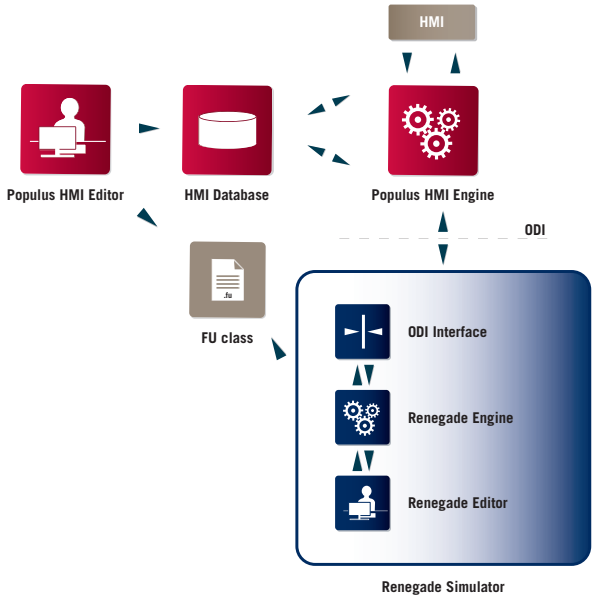
CHALMERS



Editor for application modelling and simulation

Master of Science Thesis

LINDA ERLENHOV
ANNA SÖDLING

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Göteborg, Sweden, July 2009

Editor for application modelling and simulation

Linda Erlenhov
Anna Södling

Examiner: Rogardt Heldal

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

The cover picture shows the system developed in this thesis and its relation to the system developed by Mecel.

Department of Computer Science and Engineering
Göteborg, Sweden July 2009

Abstract

This master thesis is carried out at Mecel AB, a company situated in Gothenburg that develops systems and software for the automotive industry. One problem that they have been facing is that the human-machine interfaces that a user can create with one of their products, Mecel Populus, can not be tested in a convenient way by a non-programmer.

The task of the thesis was to create system for testing purposes where a user can build a graphical statechart that represents an application, for example a radio or a CD player, which can then be simulated. By connecting this system with an existing human-machine interface for the application, the interface could be tested against the simulation.

The primary focus of the thesis work has been an implementation phase which has resulted in the Renegade Simulator, a prototype of an editor with simulation capabilities and an associated interface to be able to communicate with Mecel Populus. The development has been based on a feasibility study of domain-specific languages and of already existing systems. Today, the system does not require a user to have the ability to write any programming code, and with its graphical interface it aims to be easy to use for different types of users.

Sammanfattning

Det här examensarbetet är utfört på Mecel AB, ett Göteborgsbaserat företag som utvecklar system och mjukvara för användning inom fordonsindustrin. Ett problem de ställts inför är att de användargränssnitt som en användare kan skapa med en av deras produkter, Mecel Populus, inte kan testas på ett smidigt sätt av icke-programmerare.

Uppgiften bestod av att skapa ett testsystem där en användare kan bygga ett grafiskt tillståndsdiagram som representerar en applikation, t ex en radio eller en CD-spelare, för att sedan kunna simulera denna. Genom att koppla ihop detta system med ett befintligt användargränssnitt för applikationen skulle man sedan kunna testa detta mot simuleringen.

Den primära fokuset inom examensarbetet har varit en implementeringsfas som resulterat i Renegade Simulator, en prototyp av en editor med simuleringsmöjligheter och ett tillhörande interface för att kunna kommunicera med Mecel Populus. Utvecklingen har baserats på en förstudie av domänspecifika språk och redan befintliga system. I nuläget kräver systemet inga kunskaper i att kunna skriva programmeringskod av en användare och med sitt grafiska gränssnitt syftar det till att vara lättanvänt för olika typer av användare.

Preface and acknowledgements

This master thesis was done by two students, Linda Erlenhov and Anna Södling, at the department of Computer Science and Engineering, Chalmers University of Technology. The work was done in cooperation with Mecel AB in Gothenburg. Both students have taken part in the original development of a basic statechart editor. In addition to that, Anna has been responsible for further developing the editor in order to be user-friendly and to suit the project's needs, and Linda has been responsible for developing the interface between the statechart editor and Mecel's HMI engine.

We would especially like to thank:

Mecel AB - who gave us the opportunity to do this master thesis.

Stefan Gustavsson and Christopher Olofsson - our supervisors at Mecel AB, for their support and for answering questions regarding the project.

Rogardt Heldal, our supervisor at Chalmers University of Technology - for feedback and guidance through the thesis work.

Alex Shatalin, GMF developer - for really making it easier for us to get started and using Eclipse GMF.

Rahul Akolkar, co-editor of the State Chart XML specification - for helping us understand SCXML and the Apache Commons SCXML engine.

Daniel Martinsson and Henrik Edsparr - for their constant support.

Contents

List of Figures

1 Introduction

Mecel is a systems and software development company situated in Gothenburg. The company, which was founded in 1986, has just over 100 employees today, most of them working at the office at Mölndalsvägen. About 10% are out working in the field, as consultants at the customers companies. Mecel's primary customers are leading companies in the automotive industry, such as General Motors and Volvo.
Mecel's tagline, "We make vehicles communicate", pretty much states what their goal is; developing dependable automotive software. In addition to engineering services, they also develop a number of products, mainly within the area of in-vehicle communication, Bluetooth and engineering tools [35].

There are three primary product suites at Mecel. The first is Mecel Betula, which deals with implementation of Bluetooth connectivity in automotive systems [36]. The second is Mecel Picea, which specializes in efficient development of in-car communication technologies [37] And last but not least, there is Mecel Populus, a series of products used to simplify the development of Human-Machine Interfaces [34].

Mecel's Populus suite is a series of tools for designing and developing user interfaces for distributed embedded systems in the automotive industry. The suite consists of an editor to create a Human-Machine Interface, from now on referred to as HMI, as well as specifying the interfaces to the functional units, or FUs, run by the HMI. An FU is for example a GPS navigator, a CD player or similar devices that can be found in a car. The suite also consists of an engine to run the HMI and to communicate with all FUs via the Open Display Interface, or ODI, protocol.

Today, the HMIs created in the Mecel Populus HMI Editor has to be tested against a C or C++ implementation of an FU. This, however, implies that the HMI developer has the programming skills to create such an implementation which is not always the case. Another way to test the HMI is to connect it to an actual FU in its target environment, for example in an embedded system in a car, in order to make sure it is working properly and in the way the client wants it to. If the HMI is faulty or if the client is not satisfied for some other reason, the product needs to go back to the developer, be corrected as to fit the clients needs and then be put back in the right en-

vironment and be tested again. Since this procedure might, in a worst case scenario, be repeated several times, this is an unnecessarily time consuming and expensive way to develop the final HMI.

The problem is that the current Populus suite lacks a simple way to test the HMI outside its target environment. To further complicate matters, there is little or no possibility for the same person to develop and test the HMI, both because of the different working environments they are set in and the different areas of knowledge needed to perform the two tasks. There is a need for a solution that makes it possible to create simulations of FUs that can communicate with the HMI. In order for this communication to work there must also exist some sort of interface between an FU and the HMI Engine in Populus used to execute an HMI, using the ODI protocol.

This is both a challenging and interesting problem in several ways. The editor must be as user friendly as possible to be able to be used by non-programmers and to minimize the learning curve for the product. At the same time, it must also contain support for all the functions that might be specified for the FU. The communication interface must be general enough to be able to handle all different types of FUs since it can never know beforehand which FU the user wants to simulate. It is also a challenge to separate the communication interface from the editor and simulation part as much as possible to make it standalone and hopefully reusable in other products.

The aim of this thesis is to find a way to test an HMI without first inserting it into the target environment, by using a statechart based simulation of the desired FU, rather than the FU itself. The project is going to be developed in two different parts which together will constitute the complete simulator made to solve the problem at hand. The first part consists of the editor used to create a graphical representation of the FU that is going to be simulated. According to a requirement from Mecel this representation should be done as a statechart. It is important that the editor should be easy to use even for users who are not familiar with programming. The second part is dealing with the communication interface between the simulated FU and the HMI Engine. This interface should be standalone from the editor so that it is possible to reuse it in other applications, should the company need or want to. It is also important the final product behaves realistically meaning the HMI and HMI Engine must not be able to distinguish it from a real FU during runtime. If possible, the simulator should be able to be built in to the existing Populus editor.

The remainder of this report starts off with a background chapter describing the Mecel Populus project from which this thesis has spawned along with a brief introduction to domain-specific languages and Statechart UML. It continues with a feasibillity study carried out to investigate any existing systems and tools that could be used in solving the thesis problem. Following this is a chapter stating the prerequisites of the implementation which is described in the chapter after that. Next there is an analysis

of the implemented system and the report is then concluded with discussions about the work performed and some possible futire extension to it. The thesis also comes with five appendices, the first two containing the requirements specification and the project plan. The third contains the test specification as well as a compilation of the test results and the fourth gives a short description of the system architecture to simplify any further development. The fifth and last appendix is a user manual for the implemented Renegade Simulator.

2 Background

2.1 Mecel Populus

The Populus suite provides a set of tools for designing and developing Human-Machine Interfaces for distributed embedded systems without having to write any software. Its goal is to make HMI development different from the traditional approach of writing code, making it possible to remove the barriers between the people working with requirements, system engineering, HMI design and implementation.

The Populus suite consists of three parts; the Populus HMI Editor, the HMI Database and the Populus HMI Engine, shown in figure 1 and explained below.

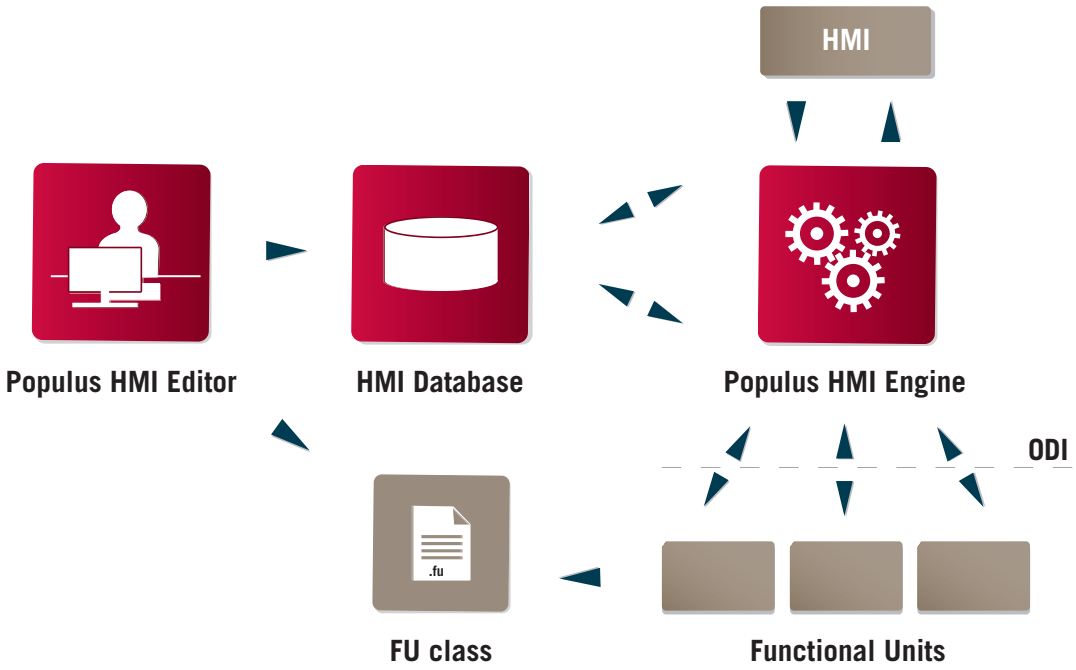


Figure 1: The architecture of the current Mecel Populus system

Populus HMI Editor is used to create both HMIs and FU classes and while the HMI can be thought of as a control panel for functional units the FU classes are textual representations of those FUs. An FU class describes for instance the different operations an FU can perform as well as any data it can store. When created the HMIs are stored in the HMI Database which is deployed together with the Populus HMI Engine. Once deployed the Populus HMI Engine runs the HMI and communicates with the FUs using the ODI protocol [34, 38].
plications and does not need to change when the HMI changes [34, 38].

2.2 Domain-specific languages

The aim of the thesis' implementation phase was to develop a prototype for a simulator tool that would allow for easy testing of a HMI without the presence of an actual FU to drive it. Since the FUs may be almost any kind of device the users, who may not have any programming skills, must be able to configure the simulator to behave as the desired FU and to send and receive different types of information to and from the HMI being tested. In cases like this a DSL, a Domain-Specific Language, may be a good solution to the problem.

2.2.1 What is a domain-specific language?

The concept of a DSL, what defines it and what its boundaries are, is somewhat unclear and one might find different definitions of it in different parts of the relevant literature, but one definition offered by Martin Fowler is:

> *Domain Specific Language (noun): a computer programming language of limited expressiveness focused on a particular domain [19]*

From this definition we get that a DSL is a computer programming language in the same way as for example Java or C is. What separates a DSL from such general purpose languages (GPLs) is its focus on a particular domain and that it trades generality for expressiveness in that particular domain [24]. While Java or C may be a general solution to many different software problems a DSL is aimed at a specific domain and cannot be used to solve problems that fall outside of the boundaries of this domain [10].

A DSL may be either textual or graphical and it will also fall into one of the two categories external or internal DSL. The difference between textual and graphical should be self explanatory, but what distinguishes an internal DSL from an external may require some more explanation. DSLs are often used within larger applications

and while an internal DSL uses the same GPL as the application it is contained in, although in a more limited style, the external one is written using a custom syntax. This custom syntax may be unlike any other existing syntax or it can be based on an already exiting syntax, like for instance XML. Both categories of DSLs have their advantages and disadvantages when it comes to their creation and use. While an external DSL allows the author the flexibility and freedom to design as he or she pleases it also charges the author with the responsibility of creating a compiler that can parse the grammar and symbols of the DSL, thereby making it a useful tool. An internal DSL on the other hand is a subset of an already existing language and as such enjoys the benefits of existing compilers and parsers. The downside is that the author is constrained by the host language which may, should it not be flexible enough or the author not skilled enough, prevent him or her from creating a truly efficient DSL [45, 46].

2.2.2 Why use a domain-specific language?

When DSLs are used there is always a clearly defined problem domain within which the DSL is to be used. If there is no such domain then the language in question is, as per the definition in the previous section, not a DSL, but more likely a GPL. Within this domain, which may vary in size, people may be working who have a very good understanding of the domain. These are the domain engineers or domain experts. As an example, let's assume there exists a problem within this domain and that this problem is to be solved by creating a new tool. The experts may lack the programming skills needed to create this tool and the programmers who are to implement the tool lack the domain knowledge to fully understand the problem. This situation is not uncommon in software projects and this might lead to collaboration issues between the two groups of people. Collaboration issues like this have the potential to become a major source of problems when it turns out that the programmers and the experts are not really understanding each other. By creating a DSL that defines the behaviour of the tool in terms familiar to the experts they can be involved in the development process, validating the programming (within the limits of the DSL) done by the programmers and hopefully spotting mistakes before they become real problems. In short, the use of a well written DSL may greatly improve the communication between programmers and domain experts and thereby shortening the development process [9, 19, 39, 47].

2.2.3 Domain-specific languages in the thesis work

The simulator developed in this thesis work is driven by a statechart created using a DSL based on Statechart UML. The fact that it is made up of a subset of existing syntax defines this DSL as an internal DSL.

2.3 Statechart UML

Statechart UML is used to create diagrams that depict the different states a system can be in and how the system behaves in response to different events. The example in figure 2 shows a very simple statechart representing something that could be built and simulated with the system developed within this thesis work, namely a CD player. The system has two states called Stopped and Playing and there are two transitions that make it possible to move between them. In this case the upper transition is triggered by the event issued when someone presses the play button, thereby moving the CD player to the Playing state. However, the transition will only be taken if a certain guard, a boolean expression, is true. In this case the guard is named ”cdInserted”, meaning that there should be a disc inserted into the player. If not, the system will remain stopped. When the transition finally is taken an action will occur which in this example means that the CD will be played. Once playing the CD player can again be placed in a stopped state by an event caused by the pressing of a stop button.

Figure 2: A simple example of a statechart

A UML statechart also offers the possibility to perform actions upon entering or exiting a state. These actions are atomic and can for example be the updating of a data value or the starting of a timer.

The example below of an extended version of the earlier CD player shows how it is possible to make a more elaborated statechart for an application while still using the same set of elements. There are more states and several ways to travel between them, giving the application more functionality. Some of the transitions are triggered by manual signals, like the one marked ”Play[cdInserted]/Sound” going from Stopped to Playing, while others are triggered when a certain timer has expired. An ex-

ample of this type of timer is the transition marked ”100ms[isStopped]” which leads from Skip Rew to Stopped and will be taken after 100 ms assuming that the ”isStopped” guard is true [8, 15, 33, 41, 44].

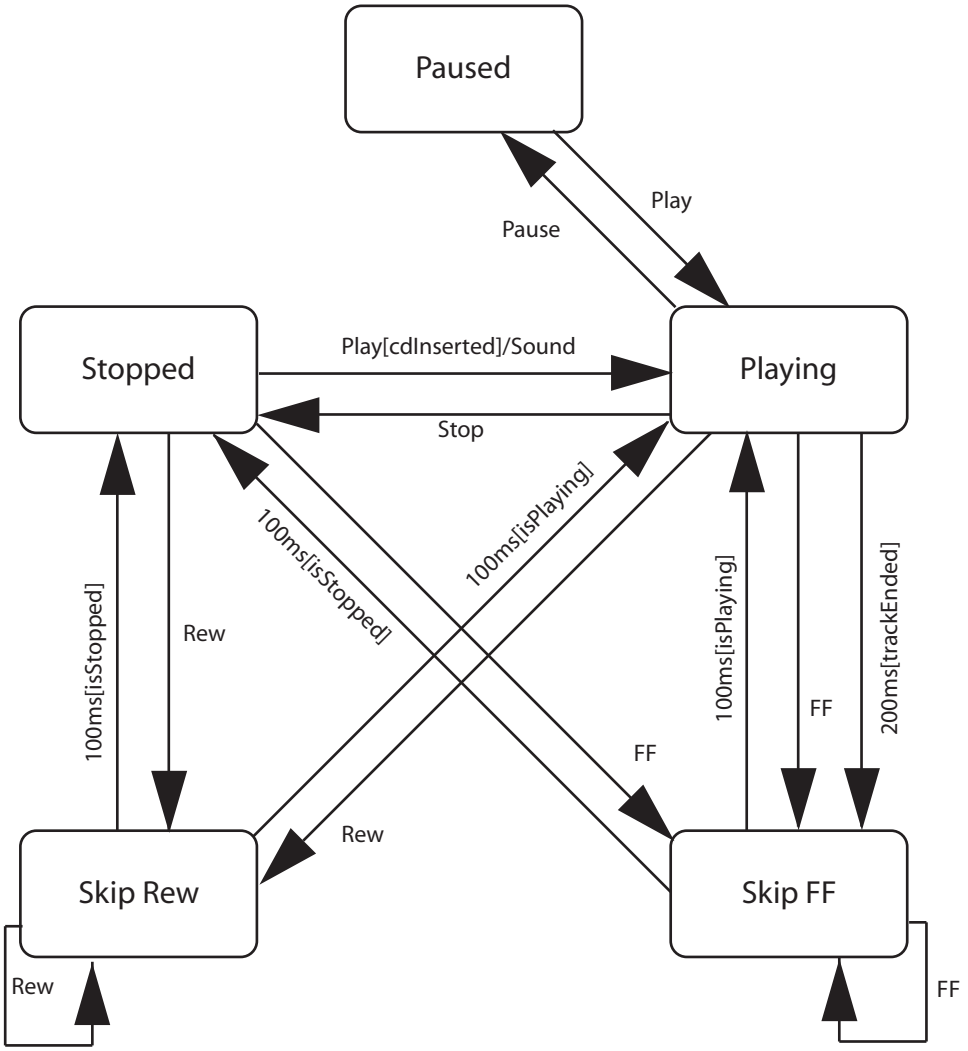


Figure 3: An example of an extended statechart

3 Feasibility Study

3.1 Existing tools and systems

3.1.1 SCXML

Statechart XML or SCXML for short is published by W3C and is an XML based language used for representing UML statecharts [48]. It uses a set of XML tags and a terminology that is focused specifically on the statechart domain. Although still a work in progress and thereby lacking some of the functions found in UML statecharts it offers a construct called custom action (not to be confused with the UML action), that can be created to represent any function that can be used in a statechart. As many custom actions as is needed can be created in order to close the gap between SCXML and Statechart UML functionality.

