Automatic Model Checking of UML models

Master presentation

Informatics and applied Mathematics Institute

at Bern University

Author:

Ciro Larrzabal Mazuelo

ciro.larrazabal@students.unibe.ch

2008

Tutor of the work:

Dr. Thomas Studer

Informatics and applied Mathematics Institute

Abstract

In the last decade, automatic model checking has evolved a lot in maturity and stability, mainly because its tools are used in the processing and control of chips. Today, there are some attempts to migrate these tools to another areas like, for example, software models checks. This work will try to be one of these attempts.

1 Introduction

Automatic model checking consists in a process which verifies if a model complies with, or has, certain chosen properties. Normally, such properties are expressed by means of a logical language. The development of such a process has taken big steps, and nowadays we count with many different tools. Most of these tools are developed for the sake of hardware design(for chip circuits).

Model checking is not well developed in the software area. We can show how to express a software model for a model checker and use this translation as a tool to improve such a model, because the model checker will help in the analysis of the model.

When we are constructing a new software or we are trying to make corrections in the scope of a big project, we often face problems related to flaws in design, these problems are usually more expensive and could take additional resources to be fixed when the detection takes place further in time. It takes as much to detect as its cost. An early detection, performed in the design part, lowers dramatically the costs if we compare it with a detection in production time.

It is possible to try to reduce future problems if we achieve such an early detection. We can detect the model's desired and non desired properties as well as the different states of our system using a tool of this characteristics.

In other words, a tool has been created in order to verify software's properties in the project's design time.

To make this tool, we have used a very well know software, although there were some adaptations needed to be able to fulfill the requirements. In plain, we made an interface directed towards well matured software. This interface is a wrapper or an artifact which can translate from the state diagram design of a system into the language of our checker. This means we must identify the model design's entities and convert it into a different set of entities, which are then used by the checker. We must parse and generate the rules or properties that our system needs to prove.

The input of this process is the state diagram of the UML model language. This diagram may be described as a XML file, a XMI file to be precise. Both this file and a program written with PHP/javascript allow us to use xsl and xslt processes in order to convert the file into a NUSMV file. The latter will be the input for a NUSMV model checker.

Our system must have a set of finite states, even though this number may be reduced using abstraction. With that aim in mind, we can easily apply it in examples where this process will be useful and successful.

This document is divided into several sections.

The first one covers the work's background: not only model checking (model checking, automaton, temporal logic) and software representation (XML, UML) may be found inside its bounds, but also additional information about the software techniques(XSL, XPATH) used for them as well.

The second part provides information about the tools that have been used to perform the checking (NUSMV), to represent the software model (XMI, Umbrello,CTL) or to process and handle the information (PHP, PHP-XSLT, AJAX, JQUERY).

The third part involves the application itself (the model of the wrapper and the application, several practical examples, automation of the properties checking and the tool's output).

2 Background

2.1 Model Checking

Model checking has been discussed in the introduction but we need to define formally the concept of Model checking.

> Model checking is an automatic technique for verifying finite state concurrent systems such as sequential circuit designs and communication protocols [12].

This definition may be understood as the automatic checking of properties in a model. To do achieve it, we need to express that model in a way with which we will be able to inquire further about the properties.

There are other ways to succeed in this. One is simulation, another is prototyping, but these other solutions are very expensive. The best approach is to express the model in a propositional temporal logic language, which allows deduction and brings us a logic inference. Thus, we can process it with formal methods.

The usual way to express the model is through a state-transition graph. An efficient search procedure is normally used to determine automatically if the properties are satisfied by the graph. This technique was developed by Clarke and Emerson [11, 10] and Quielle and Sifakis[9] independently. The most important characteristic in this method is that it may be highly automatic and can be used for extremely complex models.

Another improvement with this procedure is the high level of the model's representation; which provides answers in a efficient way. These answers can also be very rich whenever a counterexample is required.

A well-known problem of this technique is the *state explosion*. It is the reason why many researchers disable formal verifications; although avoiding its impact is possible with the use of ordered binary decision diagrams *OBDD*[8].

This inclusion simplifies the way of expressing transitions between the states in a complex set of variables, turning the state explosion into a simpler problem. This new solution for the problem, together with the original algorithm, is called *symbolic model checking*.

By means of it, a successful automatic model checking will be obtained but only if we are capable of representing our model as a states transition graph.

Clarke wrote about two ways to do that [12]; one of them is compositional reasoning and the other is abstraction.

The first exploits the modular structure of complex circuits and protocols. Like in modular languages, we can express aset of combined states in a part of the system as states of a new abstract entity which will be handled like a submodule. For instance, if the parent (abstract module) has a property, such property will be true for at least one of the elements of the submodule.

The second method involves the mapping of the system's actual values and representing them abstractly. It provides a smaller set and thereof a smaller quantity of states.

The algorithms and deeper concepts about the checker's inner working may be found in [12].

2.2 Symbolic Model Checking

As stated before, we can assume the states diagram and the state graph of the system after the transition can be expressed for an OBDD and which makes for a more concise representation of the transition relationship.

The symbolic model checking algorithm is implemented with a procedure *Check* that takes the CTL formula to be checked as its argument and returns an OBDD that represents exactly those states of the system that satisfy the formula.

Both the algorithms and theory itself that are required to resolve these problems can be expressed as OBDD functions or in a recursive definition of Symbolic Model Checking, more about the procedure is in [12].

For our project, we need to express the software's model as a state-transition graph. A possible method consists in using the intrinsic automaton representation of our software and consequently making the graph with such a basis.

From the model checkers perspective, an automaton may be used directly instead of expressing the states-transitions graph. But in order to test or probe any properties we need more powerful processes, for example, a deductive verification. This kind of verification is only possible whenever we may express and use axioms and logic for the analysis.

All these reasons prove our project to be a better solution to express our software like a set of simple automatons and this interaction like a special set of states and transitions.

In other cases, the process is time consuming and is not an easy task. The use of experts is also very important and some properties are very hard to express.

Therefore, we express the state diagram of the system like a set of states-transitions graph.

Nowadays, there are not many open software tools that are capable of a symbolic automatic model checking and which have improved along the time and the theory development. These achievements are due to work in temporal logic performed by many logicians.

Most of the tools have been developed with circuit's model checking in mind, which means that most of the tools are designed to describe that kind of models and are not oriented to a software's state diagram.

In the next sections we shall browse the basic theory of automatons (with the goal of understanding the basics of what we need to translate) and temporal logic (in order to understand the properties and uses of our tool).

2.3 Automaton

In the previous section, we acknowledged that it is very important to define our model in a formal and structured way, such a way it should be susceptible to a logic inference use. One of the reasons to do that is to define our system like a collection of automatas.

An automaton (automata) is a mathematical model for a machine with finite states (FSM). This machine goes from state to state according to a transition function and a symbols input. The symbols are read one at a time and whence depleted, the automaton is stopped.

A symbol is a piece of arbitrary data, which has some meaning or effect on the machine. Symbols are sometimes just called "letters".

The concatenation of symbols is called word; the set of possible symbols is called alphabet.

A set of words describes the language and could be, or not, infinite.

The automaton's state at the end of the process declares if the words are accepted or rejected, depending on if the state belongs to a set of accepted states. If the word is accepted, it is defined as a word accepted by the automaton; therefore the set of all words accepted by this automaton is defined as the language accepted by the automaton.

This language can be thought like a subset of the Kleene star over the alphabet. For that reason we can say that a language is under the Kleene closure.

> An automaton is a 5-tuple $\langle Q, E, T, q_0, l \rangle$ where Q is a finite set of states, E a finite set of transition labels,$T \subseteq Q \times E \times Q$ is the set of transition, q_0 is the initial state of the automaton and finally l is the mapping which associates with each state of Q the finite set of elementary properties which hold in that state.

The automaton could be classified according with some of its properties. We may distinguish among three different kinds of automatons: the deterministic, the non-deterministic and, finally, the non-deterministic with ϵ transitions.

A variety contains a transition for each state and each symbol in the alphabet. The second one has not transitions for each symbol or has more than one for each one, which means that the automaton will reject the set if it can not find a path to read all the input. The third kind is a specific case of the second one in which the automaton makes transitions without a symbol, by means of ϵ transitions.

The latter two kinds of automaton, may be reproduced as a deterministic automaton.

These kinds of automatons accept the same type of sets of language. The family of languages accepted is denominated the regular languages family.

More powerful automatons can accept more complicated languages.

Some of these automatons are the push down automatons(PDA) which additionally carry memory in the form of a stack. The transition function will now also depend on the symbol(s) on top of the stack, and it will also specify how the stack is to be changed at each transition, the non-deterministic PDA accepts context-free languages.

If the automaton possesses an infinite memory in the form of a tape, and a head which can read and change the tape, and move accordingly in either direction along the tape; we are in front of a Turing machine. Turing machines are equivalent to algorithms, and are the theoretical basis for modern computers. Turing machines decide/accept recursive languages and recognize recursively enumerable languages.

If the infinite tape is a proportional tape according to the size of the input string in a Turing machine, we are in front of a linear-bounded automaton (LBA). This automaton accepts context-sensitive languages.

Automatons have many different uses. If we added this automaton to a particular environment, artificial life may be simulated. This is the case of the so called "cellular automatons".

Automatons are used because they are easily represented in software, and can be used with several tools in order to achieve automatic deduction of questions expressed in temporal logic.

2.4 Temporal Logic

> Temporal logic is a form of logic specifically tailored for statements and reasoning which involve the notion of order in time.

Temporal logic is a definition alternative for describing properties with dynamic behavior. The natural way to express properties consists normally in an expression in first-order logic, although such an expression involves heavy notation which is not necessary.

A unified approach to program verification was suggested by Pnuely where the basis of a temporal reasoning was introduced [14].

Temporal logic notation is clearer and simpler. It bears easy to use concepts and its operators mimic linguistic constructions. On top of this, it presents formal semantics and it is considered an indispensable language specification tool[5].

Temporal logic uses atomic propositions to declare statements about states, these propositions are assembled into a set denoted as $Prop = \{P_1, P_2, ...\}$ and a proposition P is (defined as) true in a state q if and only if $P \ni l(q)$.

Temporal logic is based in classical propositional Boolean logic, thatswhy we have the true and false constants, the negation $\neg$, and the Boolean connectives conjunction $\wedge$, disjunction $\vee$, logical implication $\Rightarrow$ and double implication $\Leftrightarrow$. The conectors allow constructing complex statements relating several simpler subformulas.

Temporal logic contains also the so called temporal operators, which allow defining of the states sequence along the execution. The simplest are X (XP means the next states satisfy the property P), F (FP will be satisfied in a future state) and G (all future states satisfy the property).

Temporal logic brings also the possibility to arbitrarily nest the various temporal operators which provides additional power to the process.

There are other two operators **U** and **W**. When we declare that P **U** Q we mean that the proposition P will be true until the proposition Q will be reached; in contrast to **U**, P **W** Q, that expresses that if the proposition P is true, it will remain true regardless the inexorable occurrence of Q.

The final part of the temporal logic definition is the expression that helps to express the execution. The possible quantifiers are **A** and **E** and they are denominated path quantifiers. $\mathbf{A}\phi$ states that all executions out of the current state satisfy the property ϕ. On other hand, **E** states an execution end which satisfies the property.

2.5 General properties to check with Temporal Logic

Verification by means of model checking requires some expertise. Expertise alone can suggest the appropriate formulation of a problem, the way around the specific tool's limitations at hand and the proper interpretation of the results obtained ultimately (or lacked thereof).

Acquiring this skill is a maturing process that occurs over time: hands-on experience with verification problems is required, together with some long term pondering over the model checking activity itself.

The next ones are the most important or most studied properties in the area of temporal logic and automatic model checking.

2.5.1 Reachability

States that some particular situation may be reached.

Normally when we try to express this property, we often find the operators **EF** so reachability properties can be defined as those. These properties are the easiest to verify because when the model checking tool is able to construct the reachability graph of a model, it can answer all the reachability questions of the model just by analyzing the graph.

There are more techniques to answer or simulate the behavior of this property. One of them consists in adding a state to the initial one until there are no more possible states to add. The problem, of course, is the potential variable explosion.

2.5.2 Safety

Under certain conditions something never occurs.

To verify this property, the best way is with the combinators **AG**. It may generally be said that a safety problem is the negation of a reachability property.

Thus, the methods to calculate and resolve are very similar.

2.5.3 Fairness

Under certain conditions, something will (or will not) occur recurrently and indefinitely.

2.5.4 Liveness

Under certain conditions, something will ultimately occur.

2.5.5 Deadlock-freeness

States there are no deadlocks in the model.

2.6 XML

XML is a text format in which the state's diagrams we have used are expressed.

XML is the abbreviation of eXtensible Markup Language, which is derived from SGML (ISO 8879).

This work was initially designed to meet the challenges of a large-scale electronic publishing without the complication that SGML presents. Therefore, XML is playing an increasingly important role in the exchange of a wide variety of data on the web and elsewhere.

XML describes a class of data objects called XML documents and partially describes the behavior of computer programs which process them [15].

These documents are made up of storage units called entities, and such entities are enclosed in a markup which could contain attributes also.

Entities may be parsed or unparsed characters, the markup is a subset of these characters The markup should be encoded as a description of the document's layout and logical structure.

The next definitions may be found in [15].

> A software module called **XML processor** is used to read XML documents and provide access to their content and structure.
>
> It is assumed that an XML processor is doing its work on behalf of another module, called the **application**.

In our project, the applications are two Umbrello, and our application itself is developed in PHP.

The goals for XML are alike to ours, the most important among them is its possible use as an exchangeable text document.

The specifications of XML can be found in [13, 1, 2, 3, 4].

