from a source code.

### 3.3.3 Enterprise JavaBeans

Enterprise JavaBeans (EJB) is a component model developed by Sun Microsystems with current version 3.0 [SunEJB 05]. EJB has quite limited scope but despite its limitations, it has been widely used and popular in Java community.

EJB is primarily used for a client – server model of distributed computing. EJB components are limited to Java programming language, but they may be invoked from various other languages e.g. C++, C#, Visual Basic .NET.

EJB specification introduces three kinds of components called *beans*: *Entity beans*, *Session beans* and *Message – driven beans*. Communication between beans or between client and a bean is performed using Remote Method Invocation, which is a Java implementation of a Remote Procedure Call.

- *Entity bean*: its purpose is to access to data remotely over network. Each entity bean represents an object view on one record from the database and is defined by primary key. They may be shared between multiple users that use a primary key to access a particular bean. Invocations are performed synchronously. Entity beans are statefull due to permanent storage background.
- *Session beans*: an opposite to entry beans, they are not permanent and have no primary key since they are not backed by a database or other form of permanent storage. Session beans are not shareable. Invocations of session beans are synchronous. Session beans may be statefull or stateless. Statefull bean maintains its state across different method calls. It is intended to be used by one remote client at a time. Stateless bean does not hold its state and it may be used by more than one remote client at a time.
- *Message – driven beans*: as session beans, do not represent any data directly, however they may access any data in an underlying database. They are executed when a message from a client arrives on a server. This means that their invocation in asynchronous.

The interface of a bean is a set of owning ports. Beans expose two kinds of interfaces:

- *remote interface*: represents provisions of a bean. Provides an access point for a client and must be implemented by a developer of a bean.
- *home interface*: provides methods for creating and finding beans. Home interface is automatically provided by an EJB container.

Both kinds of bean interfaces are provided interfaces. EJB does not support required interfaces of a bean.

EJB *container* is an application server for executing beans.

## 3.4 Research Component Models

A robust market in software components requires standard component models and frameworks. Standard component models need to be general and applicable to many different problem domains in order to be widely used. However, experience has shown that different application domains have different requirements for performance, security, availability and other quality attributes. This argues the need for more than one, and possibly many component models and frameworks.

Research component models involve various characteristics that are not supported at all or that are supported partially by existing industrial component models. Therefore, they are more suitable for using in specific problem domains.

The SaveComp Component Model (SaveCCM) is a modelling language for embedded systems designed with vehicle applications and safety concerns in focus. The SaveCCM component model was developed within the SAVE project on Mälardalen University, Västerås, Sweden. It is intended to be sufficiently expressive for the needs of embedded control designers, while at the same time being restricted enough to facilitate predictability, dependability and analysis. The SaveCCM component model will be described in detail in Section 5.

# 4. Unified Modeling Language

This section provides an introduction to Unified Modelling Language, it describes UML infrastructure and superstructure and the UML metamodel in context of four – layer metamodel hierarchy. Section 4.5 gives an analysis of UML component model which is used to design component – based systems. Finally, the last section describes the ways to tailor UML for specific problem domains.

## 4.1 About Unified Modelling Language

The Unified Modeling Language (UML) is a visual language for designing, specifying and analysis of an application or a system structure, behaviour, business processes and data structure of object oriented software systems. It is a general-purpose modelling language that can be used with all major object and component methods, and can be applied to all application domains and implementation platforms (e.g. J2EE, .NET).

UML is maintained by Object Management Group (OMG) and the current version is v2.1.1 adopted in February 2007. The OMG adopted UML 1.1 specification in November 1997. Since then UML had several minor revisions, the most recent being the UML 1.4 specification, which was adopted in May 2001. For more than ten years, UML has been the industry standard and as the de facto standard modelling language, the UML facilitates communication and reduces confusion among project stakeholders.

## 4.2 The UML Metamodel

UML 2.0 is a metamodel that is architecturally aligned within the OMG *four – level metamodel hierarchy*. In the OMG four – level architecture, a model at one layer is used to specify models in a layer below. In turn, a model at one layer can be viewed as an "instance" of some model in the layer above (except the first layer). Also, each layer adds a new functionality and refines the semantics against the layer above. The main advantage of having a shared metamodel is that generic tools can be built that operate on any MOF – compliant model (MOF – Meta Object Family is one of the models form the highest layer). The four layers are the *meta-metamodel layer* (M3), the *metamodel layer* (M2), the user *model layer* (M1) and the *user object layer* or *runtime layer* (M0). The UML metamodel sits at the M2 layer and as such it is described as an instance of a meta-metamodel from the M3 level.

The top layer, *meta-metamodel layer* referred as M3 defines basic constructs (a language) for use in layers below. The *Meta Object Family* (MOF) metamodel is an example of this layer. MOF is also a base model for UML layer.

The next layer, *metamodel layer* – M2 is an instance of M3. The UML is positioned at M2 layer as an instance of MOF. The M2 layer is more complex than M3 layer, its main role is to define semantics for how elements from a user model (layer M1) that are instances of elements from the UML layer (layer M2) get instantiated in the runtime layer (layer M0).

The M1 layer is the layer where user models are created. Instances of models from M1 layer are running applications at M0 layer.

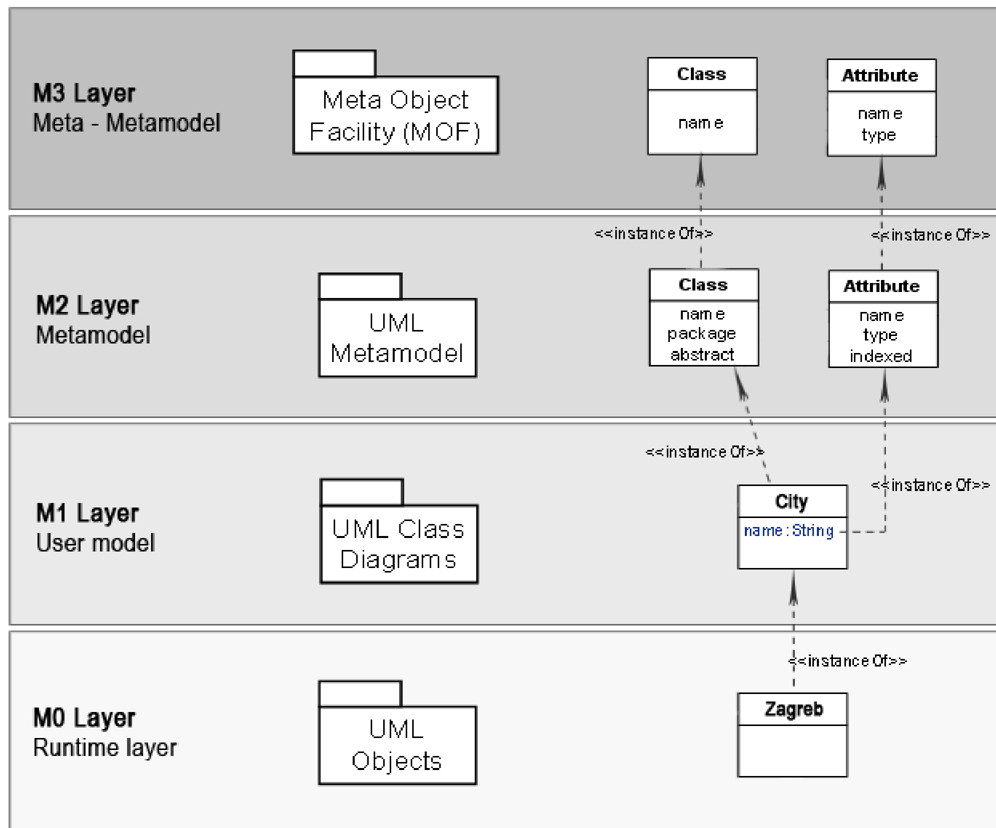


Figure 4-1: OMG four – level metamodel hierarchy

The four – level metamodel is shown in Figure 4-1. As an example, a Class and Attribute metaclasses are presented. UML metaelements Class and Attribute are instances of Class and Attribute meta-metaelements from MOF. City is a user defined class (as an instance of the UML metaclass Class) with an attribute "name" (as an instance of UML metaclass Attribute), while Zagreb is an instance of City.

When talking about elements it is important to distinguish which layer context the talking involves (distinguish between metamodels and models). In the following text, the `Courier New` font will be used to refer to UML elements that belong to M2 layer – UML metaclasses, and Arial font will be used when talking about elements from M1 layer – user model elements. User model elements will be

referred to with the name of the metaclass instead of using expression "instance of the metaclass *".

Also, for better legibility, when referring to UML packages *Italic* style will be used in further text.

## 4.3 The UML Infrastructure Architecture

The UML infrastructure is defined by the *InfrastructureLibrary*, which is a base for various metamodels including MOF. InfrastructureLibrary consists of the *Core* and *Profiles* packages, where the latter defines the mechanisms that are used to customize metamodels and the former contains core concepts used when metamodelling.

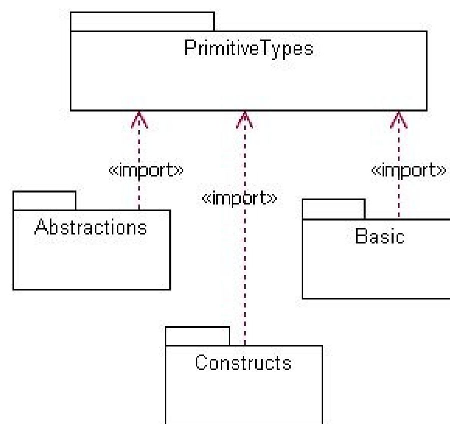


Figure 4-2: The Core Packages

*Core* package is divided into a number of packages: *PrimitiveTypes*, *Abstractions*, *Basic*, and *Constructs* as it is shown on Figure 4-2 [UMLi 07]:

- *PrimitiveTypes* package contains a set of basic, predefined types used for defining syntax of metamodels. Four types are defined, *Integer*, *Boolean*, *String* and *UnlimitedNatural*.
- *Abstractions* package defines abstract metaclasses that are intended to be further specialized, those metaclasses define basic elements and associations used in metalanguages (for example Classifier, Element, Generalization etc.).
- *Constructs* package, contrary to *Abstractions* package, mostly contains concrete metaclasses such as expressions, constraints and others.
- *Basic* package represents a minimal modelling language with constructs that are used as the basis for the produced XMI for UML, MOF, and other metamodels. This package is used to build other metalanguages

and contains constructs for specifying data types, types of elements, packages, and class – based modelling.

*Profiles* package allows extending existing metamodels in order to adapt it to specific problem domains or specific platforms (see section 4.6.2).

The *Kernel* package of *Classes* in UML 2.1.1 Superstructure is the central part of UML, it contains the modelling concepts of the UML, including classes, associations, and packages. All the metaclasses of every other package are directly or indirectly dependent on it. The *Kernel* package mostly reuses InfrastructureLibrary and adds some more modelling capabilities. Reusing of InfrastructureLibrary is done by bringing together the different packages of the Infrastructure using package merge. *PrimitiveTypes* and *Constructs* packages are explicitly merged from the InfrastructureLibrary, while all other packages of the InfrastructureLibrary::Core are implicitly merged through the ones that are explicitly merged.

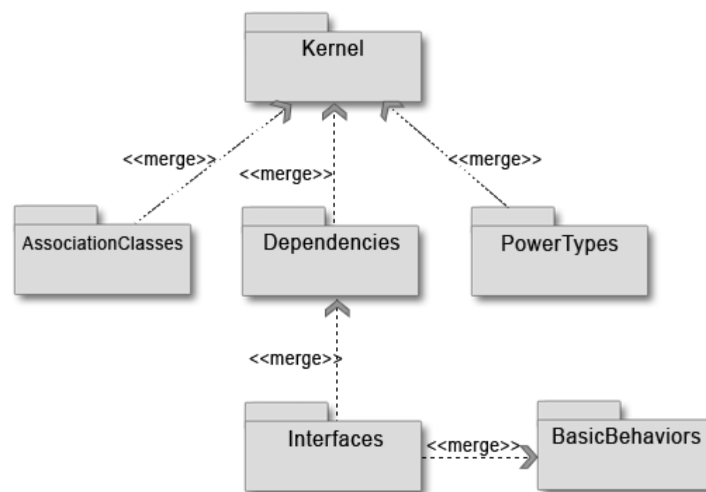


Figure 4-3: Sub-packages of the Classes package and their dependencies

Figure 4-3 shows the sub-packages of the *Classes* packages and their dependencies on the *Kernel* package.

## 4.4 The UML Superstructure Architecture

The UML Superstructure is defined within the *UML* package which is divided into a number of packages that deal with structural and behavioural modelling, as shown in Figure 4-4.

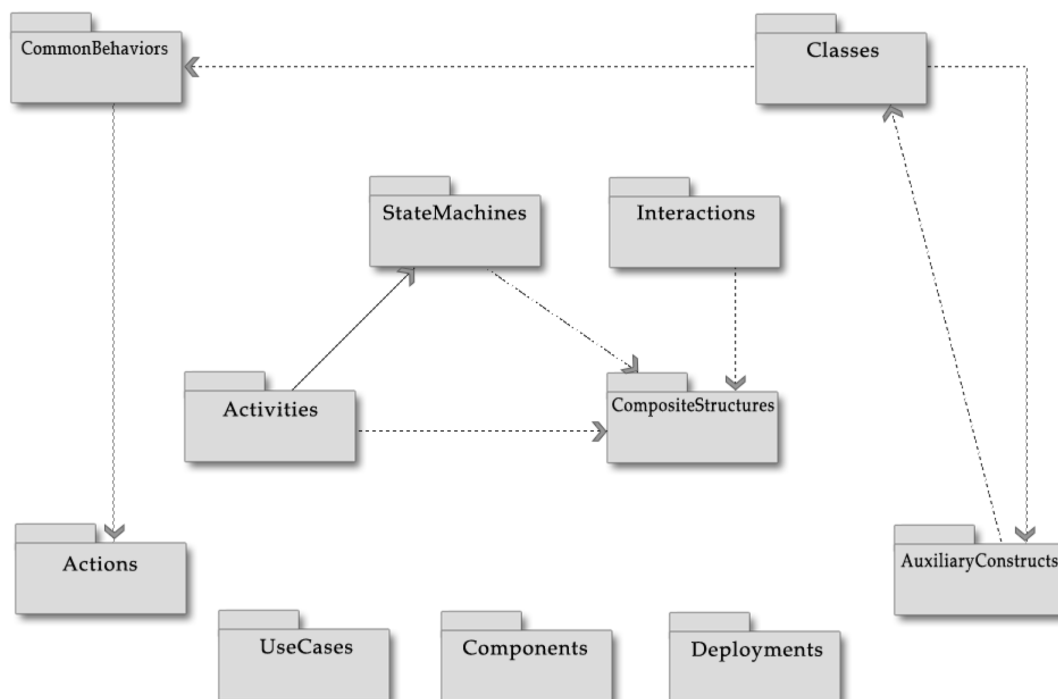


Figure 4-4: The top-level package structure of the UML

UML consists of a big number of elements organized in a package hierarchy using various associations. UML elements are instances of elements from MOF. Constraints and attributes of UML metaclasses are used to refine semantics of instantiated MOF metaclasses.

## 4.5 The UML Component model

This section provides an analysis of UML component model. Section 4.5.1 describes the *CompositeStructures* package while section 4.5.2 analyses *Components* package and the `Component` metaclass.

Some parts of UML component model are not relevant for the goals of this thesis, thus they will not be analysed. Also, graphic notation used in UML will not be explained, it is considered that the reader is acquainted with this notation.

It is important to explain the term of *classifier* that will be used in this section. A `Classifier` is an UML metaclass. According to UML superstructure specification [UMLs 07], a `Classifier` is an abstract metaclass defined as "a classification of

instances, it describes a set of instances that have features in common". A `Classifier` represents a namespace whose members can include features.

## 4.5.1 Composite structures

*CompositeStructures* package introduces constructs for modelling internal structures, interconnections and collaborations of elements. Constructs defined in this package are used in the *Components* package to describe characteristics of a `Component` metaclass.

*CompositeStructures* package contains five sub-packages, *Collaborations*, *InternalStructures*, *Ports*, *StructuredClasses* and *Actions*. Each of them is shortly described in this section.

### 4.5.1.1 InternalStructures package

The *InternalStructures* package has mechanisms for specifying structures of interconnected elements that are created within an instance of a containing classifier.

Internal subelements are defined using metaclass `Property`, and are called *parts*. Internal structure is the implementation of a component by means of a set of parts connected together in a specific way. Each part has a name and a type. A name of the part defines its role within its owning classifier. Part's type declares a classifier that will be instantiated when instancing the part's owning classifier.

Important characteristic of a part is its a *multiplicity*. The default multiplicity is one, this means that there is one part instance in a component instance. If the multiplicity is greater than one, then there will be more than one instance of a particular part in a component instance.

A structure of this type represents a decomposition of classifier and is referred to as its "internal structure".

Figure 4-5 shows an example of internal structure. A "Cruise control" component contains three parts with types: Mode, SpeedController, DistanceController and with roles: *modeLogic*, *speedCtrl*, and *distCtrl*. When instancing a Cruise control component, three instances of its parts will be created, namely one instance of Mode, one instance of SpeedController and one instance of DistanceController. For better clearness, interconnections between parts are not presented. The Cruise control system example will be discussed in SaveCCM section.

