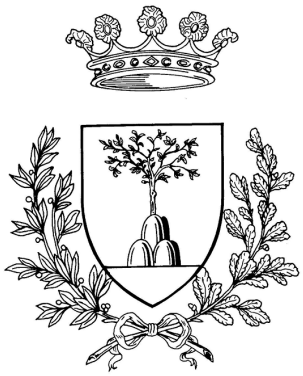
NICOLA MIGLIORINI

MODEL TO TEXT TRANSFORMATION: FROM UML STATECHARTS TO SCXML BASED CONTROL APPLICATIONS

UNIVERSITÀ DEGLI STUDI DI FERRARA
FACOLTÀ DI INGEGNERIA



CORSO DI LAUREA IN INGEGNERIA ELETTRONICA

MODEL TO TEXT TRANSFORMATION: FROM UML STATECHARTS TO SCXML BASED CONTROL APPLICATIONS

Laureando
Nicola Migliorini

Relatore
Prof. Marcello Bonfé

Correlatori
Dott. Gianluca Chiozzi
Ing. Luigi Andolfato

Anno accademico 2009/2010

Nicola Migliorini: *Model to Text Transformation: from UML Statecharts to SCXML based control applications,* Corso di laurea in ingegneria elettronica, Università degli studi di Ferrara © March 2011

SUPERVISORS:
Prof. Marcello Bonfé
Dott. Gianluca Chiozzi
Ing. Luigi Andolfato

Forget about the troubles of the past and live in the present.

Hakuna matata
— Timon & Pumbaa

Dedicated to the loving memory of my father.
1951 – 1995

ABSTRACT

That of UML statecharts is a very flexible formalism. UML statecharts are used to describe the logic of a software system. They are very useful in designing phase of the system and helpful in the production of the documentation.

In this work I propose the realization of a code generator developed at ESO. The tool take as input a UML state machine and the generated code will be the skeleton of an application. Developers must complete the application with hand-written code that bound the generated code to the desired target platform. This kind of approach promote the possibility to reuse models to develop applications for different platforms.

SOMMARIO

Quello degli UML statecharts è un formalismo molto flessibile. Gli statecharts vengono utilizzati per descrivere la logica di un sistema software. Sono molto utili in fase di progettazione e di supporto nella realizzazione della documentazione.

In questo lavoro presento la realizzazione di un code generator sviluppato presso ESO. Il code generator è in grado di produrre un'applicazione a partire dal modello UML di una macchina a stati finiti. Il codice generato costituisce lo scheletro di un'applicazione che lo sviluppatore dovrà completare con la particolare implementazione che lega il tutto alla piattaforma per cui si sta sviluppando. Questo tipo di approccio garantisce e promuove la possibilità di riutilizzare i modelli per sviluppare codice su diverse piattaforme.

ACKNOWLEDGMENTS

Many thanks to everybody who helped me in this work. First of all to Gianluca Chiozzi and Luigi Andolfato from whom I learned a lot. Heiko Sommer, Robert Karban, Reynald Bourtembourg, and all the people at ESO for the help and the good time spent while doing my work. Many thanks to my professors, Sergio Beghelli and Marcello Bonfé, for believing in my abilities and giving me this great opportunity.

A special thanks to my mom, my brother and all my family that has always supported me in this long path. And thanks to Petra that has always been at my side, believing in me even in the hardest moments.

Finally, many thanks to all the great new and old friends who have accompanied me this far.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

ACS ALMA Common Software

ALMA Atacama Large Millimeter/Sub-Millimeter Array

ATs Auxiliary Telescopes

CWM Common Warehouse Metamodel

CASE Computer Aided Software Engineering

E-ELT European Extremely Large Telescope

ESO European Southern Observatory

GSME Generic State Machine Engine

MDA Model Driven Architecture

MDD Model Driven Development

MOF Meta Object Facility

OCL Object Constraint Language

OMG Object Management Group

PIM Platform Independent Models

PSM Platform Specific Models

SDD Software Development Division

UML Unified Modeling Language

UTs Unit Telescopes

VLT Very Large Telescope

VLTI Very Large Telescope Interferometer

XMI XML Metadata Interchange

Part I

UNDERSTANDING THE TOPIC

1 INTRODUCTION

Contents

In the 60 years of history of the programming languages, an always constant trend has been the will to raise the level of abstraction. First we passed from programming codes to low level languages like Assembly. Then came out the first procedural languages (FORTRAN and COBOL), followed by the functional languages like LISP. Coupled with the evolution of the hardware, software became more and more complex. The dimensions of applications grew, highlighting the limits of the functional approaches. After the structured paradigm of PASCAL and C, the next step was the incremental development: with the collaboration of teams of developers, small pieces of code were put together. This was the base for the *object oriented paradigm*. Code reuse was one of the main targets[13].

Nowadays a medium application consist of hundreds of thousand (if not millions) of line of code. Is not thinkable to write it without the use of tools and an appropriate approach. Again the trend is to raise the level of abstraction. With the Model Driven Development (MDD) approach, developers have appropriate tools to write and maintain huge applications. Furthermore MDD offers a good support in developing documentation. With the help of automatic code generation repetitive tasks are avoided by developers. They can focus on the logic of the application, thinking about the big picture, and letting the code generator to deal with the implementations details.

With this kind of strategies and a correct approach, time is saved and resources are used in a better way.

1.1 OBJECTIVES OF THIS WORK

This project aims to the creation of a reusable State Machine Code Generator. The main purpose is to have a state machine implementation that allows execution and control of the state machine logic. As the code generator must be reusable on different

platform the choice of using open and well defined technologies like CORBA, UML and XML is obvious. The target is to obtain a running application starting from a state machine model. The obtained application will be the skeleton on which developers will build the software with the implementations details.

1.2 ESO

1.2.1 *The organization*

Since December 2010 Brazil is also a participant

The European Southern Observatory (ESO) is an intergovernmental research organization for astronomy, composed and supported by fifteen countries. Established in 1962 with an objective to provide state-of-the-art facilities and access to the sky of the southern hemisphere to European astronomers, it is famous for building and operating some of the largest and most technologically advanced telescopes in the world, such as the New Technology Telescope located at La Silla Observatory, the telescope that pioneered active optics technology, and the VLTI (Very Large Telescope Interferometer) located at Paranal.

Its numerous observing facilities have enabled many astronomical discoveries, and produced several astronomical catalogs. Among the more recent discoveries is the discovery of the farthest gamma-ray burst and the evidence for a black hole at the center of our Galaxy, the Milky Way. In 2004, the VLT allowed astronomers to obtain the first picture of an extra-solar planet, orbiting a brown dwarf 173 light-years away. The VLT has also discovered the candidate farthest galaxy ever seen by humans[10].

1.2.2 *Main projects*

All ESO observation facilities are located in Chile (because of the need to study the Southern skies and the unique atmospheric conditions of the Atacama Desert, ideal for astronomy), while the headquarters, with the scientific, technical and administrative center of the organization, are located in Garching bei München, Germany. ESO operates three major observatories in Chile's Atacama desert, one of the driest places on Earth:

- La Silla Observatory, which hosts eighteen telescopes (three of them are still operated by ESO for use by the astronomical community), a 3.6 horseshoe mount telescope(mostly used for infrared spectroscopy and for the search of extra-solar planets) and other less important facilities
- Paranal Observatory, which hosts the Very Large Telescope,the VISTA telescope and the italian VST (VLT Survey Telescope)

- Llano de Chajnantor Observatory, which hosts the APEX (Atacama Pathfinder Experiment) submillimeter telescope and where ALMA, the Atacama Large Millimeter Array, is currently under construction in a collaboration between East Asia (Japan and Taiwan), Europe (ESO), North America (USA and Canada), and Chile.

Each year, about 2000 proposals are made for the use of ESO telescopes, requesting between four and six times more nights than are available. ESO is one of the most productive ground-based observatories in the world, which annually results in many peer-reviewed publications: in 2009 alone, more than 650 refereed papers based on ESO data were published. Moreover, research articles based on VLT data are in the mean quoted twice as often as the average. The very high efficiency of the ESO's "science machines" now generates huge amounts of data at a very high rate. These are stored in a permanent Science Archive Facility at ESO headquarters. The archive now contains more than 1.5 million images or spectra with a total volume of about 65 terabytes (65,000,000,000,000 bytes) of data[10].

ESO has also hosted the European Coordinating Facility for the Hubble Space Telescope, a collaboration between ESA and NASA. The HST-ECF has been a long-term, space-based observatory. In 20 years of activities, in many ways Hubble has revolutionized modern astronomy, by not only being an efficient tool for making new discoveries, but also by driving astronomical research in general. The ST-ECF has closed and ceased operations on 31 December 2010.

1.2.2.1 *VLTI*

The Very Large Telescope (VLT) at Cerro Paranal is ESO's premier site for observations in the visible and infrared. The Very Large Telescope Interferometer (VLTI) consists in the coherent combination of the four VLT Unit Telescopes and of the four movable Auxiliary Telescopes.The VLTI provides both a high sensitivity as well as milli-arcsec angular resolution using baselines of up to 200 meters length.

The four 8.2 meters Unit Telescopes (UTs) and the four 1.8 meters Auxiliary Telescopes (ATs) constitute the light collecting elements of the VLTI. The UTs are set on fixed locations while the ATs can be relocated on 30 different stations. The telescopes can be combined in groups of two or three. After the light beams have passed through a complex system of mirrors and the Delay lines, the combination at near- and mid-infrared is performed by the instruments. A complex and high performance dual-feed system that allows Phase Referenced Imaging and Micro Arcsecond Astrometry on the VLTI will be available soon. Due to its

Figure 1: The VLTI Array on Paranal mountain.

characteristics, the VLTI has become a very attractive mean for scientific research on various objects like many stars in the solar neighborhood or extragalactic objects[9].

The VLT Common Software is the development software infrastructure platform for VLT applications. It provides the building blocks for all applications and has been used by all internal and external development teams. The size of the VLT control software, including telescope control software, is about 1.5 million lines of code and might become about 3 million lines when the full instrumentation complement is ready [15]. The VLT common software consists of a layer of software over the Unix operating system, in the case of workstations and on top of the VxWorks operating system, for the Local Control Unit (LCU) microprocessors. It provides mainly common services, like an architecture independent message system, a real-time database for all telescope and instrument parameters, error and logging systems and a large number of utilities and tools. The languages used are C (lower layers) and C++ (and Object Oriented concepts) on the workstation side, while the code running on the microprocessors is written exclusively in C. The User Interfaces on the workstations are built using the VLT Panel Editor, which is based on Tcl/Tk[11].

1.2.2.2 *ALMA*

Atacama Large Millimeter/Sub-Millimeter Array (ALMA) is a joint project between astronomical organizations of Europe (ESO),

North America (NRAO), and Japan (NAOJ). ALMA is a large radio-astronomical project that will consist of at least 50 twelve meter antennas operating in the millimeter and sub-millimeter wavelength range, with baselines up to 10 kilometers. It will be located at an altitude above 5000 meter on the Chajnantor plateau in the middle of the Chilean Atacama desert. The science commissioning of ALMA is starting now, and the first astronomical observations are scheduled for the end of 2011. At the moment

ALMA will be comprised of a giant array of 50 12-meters antennas

Figure 2: The ALMA array at the high-elevation Array Operations Site.

7 antennas are aleady in place and the project goes on at a fast pace.

ALMA Common Software (ACS) is an Object Oriented CORBA-based middleware software framework for science facilities that handles communication between distributed objects. ACS was designed and developed to support the complex control requirements of ALMA radio telescopes, but can be used to support control and data flow for any system with similar performance requirements.

ACS provides a set of packages containing development tools, and common services and patterns needed to build and deploy object oriented and distributed systems. Most of the features provided by ACS are implemented using off the shelf components. ACS itself provides "glue layers" between these components, hiding all the details of the underlying mechanisms and complex CORBA features from the developer.

As stated in [18] the ACS architecture is based on the Component-Container model. Containers provide an environment for several portions of software called Components. They also provide several services to, and manage the lifecycle of, the Components. This way, components developers can focus on domain problems rather than on software engineering issues.

There are a number of projects that have decided to base their system on ACS, like the Atacama Pathfinder Experiment, the Spanish OAN 40 meters radio telescope, the Sardinian Radio Telescope in Italy, the Hexapod Telescope in Chile or the ANKA Synchrotron in Germany. Other projects are evaluating this possibility [7].

1.2.2.3 *E-ELT*

Figure 3: A render of the E-ELT project.

The European Extremely Large Telescope (E-ELT) project aims to provide European astronomers with the largest optical-infrared telescope in the World. With a diameter of 42 meters and being fully adaptive, the E-ELT will be more than one hundred times more sensitive than the present-day largest optical telescopes. The E-ELT will vastly advance astrophysical knowledge by enabling detailed studies of planets around other stars, the first galaxies in the Universe, super-massive black holes, and the nature of the Universe's dark sector.

The E-ELT is designed as a reflecting telescope. It has a footprint of about 80 meters diameter and is about 60 meters high. The altitude and azimuth structures weigh together nearly 5000 tons. This structure supports the five mirror optical design and accommodates two Nasmyth platforms. Each platform is of about the size of a tennis court and can host several instruments. Several designs were considered for the telescope enclosure. The project settled for a rather classical dome design[8].

The E-ELT will be the largest optical telescope in the world

The E-ELT project has been ranked in the 2010-2025 ASTRONET[1] European strategic planning as one of two clear top priorities for future ground-based astronomical infrastructures[6]. The project is currently in its detailed design phase, under a grant from the European Commission. The start for E-ELT construction is planned for the end of 2011, with start of operations planned for the end of the decade[19].

1 Astronet is a consortium which gather European funding agencies in order to "establish a comprehensive long-term planning for the development of European astronomy"

The software framework is actually being defined[27]. Two of the most important aspects of the E-ELT's development are the control of high precision optics over the huge scale of the telescope and the design of an efficient suite of instruments to achieve the E-ELT's ambitious goals. All starts from a flexible suite of tools and from a software infrastructure used to configure, control and acquire data from E-ELT instruments.