XML's standard condition is of the utmost importance. It is the most used format for data exchange over the Internet and there are tools to handle and retrieve the layout and data from such a document.

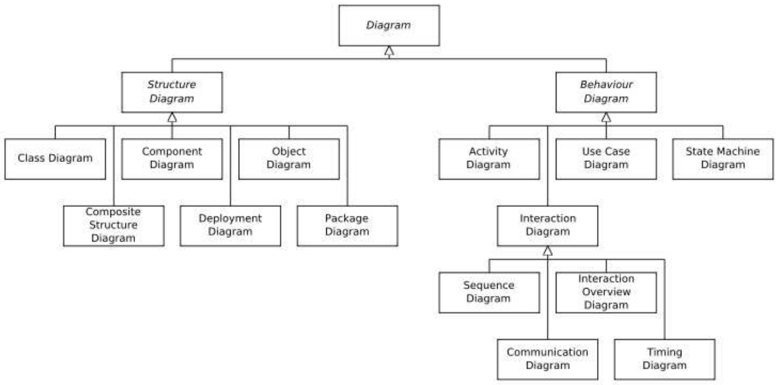


Figure 1: Class diagram of the diagrams in UML 2.0

2.7 UML

UML is a standardized general-purpose modeling language in software engineering. One of the goals of this language is to provide abstract models in a graphical notation. Each model specifies different systems of our software.

UML is officially defined by the Object Management Group. The history of this modeling approach could be regarded as an unified method founded by the Rational Software Corporation when it was the reference source for two of the most used models; OOA(object oriented analysis -Rumbaugh-) and OOD(object oriented design).

This corporation concluded that a non-proprietary modeling language would help to object oriented technology's embrace. Therefore, the specification was later proposed and finally adopted to the OMG.

Modeling in UML consists in drawing different diagrams and writing documentation such as *use cases* which drive the model elements and diagrams. UML diagrams represent three different views of a system model and for each kind of view there is a set of diagrams to achieve that view.

The three views of a system may be defined as: function requirements, structural and dynamic behavior.

The first one allows us to browse the requirements of the system through the user's eyes. The second view provides the static structure of the system; which involves objects, attributes and relationships. The third one describes dynamic behaviors such as collaborations among objects and changes to the object's internal states.

The following diagram illustrates the functional requirements (use case). It may be merged with the dynamic group and, hence, obtain a hierarchical image of the UML diagrams (1).

There is extensive information about this standard and the specification, but for our purposes we shall focus on the state machine diagram. The automatic checking will be applied onto this diagram.

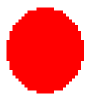


Figure 2: Representation of the initial state in UML 2.0

2.7.1 State Diagram

The UML state diagram is essentially a Harel state chart with a standardized notation.

State diagrams are used in the field of computer science, representing the behavior of a system, which is composed of a finite number of states.

State diagrams are used to describe a system's behavior. State diagrams can describe the possible states of an object depending on the different events that may occur. Each diagram usually represents objects of a single class and tracks the different states of its objects through the system [7].

State diagrams can be used to graphically represent finite state machines. Is for this fact that this kind of diagram is the ideal for our goal. This was introduced by Taylor Booth [6] in his 1967 book *Sequential Machines and Automaton Theory*. There are different kinds of these diagrams, we use UML one for our purposes.

One of the advantages of these state diagrams is the inclusion of superstates and concurrent states that allow the creation of conjunctive machines.

The diagram is divided as follows:

- Filled circle, pointing to the initial state, see Figure 2.
- Hollow circle containing a smaller filled circle, indicating the final state, see Figure 3.
- Rounded rectangle, denoting a state. The top of the rectangle contains a name of the state. It may contain a horizontal line in the middle, below which the activities performed in that state are indicated see Figure 4.
- Arrow, denoting transition. The name of the event (if any) causing this transition labels the arrow body. A watch expression may be added before a "/" and enclosed in square-brackets (event Name[watch Expression]), denoting that this expression must be true for the transition to take place. If an action is performed during this transition, it is added to the label following a "/" (event Name[watch Expression]/action see Figure 5).
- Thick horizontal line with either x>1 lines entering and 1 line leaving or 1 line entering and x>1 lines leaving. These denote join/fork, respectively, see Figure 6.

These general guidelines will be followed in order to generate a diagram:

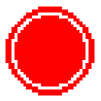


Figure 3: Representation of the final state in UML 2.0

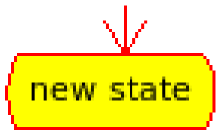


Figure 4: Representation of a state in UML 2.0

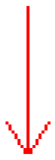


Figure 5: Representation of the transition between states in UML 2.0

Figure 6: Representation of the join or fork in UML 2.0

Place the starting point. A starting point is modeled with a filled circle. Every UML Activity or state Diagram should have a starting point on the top. This reflects the way people in Western cultures read.

An ending point is modeled with a filled circle with a border around it (the hollow one), by means of the same notation that UML State Chart diagrams use. This node is interesting because it is not always included, whenever it describes a continuous process. If this is not the case, one must be included.

Flow charting operations imply the need to simplify. As a rule of thumb, whenever an operation is so complex you need to develop an UML Activity (or state) diagram to understand it; we should consider refactoring it or generating a subdiagram.

Decision points are not included in our scope. It is possible to show that activities may occur in parallel, depicted with two parallel bars. The first bar is called a fork, it has one transition entering it and two or more transitions leaving it. The second bar is a join, with two or more transitions entering it and only one leaving it.

Some constrains will be applied to this general elements in the model during our project's development, see Section 4.2.

The restrictions for forks and joins are the following:

- A fork must have a corresponding join. In general, for every start (fork) there is an end (join). In UML 2 it is not required to have a join, but it usually makes sense and for our project is a must.
- Forks Have One Entry Transition.
- Joins Have One Exit Transition
- Each transition that exits the fork posseses its own states and does not share them.

2.8 XSL

XSLT is a language for transforming XML documents into other XML documents [18].

XSLT is designed for its use as part of XSL, which is a style sheet language for XML. In addition to XSLT, XSL includes an XML vocabulary for specifying formats. XSL specifies the styling of an XML document by using XSLT to describe how the document is transformed into another XML document that uses the formatting vocabulary.

XSLT is also designed to be used independently of XSL. However, XSLT is not intended as a completely general-purpose XML transformation language. It is instead designed primarily for the kind of transformations that are needed when XSLT is used as part of XSL.

A transformation expressed in XSLT describes rules to transform a source tree into a result tree.

The transformation is achieved by means of associations between patterns and templates. A pattern is matched against certain elements in the source tree. A template is instantiated in order to create a part of the result tree. Moreover, the result tree is separated from the source tree. Additionally, the structure of the result tree can be completely different from the structure of the source tree. In order to construct the result tree, the elements from the source tree can be filtered and reordered, and arbitrary structures may be added as well.

A transformation expressed in XSLT is called a style sheet. This is because, in the case of XSLT being transformed into the XSL formatting vocabulary, the transformation works as a style sheet.

A style sheet contains a set of template rules. A template rule has two parts: a pattern which is matched against nodes in the source tree and a template which may be instantiated to be a part of the result tree. This allows a style sheet to be applicable to a wide class of documents that have similar source tree structures.

A template is instantiated for a particular source element to create part of the result tree. A template can contain elements that specify literal result elements structures. A template can also contain elements from the XSLT namespace that make for instructions for creating result tree fragments. Whenever a template is instantiated, each instruction is executed and replaced by the result tree fragment that it generates. Instructions can select and process descendant source elements. Processing a descendant element creates a result tree fragment by finding the applicable template rule and instantiating its template. Note that elements are only processed when they have been selected by the execution of an instruction. The result tree is constructed by finding the template rule for the root node and then instantiating its template.

When a template is instantiated, it is always instantiated with respect to a current node and a current node list. The current node is always a member of the current node list. Many operations in XSLT are relative to the current node. Only a few instructions change the current node list or the current node; during the instantiation of one of these instructions, the current node list changes to a new list of nodes and each member of this new list becomes the current node. After the instruction's instantiation is complete, the current node and current node list revert to what they were before the instruction was instantiated.

XSLT makes use of the expression language defined by XPATH 2.9 for selecting elements for processing, for conditional processing and for generating text.

2.8.1 Restrictions

The reason why we do not use all the power of the XSLT transformation and only the query part of them (see Section 2.9) is the complication to handle the source tree's behavior when the position in the resulting tree depends heavily on the position of a future node to be analyzed. To avoid this problem, we can make a reprocess. But for performance reasons we choose to use the XPATH and other languages to write the output.

2.9 XPATH

XPATH is a language for addressing parts of an XML document, designed to be used by both XSLT and X Pointer.

In order to support this primary purpose, it also provides basic facilities for string, numbers and Booleans manipulation. XPATH uses a compact, non-XML syntax to ease the use of XPATH within URI's and XML attribute values. XPATH operates on the abstract, logical structure of an XML document, rather than on its superficial syntax. XPATH receives its name from its use of a path notation as in URLs for navigating through the hierarchical structure of an XML document.

In addition to its addressing use, XPATH is also designed with a natural subset in mind that may be used for matching (testing whether a node matches a pattern or not); this use of XPATH plays a very important role. Because we use it to address, testing and find nodes in the original document and to express them in our finally document.

XPATH models an XML document as a node tree. There are different types of nodes, including element nodes, attribute nodes and text nodes.

XPATH defines a way to compute a string-value for each type of node. Some kinds of nodes have names also. XPATH fully supports XML namespaces. Thus, the name of a node is modeled as a pair consisting of a local part and a possibly null namespace URI; this is called an expanded-name.

The primary syntactic construct in XPATH is the expression. An expression is evaluated to yield an object, which posseses one of the following four basic types:

- node-set
- boolean
- number
- string

The expressions in XPATH are designed for finding an object. There are functions and expressions to move along the axis and, there are another functions to test nodes and node's attributes. These functions are complemented with a set of functions over the nodes.

There are two kinds of location paths: relative location paths and absolute location paths.

A relative location path consists of a sequence of one or more location steps separated by a / symbol. The steps in a relative location path are composed together from left to right. Each step selects a set of nodes relative to a context node. An initial sequence of steps is composed together with the next step as follows.

The initial sequence of steps selects a set of nodes relative to a context node. Each node in that set is used as a context node for the following step. The sets of nodes identified by that step are used together. The set of nodes identified by the composition of the steps is this union. For example, child::div/child::para selects the para element children of the div element children of the context node or, in other words, the para element grandchildren that have div parents.

An absolute location path consists of / optionally followed by a relative location path. A / by itself selects the root node of the document containing the context node. If it is followed by a relative location path, then the location path selects the set of nodes that would be selected by the relative location path relative to the root node of the document containing the context node.

A location step can be any of the next elements:

- a node
- a pair of non-zero positive integers
- a set of variable bindings
- a function library
- the set of namespace declarations in scope for the expression

The axis through we can move are the next ones:

- 'ancestor'
- 'ancestor-or-self'
- 'attribute'
- 'child'
- 'descendant'
- 'descendant-or-self'
- 'following'
- 'following-sibling'
- 'name space'
- 'parent'
- 'preceding'
- 'preceding-sibling'
- 'self'

Most of them are related to the actual node and the relationship in the XML tree of the document.

The following are the core functions in XPATH. With these functions we can narrow or broaden the query. We may divide the functions in four types, the first group belongs to the ones which work with a set of Nodes.

- last() The *last* function returns a number equal to the context size from the expression evaluation context.
- position() The *position* function returns a number equal to the context position from the expression evaluation context.
- count(node-set) The *count* function returns the number of nodes in the argument node-set.
- node-set id(object) The *id* function selects elements by their unique ID. When the argument to id is of type node-set, then the result is the union of the result of applying id to the string-value of each of the nodes in the argument node-set. When the argument to id is of any other type, the argument is converted to a string by means of a call to a string function; the string is split into a white space-separated list of tokens; the result is a node-set containing the elements in the same document as the context node that have a unique ID equal to any of the tokens in the list.
- local-name(node-set?) The *local-name* function returns the local part of the expanded-name of the node in the argument node-set that is first in the document order. If the argument node-set is empty or the first node has no expanded-name, an empty string is returned. If the argument is omitted, it defaults to a node-set with the context node as its only member.
- name space-Uri(node-set?) The name *space-Uri* function returns the name space URI of the expanded-name of the node in the argument node-set that is the first in document order. If the argument node-set is empty, the first node has no expanded-name or the name space URI of the expanded-name is null, an empty string is returned. If the argument is omitted, it defaults to a node-set with the context node as its only member.
- name(node-set?) The *name* function returns a string containing a QName representing the expanded-name of the node in the argument node-set that is the first in the document order. The QName must represent the expanded-name with respect to the namespace declarations in effect on the node whose expanded-name is being represented. Typically, this will be the QName that occurred in the XML source. This does not need to be the case if there are namespace declarations in effect on the node that associates multiple prefixes with the same namespace. However, an implementation may include information about the original prefix in its representation of nodes. In this case, an implementation may ensure that the returned string is always the same as the QName used in the XML source. If the argument node-set is empty or the first node has no expanded-name, an empty string is returned. If the argument is omitted, it defaults to a node-set with the context node as its only member.

The second group are the ones that work with strings:

- string(object?)

- concat(string, string, string*)
- starts-with(string, string)
- contains(string, string)
- substring-before(string, string)
- substring-after(string, string)
- substring(string, number, number?)
- string-length(string?)
- normalize-space(string?)
- translate(string, string, string)

The third group consist of the Boolean functions:

- Boolean(object)
- not(Boolean)
- true()
- false()
- Lang(string)

The fourth and last group are the ones which work with numbers:

- number(object?)
- sum(node-set)
- floor(number)
- ceiling(number)
- round(number)

The definitions of the last three groups of functions are explained in [17].
The context position is always less than or equal to the context size.
In our project we use XPATH intensively to find out the relationship between the objects that are supposed to represent the XMI file.