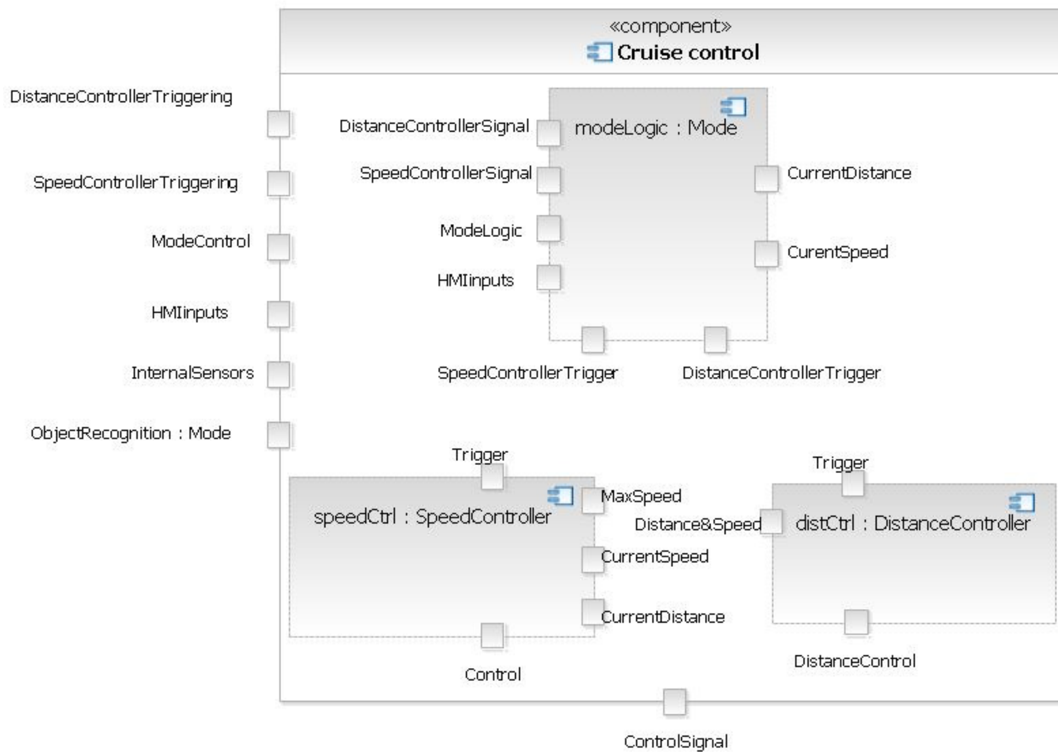


Figure 4-5: A component with internal structure

#### 4.5.1.2 Ports package

`Port` represents an interaction point of a classifier, thus enabling isolating a classifier from its environment. All communication between internals of a classifier and its environment passes through ports.

UML 2.0 enables attaching multiple ports to a component. This allows separating different communication paths and distinguishing them depending on a port on which they appear.

On Figure 4-5 a Cruise control component with several ports is presented. The ports on this example do not have any interfaces attached.

#### 4.5.1.3 Collaborations package

Collaborations allow describing the relevant aspects of the cooperation of a set of instances by identifying the specific roles that the instances will play. They help understanding mechanisms and communication paths used in design.

More details on *Collaborations* package can be found in [UMLs 07].

#### 4.5.1.4 StructuredClasses package

*StructuredClass* package is quite simple package, but very important for the UML component model. It extends the metaclass `Class` with the ability to have internal structure and to own `Ports`.

Unlike instances of `Class` form *Classes* package that cannot have ports and internal structure, instances of `Class` form *StructuredClass* package can own its communication ports and subelements.

#### 4.5.1.5 Actions package

This package adds some actions that are specific to the features from *CompositeStructure* package.

*Actions* package is, contrary to all other packages, used for modelling behaviour and not the structure, therefore it is not important for this analysis. More information about this package can be found in [UMLs 07].

### 4.5.2 Components package

The UML *Components* package contains a set of constructs (e.g. components, connectors) for designing software systems of arbitrary size and complexity. The package specifies a component as "as a modular unit with well-defined interfaces that is replaceable within its environment" [UMLs 07]. The *Components* package supports the specification of both logical components (e.g., business components, process components) and physical components (e.g., EJB components, CORBA components, COM+ and .NET components, etc.), along with the artifacts that implement them and the nodes on which they are deployed and executed.

**BasicComponents and PackagingComponents packages**

*Basic Components* package considers a component as an executable element in a system. It defines a component as a specialized class that has an external specification in the form of one or more provided and required interfaces, and an internal implementation consisting of one or more classifiers that realize its behaviour.

*Packaging Components* package focuses on defining a component as a group of elements. It extends the concept of a basic component to formalize the aspects of a component as a "building block" that may own and import a (potentially large) set of model elements.

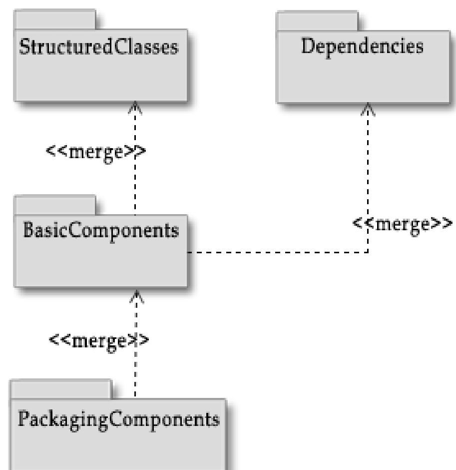


Figure 4-6: Dependencies of the Component packages

Figure 4-6 shows the dependencies between packages described in this section (dependencies to *Kernel* are not shown).

#### 4.5.2.1 Component

According to UML Superstructure Specification [UMLs 07], a component in UML is "a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment".

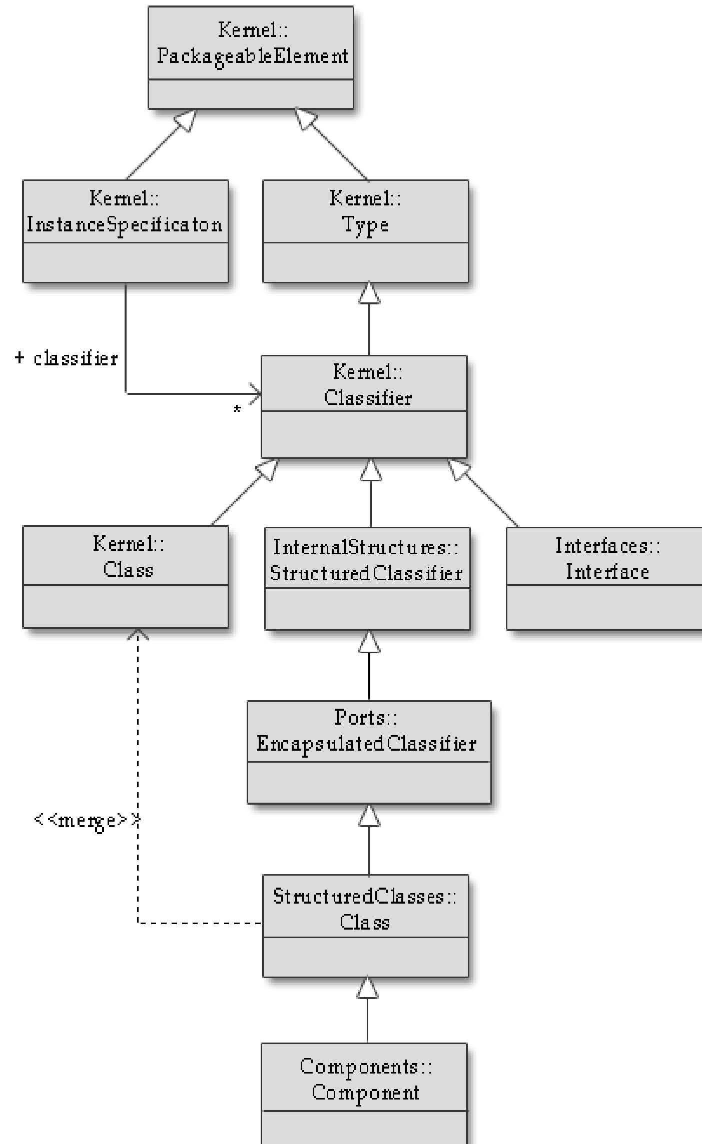


Figure 4-7: UML metaclasses included in UML 2.0 Component Model

On Figure 4-7 a part of UML 2.0 class hierarchy is shown, with an inheritance tree of metaclass `Component` from metaclass `Class`.

`Component` is a descendant of a `Class` from *StructuredClasses* package, this inheritance allows components from user models to have attributes and methods and enables participation of components in various `Generalizations` and `Associations`. A `Generalization` presents a relationship between a more

general classifier and a more specific classifier, it is used to model inheritance between elements, an `Association` is a relationship that can occur between typed instances, for more information on `Generalization` and `Association` see [UMLs 07].

In addition, since a `Class` itself is a subtype of an `EncapsulatedClassifier`, a `Component` may optionally have an internal structure and own a set of `Ports` that formalize its interaction points.

**Component realization**

The main purpose of a `Component` is to model a part of an application typed by its provisions and requirements. When instancing an application an element that realizes component's behaviour is used instead of a component. UML allows specifying realizing element of a component. Most of the instances that a designer comes across when using UML are object instances, component instances, node instances etc. Generally speaking, an instance is a feature that takes up some space in the physical world. For example, an instance of a node is a physical computer in a room, an instance of a component takes up some space on the computers file system etc, meaning that component instances are actually artifacts.

**Provisions and requirements**

A component can have any number of provided and required `Interfaces`, that are the basis for wiring components together either using `Dependencies`, or by using `Connectors`. There are two ways in UML 2.0 to model components provisions and requirements:

- Provided and required interfaces attached directly to a component. Provisions and requirements specified this way are anonymous since they do not have identifier.
- Using ports with provided and required interfaces attached. Since ports attached to a component may have a name, therefore a component specifies a named set of provisions and requirements.

Important to note is that interfaces map at multiple levels. The design-level interface which is used or realised by a component, will be maped to a implementation – level interface of a artifact that implements that component, as it is depicted on Figure 4-8.

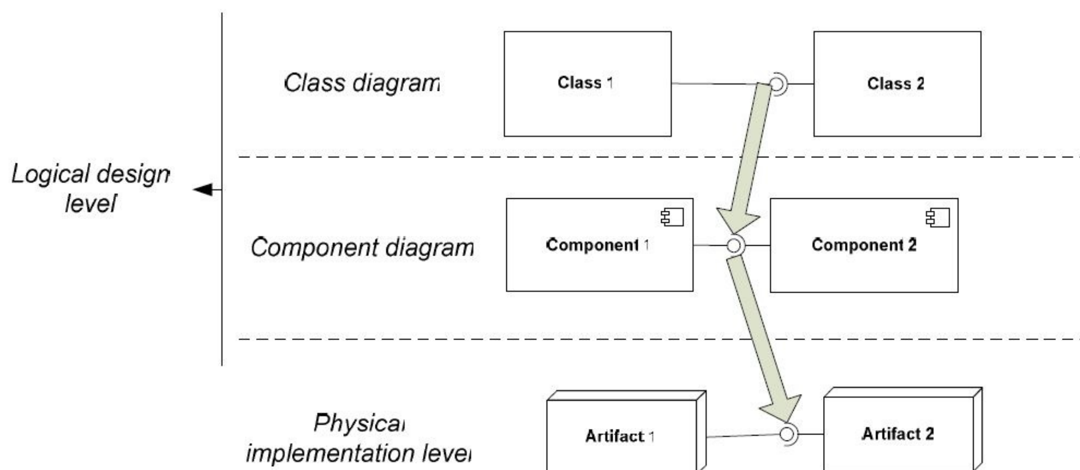


Figure 4-8: Mapping component interfaces at multiple levels

A component has an *external view* (or "black-box" view) by means of its public properties and operations, all other *internals* are hidden from component's environment. A component provides *access points* – *ports* through which it communicates with its environment.

A component also has an *internal view* (or "white-box" view) by means of its private properties and realizing classifiers. This view shows how the external behaviour is realized internally.

**Internals**

UML gives a lot of liberty in defining internal structure of a component. Not only a component, but even a class, interface, an instance of a component or a class may be internal element of a component. In UML 2.0, there are two ways of specifying an internal structure of a component:

- Using metaclass `Property`. A `Component` as a descendant of `StructuredClassifier` may have an internal structure in form of *parts* (parts are explained in Composite structures section). A component with internal structure specified using `Property` metaclass is presented on Figure 4-5.
- Using metaclass `PackageableElement`. A `Component` may contain instances of descendants of metaclass `PackageableElement`. `PackageableElement` is a base metaclass for various UML metaclasses (see Figure 4-7) thus allowing a Component to contain other components, interfaces and classes. Figure 4-9 shows an example of a Cruise control component with three subcomponents.

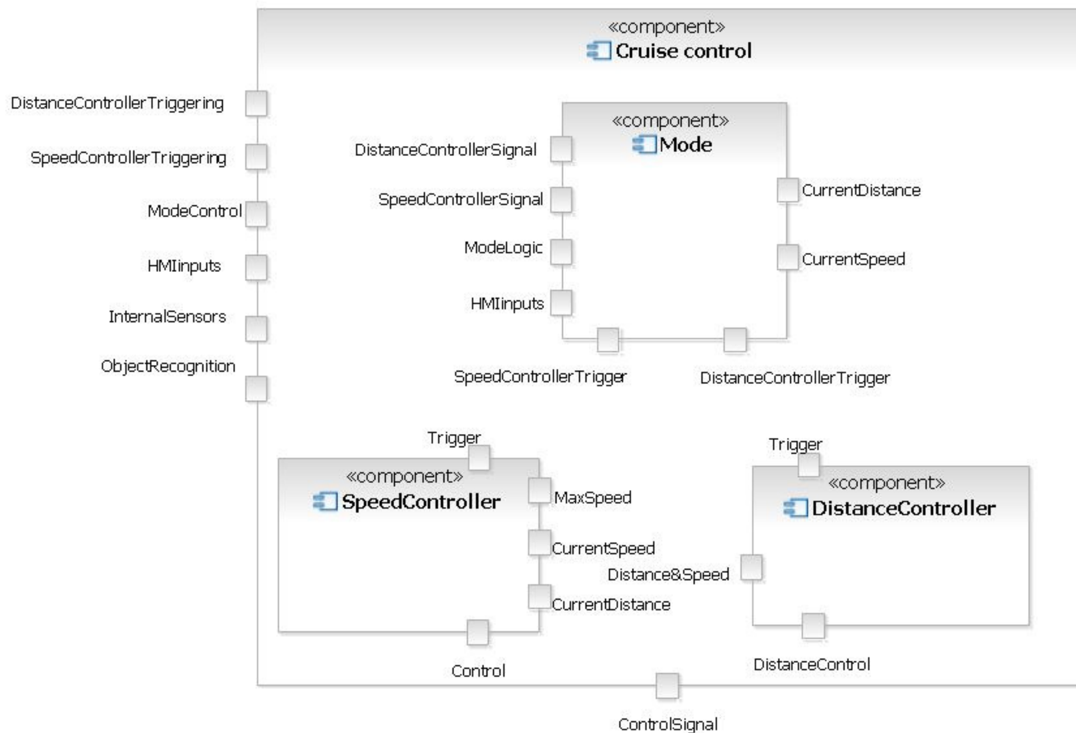


Figure 4-9: A component with packaged elements

#### 4.5.2.2 Connector

A Connector is communication relationship between two parts or two ports in a context of a component. Connectors are used to interconnect provisions and requirements of a component. Since provisions and requirements can be specified using ports or can be directly attached to a component, a connectors may be attached to either ports, interfaces or components.

UML 2.0 diferentiates two kinds of connectors:

- *Assembly connector* – may be defined from a required interface or a port to the provided interface or a port. In other words, it connects two components and their provisions and requirements. Assembly connector defines that one component provides the services which another component requires.
- *Delegate connector* – connects component and its subcomponent. It is used to connect component's provisions to its subcomponent's provisions, or component's requirements to its subcomponent's requirements. It represents the forwarding of signals, a signal that arrives at a port with a delegation connector to a subcomponent, will be passed on to that target for handling.

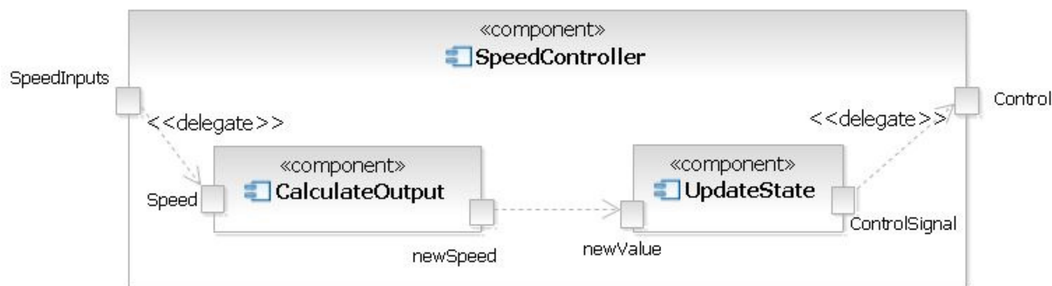


Figure 4-10: Component with two subcomponents and three connectors

Both kinds of connectors, assembly and delegate connectors are depicted on Figure 4-10. First delegate connector connects port SpeedInput and a subcomponent's Speed port, while the other delegation connects ControlSignal port of the subcomponent and Control port of owning component. Note that these two delegation connectors have opposite directions. An assembly connector on this example connects subcomponents form newSpeed port to newValue port.

## 4.6 UML Extensibility Mechanisms

The UML is broadly applicable to different types of systems, domains, methods and processes, which is why it is such a popular and broadly used language. However, even though the UML is very well defined, with increasing knowledge and experience comes the need for greater domain specialization.

UML includes special extensibility mechanisms, which can be used to define domain-specific modelling languages (DSML) that are based on UML. Extensibility mechanisms provide an ability to customize and extend the UML by adding new building blocks, creating new properties and specifying new semantics in order to tailor a modelling language to a specific problem domain in a controlled way. UML offers two ways of doing so: *first – class extension* mechanism and *profile extension* mechanism.