1.3 STRUCTURE OF THE DOCUMENT

This document is organized in parts, each of which is divided in chapters, with the the following structure:

- `Part one: Understanding the topics` presents an overview of the topics touched developing the code generator, object of my thesis
  - Chapter 2 introduces the concept of `Model Driven Development`
  - Chapter 3 is a presentation of the state machine and the different incarnations in languages like UML and SCXML. A first introduction to the Apache SCXML engine is also in this chapter.
  - Chapter 4 outlines the tools used in the development of the code generator.
- `Part two: Developing the solution` outlines the work done at ESO to project and develop the code generator.
  - In Chapter 5 a first mapping from UML to SCXML is presented
  - Chapter 6 examines the code generator and a running example developed at ESO for the components of a control system
  - Chapter 7 keeps an eye on the possible future improvements of my work

2 MODEL DRIVEN DEVELOPMENT

Contents

Model Driven Development focuses on developing software components using abstractions of software systems, i.e. models. Models are also used, with a certain amount of details, to describe a software application. MDD is meant to simplify the process of design the components of the system and the relations between different modules.

2.1 WHY WE MODEL

With the support of a graphical model representation, the whole structure of the application is easy to understand at a glance and also newcomers can easily jump in a work in progress. This helps a lot in projects where a big team is involved in the development. Models help to keep the focus on details without loosing the big picture and make the collaboration much easier. Models are then converted into code. The conversion can be made manually or automatically, using tools designed for this. Automatic code generation from models reduces the manually written lines of code and with this the number of errors per line of code. Code reuse is encouraged and made easier. In fact, mostly in big projects, a large number of line of code is repeated in various components. Having a tool that can do this repetitive work not only leads to an increased productivity, but also to a coherent and cleaner code. In addition enhancements made on the code generator are propagated on all the generated components. This is a strength but also a critical point since if a bug is present on the code generator all the generated code will be affected.

But the advantages of the MDD are also others. With an higher level of abstraction developers can forget about the implementation details. Specify less and generate more. Also this has an other side of the coin: less control on detail leads to rigidity.

The Model approach is strongly supported by the Object Management Group (OMG) that, supplying open and vendor-neutral interoperability specifications, provides a basis for developing this strategy. With the Model Driven Architecture (MDA), OMG

offers "an architecture that makes interoperability central to your infrastructure" [25]. The OMG Model Driven Architecture encapsulates many important ideas, most notably the notion that real benefits can be obtained by using visual modeling languages to integrate the huge diversity of technologies used in the development of software systems. In the era of Internet and web services, achieving interoperability is a necessity. With the model approach developers can make a separation between the specification of system functionality and the specification of the implementation of that functionality on a given platform. In MDA this is a key distinction that is fundamental and leads to Platform Independent Models (PIM) and Platform Specific Models (PSM). PIM specifies services and interfaces that the software systems must provide, independent of software technology platforms. The PIM is further refined to a PSM which describes the realization of the software systems with respect to the chosen software technology platforms. Again [25] gives a definition for both these kinds of models and states that modeling staying away from platform specific details makes things easier when implementing the applications for different platforms. The integration and interoperability are easier to achieve thinking with platform independent models, and thereafter easier to implement with platform dependent details. In addition, the model approach faces also the problem of the proliferation of middleware frameworks. Big enterprises have to deal with different middleware platforms and again the necessity to achieve interoperability is evident. Each department of the enterprise may have different requirements for projects, or simply, a mix of technology to deal with. It's not worthy to port different already working applications to a single middleware platform if it's possible to make them cooperate with a smart project mentality. These are the main reasons that make the model approach so important. With a well designed model developers gain the flexibility and ability to obtain valid code even when the infrastructure changes over time. And as the whole approach is platform independent, when a new and better technology will be available, integrating this on the project will be an easy step[16].

Division between what is dependent from the platform and what belongs only to the logic of the system

Let's look at this question from the point of view of the construction trade. Architects design buildings. Builders use the designs to create buildings. The more complicated the building, the more critical the communication between architect and builder. Blueprints are the standard graphical language that both architects and builders must learn as part of their trade.

"Writing software is not unlike constructing a building. The more complicated the underlying system, the more critical the communication among everyone involved in creating and deploying the software"[25]. In the past decade, the Unified Modeling Language (UML) has emerged as the software blueprint language

for analysts, designers, and programmers alike. It is now part of the software trade. The UML gives everyone from business analyst to designer to programmer a common vocabulary when talking about software design.

2.2 MODELS

In a model only the relevant aspect of the system are represented

But what exactly is a model? A model is a representation of a system. A model group a set of concepts about the subject it represent. There concepts are called abstractions. Models consist of objects that interact by sending each other messages. Thinking of objects as live entities, they have things they know (attributes) and things they can do (behaviors or operations). The values of an object's attributes determine its state. But the strength of the model representation is the possibility to simplify the reality. It's not possible to represent every detail of the subject, as the amount of informations is to big. Focusing on the relevant informations related to the problem we are facing, helps on the understanding of the problem itself and leads to a solution in a easier way. The developer must choose the level of details to be described. And this is the key to manage the complexity involved in system development.

2.3 THE MDA APPROACH

MDA: the approach to model driven development proposed by OMG

In early 2002, the OMG described the general principles for MDA[21]:

> *"The MDA defines an approach to IT system specification that separates the specification of system functionality from the specification of the implementation of that functionality on a specific technology platform. To this end, the MDA defines an architecture for models that provides a set of guidelines for structuring specifications expressed as models."*

> *"The MDA approach and the standards that support it allow the same model specifying system functionality to be realized on multiple platforms through auxiliary mapping standards, or through point mappings to specific platforms, and allows different applications to be integrated by explicitly relating their models, enabling integration and interoperability and supporting system evolution as platform technologies come and go."*

In other words with MDA the development of a software system is based on the separation of the business aspects from the implementation aspects. Modelling these different aspects with

various level of abstraction, focusing on models rather than on code, and derive the code from models with automatic generation will reduce complexity.

MDA provides a set of specifications that support the modeling of functional aspects of a business application in the form of a Platform Independent Models. As its name implies, a PIM is a formal model that is independent of any specific implementation technologies. So the same PIM ultimately could be used to develop an implementation based on Java, or .Net, or any other platform. The Model Driven Architecture then provides a systematic way to map a PIM to one or more Platform Specific Models. As its name implies, a PSM is targeted for a particular computing platform, such as J2EE or .Net. Using other features of MDA, the resulting PSM then can be used to generate code, data structures, configuration information, as necessary for the chosen deployment technologies.

The key concepts of MDA are abstraction and automation

Using MDA, the development of a system can be faster. PIM can be developed by business analysts with little or no technical backgroung but with a good knowledge of the domain of the application. Advanced programmers can focus on the implementation details of the application.

In addtion MDA can be used to integrate applications between different platforms, encouraging the reuse of elements across different frameworks.

But all the advantages have a price. The heavy use of standards is required to achieve interoperabilty. Developers must be pragmatic in thei approach of implementing MDA. The use of already existing solutions and well accepted approaches is the base of this development strategy.

Portability, cross-platform interoperability and platform independence are achieved through the use of such well known standards as CORBA or XML. The qualities of portability, interoperability, and platform independence lead to the domain specificity, the ability to focus on solving particular business problems. This means developping the best solutions beeing aware of the domain of the application: a bank, an hospital or a space ship. In either case there is no need to worry about how these solutions should be coded, or how they will run on a specific platform. That set of worries is deferred till later in the development.

With the classic approach each project is cartried out by a team of developers that must have a knowledge of the problem domain and of the platform adopted. If for some reason the team breaks up, or leave the company all the knowledge is lost. With MDA also the teams are separated. The platform knowledge and the domain knoledge are divided and each team take care of his own area. If something change at the business level, domain experts can update the corresponding models. Platform experts provide

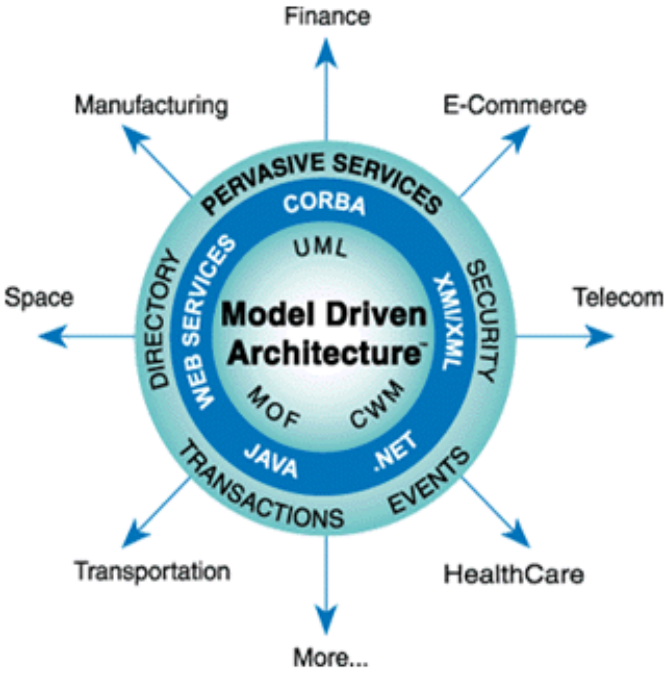


Figure 4: OMG's Model Driven Architecture from OMG website[23].

technical details of their respective platforms, which are then semi-automatically added to the mix. Finally, developers are free to focus on what they are best at: choosing the most appropriate technology, and obtaining the best results.

Division between domain experts and advanced programmers

All this leads to software being less sensitive to changes in personnel (if someone joins a project it is far easier to understand the high-level model of the software application compared to trying to understand the behavior of the application by reading source code) and being less sensitive to changes in technology (if a technology changes we don't need to change all models but only the PSM. After changing this the application can be quickly generated with the new technology).

Although the standards adopted from OMG have been defined years ago we are far from having a complete definitions. UML, Meta Object Facility (MOF), Object Constraint Language (OCL), Common Warehouse Metamodel (CWM), and XML Metadata Interchange (XMI) are evolving every day. It is often necessary to transform models from one language to another, or even between different "dialects" of UML. This can be done only if all the formal languages used are consistent and compatible. This is the MDA infrastructure and must be solid and coherent.

MDA is a philosophy and not a standard. But it is based on a whole array of standards, many of which are evolving. It is a promising work in progress that will lead to an high level of productivity, flexibility and standardisation.

3

STATE MACHINES

Contents

> *"A finite state machine is an abstract machine that defines a finite set of conditions of existence (called "states"), a set of behaviors or actions performed in each of those states, and a set of events which cause changes in states according to a finite and well-defined rule set"*

This is how Bruce Powell Douglass, an expert with 30 years of experience in real time and embedded systems from IBM, defines a finite state machine [14]. Therefore a finite state machine is a model representing a reactive system and focusing on its behavior. It consists of a set of states and transitions between them. Every transition is triggered by an event. Events are the inputs of the systems. In other words events can lead to a different state in which one or several actions can be performed. Actions can be associated with states or with transitions and represent the system's output.

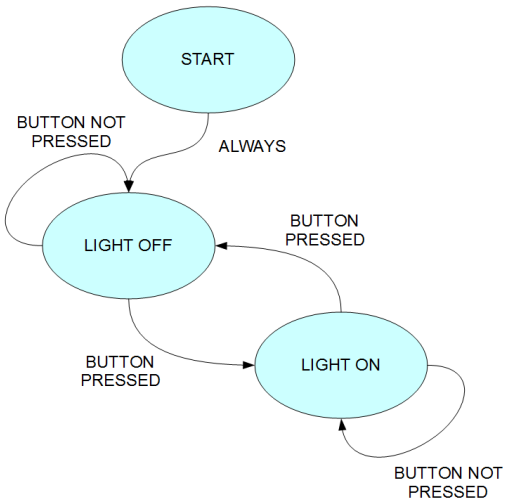


Figure 5: A simple state machine.

In Other words a state machine is a kind of "black box" that responds to external stimuli. Given the current state of the device, a given input leads to a particular output and to a new state. All this is represented using simple diagrams. These diagrams are similar to a flow graph. Fig.5 shows a simple state transition diagram of a switch.

3.1 STATECHARTS

Statecharts (Fig.6) were initially defined by D. Harel in his milestone "Statecharts: a visual formalism for complex systems" [20]. Essentially he introduced in the finite state formalism concepts like hierarchy, orthogonality and broadcast communications. The obtained result is an extension of the state machine formalism with reduced visual complexity. Graphs are easy to understand even without a deep knowledge of the rules of the statecharts. Many situations are represented with less elements and this, again, leads to a minor complexity.The idea was then adopted from OMG and statecharts are now part of UML standard [22]. Let's have a quick look to the statechart formalism.

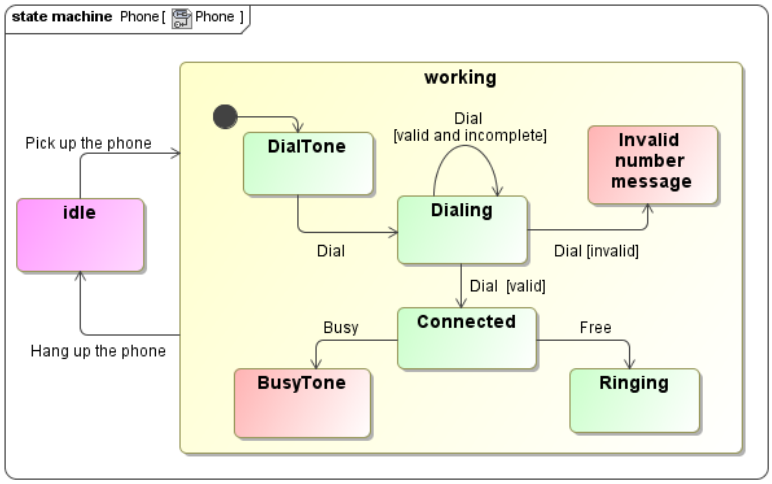


Figure 6: A simple statechart.