3 Used Tools

3.1 NUSMV

NUSMV is a symbolic model checker developed at the Carnegie Mellon University, ITC- IRST, University of Genova and the University of Trento.

NUSMV is a reimplementation and extension of a prior one developed for the Carnagie Mellon University: SMV, based in BDD.

NUSMV uses the definition of a automaton to express the model.

To express the properties or to check some other ones, it uses temporal logic, in particular CTL.

Obviously the development of temporal logic and its formality is very wide. In order to maintain our generality we choose the well known CTL (2.4).

3.1.1 Describing Automaton

NUSMV uses a declarative perspective in order to describe automatons. First of all, the name of the automaton and the possible states are declared. The next example defines two automatons, one whose name is variable_name and has the states 0 to 3 and another automaton another_variable with the states on and off.

```
VAR
  variable_name : 0 .. 3 # Possible value
  another_variable: {on,off}
```

The next step is to define in the automata the behaviour rules in order to describe the rules of evolution for each of our automatons.

For the first automaton of our last example we may declare that the automaton variable_name will turn from the current state to the next one (numericaly) only if the current state of the automaton another_variable is on and the current state of the automaton variable_name is lower than 3, in other case we declare that the automaton will remain in its current state.

On the other hand, we could wish that the automaton variable_name changed to the state off only if it is in the state on and the state of variable_name is 2.

```
ASSIGN
 next(variable_name):= case
   another_variable = on
& variable_name <3: another_variable + 1;
   1:another_variable;
 esac;
 next(another_name):= case
   another_variable = on & variable_name =2: off;
   1:another_variable;
 esac;
```

These last rules of evolution are not very clever because we may not achieve the state 3 in the automata variable_name. But for this example's sake they shall be used.

The next step is to define the automatons' initial states, by means of the function init.

```
init(variable_name):= 1;
init(another_variable):= on;
```

The follow syntax definition is of expression allowed in this part of the definition of the automata.

In the syntax described below, an atom may be any sequence of characters in the set of the alphanumeric symbols plus the hiphen and underscore. All characters in a name are significant, and case sensitive, white-space characters are space, tab and newline. Any string starting with two dashes (–) and ending with a new line is a comment. A number is any sequence of digits. Any other tokens recognized by the parser are enclosed in quotes in the syntax expressions below.

Expressions are built out of variables, constants and a collection of operators, including Boolean connectives, integer arithmetic operators, and case expressions. The syntax of expressions is as follows.

```
expr ::
       atom               -- a symbolic constant
       | number           -- a numeric constant
       | id               -- a variable identifier
       |"!" expr          -- logical not
       |expr1 "&" expr2   -- logical and
       |expr1 "|" expr2   -- logical or
       |expr1 "->" expr2  -- logical implication
       |expr1 "<->" expr2 -- logical equivalence
       |expr1 "=" expr2   -- equality
       |expr1 "!=" expr2  -- disequality
       |expr1 "<" expr2   -- less than
       |expr1 ">" expr2   -- greater than
       |expr1 "<=" expr2  -- less that or equal
       |expr1 ">=" expr2  -- greater than or equal
       |expr1 "+" expr2   -- integer addition
       |expr1 "-" expr2   -- integer subtraction
       |expr1 "*" expr2   -- integer multiplication
       |expr1 "/" expr2   -- integer division
       |expr1 "mod" expr2 -- integer remainder
       |"next" "(" id ")" -- next value
       |set_expr          -- a set expression
       |case_expr         -- a case expression
```

An id or identifier is a symbol or expression which identifies an object, such as a variable or a defined symbol. Since an id can be an atom, the ambiguity is flagged by the interpreter as an error.The expression next(x) refers to the value of identifier x in next state. The order of parsing precedence from highest to lowest is

```
*,/
+,-
mod
=,!=,<,>,<=,>=
!
&
|
->,<->
```

How is used all this will be detailed in depth when we start to develop the wrapper, see Section 4.2.

3.1.2 Verification

The query for a module or for the interaction of the system is maded following the syntaxis:
SPEC (temporal logic expression).
For example, the next one is used to verify if there is always a value of true.

```
SPEC
  AG EX 1
```

3.1.3 Network

To define the bond between two automatons, we must define each one of them as an object which has received a parameter as input; and a main automaton (as an abstraction which contains all the others) that points out such a relationship.

```
MODULE main
VAR
   elem1 : A1;
   elem2 : A2(elem1.res1, elem1.res2);
   elem3 : A3(elem2.out);
SPEC
   (elem2.out=1)->...
```

In this case, the output from A1 is the input to A2, and the output of A2 is the input to A3. Obviously our network is synchronous. To make an asynchronous network we must change from a module to a process in order to indicate to our automaton that the variables are changed according to the global variables. Therefore, we understand that whenever we command the change the word *MODULE* to *PROCESS*, the name of the variables that present a statical behavior in running time will be also relative to the module where they are encapsulated.

Another big difference between MODULES and PROCESS is that we can use in process the pass of variables, in this way we can share variables among the differents process and have a way to coordinate the process.

In particulary we use alot of proccess for the quality to provide us a way to coordinate and syncronize our automatas. We construct a master module wich take control of the subprocess to handle differnst situation, we use this to coordinate the stop of a parent module while the child evolve, or the parallel evolution when a fork call for one.

3.2 XMI

3.2.1 Umbrello

Umbrello UML Modeler is an UML diagram tool that can support the software development process, specially during the analysis and design phases of this process.

Umbrello UML Modeler is Free Software and available at no cost, this tool is since the version 1.4 (actual is 2.1.0) XMI standard compliant. This is critical to our means because we can use this tool to make the tests, but in fact it is compatible with other tools that are XMI compliant too, for instance argouml, rational rose, or Poseidon.

Another very useful function is the diagrams generation, which is a function implemented in Umbrello and invoked from the command line. But only if you have a X environment.

The lack of a tool to make the UML diagrams in png or SVG format may prove to be frustrating; in order to use them in console mode. The closest is a tool that provides such a functionality but is not compatible with our version of XMI (it use version 2 and 1.5 is required).

3.3 PHP

PHP, which stands for *PHP: Hypertext Preprocessor* is a widely-used open source general-purpose scripting language that is specially suited for web development and may be embedded into HTML. Its syntax draws upon C, Java, and Perl, and is easy to learn. The main goal of the language is to allow web developers to write dynamically generated web pages quickly, but you can do much more with PHP.

What distinguishes PHP from client-side code (for example Javascript) is that the code is executed on the server side, generating HTML on the fly which is then sent to the client. The client would receive the results of running that script, but would not know what the underlying code was. You can even configure your web server to process all your HTML files with PHP, and then there is really no way that users can tell what you have up your sleeve.

The most important advantages of using PHP are that it is extremely simple for a newcomer, but offers many advanced features for a professional programmer at the same time. One must not be afraid to read the long list of PHP's features. One can jump in, in a short time, and start writing simple scripts in a few hours, that is the reason why this script language was chosen as my application's language.

PHP and HTML interact a lot: PHP can generate HTML, and HTML can pass information to PHP, PHP can also retrieve information from other sources as files and execution outputs.

3.4 PHP-XSLT

We use the DOM extension in order to operate the *domdocument* and *domxpath* classes. Both of them are part of the new API in PHP5. There is another class, the XSL extension, which implements the XSL standard using the XSLT transformation as well as the libxslt (a well documented library in the POSIX-compatible operating systems). The first one will fit in our purposes. It is also possible to use the *simplexml* extension.

3.5 AJAX

Ajax (asynchronous Javascript and XML), or AJAX, is a group of interrelated web development techniques used for creating interactive web applications or rich Internet applications. With Ajax, web applications can retrieve data from the server asynchronously in the background without interfering with the display and behavior of the existing page. Data is retrieved using the XMLHttpRequest object or through the use of Remote Scripting in browsers that do not support it. Despite the name, the use of Javascript, XML, or its asynchronous use is not required.

AJAX is based on the following web standards:

- Javascript
- XML
- HTML
- CSS

The web standards used in AJAX are well defined, and supported by all major browsers.

3.6 JQUERY

JQUERY is a lightweight Javascript library that emphasizes interaction between Javascript and HTML. It was released January 2006 at BarCamp NYC by John Resig.

Dual licensed under the MIT License and the GNU General Public License, JQUERY is free and open source software. JQUERY contains the following features:

- DOM element selections
- DOM traversal and modification, (including support for CSS 1-3 and basic XPATH)
- Events
- CSS manipulation
- Effects and animations
- Ajax
- Extensibility
- Utilities - such as browser version and the each function.
- Javascript Plug-ins

Due of the reason of DOM traversal and non intrusive Javascript code this library was included in the project, the DOM manipulation is achieved through XPATH.

4 Application development

4.1 Model of the wrapper

As detailed in the introduction, the project consists in a tool to make documents from a UML diagram comply with the syntax of the model checker. With that aim in the horizon, the converter is implemented with aid from a program language and its extensions (XSLT and XPATH).

The application is an object oriented application which is run through the web server. The interaction between the client and the server is through a browser. In order to ease the interaction, the interface uses the (ad-hoc) standards of the current interaction (non obstrusive java script and Ajax messaging).

The application itself reads the original XMI file, and makes its own DOM structure out of it, then converts this file, with help of different rules from the program and makes searches in the DOM structure, in a new hierarchy, then assigns the hierarchy to the corresponding class. This class will print the new file in that format.

During the project's development, some issues arose that complicated the translation of the documents. First of all, the parallelism between the components will be explained. Then the differences will be detailed. Finally, the issues mentioned before will be browsed; as well as the solutions adopted in order to solve them.

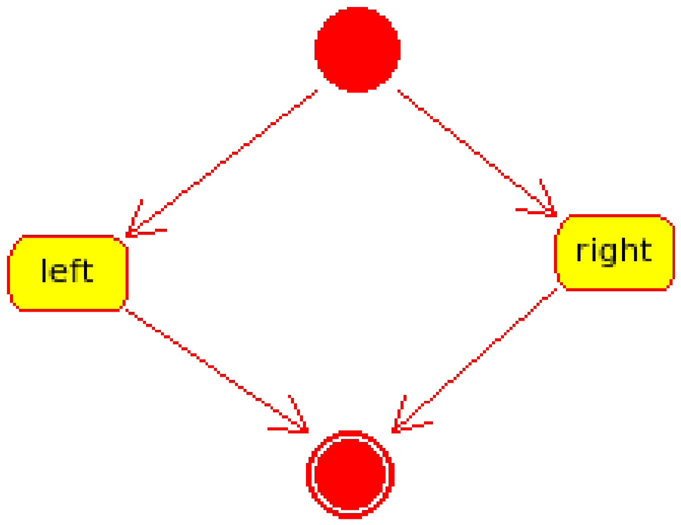


Figure 7: A simple State Diagram

An automata in UML may be described in a state diagram, with a start state and a final state (see Figure 7), the same can be achieved in NUSMV with the description of a module, and the definition of the states in that module.

It seems easy to make a module for each automata in the UML model with a simple declaration of the states in the module, but the complication comes whenever we must describe the process and evolution of the automaton. In NUSMV, this is not so complicate. In fact, there are ways to declare the evolution in terms of future states; and add conditions to this condition to make the evolution.

With only that in mind we made a simple XSLT document (without PHP) which performs the translation, but fails to make a proper translation because it has not enough information or the result is incomplete. One of the issues consists in finding the route where the problem (when the checking fails) occur. This happens because the the the path has the problem but because we can not tell that (there isn't a way to point out a particular path), fails over the initial state of the automaton is origin, normally the problem appears when a property is not achieved. They can not show us anything to debug (the path followed would be of deep interest).

One way to solve this problem is to obtain a parallel definition of the relationships between states, as another automaton which changes its state to point out the position of the path. This solution brings another issue: how to synchronize the evolution of the states in the automatas. It may be made by means of conditional paths as will be explained later.

In this example, the initial state must be chosen between the left path or the right path. If the left is chosen, the new state will be *left*, otherwise *right* shall be taken.

The old solution obtains an output such as this:

```
automate:{initial, left, right};
```

Declaration of the evolution

```
next(automate):= {left,right};
```

In this example, if we ask whether the *right* node is reachable, a failure shall be obtained. This happens because in one possible case, automate may choose the left node and that will be the last node thus *right* will be not reachable, which is not exactly true. To know exactly , and therefore to make the checker probe other possibilities we must construct the path as an automaton.

The solution in this very first example is not so complicated but we must add another state just to explain another detail about the change.

Declaration of the states.

```
automate := {initial, left, right, final};
path :={initialleft, initialright, rightfinal, leftfinal};
```

As you can see, the name of the link between the states is called like the states which the union is for. The evolution of the automata and the path must be declared too.

```
next(automate) := case
automate=initial & path = initialleft : left;
automate=initial & path = initialright : right;
automate=left & path = leftfinal : final;
automate=right & path = rightfinal : final;
1 : automate;
esac;
next(path) := case
next(automate) = left : leftfinal;
next(automate) = right : rightfinal;
automate = initial : {initialright, rightfinal};
1:path;
esac;
```

This solution allows the use of NUSMV to make subdefinitions too and a more complex description of the classes by abstraction in layers. These solutions come together with a *ad-hoc* rule: the parent state has the same name of the graphic which explains its details.

The main diagram needs a name too, the defacto name is used from Umbrello (State Diagram), and a token has been implemented in order to express the process flow.

This idea has other implications, as for instance the fact that our model needs a meta-model to control the token, and therefore the flow of the states.

To achieve this goal, we build the model checker as a main program which has an instance of each diagram module, like a function, and a parameter which indicates the actual state of the parameter for that function. That means each subdiagram is a module itself, and each module is delayed until the turn for that module allows the module to start. For that reason we include a condition to be added in each module, making them rely on that conditional and letting the evolution and transitions occur under a main control.