3.1.2 Rational Software Architect

One of the larger existing products found was Rational Software Architect, RSA, created by IBM. It is a tool made for building UML statecharts via a simple graphical point-and-click interface and transforming them to ordinary programming languages [30]. Out of the box it does not transform into SCXML, but with the help of an additional plugin that could be achieved too.

RSA is built on the Eclipse platform, but it is not open source software and can therefore not be developed any further. It is possible to create other standalone features to use in addition to the ones provided in the original system though [25, 26, 27].

3.1.3 UniMod

Another already existing product was UniMod, short for Unified Modeling, which is a project focused on designing and implementing applications such as FUs [17]. It contains two environments, one for designing the application in a sort of class diagram where the connectivity between the elements of the application is defined, and one runtime environment where the user can build a statechart corresponding to the application and run a simulation of it. Along with the class diagram, appropriate Java classes corresponding to the different elements are created automatically. Often these files must be manipulated by hand in order to achieve the required behaviour of the application. After building the application diagram, the user can generate an XML representation of the statemachine and then run it using a runtime framework that is part of Java Finite State Machine Framework [16]. It also has a debugging mode, making it possible to track where you are in the statemachine during runtime.

UniMod is an open source plugin to Eclipse and uses the GNU Lesser General Public License, LGPL [20]. This means that it is allowed to use the code freely, but the parts of the developed product that uses it also becomes open source under the same license rules. It is possible to extend and modify the existing Java code, thereby equipping the system with any functions that might be needed [18].

3.1.4 Graphical Modeling Framework

The Graphical Modeling Framework, GMF, is a framework for the Eclipse platform used to develop graphical editors that are primarily used for diagrams such as statecharts [12]. A developer starts by creating a model, for example a class diagram or an XML schema, from which GMF can auto generate Java classes that make up an editor. This means that there is no need for any Java code to be manually written in order for the editor to function. With a model that is detailed enough and by adding diagram parts defined by ordinary image files like JPEG or GIF GMF can create a tailored diagram editor to suit a specific project. If more advanced features are required it is also possible to modify the generated code by hand in order to make further changes to the editor.

Figure 4 shows the GMF dashboard, used in Eclipse as a guide through the development process of a GMF editor. It distinguishes all the models that have to be created and combined to create an editor. Just as a DSL is defined by its metamodel a GMF editor is defined by its domain- or ecore model which is also the first thing created by the developer. This ecore model contains definitions for all the entities and relationships that will be part of the editor, such as states and transitions [21, 40]. From the ecore model the developer can then derive a domain generator model. This mo-

del takes the definitions in the ecore model and generates parts of the editor code, mainly for the modelling parts like the interfaces of the transitions and states.

The graphical and tooling models are very much alike, since they both contain information about the visual elements in the editor. The first one is used to define the graphical elements used when drawing in the editor and the second one to define the different tools in the editors palette. By binding the graphical and tooling models to the ecore model, the developer receives the mapping model containing all the information about the GMF editor. This can then be transformed into the final model, the diagram editor generator model. This model provides validation support, pointing out the occurrence of any errors in the previous models, as well as containing all the properties needed for generating the source code for the diagram editor.

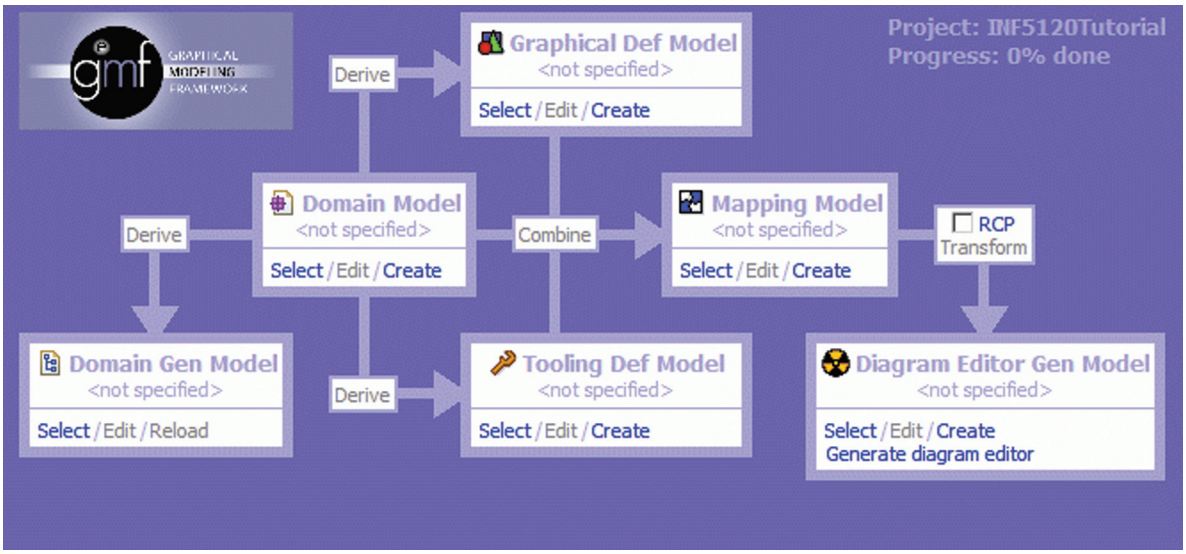


Figure 4: Screenshot of the GMF dashboard

An editor created with GMF can be made to be a standalone product rather than having to be run within Eclipse. GMF is rather new but was found to be quite well documented with an API as well as several examples and tutorials [13, 14, 22, 28, 43].

3.1.5 Apache Commons SCXML

Apache Commons SCXML is aimed at creating and maintaining a Java based engine capable of executing a statemachine defined in an SCXML file [5]. It is part of the Commons project which is an Apache project focused on reusable Java components [1]. Commons SCXML is a rather new product, having been around for less than two years, but it is very well documented with an extensive API. Some tutorials and ex-

amples are also provided, although, at least to this date, not very many or detailed. Just like UniMod, Commons SCXML is open source and placed under LGPL. This makes it possible to adjust the code to fit the specific target system but the resulting product, or at least the parts that uses the Commons code, must be open source too.

3.2 Discussion and final decisions

Directly after the study, the initial thought was to use one of the researched systems. That way any redundant implementation would hopefully be reduced to a minimum, allowing more time to be spent on extending the developed system with more advanced functions. This meant using either UniMod or RSA as they both contained diagram editors and functions that could perform a transformation to runnable code. In UniMod, it was also possible to actually run the resulting code.

RSA was quite quickly dismissed from the proposals as it was not free to use, which was important for being able to use it in the thesis work. It was also not sure how easy it would be to expand and complement the existing editor since it was not open source. The system also had another large flaw which was the lack of support for timers, the ability to move from one state to another after a certain time given by the user. Being able to use timers was a requirement from Mecel since timers are quite common when using statecharts and especially when simulating FUs.

For a while, UniMod seemed to be a good choice of system to work with. It contained almost all the basic features needed for the graphical part of the thesis project along with its own internal engine to run the statechart. However, a closer look showed that the code base was very large and not too well documented, and it was decided that it would take too much time to comprehend it well enough to be able to modify it. Another negative was that the construction of the initial class diagram required too much programming time and skills from the user. Finally, the current version of UniMod was designed for an older version of Eclipse and there was some issues with getting it to work correctly on the newer version that was already chosen for the thesis project.

As all candidates were dismissed the only remaning and viable option was to build the statechart editor from scratch. Since it was already stated that the programming part of the thesis work would be done in Eclipse, GMF was the obvious choice for this development. It provided an easy way to create an editor with the most basic functions for drawing diagrams and it was also the suggested choice by Mecel.

There was still a need for a way to represent the graphical statechart in code so that it could be run by an engine. The natural choice in this case became using SCXML. The other possible option, and also the initial suggestion from Mecel, was to use XML, but since the editor should be designed only for statecharts, it seemed like a much better idea to go with SCXML, a subset of XML adapted especially for this purpose. It was still easy to use and comprehend based on the knowledge of XML and considering the time frame for the thesis work the learning curve was considered to be acceptable.

When the decision to use SCXML was made, the following decision to use the engine from Commons SCXML was not a hard one to make. It was designed to run statecharts defined with SCXML and as it was built in Java it could easily be adapted to suit the needs of the project. Also, it removed the need to implement an engine from scratch, thereby saving both time and trouble. One possible negative was the fact that it was open source, but after getting the green light from the supervisors it was decided to use the Commons SCXML Engine anyway.

4 Prerequisites

4.1 Problem domain

In order to develop a good and usable system the boundaries of the problem domain had to be identified along with the intended users. As the intention was to use the developed simulator together with the Populus Suite it followed that the problem domain would fall within the domain of HMI development. Following the same logic the intended users would be users with good knowledge of the Populus tools provided by Mecel. In addition to this Mecel also suggested that the simulator would use UML to represent FU behaviour as statecharts and as a result the intended users also had to have some knowledge of Statechart UML.

4.2 Project model

Once the feasibility study was completed the development phase could begin. First off requirements were gathered and compiled into the requirements specification found in Appendix A. Next the project plan, Appendix B, was reworked so that the time plan that was initially created at the start of the project also included the iterative development process that was about to start. Once these steps had been taken along with some initial discussions about the design of the system the actual implementation started.

Throughout the implementation phase continuous testing has been performed to ensure a working system. Also weekly meetings were held with the project supervisors to make sure the system prototype was progressing according to their requirements.

4.3 Requirements

The requirement specification consists of two groups of requirements where the first deals with system requirements and the seconds with developer requirements. The system requirements are based on discussions with the project supervisors and aims to ensure that the system functionality will be according to their wishes. The developer requirements are there to ensure that the work on the simulator can be continued after the thesis project comes to an end.

The following sections describes some of the more important requirements and the full requirement specification can be found in Appendix A.

4.3.1 System requirements

GI1: The drawing of a statechart shall be done using a simple point-and-click method.
Simple point-and-click shall be used to allow non-programmers to easily use the system.

NGI1: The interface shall communicate with the HMI Engine via the ODI protocol over TCP/IP.
A real FU would communicate with an HMI using the ODI protocol over a TCP/IP connection and as it is important that an HMI cannot distinguish between a real FU and a simulated one the simulator must utilize the same communication methods as a real FU.

F4: The simulator shall be able to handle simple, non-nested statecharts.
As any FU, simple or complex, can be represented by a non-nested statechart the simulator must be able to handle any such statechart.

F13: The user shall only be able to use the elements predefined in a FU class file.
As HMIs are constrained by the FU class they are based on it is important that the simulator statecharts created for an HMI cannot contain data or functions that cannot be handled correctly in in the HMI.

CO1: The ODI Interface shall be separated from the editor.

Mecel wants the option of reusing the ODI interface created for the simulator in future applications.

4.3.2 Developer requirements

Eclipse shall be used for development
The Eclipse IDE [11] is already used at Mecel and shall therefore, and because it supports GMF, be used for the development in this project.

ClearCase shall be used for version control
Mecel uses the Rational ClearCase [29] and as version control was required for the thesis project ClearCase shall be used to store any developed code.

4.4 Simulator structure

It was decided early on that the system was going to have two distinct parts, one graphical and one non-graphical. The first would present the user with the editor and tools used to create statechart diagrams while the second would contain the engine driving the simulator along with the communication interface. Developing the system like this would comply with Mecels requirement to have the communication interface, the ODI Interface, reusable and it would also allow for the development of the two parts simultaneously, reducing the time used for the implementation and making it easier to test the individual parts during development. The final system was made to consist of three parts where the non-graphical part was made up of the ODI Interface and the Renegade Engine and the graphical part made up of the Renegade Editor as depicted in figure 5 on. An overall description of the system can be found in Appendix D.

In order to provide the user with a familiar GUI it was important that the terminology used in the Renegade Editor did not differ too much from the terminology used in the Populus suite. However, some of the terms used in Statechart UML also exist in Populus but there they have a completely different meaning. For instance, an action in Populus is described as an operation on an FU like pressing the play button on a CD player. This function would however correspond to what is called an event in Statechart UML. To avoid any confusion the Populus terminology is used in the Renegade Editor and also throughout this report if not otherwise stated.

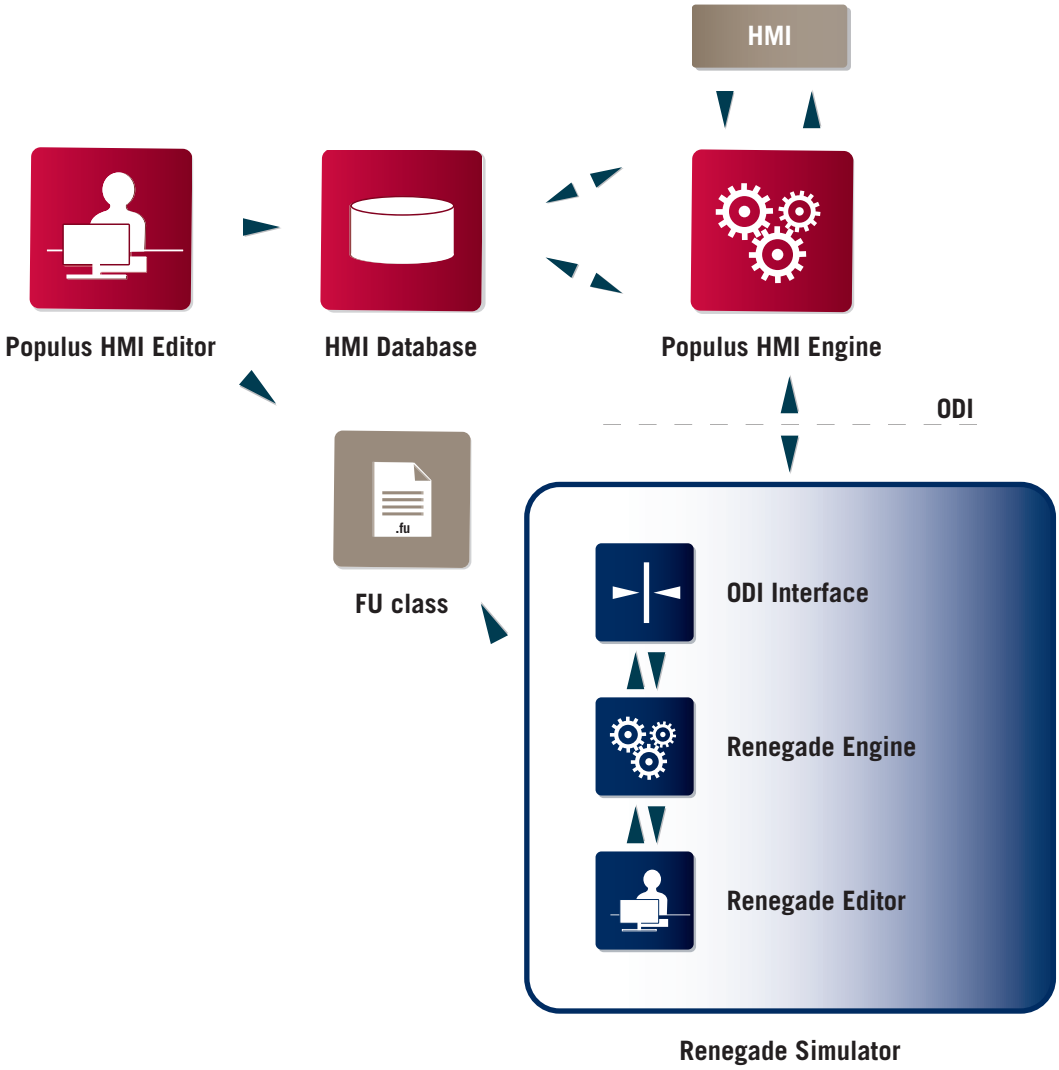


Figure 5: The architecture of the current Mecel Populus system, with the addition of the Renegade Simulator

5 Implementation

5.1 Renegade Editor

The implementation task was to create a system consisting of an editor for the modelling of a statechart that in turn is used for simulating an application (for example a radio or a CD player) using a simulation engine. This simulator is referred to as Renegade Simulator and the application is the previously mentioned FU. This together with Mecel's existing Populus HMI editor, used for defining an HMI, and their HMI Engine, used for running the HMIs defined in said editor, will be used to simulate a complete system.

5.1.1 Basic design

In order to create an editor in GMF, there was a need for an ecore model from which the creation process could start. For the purpose of this project this model was made up of the class diagram depicted in figure 6. The ecore model describes the different types of objects that can be used in the Renegade Editor, the attributes defined for these objects and the different relations between the objects. As shown in the figure a statechart can contain at most one initial state, zero or more final states and any number of ordinary states and transitions. Each of the three state types has a name attribute, zero or more transitions and zero or more data elements connected to it. When a transition is connected to a state the transition must be aware of both the state from which it originated and the state to which it is leading. A data object on the other hand is simply a data carrier with no knowledge of the state to which it is connected. Each class in the class diagram has one or more attributes defined for them and in the Editor these attributes are given values through the use of configuration wizards which are further described in section 5.1.3.

Using this class diagram as a foundation the editor could be developed using the dashboard tool provided by GMF. In the first iteration, no extra design choices were made and the creation process was more or less just run through from start to finish. This resulted in a rather plain editor, not at all adapted to the demands of the thesis

project apart from providing functions for drawing nodes and vertices. Beside its unimpressive look and features which can be seen in figure 7, this editor had all the necessary files and packages needed for any working editor.

