UNIVERSITY OF ZAGREB
and
MÄLARDALEN UNIVERSITY

THESIS

TRANSFORMATION BETWEEN JAVABEANS AND SAVECOMP SOFTWARE COMPONENT MODELS

Juraj Feljan

Zagreb, July 2008

Abstract

Component-based software engineering (CBSE) is an approach where systems are built from preexisting software components. To ensure interoperability between components, component specifications conform to a particular component model. Many component models exist and there is often a need for transformation between two different component models.

In this thesis the basic concepts of CBSE are described. Two component models are presented – JavaBeans and SaveCCM. A transformation between them is discussed and implemented.

Key words: component-based software engineering, transformation between component models, JavaBeans, SaveCCM

Sažetak

Programsko inženjerstvo temeljeno na komponentama je pristup koji razmatra izgradnju sustava od postojećih programskih komponenti. Ne bi li se osiguralo zajedničko funkcioniranje komponenti, njihove specifikacije slijede određeni komponentni model. Razvijeni su mnogi komponentni modeli pa se javlja potreba za transformaciju među njima.

U ovom radu su opisani osnovni koncepti programskog inženjerstva temeljenog na komponentama. Dva komponentna modela su prezentirana – JavaBeans i SaveCCM. Transformacija među njima je diskutirana i izvedena.

Ključne riječi: programsko inženjerstvo temeljeno na komponentama, transformacija između komponentnih modela, JavaBeans, SaveCCM

Table of Contents

1 Introduction 1
2 Component-based software engineering 2
  2.1 Components 4
  2.2 Interfaces 5
  2.3 Contracts 6
  2.4 Component models 7
  2.5 Component frameworks 8
  2.6 Component-based software development process 8
    2.6.1 Development of components 8
    2.6.2 Development of systems 9
  2.7 CBSE versus OOP 9
  2.8 CBSE for embedded systems 9
3 The JavaBeans component model 11
  3.1 Properties 13
    3.1.1 Simple properties 14
    3.1.2 Indexed properties 14
    3.1.3 Bound properties 14
    3.1.4 Constrained properties 14
  3.2 Events 14
  3.3 Methods 15
  3.4 Customization 15
  3.5 Introspection 16
    3.5.1 Design patterns 17
      3.5.1.1 Design patterns for properties 17
      3.5.1.2 Design patterns for events 18
      3.5.1.3 Design patterns for methods 18
    3.5.2 Introspection API 18
  3.6 Persistence 18
    3.6.1 Default serialization using the Serializable interface 18
    3.6.2 Selective serialization using the transient keyword 19
    3.6.3 Selective serialization using the writeObject and readObject methods 19
    3.6.4 Selective serialization using the Externalizable interface 19
    3.6.5 Long term persistence 19
  3.7 Packaging Java beans 19
4 The SaveComp Component Model 21
  4.1 Ports 22
  4.2 Components 23
    4.2.1 Clocks and delays 23
    4.2.2 Composite components 24
  4.3 Switches 24
  4.4 Assemblies 25
  4.5 SaveCCM XML syntax 25
  4.6 SAVE-IDE 29
5 Transformation from SaveCCM to JavaBeans 31
  5.1 Development of the SaveCCM Java model 31
    5.1.1 Executor 32
    5.1.2 Immediate connections and delegations 33
    5.1.3 Ports 33
    5.1.4 Components 34
    5.1.5 Clocks and delays 36

5.1.6 Unsupported elements 36
5.2 Implementation of the transformation 36
5.2.1 Parsing the .save file 36
5.2.1.1 SaveCCM DTD and SaveCCM schema 37
5.2.1.2 Changes to the savexmlgenerator.mt file 38
5.2.2 The unmarshall method 40
5.2.3 The copyClasses method 40
5.2.4 The generateDataPortClasses method 40
5.2.5 The generateComponentClasses method 40
5.2.6 The generateSystemDescriptionClass method 41
5.3 Limitations of the transformation 41
5.4 Performing the transformation 42
5.5 Transformation example 43
5.6 Possibilities for improvement 44
5.7 SaveCCM and SAVE-IDE errors 45
6 Conclusion 47
7 Bibliography 48
8 List of abbreviations 50

Index of Figures

Figure 2.1: A component and its interface 6
Figure 2.2: Component-based software development process 8
Figure 3.1: Examples of beans 12
Figure 3.2: NetBeans IDE property sheet 16
Figure 4.1: Formal definition of SaveCCM semantics 21
Figure 4.2: SaveCCM graphical syntax 22
Figure 4.3: SaveCCM clock and delay components 24
Figure 4.4: A switch condition 25
Figure 4.5: Application definition 26
Figure 4.6: Component, switch and assembly type definitions 27
Figure 4.7: Component realisation definition 28
Figure 4.8: Component, switch and assembly instantiation 28
Figure 4.9: Port definitions 29
Figure 4.10: Connection definition 29
Figure 4.11: SAVE-IDE Architectural Editor 30
Figure 5.1: Transformation from SaveCCM to JavaBeans 31
Figure 5.2: UML diagram of a partial SaveCCM Java model 32
Figure 5.3: Promoting trigger events 34
Figure 5.4: UML diagram of the Component class 35
Figure 5.5: JAXB binding compiler 37
Figure 5.6: JAXB binding runtime framework 37
Figure 5.7: The Save2Java tool 42
Figure 5.8: A simple SaveCCM system 43

Index of Code Snippets

Snippet 5.1: Executing the components 33
Snippet 5.2: Methods implementing the transformation 36
Snippet 5.3: Unmarshalling the .save file 40
Snippet 5.4: Validation of the .save file against the SaveCCM schema 40
Snippet 5.5: The execute method in the generator1_1 class 43
Snippet 5.6: The execute method in the generator2_4 class 43
Snippet 5.7: The execute method of the comparator_7 class 44
Snippet 5.8: The execute method of the print_11 class 44

Index of Tables

Table 5.1: Changes to the savexmlgenerator.mt file 39

1 Introduction

Today we are witnesses of a great expansion of software use in industry, business, traffic, health, research, education, entertainment, literally all aspects of everyday life. Software runs banks, hospitals, cars, planes, schools, it has become inevitably intertwined with our daily routine. But software systems are becoming extremely large and complex, thus increasing the time and cost to develop and maintain them. So methods for navigating and simplifying the whole software lifetime cycle need to be defined. The discipline handling this issue is *software engineering*. Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software [1]. A promising new sub discipline of software engineering recently emerged – it is called *component-based software engineering* (CBSE).

Component-based software engineering is an approach where systems are built from preexisting software components. The advantage of this approach is that components can be developed separately from systems. To ensure interoperability between components, component specifications conform to a particular component model. A component model defines the rules for component specification and communication between components. Many component models exist and there is often a need for transformation between two different component models. Although component models are in principle similar, many details in which they may differ can make the transformation difficult to achieve.

The purpose of this thesis is to examine the possibility of transformation between JavaBeans and SaveComp component models. First a theoretical introduction to CBSE is given in Chapter 2. Then key aspects of JavaBeans are described in Chapter 3. SaveCCM is presented in Chapter 4. The main part of this thesis is Chapter 5 - a discussion about and implementation of the transformation.

The reader of this thesis should have prior knowledge of the Java programming language and basic knowledge of XML in order to understand the transformation implementation.

2 Component-based software engineering

Traditionally, software was developed with a single system in focus, by concentrating on meeting the budget and delivery deadline, with no evolutionary aspects in mind. This often led to failing to fulfil quality requirements and the increase in maintenance costs. Today the main challenge for programmers is to cope with the mentioned complexity and react quickly to change. Also, there is an increasing demand for software to be robust, reliable, flexible, adoptable etc. One possible way to handle this is by applying c*omponent-based development* (CBD), an approach where systems are built from preexisting components, as opposed to building a system from scratch.

CBD is a concept well known in building hardware systems. For instance, connect various integrated circuits (ICs) together in the correct way and you get a radio. Cars are made from preexisting components. Another example is assembling computers – a customer can pick out the components (CPU, memory, hard disk) and get a computer that meets their exact desires and needs. Development of software systems is progressively moving in a similar direction – software components should give programmers the same benefits as hardware components do to computer assemblers. Programmers should be able to connect software components together building different applications, just as an electrical engineer connects ICs when building radios.

In contrast to a traditional approach of “reinvent, recode, retest”, CBD promotes a new approach – “reuse”. Reusability can reduce production costs, while allowing faster development of new software. Since a component is intended for use in different systems it can be repeatedly tested in various contexts, thus increasing its quality.

CBD introduces many advantages to software development such as:

- increased understandability of complex systems, which allows easier system development and maintenance,
- lower development and maintenance costs,
- shorter time to market,
- reuse of components across several systems.

But it also holds many pitfalls [2]:

- It is sometimes unclear what can and can not be reused.
- It takes more time and effort to build units that are reusable.
- Reusable components are intended for use in many different, and possibly unknown applications, which makes it difficult to define precise requirements.
- To be reusable, components must be sufficiently general, scalable and adoptable, which makes them more complicated to use and more demanding on computing resources.
- Component maintenance can be very expensive, since components must respond to different requirements and run in different environments.

CBD needs a systematic and disciplined approach that can utilize its advantages and neutralize the risks. This is the role of *component-based software engineering* (CBSE). CBSE is a branch of software engineering focused on developing software components and systems constructed from those components. It provides methods and tools for supporting different aspects of CBD[1].

The idea behind building software from prefabricated components is not new – it was first published in Douglas McIlroy's address at the NATO conference on software engineering in Garmisch, Germany in 1968. However, CBSE is merely in its starting phase. It covers many software engineering disciplines which have not yet been fully defined and exploited. But it is expected to improve the development of software in general. It is considered to be a big step in software development methodology, similar to the transition from assembly to high level programming languages around 1970, or the jump from procedural to object-oriented programming languages around 1990. CBSE promises to accelerate the software development process, reduce its cost and ease the maintenance of software systems.

The major goals of CBSE are [3]:

- providing support for the development of systems as assemblies of components,
- supporting the development of components as reusable entities,
- facilitating the maintenance and upgrading of systems by customizing and replacing their components.

Some of the main challenges CBSE is facing are [2]:

- component specification – still no consensus has been made about what software components exactly are and how they should be specified,
- component-based software life cycle
  - the development phase – the development of components may be completely independent of the development of systems using them, so components usually lack features that the system requires
  - the maintenance phase – it is unclear who is responsible if the system fails, the system producer or the component producer,
- composition predictability – even if all attributes of components are known, it is unclear as how they define the attributes of the whole system,
- tool support – the objective of CBSE is to build systems from components simply and efficiently, this can only be achieved through extensive tool support (for instance component selection and evaluation tools, component repositories, component-based design tools, run-time system analysis tools, etc.).

The assembly of components should be smooth and simple in an ideal world. A system incorporating components should know everything about them – their interfaces, their

1 CBD and CBSE are sometimes used as synonyms. In this thesis CBSE is considered to be a systematical and disciplined approach to CBD.

functional and non-functional[2] properties. Also, the components should know exactly what the system needs. But in the real world systems are built from already developed components only when appropriate and often by developing new code for the specific system. The system may know about the syntax of components' interfaces, but not necessarily their other properties. So connecting components in the real world can get quite complicated, thus emphasizing the need for CBSE [2].

The basic concepts of CBSE are: components, interfaces, contracts, component models and component frameworks. They are described in next chapters.

2.1 Components

A *software component* is the fundament of CBSE. Although this concept may intuitively be clear or obvious, still there is no agreement as to what a software component exactly is. Not long ago the default notion of a component was the software module, because both a module and a component adhere to the concept of building parts of a software system. But in recent years, as CBSE evolved, experts have provided several definitions for the term software component [4].

One of the most accepted definitions is given by Clement Szyperski: "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third party."

This definition highlights the component properties that are not addressed in traditional software modules – context independence, composition, deployment and contracted interfaces.

For a component to be deployed independently there must be a clear distinction from its environment and other components. Since a component communicates with the environment via its interface, the interface must be clearly specified. The implementation must be well encapsulated in the component and not directly reachable from the environment. This makes a component a unit of third-party[3] deployment [2]. The motivation for third-party deployment is quite clear – the use of components should not depend on the tools and knowledge about the component that is only in the possession of the component provider. A component-based system can consist of components from multiple, independent sources.

A component must have a meaningful functionality by itself. It is a unit of deployment, so it is never deployed partially. It must explicitly specify its needs, i.e. what the deployment environment has to provide in order for the component to function. A component must be specified in a way that it is possible to connect it with other components and integrate it into systems in a predictable way. Component integration and deployment should be independent of the component development life cycle and there should be no need to recompile or relink the application when updating with a new component.

2 Also known as extra-functional properties or quality attributes. They include performance, required resources, reliability, availability, accuracy, worst case execution time, latency, security etc.

3 A third party is one that can not be expected to have access to the construction details of the component.

The most important feature for a component is the separation of the interface from the implementation. The component is visible exclusively through its interface. Hence, there is a need for a complete specification of a component, including its functional interface, non-functional characteristics, use cases, tests and so on. Unfortunately, current technologies fail in specifying semantics and non-functional properties.

A component has two parts: an interface and some code. The interface is the only point of access to the component. Thus it should ideally contain all of the information that users need to know about the component: what it does, how and where it can be deployed, its context dependencies. The code, however, should be completely inaccessible. The specification of a component therefore must consist of a precise definition of the component's operations and context dependencies. In current practice, component specification techniques specify components only syntactically. Specification of semantics and non-functional properties of components is still an open area of research [2].

It takes significant effort to write a software component that is effectively reusable. The component needs to be:

- fully documented,
- more thoroughly tested,
- built with an awareness that it will be put to unforeseen uses.