The conditional part is the control part of our system. In the definitions of each module they play the role of *turn* or token with which they can start or not, this is why they make for the control of the states' evolution that are expressed.

When the token stays in the main diagram, it means that it does not reach a state with a subdiagram, but in that case the token changes to that diagram, halts the main diagram and starts the subdiagram. Then stay in that position until the subdiagram reaches an end node or a subsubdiagram is reached in the first case. The control is then given back to the main process in the latter to the subsubprocess, the other processes are held in halt.

Moreover, the names of the submodules are the names of the states plus the extension proc, and the instance of each module is the name of the state, outer the main module which name is pr (from principal).

```
MODULE main
 VAR
   turn:{ main, left};
   pr: mainproc(turn);
   left: leftproc(turn);

 ASSIGN
   init(turn) := main;
   next(turn) := case
                  next(pr.state=left) & next(left.state)
= start:left;
                  pr.state=left & left.state = end: main;
                  1:turn;
                esac;
```

The changes applied to the module parts of each diagram are just to express and follow the control rules. This example expresses the following.

```
MODULE mainproc(eventParent)
VAR
    event:{ startready, readyleft, leftend, readyright
, rightend,  endend };
    state:{ ready,start,right,left,end, error };
ASSIGN
```

```
  init(state) := start;
  init(event) := startready;

next(path) := case
next(automate) = left : leftfinal;
next(automate) = right : rightfinal;
automate = initial : {initialright, rightfinal};
1:path;
esac;

next(state) := case
                   eventParent != main& state = end : start;
                   eventParent != main & state != end: state;

                   eventParent != listening& state = end : start;
                   eventParent != listening & state != end:
state;

                   state = start & event = startready : ready;
                   state = ready & event = readyleft : left;
                   state = left & event = leftend : end;
                   state = ready & event = readyright : right;
                   state = end : end;
                   1:error;
                esac;

  next(event) := case
                   next( state ) = left : leftend;
                   next( state ) = start : startready;
                   next( state ) = ready :
{ readyright , readyleft };
                   next( state ) = right : rightend;
                   next( state ) = end :  event;
                   1 : event;
                esac;
```

Most of the modules are written like this prior one, except for the part where the definition of *state* is detailed. For most of the modules, but *main*, the definition is included as mentioned before. For the normal case, the first definition is changed to be settled to the initial state (eventParent != main : start;)

This organization in modules provides additional flexibility and a broader reach.

Another issue was forks and joins handling. In NUSMV there may be simultaneously states inside a module, but only when there are different procedures. This is the reason why we need to define a subset of the diagram's states and isolate them. These subsets contain all the states that take part in the path between the fork and its join (this is also the reason why it is so important to have a join for each fork). With the elements of this subset we build a new module with so many own paths as the fork defines (these parts define a sort of domain; thus, it is not possible to share the states) and a join which joins all these paths.

The part which involves the fork/join set in the original diagram will be replaced for a state with the name of the nodes before the fork and after the join. It also handles the same rules than the other processes in our meta system.

A example about this concept can be found in the final example in the Section 4.3.

This project also works with the automatic generation of the most used property, reachability. The process of the document's generation is taken advantage of in order to make at the same time a structure of the elements in each diagram and their relationships. From this point of view, it is relatively easy to write the temporal proposition that expresses such a property.

The properties are expressed:

```
<modulename>.state=<name state>;
```

4.2 Model of the application

In this section, the relationship between the XMI file and the NUSMV will be exposed, as well as the means to obtain the information through the use of XPATH.

To understand this process it is important to know how each part of the model is expressed in the Document (XMI).

The document is started with the version and encoding specification, as well as with the XMI tag. This part of the document includes the version of the XMI, (in our case 1.2) and the namespace that used (UML 1.3).

Inside the bounds of this tag (and its close tag) the document has at least two parts. The first one is delimited with the XMI tags *header* and the second with *content*. It is possible to find a third one that will cope with *extensions*. In the first part, there is information about the tool that built the diagram; in addition to its version and encoding. In the second one, the description of the model itself may be found. The UML tag *model* includes the definition of the model and its relationships.

In *model* we may also find the tag Namespace.ownedElemt to define the different diagrams. This tag has the definition of folders for our model, these folders are normally used to contain a certain kind of diagram diagrams each. Therefore, we may find the *Logical view* folder, *use case view.* etc. specified in the **Logical View**.

In the Logical view the different diagrams are to be found. The tag *diagrams* encloses all the other (logical) diagrams. Each diagram is defined with the tag *diagram.* This tag must have a name (the name for the main diagram is "state diagram") and the attribute **type** is *5* (in case of a very special definition these properties can be easily changed to search for new ones).

The XPATH to find the diagrams is

```
//diagram[@type=5]
```

This makes for a search in all the DOM trees for the nodes which are enclosed with the tag *diagram*, but only for the ones that have the attribute *type* set to 5.

Then, further in the XMI document, we may browse for each diagram node (of the DOM tree) the definition of the differents elements of each one of them.

In the XMI document we may note that each diagram has two kind of elements and a couple of attributes. The elements are *widgets* and *associations.* The first one encloses the diagram's entities and the second one the relationships between elements.

The widgets enclose the elements *statewidget* and *fork/join.* The statewidget element, as its name points out, is the entity where the information about a state, like name, id, position type and more; is defined. Our focus will be set on name, id, and type. The other attributes are not so important for the construction of the model. The second kind of elements (fork/join) defines a fork or a join. XMI uses for both the same entity, the only difference is that in the definition of associations, which will explained later, the id is of the utmost importance.

In the association part we have just one kind of element. An elementmay hold information and other elements. Most of the information is about which elements in the widget part are related and, how they are related (the cardinality of the connections in the elements and its direction).

With the information from both parts, we can make a distinction between the different diagrams in order to separate the main diagram, the child diagram and finally the fork/join behavior.

In order to state a difference between the main diagram and the child ones we just perform a search on the name. When no names are available, or the default name is used, we assume that that diagram is from the main diagram. The search-question with XPATH is:

```
//diagrams[@type=5]/digram[@name='' or @name='state diagram']
```

In our program the function getAttribute is used.

With that information we can filter and organize the next query, and use that to know about their children and the relationships above them.

Later we proceed to make the diagrams. If we find an element (state), we add this to the array of states. There are three kind of states: the normal ones, the initial ones, and the final ones. Each one of them may be recognized thanks to the value of the attribute *statetype.* This attribute is 1 for the normal states, 2 for the end ones and 0 for the initial states.

If we found the element fork/join, we make an exception for all the elements that are between the definition of this fork and its join. The elements and associations involved are extracted and a new diagram is built out of them.

For each of these lists, we look for the initial state element. This will be our first element. Then, for each one of the other elements in the array of elements, we build the relationships with its own relationship array.

The elements in the fork-join process are denominated as a brand new element and are dealt with as an unique state (like a subdiagram).

Due to this important exception, it is mandatory to use additional resources apart from xsl in order to make the transformation.

In our program, we divide and create an object diagram for each instance that is needed. This diagram could be one of the next ones: main , fork or nested. Whenever the final document must be constructed, the information of this objects is written consequently.

A special point is the fork/join diagram, because this must have so many independent threads in the module as out arrows are in the fork. Therefore, the diagram has more than one state instance at a time and more than one direction control. Also, the end state is only achieved when all the independent threads are concurrent with the end state. This part poses many questions, like: "what should be the treatment of a threat when the other lines of process are delayed?", "where should the state be when they all are in end" or "what is done when the independent threads are not so independent?".

The most interesting part is the creation of the overall control from the information of the different diagrams. This system diagram is constructed with information from the other diagrams; and takes care of the relationship between the different diagrams. Most of the relationships are made at parse time and are of the kind *parent child* or *join/fork*.

4.3 Examples

4.3.1 Achieved Output

The output of our system is made with the information of the objects created in the parse layer. Such objects are diagrams which have states, and each state has also its own way to express itself.

Most of the objects fill out a template in order to provide the expression of their content. This way we can reutilize the code more times in order to build the whole document.

The templates are defined with the same structure of the final document and are all written in a monolithic document. The templates are also used to make automatic questions (reachability) about the information of the states in each diagram, their location and their relationship with the other diagrams.

The other generated texts like, for instance, the auto-generated questions or the system questions; are saved in different documents.

This auto-generated question are the set of normal question that can be easily modeled from the diagrams self, like by example the property of reachability

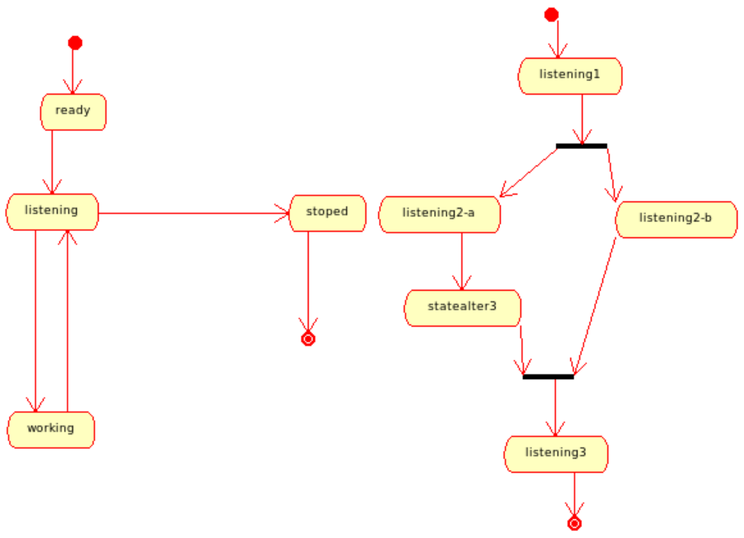


Figure 8: State diagram of our example

for each one of the states in all the modell. Other group that is also important at the begining of the work are the debuging properties (safety with the special states generated with the tool).

In our interface all this automatic questions are generated with the checking of a combo in the interface.

The questions self are generated at conversion time, when all the elements are integrated in our modell object. The question have all the syntax required for the NUSMV.

The system generates a folder for each model to be analyzed; in order to keep the information, it also generates a document for the user defined questions.

These folders are under the *tmp* one, under the same folder than the rest of the applications. The main goal is to produce objects dynamically and write a file out of those objects posteriorly.

4.3.2 Example

In our example, we use the UML diagrams, see Figure 8.

These two diagrams are the state diagrams of a client server system.

The first one makes for the main diagram and the second details an extension for the main one.

Both definitions are represented in XMI as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<xmi verified="false" xmi.version="1.2"
timestamp="2008-07-07T15:16:22"
```

```
xmlns:UML="http://schema.omg.org/spec/UML/1.3" >
 <xmi.header>
  <xmi.documentation>
   <xmi.exporter>umbrello uml modeller
http://uml.sf.net</xmi.exporter>
   <xmi.exporterVersion>1.5.8</xmi.exporterVersion>
   <xmi.exporterEncoding>UnicodeUTF8</xmi.exporterEncoding>
  </xmi.documentation>
  <xmi.metamodel xmi.version="1.3" href="UML.xml"
xmi.name="UML"/>
 </xmi.header>
 <xmi.content>
  <UML:Model isSpecification="false" isAbstract="false"
isLeaf="false"  xmi.id="m1" isRoot="false" name="UML Model" >
   <UML:Namespace.ownedElement>
    <UML:Stereotype visibility="public" isSpecification="false"
namespace="m1" isAbstract="false" isLeaf="false"
isRoot="false" xmi.id="folder" name="folder" />
.
.
.
       </UML:Namespace.ownedElement>
      </UML:Package>
     </UML:Namespace.ownedElement>
     <xmi.extension xmi.extender="umbrello" >
      <diagrams>
       <diagram showopsig="1" linecolor="#ff0000" snapx="10"
showattribassocs="0" snapy="10" linewidth="0" showattsig="1"
showpackage="0" showstereotype="0" name="state diagram"
font="Sans,10,-1,5,50,0,0,0,0,0" canvasheight="602"
canvaswidth="603" localid="" snapcsgrid="0" showgrid="0"
showops="1" usefillcolor="1" fillcolor="#ffffc0" zoom="100"
xmi.id="oswc7FMsG3Ib" documentation="" showscope="1"
snapgrid="0" showatts="1" type="5" >
        <widgets>
         <statewidget width="51" x="270" y="81"
usesdiagramusefillcolor="0" statename="ready"
usesdiagramfillcolor="0" isinstance="0" fillcolor="#ffffc0"
height="28" linecolor="#ff0000" xmi.id="kPbLcusdfz7Z"
statetype="1" usefillcolor="1" documentation="" linewidth="none"
font="Sans,10,-1,5,50,0,0,0,0,0" >
          <Activities/>
         </statewidget>
         <statewidget width="10" x="292" y="36"
usesdiagramusefillcolor="1" statename="State"
usesdiagramfillcolor="1" isinstance="0" fillcolor="none"
```