### 4.6.1 First – class extension mechanism

The first approach is based on a hierarchical model of UML metaclasses; it modifies an existing UML metamodel. This is handled through MOF, which sets no restrictions on what is allowed to change in a metamodel, metaclasses and relationships can be added or removed, as it is necessary. A new metaclass may be defined by inheriting from an existing UML metaclass, with constraints, meta-attributes and associations specified, thus providing desired semantics required by elements from a user model. The new metaclass becomes a part of UML metamodel and is treated as other UML metaclasses.

However, this mechanism has some drawbacks:

- Portability of user model that uses new metaclasses is limited (UML tool must be extended with the new metaclasses).
- Support of extending UML metaclass hierarchy in existing CASE (Computer Aided Software Engineering) tools does not exist.
- User model designed with non-extended UML must be redrawn to use a new defined metaclasses. In modelling CASE tools, there is no support for automatic replacement of elements, therefore this replacement is in most cases impossible due the size and complexity of user model.

UML metamodel is restricted as *read-only* that is why it is not possible to define a new base class for existing UML metaclasses. This means that it is not allowed to modify, but only to extend the UML model.

### 4.6.2 Profile extension mechanism

Profile extension mechanism uses *Profiles* package from InfrastructureLibrary (see The UML Infrastructure Architecture section). The *Profiles* package provides mechanisms that allow metaclasses from existing metamodels to be extended. The UML metamodel can be tailored for different platforms (such as J2EE or .NET) or domains (such as real-time or business process modelling).

Unlike the *first – class* approach, profiles mechanism does not call for defining a new metaclasses but for extending the existing ones. The extension mechanisms are: *stereotypes*, *tagged values* and *constrains*.

In UML 1.1, stereotypes and tagged values were used as *string – based extensions* that could be attached to UML model elements in a flexible way. The UML 2.0 specification has carried this further, by defining it as a specific *meta-modelling technique*. Stereotypes are specific metaclasses, tagged values are standard meta-attributes, and profiles are specific kinds of packages.

#### Stereotype

*"Stereotype* is a kind of model element defined in a model itself" [UMLu 07]. Stereotypes provide a way to define a virtual metaclasses from existing UML metaclasses. It does not define a new metaclass but a new kind of an existing metaclass with additional semantics.

Stereotype cannot be instantiated in a user model, it may only be applied to an existing element in a model thus creating an instance of a virtual metaclass defined by a stereotype.

The information content of stereotyped element (element from the user model with the stereotype applied) is the same as of the existing model element. This

permits the CASE tool to store and manipulate the stereotyped element the same way as it does with the existing element.

Stereotype can contain a set of constraints and tagged values.

### Constraint

*Constraints* are used to put restrictions of semantics on a new metaclass (for example, a ‘SaveClock’ component must have only one port). Constraints associated with a stereotype are applied to elements when applying a profile to a user model or when applying a stereotype to a model element.

Constraints may be written as free-form text. There is often a need to specify constraint semantics more precisely, then the UML's Object Constraint Language (OCL) can be used.

### Tagged values

*Tagged values* are name-value pairs that express properties of an element. They are an equivalent of meta-attributes of metaclasses. A tagged value has a name and a type and is owned by a stereotype. These properties are appended to elements when applying a stereotype to an element.

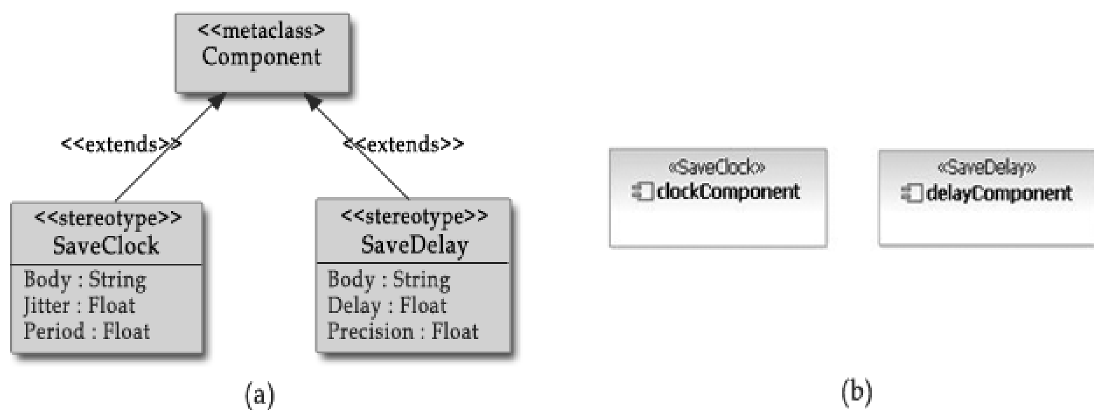


Figure 4-11: Profile example, (a) definition, (b) application

Figure 4-11 (a) shows a definition of two stereotypes `SaveClock` and `SaveDelay`, which may be applied to a metaclass `Component`. `SaveClock` stereotype has three tagged values, Body, Jitter and Period. `SaveDelay` stereotype also has three tagged values, Body, Delay and Precision.

Figure 4-11 (b) shows two components, *clockComponent* with a `SaveClock` stereotype applied, and *delayComponent* with a `SaveDelay` stereotype applied.

In four layer metamodel hierarchy, user defined profile is at the same layer as user model. Elements are instances of metaclass `Class` and `Stereotype` from *Profiles* package.

#### 4.6.2.1 Description of classes in Profiles package

*Profiles* package specifies several metaclasses in order to provide lightweight extension mechanism to the UML standard. These classes are: `Class`, `Extension`, `ExtensionEnd`, `Image`, `Package`, `Profile`, `ProfileApplication` and `Stereotype`. Some of them are shortly described in this section, for more details refer to [UMLs 07].

**Package**

In the `Package` metaclass, a package functionality is extended with an ability to indicate the profiles applied to a package. A package can have one or more `ProfileApplications` to indicate which profiles have been applied. Because a profile is a package, it is possible to apply a profile not only to packages, but also to profiles.

**Profile**

A `Profile` is a coherent set of extensions applicable to a given domain or purpose. This metaclass inherits `Package` metaclass and is a restricted form of a metamodel that must always be related to a *reference metamodel*, such as UML. A `Profile` defines a limited capability to extend metaclasses of the reference metamodel and it cannot be used without its reference metamodel.

**Stereotype**

Stereotype describes how an existing metaclass can be extended. It is a kind of `Class` and extends `Classes`.

Each stereotype S must extend at least one metaclass C. The properties of S encode the additional "semantics" of the instances of C stereotyped by S as compared to those that are not. Stereotype must always be used in conjunction with one of the metaclasses it extends.

Each stereotype may extend one or more classes through extensions as part of a profile. Similarly, a class may be extended by one or more stereotypes. Stereotype is the only kind of metaclass that cannot be extended by stereotypes.

A number of UML pre-defined stereotypes exist that apply to a component. For example, «subsystem» to model large-scale components, and «specification» and «realization» to model components with distinct specification and realization definitions etc.

#### 4.6.2.2 Characteristics of profile extension mechanism

A UML profile is a specification which is used to:

- Identify a subset of the UML metamodel that can be used to design models from a certain domain.
- Specify "well-formedness rules" beyond those specified by the identified subset of the UML metamodel ("Well-formedness rule" is a term used to describe a set of constraints written in UML's Object Constraint Language that contributes to the definition of a metamodel element).
- Specify "standard elements" beyond those specified by the identified subset of the UML metamodel ("Standard element" is a term used to describe a standard instance of a UML stereotype, tagged value or constraint).
- Specify common model elements, expressed in terms of the profile.

Developing new metamodels for new models is highly risky for any organization. Using UML profiles guarantees that the adapted models will still be legal UML models. Standard solutions exist for many industrial models such as CORBA, System on a Chip (SoC), Systems Engineering (SysML), EJB etc.

However, when developing a UML profile, a developer must keep in mind that extension, by definition, changes standard form of UML and may therefore lead to interoperability problems.

# 5. SaveComp Component Model

This section gives an overview of SaveCCM – a research model developed at Mälardalen University, Sweden. Only the SaveCCM component model will be analysed, SaveCCM Core component language and SaveCCT component technology are out of scope of this thesis. The section 5.4 provides an example of a Cruise control system to illustrate SaveCCM.

## 5.1 The SaveCCM application domain

SaveComp Component Model (SaveCCM) is a research component model for embedded systems developed within the Save project at Mälardalen Real-Time Research Centre, Dept. of Computer Engineering, MdH, Västerås, Sweden.

SaveCCM is intended for modelling vehicular systems using CBD approach, precisely, it is anticipated for designing safety – critical sub-systems responsible for controlling the vehicle dynamics, including power-train, steering, braking, etc.

The possible reason for the limited success of CBSE in the embedded systems domain is the inability of available component technologies to provide solutions that meet typical embedded application requirements, such as resource – efficiency, predictability, and safety [Åkerholm 07]. Experience has shown that for many embedded system domains efficiency in run – time resources consumption and prediction of system behaviour are more important than efficiency in the software development. Contrary to many of the current component technologies, SaveCCM focuses on predictability and analysability more than on flexibility. Except systems with predictable behaviour, SaveCCM supports the development of resource-efficient systems.

There are many similar component technologies for development of embedded systems, such as Koala and Rubus used in industry and the research technologies PECT, PECOS and ROBOCOP.

This component model is intended to be sufficiently expressive for the needs of embedded control designers, while at the same time being restricted enough to facilitate predictability, dependability, and analysis. In addition, within the SAVE project a SaveIDE tool is developed. It is a development environment for modelling embedded systems using SaveCCM.

## 5.2 SaveCCM – syntax and semantics

SaveCCM is based on a textual XML syntax (for more about this syntax refer to [Hansson 04][SaveR 07]), and a modified subset of UML 2.0 component diagram is used as a graphical notation. The semantics is formally defined by a two-step

transformation, first from the full language to a similar but simpler language called SaveCCM Core, and then into timed automata with tasks.

In SaveCCM, systems are built from interconnected elements with well-defined interfaces consisting of input and output ports. An important characteristic of SaveCCM is the distinction between data transfer and control flow, this is achieved by distinguishing two kinds of ports, *data ports* where data of a given type can be written and read, and *trigger ports* that control the activation of components. The port can also be *combined*, having both data and triggering functionality. The separation of data and control flow allows the model to support both periodic and event – driven activities (execution can be initiated by clock or external elements). In addition, due to this separation the resulting design is analysable with respect to temporary behaviour, thus allowing analysis of schedulability, response time, execution time etc.

## 5.3 Architectural elements

The main architectural elements in SaveCCM are *components*, *switches* and *assemblies*, where switches and assemblies may be considered as special kinds of components.

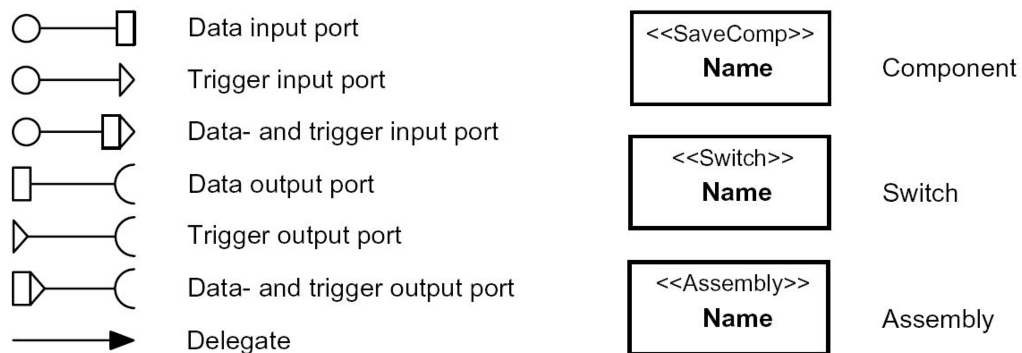


Figure 5-1: Graphical notation of SaveCCM

Graphical notation of SaveCCM is depicted on Figure 5-1.

#### Interfaces

The functional interface of every modelling element is defined by a set of ports associated to the element. SaveCCM separates input and output ports, and already mentioned data and triggering ports. Systems are built from components by connecting input ports to output ports. Ports can only be connected if their types match, i.e. identical data types are transferred.

#### Quality attributes

In addition to ports, the interface of an element may contain quality attributes each associated with a value and possibly a confidence measure. These attributes hold the information about the (worst case) execution time, reliability estimates, safety models etc. The quality attributes are used for analysis, model extraction and for synthesis. A list of quality attributes is included in the definition of components.

### 5.3.1 Components

Components are the main architectural element in SaveCCM, it is the basic unit of encapsulated behaviour. The functionality of a component is usually implemented by a single entry function in C, but it is also possible to have more complex components that consist of a number of possibly communicating tasks. Component is defined by an entry function, associated ports and optional quality attributes. The only allowed dependencies between components are the ones explicitly captured through component ports.

Each port providing data that is used as a parameter of the entry function is marked as a *bind port.*

#### Execution model

On a high level, a component is either waiting to be activated (triggered) or is executing. Initially, component is *inactive*. When all trigger input ports are activated, component has been *triggered* and it changes its state from inactive to *executing state*. The first phase of execution is *read* phase – the current value at each input data port is stored internally to ensure consistent computation. Then component performs its computations based on this inputs and possibly an internal state. After a *computation* (*execute*) phase is finished (the function has been computed, or in case of a complex component – when all tasks have finished), starts the *write* phase where the output is written to the output data ports. Finally, the component activates the outgoing trigger ports and returns to idle state by resetting input trigger ports.

This execution model, with "read – execute – write" semantics ensures that when a component is activated (triggered), the execution is independent of any concurrent activities.

#### Component realization

Each component can have any number of models associated to it. A *model* is an element which implements component's behaviour. It presents an instance of a component. A model can be an external file, or embedded as text within the element.

#### 5.3.1.1 Clock and Delay

To ensure manipulation of timing and triggering SaveCCM provides two special types of components, Clock and Delay with graphical presentation shown in Figure 5-2.

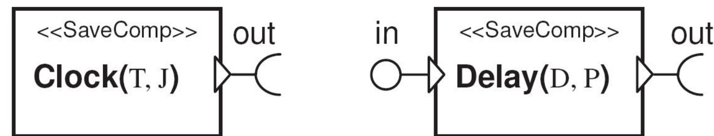


Figure 5-2: Clock and Delay components, graphical presentation

Clock is a trigger generator, it has two parameters period (T) and jitter (J). A period starts every T time units and a trigger is generated within J time units after the start of each period.

Delay is used to hold the trigger signal for some time period. It has parameters delay (D) and precision (P) It will delay within D and D+P time units from receiving a trigger until generating a trigger.

#### 5.3.1.2 Composite component

Composite component is a special kind of component where the functionality of the component is specified by an internal composition instead of using an entry function.

The execution model is similar to the execution of the basic component. It becomes active when all its trigger ports are activated. In the read phase, data is transferred and subcomponents are activated. The execute phase will perform computations of internal components, until no internal component is triggered or active. Finally, the write phase follows and then the component returns to idle state.

### 5.3.2 Switches

Switch is used to change component interconnection structure, this allows conditional transfer of data or triggering between components. This conditional transfer can be specified statically for pre-runtime static configuration, or dynamically.

Switch uses a set of connection patterns and connection conditions. Connection pattern is mapping from input port to output port, thus specifying a way of connecting the switch ports. Each connection pattern is guarded by one connection condition, which is a logical expression over the data available at the input data or combined ports of the switch, defining the condition under which that pattern is active. Switch input port that occurs in some connection condition is marked as a *set port*.

Unlike components, switches can not be activated, they respond instantly at the arrival of data or trigger signal on some of the input ports. Switch does any other computation except evaluation of connection conditions.

To use switch for pre-run-time static configuration, fixed values need to be statically bound to the data or combined input ports. Switches can also be used dynamically e.g. for specifying modes and mode-switches, each mode corresponding to a specific static configuration. By changing the port values at run-time, a new configuration can be activated, thus achieving a mode-shift.

### 5.3.3 Assemblies

Assembly is an encapsulated subsystem. Its internal structure, subcomponents and interconnections are hidden from its environment and can be accessed only through assembly ports.

Contrary to components, assembly can not be triggered. Data and trigger signals are immediately processed. Therefore, assembly should not be considered as a component composition mechanism but as way of naming a collection of components and hiding its internal structure.

### 5.3.4 Ports

As it is already mentioned, SaveCCM differentiates several kinds of ports. Referring to port's direction, port can be input or output port, and referring to the signal type, port can be data, trigger or combined port. Component input ports, the output ports of the whole system, and switch set ports, are one-place buffers with overwrite semantics. The other ports are only conceptual interaction points through which data passes immediately.

Every data port has its *type* and optionally an initial value.

An *external port* is a special kind of port that is not connected with any other port in the model, but has an extra label mapping it to some external entity (for example to I/O-ports, interrupts, and real-time database pointers). The format of this label depends on the external entity to which the port is mapped. External ports are not allowed internally within a composite component.

### 5.3.5 Connections

Connections define how data and control can be transferred between components. SaveCCM offers two types of connections: *immediate* and *complex*. In SaveCCM reference manual [SaveR 07], immediate connection is defined as a "loss – less, atomic migration of data or trigger signals from one port to another, as would typically be the case between components located on the same physical node". However, in distributed systems and in early stages of modelling, another, more general connection is convenient – a complex connection. A complex

connection represents a link with a possible delay or information loss. The characteristics of a complex connection are explicitly modelled by a timed automaton.

Connections in SaveCCM model may only connect the ports of the corresponding types. This means that trigger ports can only be connected to trigger ports, and data ports can only be connected to data ports with the additional demand for data type compatibility. Combined ports are treated as union of one data and one trigger port. In addition, a connection from an assembly or composite input port to an input port of an internal element, or from an internal output port to an assembly or composite output port, are referred as *delegation*.