2.2 Interfaces

Interfaces are access points to components. They define the means to connect to the component. Clients access components through their interfaces, since they are the only visible part of the component. A component can have multiple interfaces, if it for instance offers multiple services.

Interfaces provide no implementation whatsoever, they just name a set of operations the component can perform. As already stated, it is essential that interfaces remain separated from the implementation. This allows two things [2]:

1. an implementation can be changed without tampering with the interface, so the system does not need to be rebuilt,
2. new interfaces and implementations can be added without change to the existing interface.

There are two kinds of interfaces (Figure 2.1):

1. exported (provided) interface – states what services the component offers to the environment,
2. imported (required) interface – defines what services the component requires from the environment.

Figure 2.1: A component and its interface

Interface specification in current technologies has two serious shortcomings. First, the semantics of the operations should be specified by the interface, which is not the case. Interfaces only specify their syntactics, i.e. the inputs and outputs and give very little information about what the component does. Also, semantic information about context dependencies of components (development and deployment environments) can not be expressed by interfaces. The second drawback is that interfaces are only capable of specifying the functional properties of the component. Non-functional properties are not handled by them. This has resulted in the need for a facility that clearly specifies behaviours of components.

2.3 Contracts

Contracts provide a more accurate specification of the component's behaviour. Although interfaces and contracts may seem alike, they are different concepts. An interface is a collection of operations of a component, while a contract specifies the behavioural aspects of a component or the interaction between different components.

For each operation the contract states the [2]:

- invariant – a list of constraints that the component will maintain,
- precondition – a list of constraints that need to be met by the client,
- postcondition – a list of constraints the component promises to establish.

The client has to establish the precondition before executing an operation. The component can rely on the precondition being met before a call to the operation. The component has to establish the postcondition before returning to the client. The client can rely on the postcondition being met when the operation returns.

Besides specifying the behaviour of a single component, contracts are used to specify interactions between groups of components. In this context they specify [2]:

- the participating components,
- the role of each participating component,
- the invariant to be maintained by the components,
- the specification of the methods that instantiate the contract.

Hierarchically contracts can be divided into levels [5]:

1. Syntactic level. Specifies the operation a component can perform and input/output parameters a component requires.
2. Behavioural level. Specifies pre and postconditions for an operation.
3. Synchronization level. Specifies synchronization between different services and method calls.
4. Quality of service level. Specifies non-functional properties.

2.4 Component models

A *component model* enables interoperability between components, it specifies, at an abstract level, the standards and conventions which developers and users of components must follow. Compliance with a component model is one of the properties that distinguishes components from other software entities [2]. In that context Bill Councill and George T. Heineman define a component as a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard. They also state that a component model 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.

Component models prescribe how components interact with each other and impose the following standards and conventions [6]:

- Component types. A component model requires that components implement one or more interfaces. In this way a component model defines one or more component types. Different component types can play different roles in systems and participate in different types of interaction schemes.
- Interaction schemes. Component models specify how components are located, which communication protocols are used and how qualities of service such as security and transactions are achieved. A component model describes how components interact with each other or how they interact with the component framework (more in Chapter Component frameworks).
- Resource binding. The process of composing components is a matter of binding a component to one or more resources. A resource is either a service provided by a framework or by some other component deployed in that framework. A component model describes which resources are available to components, and how and when components bind to these resources.

In a way, component models define the architecture of systems. This limits the flexibility of systems, but also speeds up the development process.

The most important industrial component models currently are CORBA, COM/DCOM/COM+, .NET, JavaBeans and Enterprise JavaBeans.

2.5 Component frameworks

A *component framework* is an implementation of services that support or enforce a component model [6]. The main purpose of a framework is to endorse the process of component composition. A framework is a support infrastructure for the component model.

A component framework can be viewed as an operating system for components. From that viewpoint, components are to framework what processes are to the operating system. The difference is that component frameworks are more compact than operating systems. They are specialized to support a limited range of component types and interactions between those types. By limiting the diversity, component composition becomes simpler, more robust and more predictable.

2.6 Component-based software development process

The development process of component-based software is divided into two processes:

- development of components and
- development of systems.

In many real situations these processes will be combined, maybe even not distinguished as separate activities. However, their separation is possible and desirable. It allows parallel development which results in shorter time-to-market. It also enables organization of a global component market.

In the component development process components are developed and stored in a common component repository. In the system development process, components are selected from the component repository and used to build systems. This is shown in Figure 2.2. For the component development, design for reuse is the main concern. For the system development, the emphasis is on finding the proper components and verifying them [7].

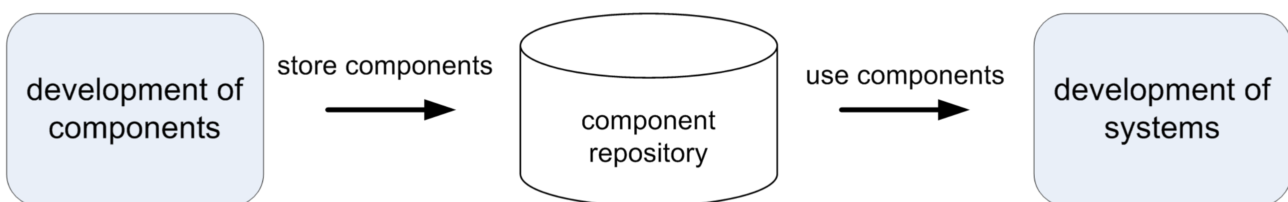


Figure 2.2: Component-based software development process

2.6.1 Development of components

Components are developed to be fully reusable and available for composition by third party. Building reusable units requires great design and development efforts. There is no, or very little knowledge about the systems the component will be used in. Thus, managing requirements is more difficult. A precise component specification is required. The components should be tested in isolation, but also in different configurations.

2.6.2 Development of systems

In the requirements analysis and specification phase it is important to analyse if requirements can be fulfilled by available components. Since appropriate components can not always be found, there is a risk that the new components have to be developed. As this can be expensive or time consuming, another possibility is negotiating the requirements and modifying them so existing components can be used. Components will often implement a generalized functionality to be reusable. Thus, most components have to be adapted to fit the system design.

The maintenance stage of a system life cycle is based on updating, replacing or adding new components. This approach makes the maintenance easier, and a component-based system is more suitable for changes, than a monolithic one. After replacing a component the whole system does not have to be rebuilt.

2.7 CBSE versus OOP

CBSE and the object oriented paradigm (OOP) share some basic concepts. For instance, both promote reusability. The terms object and component are often thought to be synonymous or very similar. However, there are important distinctions between objects and components [8]:

- Components can be collection of objects.
- Components often use persistent storage, whereas objects have local state.
- Components have a more extensive set of intercommunication mechanisms than objects, which usually use the messaging mechanism.
- Components are often larger units of granularity than objects and have complex actions at their interfaces.

The idea in OOP is that software should be written according to a mental model of the objects it represents. OOP focuses on modelling real-world interactions and attempts to create “verbs” and “nouns” which can be used in intuitive ways. CBSE, by contrast, makes no such assumptions and instead states that software should be developed by gluing preexisting components together, much like in the field of electronics or mechanics. It accepts that the definitions of useful components, unlike objects, can be counter-intuitive.

2.8 CBSE for embedded systems

According to Michael Barr an *embedded system* is a special-purpose computer system designed to perform one or a few dedicated functions. This is in contrast with general-purpose computers, such as a personal computers, which can do many different tasks depending on programming. Embedded systems control the majority of the common devices in use today, in range from portable devices such as digital watches and MP3 players, to large stationary installations like traffic lights or factory controllers. 98% of all computer systems belong to embedded systems [9].

Traditionally embedded systems are still developed in an assembler or in C. Current

monolithic and platform-dependent embedded systems are difficult to port, upgrade and customize. They offer very limited opportunities for reuse. CBSE for the embedded domain should enable reuse of once developed parts and provide a way for more efficient development that results in more reliable systems.

Currently, there is a lack of widely adopted component technology standards which are suitable for embedded systems. Tools that support component based development are still missing. There is a lack of efficient implementations of component frameworks, which have low requirements on memory and processing power [9]. At this time there are several component models trying to cope with the mentioned problems – Koala, Pecos, SaveCCM. CBSE methods for embedded systems are still in the research phase.

Most of embedded systems need to work in real-time and often have a safety-critical role. Due to their need to blend into the environment they usually have very limited memory and processing power at their disposal. In many domains their life cycle can stretch to several decades, thus demands on reliability, robustness and availability are important [9].

Most general purpose component models (JavaBeans, EJB, .NET, CORBA etc.) are built to maximize the efficiency of the development process, counting on the powerful hardware to deal with the heavy overhead of the model and the framework. They focus on functionality, flexibility, run-time adaptability, simpler development and maintenance. However, the design of embedded systems must consider additional constraints [10]:

- Embedded systems must satisfy constraints on non-functional properties such as timing, reliability, robustness, availability etc.
- It is often important that functional and non-functional properties are statically predictable, in particular if the system is safety-critical.
- Embedded systems must often operate with scarce resources (including processing power, memory, communication bandwidth).

3 The JavaBeans component model

The JavaBeans technology is a portable, platform-independent software component model for the Java SE platform. It was introduced in 1997. Since then it has not changed much, but has grown in significance. The technology encompasses a Java package (`java.beans`) and a document (JavaBeans specification). The document describes how to use the classes and interfaces from the package to implement "beans functionality" [11].

The basic idea behind JavaBeans is this: a Java bean[4] is a Java class that complies to certain conditions. Almost every software component is a class. What makes it a component is its conformance to a software component specification. The JavaBeans specification is a document that describes what a Java class must "do" to be considered a Java bean [11].

The JavaBeans component model is relatively simple, the whole specification ([12]) is a 114 page document (in comparison with, for instance, the 562 pages of Enterprise JavaBeans[5] specification, or 350 pages of CORBA specification). It focuses on making small lightweight components easy to implement and use, while making heavyweight components possible. Basic JavaBeans concepts can be learned very quickly, little effort is needed to start writing and using simple beans [12].

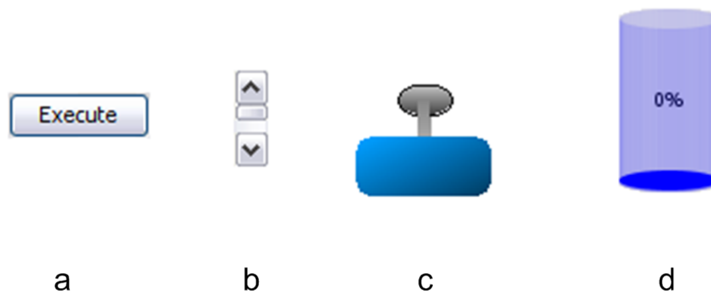
The JavaBeans specification defines a bean as a reusable software component that can be manipulated visually in a builder tool. This covers a wide range of different possibilities, as builder tools can be web page builders, visual application builders, GUI layout builders, even server application builders. For instance, the NetBeans IDE GUI builder and Eclipse Visual Editor are builder tools that allow visual building of GUIs, using Swing[6] and custom developed Beans. Beans may be simple GUI elements such as buttons and sliders, or sophisticated visual software components such as database viewers or data feeds (Figure 3.1[7]).

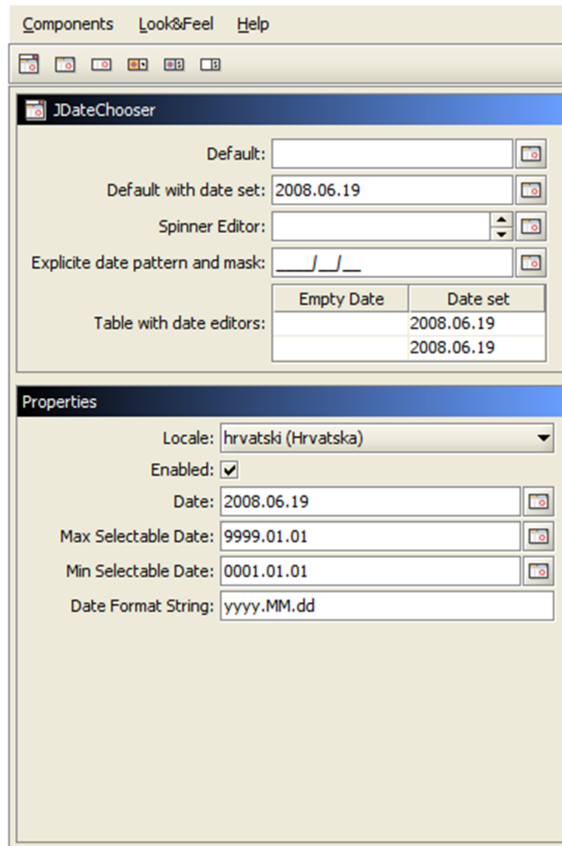
4 The term "JavaBeans" stands for the technology, while the term "Java bean" signifies a particular software component that conforms to the JavaBeans component model.

5 It is important to differentiate between JavaBeans and Enterprise JavaBeans. Both technologies are software component models, use a similar name and are implemented in Java. However their purpose and architecture are different. JavaBeans is a general purpose component model, while Enterprise JavaBeans is a component model specific for the enterprise environment.

6 Swing is the Java SE framework for GUIs. All common GUI components (buttons, panels, check boxes etc.) from Swing are Java beans.