Statecharts consist of three primary entities: *states*, *transitions*, and *actions*. As with the FSM formalism, states represent conditions of existence that persist for a significant period of time. Transitions are the means by which objects change states in respond to events. A state machine can execute actions at various points in a state machine, such as when an event triggers a transition, when a state is entered or when a state is exited. Actions may be simple statements, such as an increment of a variable, or they can invoke operations defined within the context object or other objects. Graphically a state is indicated with a rounded box. A name in the box identifies the state.

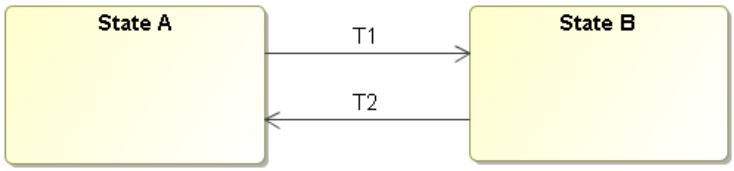


Figure 7: States and transitions.

A transition is a relationship between two states indicating that a machine in the first state will enter the second state when a specified event occurs and specified conditions are satisfied. On such a change of state, the transition is said to fire. It is expressed in the form:

TRIGGER[GUARD]/ACTION

- TRIGGER is the event that fires a transition.
- GUARD is a boolean condition. Conditions on transitions must be mutually exclusive.
- ACTION is some behavior executed during the transition.

All parts are optional. Every Transition must have at least one source and one target. A "dangling" Transition is not allowed. An internal transition is a transition without a target. This kind of transition could be used as an event handler. The event is processed without leaving the state. This is different from another special transition, the self-transition. This is a transition which has as target the source of the transition itself. With this kind of transition exit and entry actions are triggered every time the transition is taken.

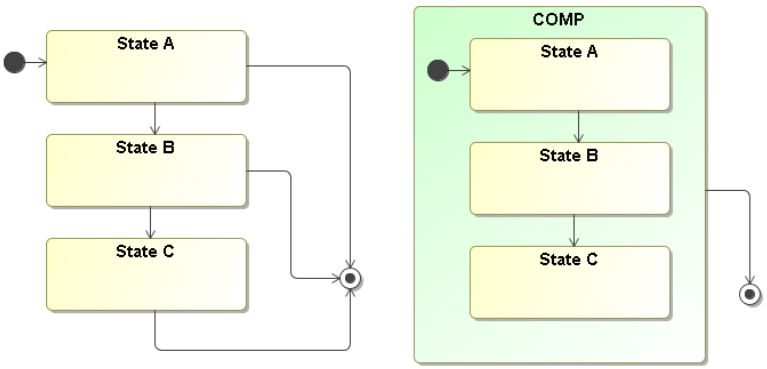


Figure 8: Use of a composite state.

Transitions are indicated by an arrow joining a state with another state or a pseudo-state (history states or final states). A transition can target elements at every level.

A first useful difference with the state machines is the concept of *complex states*. A super-state could be defined; this state identifies some features common to the sub-states. The main idea is to group some states and to reduce the number of transitions from the single states. Instead of having a transition from every sub-state, a single arrow can leave the super-state. This feature can represent an XOR between the inner states.

Every diagram must have a starting point. This is called "*Initial pseudo-state*" and is indicated with a black circle with an outgoing arrow. The arrow is pointing the starting state. Also inside composite states a default starting point is needed. the same formalism of the initial pseudo-state is used. In Fig.8 an example.

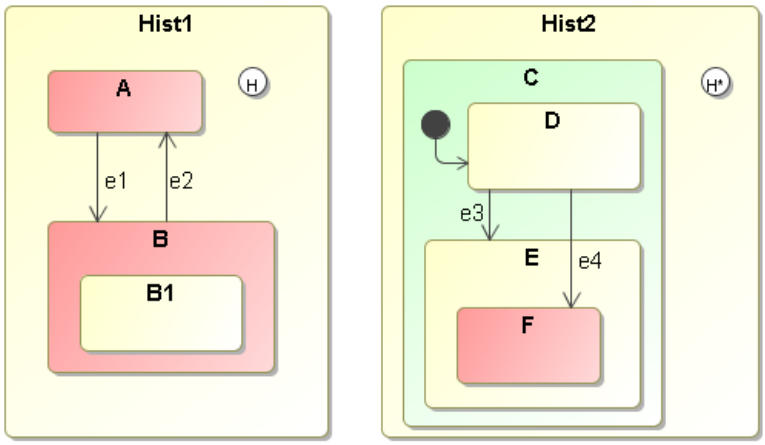


Figure 9: History pseudo-states.

Another kind of pseudo-state is the "history" element. This element represents the most recent active substate of its containing state. History is available in two version: *shallow history*, and *deep history*. The first is represented with a circle with an "H" inside and it allows to enter a composite state redirecting the flow to the last visited state from the ones the history father contains. The second version, the deep history, is indicated with a circle containing a "H*" symbol. This means the same of the previous item, but with recursion in the sub-states. In Fig.9 you can see the different cases. On the left if the state Hist1 is entered via the history pseudo-state, the machine will remember in which of the two state was between A and B, without considering B1. The case on the right instead, with the deep history, will remember also all the substate configurations inside Hist2.

When running, a finite state machine can be in several different states in the same moment. The concept of orthogonal states is represented by splitting a state in different regions (AND decomposition). Being inside this state means being in all his orthogonal regions simultaneously. Each region can be seen as a separate state machine running in a concurrent way. Graphically

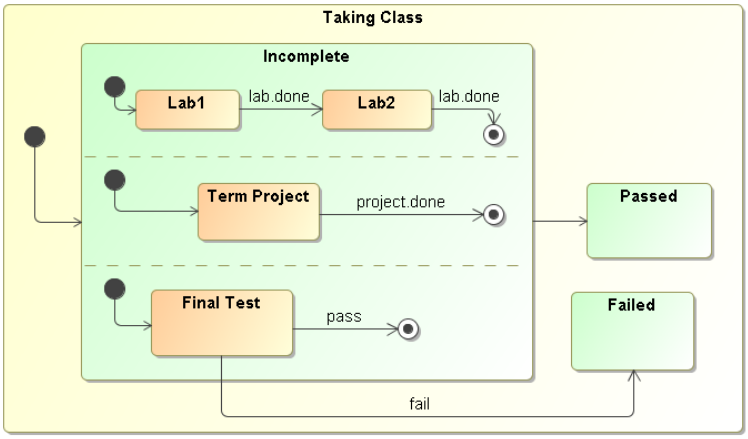


Figure 10: Use of orthogonal regions.

this concept can be represented with a dot line inside a state. This specifies a concurrent region. Fig.10 shows an example.

A very important feature is the possibility for the state machine to execute some actions. An action is an executable *atomic* computation, like a variable assignment, the rise of an event or performing I/O. It is supposed to happen instantaneously and cannot be stopped. Actions can take place while entering or leaving a state, in a transition or in response to an event. Entry and exit actions are a special kind of actions. They are dispatched entering or leaving a state, no matter which transition is taken. These action cannot have arguments or guards and are executed in any case. Entry and exit actions are indicated in the diagram with the keywords "entry" or "exit" followed by a slash and by the action name or command. Actions can be invoked also during a transition. In this case the action is executed after leaving a state and before entering the target.

The idea behind activities is that sometimes the abstraction of actions is not enough. Actions are supposed to be *instanta-*

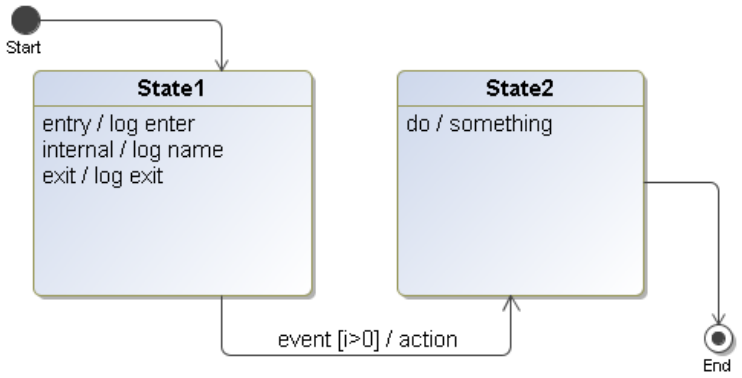


Figure 11: Use of actions and activities.

neous: once started, the state machine will wait until the action is finished. Some executions could not be represented in this way. They can take *a finite amount of time* and the developer could want to consider it. Furthermore could be necessary to block an execution because, for example, in the meantime we are leaving a state. This can be done with the Do-Activities. On the diagram this is identified with the "do" keyword followed by a slash and the invocation for the activity. On Fig.11 you can see a summary on how to use actions and activities.

3.2 UML: A VERY GENERAL PURPOSE DEFINITION FOR STATE MACHINES

The Unified Modeling Language is a general purpose graphical modeling language widely used in software development. UML aims to be a standard modeling language which can model concurrent and distributed systems. It is a de facto industry standard, and is evolving under the auspices of the Object Management Group (OMG). Version 2.3 [4] is the reference for this work. UML offers a family of formalisms useful to describe object oriented software systems. The language puts the emphasis on the graphics notation and allows the developers to focus on the structure and the design of applications. UML is flexible and comprehensive as it can be used to model anything. It is easily extensible by the user to fill any modeling requirement. An extension of the language for a given context is called *UML profile*. Diagrams could be mapped into any kind of hi-level language. The choice is left to the developer and bounded only to the tool you use.

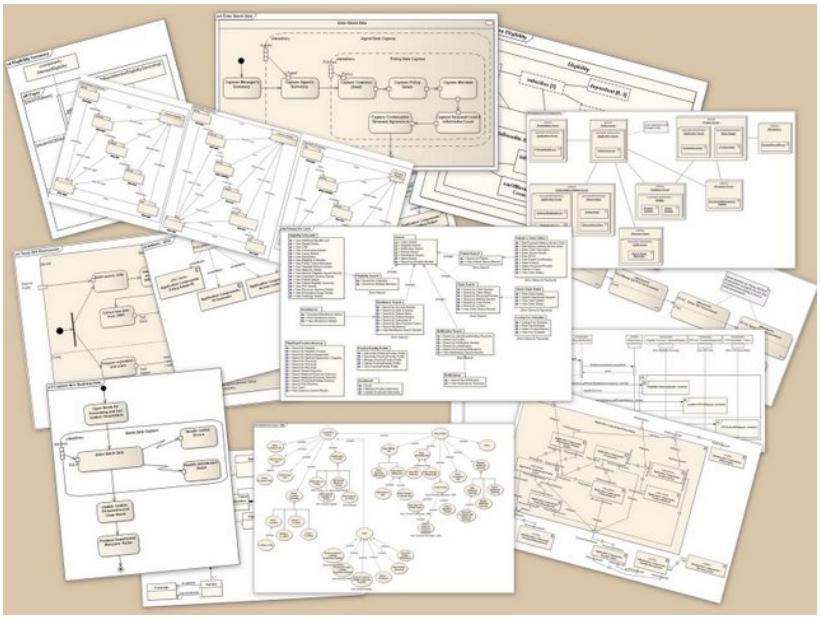


Figure 12: A collage of UML diagrams.

Focusing on state machines, the most used formalism is UML statechart. The formalism is a part of the UML standard and was originally defined following the specifics from D. Harel [20]. UML statechart diagram is an object-based variant of Harel's statechart. With UML statechart diagrams you can describe the behavior of a system with a quite simple and easy to understand graphic formalism. Statecharts add to the classical state diagrams a formalism to describe multiple cross-functional state diagrams within a state machine without loosing readability. The formalism offers the possibility to model superstates, pseudo-states and activity as a part of a state. In addition UML statechart imports the concepts of hierarchically nested states and orthogonal regions, as defined by Harel, and extends the notion of actions. [24]

The adoption of statechart concepts from OMG leads to a formal description and to a standardization of Harel's formalism, giving to it a solid base. Moreover the standard doesn't specify any detail for the implementation letting the developers to take care of the details.

3.3 SCXML: A WELL DEFINED STANDARD

State Chart XML (SCXML) is currently a working draft published by the World Wide Web Consortium [5] which provides rules to describe statechart models in a XML dialect. Events, transitions and policies to interpret the behavior of state machine are used to describe complex state machines with features such as sub-states, parallel states, synchronization and concurrency. SCXML is not yet a standard but it is a work in progress. The latest working draft is dated December 2010.

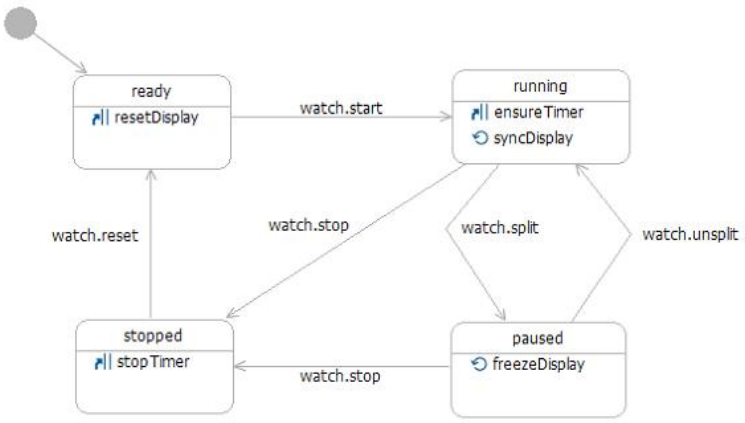


Figure 13: Stopwatch example from Apache Commons website.

With SCXML you can describe complex state machines using an XML based markup language. In Fig.13 there is the classical stopwatch example taken from the Apache Commons [3] website.

On the example the structure of the model is easily recognizable: there are states, with a unique name and inside every state the transitions. Each transition is triggered in response to an event. Following a transition the machine change its state to the targeted one. Models representing simple state machines are easy to read and easy to handle.