## 5.4 The Cruise Control Example

The Adaptive Cruise Controller (ACC) [Åkerholm 05] has been a recurring example throughout the development of SaveCCM. The purpose of this running case-study has been to continuously evaluate and improve the component model. Therefore a simple design of ACC [Hansson 04] is demonstrated here as an example of use of SaveCCM.

The ACC system helps the driver to keep a desired speed and a safe distance to a preceding vehicle. The ACC automatically adapts the distance depending on the speed of the vehicle in front, while keeping the gap large enough to avoid collisions. The SaveCCM model of ACC system is presented on Figure 5-3, and it can be divided to three parts: input, control and actuate.

There are three different sources of input to ACC: the Human Machine Interface (HMI) (e.g., desired speed and on/off status of the ACC system), the internal vehicular sensors (e.g., current speed), and the external vehicular sensors (e.g., distance to the vehicle in front). Each of the inputs is modelled as one component. The outputs can be divided to two categories, the HMI outputs (returning driver information about the system state) and the vehicular actuators for controlling the speed of the vehicle. The application also has two trigger frequencies, 10 Hz and 50 Hz.

The ACC system is designed as a SaveCCM assembly ("ACC Application" on Figure 5-3) built from three basic components (Object recognition, ACC Mode Logic and HMI outputs) and one sub-assembly (ACC controllers). Internal design of ACC controllers sub-assembly is provided by two components (Distance Controller and Speed Controller) and one switch (Mode). The detail description of functionality of those elements can be found in [Åkerholm 05], [Hansson 04], [Åkerholm 07].

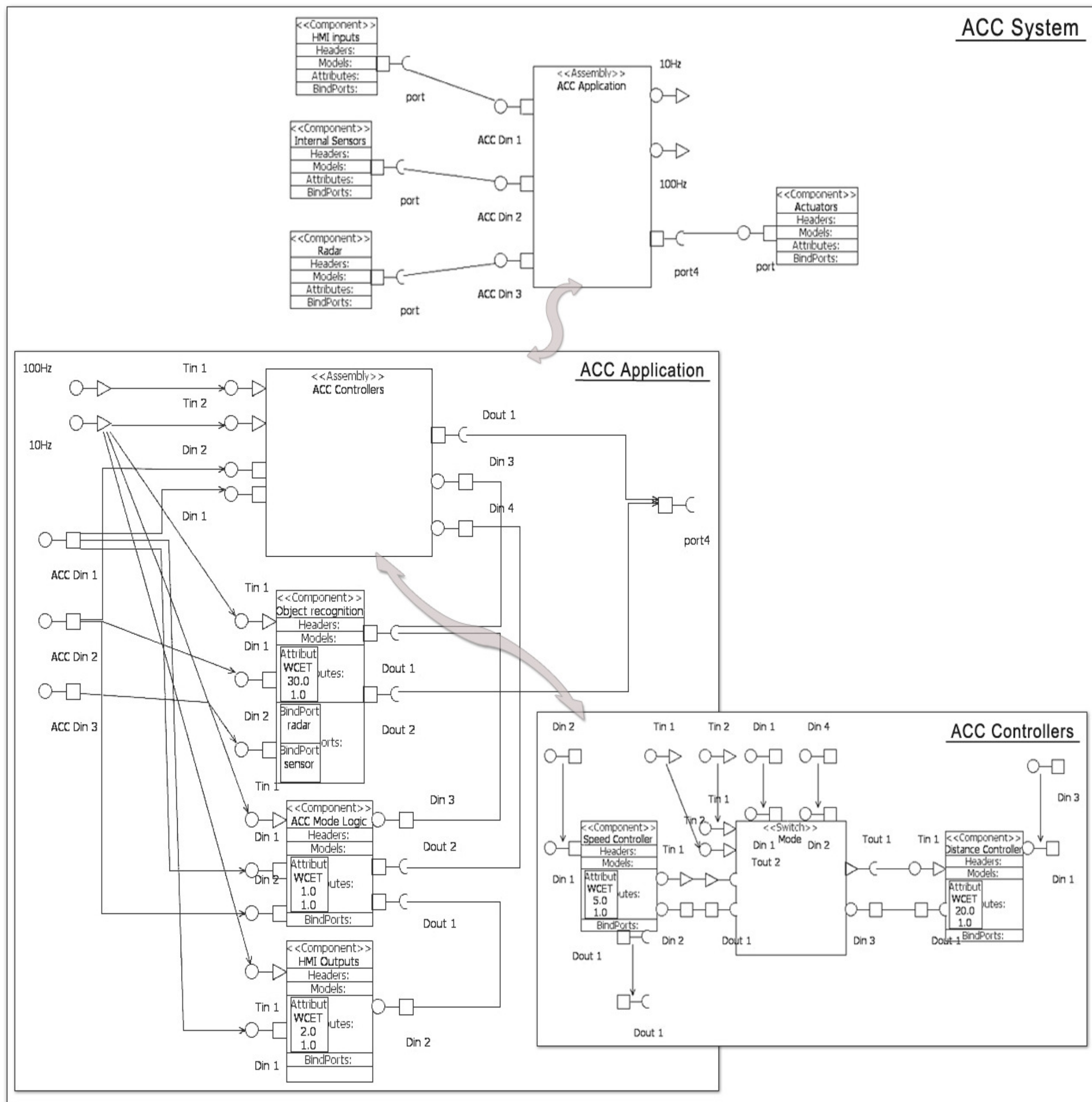


Figure 5-3: SaveCCM model of ACC system

This example shows that expressiveness of the SaveCCM component model is sufficient for efficient application of component- based principles in the domain of vehicular systems. It can seamless and easily support typical requirements that arise when designing advanced vehicular functionality, e.g., connections with data, triggering and both, assemblies, feedback, and mode changes.

# 6. SaveUML profile

This section describes a formal way of representing the SaveCCM component modelling language using the UML 2.0. UML representation of the SaveCCM is achieved through a mapping of SaveCCM component model elements to UML elements. This section describes this mapping in the form of a UML profile named *SaveUML*.

## 6.1 Background

UML inherent expressiveness allows users to model everything from enterprise information systems and distributed web-based applications to real-time embedded systems. It may be used to visualize, specify, construct, and document the artifacts of a system. All these qualities are the reason why UML became a *de facto* standard for modelling. On the other hand, there is still a need for specialization of modelling languages for specific problem domains. In previous section one research model - SaveCCM was presented. In order to join those two techniques, a mapping from UML 2.0 component model to SaveCCM model can be created. This mapping can be achieved using one of the UML extensibility mechanisms – creating a UML profile.

Considering that UML profiles are a standard UML extension mechanism (see section 4.6.2) and are therefore a part of UML's metamodel, they are as widely recognized as UML itself and should be supported by all standard modelling CASE tools. This possibility of customizing UML for specific domain purposes while remaining in boundaries of the UML standard and keeping the possibility of using UML CASE tools, presents a reasonable motivation for customizing and using UML instead of a specific modelling language.

## 6.2 The process of mapping UML to SaveCCM

The process of mapping SaveCCM language elements to UML 2.0 elements consists of three phases.

- *Identification of SaveCCM and UML language elements*. Before creating a profile, the detail analysis of UML 2.0 component model and SaveCCM model must be accomplished. This analysis enables survey of similarities between component models and identification of compatible elements. Also it is important to indentify all elements from SaveCCM that need to be translated to UML elements and corresponding UML elements that can be used for mapping.
- *Identification of SaveCCM language constraints*. Designing SaveCCM elements with UML 2.0 elements brings up various problems resulting

from a strict syntax of SaveCCM and universality of UML 2.0 specification. Therefore a set of constraints must be created to refine UML 2.0 component model semantics to be suitable for designing SaveCCM modelling elements.

- *Translation of previously identified elements* during which a suitable UML element is found for every SaveCCM language elements and it is then further customized through the use of necessary stereotypes, properties and constraints.

The result of this process is the profile specification (section 6.3) which describes the generated UML profile as a list of its stereotypes, basic UML elements they extend and source SaveCCM elements they represent. This specification does not describe the translation process but only describes the final result, which is the profile specification. The profile specification is a general profile description and needs to be implemented in a design tool in order to be used for a concrete model design of SaveCCM models in UML.

### 6.2.1 Specific design decisions

During the process of creating SaveUML profile, after a comprehensive analysis of both component models, several design decisions were made.

#### 6.2.1.1 Components

Since the SaveCCM is intended for modelling of component – based systems, the basis for main architectural elements in SaveCCM is a component. SaveCCM introduces three main architectural elements; component, assembly and switch. In addition, three subtypes of SaveCCM component are defined; clock, delay and composite component. All together, this makes six different kinds of a component in context of CBD.

UML 2.0 component model provides only one kind of a component. For needs of SaveCCM it is necessary to distinguish between six kinds of a component, therefore it is required to define six virtual metaclasses – stereotypes that will extend the UML `Component` metaclass. This will allow applying those stereotypes to components within the user model to distinguish between SaveCCM architectural elements.

#### 6.2.1.2 Subcomponents

SaveCCM offers two elements that may have an internal structure defined, assembly and composite component.

In UML 2.0 component model, there are two ways of specifying an internal structure of a component (section 4.5.1.1 and section 4.5.2.1):

- Using metaclass `Property`.

- Using metaclass `PackageableElement`.

The first version of SaveUML profile [SPprof 08] created within the SaveUML project [SP web], presumed that subcomponents will be defined using UML metaclass `Property`. The advantage of this approach is that subcomponents are defined outside the owning element *at one place* and then referred to as a *type* of the part (reminder: internal subelements that are defined using metaclass `Property` are called *parts*). This way, as many *parts* as needed can refer to the same component at the same time. The changes made to the component will reflect to all parts referring that component.

However, this approach has a drawback. Since a metaclass `Property` is not a subtype of an `EncapsulatedClassifier`, it may not have an internal structure. This means that it is not possible nesting the components to arbitrary depth. A component can have only one – level internal structure (the only solution to this problem is to define internals of a subcomponent outside the owning component). In addition, using `Property` for defining subcomponents is not semantically correct in regard to the CBD.

Considering above-mentioned disadvantages of using `Property` metaclass, the latter approach - using metaclass `PackageableElement`, was chosen. This method enables a component to contain other components. When using a component as a subcomponent, it is possible to define a hierarchical composition of components and its nested subcomponents to arbitrary depth at one place with no need to define internals of subcomponents outside the owning component. Such a definition of subcomponents is called *embedded definition* of components.

#### 6.2.1.3 Interfaces

In SaveCCM the functional interface of every modelling element is defined by a set of ports associated to the element.

UML 2.0 provides an `Interface` metaclass for this purpose and it supports two ways of specifying provided and required interfaces (detail description is provided in section 4.5.2.1). `Interface` provides a way to partition and characterize groups of *properties* and *operations* that a component possesses.

By reason of different semantics of interface in SaveCCM and UML, it is decided not to use UML interfaces in SaveUML profile. It is supposed that when modelling a user model in UML using SaveUML profile, interface of a component will be determined by the ports owned by a component, as it is done in SaveCCM.

## 6.3 The SaveUML profile specification

Mapping from UML to SaveCCM requires refinement of UML semantics and creating new metaclasses. As it is described above the most suitable way of extending UML for this problem is creating a UML profile, therefore a SaveUML profile was created. This section provides its specification.

The current release of SaveUML profile is release 0.8, version 234. The SaveUML profile developed within the SaveUML project had the latest release 5, version 42. For better clearness, when referring to SaveUML profile developed within the SaveUML project, the term "SaveUML profile release 5" will be used.

A description of each stereotype is broken down into sub sections:

- The heading gives a short description of the concept behind the stereotype.
- *Base classifier*: the UML metaclass that this stereotype extends.
- *Tagged values*: the list of tagged values that this stereotype defines. For each tagged value its name, type and a short description is provided.
- *Constraints*: a list of OCL constraints applied to this stereotype. For each constraint its name and a short description is provided.

### 6.3.1 Imported libraries

Initially every UML profile imports UML metamodel, which is the one being extended by the profile. Optionally a profile can import several other libraries containing data types that can be used in order to define the type of each tagged value within that profile. The most commonly used library is the UML *PrimitiveTypes* package (described in section 4.3), which defines *Integer*, *Boolean*, *String* and *UnlimitedNatural* data types. SaveUML profile imports this library however, in order to define properties such as period and jitter (clock components), delay and precision (delay components) and credibility (attributes), used in SaveCCM, *float* data type is needed. Therefore, SaveUML profile imports one more library – *JavaPrimitiveTypes library* which provides *Float* data type.

### 6.3.2 Profile constraints

Due to the universality of UML and a strict syntax of SaveCCM, a number of constraints had to be applied to created stereotypes. For defining those constraints the Object Constraint Language (OCL) was used.

OCL [OCLs 06] is a formal language used to describe expressions on UML models. It is maintained by OMG and its current version is version 2.0.OCL expressions typically specify invariant conditions that must hold for the system being modelled or queries over objects described in a model. OCL is a pure specification language, when the OCL expressions are evaluated, they do not

have side effects (i.e., their evaluation cannot alter the state of the corresponding executing system).

The SaveUML profile defines a number of constraints that are listed in this specification. This specification only lists the name of the constraint and the explanation of its purpose. OCL definitions of constraints are not listed.

### 6.3.3 Description of stereotypes

#### 6.3.3.1 SaveAssembly

This stereotype is used to model an assembly in SaveCCM domain. An assembly can generally be described as an encapsulated subsystem. It is used in system modelling as a black box with some predefined functionality that can be used as it is. Although it might not dynamically correspond to the behaviour of a component, a UML component was still chosen as the best base classifier.

The subelements of SaveAssembly can be other components and are added using metaclass `PackageableElement`.

**Base classifier:** `Component`

**Tagged values:** This stereotype does not have tagged values defined.

**Constraints:**

[1] *AssemblyAttributes*: UML enables a `Component` to contain several types of attributes: ports, attributes and operations. SaveAssembly component can only own ports. No other attributes are allowed.

[2] *AssemblyConnections*: No possible connections except the ones explicitly captured by the ports are allowed.

[3] *AssemblyInterfaces*: It is not allowed to use interfaces to define provisions and requirements. Provisions and requirements are determined by the ports owned by the SaveAssembly component as it is defined in SaveCCM.

[4] *AssemblyPackagedElemets*: Internal structure of a SaveAssembly component may be defined using packaged elements. The only allowed packaged elements are `Components` (because metaclass `Component` is the base classifier for all stereotypes that represent SaveCCM main architectural elements). Using parts is not allowed.

[5] *AssemblyPorts*: All ports owned by a SaveAssembly component must have appropriate stereotype applied. By appropriate stereotype it is considered to be one of the stereotypes defined by SaveUML profile that are used to model SaveCCM ports. If some of the ports would not have stereotype applied it would be ambiguous what SaveCCM port it represents (as SaveCCM distinguishes several types of ports).

[6] *NumberOfStereotypes*: It is not allowed to apply more than one stereotype to the same modelling element.

#### 6.3.3.2 SaveAttribute

Attribute is a part of a components (abstract) interface in the SaveCCM domain and is used for defining a components quality information. SaveUML profile release 5 used the `Class` metaclass as the base classifier for SaveAttribute. However, the term of *class* is a very comprehensive abstraction, and therefore is not suitable for presenting a component quality attributes. For this reason, it is decided to use `Property` metaclass, which is usually used for modelling component attributes in UML.

**Base classifier:** `Property`

**Tagged values:** Defined tagged values are equivalent to the properties existing in SaveCCM, and those are:

- *Type : String*
- *Value : Float*
- *Credibility : Float*

**Constraints:**

[1] *PropertyConnectors*: It is not allowed to use any connectors for connecting a components attribute with other modelling elements.

[2] *PropertyNumberOfStereotypes*: see the SaveAssembly constraint [6].

#### 6.3.3.3 SaveBindPort

In SaveCCM, *bind ports* are used to bind a component's port with a parameter of the component's entry function. They can be looked upon as a property of a component. Therefore, SaveUML profile uses `Property` metaclass as the base classifier for SaveBindPort stereotype.

Each SaveBindPort element is added to a SaveComponent as its attribute.

**Base classifier:** `Property`

**Tagged values:**

- *Argument : String*. A name of the parameter of the entry function
- *Bind : String*. A name of the port that is bound. There is a shortage of this tagged value in comparison with the "Bind" property used in SaveIDE. SaveIDE offers a dropdown list of available ports that can be bound. When using SaveUML profile, user has to enter the name of the port instead of choosing from a dropdown list. This creates a possibility for entering illegal values (e.g., non-existing ports), therefore a constraint that checks the correctness of entered value had to be defined.

**Constraints:**

[1] *BindPortExist*: Checks if the port stated in "Bind" tagged value exists (checks that the component owns the port with the stated name).

[2] *BindPortType*: Trigger ports, data output ports and combined output ports are not allowed to be bound.

[3] *PropertyConnectors*: see the SaveAttribute constraint [1].

[4] *PropertyNumberOfStereotypes*: see the SaveAssembly constraint [6].

#### 6.3.3.4 SaveClock

This stereotype is intended for modelling the SaveCCM Clock component. Since Clock is a kind of a component, the `Component` metaclass is used for the base classifier.

**Base classifier:** `Component`

**Tagged values:** Defined tagged values are equivalent to the properties existing in SaveCCM, and those are:

- *Body : String*
- *Jitter : Float*
- *Period : Float*

**Constraints:**

[1] *ClockAttributes*: UML enables a `Component` to contain several types of attributes: ports, attributes and operations. SaveClock component may own ports and quality attributes (properties with the SaveAttribute stereotype applied). Other attributes are not allowed.

[2] *ClockPorts*: According to SaveCCM specification, a Clock component has only one port – output trigger port. This constraint checks if that condition is fulfilled.

[3] *ComponentConnections*: see the SaveAssembly constraint [2].