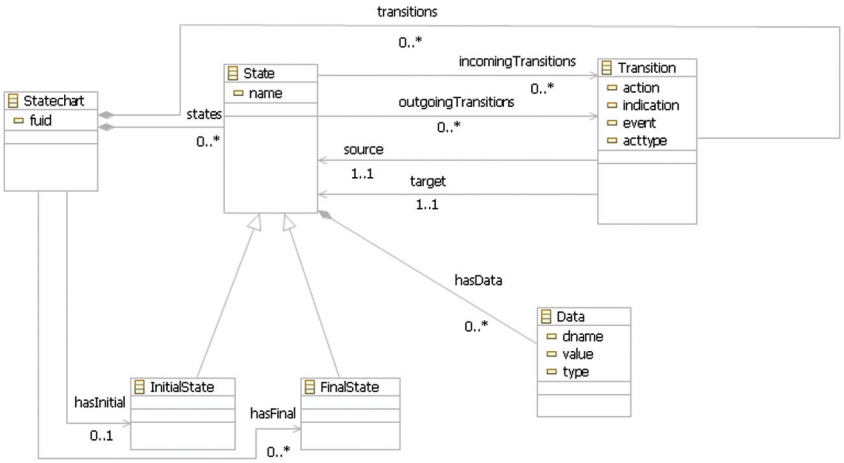


Figure 6: The ecore model, in form of a class diagram, used for the Renegade Editor

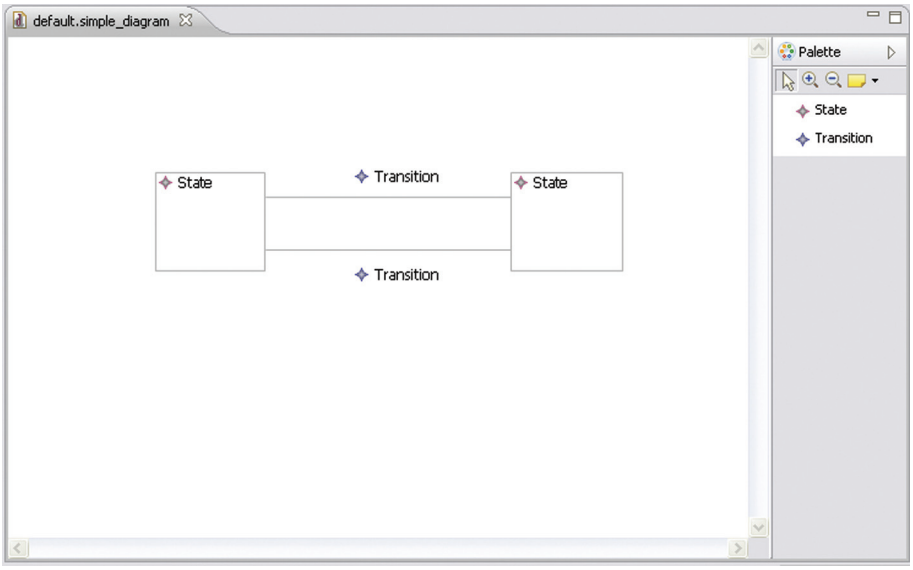


Figure 7: A very basic diagram editor

The next steps, concerning graphics, internal functions and user interaction, were iterated several times before reaching the finished editor. The result of these iterations can be seen in figure 8, showing a diagram that is somewhat similar to that in figure 7, but more pleasing to the eye. Although this can not be shown in a picture, this diagram is also easier to manipulate for a user and contains support for more functions than the previous one.

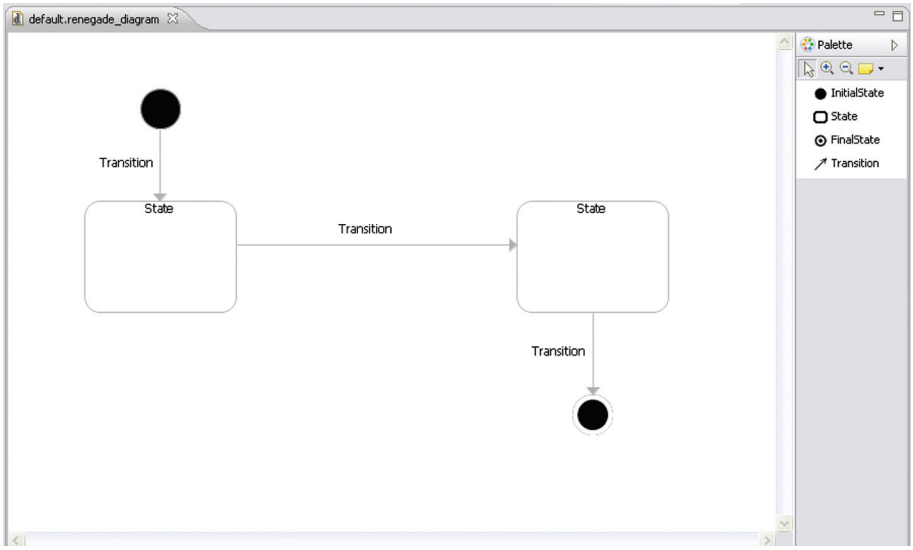


Figure 8: A more developed version of the diagram editor, showing canvas and palette.

By using this iterative development process, small amounts of functionality could be added in each iteration. This allowed for testing and approving new functionality before adding further features, which is in sharp contrast to an all-at-once type of development, which most likely would have resulted in more time spent on redesign and corrections. This way, it was possible to always have a working version of the Renegade Editor to use in demonstrations.

5.1.2 Graphical Design

Since this was going to be an editor for statecharts, it was decided that the various graphical parts should be modelled to look like the ones in Statechart UML. This was also chosen in order to let a user work with shapes that was familiar to him or her. The original nodes and vertices were remodelled to look like UML states and transitions, the states shown as boxes with rounded corners and the transitions as arrows. Also, the nodes for the special states, the initial and final states, were desig-

ned according to the UML standard as a black circle for the initial state and a black and white circle for the final state. These design choices are depicted in figure 9.

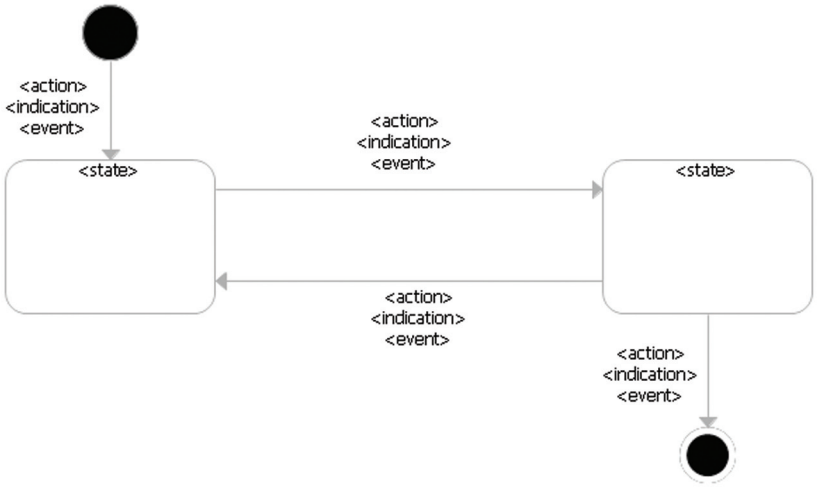


Figure 9: An unconfigured diagram drawn in the Renegade Editor

There were also some constraints defined for the transitions. There can be no transitions going in to an initial state, since it should never be possible to return there once a simulation has started, and there can be no transitions going out of a final state, because when a final state is reached the simulation is finished and should not be able to proceed into a new state.

Both the similarities as well as the differences between the Populus terminology and the Statechart UML terminology became somewhat of a dilemma when designing the Renegade Editor. For that reason the decision was made to abandon the UML way of naming functions and use the Populus terminology instead. It was considered to be better to stay with this terminology since the Renegade Editor was supposed to be a complement to, and possibly even a part of, the Populus suite. Using two different terminologies within the same program would be confusing to the users, so this deviation from UML standards was decided to be an acceptable one. The elements of the created diagrams would still be labelled according to UML as to not cause any further confusion when looking at a diagram. The actions are displayed in plain text, the indications are enclosed in brackets, and the events are shown with a forward slash in the beginning as shown in figure 10.

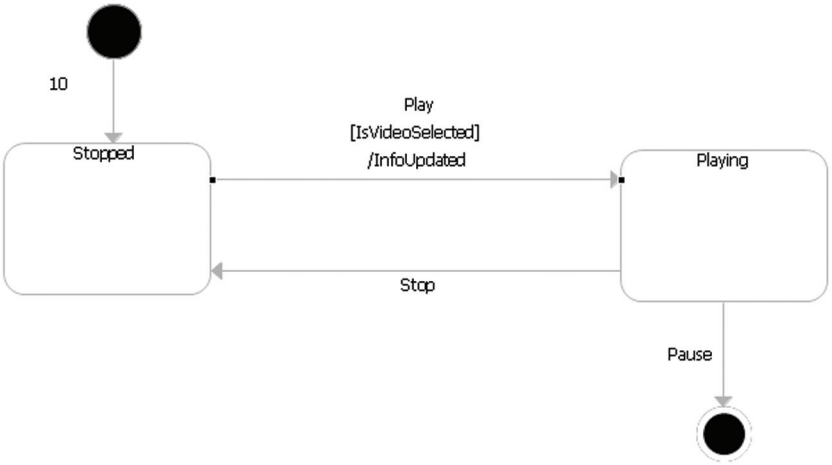


Figure 10: A configured diagram drawn in the Renegade Editor

The Renegade Editor also had to allow the user to update data and indication values during the simulation and there was a discussion about where these settings should be performed. The two options were either when entering a state or when travelling to said state. From the existing FU classes it was found that the most natural choice would be to set these values upon entering a state. For example it is more correct to say that a CD player is playing only after it has actually begun playing than it is saying it just before it begins to play.

Even though data and indications were to be set in the different states of a statechart it was decided that neither the data values nor the indications were to be visible directly in the state element in the diagram. The reason for this was that it was found, even for the small example diagrams used during development, that as the number of values grow the size of the state element must be increased to accomodate them all. This in turn would lead to large and cluttered diagrams that would be hard to overview and grasp.

5.1.3 User interaction

All the information that can be placed in the statechart in the Renegade Editor is predefined in an FU class. The expected user of the system was supposed to have knowledge of how to create these files, so it could be understood that he or she could read them and would know from the start what information was set in the current file. Even though this would make it possible for the user to enter the appropriate state

names, actions, indications and so on, using text boxes connected to the different diagram parts, this was not considered to be the best approach. It was rejected for two reasons, the first being that allowing users to enter the information on their own would result in a higher risk of errors, since the text might be spelled incorrectly and therefore not be matched to the FU class file. The second was that since all the information concerning a certain FU was already available in a file it seemed like a waste of resources not to use it rather than having it as a reference on the side. Hence, a link between the diagram and the FU class was implemented.

There was still the issue of entering the information in the correct places in the statechart. A first proposal was to use drop-downboxes in the states and on the transitions. The information was to be loaded into these boxes and the user could then choose for example what action would trigger a certain transition. This seemed like a good idea in the beginning, but as the development progressed it became obvious that it would be too muddled and confusing to use this design and it ended up being rejected.

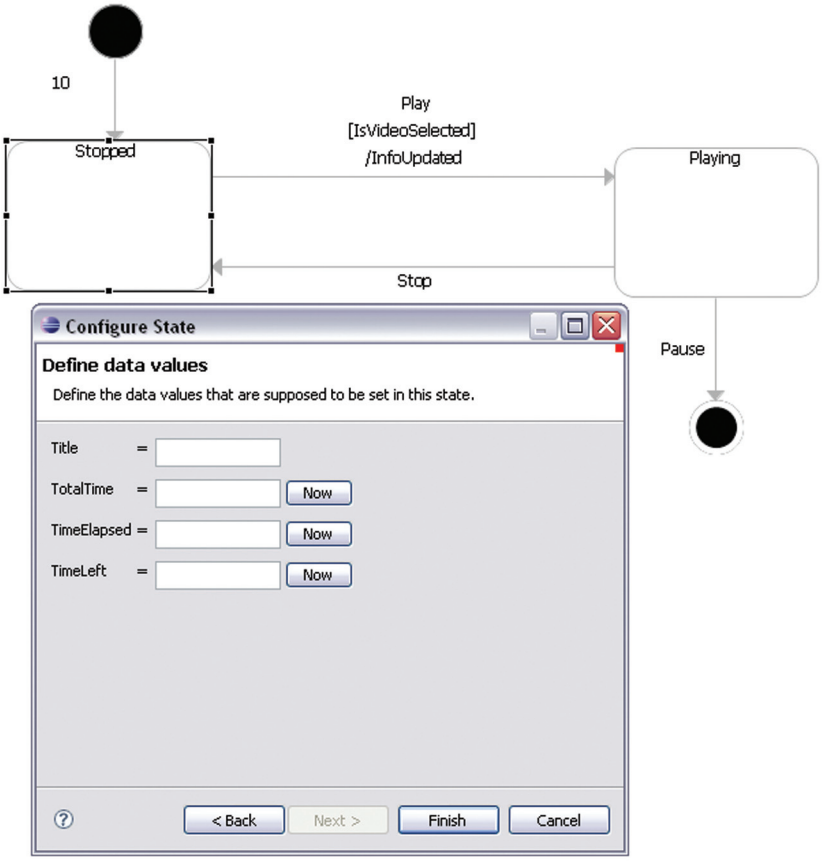


Figure 11: An example of a window displaying a wizard page in the Renegade Editor

The proposal that ended up to be the final solution was to create two configuration wizards, one for states and one for transitions. By selecting a certain state or transition, a wizard is made available from the context menu of the selected diagram part to guide the user through the configuration, as seen in figure 11. On each wizard page there is one or more drop-down boxes with information loaded from the FU class file, where the user can choose for example one of the predefined state names, or what event should be sent out when taking a certain transition. This step-by-step solution was deemed to be the most user friendly, since it could contain some explanatory text on each page in order to make the statechart configuration even easier and it also allowed the user to concentrate on one part of the configuration at a time. In addition to this it was also possible to save the user's choices so that if he or she needed to reconfigure for example a transition, the chosen action, indications and event would already be chosen in the wizard. This way the configuration did not have to be done all over again, the user could just click through the pages that were not supposed to be changed without having to make the same choices again.

5.1.4 Simulation feedback

In order to provide information about the progress and outcome of the simulation, there was a need for a feedback mechanism in the Renegade Simulator. This feedback is provided in two different ways where the first is presented visually in the Renegade Editor and the second is given as a textual log. The use of a text log was convenient because SCXML has a built-in support for creating such logs. This log, implemented using log4j [2], can give the user information about when a state is entered or exited, when data is set and indications sent and so on. Depending on how it is configured it can also provide more detailed system information making debugging and error correction easier. The visual feedback is given through the graphics of the statechart diagram. It shows the user which state the simulation is currently in by marking it with red border making it possible to follow the movement through the statechart during the simulation.

5.1.5 SCXML Transformation

When a statechart diagram is created in the Renegade Editor two files are created. One is the actual graphical diagram file and the other is a textual representation of the diagram in the form of XML Metadata Interchange, XMI [49]. To be able to run the simulation using the Renegade Engine the XMI had to be translated into runnable SCXML. Since both files, the XMI and the SCXML, are based on XML, the transformation was to be done using XSLT.

In the Renegade Editor the transformation process can be done in two different ways. The user can either choose to start a simulation and by doing so the application creates an SCXML file which is sent to the engine as input. It is also possible to only perform the transformation and saving the SCXML file to disk. By being able to actually look at the SCXML file the user could possibly be helped in a debugging process or could, presuming that he or she has the necessary skills, make manual changes to the statechart behaviour.

5.2 ODI Interface

The purpose of the ODI Interface is to handle the communication between the Populus HMI Engine and the Renegade Engine which is an extension to the Commons SCXML Engine. The two engines can not communicate directly since their messages are on different forms. The ODI Interface receives the different messages sent between them and translates them into the correct form in order for the receiving engine to understand them. In this thesis four basic types of ODI messages, described below, are used.

5.2.1 ODI Message Description

When something has happened in the HMI, for instance when a user has selected an item in a menu or pushed a button, information about this is sent to the simulator as an ODI Action message. When the ODI Interface receives this message from the HMI Engine it triggers the corresponding transition in the statechart. Figure 12 displays such a flow of events.

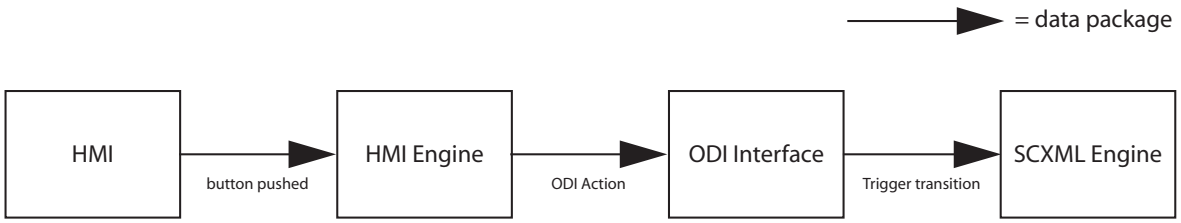


Figure 12: An example of a course of events that include an ODI Action message

The ODI Event message is used to tell the HMI to perform things like displaying a popup window or in some other way notify the user of changes in the FU. The ODI Event message is triggered when a taken transition is configured with an event. An example could be when moving from a stopped state to a playing state a pop up window saying ”Playing” should be displayed. A course of events that include an ODI Event message can be viewed in figure 13 below.

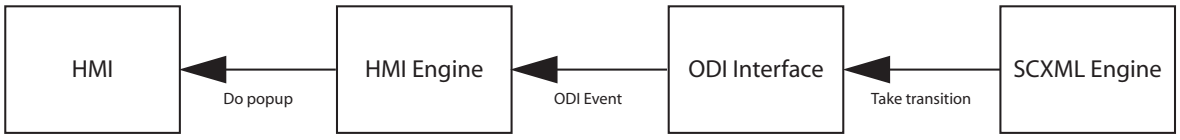


Figure 13: An example of a course of events that include an ODI Event message

Populus indications are booleans used in the HMI to keep track of how the HMI should be presented in different states, like ”the play button should be enabled when in the stopped state” or as a guard inside the statechart allowing the use of a transition. Whenever an indication is changed in the statemachine the ODI Interface should send a message, called an ODI Indication message, to the HMI Engine. This message contains all of the indications present in the system, whether they have been updated or not. An example of a course of events that include an ODI Indication message can be viewed in figure 14 below.

Figure 14: An example of a course of events that include an ODI Indication message

The ODI Dynamic Data message is actually two different types of messages, the ODI Dynamic Data Request Message and the ODI Dynamic Data Response message. The HMI Engine sends a request message on behalf of the HMI to the ODI Interface in order to register or unregister a subscription to data updates. As long as an HMI has subscribed to a certain set of data it should receive a message whenever this data has changed. This is done by sending a response message containing the updated value to the HMI Engine. The message itself can be a single frame message or a multi-frame message and the difference between them lies on the different data types they can carry. A frame is a block of data with fixed size and the larger the message the more frames are needed. There are 22 different types of data that the ODI protocol can handle and they can represent among other things the speed value of the car or the title of a song being played. Figures 15 to 17 illustrate example scenarios where data request and response messages are sent.