7 Bar and tank bean images are from Mikulaj M.: Komponentno programiranje Java Beans tehnologijom, thesis no. 1485, Faculty of Electrical Engineering and Computing, University of Zagreb, 2006.
Calendar bean image is from Toedter K: JCalendar, 2006, http://www.toedter.com/en/jcalendar/index.html.





e

a - button bean
b - scroll bean
c - bar bean
d - tank bean
e - calender bean

Figure 3.1: Examples of beans

This visual manipulation is a key aspect of the technology. However, although beans are primarily targeted at builder tools, they are also entirely usable by human programmers, as their use is not dependent on builder tools.

Each Java bean has to be able to run in two different environments. First it needs to be capable of running inside a builder tool. This is referred to as the design environment or design-time. The bean must be able to provide the builder tool with design information, so a user is able to customize it. For this customization process a lot of extra baggage (metadata, property editors, customizers, icons etc.) is carried by the bean. The bean must also be able to be used during run-time within a generated application. During run-time there is much less need for customization of the behaviour and appearance, so a bean

carries less baggage than during design-time [13].

Many beans have a strong visual aspect, but while this is common, it is not required. Beans can be visual or non-visual (invisible). The GUI representation of beans may be the most obvious and compelling part of the JavaBeans technology. However, it is possible to implement non-visual beans that have no GUI representation. These beans are still able to call methods, fire events etc. They are also represented visually in a builder tool, so they can be configured. They simply have no screen appearance of their own [12]. In other words, non-visual beans are invisible only at run-time, but are visible during design-time. Visual beans are visible both during design-time and run-time.

For instance, a two dimensional graph bean or a calendar bean must have a visual representation to be useful. A database connection bean or a spelling bean do not require a GUI representation in an application, however they must be visible during design-time in order to be configured.

Individual Java beans vary in the functionality they support, but their typical unifying features are:

- properties,
- events,
- methods,
- customization,
- introspection and
- persistence.

These are key concepts of the JavaBeans technology and are discussed in the following chapters.

3.1 Properties

A *bean property* is a named attribute of a bean that can affect its behaviour or appearance [14]. Examples of bean properties include colour, label, font etc. Properties can have arbitrary types, including both primitive types and class or interfaces types.

Properties are accessed via method[8] calls on their owning object. For readable properties there is a getter method to read the property value. For writeable properties there is a setter method to allow the property value to be updated [12].

There are four types of properties defined in the JavaBeans specification:

1. simple,
2. indexed,
3. bound and
4. constrained properties.

8 These are accessor methods, or the getter and setter method.

3.1.1 Simple properties

A *simple property* has a single value whose changes are independent from other properties.

3.1.2 Indexed properties

Indexed properties support a range of values instead of a single value. It is possible to read or write a single element or the whole array corresponding to the indexed property.

3.1.3 Bound properties

Sometimes when a bean's property changes, another object might need to be notified of the change and react to it. These are *bound properties*. Whenever a bound property changes, a notification of the change is sent to interested listeners. The `PropertyChangeEvent` class encapsulates property change information. Listeners must implement the `PropertyChangeListener` interface.

Bound properties are normally used when a number of beans want to keep a shared value. For instance, for maintaining a common background colour. When one bean changes its background colour, the change is then promoted to all beans registered as listeners of that property change. That way they can adjust their background colours too.

3.1.4 Constrained properties

Constrained properties are similar to bound properties. When a constrained property changes, an event is generated. However, the change is not necessarily accepted by the listeners, it first needs to be validated. If the change is not appropriate for a listener, it can be rejected, i.e vetoed by throwing a `PropertyVetoException`.

The setter method for a constrained property needs to be implemented in the following way:

1. Save the old value in case the change is vetoed.
2. Notify listeners of the new proposed value, allowing them to veto the change.
3. If no listener vetoes the change, set the property to the new value.

If a registered listener vetoes a proposed property change, the source bean with the constrained property must:

1. Catch the `PropertyVetoException`.
2. Revert to the old value of the property.
3. Inform the listeners that the old value is restored [14].

3.2 Events

Beans use the Java Event Model for communication. *Events* provide a convenient mechanism for allowing beans to be plugged together in a builder tool. Builder tools can discover which events a bean can fire (more in Chapter Introspection).

For a bean to be the source of an event, it must implement methods that add and remove listeners for that type of event. For a bean to receive an event, it must implement an event listener interface.

3.3 Methods

The *methods* of a bean are normal Java methods which can be called from other objects. A bean's methods represent its interface, a point for bean access and manipulation.

3.4 Customization

When a user is composing an application in a builder tool he needs to be able to customize the beans he/she is using. *Customization* is a process of modifying the appearance and behaviour of a bean within a builder tool, so that it meets the user's specific needs. Customization is done at design-time and can be done in two ways:

- by using property editors and
- by using customizers.

A *property editor* is a class that allows GUI editing of a property. Bean developers may provide property editors for any new data types that they deliver as part of their bean. Property editors for known Java types are usually integrated in builder tools. A builder tool can find out what properties a bean has (more in Chapter Introspection). These properties are then used to construct a GUI property sheet that lists the properties and provides a property editor for each property. A property sheet from NetBeans IDE is shown in Figure 3.2. The user can then use this property sheet to update the various properties of the bean [12]. Property editors must implement the `java.beans.PropertyEditor` interface.

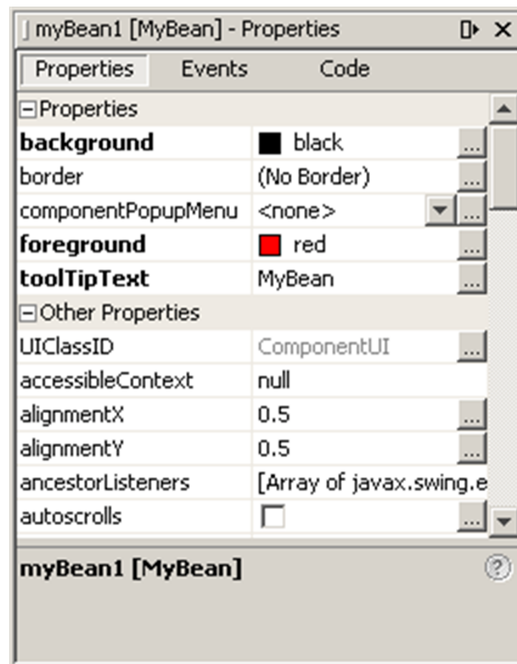


Figure 3.2: NetBeans IDE property sheet

Property editors are normally used with small or medium sized beans. Often it is undesirable to have all the properties of a bean listed on a single, sometimes huge property sheet. Thus, for more complex beans a more sophisticated means of customization is provided. For example, a bean developer should be allowed to write customization wizards which guide the user through a step by step customization. This is done using *customizers*.

A customizer is a class that specifically targets a bean's customization. Customizers are used where sophisticated instructions are needed to configure a bean and where property editors are too primitive to achieve bean customization [14]. Each customizer should inherit either directly or indirectly from `java.awt.Component`. They also need to implement the `java.beans.Customizer` interface.

A property editor relates to a single property, while a customizer is associated with the whole bean. A property editor may be instantiated as part of a bean customizer.

3.5 Introspection

Introspection is the automatic process of analysing a bean to reveal its properties, events and methods. Introspection is used by builder tools. By uncovering beans' properties, events and methods, tools provide easy visual manipulation of beans.

Introspection can be done in two ways:

- by using design patterns and
- by using Introspection API.

The idea is to allow bean developers to work entirely in terms of Java and avoid using a

separate bean specification language. This is achieved through *design patterns*, i.e. writing the beans code by following specific rules. In this way, introspection is done automatically on simple beans, without requiring developers to perform extra work to support introspection. However, for more sophisticated beans, the developers are allowed full and precise control over which properties, events and methods are exposed. This is achieved using the Introspection API [12].

A composite mechanism is applied – by default a low level reflection mechanism is used to study a bean's methods. Then design patterns are used to deduce what properties, events and methods the bean possesses. However, if a bean developer chooses to provide a class describing the bean, then this class is used to programmatically discover the bean's behaviour [12].

3.5.1 Design patterns

Design patterns refer to using conventional names and type signatures for methods. Builder tools that recognize design patterns can be written and used to analyse and understand beans. Design patterns are also a useful documentation hint for human programmers. By identifying particular methods as standard design patterns, programmers can understand and use new classes more quickly [12].

3.5.1.1 Design patterns for properties

The getter method for a simple property must start with "`get`", followed by the name of the property, with the first letter of the property capitalized. The setter method is written similarly. For a property of type `<PropertyType>` named `<propertyName>` the accessor methods would have following signatures:

```
public <PropertyType> get<PropertyName>();
public void set<PropertyName>(<PropertyType> a);
```

An indexed property is specified by following methods:

```
// methods to access individual values
public <PropertyType> get<PropertyName>(int index);
public void set<PropertyName>(int index, <PropertyType> a);

// methods to access the entire indexed property array
public <PropertyType>[] get<PropertyName>();
public void set<PropertyName>(<PropertyType>[] a);
```

The accessor methods for bound properties are defined in the same way as those for simple properties. However, event listener registration methods need to be provided.

The accessor methods for constrained properties are defined in the same way as those for simple properties, with the addition that the setter methods can throw a `PropertyVetoException`:

```
public void set<PropertyName>(<PropertyType> a) throws PropertyVetoException;
```

3.5.1.2 *Design patterns for events*

The source of an event must define the methods for registering listeners of those events. The standard design pattern for listener registration is:

```
public void add<ListenerType>(<ListenerType> listener);
public void remove<ListenerType>(<ListenerType> listener);
```

3.5.1.3 *Design patterns for methods*

By default all `public` methods are exposed. This includes any property accessor methods and any event listener registry methods.

3.5.2 Introspection API

The use of design patterns is optional. If a programmer is prepared to explicitly specify the bean's properties, events and methods, then the methods can be named arbitrarily. This is done by providing a class that implements the `java.beans.BeanInfo` interface. This interface defines a set of methods that allow developers to provide explicit information about their beans. By specifying a `BeanInfo`, a developer can hide methods, specify a bean icon, provide descriptive names for properties, define which properties are bound properties etc. Apart from the `BeanInfo` interface, in the `java.beans` package there are other classes and interfaces used for manually specifying a bean.

3.6 Persistence

In computer science, *persistence* refers to the characteristic of data to outlive the execution of the program that created it. Without this capability, data only exists in RAM, and will be lost when the memory loses power.

The mechanism that makes persistence possible is called *serialization*. Object serialization means converting an object into a data stream and writing it to storage. A serialized object can then be reconstructed by *deserialization* [14].

All beans have to persist. To do so, they must support serialization by implementing either the `java.io.Serializable` interface or the `java.io.Externalizable` interface. These interfaces offer the choices of automatic serialization and customized serialization. A class is serializable if it or a parent class from the class hierarchy implements `Serializable` or `Externalizable` [14].

3.6.1 Default serialization using the `Serializable` interface

The `Serializable` interface enables automatic serialization. It declares no methods, but acts as a marker signalling that the bean is serializable. Classes that implement `Serializable` must have access to a no-argument constructor of supertype. This constructor is called when a bean is restored from a `.ser` file. `Serializable` does not have to be implemented in a class if it is already implemented in a superclass. All fields

except those marked `static` and `transient` are serialized [14].

3.6.2 Selective serialization using the `transient` keyword

To exclude fields from serialization in a `Serializable` object, they need to be marked with the `transient` modifier.

3.6.3 Selective serialization using the `writeObject` and `readObject` methods

If a serializable class contains either of the following two methods, then the default serialization is not performed. These methods enable control over serialization of complex objects.

```
private void writeObject(java.io.ObjectOutputStream out) throws IOException;
private void readObject(java.io.ObjectInputStream in) throws IOException,
ClassNotFoundException;
```

3.6.4 Selective serialization using the `Externalizable` interface

The `Externalizable` interface is used when full control over serialization is required, for instance when a bean needs to be saved to or retrieved from a file with a specific format. To use this interface the `readExternal` and `writeExternal` methods need to be implemented. Classes that implement `Externalizable` must have a no-argument constructor.

3.6.5 Long term persistence

Long-term persistence enables beans to be saved in XML format. This is enabled through `java.beans.XMLEncoder` and `java.beans.XMLDecoder` classes.

3.7 Packaging Java beans

Java beans are packaged and delivered in JAR files. One JAR file can contain one or more beans. A JAR file containing beans must have a manifest file, which describes the beans in the JAR.

Each JAR holding beans includes the following:

- Class files representing beans. These entries must have names ending in "`.class`".
- Optional serialized prototypes of beans. These entries must have names ending in "`.ser`".
- Optional help files in HTML format to provide documentation for the beans.

- Optional internationalization information to be used by the bean to localize itself.
- Other resource files needed by the beans (images, sound, video etc.) [12].

4 The SaveComp Component Model

The SaveComp Component Model (SaveCCM) is being developed as part of the SAVE project at Mälardalen University in Västerås, Sweden. It is intended to provide efficient support for designing and implementing embedded control applications for vehicular systems, mainly focusing on the safety-critical and real-time subsystems responsible for controlling the vehicle dynamics (steering, braking etc.). Vehicular systems are usually produced in high volumes using inexpensive hardware. For use in that domain, the component model needs to support the development of systems in which tight constraints on resource usage, real-time and interactions with the environment must be satisfied. System behaviour should be predictable, both functionally and non-functionally (with respect to timeliness and resource usage). This means that predictability and analysability are more important than flexibility. The model should be as restrictive as possible, while still allowing the intended applications to be conveniently designed. SaveCCM was designed with that in mind [15].

SaveCCM is described in the language reference manual ([16]) and in several articles. The reference manual gives a more complete description than the articles, so in this thesis it will be referred to as the SaveCCM specification.