SCXML is mostly used in application where there is the need of an interactive dialog between a human and a computer like with automatic telephone services (voice access to email, call center, order inquiry, driving directions etc.). Many working implementations are available, most of them based on scripting languages like python. The Apache Foundation offers a pure Java library to parse and execute SCXML diagrams and also the Qt framework makes available a C++ implementation of an SCXML engine. The Qt framework is also used by Nokia to develop phone applications[2].

Listing 1: SCXML file for the stopwatch example

```
<?xml version="1.0"?>
<!--
 * Licensed to the Apache Software Foundation (ASF) under one
 * or more contributor license agreements.  See the NOTICE
 * file distributed with this work for additional information
 * regarding copyright ownership.
 * The ASF licenses this file to You under the Apache License
 * Version 2.0 (the "License"); you may not use this file
 * except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing,
 * software distributed under the License is distributed on
 * an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
 * KIND, either express or implied.
 * See the License for the specific language governing
 * permissions and limitations under the License.
-->
<scxml xmlns="http://www.w3.org/2005/07/scxml"
       version="1.0"
       initialstate="reset">

    <state id="reset">
        <transition event="watch.start"   target="running"/>
    </state>

    <state id="running">
        <transition event="watch.split"   target="paused"/>
        <transition event="watch.stop"    target="stopped"/>
    </state>

    <state id="paused">
        <transition event="watch.unsplit" target="running"/>
        <transition event="watch.stop"    target="stopped"/>
    </state>

    <state id="stopped">
        <transition event="watch.reset"   target="reset"/>
    </state>

</scxml>
```

4

USED TOOLS

Contents

Before presenting the used tools, a small summary about the work to be done is needed. The main objective of this work is to produce code for an application starting from a UML statechart model.

From a model to an application

The first required tool is a software capable of create and edit UML models. During my work I evaluated some options: MagicDraw from *No Magic Inc.*, Papyrus from *Eclipse Foundation* and Rational Rose from *IBM*. I ended to choice MagicDraw mainly for two reasons: the first is that the tool can export models in XMI, an interchange format used to transfer the models between different applications. The second reason is that there is a lively exchange of views between some developers working at ESO and the team at No Magic. This makes very easy the request of missing features or to correct wrong implementations in the application. Anyhow during my work I checked and verified the compatibility of my tool also with models exported from other applications.

Then I needed a tool to explore and manage models and to generate the code. Here the choice was quite easy as the *Eclipse Foundation* offers an open software development platform extensible and very flexible. For the same platform a powerful framework, explicitly designed to handle models, was available.

Finally an engine to run the state machines was required. After the decision to use SCXML to represent models was taken, mainly two were the possible options for the engine: the Apache SCXML engine and the Qt SCXML engine. Again the choice was easy as in the beginning I was working on the ACS platform. Many of the application for this platform are developed in Java, C++ or Python. Since the use of the Qt engine would have involved the adoption of big libraries not really needed for other purposes, the adoption of the Apache implementation was straightforward.

4.1 MAGICDRAW

MagicDraw is a visual UML modeling and CASE[1] tool from *No Magic Inc.*, with teamwork support. It is designed for business analysts, software analysts, programmers, QA engineers, and documentation writers. This development tool facilitates analysis and design of Object Oriented (OO) systems and databases. The tool supports UML 2.3 standard. It provides a code engi-

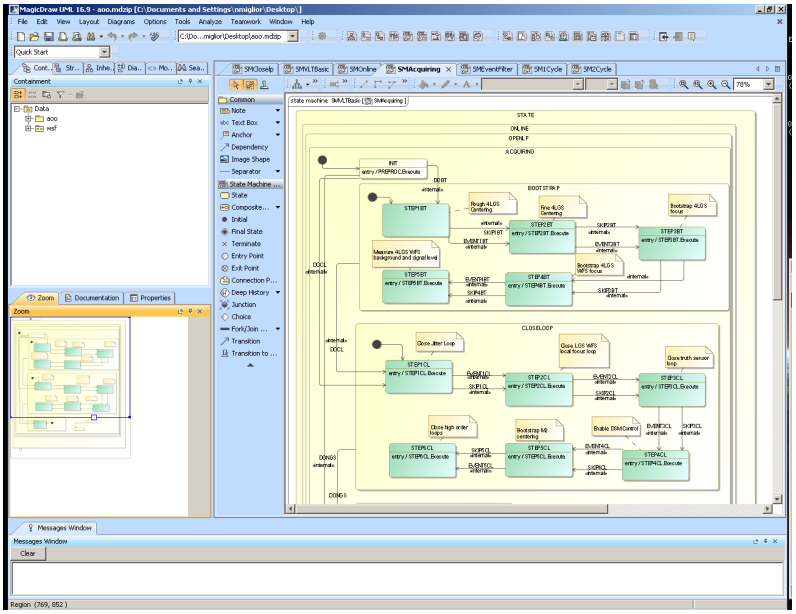


Figure 14: A screnshot from MagicDraw.

neering mechanism (with full round-trip support for J2EE, C#, C++, CORBA IDL programming languages, .NET, XML Schema, WSDL), as well as database schema modeling, DDL generation, and reverse engineering facilities.

I used this tool to draw and edit the UML state machine models. Thanks to the full support to the UML2 standard every features of the model is easily implemented. Once the model is finished, it is exported in XMI, an interchange format for UML models. This format is commonly used as a medium by which models are passed from modeling tools to software generation tools. The exported file is used to generate the SCXML model and the code for the application.

At the moment MagicDraw is offering a tool to convert the models in SCXML. This tool dosen't map the model but generates the SCXML through a simulator developed using the Apache SCXML engine. To generate the SCXML code the tool uses the calls to the Apache engine. After developing my project, I gave my contribution to the improvement of the SCXML exported by

1 Computer-aided software engineering

MagicDraw, pointing out inaccuracies and error to the No Magic team.

4.2 ECLIPSE

Eclipse.org is an open consortium of software development tool vendors like IBM, CISCO and intel. The community has interest in collaborating to create better development environments and product integration and shares an interest in creating products that are interoperable in an easy-to-use way based upon plug-in technology. The Eclipse Platform is a software development environment with an IDE and an extensible plug-in system. It is a platform designed to integrate different development tools. It supports a big variety of programming languages, from Java to PHP and Ruby. The whole environment is written in Java. Essentially is an IDE for nothing in particular. Its strength came from the plug-in system. The plug-ins can provide support for editing and manipulating additional types of resources such as Java files, C programs, Word documents, HTML pages, and JSP files. The plug-ins determine the functionality of the platform. Even MagicDraw can be used as a plugin from Eclipse.

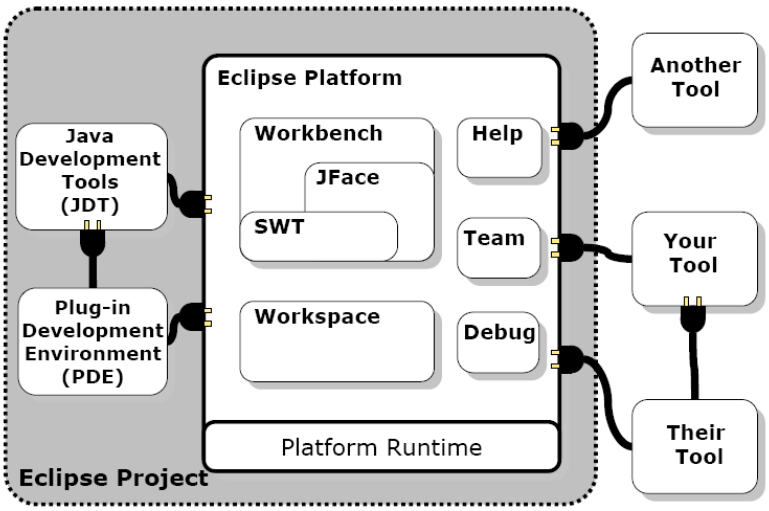


Figure 15: An overview of the Eclipse architecture.

Perspectives are arrangements of views and editors. Users can choose a perspective based on their needs of the moment. With a perspective you can control the layout of the working area, the visibility of the tools, and the global layout.

Since November 2000, Eclipse is an open source project. This choice (from IBM) leads to a really open platform and to several other advantages:

- Reuse of the existing code: why rebuild something when it exists already and it works?

- Trust: code reuse needs trust on other developers and trust must be earned by developers providing good code.
- Confidence: providing all the source code of the platform to developers helps on gaining confidence.
- Quality: the code is reviewed by the community, enhancing the quality through a collaborative approach.
- Clarity: the code is easier to understand as it's delivered with the idea of being reused by the community.
- Longevity: offering the source code to developers the platform will ensure long term support.
- Flexibility: the possibility to build your own component and integrate it in the platform can provide a tailored tool.

In my project I used the Java perspective and the Eclipse Modeling Framework.

4.3 EMF

The Eclipse Modeling Framework is a Java-based environment for development of tools and other applications based on a structured model. With this framework it's possible to generate part of the code needed to develop applications starting from a model. The repetitive part is done automatically and a skeleton for the application is generated. Then a variable amount of hand-written code is needed to complete the task.

Almost every piece of software we write interacts with some data models. These models can be defined using UML, XML, Java or some other definition language. EMF can be used to extract this intrinsic model and to generate some of the implementation code. The starting project is based on the OMG's Meta Object Facility. MOF is an abstract language and a framework for specifying, constructing and managing meta-models. Metamodels can be viewed as models of a model. With metamodeling it's possible to describe different domain specific languages with the same formalism. As the MOF model was very large and complex, with the target of simplifing also the generated code, a drastic redesign of the MOF metamodel led to the Ecore. Ecore is an implementation of a subset of the features of MOF. After this simplification the people at OMG also realized that the MOF meta model was too complex and, with the contribution of EMF designer, they defined MOF 2.0. This redesign led to the Essential MOF (EMOF) and the Complete MOF (CMOF). Ecore is more or less aligned with EMOF. The objective of a clean and better generated code was achieved and now the framework is being used by a large community of developers[26].

Ecore is a subset of the MOF metamodel

The Model to Text Transformation (M2T) is a project built on top of EMF. With M2T it is possible to transform a model directly in code for every kind of programming languages. Xpand, Jet and Acceleo are the main project developed within M2T[1].

EMF provides developers with a set of tools to interact with the Ecore model. M2T offers a leverage to explore and build the applications straight from the model, transforming each element in a piece of code.

4.3.1 *Generator Workflow Component*

The Generator Workflow component is provided to perform the code generation. The code generation is achieved calling various component and running template files. The workflow file defines the configuration variables and the sequence of the execution. Input and output files are defined here. Also tools to beautify the code are invoked from the workflow.

The workflow file define the sequence of the operations

4.3.2 *Xpand*

Xpand is the language I used from the Model to Text project. This language is specialized on code generation based on EMF models. It supports the following main language features:

- Pluggable Type System
- Dynamic Dispatch of Functions
- Aspect Oriented Programming
- Rich Expressions (OCL-like but with Java-like syntax)

This language is used in templates to control the output generation. Templates are stored in files with the extension .xpt. A template file consists of any number of "IMPORT" statements, followed by any number of "EXTENSION" statements, followed by one or more "DEFINE" blocks (called definitions). With the "IMPORT" statement a name space (like UML) is imported, and unqualified name defined in the namespace could be used instead of fully qualified names. "EXTENSION" statements is used to import additional functions and and query operations from an Xtend file. This helps to keep the code clean and easy to read. "DEFINE" blocks qualify a template unit. The body of a template can contain a sequence of statements including any text or parameter.

The strength of this language is the modularity.

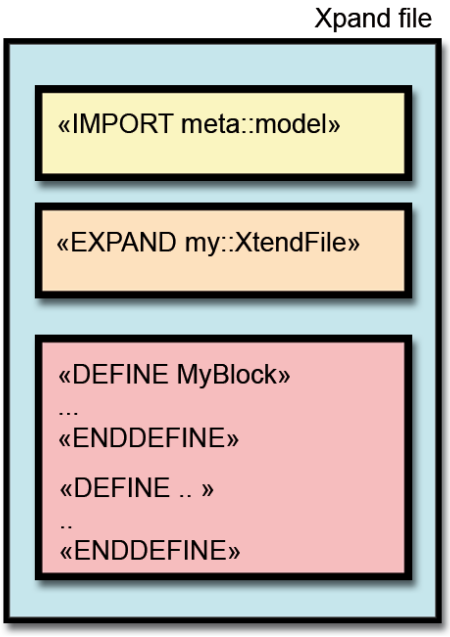


Figure 16: Structure of an Xpand file.

4.3.3 *Check*

Xpand also provides a language for checking the model consistency. It is a language to specify constraints that the model has to fulfill in order to be correct. This process is called "validation of the model". The language is very easy to understand and use. Basically, it is built around a simple expression syntax. Constraints specified in the Check language have to be stored in files with the file extension .chk. If the constraints check fails, two kind of actions could be taken:

- `WARNING`: a message is printed but the execution of the workflow is not stopped
- `ERROR`: the specified message is printed and the workflow execution is stopped

The Check language of Xpand provides also so called guard conditions. These conditions allow to apply a check constraint only to model elements that meet certain criteria, for example some actions could be taken only for a particular kind of states (like complex states or simple states). This is possible because the language is built on top of EMF. Loading the UML metamodel the language can identify the different entities of the model. More examples will be given in chapters related to the implementations of the code generator.

4.3.4 *Xtend*

The Xtend language provides the possibility to define rich libraries of independent operations and non-invasive metamodel extensions based on either Java methods or Xtend expressions. Those libraries can be referenced from all other textual languages that are based on the Xpand expressions framework. An Xtend file must have an .ext extension. Again, the use of these libraries allows to keep the code clean and readable.

4.4 APACHE SCXML AND THE APACHE ENGINE

The Apache Foundation is supporting SCXML through the Apache Commons project. Commons SCXML is a working implementation of a Java SCXML engine capable of executing a state machine defined in a SCXML document. The latest implementation of Commons SCXML is v.0.9 and is dated December 2008.

