UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

AMC - Tool support for automating Model Checking Lifecycle

Proyecto Final de Carrera

Ingeniería Superior Informática

Autor: Luis Enrique López Pons

Director: Santiago Escobar

Abril 2013

Resumen

In model checking, the most time consuming stage is oftentimes not writing the formal model (of the system to be analyzed) itself but validating the model: ensuring the correct system has been modeled, before ensuring that the system is correct. This is due to the fact that ancillary tool support for formal verification has been chronically inadequate, especially in processing and understanding the output from large and complex models. We present an integrated tool framework, called AMC and implemented in Java, to aid the model checking process in all its stages. AMC consists of a parameterized automated model generator, a translator from unstructured model checking output to a general XML trace format, an execution trace visualizer, an interactive model simulator, and an automated results analyzer that produces PDF reports from the model checking runs. As a case study, we use a clock synchronization algorithm recently developed at NASA and the evmdd-smc model checker developed at NIA.

Palabras clave: model checking, formal methods, clock synchronization, data visualization, edge value multi decision diagrams

Contents

Chapter 1

Introduction

1.1 Motivation

This project has been developed at National Institute of Aerospace (NIA) together with NASA in Hampton, Virginia (USA) from July to November 2011, thanks to the connection of research professionals in the field of Model Checking. After attending the course ”Métodos Formales en Sistemas de la Información”, given in the 5th course of Ingeniería Superior Informática in the Escuela Técnica Superior de Informática, I had the opportunity to visit NIA for an internship.

The initial offer was to develop methods and tools for the formal verification and automated testing of hybrid systems and to apply these methods to diagnostic and monitoring systems. Although my background in avionics and formal methods was really basic beforehand, working in such important environment supporting NASA technologies and learning cutting edge systems encouraged me a lot. Moreover, the different types of applications of Model Checking in several fields and facing a big challenge in an unknown environment was a big plus.

This report details the work done during the 4 months of stage.

1.2 Scope

”Systems must be dependable (available, trustworthy, safe and maintainable) and secure (integrity and confidentiality)” [1]. However, the task of controlling that a system accomplishes it, is not trivial, since it requires an exhaustive verification that most testing techniques do not apply. Therefore, formal methods are used to exploit the power of mathematical notation and proofs, and being supported by automated tools [2]. Inside the formal methods field, we find the model checking technique, which aims to verify systems deeply and exhaustively by checking all the possible paths to the states that accomplish a property.

Such model checking technique is the core of the project presented in this document, which aims to automate and support the model checking lifecycle by identifying weak points in the technique and by creating tools to help the user to validate and verify her system.

In Figure 1.1, we show, in the beginning, the real system that has to be exhaustively checked. First of all, it is modeled in the specification language of the model checker. In order to do that, it is necessary to create a formal model and define one or more properties to be checked. Nevertheless, the output of model checkers is usually very difficult to read and even worse to analyze. This is because it is a plain text file with the trace to the state that accomplishes the