Figure 15: An example of a course of events that include an ODI Dynamic Data request subscribe message

Figure 16: An example of a course of events that include an ODI Dynamic Data response message

Figure 17: An example of a course of events that include an ODI Dynamic Data request unsubscribe message

The messages received from the HMI Engine are on serialized form and the messages sent to the HMI Engine should also be on serialized form. The collection of classes and interfaces created to accomodate this functionality was collectively referred to as the Populus Java Interface (PJI) and included functionality for input and output streams, serialization and deserialization of request and response messages and different enumeration types used in the ODI protocol. A basic but incomplete implementation of the PJI had been developed by Mecel as part of a previous project, the aim of which was to create Java representations of functional units and from this foundation a working version of the PJI was created and called the ODI Interface. The developed package does not cover all aspects of the ODI protocol but it does provide enough functionality to cover the requirements of this project.

5.2.2 Populus Java Interface

As stated before the basis for the PJI implementation was incomplete and in many ways lacking the functionality necessary for this project. This proved to be a greater challenge than expected for several reasons, the main one being integration problems with the HMI Engine. The root of these integration problems was a difference between the documented ODI specification and how this had been realized

in the HMI Engine. It turned out to be quite difficult to identify this error as the HMI Engine never really reported any errors. It simply accepted the, as it viewed it, faulty message from the PJI and discarded it without complaints. The problem was only pinpointed once the HMI Engines logger application that was used to monitor what happened on the HMI Engine side had been reworked to display all messages received, even the faulty ones. Once this new logger was introduced the cause of the problem could be determined. At this point it was decided that the implementation of PJI should be changed to not follow the documented ODI specification but instead match the implementation used in the HMI Engine.

This process of debugging the communication, reworking existing code and finally solving the issues was a lengthy process but once completed the ODI Interface included working serialization and deserialization functionality for actions, events, indications and nine different types of data messages.

5.2.3 ODI Message handling

When the ODI Interface first receives a message from the HMI Engine it needs to determine what type of message it is. To do this all of the incoming and outgoing messages from and to the HMI Engine implements an interface called Message. Message is defined in PJI and contains only one function which returns the message type. When this is determined the message is then cast to the specific type of message so that the ODI Interface can get the information needed to continue. To check whether the message is sent from a HMI that is built for the running simulated FU, the ODI Interface keeps track of which simulation it is currently running by storing the ID of the FU class the simulation is based upon. By comparing this stored ID to the incoming messages FU class ID the ODI Interface can determine if the HMI is build on the same FU class as the simulation. Otherwise errors could occur like when the HMI wants to subscribe to data that does not exist in the statemachine, or when the HMI sends ODI Actions that do not correspond to any actions that can be triggered on the statemachine.

As the Commons SCXML Engine in itself has no link to the ODI Interface this had to be created in the Renegade Engine. The link was vital in order to provide the HMI with information about events from the statechart. A number of different solutions were considered where the first was to develop an event notification mechanism [42]. However, since events are predefined values that never change there is nothing to trigger the notification and the idea was discarded. As events are configured for transitions another solution was to create a listener on the transition instead of the actual event. When a transition is taken the listener checks whether there is an event present and if so forwards it to the ODI Interface which turns it into an ODI Event message which is sent to the HMI. Figure 18 shows this setup.

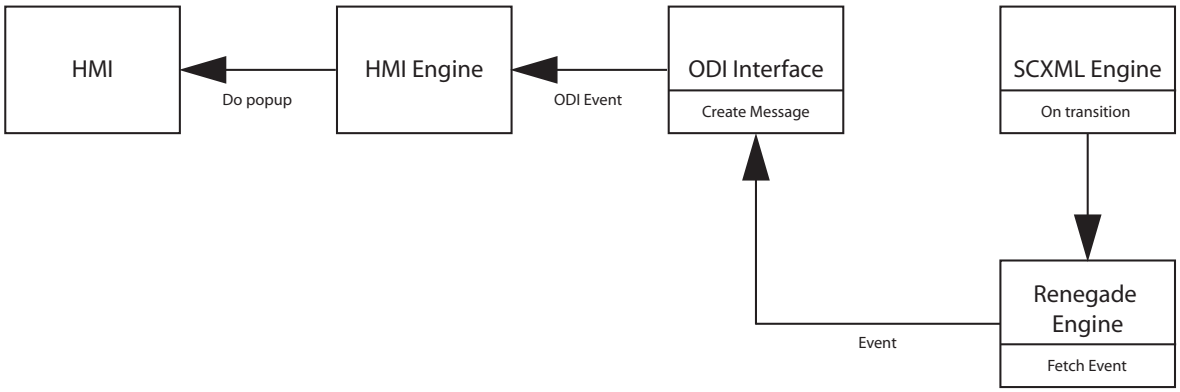


Figure 18: An example of how the ODI Interface receives an event and turns it in to an ODI Event message

Whenever an indication value is changed in the statemachine the new value needs to be sent to the HMI in order for it to be updated accordingly. The original implementation of the Commons SCXML Engine did not provide functionality that made this possible and so it had to be extended. The extension consisted of a notification mechanism that provided the ODI Interface with the updated indication value which could then be transformed into an ODI message and sent to the HMI. An example of the solution can be viewed below in figure 19.

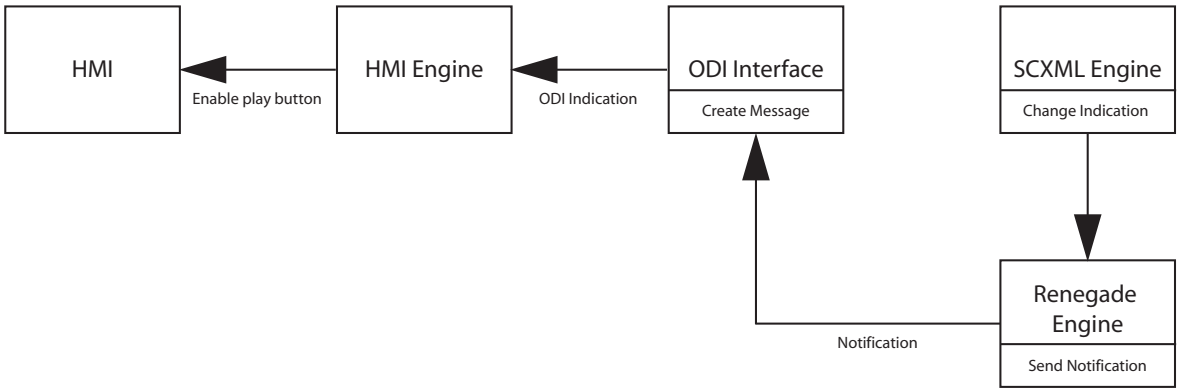


Figure 19: An example of how the ODI Interface receives a Notification and turns it in to an ODI Indication message

When the ODI Interface receives a request message for ODI Dynamic Data it first checks whether it is a request to subscribe to a certain data value or to unsubscribe said value. If the request is for a subscription the data value in question is added to the collection of subscribed data values and if the request was to unsubscribe a data value that value is removed from the same collection. Whenever a data value is

changed in the statechart the HMI should be notified. Although no action is required for unsubscribed data, for subscribed data the ODI Interface needs to compose a Dynamic Data message and send this to the HMI. For each supported data type a specific message class has been implemented.

A problem that arose when implementing the data handling functionality was how to let the ODI Interface know that data had been updated in the statechart. Because of how the Commons SCXML Engine was implemented there was no obvious best solution for this problem and a number of different solutions were considered. The first idea was to create a thread that would poll for subscribed data in the statemachine after a certain time interval. This idea was rejected because there was an uncertainty on how long this pause would have to be in order not to miss any data or use unnecessary CPU power.

A second proposal was to implement a mechanism that on each state entry would compare the data in the state to the data that was previously sent to the HMI. To avoid sending incorrect data this mechanism would use a semaphore lock [7] to prevent the statemachine from moving on. While this lock was in place a polling thread would check all the data values and if any had been updated this would be sent to the HMI. Once this procedure had been completed the lock would be released and the statemachine could move on. The proposal was discarded as it would always check all subscribed data values which could lead to a lot of unnecessary processing.

The third idea, which was eventually chosen, was to have the statemachine notify the ODI Interface whenever a data value was changed. This is basically the same solution as the one used for indications and the existing implementation only had to be changed slightly in order for it to be used for both indication and data values.

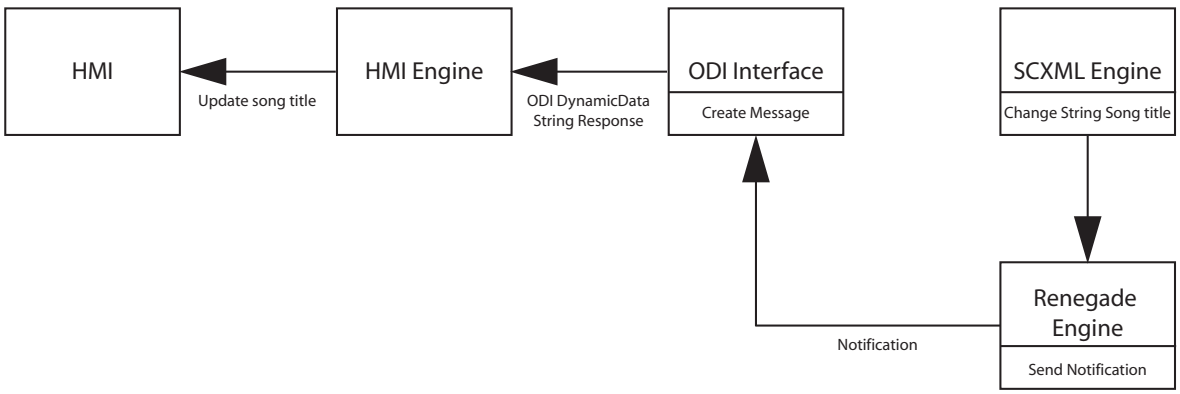


Figure 20: An example of how the ODI Interface receives a Notification and turns it in to an ODI Dynamic Data String Response message

However, in order for the ODI Interface to distinguish between the two notification types a special class, containing both the value and its type, was created and used as a data carrier. This solution, depicted in figure 20, was considered to be the most effective out of the three and it also avoided the synchronization problems that was associated with solutions involving threads.

5.3 System Verification

In addition to the continuous testing performed during the implementation phase the system was also presented to group of outside testers. These testers were, after having received a brief introduction to the system, asked to perform a set of tasks in a simple test scenario. The aim was to gauge the system's usability and to see if it was as user friendly as intended. The result of these tests can be found in the test specification (Appendix C). These user tests were carried out in the developed prototype while the earlier testing was done using the verification and test functions provided by the tools used in the project.

6 Analysis

6.1 Renegade Editor

Even though the Populus terminology was adopted for the Renegade Editor the fact that this differs from the UML terminology can still cause confusion and errors. There are several possible ways to avoid having this risk turn into a real problem. One would be to simply educate the users, another to change the terminology in Populus to match that of Statechart UML or to something all together different from Statechart UML. While the first may be the easiest to realize the last one is what would most likely remove the problem completely.

The system has a flaw concerning the naming of the statecharts made in the Renegade Editor; it is not possible to change the name of a statechart diagram once it is created. Or rather, it is technically possible, but when it has been done the configuration is lost from the diagram and can not be replaced. This is of course rather annoying from a usability point of view. Unfortunately, after doing some research, it was established that this was a quite common problem when using editors created with GMF and nothing that could easily be fixed. It is a negative aspect of using GMF, but it was considered not to be necessary to reject it as a tool because of this, as the advantages with it outweighed the disadvantages. Hopefully, this will be solved in a later version of GMF. It might also be possible to solve by changing the automatically generated code but it was estimated that this would not fit in the timeframe of the thesis work as there were other more important issues were more important to solve. For now, it will be pointed out in the user manual (Appendix E) that a name change is not recommended.

As it is now, some information is hardcoded into the system. For example, the state name is always chosen from an enumeration set in the FU class file called ”Player State”, which of course means that this enumeration set has to be present in the file. This choice was made after having studied several different FU class files provided

by Mecel. This enumeration set was present in all the files that represented the kind of functional units that the thesis work was supposed to deal with. In a future version, this choice should perhaps be changed to a textbox allowing the user to enter a custom state name, provided the care is taken to prevent misspellings and other human-made errors.

6.2 ODI Interface

Though it is considered to be the best of the three different solutions presented in section 5.2.3, the handling of Dynamic Data messages could have been solved more neatly. The reason for keeping the current solutions was that it turned out to be an acceptable way to fullful the requirements of a working system within the time frame of the project.

As a result of a limitation in Populus the implemented simulator is limited to one single HMI display and all the Dynamic Data Request subscription messages concerns that particular display. If these limitations are overcome and more displays are connected to the same FU, the subscription messages cannot distinguish from which display a certain message is coming. This could for instance lead to the adding or deletion of subscriptions for the wrong display.

Java is unfortunately not a very flexible language when it comes to bit handling. Still, since one of the early decisions in this project was to use Java, the solutions to the different problems related to this had to be solved. This turned out to be more difficult than expected because of inconsistencies between the ODI protocol specification and how it has been implemented in the HMI Engine which caused the development to become very error prone. As an example, if you set a Java byte that is part of an ODI Indication message to the number 1 the indication interpreted as number 8 would be set to true, which might not be what would be logically expected.

7 Conclusion

7.1 Result

The result of this thesis project is a working prototype of a simulator used to test HMIs. The simulator allows a user to create statecharts representing the workings of any FU defined by an FU class and to do this without writing a single line of code. Once created and configured the progress of running simulations is easily monitored through a visual feedback mechanism. The simulator connects to an HMI through a standalone communication interface which ensures that an HMI cannot distinguish a simulated FU from a real one.
The simulator fulfils all the listed shall-requirements except for one (S2) which was considered to be of little importance to the overall functionality of the system.

7.2 Discussion

Even though the thesis work has generated a runnable system, it is important to understand that it is still not complete. The time limit of the project had to be taken into consideration when it was decided how much of the implementation was actually possible to accomplish. It was obvious from the start that it was not possible to create the whole fully functional product fulfilling all the shall- and should requirements of the requirements specification. Instead, the aim has been to produce a working and well documented prototype that can be further developed should so be required. As a result of this, some requirements have been down prioritized and left for implementation later on.

The Renegade simulator can only be run as an Eclipse plugin application. While this is acceptable as long as the system is still under development, this is no longer the case if the simulator is going to be marketed to a potential customer. Not only does this solution make the system too complicated and time consuming to use in the long run, it is also not an option to require that the customer acquires and learns Eclipse in addition to the new simulator system.

Today, the system can indeed handle simulations of functional units, but so far it is only made for very simple applications with few functions and a limited amount of data. It is possible to create a simulation of a very simple music player that can handle functions like play, stop and the displaying of the current track number, but there is for instance still no support for displaying a running clock to show the track time. A complicated FU like a GPS would most probably be impossible to create with the current system. The simulator has to be extended with more, and more advanced, functions before it can be said to fulfil all the needs that a user might have.

The original idea of the thesis work was that the system would be even further developed than it is right now, but this turned out to be impossible in the end. The reason for this was that a lot of time that was originally meant to be spent on the implementation of the Renegade Simulator instead had to be spent on finishing and improving already existing components in Mecel Populus. One improvement that was made was the one of the ODI logger used to see what was sent and received through the ODI protocol. To be able to use this effectively in the debugging process it had to be changed and upgraded to show more detailed information. Also, the Populus HMI Engine had to be rebuilt several times since there were a lot of difficulties integrating the Renegade system with the Populus system, mainly because of the ODI specification being interpreted differently in the two projects.

As the implementation phase had to be extended in order to complete a working prototype the user testing had to be down prioritized. Despite this a few user test were still performed in order to try to define what should be done if the development were to continute. Because there was not enough time to find actual customers to perform the test scenarios these had to be done by other people. Although it might not be considered an optimal approach effort has still been made to select testers that fulfilled the user demands as closely as possible.

One purpose of the thesis work was to create an easy way for non-programmers to test HMIs. Unfortunately since there have not been any testing on actual customers, there is no possibility to clearly state whether this goal has been reached or not. However, the system does not require a user to write any code by hand but mostly gets the input from pointing and clicking. The only manual text input that has to be fed to the simulator is that of data values. When asking the testers and other people (both programmers and non-programmers) it was also the main opinion that a graphical image was indeed easier to grasp for a mixed group of employees than written lines of code. Because of this, even though it cannot be said that the present system is a perfectly satisfactory solution to that problem, the conclusion is that the Renegade simulator is a solution that is at least well on the way towards that goal.

7.3 Future work

If the system is to be marketed and sold, there are still some important things that need to be done before it can be considered a finished product. This chapter does not aim to give any final solutions to the remaining problems, but to point out components and functions that have yet to be developed and in some cases give a few suggestions.

7.3.1 General

In the finished system, the simulator can no longer be a part of Eclipse. As the simulator does not depend directly on any of the functions in the other Populus products it would be possible to have it as a standalone application separated from both Eclipse and Populus. Another alternative would of be to actually integrate the simulator in the current Populus suite and although both solutions are technically possible, this latter one would probably be considered the most user friendly.

7.3.2 Renegade Editor

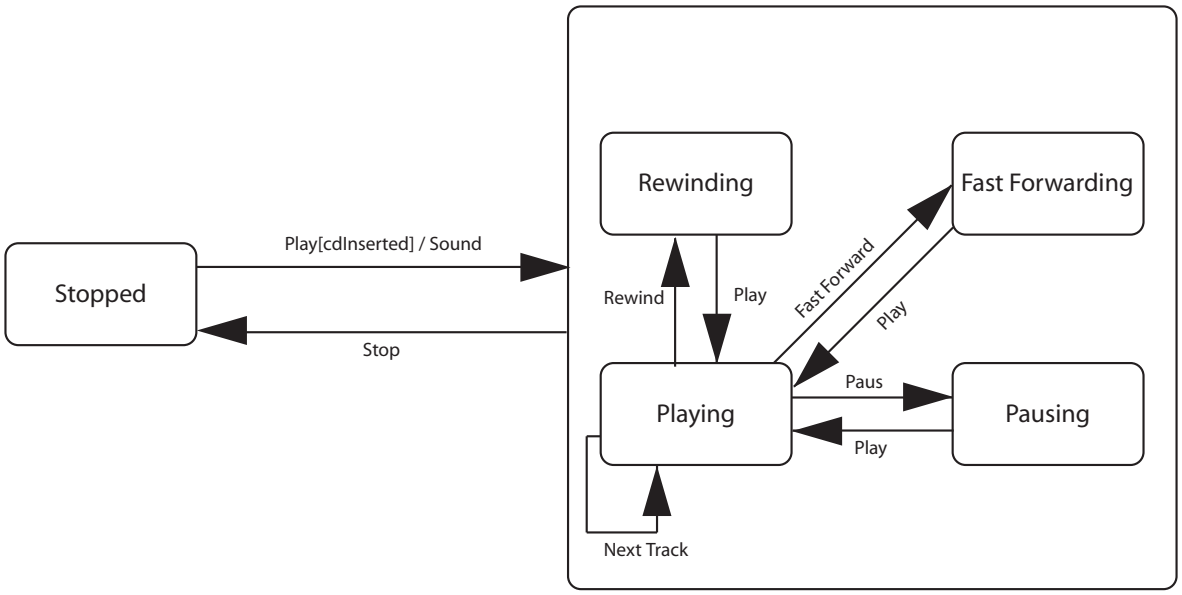


Figure 21: An example of an extended statechart, showing nested states

Simple non-nested diagrams can be created in the Renegade Editor, but in order to simulate more complex functional units without having to create very large statecharts, the editor should be extended to enable the user to draw nested statecharts. This would allow a user to place several substates (or sometimes several statecharts) inside a larger superstate. This could be proven useful, for instance when an device has an active and an inactive state, and behaves differently in these two superstates. The Commons SCXML Engine already provides support for this type of statechart, meaning that only a minor changes to the Renegade system would be neccessary.