```
height="10" linecolor="none" xmi.id="4fmmobqbTnDY" statetype="0"
usefillcolor="1" documentation="" linewidth="none"
font="Sans,10,-1,5,50,0,0,0,0,0" >
          <Activities/>
         </statewidget>
         <statewidget width="72" x="243" y="159"
usesdiagramusefillcolor="0" statename="listening"
usesdiagramfillcolor="0" isinstance="0" fillcolor="#ffffc0"
height="28" linecolor="#ff0000" xmi.id="r6OIrbpr8E7g"
statetype="1" usefillcolor="1" documentation="" linewidth="none"
font="Sans,10,-1,5,50,0,0,0,0,0" >
          <Activities/>
         </statewidget>
         <statewidget width="68" x="244" y="329"
usesdiagramusefillcolor="0" statename="working"
usesdiagramfillcolor="0" isinstance="0" fillcolor="#ffffc0"
height="28" linecolor="#ff0000" xmi.id="rwwuKaYd1oaU"
statetype="1" usefillcolor="1" documentation="" linewidth="none"
font="Sans,10,-1,5,50,0,0,0,0,0" >
          <Activities/>
         </statewidget>
         <statewidget width="60" x="464" y="160"
usesdiagramusefillcolor="1" statename="stoped"
usesdiagramfillcolor="1" isinstance="0" fillcolor="none"
height="28" linecolor="none" xmi.id="ke5AIIHGlcNX" statetype="1"
usefillcolor="1" documentation="" linewidth="none"
font="Sans,10,-1,5,50,0,0,0,0,0" >
          <Activities/>
         </statewidget>
         <statewidget width="10" x="474" y="267"
usesdiagramusefillcolor="1" statename="State"
usesdiagramfillcolor="1" isinstance="0" fillcolor="none"
height="10" linecolor="none" xmi.id="327wisr7Yr0z" statetype="2"
usefillcolor="1" documentation="" linewidth="none"
font="Sans,10,-1,5,50,0,0,0,0,0" >
          <Activities/>
         </statewidget>
        </widgets>
        <messages/>
        <associations>
         <assocwidget indexa="1" indexb="1" visibilityA="0"
widgetaid="4fmmobqbTnDY" visibilityB="0" roleBdoc="" roleAdoc=""
linecolor="none" changeabilityA="900" totalcounta="2"
changeabilityB="900" widgetbid="kPbLcusdfz7Z" totalcountb="2"
type="514" documentation="" linewidth="none" >
          <linepath>
```

```
           <startpoint startx="297" starty="46" />
           <endpoint endx="295" endy="81" />
          </linepath>
         </assocwidget>
         <assocwidget indexa="1" indexb="1" visibilityA="0"
widgetaid="kPbLcusdfz7Z" visibilityB="0" roleBdoc="" roleAdoc=""
linecolor="none" changeabilityA="900" totalcounta="2"
changeabilityB="900" widgetbid="r6OIrbpr8E7g" totalcountb="2"
type="514" documentation="" linewidth="none" >
          <linepath>
           <startpoint startx="295" starty="109" />
           <endpoint endx="279" endy="159" />
          </linepath>
         </assocwidget>
         <assocwidget indexa="1" indexb="1" visibilityA="0"
widgetaid="r6OIrbpr8E7g" visibilityB="0" roleBdoc="" roleAdoc=""
linecolor="none" changeabilityA="900" totalcounta="3"
changeabilityB="900" widgetbid="rwwuKaYd1oaU" totalcountb="3"
type="514" documentation="" linewidth="none" >
          <linepath>
           <startpoint startx="267" starty="187" />
           <endpoint endx="266" endy="329" />
          </linepath>
         </assocwidget>
         <assocwidget indexa="2" indexb="2" visibilityA="0"
widgetaid="rwwuKaYd1oaU" visibilityB="0" roleBdoc="" roleAdoc=""
linecolor="none" changeabilityA="900" totalcounta="3"
changeabilityB="900" widgetbid="r6OIrbpr8E7g" totalcountb="3"
type="514" documentation="" linewidth="none" >
          <linepath>
           <startpoint startx="289" starty="329" />
           <endpoint endx="291" endy="187" />
          </linepath>
         </assocwidget>
         <assocwidget indexa="1" indexb="1" visibilityA="0"
widgetaid="r6OIrbpr8E7g" visibilityB="0" roleBdoc="" roleAdoc=""
linecolor="none" changeabilityA="900" totalcounta="2"
changeabilityB="900" widgetbid="ke5AIIHGlcNX" totalcountb="2"
type="514" documentation="" linewidth="none" >
          <linepath>
           <startpoint startx="315" starty="173" />
           <endpoint endx="464" endy="174" />
          </linepath>
         </assocwidget>
         <assocwidget indexa="1" indexb="1" visibilityA="0"
widgetaid="ke5AIIHGlcNX" visibilityB="0" roleBdoc="" roleAdoc=""
```

```
linecolor="none" changeabilityA="900" totalcounta="2"
changeabilityB="900" widgetbid="327wisr7Yr0z" totalcountb="2"
type="514" documentation="" linewidth="none" >
          <linepath>
           <startpoint startx="494" starty="188" />
           <endpoint endx="479" endy="267" />
          </linepath>
         </assocwidget>
        </associations>
       </diagram>
       <diagram showopsig="1" linecolor="#ff0000" snapx="10"
showattribassocs="0" snapy="10" linewidth="0" showattsig="1"
showpackage="0" showstereotype="0" name="listening"
font="Sans,10,-1,5,50,0,0,0,0,0" canvasheight="602"
canvaswidth="603" localid="" snapcsgrid="0" showgrid="0"
showops="1" usefillcolor="1" fillcolor="#ffffc0" zoom="100"
xmi.id="rVFJDDjSc0qa" documentation="" showscope="1"
snapgrid="0" showatts="1" type="5" >
        <widgets>
         <statewidget width="81" x="177" y="87"
usesdiagramusefillcolor="1" statename="listening1"
usesdiagramfillcolor="1" isinstance="0" fillcolor="none"
height="28" linecolor="none" xmi.id="68Ek9LYFX7wM"
statetype="1" usefillcolor="1" documentation=""
linewidth="none" font="Sans,10,-1,5,50,0,0,0,0,0" >
          <Activities/>
         </statewidget>
         <statewidget width="81" x="166" y="382"
usesdiagramusefillcolor="0" statename="listening3"
usesdiagramfillcolor="0" isinstance="0" fillcolor="#ffffc0"
height="28" linecolor="#ff0000" xmi.id="ooVL8UQxxdHA"
statetype="1" usefillcolor="1" documentation=""
linewidth="none" font="Sans,10,-1,5,50,0,0,0,0,0" >
          <Activities/>
         </statewidget>
         <statewidget width="10" x="198" y="48"
usesdiagramusefillcolor="1" statename="State"
usesdiagramfillcolor="1" isinstance="0" fillcolor="none"
height="10" linecolor="none" xmi.id="qydSyR24aEmj"
statetype="0" usefillcolor="1" documentation=""
linewidth="none" font="Sans,10,-1,5,50,0,0,0,0,0" >
          <Activities/>
         </statewidget>
         <statewidget width="10" x="216" y="445"
usesdiagramusefillcolor="1" statename="State"
usesdiagramfillcolor="1" isinstance="0" fillcolor="none"
```

```
height="10" linecolor="none" xmi.id="ZIy6PuQBguWm" statetype="2"
usefillcolor="1" documentation="" linewidth="none"
font="Sans,10,-1,5,50,0,0,0,0,0" >
          <Activities/>
         </statewidget>
         <statewidget width="95" x="68" y="195"
usesdiagramusefillcolor="0" statename="listening2-a"
usesdiagramfillcolor="0" isinstance="0" fillcolor="#ffffc0"
height="28" linecolor="#ff0000" xmi.id="S74y9TpvIRI2"
statetype="1" usefillcolor="1" documentation="" linewidth="none"
font="Sans,10,-1,5,50,0,0,0,0,0" >
          <Activities/>
         </statewidget>
         <statewidget width="95" x="253" y="199"
usesdiagramusefillcolor="1" statename="listening2-b"
usesdiagramfillcolor="1" isinstance="0" fillcolor="none"
height="28" linecolor="none" xmi.id="7DVSBNhve6CE"
statetype="1" usefillcolor="1" documentation="" linewidth="none"
font="Sans,10,-1,5,50,0,0,0,0,0" >
          <Activities/>
         </statewidget>
         <forkjoin width="40" x="207" y="154"
usesdiagramusefillcolor="1" drawvertical="0"
usesdiagramfillcolor="1" isinstance="0" fillcolor="none"
height="4" linecolor="#000000" xmi.id="NX6JKfBiTfwR"
usefillcolor="1" linewidth="none"
font="Sans,10,-1,5,50,0,0,0,0,0" />
         <statewidget width="91" x="88" y="268"
usesdiagramusefillcolor="1" statename="statealter3"
usesdiagramfillcolor="1" isinstance="0" fillcolor="none"
height="28" linecolor="none" xmi.id="eWecjaiw5A6Q"
statetype="1" usefillcolor="1" documentation="" linewidth="none"
font="Sans,10,-1,5,50,0,0,0,0,0" >
          <Activities/>
         </statewidget>
         <forkjoin width="40" x="181" y="334"
usesdiagramusefillcolor="1" drawvertical="0"
usesdiagramfillcolor="1" isinstance="0" fillcolor="none"
height="4" linecolor="#000000" xmi.id="qz2w9GcWnjo5"
usefillcolor="1" linewidth="none"
font="Sans,10,-1,5,50,0,0,0,0,0" />
        </widgets>
        <messages/>
        <associations>
         <assocwidget indexa="1" indexb="1" visibilityA="0"
widgetaid="qydSyR24aEmj" visibilityB="0" roleBdoc="" roleAdoc=""
```

```
linecolor="none" changeabilityA="900" totalcounta="2"
changeabilityB="900" widgetbid="68Ek9LYFX7wM" totalcountb="2"
type="514" documentation="" linewidth="none" >
          <linepath>
           <startpoint startx="203" starty="58" />
           <endpoint endx="217" endy="87" />
          </linepath>
         </assocwidget>
         <assocwidget indexa="1" indexb="1" visibilityA="0"
widgetaid="ooVL8UQxxdHA" visibilityB="0" roleBdoc="" roleAdoc=""
linecolor="none" changeabilityA="900" totalcounta="2"
changeabilityB="900" widgetbid="ZIy6PuQBguWm" totalcountb="2"
type="514" documentation="" linewidth="none" >
          <linepath>
           <startpoint startx="206" starty="410" />
           <endpoint endx="221" endy="445" />
          </linepath>
         </assocwidget>
         <assocwidget indexa="1" indexb="1" visibilityA="0"
widgetaid="68Ek9LYFX7wM" visibilityB="0" roleBdoc="" roleAdoc=""
linecolor="none" changeabilityA="900" totalcounta="2"
changeabilityB="900" widgetbid="NX6JKfBiTfwR" totalcountb="2"
type="514" documentation="" linewidth="none" >
          <linepath>
           <startpoint startx="217" starty="115" />
           <endpoint endx="227" endy="154" />
          </linepath>
         </assocwidget>
         <assocwidget indexa="2" indexb="1" visibilityA="0"
widgetaid="NX6JKfBiTfwR" visibilityB="0" roleBdoc="" roleAdoc=""
linecolor="none" changeabilityA="900" totalcounta="3"
changeabilityB="900" widgetbid="7DVSBNhve6CE" totalcountb="2"
type="514" documentation="" linewidth="none" >
          <linepath>
           <startpoint startx="233" starty="158" />
           <endpoint endx="300" endy="199" />
          </linepath>
         </assocwidget>
         <assocwidget indexa="1" indexb="1" visibilityA="0"
widgetaid="NX6JKfBiTfwR" visibilityB="0" roleBdoc="" roleAdoc=""
linecolor="none" changeabilityA="900" totalcounta="3"
changeabilityB="900" widgetbid="S74y9TpvIRI2" totalcountb="2"
type="514" documentation="" linewidth="none" >
          <linepath>
           <startpoint startx="220" starty="158" />
           <endpoint endx="115" endy="195" />
          </linepath>
         </assocwidget>
         <assocwidget indexa="1" indexb="1" visibilityA="0"
widgetaid="S74y9TpvIRI2" visibilityB="0" roleBdoc="" roleAdoc=""
linecolor="none" changeabilityA="900" totalcounta="2"
changeabilityB="900" widgetbid="eWecjaiw5A6Q" totalcountb="2"
type="514" documentation="" linewidth="none" >
          <linepath>
           <startpoint startx="115" starty="223" />
           <endpoint endx="133" endy="268" />
          </linepath>
         </assocwidget>
         <assocwidget indexa="1" indexb="2" visibilityA="0"
widgetaid="7DVSBNhve6CE" visibilityB="0" roleBdoc="" roleAdoc=""
linecolor="none" changeabilityA="900" totalcounta="2"
changeabilityB="900" widgetbid="qz2w9GcWnjo5" totalcountb="3"
type="514" documentation="" linewidth="none" >
          <linepath>
           <startpoint startx="300" starty="227" />
           <endpoint endx="207" endy="334" />
          </linepath>
         </assocwidget>
         <assocwidget indexa="1" indexb="1" visibilityA="0"
widgetaid="eWecjaiw5A6Q" visibilityB="0" roleBdoc="" roleAdoc=""
linecolor="none" changeabilityA="900" totalcounta="2"
changeabilityB="900" widgetbid="qz2w9GcWnjo5" totalcountb="3"
type="514" documentation="" linewidth="none" >
          <linepath>
           <startpoint startx="133" starty="296" />
           <endpoint endx="194" endy="334" />
          </linepath>
         </assocwidget>
         <assocwidget indexa="1" indexb="1" visibilityA="0"
widgetaid="qz2w9GcWnjo5" visibilityB="0" roleBdoc="" roleAdoc=""
linecolor="none" changeabilityA="900" totalcounta="2"
changeabilityB="900" widgetbid="ooVL8UQxxdHA" totalcountb="2"
type="514" documentation="" linewidth="none" >
          <linepath>
           <startpoint startx="201" starty="338" />
           <endpoint endx="206" endy="382" />
          </linepath>
         </assocwidget>
        </associations>
       </diagram>
      </diagrams>
     </xmi.extension>
```

```
    </UML:Model>
    <UML:Model>
.
.
.
    </UML:Model>
   </UML:Namespace.ownedElement>
  </UML:Model>
 </xmi.content>
 <xmi.extensions xmi.extender="umbrello" >
  <docsettings viewid="rVFJDDjSc0qa"
uniqueid="qz2w9GcWnjo5" documentation="" />
  <listview>
.
.
.
  </listview>
  <codegeneration>
   <codegenerator language="C++" />
  </codegeneration>
 </xmi.extensions>
</xmi>
```

We may also find a fork/join element. Their definition in XMI consists in providing the definition for the fork and joiu elements as well as their associations.

The next piece of code illustrates how to define the relationships for all the elements to be showed.