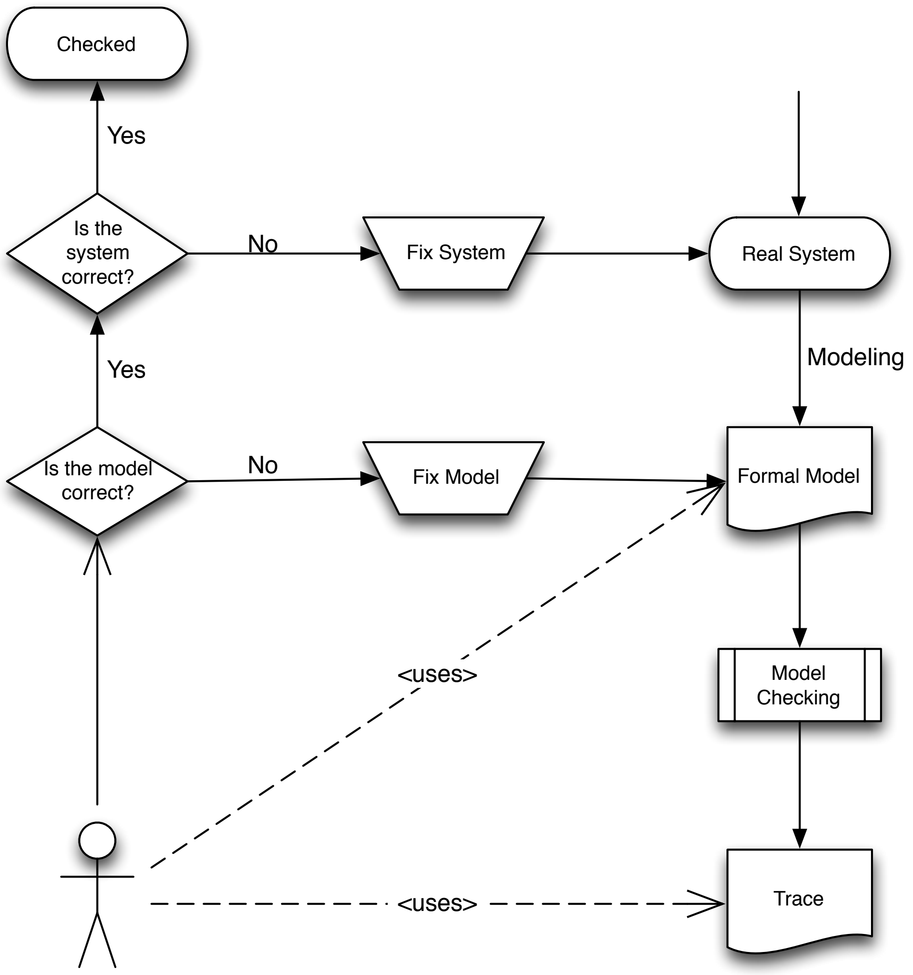


Figure 1.1: Model Checking lifecycle diagram

property, which means a trace with the value of all the variables in each state until it reaches the final state that matches the property. Indeed, this is a lot of information that is not often structured and makes really tough the task of validating and verifying the model and checking that the system works properly.

As we have seen, the Model Checking lifecycle diagram represents graphically all the stages of the model checking process, where it is possible to see that it is not easy to do the following tasks:

- **Validation**. Are we modeling the correct system?
- **Verification**. Is the system correctly modeled?
- **Check** that the **system specification** itself is correct
- **Interpret** the model checker output
- Output **analysis**

Thus, the main topic of this project is answering the following question:

How is it possible to improve the Model Checking process?