Apache Commons SCXML v.0.10 will be ready before the end of 2011

Without an engine capable of parsing and executing SCXML models these files would be useless. The Apache foundation and the open source community have developed a java based execution environment. The SCXML distribution provides a Java standalone class to test SCXML models from a command line. The choice of this implementation is dictated by the language. My work started developing an application for ACS and Java is the most used language on this platform. In addition the Apache project is well documented and supported by the community.

I set up the environment following the user guide from the Apache website [3] and tested it with the examples provided. From a command line events can be sent and interpreted and also variables can be set. The interface is rather basic but clear enough for testing purposes.

Part II

DEVELOPING THE SOLUTION

5 MODEL TRANSFORMATION

Contents

OMG definition of the UML standard is very formal. Every aspect is addressed in detail. Moving to a working implementation of the standard necessitates modifications and tricks bounded to implementations details (for example to the used language) or to choices taken during the early develop phase of the project. If on one side the standard, without any indication on "how" to implement the project, offers a solid starting point, on the other side, is unavoidable to introduce differences. This requires a research to find these differences (not always well documented). Working with something real, and that must work is very different from formulating principles without having the opportunity to verify it on the field.

So, the mapping of all these differences is not only a necessity, but should be the fundamental base for the subsequent work.

5.1 PROBLEMS

In this chapter I present a proposal for the mapping of UML State Machine to SCXML. It is not a complete mapping but it will focus on the features defined in the Generic State Machine Engine Software Requirement Specification[12]. This is an internal document from ESO that describes the functions that the GSME shall provide in order to build applications based on state machine models for the ACS or VLTSW platforms.

As the document states, the state machine engine shall support the following state machine features defined by the SCXML standard:

- composite states
- orthogonal states
- actions
- invoke (activities)
- guards
- shallow and deep history state
- initial and final pseudo-states
- IsIn

These features first introduced by D. Harel [20] proved to be important when modeling control applications for telescope domain. In particular composite states reduce the number of transitions and orthogonal states the number of states.

This was the first part of my work done in Garching bei München, home of the headquarters of ESO. The work was the starting point for the development of the UML2SCXML transformation tool.

5.2 A PROPOSAL FOR UML TO SCXML MAPPING

As stated in Chapter 3, UML is a general purpose language that aims to be a standard modeling language. State Chart XML [3] is a formal description on how to translate an UML statechart in a XML dialect. The execution environment provided from the Apache Foundation is a Java SCXML engine capable of executing a state machine defined in a SCXML document.

Working to develop a tool to translate a UML state machine model to a SCXML document brought to my attention some differences between the specifications, mainly due to the continuous changes in the standards and in the working draft, that are not always quickly implemented in the Apache engine. In addition the distance between the release date of the implementation and the latest working draft increases the number of differences. When the working draft from W3C will be ready to become a standard an update of the engine from the Apache Foundation will probably smooth all these details.

5.3 COMPARISON

I started my work comparing the UML specification from OMG[22] and the SCXML Working Draft[5] from W3C. Then I took the examples used in Harel's paper [20]. For each example in the article I tried to draw a UML diagram first, using MagicDraw. Then I wrote the corresponding code for the SCXML model strictly following the W3C's draft. After that I started testing the obtained model with the Apache SCXML engine. For each model I had to make small adjustments to fit the syntax expected from the engine. In addition, I adopted some workarounds to obtain desired features not directly supported with the Apache implementation.

5.3.1 *Simple state*

UML	State
SCXML	<state>
APACHE	<state> tag with a unique id attribute

Table 1: State

- A UML state is a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event. An object remains in a state for a finite amount of time.
- In SCXML <state> holds the representation of a generic state.
- A working SCXML model for the Apache implementation must have an unique id for each state.

In UML every element, including a state, is identified by a qualified name. With SCXML states are identified by the attribute ID, i.e. by the name of the states. This can cause some problems as when modeling a system, defining substates with the same name in different context is a common practice. While with an UML tool this is not a problem as the tool store it with a full qualified name, with an SCXML model this is an issue. Transforming the model this must be taken into account.

UML	Initial PseudoState
SCXML	initial as an attribute or <initial> tag
APACHE	initial attribute or <initial> tag (mandatory for complex states)

Table 2: Initial pseudostate

5.3.2 *Initial pseudostate*

- In UML diagrams the initial state is a pseudostate[1]. It has an arrow that points to the initial state. The used symbol is a small solid filled circle (Fig.17). With the same symbol you can specify the default substate of a complex state.

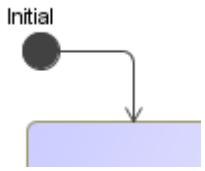


Figure 17: Initial pseudostate

- In SCXML there are two ways to indicate the first state of a state machine:
  - initial as an attribute of the <scxml> tag. The value must be the UNIQUE ID of the first state (if initial is not defined the default Initial State is the first defined state)
  - For complex states <initial> as child of a <state> tag. <initial> must enclose a conditionless <transition> with a descendent of the parent <state> as target. In complex states also the initial attribute can be used. This attribute is shorthand for an <initial> child with an unconditional transition. Again if an initial substate is not defined the first defined sub-state is taken as initial.
- With the Apache implementation an initial state MUST be defined as attribute of <scxml>. Also in complex state a default initial substate MUST be defined. If initial is defined as attribute the processed model code will be modified to obtain an equivalent <initial> tag.

1 Pseudostates do not have the properties of a full state and serve only as a connection point for transactions (but with some semantic value). Within the UML metamodel, Pseudostate is a sub-class of StateVertex.[22]

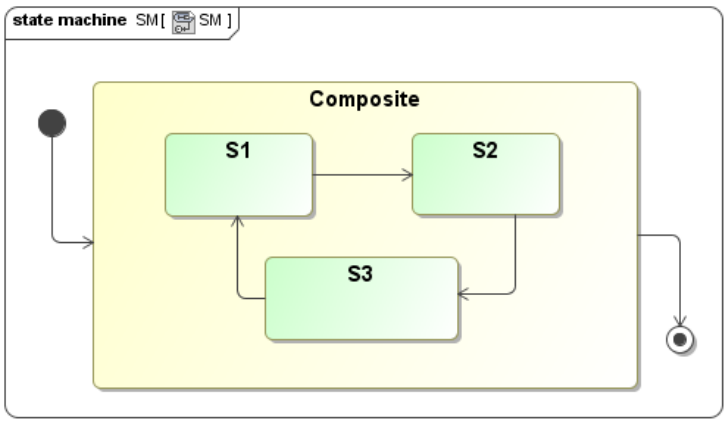


Figure 18: Which substate is the first entered?

If from a formal point of view the diagram in Fig.18 seems to be a proper UML diagram, on a working implementation this is not well defined. When entering the composite state is not possible to select the first substate entered. While the SCXML standard uses the first defined substate as a default state, this is not enough for the Apache implementation. A default substate must be explicitly defined using an initial pseudostate element inside the composite state.

5.3.3 *Final pseudostate*

UML	Final PseudoState
SCXML	<final> tag
APACHE	<final> tag with id attribute

Table 3: Final pseudostate

- In UML the Final State is a pseudostate. When it is reached the region is completed. When all the regions in the state machine are completed the entire state machine is completed. A Final Pseudostate has no exit transition. The symbol used is a circle surrounding a small solid circle (Fig.19).
- In SCXML <final> element is used to indicate the Final State of a compound state (as child of <state>) or of the entire state machine (as child of <scxml> element).
- To get a model working with the Apache engine <final> must have an id attribute with a valid id name as value.

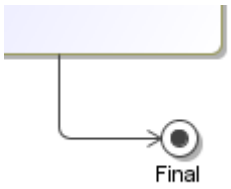


Figure 19: Final pseudostate

5.3.4 *Entry and exit actions*

UML	Entry / Exit action
SCXML	<onentry> / <onexit> tag
APACHE	<onentry> / <onexit> tag

Table 4: Entry an exit actions

- In a number of modeling situations,you might have the need to dispatch the same action whenever you enter a state, no matter which transition led you there. Similarly, leaving a state, you might want to dispatch the same action no matter which transition led you away. For doing this UML Entry and Exit Actions can be used.
- In SCXML Entry and Exit actions consist of actions performed as a part of the corresponding <onentry> or <onexit> element.
- No particular differences with the Apache implementation.

5.3.5 *Transition*

UML	Transition
SCXML	<transition> tag
APACHE	<transition> tag

Table 5: Transition

- In UML a transition is a relationship between two states indicating that an object in the first state will enter the second state when a specified event occurs and specified conditions are satisfied. On such a change of state, the transition is

said to fire. It is expressed in the form:

trigger[guard]/action

Trigger is the event that fires a transition . The guard is a boolean condition. The action is some behavior executed during the transition. All parts are optional. Every transition must have at least one source and one target. A "dangling" transition is not allowed.

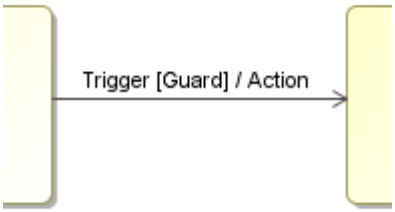


Figure 20: A transition with all its elements.

- In SCXML <transition> element is child of a <state> element and has event (valid values are SCXML events), condition (a boolean expression) and target (a state id) attributes. All attributes are optional. A transition without a target offers an event handler without the side-effect of leaving the recent state (i.e. an internal transition).

5.3.6 *Internal Transition*

UML	Internal transition
SCXML	<transition> with no target
APACHE	<transition> with no target

Table 6: Internal Transition

- Events can be handled without leaving a State. These internal transitions don't fire the state's entry and exit actions as state is not really left. Note that that's different from transitions whose target is the source state (*self transition*). In this case the state is left and re-entered and Exit and Entry Actions are triggered.
- In SCXML and Apache an internal transition is mapped as a *targetless transition*. Such transition acts as an event handler.

5.3.7 *Superstates and substates*

UML	Superstate
SCXML	<state> or <parallel> a default substate must be defined
Apache	<state> or <parallel>

Table 7: Superstates

A substate is a state that's nested inside another one. A Superstate that has substates is called a composite state. A composite state may contain either concurrent (orthogonal) or sequential (disjoint) substates.

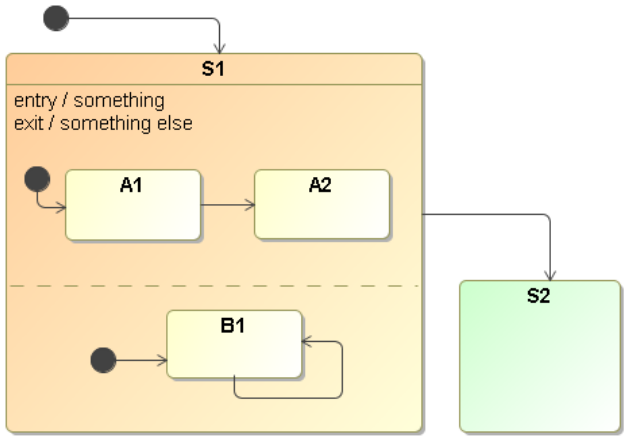


Figure 21: A configuration not supported by the Apache SCXML engine.

- In UML a substate is rendered just as a state inside another state. Substates may be nested to any level.
- In SCXML composite substates are defined with the <state> element inside a parent <state> element. The default initial state is identified with the <initial> element. If not present the default state is the first defined state. Orthogonal states are defined through the <parallel> element. Each orthogonal substate is identified with a <state> element.
- In Apache a default substate MUST be defined using <initial> for both orthogonal and composite states. In addition with <parallel> each orthogonal region of the state represents an auxiliary state wrapping the substates. For this reason each region should be identified with a name when modeling it in MagicDraw. In the current version of Apache Commons SCXML (v0.9) <parallel> can't contain an <history>,

<onentry> and <onexit> as childs. A <transition> element as child of <parallel> is ignored. A possible solution is to use a <state> wrapper element around the <parallel> to hold the deep history element, the Entry and Exit actions and the transition. Fig.22 shows a workaround for this problem.

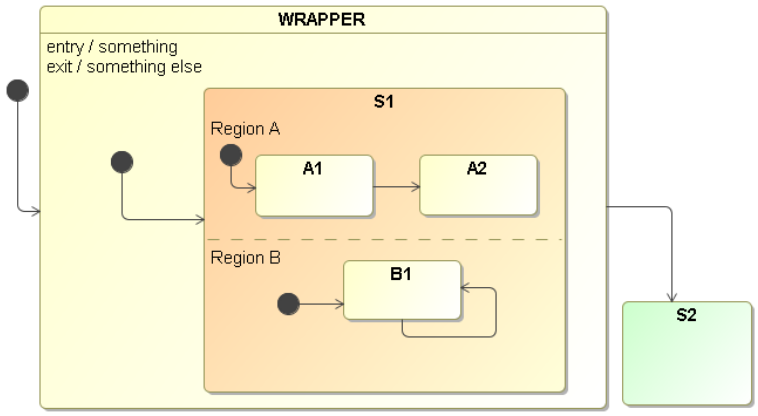


Figure 22: The workaround proposed.

UML	Substate
SCXML	<state> tag
APACHE	<state> tag

Table 8: Substates

Another limitation imposed by Apache Commons SCXML is about <invoke> element in composite states. A composite state should contain either one <parallel>, one <invoke> or any number of <state> children. That is <invoke> cannot be a child of a complex state unless the state has no substates (i.e. is a simple state).

5.3.8 *History pseudostate*

UML	History pseudostate
SCXML	<history> with a unique id
APACHE	<history> with a unique id and a conditionless <transition> element as child

Table 9: History pseudostate

- In UML diagrams a history pseudostate can remember the last active configuration of the object before leaving a composite state. There are two kind of history:
  - Shallow history can remember only the history of the immediate nested states
  - Deep history can remember the configurations of all the nested states, at any depth.

  A default state must be defined, to enter when there is no history for the state.
- In SCXML <history> is a child of <state>, must have a unique ID and a <transition> element as a child. This transition is conditionless and represents the default history state taken when there is no history for the parent state. If the history is shallow the target must be an immediate substate, otherwise can be any other descendant state.