The system should include a debugging function, allowing the user to place breakpoints at arbitrary positions in the statechart. When the simulation runs in debug mode it should be suspended whenever one of these breakpoints is reached, giving the user time to examine logs and data to assure that everything about the simulation and HMI testing works as planned.

With the introduction of a debugging function that allows a user to suspend a simulation it becomes more valuable to improve the visual feedback to also show the transitions that are taken. In its current form the simulator never lingers long enough on a transition to warrant highlighting it but once breakpoints are available the simulation may come to a halt while on a transition. Without highlighted transitions the exact state of the simulation will be unknown.

In the Renegade Editor it is possible to have several diagrams open but only one at a time can be active and in use. During the testing it was revealed that the system contains a bug which causes the visual feedback from the simulation controller to be sent to whatever diagram is currently active. If the user starts the simulation with one diagram and while running the simulation switches to another open diagram that contains all or some of same states this new diagram will be updated when the statemachine reaches a new state. This could definitely lead to confusion end errors and should be corrected in a future version of the system.

The Renegade Editor can only handle indications defined in an FU class and these indications are for the most part used to control visual aspects of an HMI. As a future improvement a user should be able to create new indications in the editor which will only serve as guard expressions in the statechart and thus never sent to the the HMI.

When the user wants to stop an active simulation he or she has to exit the Eclipse application completely and before another simulation can be started a new instance of the application has to be run. This is time consuming and would also not work when the system is moved out of Eclipse since this would most likely cause the entire program to have to be restarted. A stopping function has to be implemented and made available in the GUI.

7.3.3 ODI Interface

The ODI specification contains 22 different types of data that can be sent using the ODI protocol and the developed simulator supports nine of these. The ones that are supported are all single frame data types, with the exception of the string type, while an FU class may contain multi-frame data types such as lists. The remaining 13 must be implemented along with support for list data before the system covers the full ODI specification.

In the ODI specification there are two types of actions messages: simple actions and value actions. Both types contain a regular action message but in addition to this the value action also contains a data value which can be of any of the different types of dynamic data. The current implementation of the system handles all the ODI Action messages in the same way and does not distinguish between simple and value actions. In a future development phase this should be changed so that the data in a value action can be saved and used during the simulation.

In order for an HMI to receive notifications of data changes in the FU it must be registered as a subscriber to those data values it has an interest in. This subscription mechanism has been implemented in Renegade but due to a known problem in Populus it does not work fully. The problem is that the subscription requests are sent to the FU, in this case the simulator, upon startup which occur before the connection to the FU has been established and then again each time an event has been sent from the FU. As a result of this issue the HMI may miss several updates to data because as long as no event has been triggered the FU does not know that the HMI wants to subscribe to data changes in the system. In order for the simulation, or rather the complete system, to work as expected this problem needs to be fixed. Mecel are discussing possible solutions to the problem but as of yet none of them has been realized.

A future extension to the subscription mechanism is to allow several HMIs to subscribe to the same set of data in the same FU. In its current implementation the system only allows for one subscription per data value. As multiple HMIs may be driven by the same FU a natural way forward is to bring this multi-display support into the subscription mechanism.

When using the basic implementation of the Commons SCXML Engine it becomes rather cumbersome to handle changes to data values in such a way as was required for the simulator. The solution became a bit messy and should be reworked in future implementations, possibly using the SCXML Engine custom actions mechanism [5].

References

[1] Apache Software Foundation, Apache Commons, March 2009.
http://commons.apache.org/

[2] Apache Software Foundation, Apache log4j, August 2007
http://logging.apache.org/log4j/1.2/index.html

[3] Apache Software Foundation, Commons EL, March 2008
http://commons.apache.org/el/

[4] Apache Software Foundation, Commons JEXL, March 2008
http://commons.apache.org/jexl/

[5] Apache Software Foundation, Commons SCXML, May 2009.
http://commons.apache.org/scxml/

[6] Apache Software Foundation, Commons SCXML Usage - Five minute SCXML Tutorial, May 2009.
http://commons.apache.org/scxml/guide/scxml-documents.html

[7] A. Burns, Concurrent Programming, Addison-Wesley, 1993, ISBN: 0-201-54417-2.

[8] L. Copeland, State-Transition Diagrams, 2008.
http://www.stickyminds.com/sitewide.asp?Function=edetail&ObjectType=ART&ObjectId=6232

[9] A. van Deursen, P. Klint and J. Visser, Domain-Specific Languages. ACM SIGPLAN Notices, Vol. 35, No. 6, pp. 26-36, June 2000.

[10] Domain-specific language.
http://en.wikipedia.org/wiki/Domain_Specific_Language

[11] Eclipse Foundation, Eclipse.org home, 2009.
http://www.eclipse.org/

[12] Eclipse Foundation, Graphical Modeling Framework, 2009
http://www.eclipse.org/modeling/gmf/

[13] Eclipsepedia, GMF Tutorial, August 2008.
http://wiki.eclipse.org/index.php/GMF_Tutorial

[14] Eclipsepedia, Graphical Modeling Framework FAQ, August 2008.
http://wiki.eclipse.org/Graphical_Modeling_Framework_FAQ

[15] H-E. Eriksson, M. Penker, B. Lyons and D. Fado, UML 2 Toolkit. John Wiley & Sons, 2003, ISBN: 0-471-46361-2.

[16] eVelopers Corporation, Java Finite State Machine Framework, January 2007.
http://unimod.sourceforge.net/fsm-framework.html

[17] eVelopers Corporation, UniMod, June 2008
http://unimod.sourceforge.net/

[18] eVelopers Corporation, UniMod 1.3 Introduction, February 2008.
http://unimod.sourceforge.net/wiki/index.php/UniMod_1.3_Introduction

[19] M. Fowler, Using Domain Specific Languages. April 2008.
http://martinfowler.com/dslwip/UsingDsls.html

[20] Free Software Foundation, GNU Lesser General Public License, June 2007.
http://www.gnu.org/copyleft/lesser.html

[21] J.P. van Gigch, System design modeling and metamodeling. Plenum Publishing Corp, 1991, ISBN: 0-306-43740-6.

[22] R. Gronback, Eclipse Modeling Project - a domain-specific language toolkit. Addison Wesley, 2008, ISBN: 0-321-53407-7.

[23] J. Hanneman and G. Kiczales, Design pattern implementation in Java and Aspect J. ACM SIGPLAN Notices, Vol. 37, No. 11, p 161-173, 2002.

[24] P. Hudak, Modular Domain Specific Languages and Tools. Software Reuse, pp. 134-142, June 1998.

[25] IBM Rational Software Architect.
http://en.wikipedia.org/wiki/Rational_Software_Architect

[26] IBM Rational Software Architect - Summary.
http://www.componentsource.com/products/ibm-rational-software-architect/summary.html

[27] IBM, IBM Modeling and Integration Tools for State Chart XML, March 2007.
http://www.alphaworks.ibm.com/tech/scxml/?open&S_TACT=105AGX59&S_CMP=GR&ca=dgr-lnxw02aawscxml

[28] IBM, Learn Eclipse GMF in 15 minutes, September 2006.
http://www.ibm.com/developerworks/opensource/library/os-ecl-gmf/

[29] IBM, IBM Rational ClearCase - Software, 2008
http://www-01.ibm.com/software/awdtools/clearcase/

[30] IBM, Rational Software Architect, 2008.
http://www-01.ibm.com/software/awdtools/architect/swarchitect/

[31] Java Community Process, JSR-00152 JavaServer Pages 2.0 Specification - Final Release, November 2003
http://jcp.org/aboutJava/communityprocess/final/jsr152/

[32] F. Kronlid and T. Lager, Synergy SCXML Web Laboratory, May 2007.
http://www.ling.gu.se/~lager/Labs/SCXML-Lab/

[33] C. Larman, Applying UML and patterns: An introduction to object-oriented analysis and design and the unified process. Prentice Hall Inc, 2001, ISBN: 0-13-092569-1.

[34] Mecel AB, A solution for efficient HMI Development and Deployment. 2009.
http://www.mecel.se/products/mecel-populus

[35] Mecel AB, At the forefront of automotive technology. 2009.
http://www.mecel.se/

[36] Mecel AB, Mecel Betula Suite - Automotive Bluetooth® Platform. 2009.
http://www.mecel.se/products/bluetooth

[37] Mecel AB, Mecel Picea. 2009.
http://www.mecel.se/products/mecel-picea

[38] Mecel AB, Product Brief - Mecel Populus Suite. 2008.
http://www.mecel.se/products/mecel-populus/Product-Brief-Mecel-Populus-Rev1-08.pdf

[39] M. Mernik, J. Heering and A.M. Sloane, When and How to Develop Domain-Specific Languages. ACM Computing Surveys, Vol. 37, No. 4, pp. 316–344, December 2005.

[40] metamodel.com - Community site for meta-modeling and semantic modeling.
http://www.metamodel.com/staticpage/index.php?page=20021010231056977

[41] R. Miller, Practical UML: A Hands-On Introduction for Developers, December 2003.
http://edn.embarcadero.com/article/31863

[42] M. Moran, Notification Pattern.
http://mnmoran.org/hypothesis/notificationPattern.html

[43] J. Richley, GMF: Beyond the Wizards, November 2007.
http://www.onjava.com/pub/a/onjava/2007/07/11/gmf-beyond-the-wizards.html

[44] Sparx Systems, UML 2 Tutorial - State Machine Diagram, 2009.
http://www.sparxsystems.com.au/resources/uml2_tutoria/uml2_statediagram.html

[45] V. Subramaniam, Creating DSL:s in Java, Part 1: What is a domain-specific language?, August 2008
http://www.javaworld.com/javaworld/jw-06-2008/jw-06-dsls-in-java-1.html

[46] V. Subramaniam, Creating DSL:s in Java, Part 3: Internal and external DSLs, August 2008.
http://www.javaworld.com/javaworld/jw-08-2008/jw-08-dsls-in-java-3.html

[47] D. Thomas, The ’Language’ in Domain-Specific Language Doesn’t Mean English (or French, or Japanese, or...), March 2008.
http://pragdave.blogs.pragprog.com/pragdave/2008/03/the-language-in.html

[48] W3C, State Chart XML (SCXML): State Machine Notation for Control Abstraction, May 2009.
http://www.w3.org/TR/scxml/

[49] XML Metadata Interchange (XMI)
http://www.service-architecture.com/web-services/articles/xml_metadata_interchange_xmi.html

Appendix A

Requirements Specification

Requirements specification
Version 4.0
Editor for application modelling and simulation

Linda Erlenhov
Anna Södling

1. General

The specification concerns an editor to build a simple statechart that simulates an application.

A user should be able to choose which part of the application he/she wants to simulate and use the editor to build a statechart of that specific part. He/she should also be able to send actions and handle incoming events between the states. The specification also includes an interface between the engine and our editor.

2. References

1. Mecel's homepage about the existing editor, Mecel Populus - http://www.mecel.se/products/mecel-populus
2. "Programutveckling i liten skala – en praktisk handbok", Östen Oskarsson
3. The project's project plan

3. Definitions

HMI: Human - Machine Interface

HMIEngine: Mecels product that executes the HMI in runtime and communicates with the applications using the Open Display Interface protocol.

Simulator: The complete system, consisting of a statechart editor with an internal engine and an interface between the editor and the HMIEngine.

Populus: Mecels existing editor used for creating an HMI without having to write any code.

Action: Activity that indicates something being sent to the application, for example a press on a Play-button or a change of volume.

Event: Activity that indicates something being sent from the application, for example when a track is changed in a music player or a station is changed on a radio.

Indication: A boolean value to indicate some status of the application, for example whether a music player is playing or in stopped mode.

Dynamic data: The data (strings, integers, etc.) needed to display information about what is going on in the application, for example the title or artist of track. Considered to be dynamic since it might change between states.

ODI: Open Display Interface, the communication protocol between the simulated applications and the HMIEngine.

.fuclass file: A file containing the HMI:s pre-defined actions, events, indications and dynamic data.

4. Technical requirements

4.1 Interfaces

4.1.1 Editor interface

There shall be a graphical interface of the editor used to draw up and manipulate the simulation statechart.

GI1. The drawing of a statechart shall be done using a simple ”point-and-click”-method.

GI2. The editor shall be able to read a .fuclass file from the Populus editor.

GI3. A newly built or modified statechart shall be able to be saved in the editor and re-opened on a later occasion.

GI4. When the editor is started, the user must choose between building a new statechart or opening a previously saved statechart.

GI5. The user shall be able to display a list of the different elements (actions, events, etc.) contained in the .fuclass-file.

4.1.2 Interface between editor and HMIEngine.

There shall be a non-graphical interface between the editor and the HMIEngine.

NGI1. The interface shall communicate with the HMIEngine via an ODI-protocol over TCP/IP.

NGI2. The interface shall be able to handle incoming ODIActions.

NGI3. The interface shall be able to handle the sending of ODIIndications, ODIEvents and ODIDynamicData.

4.2 Functional requirements

F1. The system shall be able to handle simple 32-bit datatypes.

F2. The system should be able to handle more complex datatypes, such as lists and dynamic pictures.

F3. The HMIEngine should be able to be started from the editor.

F4. The simulator shall be able to handle simple, non-nested statecharts.

F5. The simulator should be able to handle more complex, nested statecharts.

F6. The simulation-engine should be stand-alone.

F7. The system shall be able to handle timers.

F8. The system should be able to handle common data variables (not only ODI-variables) , for example counters

F9. The application-simulation shall be able to be run together with the HMIEngine to simulate a complete HMI.

F10. The editor should contain a debugger-function for the statecharts.

F11. It should be possible to place one or more breakpoints in the statechart when using the debugger-function.

F12. When a breakpoint is reach, the simulation should become suspended and the user should then be able to choose whether to continue running or stepping through the simulation.

F13. The user shall only be able to use the elements pre-defined in the .fuclass-file for the transitions and data.

4.3 Relations between functions

S2-S3. If the HMIEngine is not started during a simulation, the simulation shall still run, and continue to try to connect to the engine. If the HMIEngine starts during a simulation, the simulation shall not be affected, but shall start to send/receive information to/from the engine.

GI6-F14. In order to prevent the user from making unnecessary mistakes, it shall only be possible to use the pre-defined elements. The user shall not be allowed to name the elements him/herself in the editor.

F10-F11-F12. If the requirement F10 is not fulfilled, there is no need to try to fulfil neither F11 nor F12, since they are dependent on each other. If F10 is fulfilled, then both F11 and F12 shall be ”shall”-requirements.

4.4 Operational requirements

O1. The system shall be able to run under any operating system that supports Java.

4.5 Capacity requirements

CA1. The system should be able to run up to 5 simulations at the same time on the same computer.

CA2. The system should only be able to debug one simulation at a time.

4.6 Security requirements

S1. Before the simulator is shut down, the user shall be prompted to save his/her created statecharts.

S2. Before a simulation is run, the system shall display a warning if the HMIEngine is not started, but the simulation shall still be able to run.

S3. If the HMIEngine is started during the simulation, the simulation should not be affected.

4.7 Construction requirements

CO1. The interface shall be separated from the editor.

CO2. The simulator shall be able to be built in to the existing Populus editor.

CO3. The HMIEngine shall not be able to tell the difference between a true and a simulated application.

4.8 The users capacity and capability

UC1. A user shall have basic knowledge of computers and be very familiar with the Populus editor.

UC2. After 10 minutes instruction, a user shall be able to operate the system's basic functions.

4.9 Other requirements

No other requirements exist.

5. Other information

Not applicable

Appendix B

Project Plan

Project Plan

Version 2.0

Editor for application modelling and simulation

Linda Erlenhov
Anna Södling

1 Assignment

This project plan concerns the modelling of an editor for statecharts that are used for simulating a simple application (for example a radio or CD-player) using a simulation engine. This together with Mecels existing Populus editor, used for defining an HMI, and HMIEngine, used for running the HMI:s defined in the editor, will be used to simulate a complete HMI. The assignment also includes the task of creating the interface between the engine and our editor.
The orderer of this project is Mecel AB, Mölndalsvägen 36, Gothenburg.

2 References

1. “Programutveckling i liten skala – en praktisk handbok”, Östen Oskarsson
2. The project’s requirements specification

3 Definitions

See ref. 2 (requirements specification)

4 Project management

4.1 Organization

Both project members will function as software developers and testers. There will be no pronounced project manager; both members will have equal responsibility in coming to decisions and planning the project.

4.2 Time plan

Activity	Nov	Dec	Jan	Feb	Mar	Apr
Documentation						
Obtaining knowledge						
Choice of system (s)						
Planning and project managing						
Requirement analysis						
Sysem design and implementation						
Testing						
Report writing						
Other						
Milestone		1		2		3

4.3 Activity descriptions

Documentation

Refers to the documentation of the current work in the project. The activity can be considered to be finished when the final product is delivered.

Obtaining knowledge

The time that the project members use to obtain the knowledge needed to develop the system. The activity can be considered to be finished when we have enough knowledge to be able to begin each relevant part of the project.

Choice of system(s)

The task concerning the decisions of which platform, language and system to use for the project. Should be looked upon as a “bridge” between the obtaining of knowledge and the system design/implementation. The activity can be considered to be finished when the decisions have been made and we have enough knowledge to be able to start the design/implementation phase.

Planning and project managing

Mainly the task to draw up the future activities in the project. The activity should be considered to be finished when the final product is delivered.

Requirement analysis

The task to define the system that should be delivered. Contains the developing of a requirement specification. The activity can be considered to be finished when the requirement specification is approved.

System design and implementation

Refers to the developing of the software. The activity can be considered to be finished when the system meets the requirement specification.

Testing

Means to guarantee the functionality of the system design and implementation. The activity can be considered to be finished when the system meets the requirement specification.

Report writing

The final compilation of the results and the design and implementation of the system. The activity can be considered to be finished when the final report has been turned in.

Other

Meetings, team building, etc. Anything that does not fit in under any of the other subjects.

Milestones

The milestones that are defined in section 6 Progress verifications.