```
MODULE mainproc(eventParent)
VAR
    state:{ready,start,listening,working,stoped,end,error};

event:{readylistening,startready,listeningworking,
listeningstoped,workinglistening,stopedend};
ASSIGN
  init(state) := start;
  init(event) := startready;
  next(state) := case
   eventParent != main& state = end : start;
   eventParent != main & state != end: state;
   state = ready & event = readylistening : listening;
   state = start & event = startready : ready;
   state = listening & event = listeningworking : working;
   state = listening & event = listeningstoped : stoped;
   state = working & event = workinglistening : listening;
   state = stoped & event = stopedend : end;
   state = end : end;
   1:error;
esac;
  next(event) := case
   next( state ) = ready:readylistening;
   next( state ) = start:startready;
   next( state ) = listening:{listeningworking,
listeningstoped};
   next( state ) = working:workinglistening;
   next( state ) = stoped:stopedend;
   next( state ) = end :  event;
   1 : event;
esac;


MODULE listening1listening3fork(eventParent)
VAR
  eventlistening2-b:{listening2-bend,startlistening2-b};
  eventlistening2-a:{listening2-astatealter3,
statealter3end,startlistening2-a};
  statelistening2-b:{listening2-b,start,end,error};
  statelistening2-a:{listening2-a,statealter3,start,end,error};
ASSIGN
  init(statelistening2-b) := start;
  init(statelistening2-a) := start;
  init(eventlistening2-b) := startlistening2-b;
  init(eventlistening2-a) := startlistening2-a;
  next(statelistening2-b) := case
eventParent != listening1listening3 &
statelistening2-b = end: start;
eventParent != listening1listening3 &
statelistening2-b != end : statelistening2-b;
statelistening2-b = listening2-b &
eventlistening2-b=listening2-bend: end;
statelistening2-b = listening2-b
& eventlistening2-b !=listening2-bend:
statelistening2-b;
statelistening2-b = start &
eventlistening2-b=startlistening2-b: listening2-b;
1:error;
  esac;
  next(statelistening2-a) := case
eventParent != listening1listening3 &
statelistening2-a = end: start;
eventParent != listening1listening3 &
statelistening2-a !=end : statelistening2-a;
```

```
statelistening2-a = listening2-a &
eventlistening2-a=listening2-astatealter3:
statealter3;
statelistening2-a = statealter3 &
eventlistening2-a=statealter3end: end;
statelistening2-a = statealter3 &
 eventlistening2-a !=statealter3end:
statelistening2-a;
statelistening2-a = start &
eventlistening2-a=startlistening2-a: listening2-a;
  esac;
   next( eventlistening2-b ) := case
  next( statelistening2-b) = listening2-b &
next(statelistening2-a ) = statealter3 :listening2-bend;
  next( statelistening2-b) = start : startlistening2-b;
  next(statelistening2-b) = end: eventlistening2-b;
  1 : eventlistening2-b;
   esac;
   next( eventlistening2-a ) := case
  next( statelistening2-a) = listening2-a
:listening2-astatealter3;
  next( statelistening2-a) = statealter3 &
next(statelistening2-b ) = listening2-b  &
next(statelistening2-a ) =statealter3:statealter3end;
  next( statelistening2-a) = start : startlistening2-a;
  next(statelistening2-a) = end: eventlistening2-a;
  1 : eventlistening2-a;
   esac;
DEFINE
out := statelistening2-b = listening2-b  &
 statelistening2-a = statealter3;
startin := statelistening2-b = start &
statelistening2-a = start;

MODULE listeningproc(eventParent)
VAR
state:{listening1,listening3,start,end,
listening1listening3fork,error};

event:{listening1listening1listening3fork,
listening3end,startlistening1,
listening1listening3forklistening3};
ASSIGN
  init(state) := start;
  init(event) := startlistening1;
  next(state) := case
```

```
   eventParent != listening& state = end : start;
   eventParent != listening & state != end: state;
   state = listening1 & event =
listening1listening1listening3fork : listening1listening3fork;
   state = listening3 & event = listening3end : end;
   state = start &
event = startlistening1 : listening1;
   state = listening1listening3fork & event =
listening1listening3forklistening3 : listening3;
   state = end : end;
   1:error;
esac;
  next(event) := case
   next( state ) =
listening1:listening1listening1listening3fork;
   next( state ) = listening3:listening3end;
   next( state ) = start:startlistening1;
   next( state ) =
listening1listening3fork:listening1listening3forklistening3;
   next( state ) = end :  event;
   1 : event;
esac;


MODULE main
VAR
turn:{main,listening1listening3,listening};
pr: mainproc(turn);
listening1listening3: listening1listening3fork(turn);
listening: listeningproc(turn);