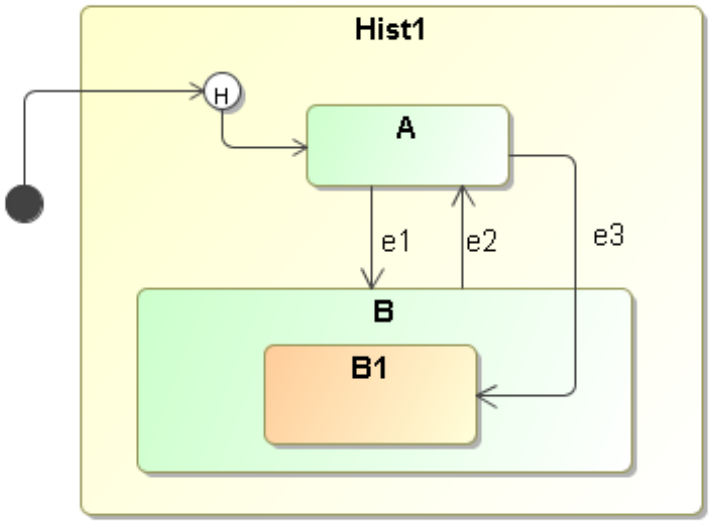


Figure 23: The history element with a default state.

5.3.9 *Activities*

- In UML an object, while in a state, can do some work. DoActivities are activities that can take a finite amount of time and can be interrupted by some events.
- In SCXML DoActivities are called with the <invoke> tag. This element starts an instance of an external service. With the <param> element data can be passed to the service.

UML	Activities
SCXML	<invoke> tag
APACHE	<invoke> tag

Table 10: Activities

- Also the Apache engine can handle DoActivities via the <invoke> element. An external state machine can be fired and the developer can let this external element deal with different implementations of activities. This allow to model a concurrent execution of the external service.

5.4 CUSTOM ACTIONS

The execution of an action represents some transformation or processing in the modeled system. As we saw in section 3.1 an action is supposed to be instantaneous. If there is the need to model some more complex behaviors, that require time to completed, a new state must be modeled. Inside the state a DoActivity will be used.

The SCXML specification describes a basic set of command implementing actions, useful for logging purposes or very simple executions. In many cases, this is not enough. The Apache implementation, as defined in the SCXML working draft, allows to define and use *Custom actions*. These are used to execute a specific implementation of arbitrary commands or code defined by the developer.

To transform this kind of actions and to generate a working SCXML model, first a fictitious namespace is needed (I used "*http://my.custom-actions.domain/CUSTOM*"). Then the custom tag, named with the same name of the custom action, is created in the proper context.

5.5 SUMMARY

Translating a model defined with UML statechart to an Apache working SCXML model is not trivial. The developer must be careful with small implementation details and in some cases special workarounds must be taken. The Apache SCXML engine provide an execution environment that can support all the features defined in the GSME Software Requirements Specification[12]. With this mapping proposal these features are interpreted correctly. It was very interesting to see how things change from the standard to a real and working implementation of the engine. Most of the

The mapping process is essential: there are many differences between the standards

problems are due to the latest modifications of the working draft and to the old implementation of the Apache engine. In addition, the fact that the W3C specifications are still in the form of a working draft keeps away the developers from investing too much time in making changes that may be not definitive. Certainly when the working draft will take the form of a recommendation a new impulse will hit also the development process of the Apache engine. Hopefully this will smooth a lot of differences and make easier the adoption of this interesting framework.

6 THE CODE GENERATOR

Contents

6.1 THE GENERIC STATE MACHINE ENGINE ARCHITECTURE

A Generic State Machine Engine is simply a SM Engine that allows generating from the same SM Model, Software Component skeletons that can be compiled on different Software Platforms. The ability to support multiple platforms increases SM Model reuse and therefore reduces development and test time.

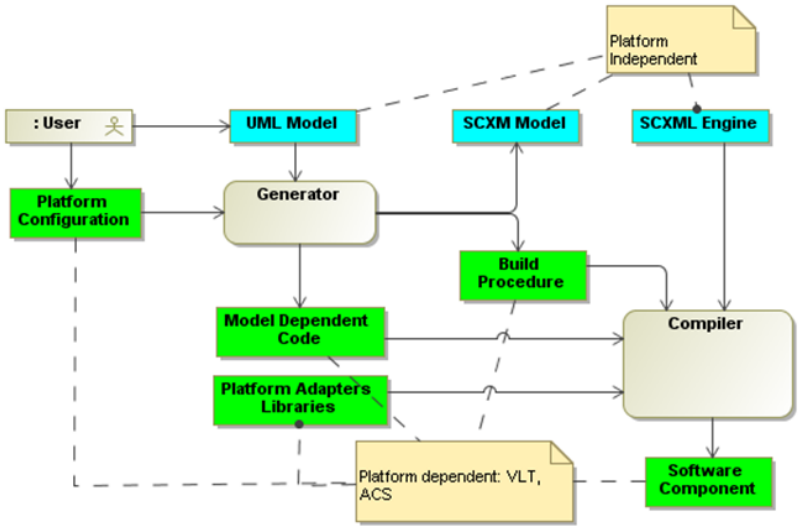


Figure 24: Generic State Machine Engine Data Flow[12].

The main goal of the Generic State Machine Engine project is to develop a tool that can facilitate the creation of state machine based Software Components for the VLT and ACS Software Platforms. The architecture is built on top of a Java SCXML Engine which executes SCXML Models. In the next future also a C++ implementation of the engine is planned. A Generator takes as input the configuration on the target platform (Platform Configuration) and the UML Model to generate: all the Model Dependent Code, the Build Procedure (for example the Makefile) and the SCXML Model. Examples of Model Dependent Code are: the code to propagate the events to the SCXML Engine, Actions, Guards, and Activities skeletons. Platform Adapters Libraries, used to provide a standard interface for services like logging, error handling, messaging etc. (i.e. to abstract the Software Platform specific services), are compiled together with the SCXML Engine, and the Model Dependent Code to build the Software Component.

6.1.1 *Model Independent State Machine Engine*

A State Machine Engine is an executable implementation of a State Machine Model. State Machine Engines can be developed using different approaches: from the simple "switch" statements, or the State Design Pattern[1], up to building an interpreter of State Machine Models. In [12] the definitions of "Model Independent State Machine Engine" and "Model Dependent State Machine Engine" have been introduced to classify two different types of State Machine Engines: the ones whose implementation does not change when a different State Machine Model is executed and the ones whose implementation does change.

In a Model Dependent SM Engine the model (states, transitions, etc.) is hard coded in the implementation language and therefore defined at compile-time. Examples are:

- SM Engine based on "switch" statements
- SM Engine based on State Pattern

Disadvantages of this approach are:

- SM model must be defined at compile-time
- The size of the application is usually larger since it includes the logic of the SM

1 From [17]: the state pattern is a behavioral software design pattern, also known as the objects for states pattern. This pattern is used in computer programming to represent the state of an object. This is a clean way for an object to partially change its type at runtime

- SM Model and source code may easily get out-of-synch if the code is not generated from the model every time it is compiled

Advantages of this approach are:

- Performances and memory consumption can be optimized (since the State Machine Engine is build specifically for that given model)
- There is no need of the model once the application is built (or generated). In many cases the model can be reconstructed from the application if/when needed.

In a Model Independent SM Engine the model is stored in memory. The SM Engine provides APIs to create states, transitions, etc and build up an in-memory representation of the model. The algorithm works with the in-memory representation of the model to decide which transitions to take, which states to visit, which actions to execute.

Disadvantages of this approach:

- Performances and memory consumptions may not be as good as the one of Model Dependent SM Engines
- The model is needed for the execution of the application
- No type-safe invocation of actions/activities

Advantages of this approach:

- SM Engine can be reused by the applications
  - Applications are smaller
  - Similar advantages of the libraries
- SM Model can be defined at:
  - Compile-time using the APIs provided by the SM Engine to build up the SM model. Development of actions, data, events is done using a compiled programming language.
  - Run-time using a parser which reads a SM Model Representation and calls the SM Engine APIs to instantiate the model. Note that in a fully Run-time SM Engine also data, actions and events are defined in the model and interpreted (i.e. the action language is interpreted). This approach provides the flexibility of changing at runtime the complete behavior of the application without the need of recompiling the application.

– Partially at compile time and partially at run-time: data, actions and events are defined at compile time while states and transitions are loaded and interpreted at run-time.

The decision of selecting which approach is the best is similar to evaluating the advantages and disadvantages of interpreted vs. compiled programming languages. The mix case, where actions and data are compiled while the state machine logic (states and transitions) are interpreted, is an interesting alternative that minimizes the disadvantages of the compiled programming languages and introduces some of the advantages of the interpreted languages: there is no need to recompile if the SM logic changes. Recompilation is needed only if data/actions change which could be a positive effect since it forces the type checking.

The ability of quickly modifying the application behavior (i.e. the SM logic) without the need for recompiling the code becomes an important feature in all the scenarios where last minute changes in the requirements have to be quickly implemented and tested. The need of a Model Independent SM Engine is one of the lessons learned in the development of previous projects at ESO

6.2 IMPLEMENTATION

To develop the code generator, as stated in the first part of this document, I used:

- MagicDraw to draw and export UML models (some tests were made also with Papyrus, a UML design tool now part of the eclipse software)
- Eclipse and the Eclipse Modeling Framework to load and manage models
- XTEND, XPAND and CHECK languages, available on top of EMF, useful to browse and analyze XMI models and to generate the code.
- Apache Commons SCXML Standalone Engine, a java class to test models

Here will follow a description of the whole process, from creating a model with MagicDraw to the testing with the Apache engine.

6.2.1 *Designing a model with MagicDraw*

First of all a UML state machine diagram is needed. With MagicDraw it's easy to draw this kind of diagram from scratch. I made

this list of properties while comparing the specifications of the standards with the Apache implementation of the SCXML engine. In a state machine diagram:

- a state machines must have a name
- a state machines must have inner states
- a state machine must have an initial state
- every state must have a name
- if present, a final state must have a name
- in composite state, a default state is needed
- in orthogonal states, every region must have a name

To work with the current version of the Apache engine some other rules must be respected:

- Entry actions are not allowed in orthogonal states
- Exit actions are not allowed in orthogonal states
- Transitions from orthogonal states are ignored

These limitations are only due to the current implementation of the engine (v.0.9) and may change with a new version.

In any case my tool will check the models and see if they fulfill these rules. Following these few guidelines a state machine diagrams can be exported in Ecore, a format that can be handled with eclipse. To do this, save the diagram and from the File menu select Export To -> Eclipse UML (v2.x) XMI File.

6.2.2 *Transforming the model*

On the activity diagram of Fig.25 are defined the steps the code generator will do to produce the application. The transformation tool will be initialized with the platform configuration details. The UML metamodel is loaded and the tool takes as input an ecore model (or a MagicDraw model exported as XMI for Eclipse). It will explore it, checking for constraints as specified in the Check file called *constraints.chk*. If the model checking is ok, the output directory is cleaned and the transformation will start. An XML file will be created for every state machine in the model. These XML are the SCXML models of the state machines. These files go through the beautifiers and the final output files are generated.

The file constraints.chk contains all the constraints I discovered and described in the mapping phase

The expand file *FSMScxml.xpt* defines how the model is explored.

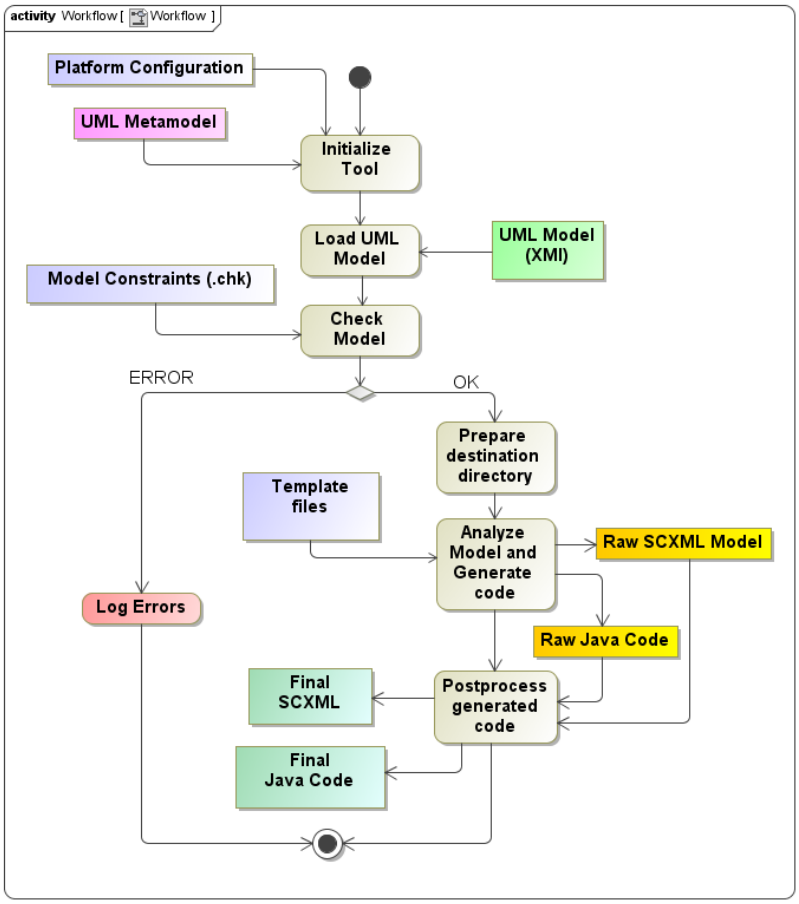


Figure 25: This activity diagram represents the workflow of the code generator.

- First of all the XML file is created and the headers are written. In this block the tool will look for the initial state of the state machine. This is a mandatory attribute for the actual implementation of the Apache engine.
- Then the top states (the most external states of the FSM) are explored in alphabetical order. This block tests if the state is a simple state or if it is a composite one and calls the respective blocks.
  - If it is a simple state the tool will test if it is also a final state (and in this case write a <final> element in the SCXML file) and it will explore actions and transitions.
  - For composite states a first test for orthogonal states is done, then it looks for the initial state (if the composite state is not empty), for child states and at the end for history pseudostates. Then, again, it explores actions and transitions.
- When all the states are explored the SCXML file is closed.