SaveCCM is based on the control flow (pipes and filters) paradigm. Data transfer and control flow are separated, which allows both periodic and event-driven activities, since execution can be initiated by either clocks or external events. This separation also allows components to exchange data without handing over the control. Another aspect of explicit control flow is that the resulting design is analysable with respect to temporal behaviour. Temporal factors (schedulability, response time etc.) are crucial for the correctness of embedded real-time systems [16].

The SaveCCM semantics is formally defined by a two-step transformation, first from the full SaveCCM language to a similar but simpler language called SaveCCM Core, and then into timed automata with tasks [16] (Figure 4.1). The timed automata semantics enables the analysability of SaveCCM models with model-checking tools such as Uppaal or Kronos [17].

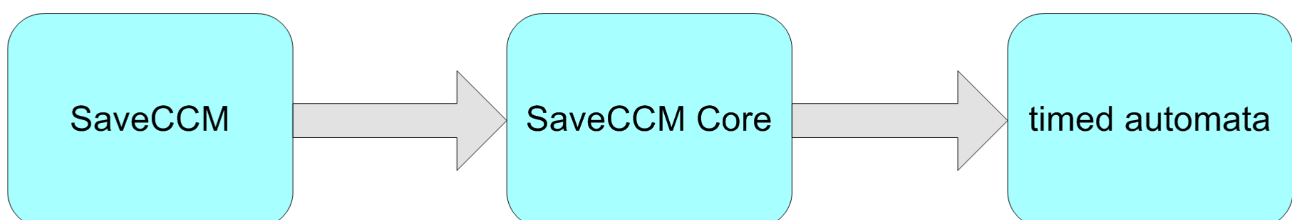


Figure 4.1: Formal definition of SaveCCM semantics

Graphical syntax of SaveCCM is based on a modified subset of UML2 component diagrams (Figure 4.2). Textual syntax of SaveCCM is XML based (it is described in Chapter SaveCCM XML syntax).

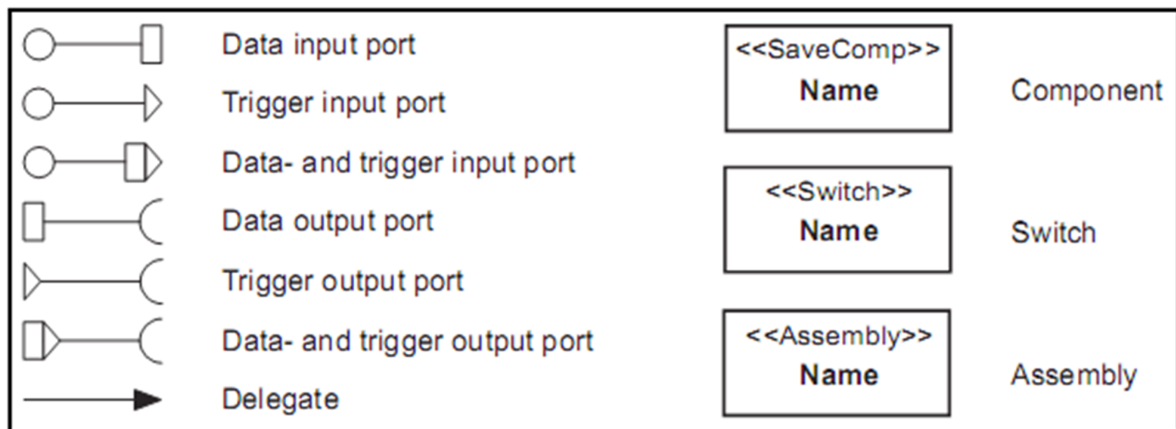


Figure 4.2: SaveCCM graphical syntax

The main architectural elements of SaveCCM are components, switches and assemblies. The interface of an architectural element is defined by a set of ports. Systems are built from architectural elements by connecting ports.

4.1 Ports

In SaveCCM there is a distinction between input and output ports, and between trigger and data ports. Trigger ports capture control flow and data ports capture the transfer of data. A port can have both triggering and data functionality. This type is known as the combined port. A combined port is an association between a data and a trigger port. It exists to reduce the wiring between SaveCCM elements, as it can be a nightmare on a big design.

Every data and combined port is typed and can contain an initial value. Only data and combined ports of matching types can be connected. Data and combined ports have overwrite semantics.

There are three types of connections – immediate connections, complex connections and delegations. Immediate connections represent lossless transfer of data or trigger signals from one port to another. Complex connections represent transfer of data or trigger signals over channels with possible delay or information loss. The characteristics of a particular complex connection are explicitly modelled by a timed automaton. Delegations are semantically identical to immediate connections, but connect two input ports or two output ports between the internals and externals of assemblies, while immediate connections connect an input port to an output port. Delegations also connect internal and external data ports in composite components.

A trigger output port can only be connected to trigger input ports, a data output port can only be connected to data input ports of the matching type, and a combined output port

can be connected to trigger, data or combined input ports of the matching type[9]. One output port can be connected to several input ports, while a single input port can only be connected to one output port [16].

An external port is not connected to any other port, but has an extra label mapping it to some external entity, for instance to a register or a database query result. A set port is used in a switch (it is described in Chapter Switches).

4.2 Components

Components are the main architectural element of SaveCCM. They represent basic units of encapsulated behaviour. The functionality of a component is typically defined by its entry function, which is written in C programming language. These are plain components. However, the functionality can also be defined by an internal composition. These are composite components. There are two additional types of components – a clock component and a delay component.

In addition to input and output ports (functional interface), a component's interface contains a series of quality attributes (non-functional interface). Each quality attribute is associated with a value and possibly a confidence measure. The quality attributes can include, for instance worst case execution time, reliability estimates etc., and are used for analysis of the developed system or for prediction of its characteristics.

A component is not allowed to have any dependencies on other components or other external software, except the dependencies visible through its ports.

A component is initially idle and remains in that state until all its input trigger ports are activated. At that point it switches to active state, i.e. it has been triggered. This initiates the “read” phase, in which all data input port values are stored internally, to ensure consistent computation. Next is the “execute” phase, in which the computations are performed. After execution comes the “write” phase, in which data is written to the component's output ports. Finally, the triggers are reset – the input triggers are deactivated and the output triggers activated. This returns the component to idle state. This strict “read-execute-write” semantics ensures that once a component is triggered, the execution is independent of any concurrent activity [16].

4.2.1 Clocks and delays

Clocks and delays (Figure 4.3) are special types of components which are in charge of manipulating trigger timing.

A clock component is a trigger generator. It has one output trigger port and no other ports. its parameters are T, for period, and J, for jitter. A new period starts every T time units, and a clock component generates a trigger within J time units after the start of each period.

A delay component detains a trigger signal. It has one input and one output trigger port and no other ports. Its parameters are D, for delay, and P, for precision. Upon receiving a

9 All this applies to immediate connections, not delegations.

trigger, the delay component waits between D and D+P time units before generating a new trigger [16].

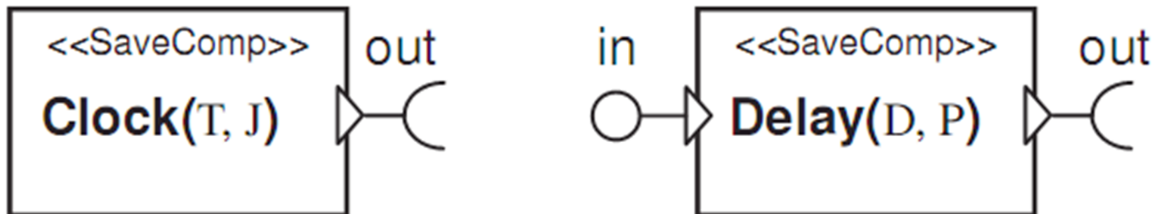


Figure 4.3: SaveCCM clock and delay components

4.2.2 Composite components

A composite component is a special type of a component, where the behaviour is specified by an internal composition. The internals and externals of composite components are connected by delegations – external input data ports are connected to internal input data ports and internal output data ports are connected to external output data ports. Triggering is not transferred this way. Instead, all trigger ports in the internal composition become active when the composite component becomes active. In the “read” phase, data is transferred to the internals and internal components are activated. The “execute” phase performs computations of internal components, until no internal component is active. In the “write” phase data is transferred to the externals, the triggers of the composite component are reset and it becomes idle [16].

4.3 Switches

Switches enable changing the structure of connections between components. They provide means for conditional transfer of data and triggering between components.

A switch has a number of mappings between its input and output ports. Each mapping has a logical expression over the values on the input ports, which is used to determine if that mapping is active or not. The ports that are present in logical expressions are called set ports. Currently the only logical expression supported is equality. Data arriving to a set port is evaluated against the value set in the set port, to determine whether a switch connection should be activated or not. This is shown in Figure 4.4. If data that arrives at the set port is of value “1”, then the connection between “in” and “out” is activated.

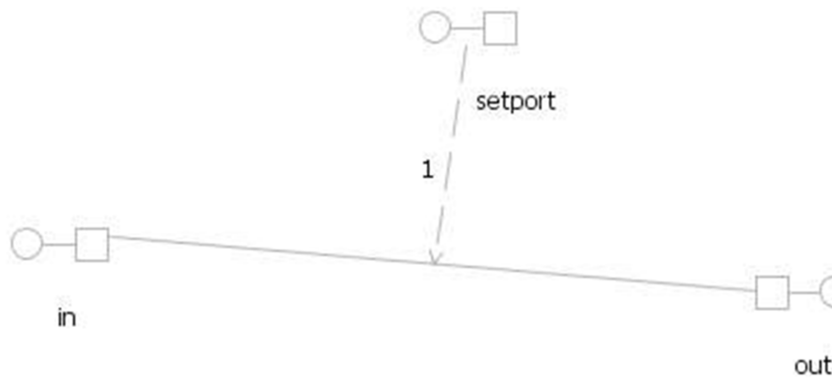


Figure 4.4: A switch condition

Switches are not triggered, they respond directly to the arrival of data or a trigger signal at an input port and immediately relay it according to the current connection patterns. They perform no computation other than the evaluation of logical expressions [16].

4.4 Assemblies

Assemblies are encapsulated subsystems, similar to composite components. Internal elements and connections of an assembly are hidden from the rest of the system and can be accessed only through the assembly's ports. Like switches, assemblies are not triggered, signals are directly relayed between externals and internals of an assembly through delegations.

Due to the strict execution semantics of SaveCCM components, an assembly does not satisfy the requirements of a component. It can break the “read-execute-write” semantics. Thus, it should only be viewed as a mechanism for naming a collection of components and hiding internal structure, and not like a mechanism for component composition [16].

4.5 SaveCCM XML syntax

In this chapter the SaveCCM XML type is described. Its formal definition is given in the SaveCCM DTD and SaveCCM schema.

The `APPLICATION` element is the root of the SaveCCM XML type. It contains an `IODEF`, a `TYPEDEFS`, a `COMPONENTLIST` and a `CONNECTIONLIST` element (Figure 4.5).

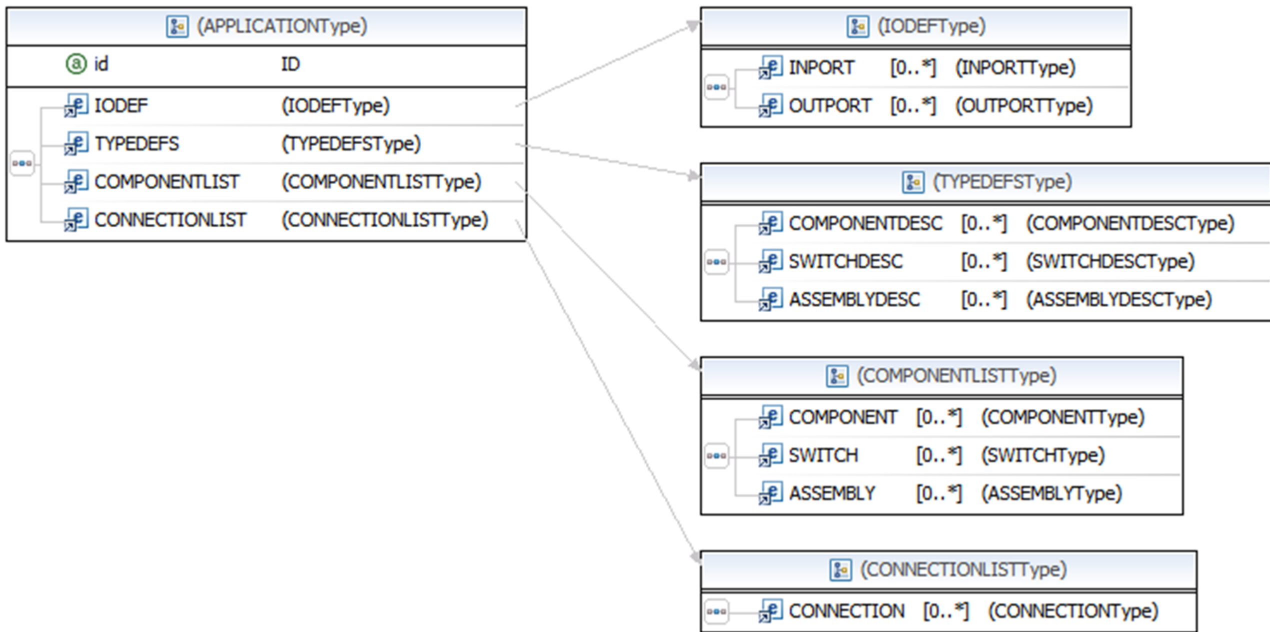


Figure 4.5: Application definition

The `IODEF` element defines the external ports of an application. The `TYPEDEFS` element gives definitions of component, switch and assembly types (Figure 4.6). The `COMPONENTLIST` and `CONNECTIONLIST` elements define a composition – either for an application, for a composite component or for an assembly [16].