[4] *ComponentInterfaces*: It is not allowed to use interfaces to define provisions and requirements of a component. Provisions and requirements are determined by the ports owned by the component as it is defined in SaveCCM.

[5] *ComponentPackagedElements*: This component is not allowed to have an internal structure (neither using packaged elements neither using parts).

[6] *NumberOfStereotypes*: see the SaveAssembly constraint [6].

#### 6.3.3.5 SaveCombinedInPort

UML port corresponds to a SaveCCM port semantically because they both represent components communication points. In both cases, ports encapsulate all interaction between the components environment and its internal structure. In SaveCCM model ports are connected to each other directly by the use of

connections. Although UML ports are intended to be used through UML interfaces attached to them, they can nevertheless be used without interfaces, i.e. they can be connected to each other directly which is equivalent to the SaveCCM model.

Considering different types of ports with respect to port direction and type, it was reasonable to design a separate port stereotype for each direction and type combination which resulted in altogether six port stereotypes. The rationale for using UML port as the base classifier is the same for all port stereotypes. Stereotypes differ in their connection possibilities which are specified through constraints.

This stereotype is used for modelling combined input port.

**Base classifier:** `Port`

**Tagged values:**

- *Type : String*. A type of data.
- *Value : String*. Initial value of data (optionally).
- *External : String*. A label mapping the port to external entity, in case a port is an *external port*.
- *SetPort : Boolean*. Indicates if a port is used in a switch condition. If it is set to 'true', then the port is used in a switch condition (a consequence is that a component that owns this port must be a switch component – component with SaveSwitch stereotype applied). Default value is set to 'false'.

**Constraints:**

[1] *NumberOfStereotypes*: see the SaveAssembly constraint [6].

[2] *OwnerComponentStereotype*: This constraint ensures that an owner component (the component that owns the port) has an appropriate stereotype applied. By appropriate stereotype it is considered to be one of the stereotypes defined by SaveUML profile that are used to model SaveCCM main architectural elements.

[3] *PortConnections*: As SaveCCM offers only two types of connections (see section 5.3.5), the port can be connected to other port with only two kinds of connectors. Those two kinds are the ones used by SaveUML profile for modelling SaveCCM connections; `Usage` and `Dependency`. No other connections are allowed.

[4] *PortConnectionStereotype*: To ensure that connections used for connecting the port to other ports have the appropriate stereotype applied, this constraint checks if the connections have one of two allowed stereotypes; SaveConnection or SaveDelegation.

[5] *PortsExternal*: External ports are not allowed internally within a composite component or assembly. If an "External" tagged value is not empty, then this constraint checks if the owner component is composite or assembly component (component with SaveComposite or SaveAssembly stereotype applied).

[6] *PortsInterfaces*: It is not allowed to attach interfaces to ports. Provisions and requirements of a component are not defined by interfaces attached to it or to its ports. Provisions and requirements are determined by the ports owned by the component as it is defined in SaveCCM.

[7] *PortsSetPort*: This constraint checks the condition that the port can be a *set port* only if its owner component is a switch component (component with SaveSwitch stereotype applied). Otherwise, the "SetPort" tagged value must be set to ‘false’.

[8] *SetPortExist*: In case a "SetPort" is ‘true’, this constraint checks if the port is used in a switch condition, otherwise it can not be a *set port*.

#### 6.3.3.6 SaveCombinedOutPort

This stereotype is used for modelling SaveCCM combined out ports. The rationale behind using UML `Port` as the base classifier for the SaveCCM combined output port element is the same as with the SaveCombinedInPort (see section 6.3.3.5).

**Base classifier:** `Port`

**Tagged values:**

- *Type : String*. A type of data.
- *Value : String*. Initial value of data (optionally).
- *External : String.* A label mapping the port to external entity, in case a port is an *external port*.

**Constraints:**

[1] *NumberOfStereotypes*: see the SaveAssembly constraint [6].

[2] *OwnerComponentStereotype*: see the SaveCombinedInPort constraint [2].

[3] *PortConnections*: see the SaveCombinedInPort constraint [3].

[4] *PortConnectionStereotype*: To ensure that connections used for connecting the port to other ports have the appropriate stereotype applied, this constraint checks if the connections have one of three allowed stereotypes; SaveConnection, SaveDelegation or SaveToComplex.

[5] *PortsExternal*: see the 6.3.3.5SaveCombinedInPort constraint [5].

[6] *PortsInterfaces*: see the SaveCombinedInPort constraint [6].

#### 6.3.3.7 SaveComplexConnection

A complex connection in the SaveCCM domain is an upgrade of the simple connection. A complex connection can have a model defining its behaviour, the same as with a component. Due this characteristic, it is reasonable to use the `Component` as a base classifier for the complex connection and to allow any number of models to be attached to it. This component will however have no ports

and only special To and From connections are allowed to be used for connecting. These connections are attached directly to the component without ports.

**Base classifier:** `Component`

**Tagged values:**

- *ComplexFrom : String*. The name of the supplier port (the port that is a starting point of the complex connection).
- *ComplexTo : String*. The name of the client port (the port that is an end point of the complex connection).

**Constraints:**

[1] *ComplexConnections*: The only allowed connectors attached to this component are the ones used for modelling SaveFromComplex and SaveToComplex connections.

[2] *ComplexFromNumber*: Since ComplexConnection component together with SaveFromComplex and SaveToComplex connectors, is used for modelling a single complex connection from SaveCCM domain, it is allowed to attach only one SaveComplexFrom connection to this component.

[3] *ComplexFromStereotype*: This constraint checks if connectors attached to this component have an appropriate stereotype applied (SaveFromComplex).

[4] *ComplexPorts*: As it is explained above, ComplexConnection component is not allowed to own any ports.

[5] *NumberOfStereotypes*: see the SaveAssembly constraint [6].

#### 6.3.3.8 SaveComponent

Using UML `Component` as the base classifier for the SaveComponent stereotype is the most logical solution considering the structure and semantics of both elements. Both a component from the SaveCCM domain and UML component hide their implementation behind one or more interfaces which handle all of the communication with the component. The interface in the SaveCCM domain is an abstract concept which is used to describe a set of ports and attributes of a component. On the other hand, interface of a UML component is a formal element connected to a port on a component which groups a set of operations that the component provides or requires. This semantically difference does not present a problem for using UML component to extend the SaveCCM component and component interfaces are therefore not used in the SaveProfile. Therefore a UML component with its ports corresponds well to a SaveCCM component with its ports.

**Base classifier:** `Component`

**Tagged values:**

- *Entry : String*. The name of an entry function that realizes the component's behaviour.
- *Filename : String*. The path to the file containing the entry function.

**Constraints:**

[1] *ComponentAttributes*: UML enables a `Component` to contain several types of attributes: ports, attributes and operations. SaveComponent component may own ports, quality attributes (properties with the SaveAttribute stereotype applied) and bind port descriptions (properties with the SaveBindPort stereotype applied). Other attributes are not allowed.

[2] *ComponentConnections*: see the SaveAssembly constraint [2].

[3] *ComponentIntefraces*: see the SaveClock constraint [4].

[4] *ComponentPackagedElements*: see the SaveClock constraint [5].

[5] *ComponentPortNames*: This constraint checks if all ports owned by a component have a unique name. This is important in order to use bind port descriptions where the name of the port has to be entered (see section 6.3.3.3).

[6] *ComponentPorts*: All ports owned by a component must have appropriate stereotype applied. By appropriate stereotype it is considered to be one of the stereotypes defined by SaveUML profile that are used to model SaveCCM ports. If some of the ports would not have stereotype applied it would be ambiguous what SaveCCM port it represents (as SaveCCM distinguishes several types of ports).

[7] *NumberOfStereotypes*: see the SaveAssembly constraint [6].

#### 6.3.3.9 SaveComposite

SaveComposite stereotype represents a SaveCCM composite component. The rationale for using the UML `Component` as the base classifier for the SaveCCM composite component is very similar to the situation with the basic component. The difference between the basic and the composite component is in the internal structure, i.e. components realization. While simple components are mostly realized by a single programming language function, composite components have their internal structure built from other components, connections etc.

The subelements of SaveComposite can be other components and are added using metaclass `PackageableElement`.

**Base classifier:** `Component`

**Tagged values:** This stereotype does not have tagged values defined.

**Constraints:**

[1] *ComponentConnections*: see the SaveAssembly constraint [2].

[2] *ComponentInterfaces*: see the SaveClock constraint [4].

[3] *ComponentPorts*: see the SaveComponent constraint [6].

[4] *CompositeAttributes*: UML enables a `Component` to contain several types of attributes: ports, attributes and operations. SaveComposite component can only contain ports and quality attributes (properties with the SaveAttribute stereotype applied). No other attributes are allowed.

[5] *CompositePackagedElements*: Internal structure of a SaveComposite component may be defined using packaged elements. The only allowed packaged elements are `Components` (because metaclass `Component` is the base classifier for all stereotypes that represent SaveCCM main architectural elements). Using parts is not allowed.

[6] *NumberOfStereotypes*: see the SaveAssembly constraint [6].

#### 6.3.3.10 SaveConnection

A connection in the SaveCCM domain is used to connect an output port to an input port if their types and data types are conformant. The connection direction must be from the output to the input port. It can therefore be stated that the output port uses the input port either to send it data in the case of data ports, signals in the case of trigger port or both data and signals in the case of combined ports.

Because of this relationship between an output and an input port, it is reasonable to use a UML usage relationship for modelling SaveCCM connection. The usage relationship indicates a dependency in which one element (output port) depends on (requires) the presence of another element (input port) for achieving its normal functionality.

**Base classifier:** `Usage`

**Tagged values:**

- *Source : String*. The source of the connection – the starting port.
- *Destination : String*. The destination of the connection – the ending port

**Constraints:**

[1] *ConnectionComplex*: This connector can not be used for connecting SaveComplex component (SaveFromComplex and SaveToComplex connections are intended for this purpose).

[2] *ConnectionComposite*: This connector can not be used for connecting a component to its subcomponent (SaveDelegation is intended for this purpose).

[3] *ConnectionConformance*: This constraint checks the type conformance of the connected ports. The type conformance condition is the same as in SaveIDE tool. Output trigger ports can only be connected to input trigger ports. Output data and ports can be connected to input data ports if their data types match (the value of their "Type" tagged values must be equal). Combined output ports can be

connected to trigger input ports or to data input ports if their data types match (the value of their "Type" tagged values must be equal).

[4] *ConnectionCyclic*: It is not allowed to have cyclic connections except for the assembly component.

[5] *ConnectionEndPort*: Checks if the ending port of the connection is an input port (direction of the connection must be from an output to an input port).

[6] *ConnectionStartPort*: Checks if the starting port of the connection is an output port (direction of the connection must be from an output to an input port).

[7] *NumberOfStereotypes*: see the SaveAssembly constraint [6].

#### 6.3.3.11 SaveDataInPort

This stereotype is used for modelling SaveCCM data in ports. The rationale behind using UML `Port` as the base classifier for the SaveCCM data input port element is the same as with the SaveCombinedInPort (see section 6.3.3.5).

**Base classifier:** `Port`

**Tagged values:** For a description of each tagged value see SaveCombinedInPort stereotype (section 6.3.3.5).

- *Type : String*.
- *Value : String*.
- *External : String*.
- *SetPort : Boolean*.

**Constraints:**

[1] *NumberOfStereotypes*: see the SaveAssembly constraint [6].

[2] *OwnerComponentStereotype*: see the SaveCombinedInPort constraint [2].

[3] *PortConnections*: see the SaveCombinedInPort constraint [3].

[4] *PortConnectionStereotype*: see the SaveCombinedInPort constraint [4].

[5] *PortsExternal*: see the SaveCombinedInPort constraint [5].

[6] *PortsInterfaces*: see the SaveCombinedInPort constraint [6].

[7] *PortsSetPort*: see the SaveCombinedInPort constraint [7].

[8] *SetPortExist*: see the SaveCombinedInPort constraint [6].

#### 6.3.3.12 SaveDataOutPort

This stereotype is used for modelling SaveCCM data out ports. The rationale behind using UML `Port` as the base classifier for the SaveCCM data output port element is the same as with the SaveCombinedInPort (see section 6.3.3.5).

**Base classifier:** `Port`

**Tagged values:** For a description of each tagged value see SaveCombinedOutPort stereotype (section 6.3.3.6).

- *Type : String*.
- *Value : String*.
- *External : String*.

**Constraints:**

[1] *NumberOfStereotypes*: see the SaveAssembly constraint [6].

[2] *OwnerComponentStereotype*: see the SaveCombinedInPort constraint [2].

[3] *PortConnections*: see the SaveCombinedInPort constraint [3].

[4] *PortConnectionStereotype*: see SaveCombinedOutPort constraint [4].

[5] *PortsExternal*: see the SaveCombinedInPort constraint [5].

[6] *PortsInterfaces*: see the SaveCombinedInPort constraint [6].

#### 6.3.3.13 SaveDelay

This stereotype is intended for modelling the SaveCCM Delay component. Since Delay is a kind of a component, the `Component` metaclass is used for the base classifier.

**Base classifier:** `Component`

**Tagged values:** Defined tagged values are equivalent to the properties existing in SaveCCM, and those are:

- *Body : String*.
- *Delay : Float*.
- *Precision : Float*.

**Constraints:**

[1] *ComponentConnections*: see the SaveAssembly constraint [2].

[2] *ComponentIntefraces*: see the SaveClock constraint [4].

[3] *ComponentPackagedElements*: see the SaveClock constraint [5].

[4] *DelayAttributes*: UML enables a `Component` to contain several types of attributes: ports, attributes and operations. SaveDelay component may own ports and quality attributes (properties with the SaveAttribute stereotype applied). Other attributes are not allowed.

[5] *DelayPorts*: According to SaveCCM specification, a Delay component has only two ports – one input trigger port and one output trigger port. This constraint checks if that condition is fulfilled.

[6] *NumberOfStereotypes*: see the SaveAssembly constraint [6].

#### 6.3.3.14 SaveDelegation

The delegation in SaveCCM is used for connecting a port of a component to a port of its subcomponent. Delegation is used while defining the internal structure of a composite component or an assembly. In the delegation relationship, data or signals from a components/subcomponents port are delegated to subcomponents/components port. Therefore, either a components port depends on its internal port for some data or signal, or a components internal port depends on a components port for data or signal. Considering the semantics, the UML dependency relationship seemed suitable as a base classifier for the delegation stereotype. The delegation connection connects ports of the same direction.

**Base classifier**: `Dependency`

**Tagged values:**

- *Source : String*. The source of the delegation – the starting port.
- *Destination : String*. The destination of the delegation – the ending port

**Constraints:**

[1] *DelegationComplex*: see the SaveConnection constraint [1].

[2] *DelegationComposite*: Either source or destination port must be owned by a composite or assembly component (the component with SaveAssembly or SaveComposite stereotype applied).

[3] *DelegationConformance*: This constraint checks the type conformance of the connected ports. The type conformance condition is the same as in SaveIDE tool. The condition checks the direction of the connected ports, their types and as circumstances require, the equality of their "Type" tagged values.

[4] *DelegationCyclic*: It is not allowed to have cyclic connection.

[5] *DelegationNotUsage*: Since `Usage` relationship is inherited from `Dependency` relationship, then usage is a kind of dependency and it is possible to apply SaveDelegation stereotype to usage relationship. Usage relationship is used for modelling the SaveCCM connection (see section 6.3.3.10), therefore it is necessary to check that SaveDelegation stereotype is applied to delegation relationship and not to usage relationship.

[6] *DelegationPorts*: Source and destination port must have an appropriate stereotype applied.

[7] *NumberOfStereotypes*: see the SaveAssembly constraint [6].

#### 6.3.3.15 SaveFromComplex

A special kind of a connection is used to connect the complex connection to a port of a component. The usage relationship is used as the base classifier based on the same rationale with the basic component connection (see section 6.3.3.10).

**Base classifier:** `Usage`

**Tagged values:** This stereotype does not have tagged values defined.

**Constraints:**

[1] *ComplexConnectionDirection*: SaveFromComplex is used to connect a complex connection to a component. This constraint checks if the client of a connection (starting point) is a component with SaveComplexConnection stereotype applied, and a supplier of a connection (ending point) must be an input port (a port with SaveTriggerInPort, SaveDataInPort or SaveCombinedInPort stereotype applied).

[2] *NumberOfStereotypes*: see the SaveAssembly constraint [6].

#### 6.3.3.16 SaveModel

A model in the SaveCCM domain represents a software realization of a components behaviour, most likely in the form of a programming language function. UML `Artifact` represents a physical piece of information that is used, among other things, by the operation of a system. A UML artifact can therefore be a model or a software. This means that the UML artifact is a good base classifier for the SaveCCM model.

A model is connected to a component through a UML manifestation relationship which indicates their semantically link. UML `Manifestation` is usually used for division between an abstraction and a manifestation, for example distinction between a class and an object, where the class is an abstraction and the object is a clear manifestation of that class. Most UML building blocks have this kind of class/object distinction, e.g. use case, use case instance etc. Therefore, manifestation relationship is suitable for this purpose. A component can have any number of models attached to it.

**Base classifier:** `Artifact`

**Tagged values:**

- *Type : String*. Describes what type of model it is.
- *Filename : String*. A file containing model implementation.
- *Body : String*. Instead of using an external file, a model can be embedded as text within the element.

**Constraints:**

[1] *ModelAttributes*: UML allows for an `Artifact` metaclass having various attributes, however SaveCCM model element can not contain any attribute (property). Except attributes, this constraint checks that there are not any connections used for connecting this element to other elements, only a manifestation relationship is allowed.

[2] *ModelRealization*: In SaveCCM only a component, clock, delay, composite component and a complex connection can have a model attached. For each manifestation relationship that connects the SaveModel artifact to another element, it is checked that the artifact is connected to an element which has an appropriate stereotype applied. By appropriate stereotype it is considered to be one of the stereotypes defined by SaveUML profile that are used to model SaveCCM elements listed above.