4.4 Progress reports

The project group should have a meeting with the supervisor at least once a week. The group should also meet with its examiner as often as is required.

Each project member should also keep a diary to record what has been done each workday.

5 Software development

5.1 Standards and procedures

5.1.1 Coding

All code and comments should be written in English. The comments should be written according to the standard of each programming language. If there are several standards, the program developers decide together which one to use.

5.1.2 Final report

The final report should be written in English, using Microsoft Word or a Word-compatible program. The final version of the report will be put together in InDesign and should be presented in both in pdf-version and a printed version.

5.2 Existing software

- The program will be developed in Eclipse, using GMF.
- If possible, we should use applicable open source-software.
- The program has to be compatible with Mecel's existing editor and engine.
- ClearCase will be used as the project's revision control system.

6 Progress verifications

During the project, the progress in the development process should be reported to the supervisor continuously. Verification of any written code can be done either between project members and supervisor or between the both project members.

We have three milestones to verify the progress of the project.

The first one is on the 19th December 2008 to sum up the progress before the winter holiday. The requirement specification and choice of systems should be finished and the software development should have begun.

The second one is on the 1st March 2009. We should be well on our way with the software development. This milestone is only used as a checkpoint to make sure that everything is moving in the right direction. We should also be able to say what optional requirements the final system will fulfil.

The last one is on the 30th April 2009. Both the system and the final report should be ready and approved to be delivered.

During the progress of the project we might also put up some internal, smaller, deadlines.

7 Document management

All code documents should be kept in the ClearCase revision control system. All other documentation (i. e. text documents, pictures and so on) should be kept in a common mailbox or something similar to that.

8 Time of delivery

The final product should be presented on the 30th April 2009 along with the final report and all other documents.

Appendix C

Test Specification and Results

Test Specification and Results

C1. Test Specification

C1.1 Creating a diagram

Create a new, empty diagram.
Name the diagram file ”mydiagram.renegade_diagram”.
Name the domain model file ”mydigram.renegade”.

C1.2 Load an FU class

Load the FU class ”TestFU.fuclass” into the diagram.
Save the diagram.

C1.3 Basic configuration

Place three ordinary states on the canvas.
Configure each state, naming them State A, State B and State C, respectively.

Place a transition from State A to State B, and place a transition from State B to State A. Configure the transitions to be ordinary transitions with the actions ToB and ToA, respectively.

Place a transition from State A to State C, and place a transition from State C to State A. Configure the transitions to be ordinary transitions with the actions ToC and ToA, respectively.

Place a transition from State B to State C, and place a transition from State C to State B. Configure the transitions to be ordinary transitions with the actions ToC and ToB, respectively.

Place an initial state an a final state on the canvas.

Place a transition from the initial state to State A.
Configure the transition to have a timer set to 10ms.

Place a transition from State B to the final state.
Configure the transition to be an ordinary transition with the action ToEnd.

Save the diagram.

C1.4 Advanced configuration

Configure the transitions between State A and State C to have the event Regular Event. Configure the transition from State B to State A to have Indication2 set to ”true”. Configure the transition to the final state to have Indication1 and Indication2 both set to ”true”, and to have the event Regular Event.

Add a transition from State B to itself.
Configure it to have the action ToB, Indication2 set to ”false” and the event Regular Event.

Add a transition from State C to itself.
Configure it to have a timer set to 5000ms, and the event Timer Taken.

Configure State A to have Indication1 set to ”true”, NumDat set to decrease the value, and TextDat set to ”State A”. Configure State B to have TextDat set to ”State B”. Configure State C to have Indication2 set to ”true”, NumDat set to increase the value, and TextDat set to ”State C”.

Save the diagram.

C1.5 Running the simulation

Start the appropriate HMI.
Start the TCP Proxy.
Run the simulation from the Renegade menu.

Try to go to State B.
Try to go to State A.
Try to go to State B.
Try to go to State B again.
Try to go to the final state.
Try to go to State C and wait there for about a minute.
Try to go to State A.
Try to go to State B.
Try to go to State B again.
Try to go to State A.
Try to go to State B.
Try to go to the final state.

C2. Test Results

The testing has been performed on a small group of people, all with basic computer knowledge. They were all given a brief introduction to the Renegade system and were also presented with the user manual, before they were asked to go through the test scenario specified in C1. The following is a compilation of the comments and critique that the users gave during the testing phase.

C2.1 Creating a diagram

It was not considered very intuitive that you had to start a new project before creating your first diagram, neither that you had to choose to create an ”example” from the menu when starting a new diagram. If you want to build a diagram for simulation in your everyday work this should not be considered an example, but rather a simulation, or simply a diagram, or something similar to that.

C2.2 Load an FU class

This part was the most simple to go through during the tests. Here we had no possibilities of confusing the user, the loading of the FU class was stated to be simple and convenient since there really was only one choice that could be made. There were some issues whether it was too hard to find the actual FU class on the computer, but that was stated to be more of a responsibility of the user rather than the system.

C2.3 Basic configuration

If the palette was not shown from the start (it could be minimized) it might be hard to find for an inexperienced user.

While doing the configuration with the wizards it was not considered to be obvious that you could click ”Finish” without going through all the steps that the wizard provided. This could cause confusion and insecurity in the user if he/she has to run through more steps than what is necessary for the desired configuration.

The texts that marks out the possible actions, indications and events run a risk to become too muddled, depending on how the transition is drawn. If two transitions are drawn close to each other there is also a risk that the configurations for each transition might be mixed up. However, it was considered a good thing that each text was bound to its own transition with a broken line. This made it easy for the user to move the text around to the spot he/she wanted the it to be in for that particular diagram.

C2.4 Advanced configuration

The wizard page where you choose the indications for the transitions is too difficult too understand. Although the first join box before the first indication is blanked out it should be removed entirely since it is never used and at the moment only serves as a confusing element to the user. If a user tries to choose more than one indication.

he/she should be warned if trying to place these indications if there are any joins missing between them. Right now, it is possible to omit the joins, creating errors in the SCXML file which causes the simulation to be incorrect.

C2.5 Running the simulation

The user always had to be sure to have the correct diagram visible in the editor when running the simulation, because otherwise, the simulation tried to run whatever diagram was active at that moment.

There was some irritation with not being able to stop the simulation using the stop button in the menu, since it was not implemented. The simulation had to be stopped by exiting the application and then it had to be started again for a new simulation to be run, which was not considered to be neither obvious nor very convenient, since the start-up was rather slow on the computer used for the testing.

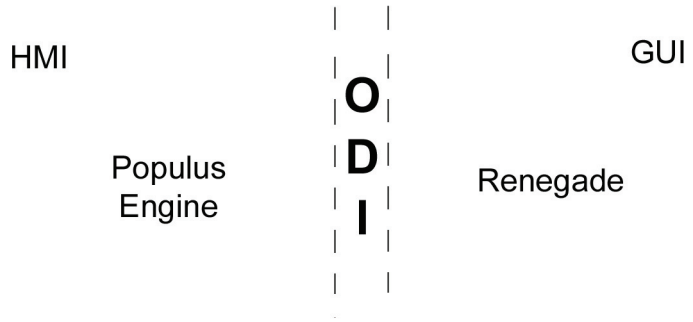
It would have been nice to get an explanation, either by log output or by a text message, when a transition can not be taken. The user might not always remember what indication value is set to what boolean value at a certain time. This lead to the next critique; it should be possible to see all the indication values from the beginning instead of them only being shown when they are updated. It might be too messy to show them in the diagram, but it could be a good idea to show them in the HMI, at least during the testing phase.

Appendix D

System Architecture

D1. Overall system architecture

Once the simulator, consisting of the SCXML based statemachine and a GUI, has been connected to the HMI the complete system can be seen as three distinct part where the Populus HMI and engine is to be considered a black box. This black box is connected to the simulator through the Open Display Interface that passes information from it to the simulator and back again. The simulator is a statemachine based on the generic statemachine implementation provided by the Apache Commons SCXML Engine project.



D2. Renegade system architecture

The Renegade simulator is a GMF based Eclipse plugin with additional functionality to fulfill the requirements of the simulator. Many of the classes are generated by GMF and as such are not really relevant for this system review. More information on these can be found in the GMF documentation. The additional classes can be grouped into the components shown in the figure below which will be explained in more details in the following sections. The two exceptions are the GUI, which is made up of Eclipse and the GMF developed editor, and the SCXML Engine. More detailed information about these specific components can be found in their respective documentation.

Controller

The GUI supplies the user with a number of buttons and menu item that can be used to not only configure the diagram components but also to run the actual test simulation. When the user starts the simulation it is the Controller that is invoked. The Controller is responsible for getting the simulation going and to update the GUI to reflect the current state of the statemachine. In short, the Controller is what connects the statechart created and configured by the user to the rest of the simulator.

Simulator Initiator

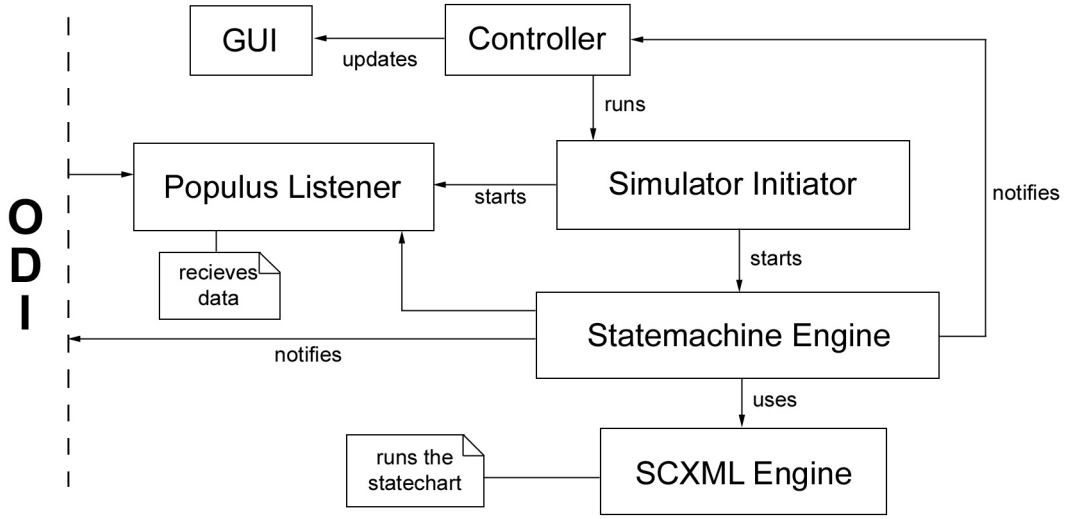
Whereas the Controller is used to start the simulation from the GUI the Simulator Initiator is what actually initiates and starts the different components that together compose the full statechart driven simulation. Once all the components have been stated and connected they will work together without the involvement of the Simulator Initiator. It is, as the name clearly states, only an initiator for the simulation.

Populus Listener

The Populus Listener is run in a separate thread and is the component that is responsible to receiving information from Populus and hand this off to the Renegade Statemachine so that it can take the appropriate action.

Renegade Statemachine

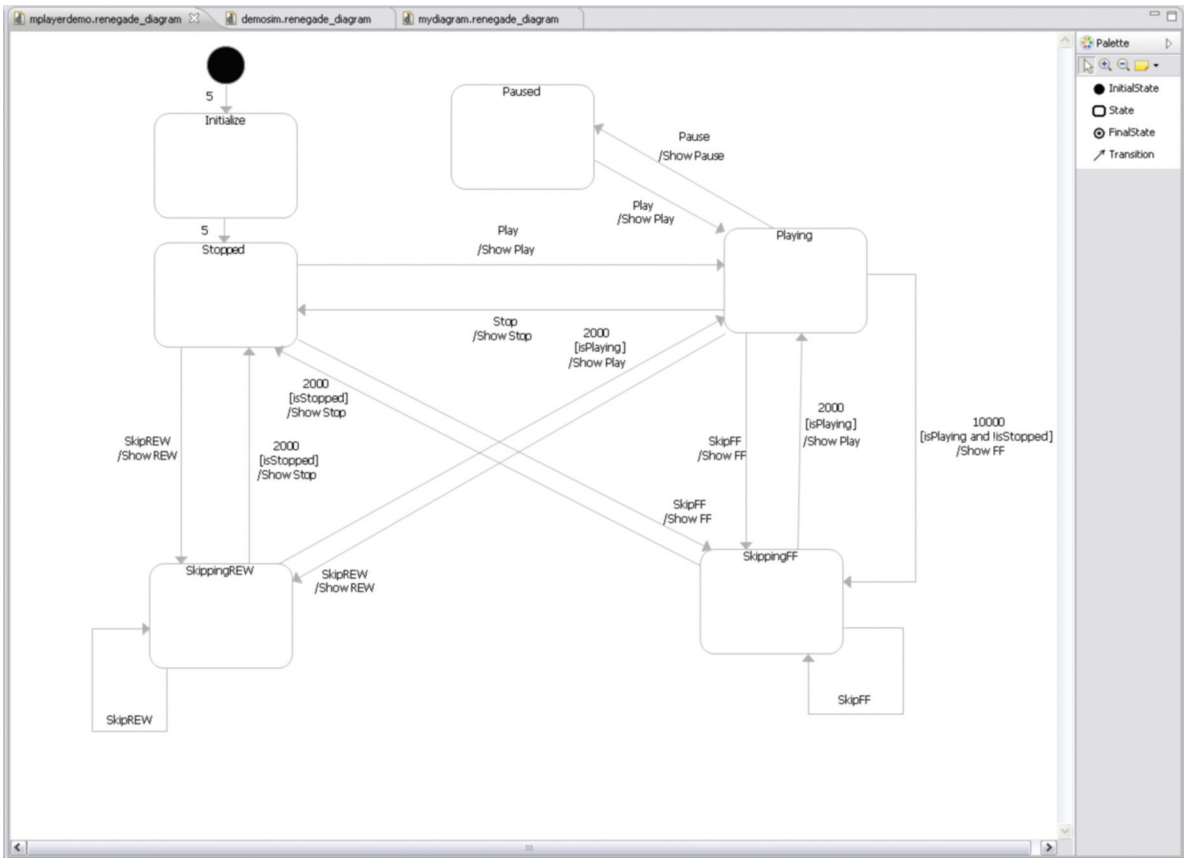
The Renegade Statemachine uses the SCXML Engine to drive the statechart, and the simulation, forward. For each state in which data is updated and for each transition that triggers an event the statemachine will send this information through the ODI to the Populus Engine. This may or may not be displayed in the HMI depending on how this has been created. The statemachine will also be called by the Populus Listener as a response to user interaction with the HMI. These interaction may or may not cause the statemachine to move from its current state to some other state connected to the current one. Whether or not the transition is taken or not depends on the guard conditions defined for the transition. If the transition is taken the statemachine will call the Controller which in turn will highlight the target state in the GUI.



Appendix E

Renegade User Manual

User manual
Renegade Simulator



Contents

1. Getting started

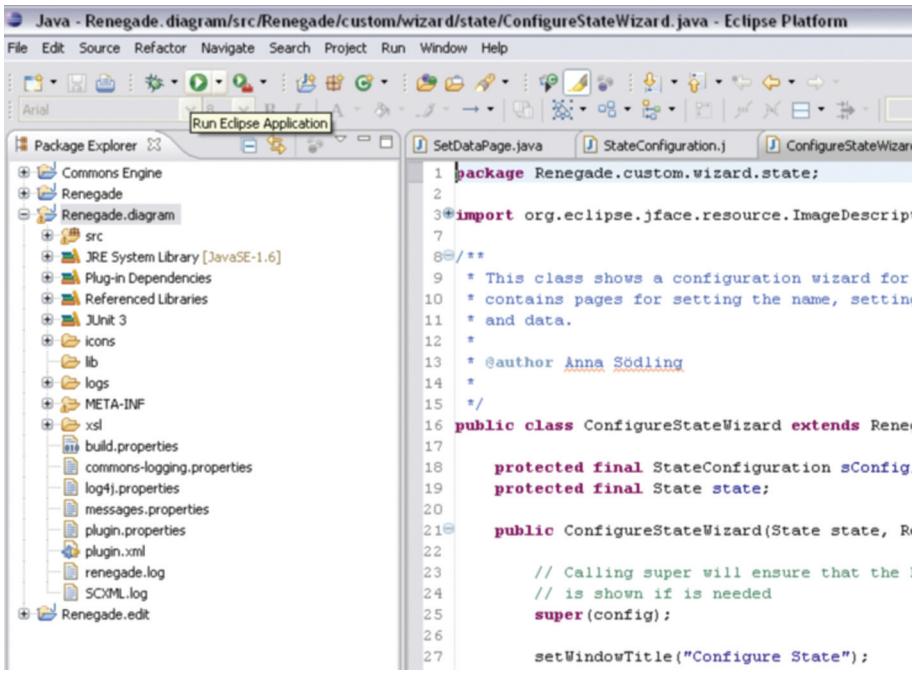
This section explains the basics of how to start using the Renegade Simulator.

1.1 Starting the application

1. Start your copy of Eclipse.

2. Make sure you have the Renegade project marked in your workspace.

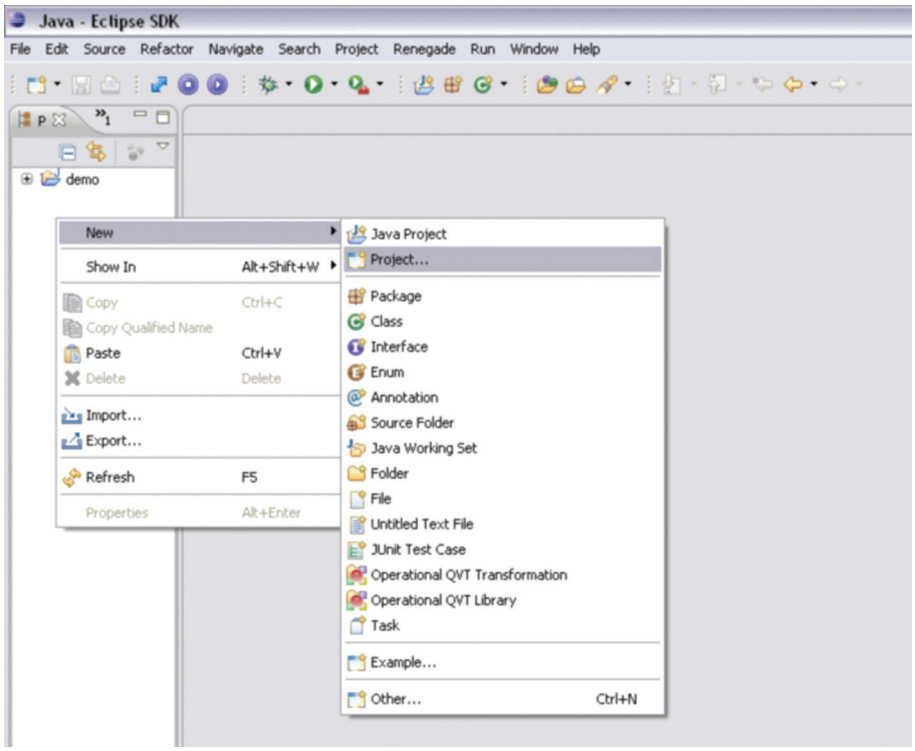
3. Select ”Run Eclipse Application” from the menu.