ASSIGN
  init(turn) := main;
  next(turn) := case
  next(listening.state=listening1listening3fork) &
next(listening1listening3.startin = 1)
:listening1listening3;
  listening.state=listening1listening3fork &
listening1listening3.out = 1: listening;
  next(pr.state=listening) &
next( listening.state) = start:listening;
  pr.state=listening & listening.state = end: main;
  1 : turn;
esac;
```

4.3.3 The application in the Web

The set of figures to come are from our applicaction when is visited with the Firefox Browser.

The Figure 9 is the first view of our project, like we can see, there are two parts , the left and right parts. The left is a list of all the project that are uploaded in the server, each project is a set of diagrams in UML. The right part are used to show information about the project.

There are two ways to display or choose a project, the first one is when we have listed the one that we want in the list, in that case, we just click over the name (is a hyperlink) to say to the application that we have choose that project, then the application send a AJAX request to the server and this answer with a JSON anser to be parsed and displayed with JQUERY in the information section of our page.

The second way to choose a project come when we don't see this one in the list, but we have the XMI document, in this case in the list section we make click over the "new" link, after that a JQUERY process bring us a modal formular to upload the document, like we can see in Figure 10.

When we have up loaded our project we can see it in the list section and go further like with the normal way.

Once a project is choosed we see something like in Figure 11. In the upper part at the information section, we have the name of the project, the list of files related, and a list of name of each module and the states of each one of them. Next to the list there is a formular, this formular has four inputs, three checkboxes, and a input text. finally the button to submit the formular.

The list of files, are about the original XMI file (see Figure 12), the transformation of the Diagrams into the syntax of NUSMV (see Figure 13), and finally the automatickally genarated reachability question of the system (see Figure 14)

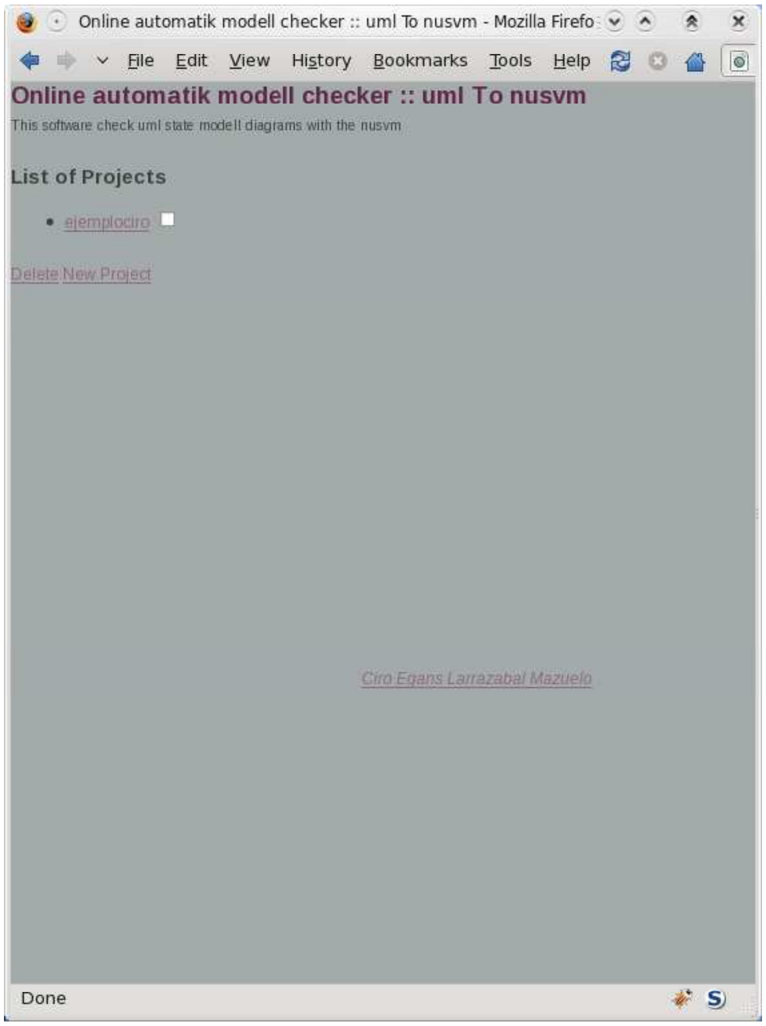
The formular, is to set the checking of properties that we want to probe in the model, the auto generated questions are in form of a checkbox, and the user defined question can be maked in the input text, the way to do that is like in a normal CTL syntax. The name of the states and the dependence can be easily obtainend from the list of states and dependances.

Once we send the formular a document is created in the server with the question of the user (if any), and apper in the list of documments. At the botton of the formular in the same section come from the server the answer from the model checker just like in Figure 15.

4.4 Shale apache project

Shale is a modern web application framework, fundamentally based on JavaServer Faces of the Appache Software foundation. Architecturally,Shale is a set of loosely coupled services that can be combined as needed to meet particular application requirements.

One of the features of shale is the dialog manager, wich is a Mechanism to define a "conversation" with a user that requires multiple HTTP requests to


Online automatik modell checker :: uml To nusvm - Mozilla Firefo
File Edit View History Bookmarks Tools Help
Done

Online automatik modell checker :: uml To nusvm

This software check uml state modell diagrams with the nusvm

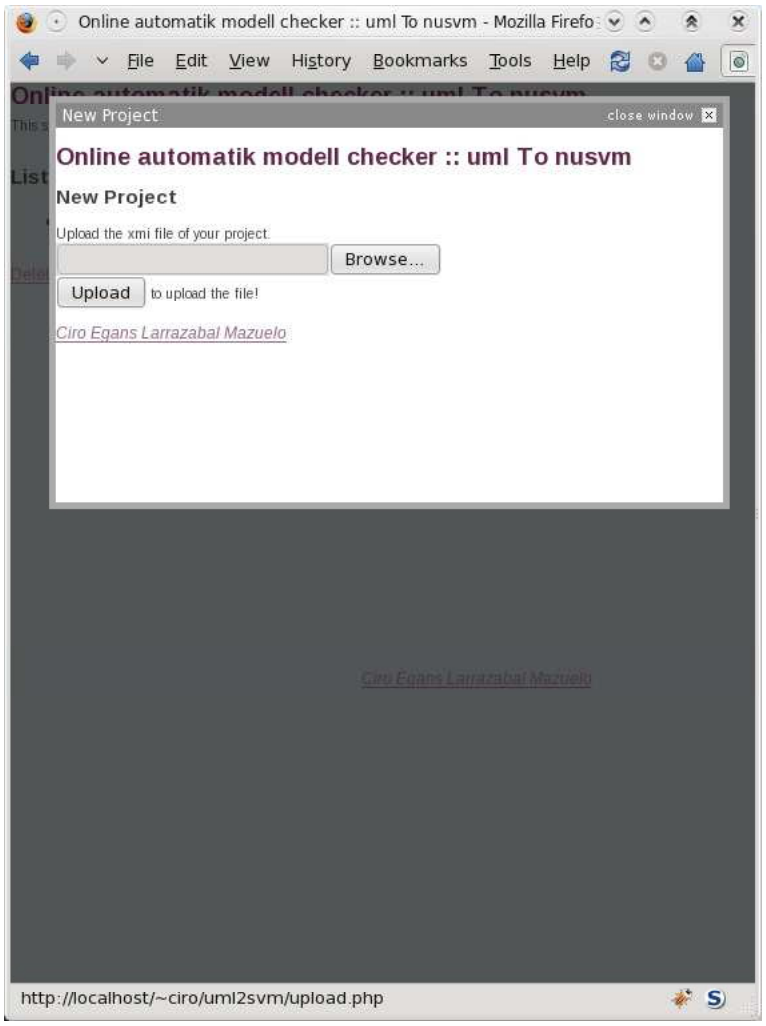
List of Projects

- ejemplociro

Delete New Project

Ciro Egans Larrazabal Mazuelo

Figure 9: Screenshot of the start point of the application


Online automatik modell checker :: uml To nusvm - Mozilla Firefo
File Edit View History Bookmarks Tools Help
New Project
close window
Browse...
Upload
http://localhost/~ciro/uml2svm/upload.php

Online automatik modell checker :: uml To nusvm

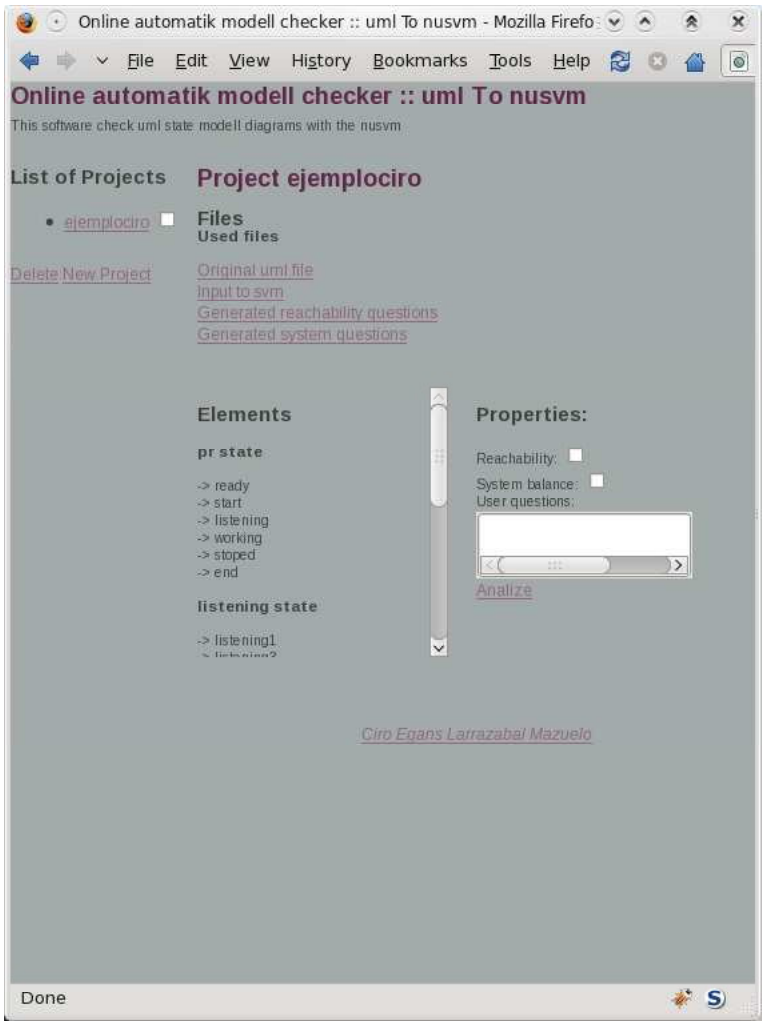
New Project

Upload the xmi file of your project.

Upload to upload the file!

Ciro Egans Larrazabal Mazuelo

Figure 10: Screenshot of the formular to upload a new UML diagram


Online automatik modell checker :: uml To nusvm - Mozilla Firefo
File Edit View History Bookmarks Tools Help
Done

Online automatik modell checker :: uml To nusvm

This software check uml state modell diagrams with the nusvm

List of Projects

- ejemplociro

Delete New Project

Project ejemplociro

Files

Used files

Original uml file
Input to svm
Generated reachability questions
Generated system questions

Elements

pr state

-> ready
-> start
-> listening
-> working
-> stoped
-> end

listening state

-> listening1

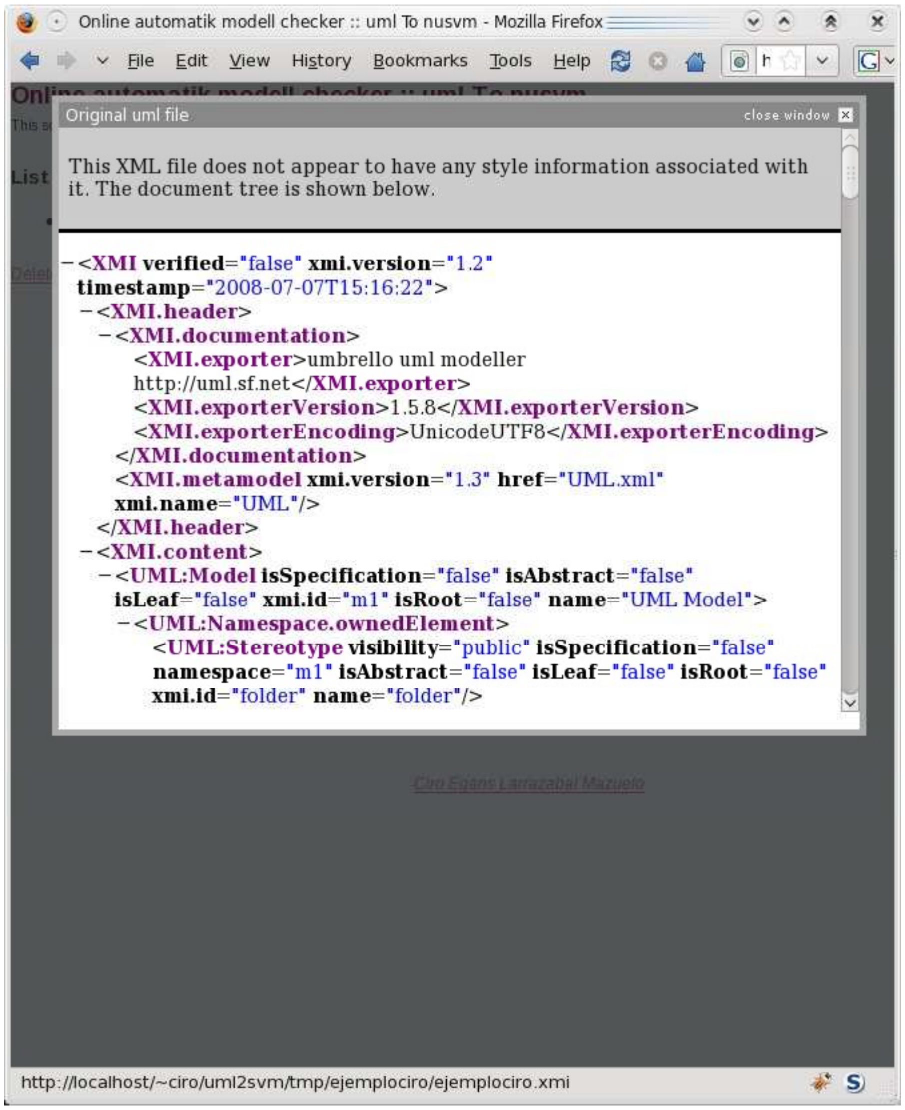
Properties:

Reachability:
System balance:
User questions:

Analize

Ciro Egans Larrazabal Mazuelo

Figure 11: Screenshot of the project after the selection of a set of diagrams

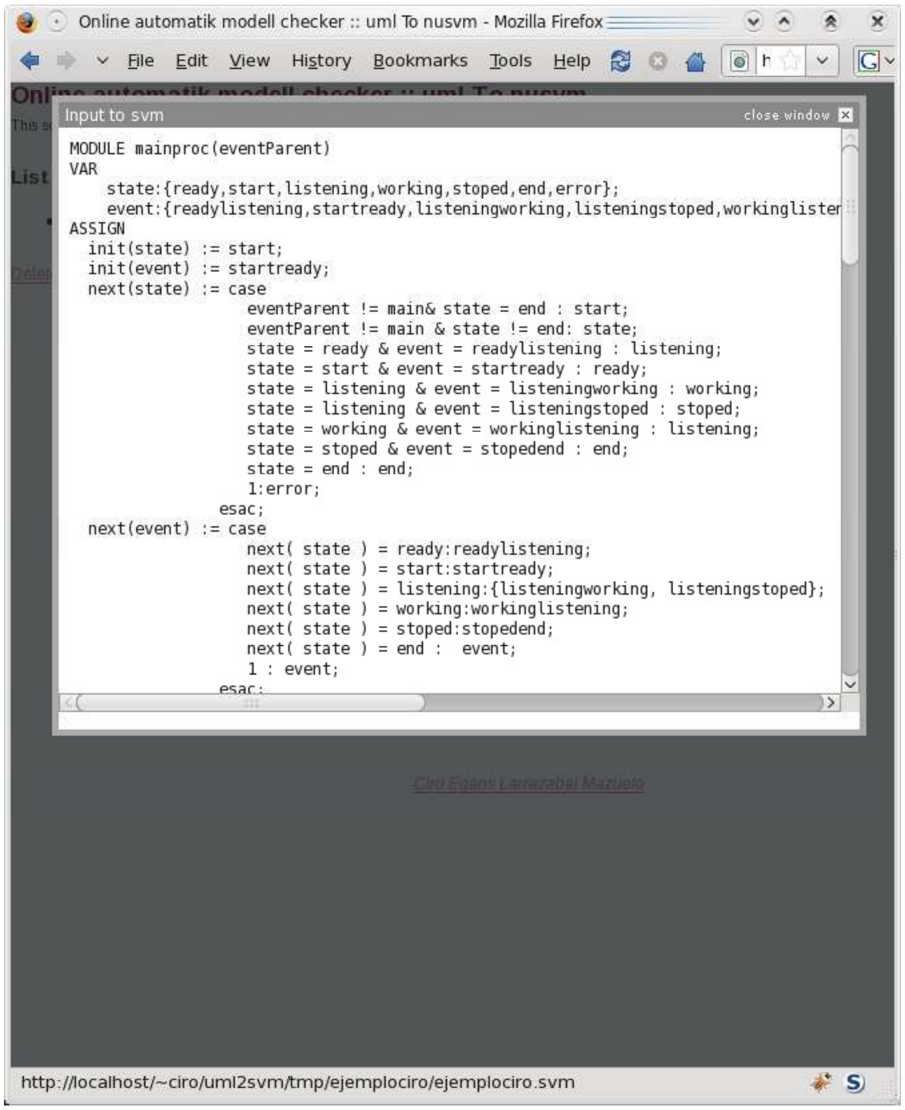

Online automatik modell checker :: uml To nusvm - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Original uml file
close window

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
-<XMI verified="false" xmi.version="1.2"
 timestamp="2008-07-07T15:16:22">
 -<XMI.header>
  -<XMI.documentation>
     <XMI.exporter>umbrello uml modeller
     http://uml.sf.net</XMI.exporter>
     <XMI.exporterVersion>1.5.8</XMI.exporterVersion>
     <XMI.exporterEncoding>UnicodeUTF8</XMI.exporterEncoding>
   </XMI.documentation>
   <XMI.metamodel xmi.version="1.3" href="UML.xml"
   xmi.name="UML"/>
  </XMI.header>
 -<XMI.content>
  -<UML:Model isSpecification="false" isAbstract="false"
   isLeaf="false" xmi.id="m1" isRoot="false" name="UML Model">
   -<UML:Namespace.ownedElement>
      <UML:Stereotype visibility="public" isSpecification="false"
      namespace="m1" isAbstract="false" isLeaf="false" isRoot="false"
      xmi.id="folder" name="folder"/>
```

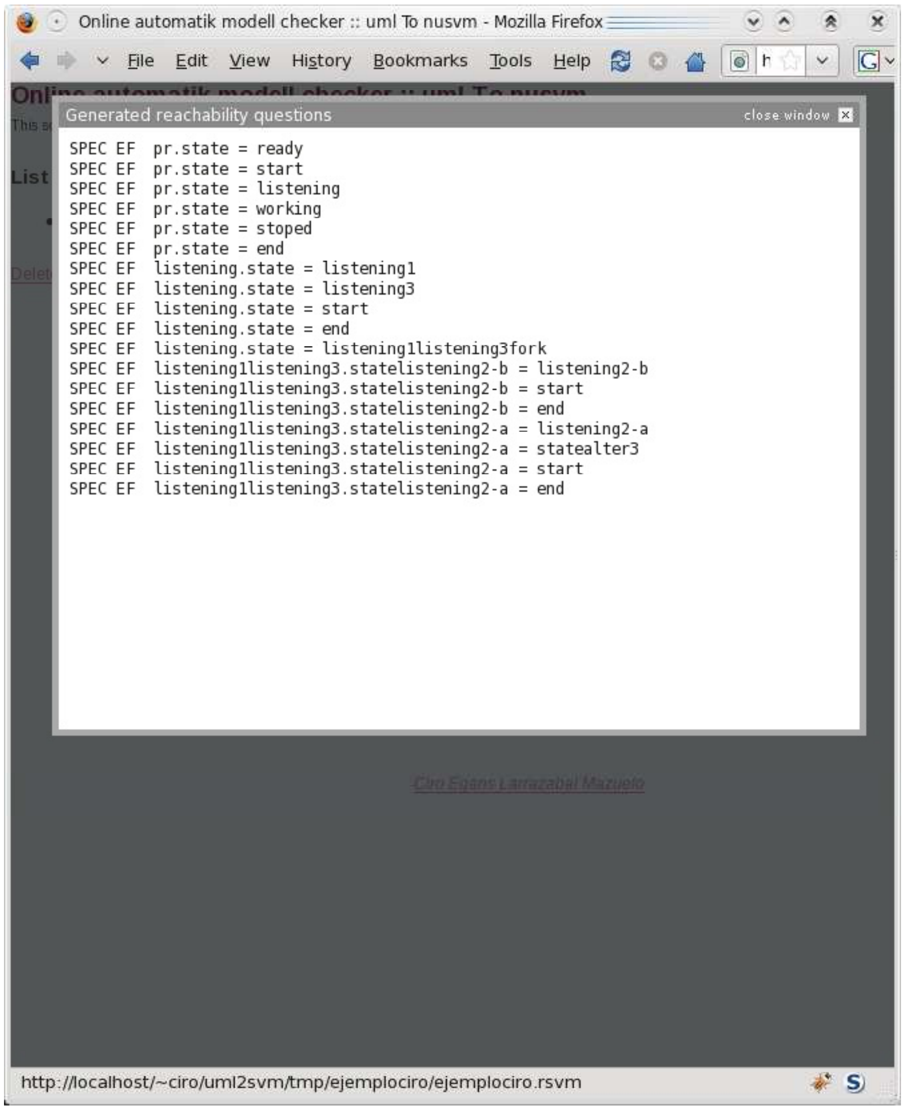
http://localhost/~ciro/uml2svm/tmp/ejemplociro/ejemplociro.xmi