[3] *NumberOfStereotypes*: see the SaveAssembly constraint [6].

#### 6.3.3.17 SaveSwitch

This stereotype is intended for modelling SaveCCM switch element. The rationale behind using UML `Component` as the base classifier for the SaveCCM switch element is the same as with the SaveComponent. The switch is externally equivalent to a component in SaveCCM, it can have ports as the component has and can be connected to any other component. The difference between the component and the switch is in its internal behaviour.

In SaveCCM, switch can have a number of connection patterns and connection conditions. In SaveUML profile those two elements are modelled using a single stereotype – SaveSwitchCondition. Each SaveSwitch component can have any number of SaveSwitchCondition elements that are added to SaveSwitch component as its attributes.

**Base classifier:** `Component`

**Tagged values:** This stereotype does not have tagged values defined.

**Constraints:**

[1] *ComponentConnections*: see the SaveAssembly constraint [2].

[2] *ComponentIntefraces*: see the SaveClock constraint [4].

[3] *ComponentPackagedElements*: see the SaveClock constraint [5].

[4] *NumberOfStereotypes*: see the SaveAssembly constraint [6].

[5] *SwitchAttributes*: UML enables a `Component` to contain several types of attributes: ports, attributes and operations. SaveSwitch component may own ports and switch condition descriptions (properties with the SaveBindPort stereotype applied). Other attributes are not allowed.

[6] *SwitchPortNames*: This constraint checks if all ports owned by a SaveSwitch component have a unique name. This is important in order to use switch condition descriptions where the name of the port has to be entered (see section 6.3.3.18).

[7] *SwitchPorts*: see the SaveComponent constraint [6].

#### 6.3.3.18 SaveSwitchCondition

This stereotype represents two elements from SaveCCM domain – connection pattern and connection condition. In SaveCCM, connection pattern and connection condition are used to model a switch behaviour. Switch connection pattern determines two ports that can be connected, while the switch connection condition defines the terms under which this connection is established. Although they model the behaviour, these two elements can also be looked upon as a property of a switch component. Therefore, SaveUML profile uses `Property` metaclass as the base classifier for SaveSwitchCondition stereotype. Each SaveSwitchCondition element is added to SaveSwitch component as its attribute.

**Base classifier:** `Property`

**Tagged values:**

- *SetPort : String*. The name of the port used in a condition (this is a part of connection condition).
- *SetPortValue : String*. The value of data on set port that needs to arrive for condition to become true (this is a part of connection condition).
- *ConnectFrom : String*. The name of the input port that has to be connected to the port stated in "ConnectTo" tagged value (this is a part of connection pattern).
- *ConnectTo : String*. The name of the output port that has to be connected to the port stated in "ConnectFrom" tagged value (this is a part of connection pattern).

**Constraints:**

[1] *PropertyConnectors*: see the SaveAttribute constraint [1].

[2] *PropertyNumberOfStereotypes*: see the SaveAssembly constraint [6].

[3] *SwitchConditionEqualPorts*: Ports stated in "SetPort", "ConnectFrom" and "ConnectTo" tagged values are not allowed to be equal. It is not allowed to connect a port to itself or to use a set port in connection pattern.

[4] *SwitchConditionFrom*: Port stated in "ConnectFrom" tagged value must be an input port of a switch component (a port with a SaveTriggerInPort, SaveDataInPort or SaveCombinedInPort stereotype applied).

[5] *SwitchConditionPorts*: Checks if the ports stated in "SetPort", "ConnectFrom" and "ConnectTo" tagged values exists (checks that the switch component owns the ports with the stated names).

[6] *SwitchConditionSetPort*: Only a data or combined input port can be a set port. This constraint checks that the port stated in "SetPort" tagged value has an appropriate stereotype applied (SaveDataInport or SaveCombinedInPort stereotype). Also it checks that the port stated "SetPort" tagged value is marked as a set port (the "SetPort" tagged value of the port must be set to ‘true’).

[7] *SwitchConditionTo*: Port stated in "ConnectTo" tagged value must be an output port of a switch component (a port with a SaveTriggerOutPort, SaveDataOutPort or SaveCombinedOutPort stereotype applied).

#### 6.3.3.19 SaveToComplex

A special kind of a connection is used to connect the port of a component to a complex connection. The usage relationship is used as the base classifier based on the same rationale with the basic component connection (see section 6.3.3.10).

**Base classifier:** `Usage`

**Tagged values:** This stereotype does not have tagged values defined.

**Constraints:**

[1] *ComplexConnectionDirection*: SaveToComplex is used to connect a component to a complex connection. This constraint checks if the client of a connection (starting point) is an output port (a port with SaveTriggerOutPort, SaveDataOutPort or SaveCombinedOutPort stereotype applied), and a supplier of a connection (ending point) must be a component with SaveComplexConnection stereotype applied.

[2] *NumberOfStereotypes*: see the SaveAssembly constraint [6].

#### 6.3.3.20 SaveTriggerInPort

This stereotype is used for modelling SaveCCM trigger in ports. The rationale behind using UML `Port` as the base classifier for the SaveCCM trigger input port element is the same as with the SaveCombinedInPort (see section 6.3.3.5).

**Base classifier:** `Port`

**Tagged values:**

- *External : String*. A label mapping the port to external entity, in case a port is an *external port*.

**Constraints:**

[1] *NumberOfStereotypes*: see the SaveAssembly constraint [6].

[2] *OwnerComponentStereotype*: see the SaveCombinedInPort constraint [2].

[3] *PortConnections*: see the SaveCombinedInPort constraint [3].

[4] *PortConnectionStereotype*: see the SaveCombinedInPort constraint [4].

[5] *PortsExternal*: see the SaveCombinedInPort constraint [5].

[6] *PortsInterfaces*: see the SaveCombinedInPort constraint [6].

#### 6.3.3.21 SaveTriggerOutPort

This stereotype is used for modelling SaveCCM trigger out ports. The rationale behind using UML `Port` as the base classifier for the SaveCCM trigger output port element is the same as with the SaveCombinedInPort (see section 6.3.3.5).

**Base classifier:** `Port`

**Tagged values:**

- *External : String*. A label mapping the port to external entity, in case a port is an *external port*.

**Constraints:**

[1] *NumberOfStereotypes*: see the SaveAssembly constraint [6].

[2] *OwnerComponentStereotype*: see the SaveCombinedInPort constraint [2].

[3] *PortConnections*: see the SaveCombinedInPort constraint [3].

[4] *PortConnectionStereotype*: see SaveCombinedOutPort constraint [4].

[5] *PortsExternal*: see the SaveCombinedInPort constraint [5].

[6] *PortsInterfaces*: see the SaveCombinedInPort constraint [6].

## 6.4 Using SaveUML profile

The profile specification provided in previous section is a general profile description and needs to be implemented in a design tool in order to be used for a concrete model design of SaveCCM models in UML.

### 6.4.1 UML CASE tool

Among various CASE tools available, *IBM Rational Software Modeller* (RSM) was chosen as the best tool for this purpose.

Within a SaveUML project a fourteen existing UML tools were analysed, with several requests that were taken into consideration:

- Support of UML 2.0 component model (this was a major limitation with most free tools).
- UML profile support. This includes using UML profiles when modelling user models, but also the possibility of creating UML profiles.
- Importing and exporting user models in XMI (XML Metadata Interchange [XMI web]) based files.

After the analysis it was concluded that IBM RSM fulfils all requests and that it is suitable for this work. Therefore, within this thesis, SaveUML profile was implemented using IBM RSM tool and released as an *epx* file (*epx* is a profile file extension used in RSM).

### 6.4.2 Applying stereotypes to user model elements

In order to apply SaveUML stereotypes to a model element, first the SaveUML profile must be applied to the model (for information on applying UML profiles to a model see RSM help or [SPusr 08]).

After the profile was included in the model, all of its elements will be accessible in the model. Stereotype for each element can be selected in the Properties view of a model element under "Stereotypes" tab, as shown in Figure 6-1.

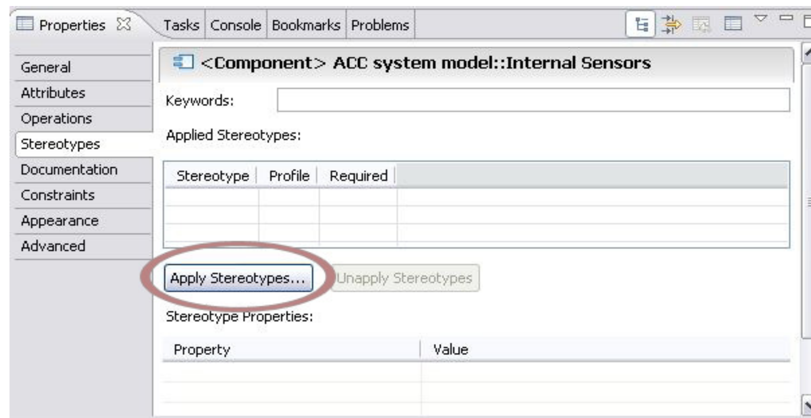


Figure 6-1: Applying stereotype – Properties view

Stereotype can be chosen in the "Apply Stereotypes" pop-up menu as shown in Figure 6-2 (a). It is important to note that there is a large number of stereotypes available in UML, therefore the stereotypes from the applied profile will be available among them. Figure 6-2 (a) shows a list of available stereotypes for a UML `Component`. After selecting a stereotype, it is rendered as a name enclosed by guillemets (French quotation marks of the form « »), placed above the name of modelling element. A component Internal Sensors with SaveComponent stereotype applied is depicted in Figure 6-2 (b).

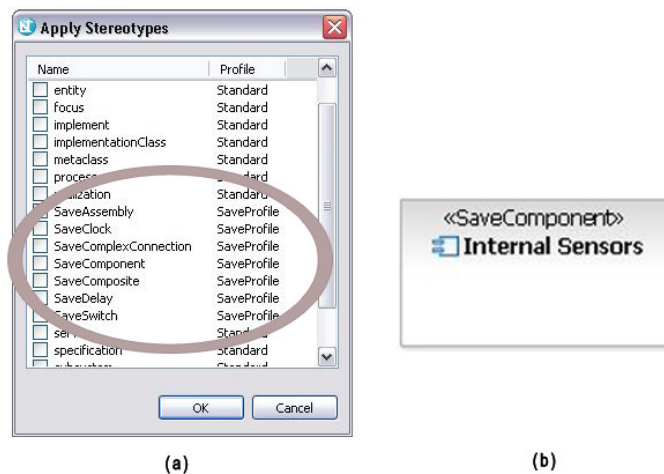


Figure 6-2: Applying stereotype – available stereotypes

In addition, after a stereotype is applied its tagged values are available in the Properties view, and can be edited entering a value in the "Value" cell, as it is shown in Figure 6-3.

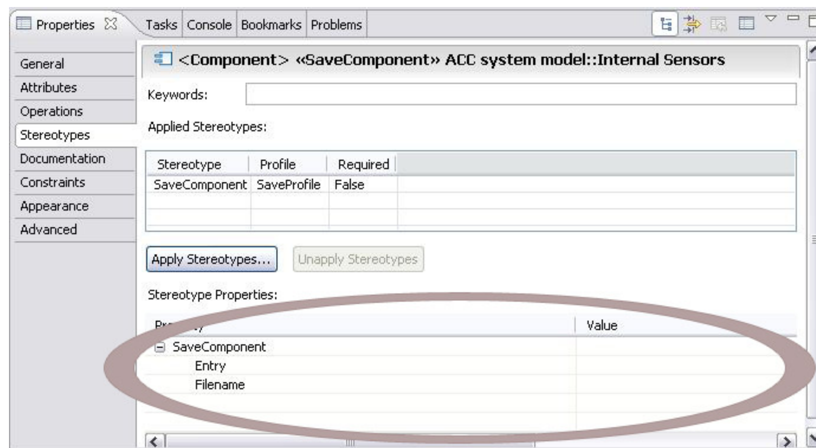


Figure 6-3: Applying stereotype – editing tagged values

### 6.4.3 Modelling UML models using SaveUML profile

SaveUML profile extends several UML metaclasses which can be used for modelling SaveCCM elements, those are: `Component`, `Property`, `Port`, `Usage`, `Dependency` and `Artifact`. This section provides examples of using some of those elements.

#### Component

For modelling SaveCCM architectural elements (using UML `Component` metaclass) it is necessary to add a UML component to the model. Then one of the available stereotypes has to be applied depending on SaveCCM element that is modelled. Applying a stereotype to a UML component is described in previous section.

#### Property and Port

`Property` is used for modelling component attributes, while `Port` is used for modelling component ports. They can be added to a component in the Properties view under "Attributes" tab, this is shown in Figure 6-4. It is possible to choose either a Port or an Attribute property.

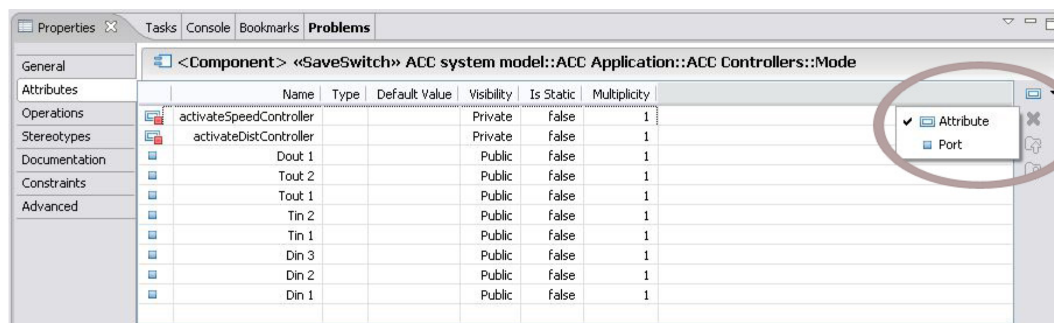


Figure 6-4: Adding a property to a component

When modelling SaveCCM port it is necessary to add a port to a component and when modelling SaveCCM *bind port*, *attribute* or *switch condition* it is needed to add an attribute to a component. After the property or a port is added, the procedure of applying stereotype and setting tagged values is the one described in section 6.4.2.

Figure 6-5 shows a screenshot of Project explorer view of ACC system model. It provides an example of a component (Object recognition) with three properties, one with SaveAttribute stereotype applied (attribute1) and two with SaveBindPort (bindDin1 and bindDin2). Also on this example there are various ports shown.

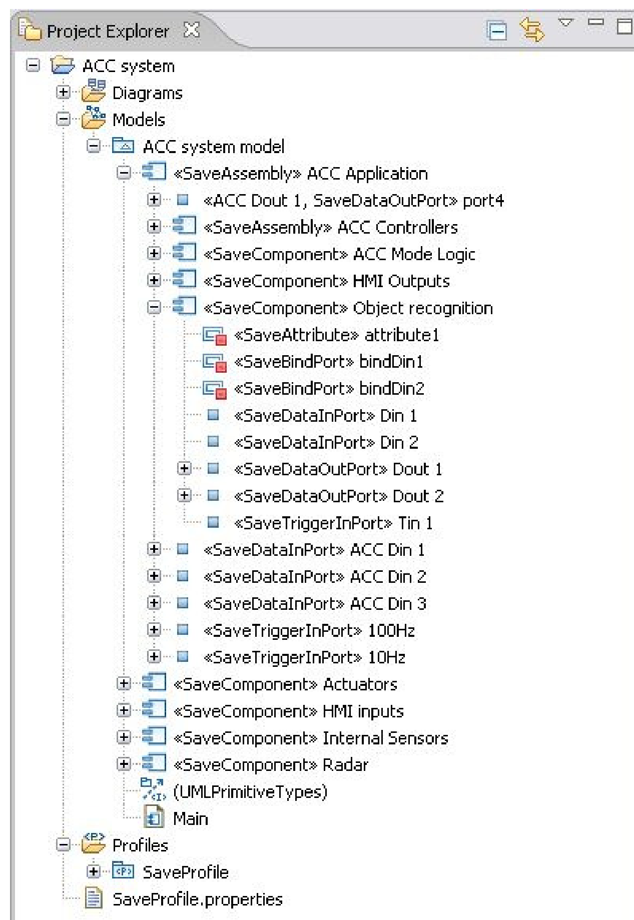


Figure 6-5: UML model of ACC system – Project explorer view

### Artifact

In SaveUML profile `Artifact` is used for modelling SaveCCM model element. It is attached to a component with a manifestation relationship. First it is necessary to add an artifact to the model, the SaveModel stereotype can be applied to the artifact (the procedure is equal to the one described in section 6.4.2). Finally a model has to be connected to the component using manifestation relationship. An example of a SaveComponent with SaveModel attached is shown in Figure 6-6.

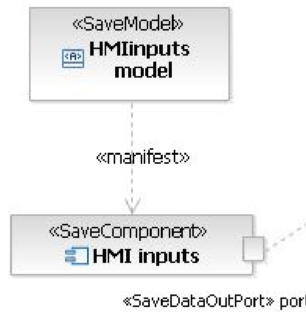


Figure 6-6: Connecting a SaveModel element to an SaveComponent model element

**Internal structure**

SaveCCM assembly and composite component elements can have an internal structure. As it is described in section 6.2.1.2, a `PackageableElement` metaclass is used for this purpose. In RSM a subcomponent can be added to a component by selecting the component in the Project explorer view and choosing the "Component" element form a popup menu under "Add UML" submenu.

Example of Figure 6-5 shows an internal structure of ACC Application component (which has a SaveAssembly stereotype applied) with various subcomponents listed.

### 6.4.4 Validating the model

Constraints defined within the SaveUML profile are used for validating a user model in order to ensure that the model is valid in consideration of the SaveCCM semantics. It is very important to validate a user model to avoid it from being invalid due to mistakes that could be done during modelling.