1.3 Objectives

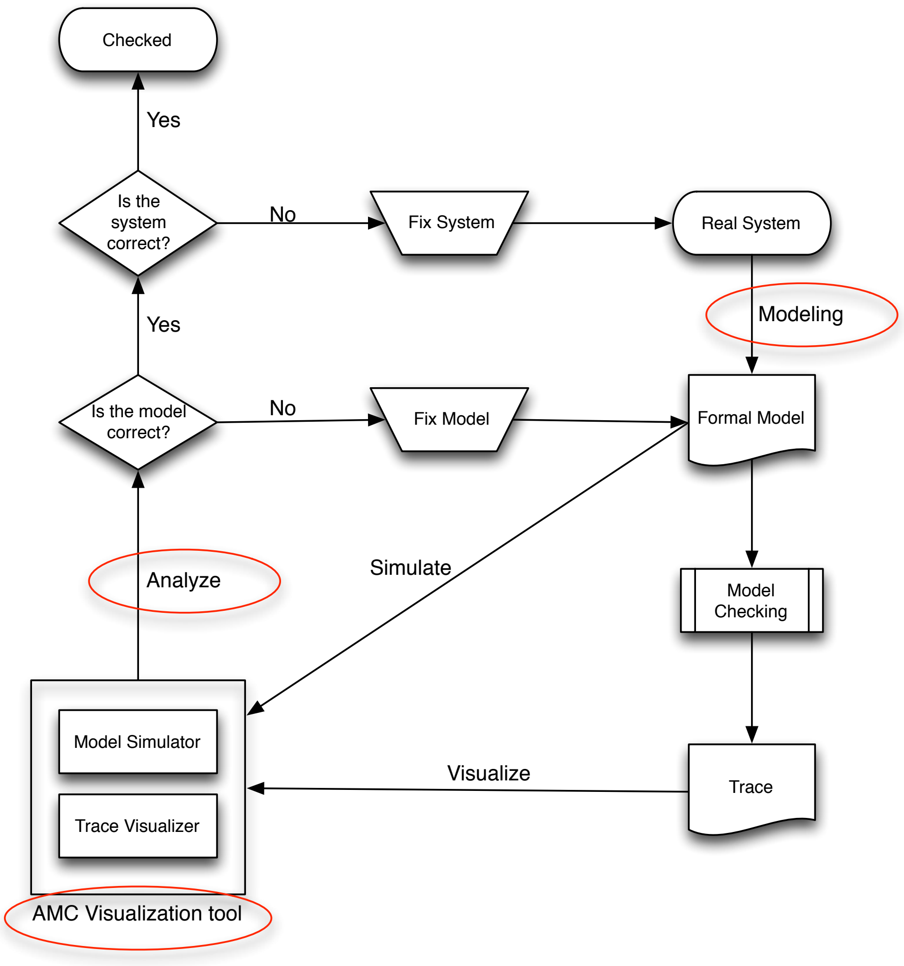


Figure 1.2: Automatized lifecycle diagram

The main objective is to improve the Model Checking Lifecycle, developing a set of tools to support the difficult tasks mentioned in the last section. In order to reach the objectives and build the tools, we have guided our effort using a relevant case study, that allowed us to continually assess the usability of the framework: the Distributed Clock Synchronization Protocol for Arbitrary Digraph, developed at NASA LarC by Mahyar Malekpour [3]. We have selected the EVMDD model checker, developed at NIA, to verify the models, due to its superior performance and minimalist syntax. So, in the modeling phase (see Figure 1.2), because of the simple syntax of EVMDD-smc, a tool for automating the creation of the different mathematical models of the real system, called Model Generator in this work, is required. In this case, it generates various instantiations of the clock synchronization protocol, for different protocol parameters.

Further on, the model checker of choice, EVMDD-smc, generates either information about how many states satisfy a property, or an output trace to one of the states that invalidates the property (a counter-example). Besides, execution traces and counter-examples for large, complex systems, are difficult to read. Therefore, one of our major goals is to visualize clearly the states of an execution trace, organizing the variables in groups and sections in order to make the trace output much more readable. Such task is performed by the Trace Visualizer.

Moreover, in order to validate and verify complex models that cannot be exhaustively verified with a model checker, due to state-space explosion and memory consumption limits, we create a Model simulator to check the states and transitions of the model.

In Figure 1.2 it is possible to see how the AMC tool is applied in the model checking lifecycle. This tool integrates in its GUI the Trace Visualizer and the Model Generator, since both explore states. Furthermore, the Trace Visualizer aims to admit traces from every model checker that follows a template and to translate the EVMDD output trace into the standard format. However, the Model Simulator only admits EVMDD models, since each model checker has its own modeling language.

Finally, the resulting information needs to be interpreted. Such information is the number of states accomplishing the properties of the protocol. So, it is important to analyze it to see whether the model is valid for the protocol and works properly. In order to do it, a tool that automates this analysis has been created, together with scripts that generates output for each topology in study. The results of the verification are synthesized in PDF reports created from the EVMDD model checker output during this project. This tool that creates reports is called Result Analyzer

In conclusion, the tools created to improve the Model Checking Lifecycle are:

1. EVMDD model generator
2. EVMDD model simulator
3. EVMDD Trace translator
4. Trace output visulizer
5. Result Analyzer

1.4 Methodology

The development of the project has been around the case study: Verification of the Distributed Clock Synchronization Protocol for Arbitrary Digraph. However, in the beginning of the project, the requirements were not completely fixed and have been changing progressively during the development. So, the methodology has been really agile and flexible. This agile procedure reviewing the work done every two weeks with my mentor and having continuous meetings with the creator of the protocol, has helped me to have a constant feedback from them and from the work in progress.

First of all, I researched about the model checking output and how it could be formatted to be easily monitorized. At the same time, I studied in depth the protocol to be able to create a model. Then, I started creating a basic model of the protocol with only two nodes and started designing the Trace Visualizer. Firstly, only for the EVMDD output and later adapted to a XML template, creating a EVMDD trace translator. Yet created the first basic model, I wrote the model generator to create bigger instantiations and at the same time finished a first prototype of the Trace Visualizer to monitorize the EVMDD output clearly. In that way, I could take advantage of the visualization tool already. Afterwards, when the first version of the Trace Visualize was finished, yet reading from XML templates, I began the Model Simulator. There I had to adapt the EVMDD grammar that was already implemented in Bison for C to Java. It took me longer than expected, since Java CC, the compiler that I used for Java, did not admit left recursion. Once I finished the Model Simulator, I had another tool that helped me in the process of validating the model, what lasted until the end of the stage. Finally, I created the Result Analyzer that allowed displaying in a report, for each topology, the precision achieved, the time and the complexity, to validate and verify the model.