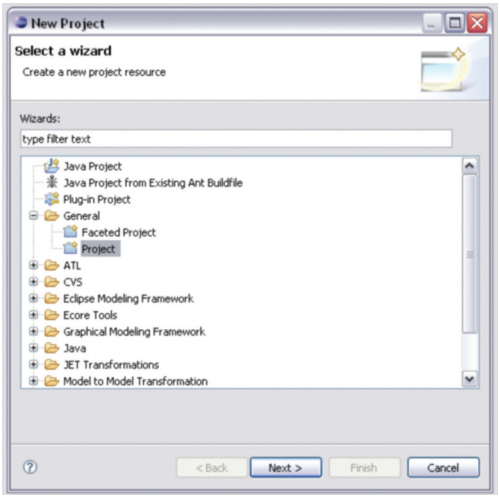
1.2 Starting a new diagram

If this is your first time using the Renegade Simulator, start with step 1. If you already have a project folder and would like to create another diagram in it, start with step 3.

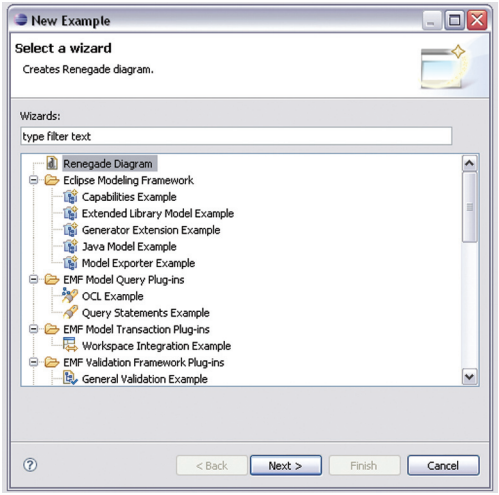
1. In your Eclipse Application, right-click in your workspace and select ”New...” -> ”Project...”.



2. Go to the folder ”General” -> ”Project”. Name your project and click ”Finish”.



3. Right-click your project folder. Select ”New...” -> ”Example...” -> ”Renegade Diagram”.



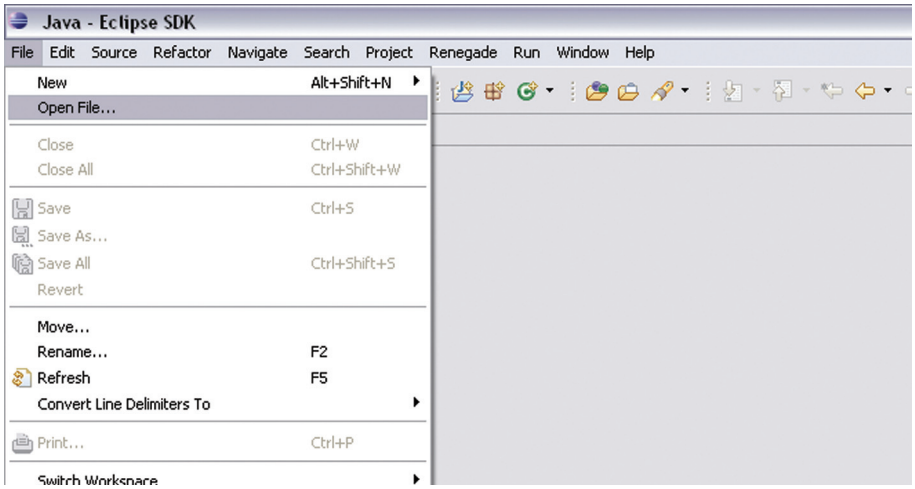
4. Make sure you have the correct parent folder and name your diagram. Press ”Next” to proceed to the next step.

5. Name your domain model. This should be the same as the diagram name. Press ”Finish”.
NOTE! Be careful when you choose the names for diagram and domain model as these cannot be changed later on without causing errors in your diagram.

1.3 Opening a diagram

If you do not want to create an entirely new diagram, you can open a previously created one.

1. Go to the menu and select ”File” -> ”Open File...”.

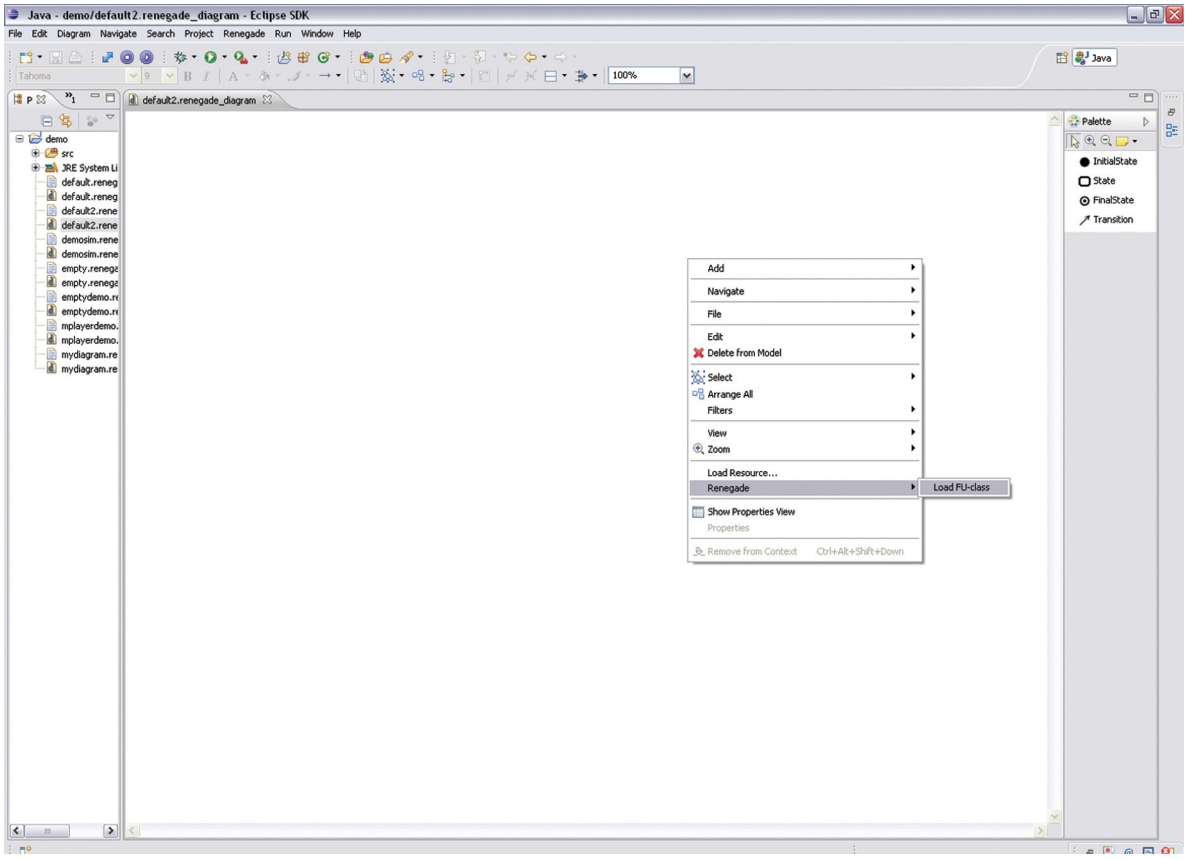


2. Locate the project folder on your computer.

3. Select the desired diagram file (ending with ”.renegade.diagram”). Double-click on the file or press ”Open”.

1.4 Load an FU class

1. Right-click anywhere on the canvas and select ”Renegade” -> ”Load FU-class”.



2. Locate the FU class that you want to use for your simulation on your computer. Double-click on the file or press ”Open”.
NOTE! Before you proceed with creating your diagram, you have to save the diagram in order for it to connect with the FU class.

1.5 Saving a diagram

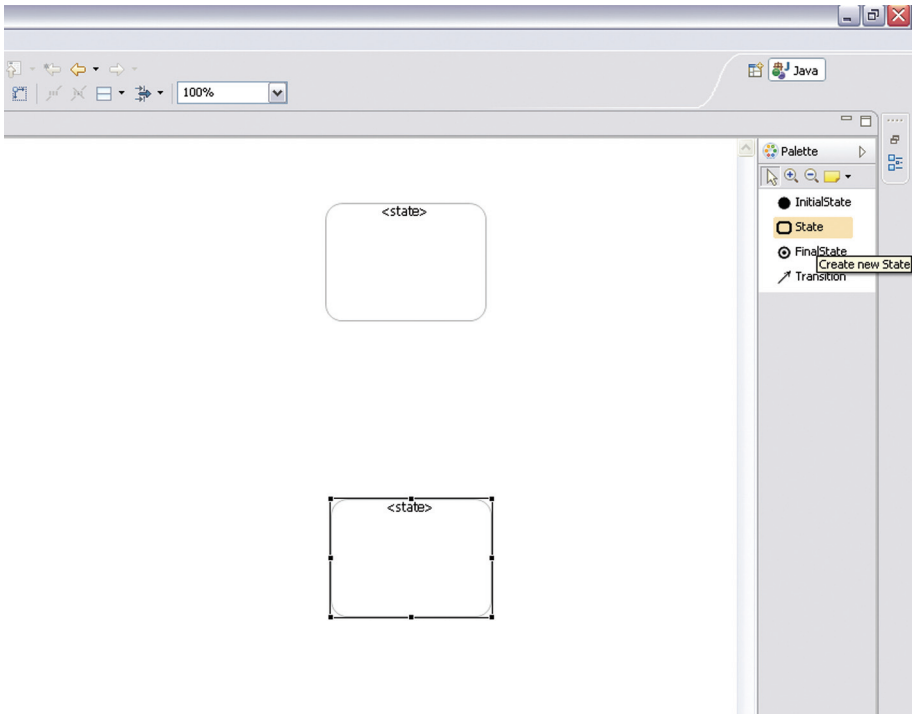
1. Go to the menu and select ”File” -> Save”, or press Ctrl + S.2. Creating a diagram

This section contains an explanation of how to draw the statechart diagram on the canvas and how to configure it properly to represent the desired functional unit.

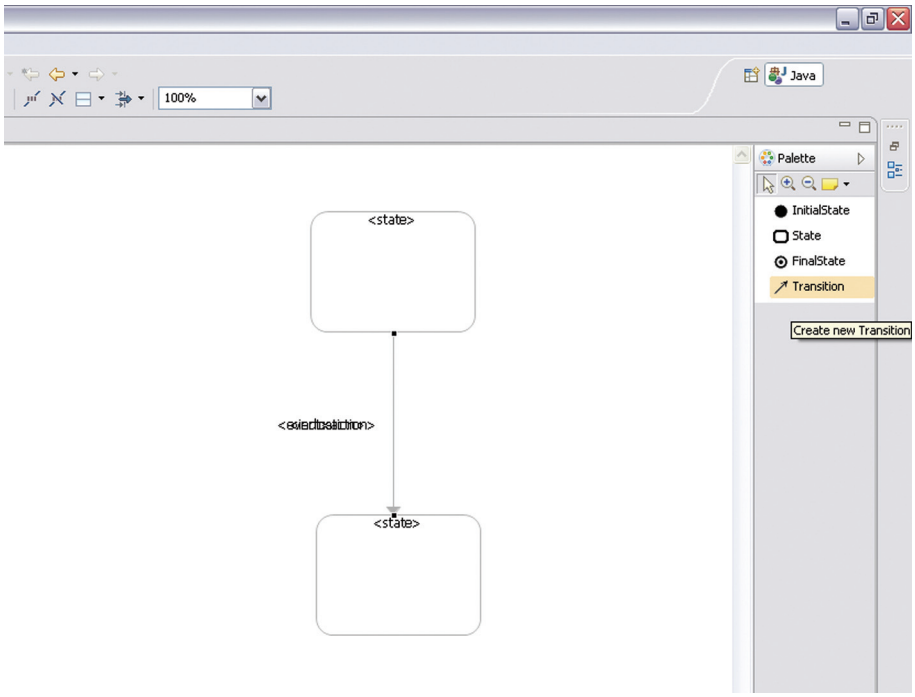
2.1 Drawing the statechart

1. Locate the palette to the left of the canvas.

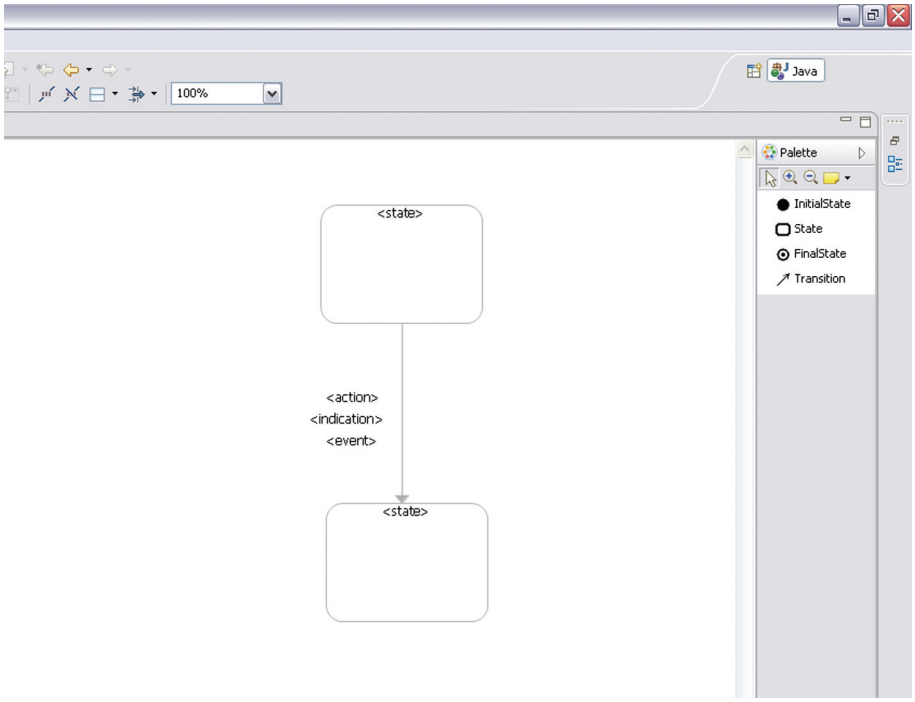
2. To draw a state: choose which type of state you want to draw from the palette and click on the appropriate place on the canvas.



3. To draw a transition: choose the transition from the palette, click and hold the left mouse button on the transitions starting state, drag it to its stopping state and finish by releasing the mouse button.

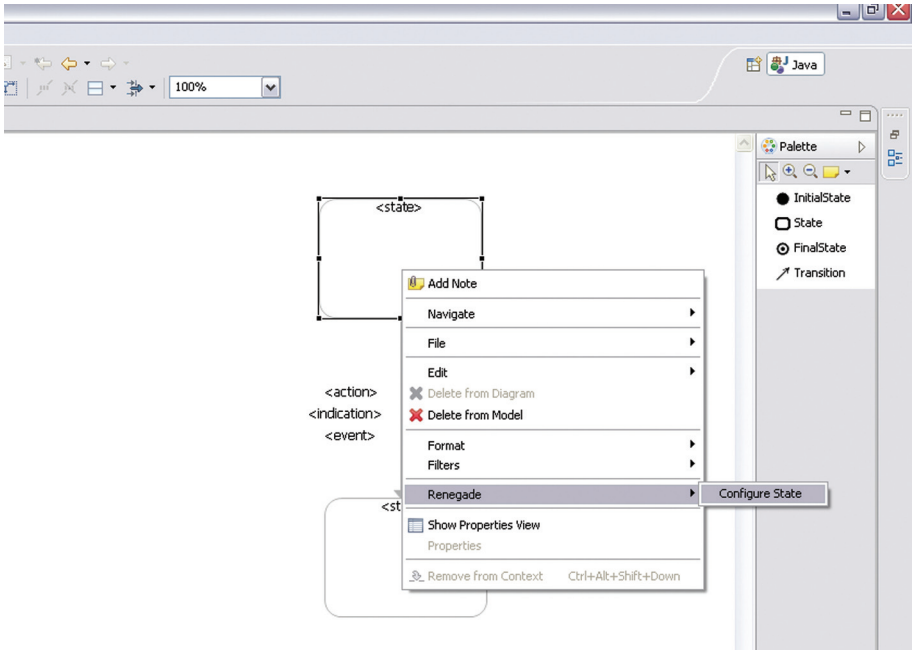


4. If needed, rearrange the markings for actions, indications and events to their desired positions. *NOTE! You can always change the shape of a transition by left-clicking anywhere on the arrow and drag it into its new position.*

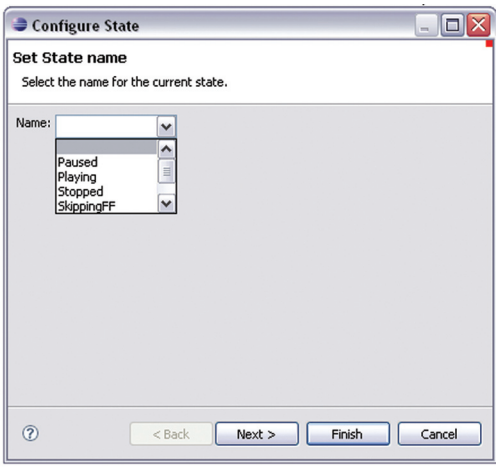


2.2 Configuring a state

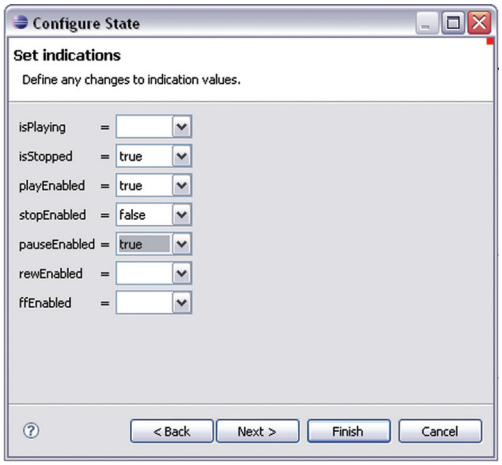
1. Right-click on the state that you want to configure. Select ”Renegade” -> ”Configure State”.



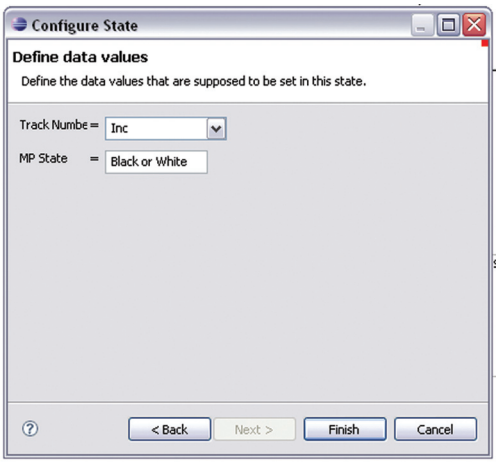
2. Select the appropriate name for the current state.
Press ”Next” to proceed to the next step.
Press ”Finish” to save and end the configuration of this state.
Press ”Cancel” to abort the configuration of this state and discard any choices that has been made.