Figure 12: Screenshot of the project when we choose to see the original XMI


Online automatik modell checker :: uml To nusvm - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Input to svm
close window
Ciro Egans Larrazabal Mazuelo
http://localhost/~ciro/uml2svm/tmp/ejemplociro/ejemplociro.svm

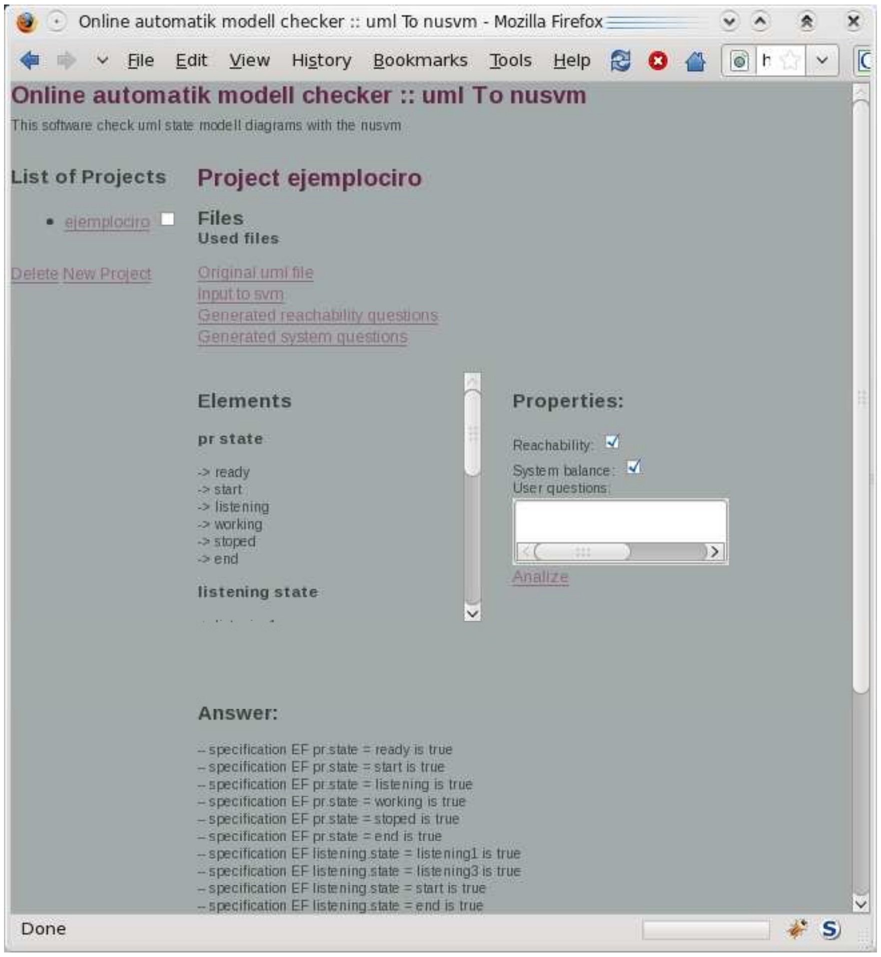
```
MODULE mainproc(eventParent)
VAR
    state:{ready,start,listening,working,stoped,end,error};
    event:{readylistening,startready,listeningworking,listeningstoped,workinglisten
ASSIGN
  init(state) := start;
  init(event) := startready;
  next(state) := case
                   eventParent != main& state = end : start;
                   eventParent != main & state != end: state;
                   state = ready & event = readylistening : listening;
                   state = start & event = startready : ready;
                   state = listening & event = listeningworking : working;
                   state = listening & event = listeningstoped : stoped;
                   state = working & event = workinglistening : listening;
                   state = stoped & event = stopedend : end;
                   state = end : end;
                   1:error;
                esac;
  next(event) := case
                   next( state ) = ready:readylistening;
                   next( state ) = start:startready;
                   next( state ) = listening:{listeningworking, listeningstoped};
                   next( state ) = working:workinglistening;
                   next( state ) = stoped:stopedend;
                   next( state ) = end :  event;
                   1 : event;
                esac;
```

Figure 13: Screenshot of the project when we choose the input file for NUSVN


Online automatik modell checker :: uml To nusvm - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Generated reachability questions
close window
Ciro Egans Larrazabal Mazuelo
http://localhost/~ciro/uml2svm/tmp/ejemplociro/ejemplociro.rsvm

```
SPEC EF  pr.state = ready
SPEC EF  pr.state = start
SPEC EF  pr.state = listening
SPEC EF  pr.state = working
SPEC EF  pr.state = stoped
SPEC EF  pr.state = end
SPEC EF  listening.state = listening1
SPEC EF  listening.state = listening3
SPEC EF  listening.state = start
SPEC EF  listening.state = end
SPEC EF  listening.state = listening1listening3fork
SPEC EF  listening1listening3.statelistening2-b = listening2-b
SPEC EF  listening1listening3.statelistening2-b = start
SPEC EF  listening1listening3.statelistening2-b = end
SPEC EF  listening1listening3.statelistening2-a = listening2-a
SPEC EF  listening1listening3.statelistening2-a = statealter3
SPEC EF  listening1listening3.statelistening2-a = start
SPEC EF  listening1listening3.statelistening2-a = end
```

Figure 14: Screenshot of the generated questions


Online automatik modell checker :: uml To nusvm - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Done

Online automatik modell checker :: uml To nusvm

This software check uml state modell diagrams with the nusvm

List of Projects

- ejemplociro

Delete New Project

Project ejemplociro

Files

Used files

Original uml file
Input to svm
Generated reachability questions
Generated system questions

Elements

pr state

-> ready
-> start
-> listening
-> working
-> stoped
-> end

listening state

Properties:

Reachability: ✓
System balance: ✓
User questions:

Analize

Answer:

```
-- specification EF pr.state = ready is true
-- specification EF pr.state = start is true
-- specification EF pr.state = listening is true
-- specification EF pr.state = working is true
-- specification EF pr.state = stoped is true
-- specification EF pr.state = end is true
-- specification EF listening.state = listening1 is true
-- specification EF listening.state = listening3 is true
-- specification EF listening.state = start is true
-- specification EF listening.state = end is true
```

Figure 15: Screenshot of the project when we execute the questions

implement, modeled as a state diagram.

SCXML or State Chart extensible Markup Language is a draft of the W3C to provide a generic state-machine based execution enviroment based on Harel State Tables [16].

This language is used for shale to implement its dialog manager.

The interaction of the systems reflect the state diagram thus if we can check the diagram we check the system self. The diagram is very similar with the one of XMI, and we can easily make a converter with a XSLT Document like the next one.

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:UML="org.omg/standards/UML" version="1.0">
<xsl:output method="text"/>
<xsl:strip-space elements="*"/>
<xsl:template match="XMI.header"/>

<xsl:template match="diagrams">
  <xsl:text > &lt;scxml xmlns="http://www.w3.org/2005/07/scxml"
version="1.0" initialstate="reset"&gt;
</xsl:text>
  <xsl:call-template name="statesName">
   <xsl:with-param name="namediagram"
select="diagram[@name='state diagram']/@name" />
   <xsl:with-param name="diagramList"
select="diagram[@type=5]" />
 </xsl:call-template>
</xsl:template>

<xsl:template name="statesName" >
  <xsl:param name="namediagram"/>
  <xsl:param name="diagramList"/>
  <xsl:for-each select="$diagramList[@name=$namediagram]">
    <xsl:variable name="assocVar" select="associations"/>
    <xsl:variable name="stateVars" select="widgets"/>
    <xsl:for-each select="$stateVars/statewidget">

<xsl:text disable-output-escaping="yes">    &lt;state id="</xsl:text>
  <xsl:variable name="stateNameVar" select="@statename"/>
  <xsl:call-template name="giveName">
    <xsl:with-param name="node" select="$stateNameVar"/>
    <xsl:with-param name="type" select="@statetype"/>
    </xsl:call-template>
    <xsl:text disable-output-escaping="yes">"&gt;
</xsl:text>
    <xsl:variable name="org" select="@xmi.id"/>
    <xsl:variable name="ccc"
select="count($assocVar/assocwidget[@widgetaid=$org])" />
    <xsl:if test=" $ccc = 0" ></xsl:if>
    <xsl:for-each select="$assocVar/assocwidget[@widgetaid=$org]">
     <xsl:call-template name="eventName">
      <xsl:with-param name="fuente" select="$stateVars"/>
      <xsl:with-param name="origen" select="$org"/>
      <xsl:with-param name="destino" select="@widgetbid"/>
     </xsl:call-template>
     </xsl:for-each>
    <xsl:if test="count($diagramList[@name=$stateNameVar]) &gt; 0">
     <xsl:call-template name="statesName">
      <xsl:with-param name="namediagram" select="$stateNameVar" />
      <xsl:with-param name="diagramList" select="$diagramList" />
     </xsl:call-template>
    </xsl:if>
    <xsl:text disable-output-escaping="yes">    &lt;/state&gt;
</xsl:text>
  </xsl:for-each>
</xsl:for-each>
</xsl:template>

<xsl:template name="giveName">
  <xsl:param name="node"/>
  <xsl:param name="type"/>
  <xsl:choose>
   <xsl:when test="$type=0">start</xsl:when>
   <xsl:when test="$type=2">end</xsl:when>
<xsl:otherwise><xsl:value-of select="$node"/></xsl:otherwise>
  </xsl:choose>
</xsl:template>

<xsl:template name="eventName">
  <xsl:param name="fuente"/>
  <xsl:param name="origen"/>
  <xsl:param name="destino"/>
  <xsl:text disable-output-escaping="yes"
>        &lt;transition event="</xsl:text>
  <xsl:call-template name="giveName">
  <xsl:with-param name="node"
select="$fuente/statewidget[@xmi.id=$origen]/@statename"/>
  <xsl:with-param name="type"
select="$fuente/statewidget[@xmi.id=$origen]/@statetype"/>
  </xsl:call-template>
  <xsl:call-template name="giveName">
  <xsl:with-param name="node"
```

```
select="$fuente/statewidget[@xmi.id=$destino]/@statename"/>
  <xsl:with-param name="type"
select="$fuente/statewidget[@xmi.id=$destino]/@statetype"/>
  </xsl:call-template>
  <xsl:text disable-output-escaping="yes">"   target="</xsl:text>
  <xsl:call-template name="giveName">
  <xsl:with-param name="node"
select="$fuente/statewidget[@xmi.id=$destino]/@statename"/>
  <xsl:with-param name="type"
select="$fuente/statewidget[@xmi.id=$destino]/@statetype"/>
  </xsl:call-template>
  <xsl:text disable-output-escaping="yes">"/&gt;
</xsl:text>
</xsl:template>

</xsl:stylesheet>
```

This XSLT file convert a UMl file in a SCXML document and, a similar one can convert this into a NUSMV file. Wich is susceptible to be automatic checked, but only if the diagram doesn't include join other forks. A more elaborate program like the one of our project could be wrote to do that conversion.

5 Conclusions

An abstract model has been prepared in order to represent an UML diagram in the model checker of our choice. This model did not work for all the models, but the reason is related to unsufficient information that underlays in the standard definition. It is very hard to foresee this before the development itself.

Maybe a major standard, or an extension of the same, could be more precise. It may also bring us a better source of information to feed the model checker.

Up to this point, the creation of a model to transform a UML state diagrams to a set of automaton that will be checked in a model checker was succesful. Anyway, for instance, very big systems with a lot of connections (with restrictions about a fork creating a set of states without overlapping) are easily susceptible to automatic checking. This indicates the model is suitable for further formalizations.

One of the uses of model checking, the formal verification, easily allows big systems to be put under a continuous checking according to a certain certification path.

Another use for our tool can be done with the extension of the SCXML document. With this inclusion we have, at least for diagrams without joins, totally auto-certificated models, which would be built from the mathematical model.

References

[1] ISO/IEC 10646-1. Universal multiple-octet coded character set (ucs). Technical report, ISO (International Organization for Standardization), 2000.

[2] RFC 3066. Tags for the identification of languages. Technical report, IETF (Internet Engineering Task Force), 2001.

[3] ISO 639. Code for the representation of names of languages. Technical report, ISO (International Organization for Standardization)., 1988.

[4] ISO 8879. Codes for the representation of names of countries and their subdivisions. Technical report, ISO (International Organization for Standardization)., 1997.

[5] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P Schnoebelen. *Systems and Software Verification.* Springer, 2001. Original French edition published by Vuibert, Paris, 1999.

[6] Taylor Booth. *Sequential Machines and Automata Theory.* John Wiley and Sons, 1967.

[7] David Braun, Jeff Sivils, Alex Shapiro, and Jerry Versteegh. Unified modeling language (uml) tutorial. Technical report, Object Oriented Analysis and Design Team, 2001.

[8] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986.

[9] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent system using temporal logic specifications: a practical approach. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 117–126, New York, NY, USA, 1983. ACM.

[10] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[11] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[12] M. Edmund Clarke, Grumberg jr Orna, and Peled Doron A. *Model checking.* Cambridge, Mass., 2 edition, 1999.

[13] The Unicode Consortium. The unicode standard, version 2.0. Technical report, The Unicode Consortium, 1996.

[14] A. Pnueli. The temporal logic of programs. In *In Proceedings of the 18th IEEE Symposium Foundations of Computer Science*, pages 46–57, 1977.

[15] w3c Foundation. Extensible markup language (xml) 1.0 (fourth edition).

[16] w3c Foundation. State chart xml (scxml): State machine notation for control abstraction.

[17] w3c Foundation. Xml path language (xpath) version 1.0.

[18] w3c Foundation. Xsl transformations(xslt) version 1.0.