1.5 Structure of the document

This document presents the introduced AMC tool support for model checkers and the case study of the model for the Distributed Clock Synchronization protocol for arbitrary digraph. Firstly, a basic background about model checking and EVMDD model checker will be given in Chaper 2. Then, the AMC visualization tools (EVMDD simulator and trace visualizer) will be explained in detail in terms of requirements, design and implementation decisions in Chapters 3 and 4. In Chapter 5, the case of study of the EVMDD model for the protocol will be presented in depth, including the model generator and the result analyzer.

Chapter 2

Background

2.1 Model Checking

Model checking has been used since 1983 to model systems in order to verify them deeply and exhaustively [4]. In order to do it, it is necessary to follow a process

1. **Modeling**: Create a mathematical model of the real system.
2. **Specification**: Define a set of properties to be satisfied.
3. **Verification**: It is automatically done.
    - Checking all the possible paths to the states that accomplish the property. (reachability analysis)
    - Checking the paths of the language accepted by the automata generated from the model to see if the property is satisfied (automata based analysis).

2.2 EVMDD

EVMDD-smc is a model checker that implements Edge Valued Multi Decision Diagrams and was developed at NIA [5]. It is a symbolic model checker that uses the following advanced tools:

1. EVMDDs for encoding arithmetic expressions

2. Identity-reduced MDDs for representing the transition relation

3. The saturation algorithm for reachability analysis

Symbolic model checkers such as (Nu)SMV or SAL are based on the library CUDD which offers an efficient implementation of BDDs and MTBDDs. For state space generation, they use a plain breadth first search (BFS) algorithm. However, EVMDD-smc is implemented with a new symbolic model checking library featuring EVMDDs for transition relation construction and saturation for state space generation. So, it comprises 7K lines of ANSI-C code for the EVMDD library and 4K lines for the model checking interface.

In the technical report [5], it is possible to see that EVMDD tool is several orders of magnitude faster in many cases. In order to prove that, it shows execution times for finding deadlocks on a suite of classical models. The search for deadlocks in a deadlock-free system is equals to building the state space.

2.2.1 EVMDD code structure

EVMDD does not allow types, arrays or other complex structures, only global, integer and simple variables. Such lightness makes models easier to compute. However, it is more difficult to write a model manually, because of the lack of arrays or other similar data structures.

In order to show the structure of an EVMDD model of a system, we present the "Dinning philosophers" problem [6] modeled with EVMDD for three philosophers.

Declarations *variable [min,max]*

```
Declarations
   fork0  [0, 1]
   phil0  [0, 4]
   fork1  [0, 1]
   phil1  [0, 4]
   fork2  [0, 1]
   phil2  [0, 4]
```

Initial states *variable=initial value*

```
Initial states
   fork0 = 0
   phil0 = 0
   fork1 = 0
   phil1 = 0
   fork2 = 0
   phil2 = 0
```

Transitions *condition ⇒ next step*

```
Transitions
  phil0 = 0 -> phil0' = 1
  phil0 = 1 /\ fork0 = 0 -> phil0' = 2 /\ fork0' = 1
  phil0 = 1 /\ fork1 = 0 -> phil0' = 3 /\ fork1' = 1
  phil0 = 2 /\ fork1 = 0 -> phil0' = 4 /\ fork1' = 1
  phil0 = 3 /\ fork0 = 0 -> phil0' = 4 /\ fork0' = 1
  phil0 = 4 -> phil0' = 0 /\ fork0' = 0 /\ fork1' = 0
  phil1 = 0 -> phil1' = 1
  phil1 = 1 /\ fork1 = 0 -> phil1' = 2 /\ fork1' = 1
  phil1 = 1 /\ fork2 = 0 -> phil1' = 3 /\ fork2' = 1
  phil1 = 2 /\ fork2 = 0 -> phil1' = 4 /\ fork2' = 1
  phil1 = 3 /\ fork1 = 0 -> phil1' = 4 /\ fork1' = 1
  phil1 = 4 -> phil1' = 0 /\ fork1' = 0 /\ fork2' = 0
  phil2 = 0 -> phil2' = 1
  phil2 = 1 /\ fork2 = 0 -> phil2' = 2 /\ fork2' = 1
  phil2 = 1 /\ fork0 = 0 -> phil2' = 3 /\ fork0' = 1
  phil2 = 2 /\ fork0 = 0 -> phil2' = 4 /\ fork0' = 1
  phil2 = 3 /\ fork2 = 0 -> phil2' = 4 /\ fork2' = 1
  phil2 = 4 -> phil2' = 0 /\ fork2' = 0 /\ fork0' = 0
```

Properties *property to check in LTL*

```
//Checking deadlocks
!EX(true)
```

Due to the complexity of creating a model for each case, a model generator is necessary to write general models. This one would be written in a real programming language (such as Java, C) to complement the minimalist syntax (see Section 5.4).