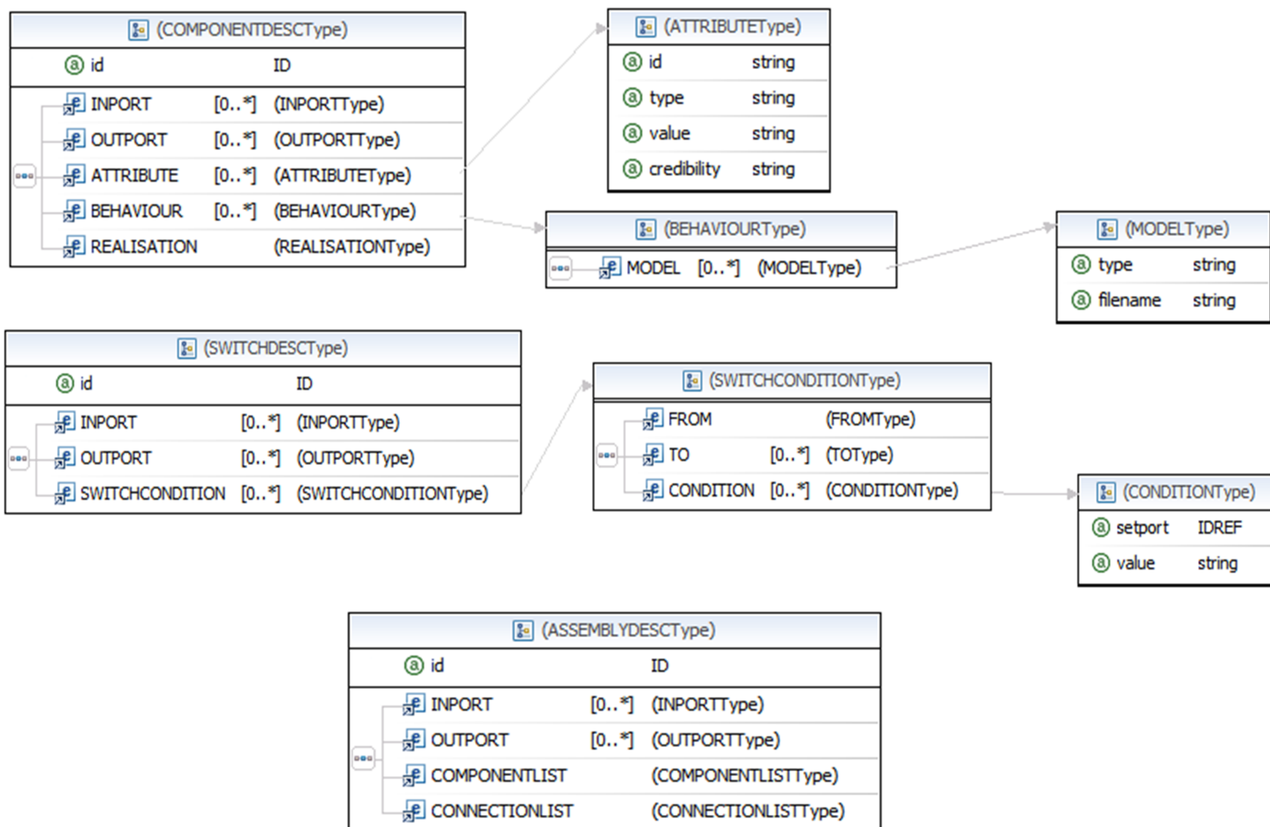


Figure 4.6: Component, switch and assembly type definitions

A component type (COMPONENTDESC element) is defined as a set of input and output ports, attributes and an implementation. The REALISATION element (Figure 4.7) describes the component's implementation as either:

- a function written in C programming language (ENTRYFUNC),
- a clock component (CLOCK),
- a delay component (DELAY) or
- a composite component ((COMPONENTLIST, CONNECTIONLIST)).

The BINDPORT element is used to map a port used within a component with an argument used in the C implementation function. A BEHAVIOUR element is a collection of MODEL elements. Models describe additional implementations of a component's behaviour. A model can be an external file or embedded as text within the element. An attribute (ATTRIBUTE element) is used to describe a non-functional property of a component [16].

A switch type (SWITCHDESC element) is defined as a set of input and output ports and switch conditions. The SWITCHCONDITION element defines under what conditions the switch ports are internally connected.

An assembly type (ASSEBBLYDESC element) describes an assembly as a set of input and output ports, and a composition of components and connections [16]

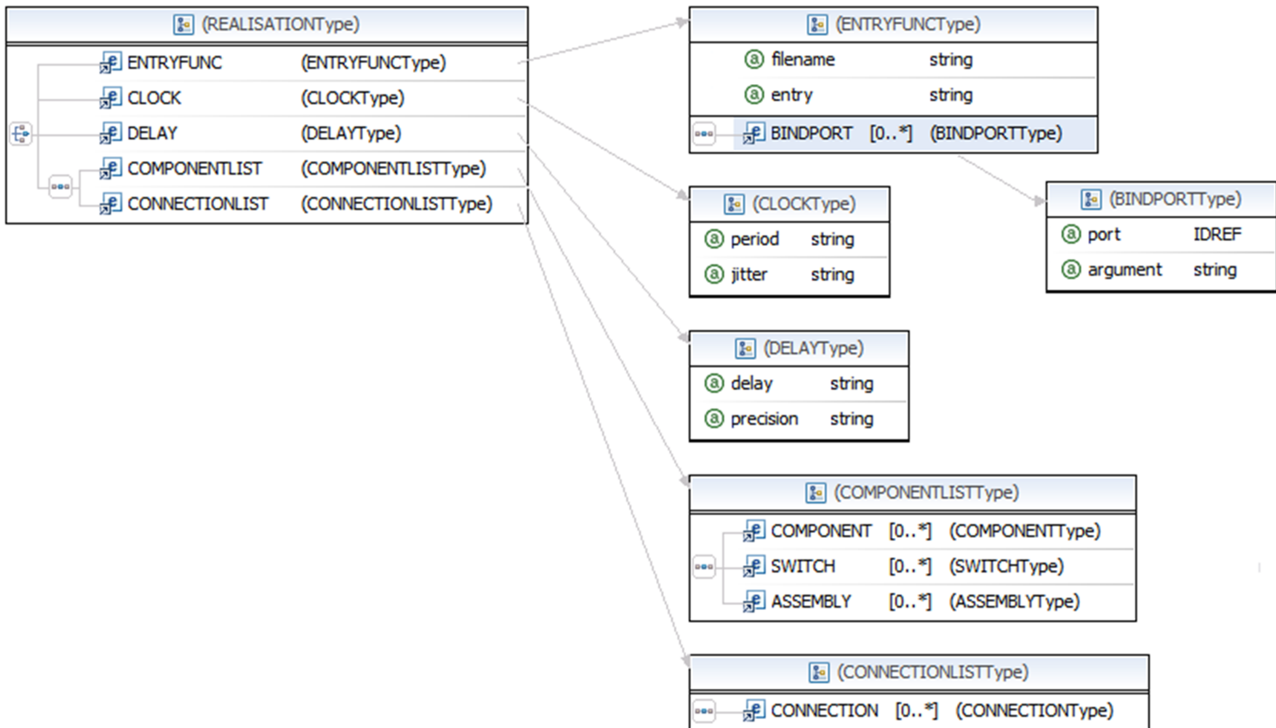


Figure 4.7: Component realisation definition

A `COMPONENT` element instantiates a component type, a `SWITCH` element instantiates a switch type and an `ASSEMBLY` element instantiates an assembly type. An instanced element is a reference to its corresponding type definition (Figure 4.8).

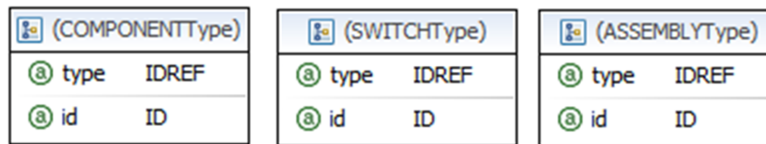


Figure 4.8: Component, switch and assembly instantiation

The `INPORT` and `OUTPORT` elements (Figure 4.9) define input and output ports, respectively. The `mode` attribute determines if a port is a data, trigger or a combined port. The `type` and `value` attributes are used to define data type and initial value of data and combined ports. The `external` attribute holds the label defining a connection to an external entity. The `setport` attribute determines if an input port is a set port, i.e. used in a switch condition [16].

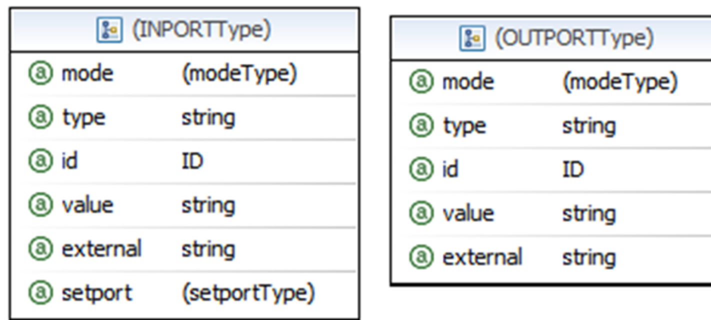


Figure 4.9: Port definitions

A `CONNECTION` element (Figure 4.10) defines a connection. It has `FROM` and `TO` elements, which are references to ports. A connection can have a `BEHAVIOUR` element that defines it as a complex connection. No behaviour means that the connection is immediate [16].

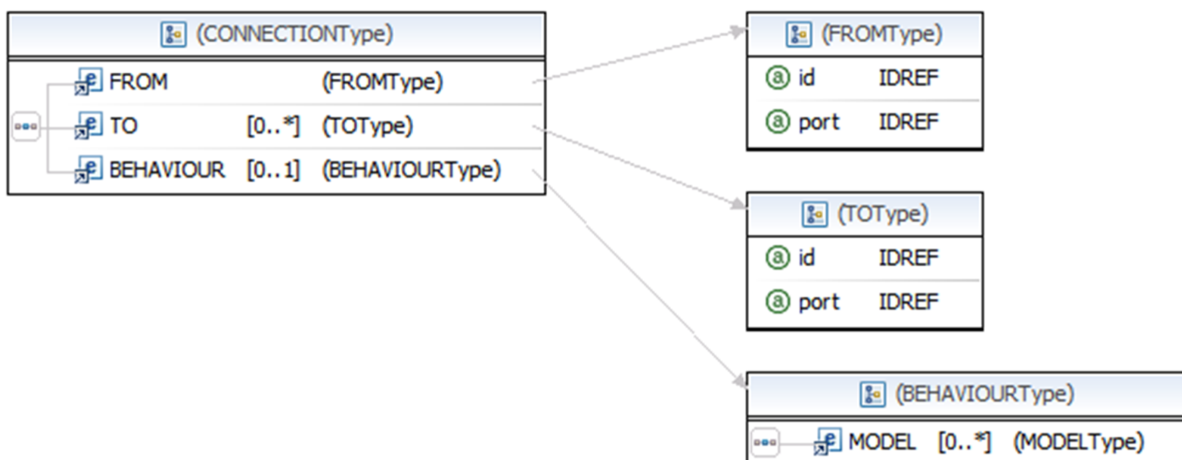


Figure 4.10: Connection definition

4.6 SAVE-IDE

SAVE-IDE is a development environment for SaveCCM. It comes in the form of a plugin for Eclipse IDE. It is intended to provide full support from designing to implementing a SaveCCM system. However, at the current time, only the design part has been finished.

The two main parts of SAVE-IDE that enable design are:

- SAVE-IDE Architectural Editor (Figure 4.11) and
- SAVE-IDE System Description Generator.

The former enables graphic modelling of a system by visually manipulating SaveCCM architectural elements. The later uses the graphical description of a system and generates its textual description. This is done by right clicking on the editor view in a blank space and

choosing `Generation` -> `System Description (.save)`. This creates a file in the `SAVE` directory. This file has the extension `save` and contains the XML representation of the designed system. It will be referred to as "the `.save` file" in this thesis. The `.save` file follows the syntax described in the SaveCCM DTD and SaveCCM schema.