With the Invoke block, if a do-activity is found in the Explore-Actions block, an invoke element is created. The invoke tag has an id attribute that is set equal to the name of the do-activity. With the code generated as Java, the invoke element will call a Java class with the same name of the do-activity. When called this class will start a new Java thread running in parallel with the main application.

Let's see the implementation in details.

6.2.2.1 *MWE workflow*

Using a declarative XML-based language I have written the workflow file. This file specifies step by step the execution of the different modules. The workflow file is used by the Modeling Workflow Engine to explore and analyze the models.

Listing 2: Workflow file

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<workflow>

  <!-- Setup URIs -->
  <property name="modelFileURI" value="./examples/
      MasterComponent/MasterComponent.uml"/>
  <property name="ouputFolderURI" value="./src-gen" />

  <!-- Setup path to platform. Usually this is the workspace
      location. -->
  <bean class="org.eclipse.emf.mwe.utils.StandaloneSetup">
   <platformUri value="." />
  </bean>

  <!-- Setup UML2 support -->
  <!-- initializes resourcemaps, urimaps, etc. -->
  <bean class="org.eclipse.xtend.typesystem.uml2.Setup"
      standardUML2Setup="true"/>

  <!-- Reads the UML model and stores it into slot named '
      mymodel' -->
  <component class="org.eclipse.emf.mwe.utils.Reader">
    <uri value="${modelFileURI}" />
    <modelSlot value="mymodel" />
  </component>

  <!-- Checking model requirements-->
  <component class="org.eclipse.xtend.check.CheckComponent">
    <metaModel id="mm" class="org.eclipse.xtend.typesystem.
        uml2.UML2MetaModel"/>
    <checkFile value="constraints"/>
    <emfAllChildrenSlot value="mymodel"/>
  </component>
```

```
  <!--  Cleaning the room. Delete all the files on the output
       dir -->
  <component id="dirCleaner"
    class="org.eclipse.emf.mwe.utils.DirectoryCleaner"
    directory="${ouputFolderURI}"/>

  <!--  generate code -->
  <component class="org.eclipse.xpand2.Generator"
      skipOnErrors="false">
    <metaModel class="org.eclipse.xtend.typesystem.uml2.UML2
        MetaModel"/>
    <metaModel class="org.eclipse.xtend.typesystem.emf.
        EmfRegistryMetaModel"/>
    <expand value="templates::Root::Root FOR mymodel" />
    <outlet path="${ouputFolderURI}">
    <!-- Code beautifier -->
      <postprocessor class="org.fornax.utilities.formatter.
          xml.XmlFormatter"/>
      <postprocessor class="org.eclipse.xpand2.output.
          JavaBeautifier" />
    </outlet>
  </component>

</workflow>
```

In the workflow first of all, the variable are initialized and the resources are loaded. Then every component is called with appropriate arguments. The relevant components are:

- the CheckComponent, calling the *constraints.chk* file
- the Generator, using the template files to explore the models

6.2.3 *Check*

The first component loaded is the CheckComponent. With the *constraints.chk* I have specified some properties that the model must fulfill. I defined these properties as described in Chapter 5.

Listing 3: constraints.chk file

```
import uml;

context StateMachine ERROR "StateMachine must have a name" :
        name.length != 0;

context StateMachine ERROR "StateMachine must have an initial
     state." :
        allOwnedElements().typeSelect(Pseudostate).select(e|(
            e.kind.toString() == "initial")&& (e.container.
            owner == this)).size > 0;
```

```
context StateMachine WARNING "StateMachine must have inner
    states" :
        allOwnedElements().typeSelect(State).size > 0;

context State ERROR "State must have a name" :
        name.length != 0;

context State if this.metaType.name == "uml::FinalState"
    WARNING "Final states must have a name" :
        name.length != 0;

context State if isOrthogonal() ERROR "Orthogonal Regions
    must have a name" :
        allOwnedElements().typeSelect(Region).select(e|(e.
            owner == this)&&(e.name.length == 0)).size == 0;

context State if isOrthogonal() WARNING "Initial state is
    missing in orthogonal region" :
        this.allOwnedElements().typeSelect(Pseudostate).
            select(e|e.kind.toString() == "initial").select(e
            |this.ownedElement.contains(e.container)).size ==
             this.ownedElement.typeSelect(Region).size ;

context State if ( isComposite() && !allOwnedElements().
    typeSelect(State).select(e|!(e.name == "Unavailable") &&
    !(e.metaType.name == "uml::FinalState") ).isEmpty ) ERROR
     "Composite States must have a default initial state" :
        allOwnedElements().typeSelect(Pseudostate).select(e|(
            e.kind.toString() == "initial")&&(e.container.
            owner == this )).size > 0;

context State ERROR "Name of states must be unique":
        !this.allOwnedElements().typeSelect(State).name.
            contains(this.name);

context State if isOrthogonal() WARNING "Entry action not
    allowed on orthogonal states":
        (this.entry.name.toString() == "null");

context State if isOrthogonal() WARNING "Exit action not
    allowed on orthogonal states":
        (this.exit.name.toString() == "null");

context State if isOrthogonal() WARNING "Transitions from
    orthogonal states are ignored":
        this.outgoing.size == 0;

context Pseudostate if kind.toString().contains("History")
    ERROR "History state must have a name":
        this.name.length !=0;

context Pseudostate if kind.toString().contains("History")
    ERROR "History state must have a default state":
```

```
        this.outgoing.toList().size !=0;

context Pseudostate if kind.toString().contains("initial")
    ERROR "Every initial state must have an outgoing
    transition":
        !this.outgoing.isEmpty;
```

First the UML metamodel is imported. Then for every element to check, a context is specified. This means that the check is not made with a simple textual analysis.

Once loaded the metamodel the Check language "*understand the model*", recognizing the different elements of the model itself. For every kind of element a different behavior can be defined and different constraint can be checked.

Two types of action can be taken:

- a *warning*, that prints a message
- an *error*, that prints a message and stops the processing of the model

6.2.4 *Xpand*

After the check procedure, the Xpand component starts to follow the *Root.xpt* template.

Listing 4: Root.xpt file

```
«DEFINE Root FOR uml::Model»
  «REM» Foreach element of type StateMachine expand Root from
       FSMScxml «ENDREM»
  «EXPAND FSMScxml::Root FOREACH allOwnedElements().
      typeSelect(uml::StateMachine)»
«ENDDEFINE»
```

For each state machine the element Root from *FSMScxml.xpt* is expanded. Also with Xpand files a context can be defined, specifying different action for different kind of elements. The "EXPAND" statement "expands" another DEFINE block (in a separate variable context) and inserts its output at the current location.

Listing 5: The Root block in FSMSCxml.xpt

```
«DEFINE Root FOR StateMachine»
«REM» Generate SCXML Model «ENDREM»
«FILE this.name+".xml"»
        «REM» Write the headers «ENDREM»
        «EXPAND SCXMLcodeScxmlCode::Headers»

        «REM» Select TopStates and explore «ENDREM»
```

```
        «FOREACH listStates().select(e|e.isTopState()).sortBy
            (e|e.name) AS st»
                «EXPAND ExploreState FOR st»
        «ENDFOREACH»

        «REM» Outer FinalStates «ENDREM»
        «EXPAND ScxmlCode::FinalState»

«EXPAND ScxmlCode::CloseScxml»
«ENDFILE»
```

The tool creates a file called as the state machine, writes the headers defined in a block later, and starts to browse the model from the most external states. For each state of the machine the block ExploreState is expanded. Then the outer final states are hooked and the corresponding tag is generated.

Listing 6: The ExploreState block in FSMSCxml.xpt

```
«REM» Explore Generic States «ENDREM»
«DEFINE ExploreState FOR State»
        «IF this.isSimple()»
                «EXPAND ExploreSimpleState»
        «ELSE»
                «IF this.isComposite()»
                        «EXPAND ExploreCompState»
                «ENDIF»
        «ENDIF»
«ENDDEFINE»
```

With this block states are divided into simple and composite and the suitable block is expanded. With simple states it checks for the name, for actions and transitions. With complex states the block is a bit more tricky:

Listing 7: The ExploreCompState block in FSMSCxml.xpt

```
«REM» Explore Complex States «ENDREM»
«DEFINE ExploreCompState FOR State»
        «IF this.isOrthogonal()»
                «EXPAND ExploreOrthState»
        «ELSEIF !this.listStates().isEmpty»
                «EXPAND ScxmlCode::StateName»
                <initial><transition target="«this.
                    getInitialState().getFullName()»"/></
                    initial>
                «FOREACH this.listCompChilds() AS cc»
                        «EXPAND ExploreCompState FOR cc»
                «ENDFOREACH»
                «FOREACH this.listSimpleChilds() AS sc»
                        «EXPAND ExploreSimpleState FOR sc»
                «ENDFOREACH»
                «IF this.hasHistory()»
```

```
                        «EXPAND ExploreHistory»
                «ENDIF»
                «EXPAND ExploreActions»
                «EXPAND ExploreTransitions»
                «EXPAND ScxmlCode::CloseState»
        «ELSE»
                «EXPAND ExploreSimpleState»
        «ENDIF»
«ENDDEFINE»
```

First it checks for orthogonal states and calls the appropriate block, then if it's not an orthogonal state the tool looks for the default state and writes the required initial tag. Each child state is distinguished between simple and composite, and again the suitable block is called recursively.

Listing 8: The ExploreOrthState block in FSMSCxml.xpt

```
«REM» Explore Orthogonal States «ENDREM»
«REM» Parallel Regions MUST have a nome in the ecore model
    «ENDREM»
«REM» Parallel Regions are mapped as Complex states: a
    default state is needed «ENDREM»
«DEFINE ExploreOrthState FOR State»
        <parallel id="«this.getFullName()»">
                «FOREACH this.allOwnedElements().typeSelect(
                    Region).select(e|(e.owner == this)) AS
                    oreg»
                <state id="«oreg.getFullName()»">
                        <initial><transition target="«oreg.
                            getInitialState().getFullName()»
                            "/></initial>
                «FOREACH oreg.allOwnedElements().typeSelect(
                    State).select(e|(e.owner == oreg)) AS
                    ost»
                        «EXPAND ExploreState FOR ost»
                «ENDFOREACH»
                «EXPAND ScxmlCode::CloseState»
        «ENDFOREACH»
        «IF this.hasHistory()»
                «EXPAND ExploreHistory»
        «ENDIF»
        «EXPAND ExploreActions»
        «EXPAND ExploreTransitions»
        </parallel>
«ENDDEFINE»
```

With orthogonal states parallel regions are mapped as complex states and a default initial state is needed for every region.

Then every inner state is explored recursively again. Even if the engine does not allow actions and transitions from orthogonal states the tool map also those. In case they are present a warning in the checking phase will inform the developer.

Listing 9: The ExploreActions block in FSMSCxml.xpt

```
«REM» Explore Actions and Activities «ENDREM»
«DEFINE ExploreActions FOR State»
        «IF this.hasOnEntry()»
                «EXPAND ScxmlCode::CAEntry»
                «EXPAND CustomActionCode::JCAEntry»
        «ENDIF»
        «IF this.hasDoActions()»
                «EXPAND ScxmlCode::Invoke»
                «EXPAND ActivityThreadCode::InvActivity»
        «ENDIF»
        «IF this.hasOnExit()»
                «EXPAND ScxmlCode::CAExit»
                «EXPAND CustomActionCode::JCAExit»
        «ENDIF»
«ENDDEFINE»
```

Entry actions are now explored. If the state has an entry action two blocks are expanded: one to generate the corresponding SCXML tag with a custom action, and the other to generate the file containing the Java implementation of the custom action. If the state has a DoActivities the invoke tag is generated and the Java code to generate a new thread is written. Finally the exit actions (if present) are explored in the same way as the entry actions.

The Xtend language has provided me the possibility to define rich libraries of independent operations and metamodel extensions based on Java methods. The defined extensions are used to generate the output.

6.2.5 *Xtend*

The *scxmlutil.ext* file contains all the needed methods to build the scxml file.

Listing 10: Some functions from ScxmlUtil.ext file

```
//Check if State is a FinalState
boolean isFinalState( State this ):
        if isComposite
                then false
        else
                if (this.metaType.name == "uml::FinalState")
                        then true
                else false;

// If this State is the topmost, returns true.
boolean isTopState( State this ):
        if( this.container.owner == containingStateMachine()
            )
                then true
```

```
        else false;

boolean isTopState( Pseudostate this ):
        if( this.container.owner == containingStateMachine()
            )
                then true
        else false;

boolean isTopState( Region this ):
        if( this.owner == containingStateMachine() )
                then true
        else false;

// Lists composite child states
List[State] listCompChilds( StateMachine this ):
        allOwnedElements().typeSelect(State).select(e|(e.
            isComposite() && (e.parentState() == this))) ;

List[State] listCompChilds( State this ):
        allOwnedElements().typeSelect(State).select(e|(e.
            isComposite() && (e.parentState() == this))) ;

//Lists simple child States
List[State] listSimpleChilds( StateMachine this):
        allOwnedElements().typeSelect(State).select(e|(!e.
            isComposite) && (e.parentState() == this)).
            removeAll(this.allOwnedElements().typeSelect(
            State).select(e|e.metaType == "uml::FinalState"))
             ;

List[State] listSimpleChilds( State this):
        allOwnedElements().typeSelect(State).select(e|(!e.
            isComposite) && (e.parentState() == this)).
            removeAll(this.allOwnedElements().typeSelect(
            State).select(e|e.metaType == "uml::FinalState"))
             ;

// Get all Transition owned by this State
List[Transition] getTransitions( State this ):
        allOwnedElements().typeSelect(Transition);