3. Set any indication values that you want to update in this state (the initial value for each indication is ”false”). If you do not want to change a certain indication value, leave the field blank.
Press ”Back” to return to the previous step.
Press ”Next” to proceed to the next step.
Press ”Finish” to save and end the configuration of this state.
Press ”Cancel” to abort the configuration of this state and discard any choices that has been made.

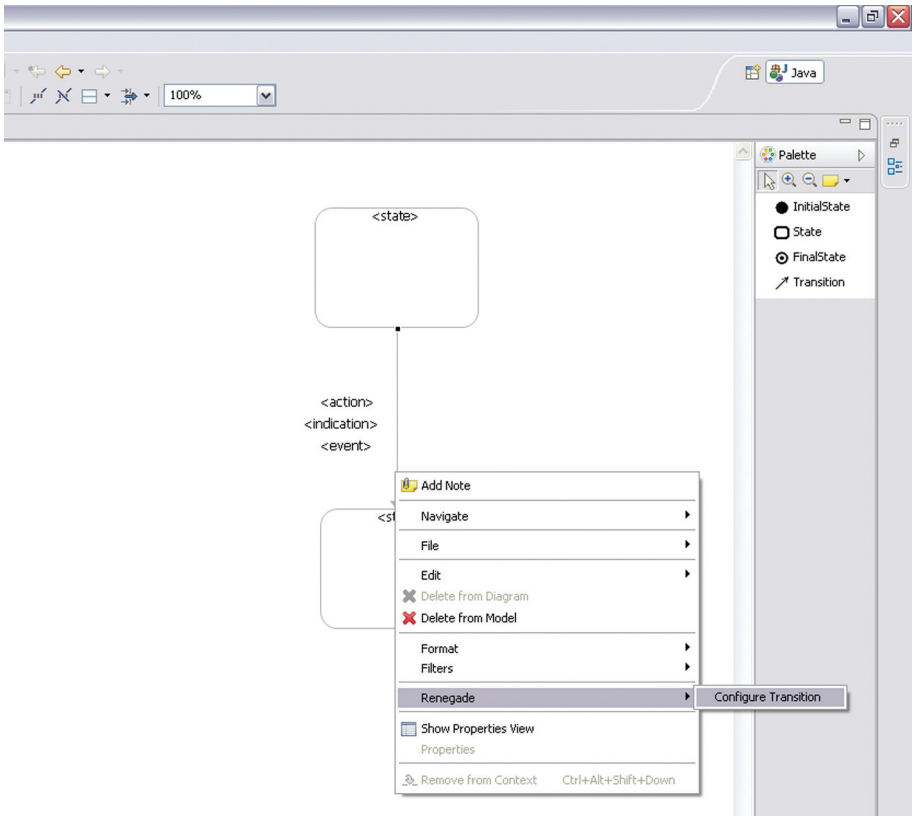


4. Set any data values that you want to update in this state. Numerical values can be increased or decreased by 1, or set to a specific value. Textual values can be set to any string. Time and date values can either be entered manually to any value or set to today’s date/the current time by pressing the ”Now”-button.
Press ”Back” to return to the previous step.
Press ”Finish” to save and end the configuration of this state.
Press ”Cancel” to abort the configuration of this state and discard any choices that has been made.
NOTE! Initial and Final states are non-configurable.

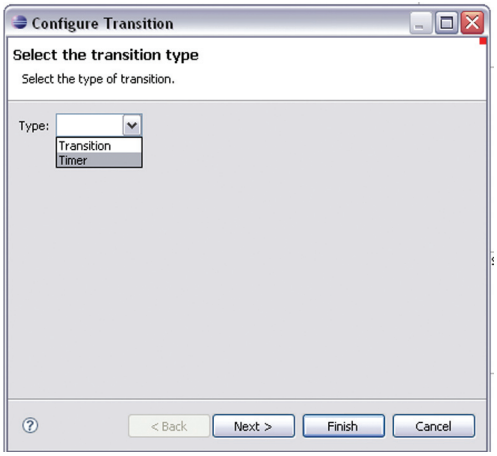


2.3 Configuring a transition

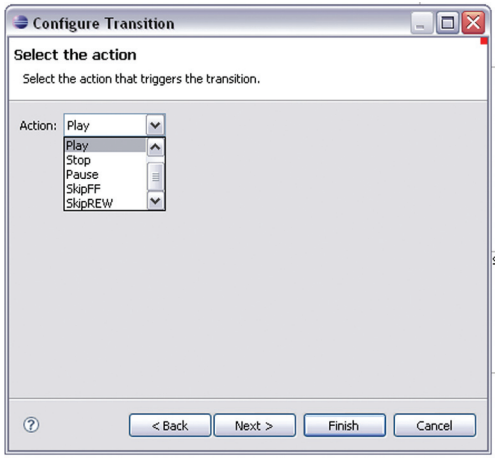
1. Right-click on the transition that you want to configure. Select ”Renegade” -> ”Configure Transition”.



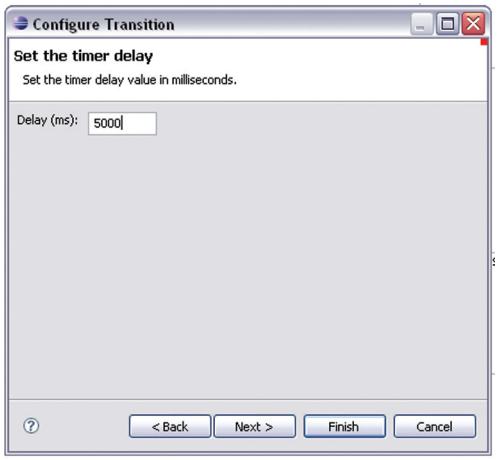
2. Select the type of transition you desire.
Press ”Next” to proceed to the next step.
Press ”Finish” to save and end the configuration of this transition.
Press ”Cancel” to abort the configuration of this state and discard any choices that has been made.



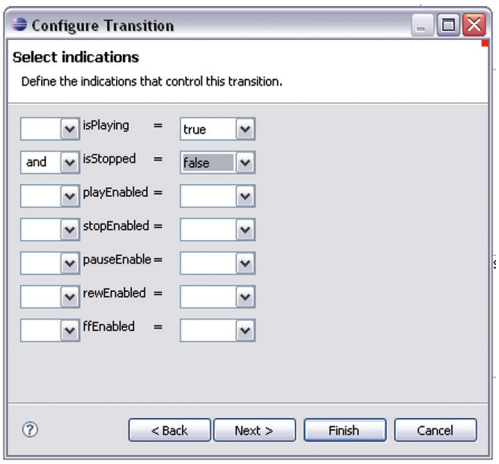
3. If you chose ”Transition” in step 2, select the action that should trigger this transition.
Press ”Back” to return to the previous step.
Press ”Next” to proceed to the next step.
Press ”Finish” to save and end the configuration of this transition.
Press ”Cancel” to abort the configuration of this state and discard any choices that has been made.



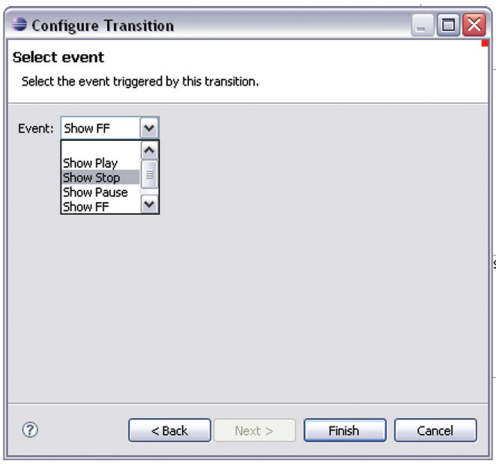
4. If you chose ”Timer” in step 2, enter the the time (in milliseconds) that you want to delay the transition.
Press ”Back” to return to the previous step.
Press ”Next” to proceed to the next step.
Press ”Finish” to save and end the configuration of this transition.
Press ”Cancel” to abort the configuration of this state and discard any choices that has been made.



5. Set the indications that you want to control this transition. Indications can be joined with ”and” or ”or”-statements.
Press ”Back” to return to the previous step.
Press ”Next” to proceed to the next step.
Press ”Finish” to save and end the configuration of this transition.
Press ”Cancel” to abort the configuration of this state and discard any choices that has been made.



6. Select the event that you want to be sent out from the simulation on this transition.
Press ”Back” to return to the previous step.
Press ”Next” to proceed to the next step.
Press ”Finish” to save and end the configuration of this transition.
Press ”Cancel” to abort the configuration of this state and discard any choices that has been made.



NOTE! When you select two or more indications, they HAVE to be joined with an ”and” or ”or”-statement between each indication. If not, this will cause errors in the simulation.

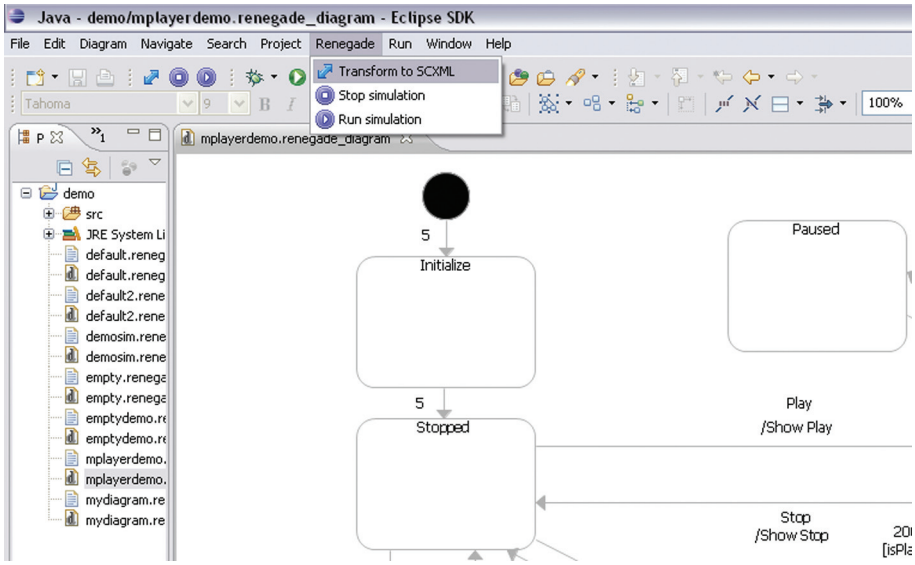
3. Simulating a Functional Unit

This section describes how to run and monitor a simulation of the statechart diagram that you have created.

3.1 Transform diagram into SCXML

The SCXML transformation is done automatically when a simulation is run, but it is possible to only do the transformation if you want to check out the code for debug purposes, for example.

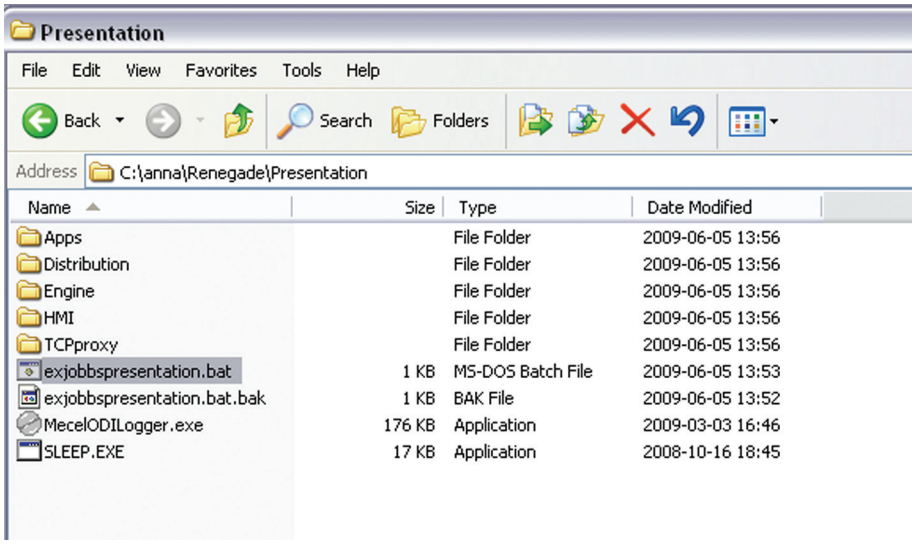
1. Go to the menu and select ”Renegade” -> ”Transform to SCXML”.



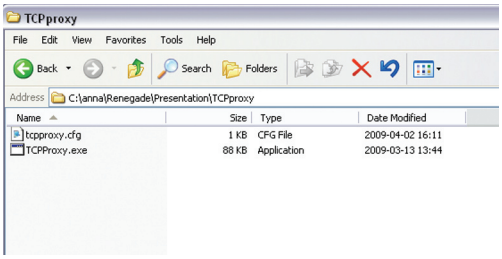
3.2 Running a simulation

1. Make sure that the correct diagram is active in the Eclipse application.

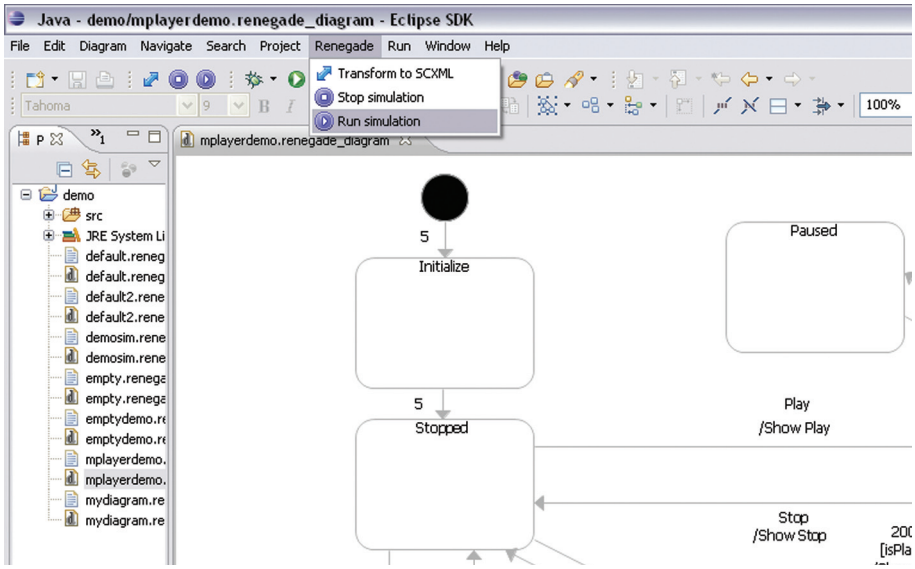
2. Start the HMI that you want to test against the simulation.



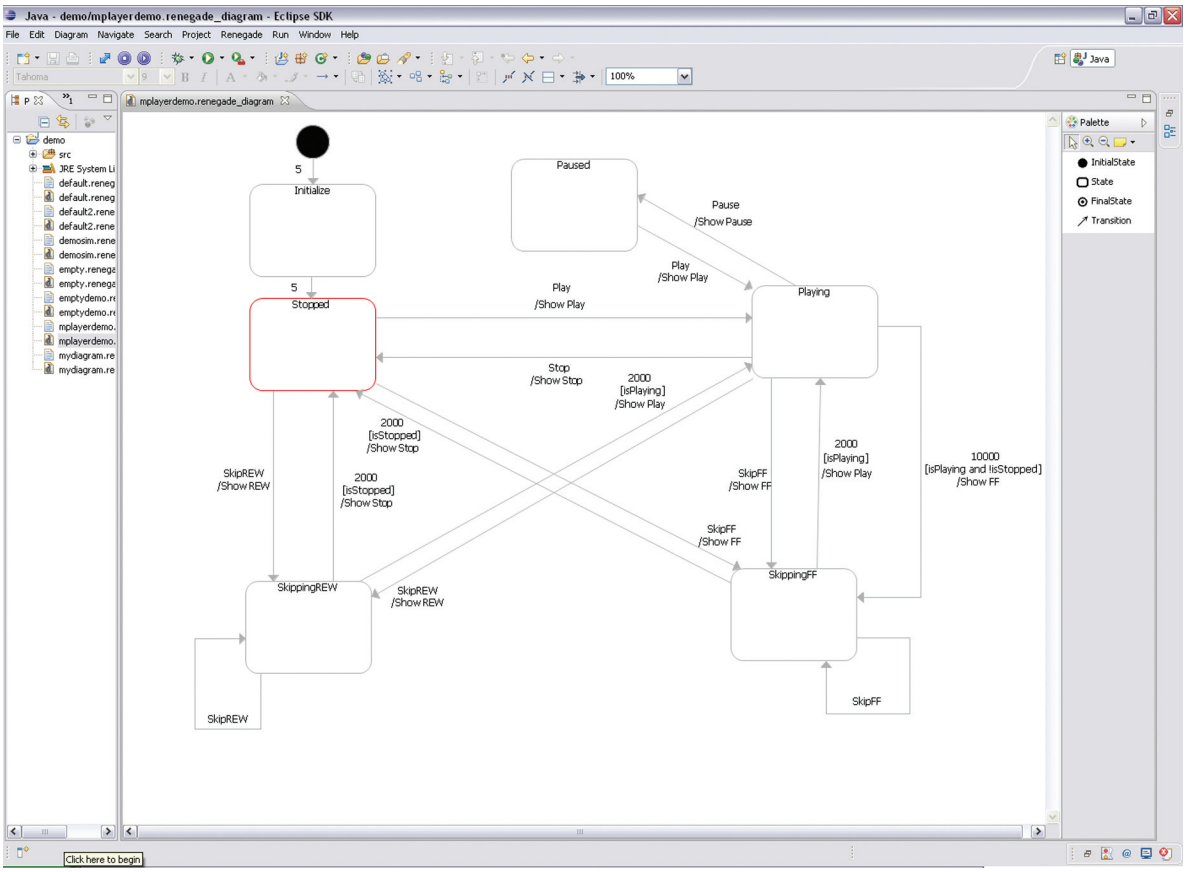
3. Start the TCP proxy.



4. Go to the menu and select ”Renegade” -> ”Run simulation”.



5. Wait for the first state to become active. This will be done when the state turn red.



6. Manipulate the HMI as desired to test it out against the simulation. You can follow the progress through the diagram through the visual feedback, always showing the currently active state in red.

3.3 Log output

In addition to the visual feedback, there are also a textual log mechanism connected to the Renegade Simulator. It can be set to either showing output from the SCXML file or pure debug information.

The log can be opened and examined in any text editor when a simulation is finished. For being able to show the log updates in runtime we recommend a program like BareTail (can be downloaded for free from http://www.baremetalsoft.com/baretail/).

3.4 Stopping and restarting a simulation.

Although there is a stop option in the menu, it is not implemented yet and therefore the simulation has to be stopped and restarted in another way for now.

1. Close the Eclipse application to stop the simulation.

2. Restart the application according to 1.1

3. Restart the simulation according to 3.2

NOTE! Although it might not be necessary, it is recommended to restart the HMI and TCP proxy too when starting a new simulation.