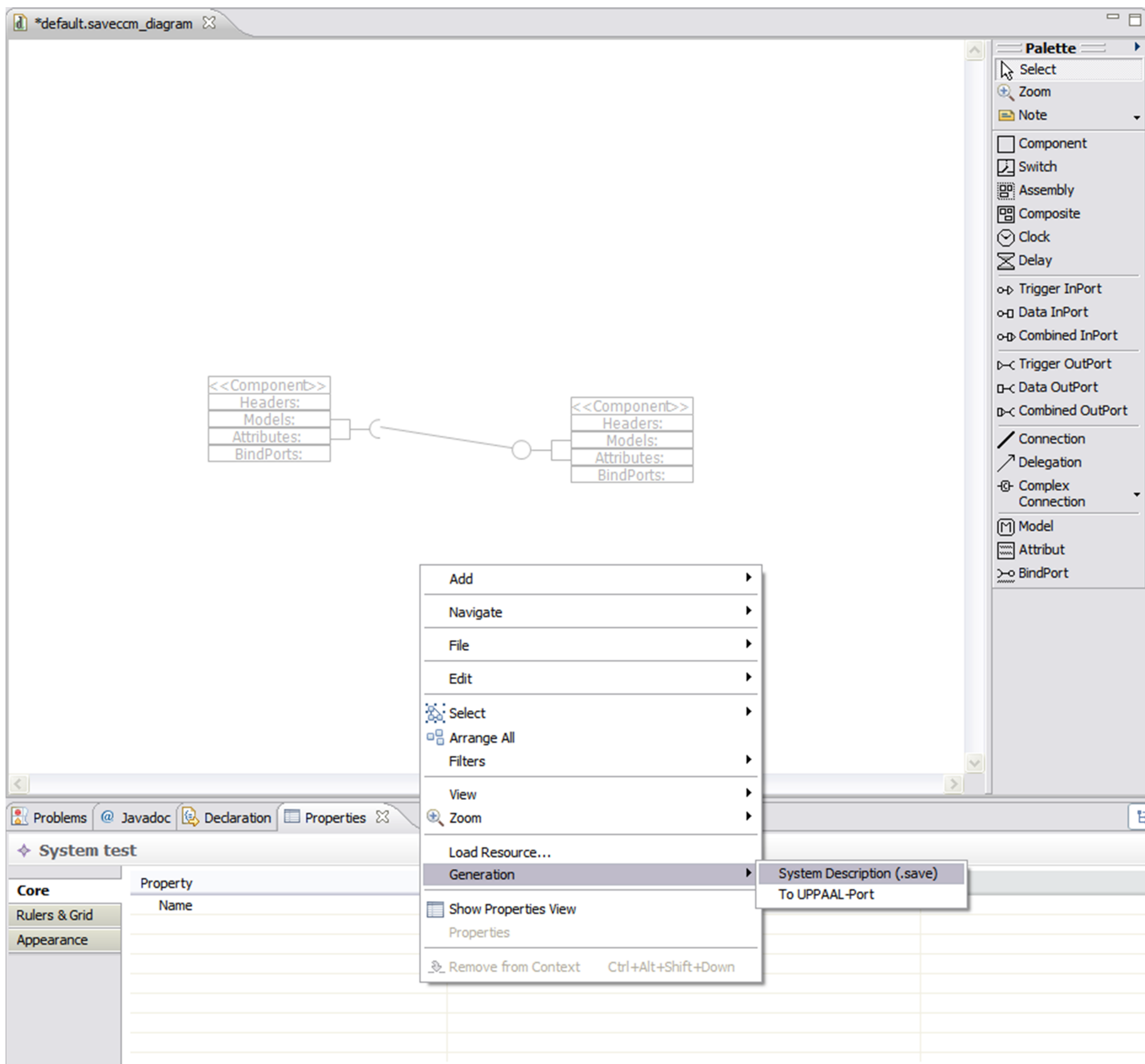


Figure 4.11: SAVE-IDE Architectural Editor

5 Transformation from SaveCCM to JavaBeans

SaveCCM is a domain specific, while JavaBeans is a general purpose component model. SaveCCM is intended for use in embedded vehicular systems. JavaBeans is used mostly for desktop and web applications.

Different component models provide different levels of support in various stages of a system development cycle. SaveCCM is used in the design phase[10], while JavaBeans provides implementation. Therefore, although their domains are quite far apart, a transformation between the two models is worth exploring. The idea is to model a system in SAVE-IDE and automatically get its implementation by applying the transformation from SaveCCM to JavaBeans (Figure 5.1).

Figure 5.1: Transformation from SaveCCM to JavaBeans

The realization consists from two crucial parts:

1. development of the SaveCCM Java model and
2. implementation of the transformation.

5.1 Development of the SaveCCM Java model

This chapter discusses the SaveCCM Java model. It is an object oriented model of SaveCCM, i.e. it gives a Java representation of SaveCCM elements.

Some classes are common to all systems, so they exist prior to the transformation, while others are generated from the system definition during the transformation. Together they form the SaveCCM Java model. The preexisting classes are: `Clock`, `Component`, `DataEvent`, `DataEventListener`, `DataInPort`, `DataOutPort`, `DataPort`, `Delay`, `Executor`, `TriggerEvent`, `TriggerEventListener`, `TriggerInPort`, `TriggerOutPort` and `TriggerPort`. An UML diagram of a part of the architecture (only preexisting classes) is shown in Figure 5.2.

10 Although SaveCCM is intended to provide support in the whole development cycle, from design to implementation, in its current form it fully supports only the design phase.

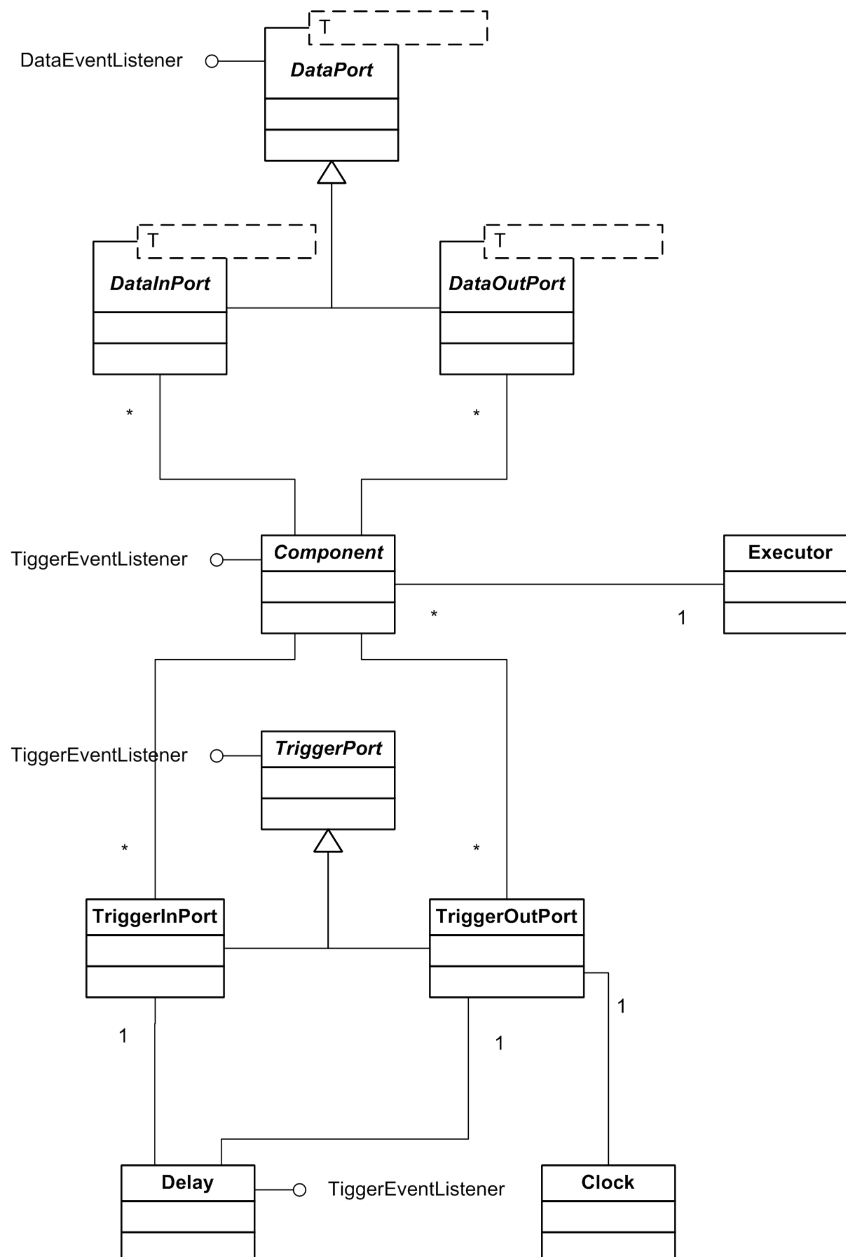


Figure 5.2: UML diagram of a partial SaveCCM Java model

5.1.1 Executor

In a real system made from embedded computers each component can be a single embedded device. So all components can execute simultaneously. On a PC that is not possible, thus there is a need for a special mechanism that simulates simultaneous execution of components. This is archived through the `Executor` class. Each generated system has one object of type `Executor`, which is in charge of executing components. This executor object holds a queue of triggered components and executes one by one, in the same order as they got triggered (Snippet 5.1). The executor object operates in its own thread.

Snippet 5.1: Executing the components

```
Component component = executionQueue.poll();
if (!stop && component != null) {
    component.read();
    component.execute();
    component.write();
    component.resetTriggers();
    component.setIdle(true);
}
```

5.1.2 Immediate connections and delegations

Immediate connections and delegations have no class representations, instead they are modelled using the Java Event Model. Connecting one port (let it be called "destination port") to another one (called "source port") is done by registering the destination port as the listener of the source port. Thus, a source port has to have methods for registering and unregistering listeners. A destination port has to implement a listener interface.

An event type is modelled by an event class and an event listener interface. In the SaveCCM Java model there are two types of events, one for data port (data events) and one for trigger port connections (trigger events). Data connections use the `DataEvent` class and corresponding `DataEventListener` interface. Trigger connections use the `TriggerEvent` class and `TriggerEventListener` interface.

5.1.3 Ports

Ports are modelled with two separate hierarchies – one for data ports and one for trigger ports. A combined port in SaveCCM is a plain association of a data and a trigger port. The SaveCCM Java model is much simpler without introducing a third type of port, so one combined port from SaveCCM becomes one data port and one trigger port in Java.

Data ports are modelled using Java Generics, which allows a single hierarchy between ports of different types. The full hierarchy includes the `DataPort<T>` generic abstract class as the root. It is extended by generic abstract classes `DataInPort<T>` and `DataOutPort<T>`. A new port type extends one of these two classes. For instance, a data input port holding a value of string type would be modelled with the `StringDataInPort` class which extends `DataInPort<String>`. So, in addition to the three preexisting abstract classes, other data port classes are generated during the transformation.

Data ports have methods for registering and unregistering data event listeners. Data input ports implement the data event listener interface, as they can listen to data output ports in connections and data input ports in delegations. Data output ports implement that interface as well, since they can listen to data output ports in delegations. On picking up a data event, a listener port updates its value with the value from the port that generated the event.

The trigger port hierarchy has `TriggerPort` abstract class as the root, which is

extended by `TriggerInPort` and `TriggerOutPort` classes. No other trigger port classes are generated during the transformation, since a trigger port does not have a type, unlike a data port.

Trigger ports have methods for registering and unregistering trigger event listeners. Trigger ports and components implement the trigger event listener interfaces. A trigger input port can listen to trigger output ports in connections and trigger input ports in delegations. A trigger output port can listen to trigger output ports in delegations. A component always listens to all of its input trigger ports so it can recognize when it becomes triggered.

The scenario shown in Figure 5.3 is used to explain how trigger events are normally promoted. It consists of component “A” and its output trigger port “outA”, and component “B” and its input trigger port “inB”. “inB” listens to “outA” and “B” listens to “inB”. When “outA” generates a trigger event, “inB” picks it up, updates its state, and promotes it to “B”.

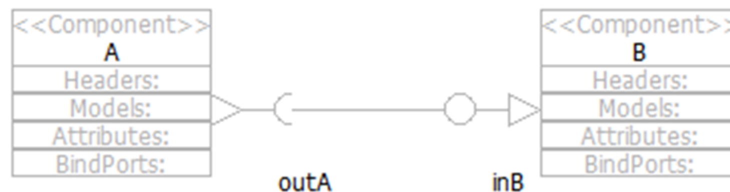


Figure 5.3: Promoting trigger events

Output ports have the priority over input ports, meaning that when a value is written to an output port it is immediately promoted to all input ports connected to it, overwriting the former values of the input ports.

External ports are merely marked with a comment `// TODO external` in the generated component classes. That way the user can decide how to handle them. This is due to the following reasons. The `IODEF` element which defines external ports is not written correctly in the `.save` file, it is always generated with no content. Also, a more precise description of using external ports is needed in the SaveCCM specification.

5.1.4 Components

Components are modelled with the `Component` abstract class (Figure 5.4), which is the root of the simple component hierarchy. For each component type defined by the `COMPONENTDESC` element in the `.save` file, one component class is generated during the transformation. The generated classes extend the abstract one.

The `Component` class holds a reference to the system's executor object (1)[11]. A component can be in two states – idle or active. If it is idle, it is not triggered and vice versa. Once it gets triggered, i.e. active, all changes to the input triggers are disregarded until after execution. After execution the triggers are reset - input triggers to inactive and

11 The numbers in brackets relate to Figure 5.4.

output triggers to active state, and the component is returned to idle state. The state is regulated by the `idle` flag and corresponding getter and setter (2).

A separate `Vector` for each port type exists - input data port, output data port, input trigger port and output trigger port `Vector` (3). There are methods for adding new and fetching existing ports (4), a method for resetting the triggers (5), a method that registers the component as a listener of all of its input ports (6), and a method for checking if the component is triggered (7). When one of its input triggers gets active, the component receives a trigger event. Then it checks the state of all other triggers – if all of them are active, the component is triggered, that is set to active state. Then it adds itself to the executor object's queue for execution.

Component classes that are generated during the transformation have to implement three abstract methods (8) from the abstract parent class:

`read` – stores the values from data ports internally (to private variables),

`execute` – executes the component, i.e. implements the component's behaviour,

`write` – writes the internally stored values to output data ports.

In the generated classes one private variable exists for every data port, which ensures consistent computation – if other values get written to data input ports during execution, they have no effect on the outcome of the computation.