In RSM tool, constraints defined within the profile are divided into two groups, constraints with *live validation* and constraints with *batch validation*. Constraints with batch validation are checked when the user runs a validation. Constraints with live validation are checked every time the model element, to which the stereotype is applied, is modified. In addition, the constraints with live validation are also checked when the user runs the validation.

**Live validation**

An example of a constraint with a live validation is the SaveAssembly constraint *ComponentConnections* that suppresses using of any connectors directly on a component (no possible connections except the ones explicitly captured by the ports are allowed). If a constraint is violated an immediate notification arises in a form of a popup window as it is shown in Figure 6-7.

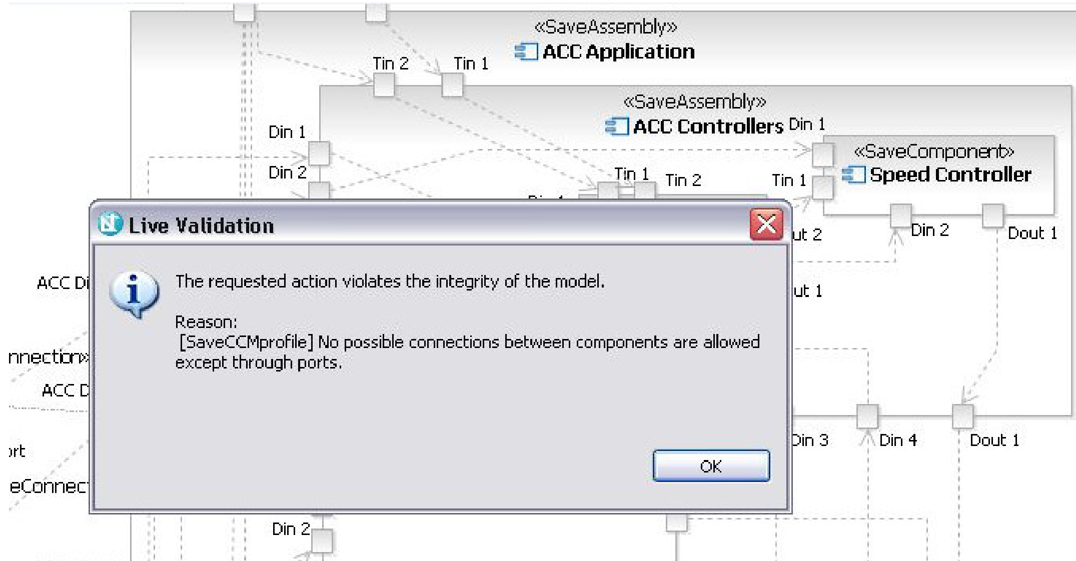


Figure 6-7: Validating model – live validation popup window

### Batch validation

An example of a constraint with a batch validation is a SaveComponent constraint *ComponentPorts*, which requires that all ports owned by a component have an appropriate stereotype, applied. To meet this demand, after adding the port to a component, it is necessary to apply a stereotype to the port. Since this is a two – step process, this constraint can not have a live validation (the constraint would be violated after the first step).

As it is already mentioned, constraints with batch validation are checked when user runs the validation. The result of a validation is a list of error messages for each constraint that has been violated. An example of error messages generated after running the validation is shown in Figure 6-8.

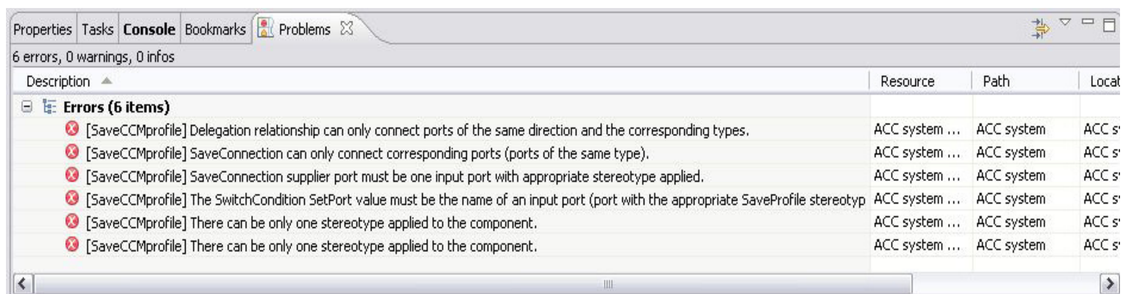


Figure 6-8: Validating model – batch validation error messages

Note that last two messages are equal. There is a possibility of reporting more than one error message for the same offence. The reason for his is that some stereotypes have the same or similar constraints applied. In the example above, an element has two stereotypes applied, since every of those two stereotypes has the *NumberOfStereotypes* constraint, each of them will report an error.

# 7. SaveUML transformations

This section describes the concept of transforming UML models to SaveCCM models and reverse. It provides a conceptual design of transformations and a demonstration the SaveUML transformation tool implemented within this thesis.

## 7.1 Abstract

SaveUML profile described in previous section allows modelling UML models with SaveCCM semantics. The subsequent step is transforming the UML model into SaveCCM model. These transformations allow joining the UML – widely accepted modelling language with SaveCCM – more domain – specific modelling language.

The transformations use XML Metadata Interchange (XMI) representations of models. XMI is a standard that enables interchange of any kind of metadata that can be expressed using the MOF specification (including both the model and metamodel information.) between modelling tools. It eases the problem of tool interoperability by providing a flexible and easily parsed information interchange format. In principle, a tool needs only to be able to save and load the data in XMI format. However, this is not yet accomplished, and there are differences in generated XMI model representations between various tools. Therefore the transformations are still tool-dependent.

## 7.2 Conceptual design

The conceptual design of SaveUML transformations is depicted graphically in Figure 7-1.

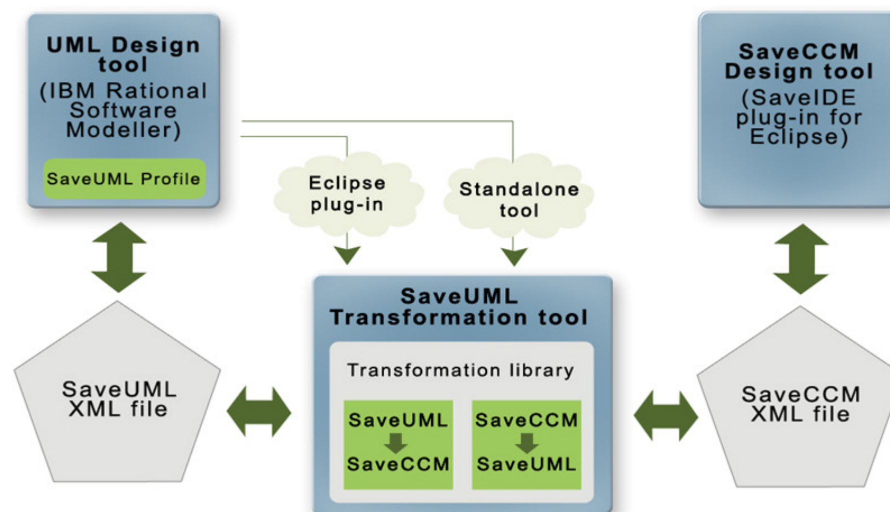


Figure 7-1: Conceptual design of SaveUML transformations

The UML CASE tool (in this case the Rational Software Modeller) is used for creating a UML user model. By applying the SaveUML profile, UML elements can be stereotyped for modelling SaveCCM elements. The application of the profile is necessary in order to create a model which can be transferred into a SaveCCM model. RSM uses an eXtensible Markup Language (XML) – based file for representing the model information.

The SaveCCM design tool is Save-IDE. SaveIDE uses several files for representing model information. Those files are also compatible with XML and are used by the transformation tool to perform the SaveCCM to UML transformation.

The transformation tool can be used either as an Eclipse plug-in or as a standalone application and performs transformation in both directions, UML to SaveCCM and SaveCCM to UML models. The transformation tool uses the transformation library to perform transformations. The transformation library contains eXtensible Stylesheet Language (XSL) style sheets for transforming from SaveUML into SaveCCM and SaveCCM into SaveUML models. Input files based on XML are parsed through the XSL style sheets and then XML – based output files, compatible with the desired tool, are generated.

## 7.3 XSLT transformations

In order to transform XML – based representations of user models, XSL transformations are used. They are a part of the transformation library which is based on the Saxon XSL processor.

Extensible stylesheet language transformations (XSLT) is a language for transforming XML documents into other XML documents. It is a part of the W3C recommendation and its current version is 1.0. XSLT is designed for use as part of XSL, which is a stylesheet language for XML.

A transformation in the XSLT language is expressed as a well-formed XML document. A stylesheet contains a set of template rules. Each template rule has two parts: a pattern which is matched against nodes in the source tree and a template which can be instantiated to form part of the result tree. This allows a stylesheet to be applicable to a wide class of documents that have similar source tree structures. On the other hand, the structure of the result tree can be completely different from the structure of the source tree. In constructing the result tree, elements from the source tree can be filtered and reordered, and arbitrary structure can be added.

Transformations between UML and SaveCCM models could also be implemented using Java programming language, since Java provides a good

support for XML parsing. However XSLT is a standard way of transforming XML files, thus it is chosen for this purpose.

Transformations will generate a valid output model only under the condition that the input model is valid. When transforming from SaveCCM to UML, SaveIDE will take care of the model validation. When transforming from UML to SaveCCM, SaveUML profile constraints are intended for model validation purposes. Therefore, it is important to use SaveUML profile when modelling. All elements that do not have a stereotype applied (except the manifestation link used for associating model to a component) will be ignored during the transformation.

### 7.3.1 User model files

In order to implement the XSL transformations, input and output file formats were examined in detail to identify how different model elements are stored in each format.

#### RSM file format

RSM stores UML models in an *emx* file which is compatible with XML. RSM also provides a possibility of exporting the model into an XML Metadata Interchange (XMI) file format [XMI web] which enables an easy interchange of metadata between modelling tools. In distinction from *emx* files that contain diagram information of a model, XMI file contains a general model information only.

Eventhough XMI is a standard format for representing UML models in XML – based files, the *emx* file was chosen to be the input/output file format for the transformations. The reason for this is that it is not possible to generate UML model diagrams in RSM from a model represented in XMI file format.

Since this decision makes the transformations dependable on the specific tool used for modelling, this problem is left open. The possibility of transforming XMI files instead of transforming *emx* files should be reconsidered in future development of the tool.

#### SaveIDE file format

SaveIDE tool also stores models in XML – based format, however it uses separate files for model information and for diagram information. There are several different files:

- *saveccm*: a file with *saveccm* extension is the most important file. It holds model information including all elements existing in the model, their properties and their relations. This file is used for generating diagram files and for generating *save* files. SaveIDE tool uses this file at the design – time to store all the information related to the model.

- *save*: after the SaveCCM model is created a user can generate a *save* file. Document type definition (DTD) of this file is described in SaveCCM reference manual [SaveR 07]. Just as the *saveccm* file, *save* file contains general information about the model. However, it is not possible to generate diagram files from *save* file. Another difference from *saveccm* file is that *save* file is not used during design – time, it is generated on users explicit request.
- *saveccm_diagram*: the file with *saveccm_diagram* extension holds the information about the visual representation of model elements (their layout). SaveIDE tool generates it from *saveccm* file. This file contains only the diagram information, while general information about the model is to be found in *saveccm* file.
- *composite_diagram*, *assembly_diagram*, *switch_diagram*: these three kind of files are intended tor visual representation of internal structure of composite component, assembly and switch, respectively. Unlike RSM, model diagram in SaveIDE (*saveccm_diagram* file) does not expose internal structure of model elements. For each model element that can have an internal structure, SaveIDE offers functionality for generating separate diagram file that represents its internal structure. This means that for one model, a number of *composite_diagram*, *assembly_diagram* and *switch_diagram* files can exist, for each composite component, assembly or switch existing in the model.

It is decided that *saveccm* file will be used as input/output file in transformations. *Save* file would probably be more convenient for this purpose because it is not used by SaveIDE during design time, therefore it has not additional information used by the tool and can not be changed. However, a disadvantage of using *save* file is the fact that it would not be possible for user to generate diagram files.

Diagram files are not transformed due to the fact that SaveIDE offers functionality for generating diagram files from *saveccm* file.

### 7.3.2 Input parameters

In order to customize XSL transformations they accept a few input parameters that determine how the transformations will be executed.

#### Model name

Both, UML to SaveCCM and SaveCCM to UML transformations receive a name of the model that is transformed. The name of the model is determined by an input file name.

#### Model object

When transforming from UML to SaveCCM model, there is one more input parameter – *model object*. Model object defines the top-level object of the model, in other words, it specifies the object that is modelled.

When user is creating a new SaveCCM model using SaveIDE tool, he can choose between various model objects. The offered objects are all SaveCCM architectural elements (including component, switch, ports, connectors etc.) plus a "system" model object. However, since RSM tool has some restrictions, a set of model objects that can be sent to SaveUML transformations as an input parameter contains only the objects that have sense. For example, RSM does not allow for a port to be a stand-alone element, the port can only be added to an existing component and it depends upon it, therefore port model object is not offered in SaveUML transformation tool. The model objects that UML to SaveCCM transformation supports are system, assembly, composite, component, clock, delay and switch.

The approach of providing model object parameter to SaveUML transformations differs from the one used in transformation tool created within the SaveUML project [SP web]. That tool distinguishes two kinds of models "saveccm diagram" and "assembly diagram". When user chooses a transformation of "saveccm diagram", a *saveccm* file with a system model object and a *saveccm_diagram* file are generated. When user chooses a transformation of "assembly diagram", a *saveccm* file with an assembly model object and an *assembly_diagram* file are generated. The author's opinion is that this tool offered the transformation of only two specific cases – system model object and assembly model object. The distinction between SaveCCM models was made according to the diagram file it contains. However, this approach is not correct, since an *assembly_diagram* represents the visual representation of assembly element and each SaveCCM model can have a number of *assembly_diagram* files, one for each assembly component it contains. The same observation goes for composite_diagram and switch_diagram files.

## 7.4 Transformation tool

This section describes the SaveUML transformation tool.

### 7.4.1 System specification

The UML design tool used is the third party software IBM RSM version 7.0.0. The UML profile is created with and for RSM and is only tested with this software. Regarding the UML standard, profiles should be universal. However, this has not yet been accomplished and the implementation is specific for different tools.

SaveCCM models are created with the SaveIDE, which is a plug-in for Eclipse. SaveIDE is not a commercial software but a research project at MdH. Both RSM and SAVE-IDE can produce XML - based representations of models and diagrams.

The transformation is specified by XSL stylesheets. The XSLT processor used is Saxon-B 9.0.

The transformation tool is developed using Java 1.5.

### 7.4.2 The transformation tool architecture

The overview of the SaveUML transformation tool architecture is presented in Figure 7-2 (figure is taken from [SPfin 08]).

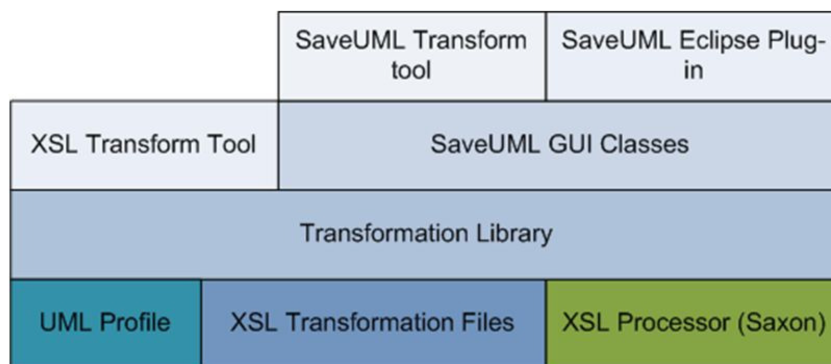


Figure 7-2: SaveUML transformation tool architecture

The base of the transformations are XSL transformation files and XSL processor. Transformation library comprise those files, together with SaveUML profile and Java classes that use XSL transformation library to perform the transformation. The front-end for the transformation library is a graphical Java tool. The GUI classes are the same in both RSM Eclipse plug-in and in the standalone application. The only difference is in the interface that uses these classes.

### 7.4.3 Error handling

SaveUML transformation tool handles several different errors and exceptions that can arise during the process of transformation and those are:

[1] *Source model has the same format as target model*. This situation is not allowed for obvious reasons, having the same input and output file is not possible, thus the transform button will be disabled.

[2] *The source model file extension is invalid*. The tool checks for the correct file extension. In case of the wrong extension, a warning will be generated. However, it is up to the model designer to make sure the model is valid. The transformation tool will transform the input file without checking if. Checking the validity of the model is decided not to be necessary because of the presumption that the user will not try to abuse the tool.

[3] *Target file already exists*. If the output file already exists, the user friendly interface reports a warning message about the file already existing on the file system and provides the option for overwriting.

[4] *Wrong model type chosen*. As it is described in section 7.3.1, SaveIDE uses various different types of model objects. When starting the transformation, the user can choose the type of the model objects (only the types that are sensible are offered, see section 7.3.2). SaveUML recognizes the actual type of the model, and in case of the wrong choice, suggests the correct choice to the user.

Different errors could be made by the user while using the RSM tool for modelling a UML user model. Since RSM is equipped with its own error handling methods, SaveUML transformation tool does not get in the way and allows the user to use the RSM in any way desired. RSM is not intended exclusively for UML to SaveCCM transformation, but can be used as a UML tool as well, therefore SaveUML error handling would obstacle the user when using RSM for general purposes. To avoid a user from creating an invalid UML model, constraints are implemented within the SaveUML profile (see section 6.3). It is strongly recommended to run the validation of the model prior to transforming the UML model to SaveCCM model.

When modelling a SaveCCM model, SaveIDE tool will take care of the model validation as it has its own constraints implemented.