//Check if the state has entry or exit actions
boolean hasOnEntry( State this ):
        !(this.entry.name.toString() == "null");

boolean hasOnExit( State this ):
        !(this.exit.name.toString() == "null");

//Check if State has activities
boolean hasDoActions( State this ):
        !(this.doActivity.name.toString() == "null" );

//Check if transition has a guard
```

```
boolean hasCond( Transition this ):
        !(this.guard.name.toString() == "null");

//Check if transitions has an event that triggers itself
boolean hasEvent( Transition this ):
        !(this.trigger.isEmpty);

//Check if transition has target
boolean hasTarget( Transition this ):
        !(this.target.name.toString() == "null");

//Check if transition has Actions
boolean hasAction( Transition this ):
        !(this.ownedElement.typeSelect(Activity).isEmpty);
```

6.3 A RUNNING EXAMPLE: MASTERCOMPONENT

One of the of the objective of my work, was to be able to generate the code for a working MasterComponent from the corresponding UML model. This state machine use many of the supported features of the code generator and was the ideal test for the tool.

The Master Component is an ACS component that represents an ALMA subsystem ('subsystem' in its technical meaning) toward the rest of the ALMA software system. It manages life-cycle details and provides information on the current subsystem state and a number of modes.

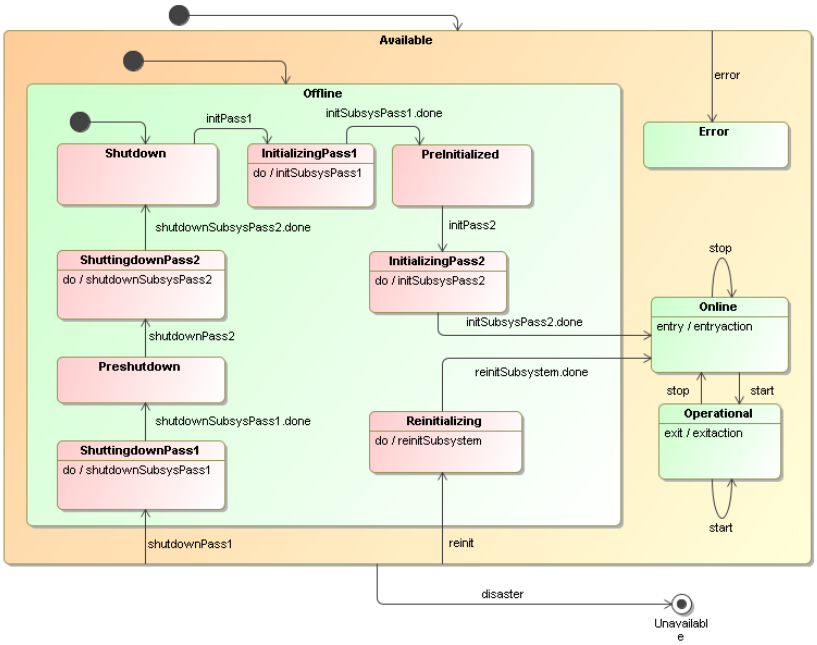


Figure 26: The MasterComponent State Machine.

There are two logical high level states:

- Available: When a Master Component can be contacted and is able to receive commands it is Available
- Unavailable: When an attempt to contact a Master Component fails with an exception, it is Unavailable. This happen if the system has no power or if the software has never been started or crashed.

Available and Unavailable are not real states, in the sense that they are not really coded in the State Machine. The Executive or another client will declare that a component is Unavailable if it fails to get in touch with it, Available in all other cases. Whenever a Master Component is instantiated, it will be New and the constructor will leave it in an Available substate when "construction completed" or it will die in case of disastrous failure and become Unavailable

6.3.1 *Available substates*

These are the real states that are represented in the state machine of a Master Component

- Offline is the state when the component exists but has not been initialized. There are three substates:
  - Shutdown: The component goes in this state upon initial creation and upon completion of the shutdown procedure. Here the component can be contacted but is not able to perform any activity.
  - Initializing: When receiving the init() command the component tries to initialize itself. This can take a sizable amount of time and therefore it goes into a transient Initializing substate. When "init completed" successfully, it transitions to the Online state. Notice that the transition from Shutdown to Initializing implies a "hard" initialization, where everything is reset, while in any other case the init() command is only performing a "soft" initialization. For example hardware is not reset in this case.
  - Shuttingdown: The component goes into this transient state upon receiving the shutdown() command. A shutdown can require a sizable amount of time, and therefore the component remains in this substate until completed and then automatically transitions to Shutdown
- Online: In the Online state the subsystem is able to receive commands and perform engineering work, but it is not

fully functional for science operation. The start() command would take it into the Operational state

- Operational: In the Operational state the subsystem is fully functional and able to perform science. The stop() command would take it into the Online state.
- Error: When an error occurs that cannot be immediately automatically recovered (but the component is not physically destroyed and it is still possible to interact with it), the Component goes into the Error state. A typical case is when there are hardware problems with devices in the subsystem or with resources like disk space. In such cases normally an intervention from the operator is required. After the operator has fixed the problem, he/she can issue an init() or shutdown() -> init() sequence to soft/hard initialize the Master Component.

In principle a Master Component could automatically go from Online to Operational upon init (as requested by Pipeline) if the two states are effectively identical.

6.3.2 *Substates of Online and Operational*

Applications can define context specific substates for Online and Operational. Executive would not make general assumptions on such substates. They will be eventually used for context specific purposes that require knowledge of the details of the state machine of a specific subsystem. It would be in any case better to use common names for equivalent substates in different subsystems. Here are some examples:

- OK: everything is fine and alive
- IDLE / BUSY: as an alternative to OK for subsystems that can be just IDLE waiting for commands or are BUSY and therefore cannot perform other actions in parallel. For "intermediate" situation, the usage of OK is usually better.
- RECOVERING: when a subsystem is recovering from a temporary bad situation/error

6.3.3 *A few modifications*

Starting from the original concept of the MasterComponent as developed in ESO, I made some changes to the model. First of all I added a default state to all the composite states, following the guidelines I have emphasized in the first part of my work. These initial states are needed to identify the default starting substate

in a composite state. Then I added events to the transitions. Among these, there are some special ones. These events are called "*.done events*" and they are generated in the state machine when a DoActivity, started in the substates, has finished his job. This is done every time an external service is called via the <invoke> tag, as stated in the SCXML standard. I also added some entry and exit actions for testing purposes.

6.3.4 *The generated files*

Once the files are generated developers have only to write the code for actions and activities

The generator will produce a Java application. The following files are created starting from the model:

- *StatemachineName.xml*: the SCXML model
- *Application.java*: the main class containing informations and data for the executor
- *JavaInvoker.java*: the class implementing Invoker. For each activity an instance of JavaInvoker is created. This will launch the corresponding thread with the implementation of the activity through the start() method.
- *ActivityThread.java*: a generic class for activity threads. In this class there is the implementation of the generic stop() method for the activity threads
- *MyTreadActivityName.java*: for each activity a specific class is generated. These classes extend ActivityThread.java and must be filled with the correct implementation of the Activities. When the activity is finished a special event (*.invoke.done) is sent to the parent state. This must be the last generated event from the thread.
- *StatemachineNameActionList.java*: create the list of Custom Actions used in <onentry> and <onexit> blocks
- *CAActionName.java*: a class for each custom action containing the implementation of the custom action
- *CustomActionListMap.txt* and *DoActivitiesMap.txt*: two text files with the list of Activities and Actions and the names of the respective Java classes. To be used in future for a different mapping (each action or activity could be mapped as a method of a class).

The generated files must be filled with the desired implementation of actions and activities. Developpers have only to write the desired implementations for actions and activities. Only the files *MyTreadActivityName.java* and *CAActionName.java* have to be modified with hand-written code.

To run the generated application Application.java must be started. The application will launch the SCXMLExecutor and on the console window a very basic interface will log events, current states and transitions. Events can be fired from the console. The application use the SCXML model to react to the events.

The application behave as expected: the SCXML model is correctly interpreted. In addition DoActivities are implemented as threads, to model the concurrency in the execution of the processes. If the state where the DoActivity is called is left the activity thread is stopped as as provided in the standards.

7

CONCLUSIONS AND FUTURE WORK

Contents

I developed this project working at the headquarters of the European Southern Observatory, located in Garching bei München, Germany. Here I worked for the Software Development Division in a team of 20 people.

In this work I addressed the following topics:

- A first part where I gave an introduction on the Model Driven Development principles and on the statecharts formalism, to give to the reader the basic knowledge needed to better understand my work.
- A description of the standards adopted to model state machines in this project (UML and SCXML).
- A brief introduction on the available tools used to edit and manage models and useful for the automatic code generation.
- The mapping proposal I developed and used to transform a UML model in the corresponding SCXML model. A formal mapping for this kind of transformation has never been presented in the literature.
- The design and the implementation of a code generator capable of producing code for an application. The logic of the application is based on the state machine described using an SCXML model.

7.1 TARGETS ACHIEVED

With my work I achieved all the prefixed objectives. First of all, staring from a UML state machine model, the corresponding SCXML model is properly generated. All the features of the model are translated with using the correct SCXML syntax.

The skeleton of the application is generated. The Apache SCXML Engine is enclosed in the application and is used to parse the SCXML model. The model encloses the logic of the

application and is used by the SCXML engine to react to the external events.

Developers have to fill the generated skeletons with hand-written code. This code tie the application to the implementation details.

7.2 WHAT TO DO NEXT?

Once the design of the application is defined, is time to focus on the refactoring of the code. The application should be modified to run as an ACS component. People at ESO are now working to integrate my project in the ACS Code Generator, a tool capable of generate Java application for the ALMA platform.

Here follow possible developments of this work:

- For the moment the generated code is for a Java application. Next step may be to generate in a different language, like C++, and for a different platform like the VLT software platform.
- The automatic regression testing procedure of the code should be implemented to check that the application behave properly. A model for this purpose should be designed and a test script prepared.
- Another useful feature should be a simulation toolkit: a visual representation of the state machine with an highlighting of the flow could help to monitor the application and to spot possible errors on the model.

Part III

APPENDIX

A

CODE LISTINGS

A.1 THE CODE GENERATOR

The complete listings of the developed tool could be downloaded from the page of the project on Google Code. The home page of the project can be reached browsing following URL:

http://acsccg.googlecode.com

Everyone can download the code using a svn client. Use this command to anonymously check out the latest project source code:

```
# Non-members may check out a read-only
# working copy anonymously over HTTP.
svn checkout http://acsccg.googlecode.com/svn/trunk/
acsccg-read-only
```

The *gsmecg* folder contains the full Eclipse project, with the examples used to define the mapping and the test models.

Here follows a description of the structure of the project. *gsmecg* is the main folder of my project.

- *src*: this folder contains all the source code. The file *workflow.mwe* with the sequence of the operations to be executed, the file *constraints.chk* contains all the rules for the models as specified in the mapping phase.
  - In the *templates* folder you can find all the template files used to explore models. The file are written using The Xtend and Xpand languages.
- *lib*: in here are all the libraries needed to run the code generator and the generated application.
- *src-gen*: this folder is used to store the generated application.
- *examples*: here you can find all the examples used to develop the code generator. In every folder there is the MagicDraw file (*.mzip*) and the exported model in XMI format.

BIBLIOGRAPHY

[1] Model to text (m2t) eclipse project. URL http://www.eclipse.org/modeling/m2t/.

[2] Qt labs website. URL http://developer.qt.nokia.com.

[3] Commons SCXML, July 2010. URL http://commons.apache.org/scxml/.

[4] UML specification, May 2010. URL http://www.omg.org/spec/UML/2.3/.

[5] State Chart XML (SCXML): State machine notation for control abstraction, December 2010. URL http://www.w3.org/TR/2010/WD-scxml-20101216/.

[6] Astronet website, 2011. URL http://www.astronet-eu.org/.

[7] The atacama large millimeter/submillimeter array, March 2011. URL http://www.eso.org/sci/facilities/alma/.

[8] The european extremely large telescope ("e-elt") project, March 2011. URL http://www.eso.org/sci/facilities/eelt/.

[9] La silla paranal observatory, March 2011. URL http://www.eso.org/sci/facilities/lpo/.

[10] Eso website, 2011. URL http://www.eso.org/.

[11] Eso - the very large telescope, 2011. URL http://www.eso.org/paranal/.

[12] L. Andolfato and G.Chiozzi. Generic state machine engine software requirement specification.

[13] Thomas J. Bergin and Richard G. Gibson. *History of Programming Languages*. Addison Wesley, 1996.

[14] Bruce Powel Douglass. Uml statecharts. *Embedded Systems Programming*, January:22–42, 1999.

[15] G. Filippi, P. Sivera, and F. Carbognani. Software engineering practices for the ESO VLT programme, 2001.

[16] David S. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. Wiley, 2003.

[17] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[18] G.Chiozzi, A.Caproni, R.Cirami, P.Di Marcantonio, D.Fugate, S.Harrington, B.Jeram, M.Plesko, M.Sekoranja, H.Sommer, and K.Zagar. The alma common software, acs status and developments, 2005.

[19] Roberto Gilmozzi and Jason Spyromilio. The 42m european elt: status, 2008. URL `http://www.vt-2004.org/sci/libraries/SPIE2008/7012-19.pdf`.

[20] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[21] Object Management Group. MDA specifications, August 2010. URL `http://www.omg.org/mda/specs.htm`.

[22] Object Management Group. UML Superstructure specification, May 2010. URL `http://www.omg.org/spec/UML/2.3/Superstructure/`.

[23] Object Management Group. Object Management Group website, March 2011. URL `http://www.omg.org/`.

[24] Miro Samek. *Practical UML Statecharts in C/C++, Second Edition: Event-Driven Programming for Embedded Systems*. Newnes, 2008.

[25] Richard Soley. Model driven architecture, November 2000. URL `http://www.omg.org/~soley/mda.html`.

[26] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *Eclipse Modeling Framework*. Addison Wesley, 2008.

[27] Peter J. Young, Mario J. Kiekebusch, and Gianluca Chiozzi. Instrument control software requirement specification for extremely large telescopes.