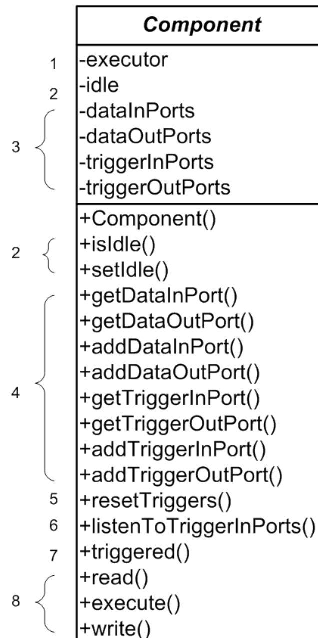


Figure 5.4: UML diagram of the `Component` class

5.1.5 Clocks and delays

Clocks are represented by the `Clock` class. Each clock operates in its own thread. A delay (modelled by the `Delay` class) operates in the same thread as the clock connected to it.

In SAVE-IDE it is legal for clocks and delays to have any number and any type of ports. This is in contradiction with SaveCCM specification, so the `Clock` and `Delay` classes allow only ports stated in the specification - one output trigger port for a clock, one input and one output trigger port for a delay.

Although clocks and delays in SaveCCM are special types of components, in the SaveCCM Java model their classes are in no relation to the component hierarchy. However, this has no effect on the transformed systems.

5.1.6 Unsupported elements

Switches, assemblies, composite components and complex connections are currently not supported.

5.2 Implementation of the transformation

The transformation is realised using the Java programming language. It is implemented by the `hr.fer.rasip.save_to_java.Transformer` class. Unfortunately, the code that preforms the transformation can be quite difficult to read. The implementation is broken down into five methods, each handling one logical piece of the transformation (Snippet 5.2). The methods are discussed later in this chapter. First a description of parsing the input file is given.

Snippet 5.2: Methods implementing the transformation

```
this.application = this.unmarshall();
this.copyClasses();
this.generateDataPortClasses();
this.generateComponentClasses();
this.generateSystemDescriptionClass();
```

5.2.1 Parsing the `.save` file

As mentioned before, a SaveCCM system is defined in the `.save` file, which is in XML format. This file is the input of the transformation.

Parsing the `.save` file is done using Java Architecture for XML Binding (JAXB). JAXB (a part of the Java SE 6) enables mapping between XML and Java. It provides two main features:

1. marshalling Java objects into XML, i.e. transforming a Java object hierarchy into XML format and

2. unmarshalling XML into Java objects, i.e. transforming an XML document into a hierarchy of Java objects.

JAXB has two main parts:

- the binding compiler and
- the binding runtime framework.

The binding compiler (`xjc`) (Figure 5.5) transforms an XML schema into a collection of Java classes that matches the structure described in the schema. These classes are annotated with special JAXB annotations, which provide the runtime framework with the mappings it needs to process the corresponding XML documents.

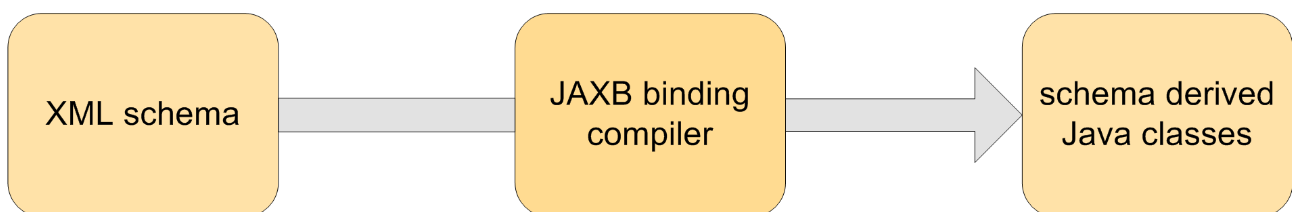


Figure 5.5: JAXB binding compiler

The binding runtime framework (Figure 5.6) provides a mechanism for marshalling and unmarshalling XML documents. Marshalling is equivalent to XML writing and unmarshalling to XML reading. The binding framework also enables validation of an XML document against its corresponding schema.

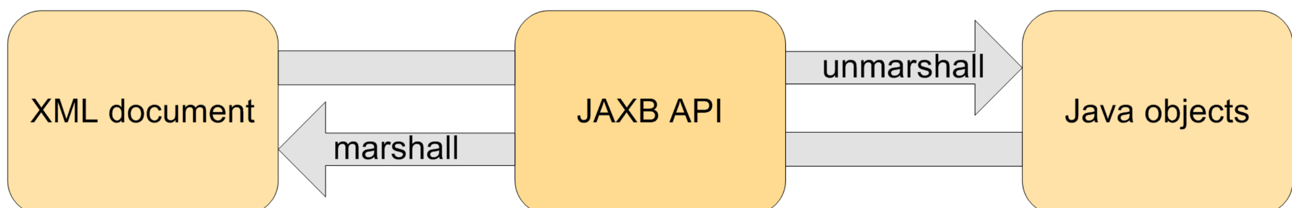


Figure 5.6: JAXB binding runtime framework

Combined, these two parts produce a technology that lets Java developers easily manipulate XML data in the form of Java objects, without having to know the details of XML processing or XML Schema[12] [18]. This simple and intuitive use is the main reason for using JAXB instead of a DOM or SAX parser for this project.

5.2.1.1 SaveCCM DTD and SaveCCM schema

DTD and XML Schema are languages that provide specifications of a type of XML document. The specification is expressed in terms of constraints on the structure and content of documents of that type.

12 Schema with an upper-case 'S' refers to the language, and schema with a lower-case 's' to the document written in Schema language, a document that describes an XML type.

DTD is native to XML specification, however it has relatively limited capability. XML Schema is richer, it can describe structural relationships and data types that can not be expressed (or can not easily be expressed) in DTD [19]. XML Schema is required by the current version of JAXB (2.1), as DTD is only experimentally supported.

The SaveCCM DTD[13] is provided in the SaveCCM specification, unlike the SaveCCM schema. The SaveCCM schema was automatically generated from the SaveCCM DTD. Various tools allow this. Here it was done using Microsoft Visual Studio.

Prior to generating the schema, the SaveCCM DTD was slightly modified, since it had contained an error. The `BEHAVIOUR` element can occur zero or more times, not one time and one time only, as was stated. So the above line was replaced with the one below:

```
<!ELEMENT COMPONENTDESC (INPORT*, OUTPORT*, ATTRIBUTE*, BEHAVIOUR,
REALISATION)>

<!ELEMENT COMPONENTDESC (INPORT*, OUTPORT*, ATTRIBUTE*, BEHAVIOUR*,
REALISATION)>
```

Once the SaveCCM schema is obtained, it can be binded. The JAXB binding compiler is located in the `bin` directory of the JDK installation. The binding is done with the following command:

```
xjc SaveCCM.xsd
```

This results in generating a number of Java classes in the `se.mdh.save.saveccm` package. They are used for parsing the `.save` file.

5.2.1.2 Changes to the `savexmlgenerator.mt` file

The `.save` file generated by SAVE-IDE in both currently available versions (0.5 alpha and the snapshot release from 2008-02-01) required modifications to be usable with JAXB. The modifications were done by editing the file in charge of generating the `.save` file – `savexmlgenerator.mt` in the `se.mdh.mrtc.saveccm.xmlgenerator.jar` archive.

The changes are listed in Table 5.1.

13 Analogue to Schema and schema, DTD is a language and SaveCCM DTD is a document written in DTD language, a specification of the SaveCCM XML type.

Table 5.1: Changes to the `savexmlgenerator.mt` file

original	replaced with
<?xml version="1.0"?>	<?xml version="1.0" encoding="utf-8"?>
<!-- DOCTYPE APPLICATION SYSTEM "sys/savecomp.dtd” -->	removed
<APPLICATION id="<%name%>">	<APPLICATION id="<%name%>" xmlns="http://save.mdh.se/SaveCCM" xmlns:xsi="http://www.w3.org/2001/XMLS chema-instance" xsi:schemaLocation="http://save.mdh.se /SaveCCM resources/SaveCCM.xsd">
<%if (name.length()>0){%> <%name%>#<%id%> <%}else{%> #<%id%> <%}%>	<%if (name.length()>0){%> <%name%>_<%id%> <%}else{%> _<%id%> <%}%>
<%if (compose.length()>0){%> <%componentlist%> <%}else{%> <%}%>	<%if (compose.length()>0){%> <%componentlist%> <%}else{%> <COMPONENTLIST> </COMPONENTLIST> <%}%>
<%if (connects.length()>0|| define.length()>0){%> <%connectionlist%> <%}else{%> <%}%>	<%if (connects.length()>0|| define.length()>0){%> <%connectionlist%> <%}else{%> <CONNECTIONLIST> </CONNECTIONLIST> <%}%>

The change stated in row 1 adds the encoding declaration to the `.save` file. The change in row 2 removes the commented SaveCCM DTD reference, since the SaveCCM DTD is replaced by the SaveCCM schema. Those two changes are not crucial[14].

The change in row 3 adds a namespace declaration and a reference to the SaveCCM schema. The row 4 change replaces '#', in the `id` attribute of various elements, with '_', because '#' is an illegal character in the ID and IDREF built-in data types of XML Schema. The row 4 and 5 changes add `COMPONENTLIST` and `CONNECTIONLIST` elements to the `.save` file when the system has no components or connections, respectively[15]. These changes are crucial and allow the `.save` file to be parsed using JAXB and to be a valid XML document[16]. This means that the transformation is not compatible with the currently available versions of SAVE-IDE – the original `se.mdh.mrtc.saveccm.xmlgenerator.jar` file has to be replaced with the edited one.

14 The row 2 change would have been crucial if the SaveCCM DTD reference was not commented.
15 The purpose of that kind of system is questionable, but it is legal in SaveIDE.
16 An XML document does not have to have a schema, but if it does, it must conform to that schema to be a valid XML document.

5.2.2 The `unmarshall` method

Before being parsed, the `.save` file needs to be unmarshalled, which is done by the code in Snippet 5.3. After unmarshalling, the root element can be fetched. All other elements and attributes are reachable from the object representing the root element. This method returns the root object. Validation of the `.save` file against the SaveCCM schema is also done in this method (Snippet 5.4).

Snippet 5.3: Unmarshalling the `.save` file

```
JAXBContext jc = JAXBContext.newInstance("se.mdh.save.saveccm");
Unmarshaller u = jc.createUnmarshaller();
```

Snippet 5.4: Validation of the `.save` file against the SaveCCM schema

```
SchemaFactory factory =
SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
Schema schema = factory.newSchema(new File("resources/SaveCCM.xsd"));
u.setSchema(schema);
```

5.2.3 The `copyClasses` method

As stated in the Chapter Development of the SaveCCM Java model, some classes are the same for every system and are not generated from the information contained in the `.save` file. They are merely copied from the `resources` directory to the desired package using the `copyClasses` and the `copyFile` methods.

5.2.4 The `generateDataPortClasses` method

`COMPONENTDESC` elements in the `.save` file are scanned for `INPORT` and `OUTPORT` child elements whose `mode` attribute has the value “`data`” or “`combined`”. The value of the `type` attribute is then compared to supported types using the `getPortType` method. If the type is successfully obtained, a new data port class is generated.

So far byte, boolean, character, integer, float, double and string data types are supported. New data types can be added by modifying the `getPortType` method. If an unsupported data type is encountered, an `UnknownDataTypeException` is thrown and the system cannot be transformed to JavaBeans. A possible recovery strategy can be editing the `.save` file (replacing the unsupported type) and running the transformation again.

5.2.5 The `generateComponentClasses` method

For each `COMPONENTDESC` element, the `REALISATION` child element is parsed. If the component is a plain component[17] then a new component class that extends the

17 As opposed to clock, delay or composite component.

Component class is generated. The class is named by the value of the id attribute of the COMPONENTDESC element. In the constructor, the ports are added.

If the component is a clock or a delay nothing is generated, since those classes already exist. Composite components are not supported.

The ATTRIBUTE and BEHAVIOUR child elements of the COMPONENTDESC element are ignored. ATTRIBUTE specifies non-functional properties of components which are not of great importance for implementing the system in Java. BEHAVIOUR specifies additional behaviours of the component. It is unclear from the SaveCCM specification how it is decided which additional behaviour is to be executed at a certain time, so the additional behaviours are ignored. The default and only behaviour of the component is defined by its execute method, and this is sufficient for implementing the system in Java.

The BINDPORT child element of the ENTRYFUNC element corresponds to the component's implementation in C. Since the C code is not used in any way, the BINDPORT is ignored.

5.2.6 The generateSystemDescriptionClass method

This method generates an executable class (contains a public static void main(String[] args) method) which describes the SaveCCM system defined in the .save file. The class is named by the value of the id attribute of the APPLICATION element.

First all of the components present in the system (that is, defined by the COMPONENTLIST element) are instantiated. An executor object is also instantiated. Then all ports are connected according to the CONNECTIONLIST element. After connecting the ports, their initial values are set, as stated in the COMPONENTDESC elements. Since output ports have the priority over input ports (the values from the output ports must be retained), initial values are set in the order – first input ports, then output ports. Finally the system is initiated by starting the executor object and clocks, if any present.

5.3 Limitations of the transformation

There are restrictions to what type of SaveCCM system can be transformed. Some of them come from not supporting all SaveCCM elements and some from errors in SAVE-IDE. The restrictions are denominated in this chapter.

Composite components, switches, assemblies and complex connections must not be used. This is obvious since they are not supported.

A clock must have one trigger output port and no other ports. A delay must have one trigger input port, one trigger output port and no other ports. This is explained in Chapter Clocks and delays.

The clock and delay attributes (period and jitter, delay and precision) must be integers. This will be explained in the Chapter SaveCCM and SAVE-IDE errors.