Chapter 3

Requirements specification

The requirements specification follows the IEEE Recommended Practice for Software Requirements Specifications [7]. The principal objective of this chapter is to explain the functionalities of the AMC framework, both generally and in detail. Moreover, the non functional requirements related with performance, mantainability and portability are explained together with the constraints of the system.

3.1 Introduction

The AMC framework consists of different tools that help in the model checking process. The AMC visualization tool aims to simulate an EVMDD model and to display clearly a trace output. So, it allows the user to navigate among the trace and to organize the variables in order to analyze the model. Therefore, there are two modules: the trace visualizer and the model simulator. Apart from this, there is an EVMDD model generator and a report generator to interpret the information of the number of states accomplishing a property.

Consequently, before designing and developing each tool, it is necessary to know clearly what specifications they are going to have, both functional and non fuctional requirements. So, they must be listed in detail, in order to design, develop and test correctly every tool in the AMC framework. Such specifications have been defined according to the recommendations of my supervisor. Moreover, it is really important to know what users the tool will have. All this will be presented in this chapter.

3.2 Overall requirements description

The general requirements for each tool are listed in this section.

3.2.1 AMC Visualization Tool

Simulator

1. Simulate the state space of the model
    (a) Load Model

   (b) Display the state space
   (c) Highlight changes in the values from one state to the next one

2. Navigate among the states
   (a) Forwards
      i. Choose the transition
      ii. A random transition is chosen
   (b) Backwards
   (c) Go to initial state

Trace visualizer

1. Visualize the trace of a counterexample.
   (a) Load a trace
   (b) Display the variables of the initial state and their values
   (c) Highlight changes in the values from one state to the next one
2. Navigate among the states
   (a) One step forward
   (b) Interval step forward
   (c) One step backward
   (d) Interval step backward
   (e) Go to initial state
   (f) Go to last state of the trace
   (g) Go to a selected state in the trace
   (h) Play steps forward automatically
3. Trace visualizer settings
   (a) Set the size of the interval step
   (b) Save configuration

General settings

1. Organize the groups in sections
   (a) List sections and their corresponding groups
   (b) Create a new section
   (c) Change section name
   (d) Delete a section

   (e) Change a group to a different section
   (f) Change visibility of groups

2. Organize the variables in groups
   (a) List groups and their corresponding variables
   (b) Create a new group
   (c) Change group name
   (d) Delete groups
   (e) Change a variable to a different group
   (f) Change visibility of variables

3.2.2 Model Generator

1. Generate a EVMDD model of the Clock Synchronization Protocol

3.2.3 Result Analyzer

1. Generate a PDF report with Clock Synchronization Protocol results

3.3 User characteristics

3.3.1 Simulator

The user has to:

- be able to provide an EVMDD model
- understand the basic concepts of model checking
- know what the model does in order to be able to analyze it.

3.3.2 Trace visualizer

The user has to be able to provide:

- a trace output of a model according to the XML output template (See Appendix B)
- a trace output of a EVMDD model.
- know what the corresponding model does in order to be able to analyze it.

3.3.3 Model Generator

The user has to know the system in depth to introduce correctly the parameters to obtain the desired model.

3.3.4 Result Analyzer

The user has to understand the protocol to be able to interpret the information displayed in the report.

3.4 Constraints

3.4.1 Simulator

- The model must be an EVMDD model.
- A maximum of 1000 states can be simulated, in order to avoid too much memory consumption.

3.4.2 Trace visualizer

The output trace file must follow the structure of the following XML template file. See Apendix B

3.4.3 Model Generator

It is valid only for the Clock Synchronization protocol (See Section 5).

3.4.4 Result Analyzer

- It is valid only for the Clock Synchronization protocol (See Section 5).
- The report must be compiled in PDF Tex.

3.5 Functional requirements

In this section, the requirements description listed in Section 3.2 are explained in detail. Each table corresponds to a requirement in that list.