Transformation tool will transform the input file without checking the file extensions to see if the format of the file is correct. However, if the input file has a certain extension, but not the contents which are transferable, the transformation tool will return an output file. This output file will be an accurate transformation of the inaccurate input file and will not be usable with the desired tool. SaveUML does not check for these kinds of errors because it is presumed that they should not happen unless the user intentionally tries to provoke the error.

### 7.4.4 Using SaveUML transformation tool

The SaveUML transformation tool is available as a stand-alone tool or as an RSM plug-in. Their external interfaces are very similar.

#### Stand-alone tool interface

When using the stand-alone tool, the interface provides the user with the complete control panel allowing the setup of all necessary options. The control panel allows the user to set up all features of the transformation, such as the direction and both input and output files. Depending on the direction, the standalone application will/will not allow the user to perform certain actions and provide the necessary warnings/error messages. It also provides an interface for

exporting the SaveUML profile from the Java library. The front-end interface of the standalone application is shown in Figure 7-3.

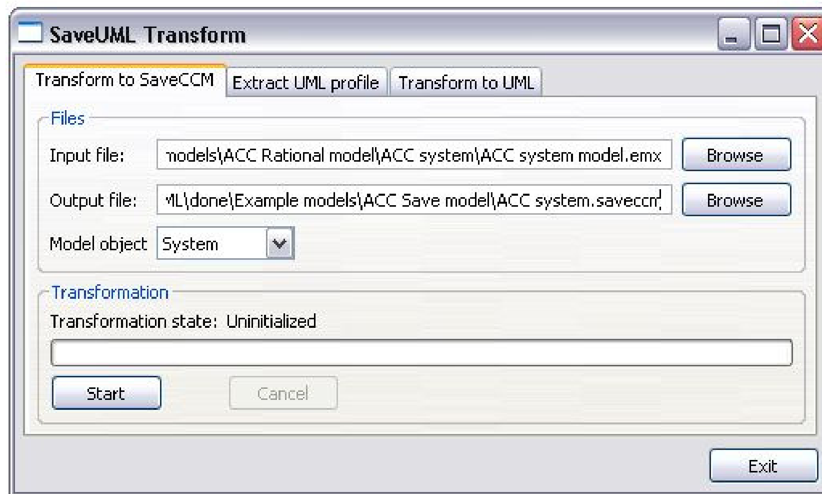


Figure 7-3: The stand-alone application graphical interface

### RSM plug-in

Another external interface of the tool is a RSM plug-in which makes the tool easier and more practical to use by RSM users.

The tool is available from the pop-up menu after right clicking the desired model file or the project file, and is presented as "SaveUML" item as it is depicted on Figure 7-4. This allows importing SaveUML profile into the workspace and extracting the selected UML model to SaveCCM model.

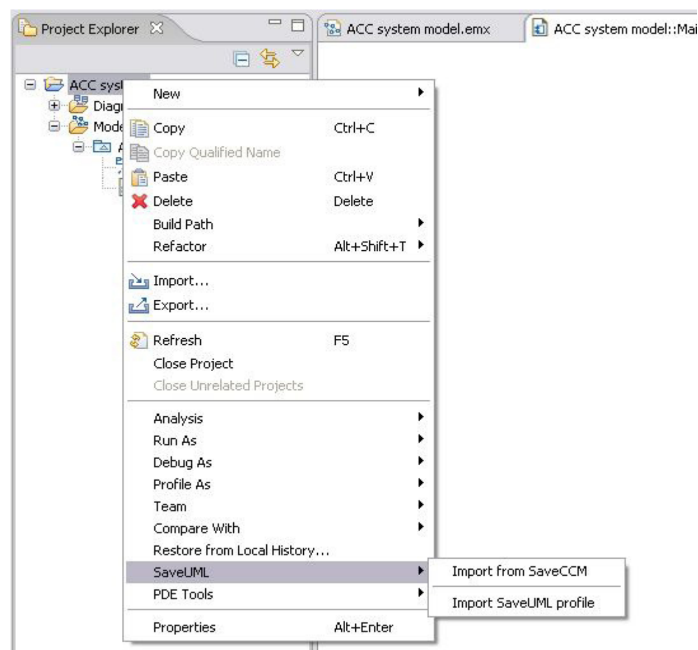


Figure 7-4: Using RSM plug-in, pop-up menu

When transforming the UML model to SaveCCM model user can specify the output file and the desired model object. The user is not allowed to change the input filename, since the file of the model that was right-clicked is assumed. This is depicted in Figure 7-5.

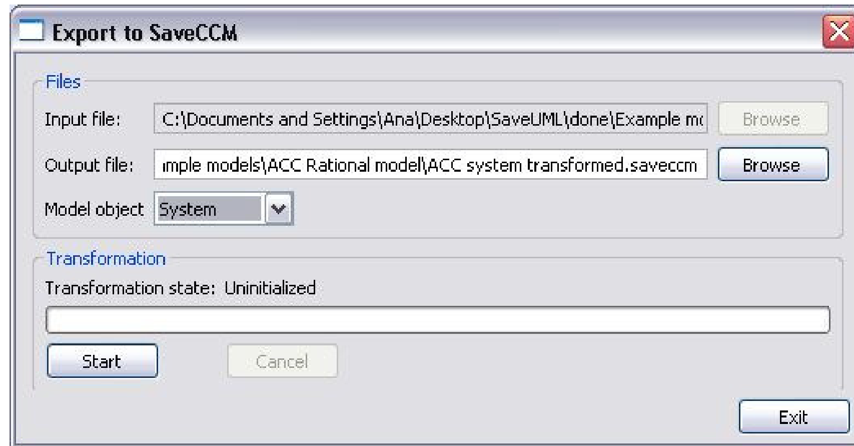


Figure 7-5: Using RSM plug-in, exporting UML model

If the model object chosen by the user differs from the model object detected by the tool the warning message will be generated as it is shown on Figure 7-6.

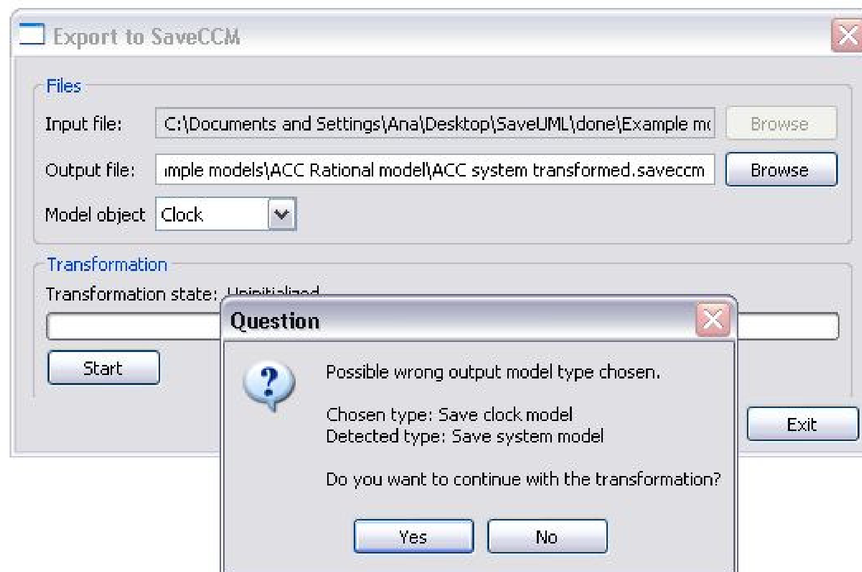


Figure 7-6: Using RSM plug-in, warning message

Finally, it the transformation was successful an appropriate message will be generated.

To demonstrate the SaveUML transformation tool an ACC system UML model was created (ACC system example is described in section 5.4), the model is presented on Figure 7-7.

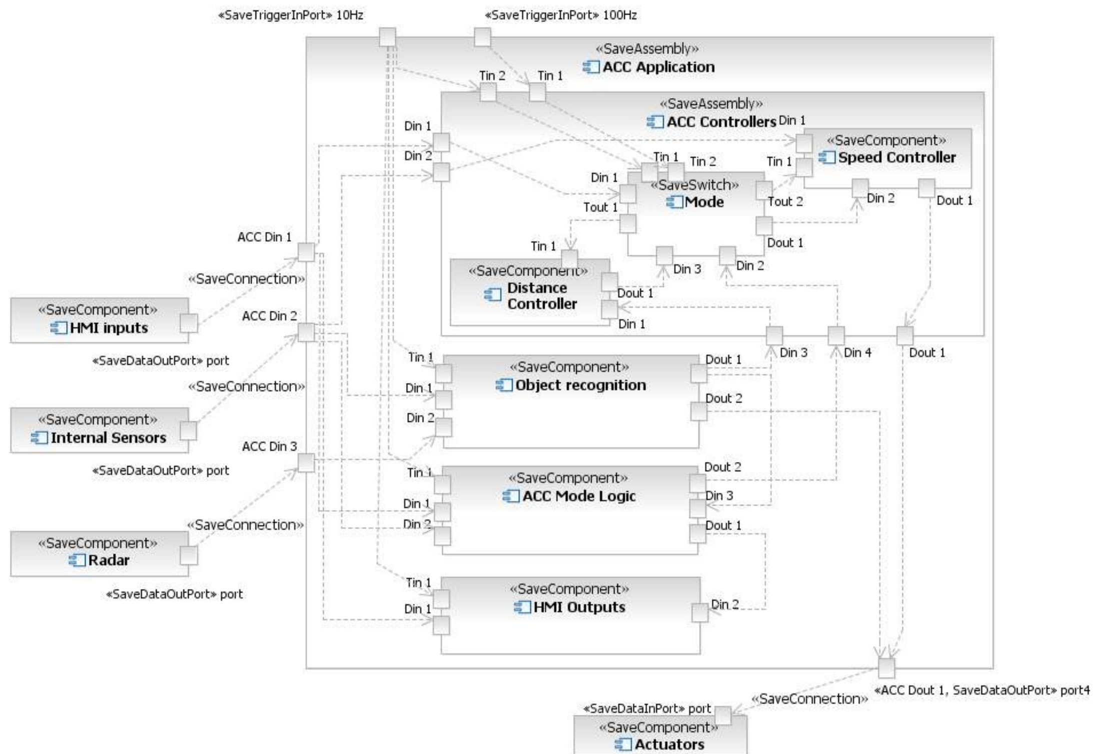


Figure 7-7: UML model of ACC system

SaveUML transformation tool transforms this model into a SaveCCM model consisting of one *saveccm* file. The screenshot of this file form SaveIDE is shown on Figure 7-8. After the transformation, using SaveIDE tool, it is possible to generate diagram files for visual presentation of the model (diagram files of the model are shown on Figure 5-3).

ACC system.saveccm
ACC system.saveccm_diagram
Resource Set
platform:/resource/SaveACC/ACC%20system.saveccm
System ACC system
Connection
Connection
Connection
Connection
Component HMI inputs
Component Internal Sensors
Component Radar
Component Actuators
Assembly ACC Application
Trigger In 100Hz
Trigger In 10Hz
Data Out port4
Data In ACC Din 3
Data In ACC Din 2
Data In ACC Din 1
Delegation
Delegation
Delegation
Delegation
Connection
Connection
Delegation
Delegation
Delegation
Delegation
Delegation
Delegation
Delegation
Delegation
Delegation
Connection
Connection
Delegation
Selection | Parent | List | Tree | Table | Tree with Columns

Figure 7-8: Transformed ACC system model

# 8. Conclusion

Visual design is an important part of development process in software engineering. There are various modelling languages available for this purpose. One of the most accepted ones is UML. UML can be used in almost all problem domains from enterprise information systems and distributed web-based applications to real-time embedded systems. However, even though the UML is very well defined, with increasing knowledge and experience comes the need for greater domain specialization.

This thesis presented the way of mapping UML elements to SaveCCM research model. The most suitable way of achieving this mapping is using one of the two possible UML extension mechanisms – UML profiles. UML profile extension mechanism is very powerful mechanism that allows tailoring UML for a specific domain and refining the UML semantics. The SaveUML profile developed within this thesis shows how to map a strict SaveCCM semantics to UML. To map SaveCCM architectural elements various stereotypes were used, optionally each stereotype had several tagged values for each property that a SaveCCM element contains. To provide a SaveCCM semantics, constraints implemented in OCL were used.

Since SaveCCM is suitable for analysis of systems from vehicular safety systems domain, it is very useful to develop transformations between UML and SaveCCM models. The SaveUML transformation tool developed within this thesis transforms XML-based representations of models using XSL transformations. The standard format for exchange of UML models is XMI, however there are some differences in generated XMI representations of models in various tools. Therefore, current transformations are dependable on the specific tool used for modelling (in this case IBM RSM ). The possibility of transforming XMI files instead of transforming specific tool format (*emx* files) should be reconsidered in future development of the transformation tool.

# References


**[Åkerholm 05]** Åkerholm, M.; Möller, A.; Hansson, H.; Nolin, M.: *"Towards a dependable component technology for embedded system applications"*, 10th IEEE International Workshop on Object – Oriented Real – Time Dependable System, 2005.

**[Åkerholm 07]** Åkerholm, M. et al: *"The Save approach to component – based development of vehicular systems"*, The Journal of Systems and Software 80 (2007)

**[Bass 98]** Bass, L.; Clements, P.; Kazman, R; *Software Architecture in practice*, Boston, MA.: Addison – Wesley, March 1998

**[Booch 93]** Booch, G.*: Software components with Ada: Structure, Tools and Subsystems*, 3rd ed., Reading, MA: Addison – Wesley, 1993.

**[Crnković 02]** Crnković, I.; Larsson, M.: *Building Reliable Component – Based Software systems*, Artech House, 2002.

**[Gao 03]** Gao, J.; Tsao, J.; Wu, Y.: *"Testing and Quality Assurance for Component – Based Software"*, Boston: Artech House, 2003.

**[Hansson 04]** Hansson, H.; Åkerholm, M.; Crnkovic, I.; Törngren, M.: *"SaveCCM – a component model for safety-critical real-time systems"*, Proceedings of 30th Euromicro Conference, Special Session Component Models for Dependable Systems, 2004.

**[Lüders 06]** Lüders, F.: *"An evolutionary approach to software components in embedded real – time systems"*, Dissertation, Mälardalen University, Västerås, Sweden, 2006.

**[Meyer 92]** Meyer, B: *"Applying 'Design by contract'"*, IEEE Computer 25, 10, October 1992

**[Meyer 97]** Meyer, B.: *Object – Oriented Software Construction*, 2nd ed., London, UK: Prentice-Hall International, 1997.

**[OCLs 06]** OMG: *Object Constraint Language Specification*, version 2.0, May 2006.

**[SaveD 07]** Monot, A.; Noyrit, F.: *SaveIDE – Developer Documentation*, October, 2007.

**[SaveR 07]** Håkansson, J.: *The SaveCCM Language Reference Manual*

**[Selic 07]** Selic, B.: "*A Systematic Approach to Domain-Specific Language Design Using UML"*, 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing, 2007.

**[SPdes 08]** SaveUML project: *SaveUML Design description*, version 1.1, January 2008.

**[SPfin 08]** SaveUML project: *SaveUML project Final presentation*, January 2008.

**[SPprof 08]** SaveUML project: *SaveUML profile specification,* version 1.0, January 2008.

**[SPusr 08]** SaveUML project: *SaveUML Users manual*, version 1.0, January 2008.

**[SUo 07]** SaveUML project: *Overview of the UML Component diagram*, 2007.

**[SunEJB 05]** Sun Microsystems: *Enterprise JavaBeans Specification*, Version 3.0 Final Release, 2005.

**[Szyperski 99]** Szyperski, C.: *Component Software – Beyond Object-Oriented Programming*, Reading, MA: Addison – Wesley, 1999.

**[UMLi 07]** OMG: *Unified Modeling Language Infrastructure Specification*, Version 2.1.1, February 2007.

**[UMLs 07]** OMG: *Unified Modelling Language Superstructure Specification*, Version 2.1.1, February 2007.

**[UMLu 07]** Rumbaugh, J.; Jacobson, I.; Booch, G.*: "The Unified Modelling Language Reference Manual"* 2nd ed., MA; Addison – Wesley, July 2004.

**Web resources**

**[EJB web]** Enterprise JavaBeans : *http://java.sun.com/products/ejb/*, April 2008.

**[IDL web]** OMG Interface Definition Language : *http://www.omg.org/gettingstarted/omg_idl.htm*, April 2008.

**[MOF web]** OMG Meta Object Facility . *http://www.omg.org/mof/*, April 2008.

**[MSCOM web]** Microsoft COM : *http://www.microsoft.com/com*, April 2008.

**[MS.NET web]** Microsoft .NET framework : *http://msdn.microsoft.com/netframework*, April 2008.

**[OMG web]** Object Management Group : *http://www.omg.org*, April 2008

**[ORPC web]** The Open Group :Remote Procedure Call, Document Number: C706, *http://www.opengroup.org/onlinepubs/9629399/*, April 2008.

**[Rationalweb]** IBM Rational Software Modeller web page: *http://www.ibm.com/software/awdtools/modeler/swmodeler/*, April 2008.

**[Save web]** SAVE project web page : *http://www.mrtc.mdh.se/SAVE,* April 2008.

**[SaveIDE web]** SaveIDE tool web page : *http://save-ide.sourceforge.net/*, April 2008.

**[SP web]** SaveUML project web page: *http://www.fer.hr/rasip/dsd/projects/saveuml* , April 2008.

**[UML web]** OMG Unified Modelling Language : *http://www.uml.org/*, April 2008.

**[W3C web]** World Wide Web Consortium: *http://www.w3.org*, April 2008

**[XMI web]** OMG XML Metadata Interchange : *http://www.omg.org/technology/xml/index.htm*, April 2008.

**[XML web]** eXtensible Markup Language : *http://www.w3.org/XML/*, April 2008.

**[XSLT web]** XSLT web page: *http://www.w3.org/TR/xslt*, April 2008.