5.4 Performing the transformation

The transformation is performed in two steps – the automatic and the manual step. The description in Chapter Implementation of the transformation relates to the automatic step. It is done using the SaveToJava tool (Figure 5.7), which is implemented by the `hr.fer.rasip.save_to_java.TransformerGUI` class. The tool comes in the form of an executable JAR file. It can be executed by a double mouse click or from the command line with the following command (just be sure to have the `resources` directory in the same directory as the the JAR):

```
java -jar SaveToJava.jar
```

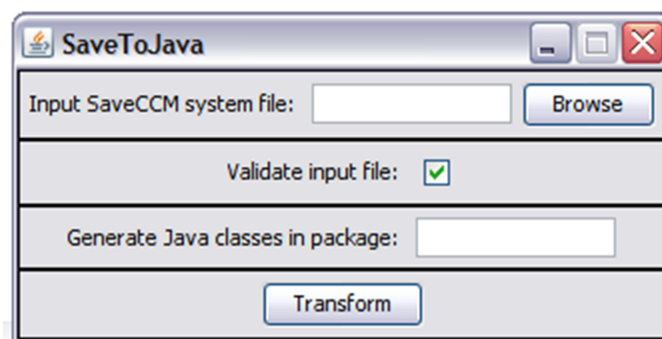


Figure 5.7: The Save2Java tool

Prior to transforming, the user must provide:

- the path to the `.save` file describing the SaveCCM system to be transformed,
- the desire whether to validate the `.save` file against the SaveCCM schema and
- the output package in which the Java classes will be generated.

The output of the automatic step of the transformation is a directory structure matching the desired output package and containing the generated Java classes.

In the manual step, the user needs to manually edit the generated code. Since the SaveCCM components are by default implemented in C, the generated Java component classes have empty `execute` methods and the user needs to provide them. Also, in the executable system class, the user needs to provide the code for terminating the system, as the system can be executed indefinitely, depending on its structure. The user can also provide code for handling external ports, as explained in the Chapter Ports.

No generated code is locked for editing[18], so the user can edit the generated system in a way he/she desires. This allows, for instance, that only a part of the system is built in SaveCCM and it is finished using Java.

18 Which is not an uncommon situation with tools that automatically generate code.

5.5 Transformation example

This chapter gives an example of performing the transformation from SaveCCM to JavaBeans. The system is show in Figure 5.8. It consists of a clock and four components. The “generator1” and “generator2” components generate a random number from 0 to 9 and write it to their output ports. These numbers are compared by the “comparator” component. After the comparison “comparator” writes the result to its output port. The result is printed in the console by the “print” component. The number generators are triggered by the clock. They trigger the comparator. The comparator triggers the printer.

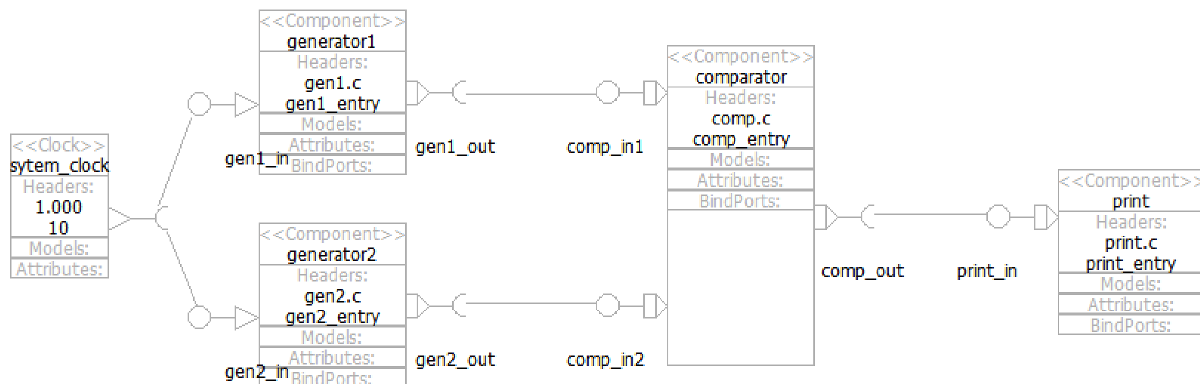


Figure 5.8: A simple SaveCCM system

First the system is designed in SAVE-IDE and the `test.save` file is generated. Then the automatic step of the transformation is performed – the SaveToJava tool is executed, the `test.save` file is selected as input and the `test` package as output of the transformation. By clicking the `Transform` button, the Java classes are generated in the `test` directory, which is then imported into a new Java project in Eclipse. As part of the manual step of the transformation, the `execute` methods of components are edited. They are shown in the following code snippets.

Snippet 5.5: The `execute` method in the `generator1_1` class

```
public void execute() {
    try {
        Thread.sleep(100);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    this.gen1_out_2 = new Integer(new java.util.Random().nextInt(10));
}
```

Snippet 5.6: The `execute` method in the `generator2_4` class

```
public void execute() {
    try {
        Thread.sleep(200);
    } catch (InterruptedException e) {
        e.printStackTrace();
```

```
    }
    this.gen2_out_5 = new Integer(new java.util.Random().nextInt(10));
}
```

Snippet 5.7: The `execute` method of the `comparator_7` class

```
public void execute() {
    try {
        Thread.sleep(100);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    if (this.comp_in1_8 < this.comp_in2_9) {
        this.comp_out_10 = new String(this.comp_in1_8 + " less than " +
this.comp_in2_9);
    } else if (this.comp_in1_8 > this.comp_in2_9) {
        this.comp_out_10 = new String(this.comp_in1_8 + " greater than " +
this.comp_in2_9);
    } else {
        this.comp_out_10 = new String(this.comp_in1_8 + " equal to " +
this.comp_in2_9);
    }
}
```

Snippet 5.8: The `execute` method of the `print_11` class

```
public void execute() {
    try {
        Thread.sleep(300);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println(this.print_in_12);
}
```

The system is started by executing the `test` class and stopped by violently terminating the the program. The output looks like this:

```
6 equal to 6
1 less than 9
8 greater than 3
8 greater than 5
9 greater than 7
5 less than 9
8 greater than 3
```

5.6 Possibilities for improvement

In this chapter a few possibilities for future improvement of the SaveCCM Java model and the SaveToJava tool are given.

The most obvious improvement is adding support for unimplemented SaveCCM elements – switch, assembly, composite component and complex connection.

SAVE-IDE is an Eclipse plugin, it is obvious that the SaveToJava tool should also exist in that form. Some research has been made in that direction, but has not been finished. This is the idea behind the SaveToJava plugin. Right clicking the `.save` file in Eclipse opens a menu. A command would be added to that menu, for instance `Transform to Java`. Clicking it would pop up a graphical user interface similar to the one of the SaveToJava standalone tool. After entering the necessary input (project and package name) and clicking the `Transform` button, a new project would be created. That project would contain the generated system files. This would eliminate the need for importing the generated files in Eclipse, which is necessary when using the SaveToJava standalone tool.

The generated Java system is a console application. A possible improvement is providing some kind of graphical user interface that would allow a direct manipulation (making new component instances, defining connections between ports etc.) and execution (starting and stopping) of the generated system.

Since the user needs to manually define the `execute` method of each component, a nice feature would be having an automatic translation of the behaviour definition in C to a behaviour definition in Java. However this is quite complicated to achieve, as it includes translation between programming languages. It exceeds the scope of this project and requires a more specific definition of the C implementation in the SaveCCM specification.

5.7 SaveCCM and SAVE-IDE errors

Some errors in SaveCCM and SAVE-IDE were found during the work. They are denominated in this chapter. Also some suggestions for improving SAVE-IDE are given.

In the SaveCCM DTD it states that the `BEHAVIOUR` element can occur one time and one time only, but it can occur zero or more times. This is discussed in Chapter SaveCCM DTD and SaveCCM schema.

The `IODEF` element is not written correctly in the `.save` file, it is always generated with no content. This is discussed the Chapter Ports.

Clock and delay can have any number and any type of ports defined in SAVE-IDE. This is discussed in the Chapter Clocks and delays.

The clock and delay attributes get formatted before being written in the `.save` file. The '.' character is inserted after every three digits from the decimal point leftward. The decimal point is written as the ',' character. For instance “1000000.5” is written as “1.000.000,5”. This complicates the parsing of the attributes from strings to numbers. A better solution would be not to insert the '.' character and to use it as the decimal point. There is also an occasional bug with the decimal part of the attributes, for instance “1000000.1” gets written as “1.000.000,125”.

Delegations are not written correctly in the `.save` file. They contain only the `FROM`, and lack the `TO` element.

The `SWITCHDESC` element is not written correctly in the `.save` file. There is always one `SWITCHCONDITION` child element generated, however there should be as many as there are conditions. Also, the `SWITCHCONDITION` element contains `CONNECTION` child elements, while it should contain `CONDITION` elements instead.

SAVE-IDE allows some actions which should be illegal. For instance:

- Ports of different types can be connected. This is done in the following way – first ports of matching types are connected, then one port type is changed. The connection between the ports remains, while it should be broken.
- The same two ports can be connected by several connections. This makes no sense.
- In composite components external and internal triggers can be connected by delegations. However this makes no sense, since triggering in composite components is not transferred like data.

Implementing such constraints in SAVE-IDE could make the mechanism behind the SaveCCM system validating quite heavy, so it is understandable that not all possible constraints are enforced.

Since the `.save` file is the only file required for fully describing a SaveCCM system, a nice feature in SAVE-IDE would be the automatic generation of all necessary project and diagram files from the `.save` file. This would allow importing an existing system in SAVE-IDE. Currently this is possible only if the whole SaveCCM system's project directory exists.

6 Conclusion

Although SaveCCM is intended to provide support throughout the whole system development cycle, currently only the design phase support is complete. Thus, SaveCCM lacks implementation. A SaveCCM Java model was developed, which represents the SaveCCM elements in the object oriented paradigm. A transformation from SaveCCM to JavaBeans was carried out, providing implementation for SaveCCM.

Despite that the transformation is not fully completed and has potential for improvement, the main task is accomplished – it is proven that SaveCCM can be implemented in Java. Since the quality attributes from a SaveCCM system are not retained in a generated Java system, the transformation is intended for demonstration purposes only.

Apart from the transformation, this thesis also contributes through reporting discovered SaveCCM and SaveIDE errors.

This transformation in the opposite direction, from JavaBeans to SaveCCM, was not considered in this thesis. The purpose for the transformation in this direction is questionable. Also it is debatable if it is even possible.

7 Bibliography

[1] IEEE Standard Glossary of Software Engineering Terminology, IEEE

[2] Crnković I., Larsson M.: Building Reliable Component-Based Software Systems, Artech House, 2002

[3] Heineman G. T., Councill W. T.: Component-Based Software Engineering: Putting the Pieces Together, Addison-Wesley Professional, 2001

[4] Gao J.Z., Tsao H.-S., Wu Y.: Testing and Quality Assurance for Component-Based Software, Artech House, 2003

[5] Beugnard A., Jézéquel J.-M., Plouzeau N., Watkins D.: Making Components Contract Aware, http://people.cs.uchicago.edu/~robby/contract-reading-list/contract-aware-components.pdf

[6] Bachmann F., Bass L., Buhman C., Comella-Dorda S., Long F., Robert J., Seacord R., Wallnau K.: Technical Concepts of Component-Based Software Engineering, http://www.sei.cmu.edu/staff/kcw/00tr008.pdf

[7] Crnković I., Larsson S., Chaudron M.: Component-based Development Process and Component Lifecycle, http://www.mrtc.mdh.se/publications/0953.pdf

[8] D'Souza D.F., Wills A.C.: Objects, Components, and Frameworks with UML: The Catalysis Approach, Addison-Wesley, 1998

[9] Crnković I.: Component-based approach for embedded systems, http://research.microsoft.com/~cszypers/events/WCOP2004/18-Crnkovic.pdf

[10] Component-based Design and Integration Platforms, ARTIST, http://www.irisa.fr/triskell/publis/2003/Jezequel03b.pdf

[11] Johnson M.: A walking tour of JavaBeans, 1997, http://www.javaworld.com/javaworld/jw-08-1997/jw-08-beans.html

[12] JavaBeans specification, Sun Microsystems, 1997 http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html

[13] Introduction to Java Beans, http://www.drbob42.com/JBuilder/jb210t.htm

[14] JavaBeans tutorial, Sun Microsystems, http://java.sun.com/docs/books/tutorial/javabeans

[15] Crnković I., Hansson H., Törngren M., Åkerholm M.: SaveCCM – a component model for safety-critical real-time systems, http://www.mrtc.mdh.se/index.php?choice=publications&id=0757

[16] Håkansson J.: The SaveCCM Language Reference Manual, http://www.idt.mdh.se/kurser/cd5490/2007/lectures/SAVE-REF.pdf

[17] Carlson J., Håkansson J., Pettersson P.: SaveCCM An Analysable Component model for real time, http://www.mrtc.mdh.se/index.php?choice=publications&id=1024

[18] Ferguson Smart J.: Java-XML mapping made easy with JAXB 2.0, 2006, http://www.javaworld.com/javaworld/jw-06-2006/jw-0626-jaxb.html

[19] Ort E., Mehta B.: Java Architecture for XML Binding (JAXB), 2003, http://java.sun.com/developer/technicalArticles/WebServices/jaxb

8 List of abbreviations

CBD	Component-based development
CBSE	Component-based software engineering
DTD	Document Type Definition
GUI	Graphical user interface
IDE	Integrated development environment
JAR	Java Archive
Java SE	Java Platform, Standard Edition
JAXB	Java Architecture for XML Binding
JDK	Java SE Development Kit
OOP	Object oriented paradigm (Object oriented programming)
SaveCCM	SaveComp Component Model
XML	Extensible Markup Language