3.5.1 AMC Visualization Tool

Simulator

1. Simulate the states of the model

Id	1.1
Use Case	Load Model
Description	Visualize the variables of the initial state and their values. The variables will be organized in preset groups and these ones in preset sections.
Precondition	The variable names must have the identification for the preset group in the following form *name_group*. If there is no group, they will be assigned to a global one.
Input	EVMDD model. .sm file
Output	Table displaying the values of the variables organized in the predefined groups and these ones in the predefined sections.
Error	Error in the model or error in the file extension .sm

Id	1.2
Use Case	Visualize simulated states
Description	Visualize the variables of the simulated states and their values. The variables will be organized in preset groups and these ones in preset sections.
Precondition	The user must have selected a concrete or random transition from the available ones.
Input	Values of a state of the model and an available transition according to those values.
Output	Table displaying the values of the variables of the simulated state.
Error	There are no available transitions.

Id	1.3
Use Case	Highlight changes in the values.
Description	Highlight the values of the variables that have changed in the new simulated state.
Precondition	A new state has been simulated.
Input	State values and a transition.
Output	Values that have changed in the new state highlighted.
Error	-

2. Navigate among the states

Id	2.1
Use Case	Simulate one state forward
Description	Visualize the values of the variables of the next state according to a transition. The variables will be organized in groups and these ones in sections.
Precondition	The user must have selected a transition from the available ones or a random transition is selected.
Input	Values of a state of the model and an available transition according to those values.
Output	Table displaying the values of the variables of the simulated state.
Error	There are no available transitions.

Id	2.2
Use Case	Simulate one state backward
Description	Visualize the values of the variables of the last simulated state. The variables will be organized in groups and these ones in sections.
Precondition	It is not the initial state.
Input	-
Output	Table displaying the values of the variables of the simulated state.
Error	-

Id	2.3
Use Case	Go to the initial state
Description	Visualize the values of the variables of the first state. The variables will be organized in groups and these ones in sections.
Precondition	-
Input	-
Output	Table displaying the values of the variables of the initial state.
Error	-

Trace Visualizer

1. Visualize the trace of a counterexample

Id	1.1
Use Case	Load a trace
Description	Visualize the variables of the initial state and their values. The variables will be organized in groups and these ones in preset sections.
Precondition	The trace of the counterexample must follow the trace template.
Input	A trace of a counterexample generated by a model checker.
Output	Table displaying the values of the variables organized in the predefined groups and these ones in the predefined sections.
Error	The trace does not follow the template

Id	1.2
Use Case	Visualize the variables of the states
Description	Visualize the variables of every state of the trace and their values. The variables will be organized in groups and these ones in sections.
Precondition	A correct trace must have been loaded.
Input	The values of the variables of the corresponding state to visualize.
Output	Table displaying the values of the variables of the corresponding state.
Error	There are no states in the trace.

Id	1.3
Use Case	Highlight changes in the values.
Description	Highlight the values of the variables that have changed from the last state.
Precondition	The next state of the trace is visualized.
Input	Next state values in the trace.
Output	Values that have changed in the new state highlighted.
Error	-

2. Navigate among the states

Id	2.1
Use Case	Go one state forward
Description	Visualize the values of the variables of the next state. The variables will be organized in groups and these ones in sections.
Precondition	There must be steps forward left in the state.
Input	Next state values in the trace.
Output	Table displaying the values of the variables of the next state.
Error	There are no more states in the trace.

Id	2.2
Use Case	Go an interval step forward
Description	Visualize the values of the variables of the next state according to the interval value. The variables will be organized in groups and these ones in sections.
Precondition	There must be steps forward left in the state.
Input	Next interval state values in the trace.
Output	Table displaying the values of the variables of the next interval state.
Error	There are less states than the interval in the trace.

Id	2.3
Use Case	Go one state backward
Description	Visualize the values of the variables of the state before. The variables will be organized in groups and these ones in sections.
Precondition	It is not the initial state.
Input	Last state values in the trace
Output	Table displaying the values of the variables of the last state.
Error	It is the initial state.

Id	2.4
Use Case	Go an interval step backward
Description	Visualize the values of the variables of the state before according to the interval value. The variables will be organized in groups and these ones in sections.
Precondition	There must be more or same states than the interval between the current state and the initial state.
Input	Before interval state values in the trace.
Output	Table displaying the values of the variables of the before interval state .
Error	There are less states than the interval between the current state and the initial state in the trace.

Id	2.5
Use Case	Go to the initial state
Description	Visualize the values of the variables of the first state. The variables will be organized in groups and these ones in sections.
Precondition	-
Input	Initial state values in the trace.
Output	Table displaying the values of the variables of the initial state.
Error	-

Id	2.6
Use Case	Go to the last state of the trace
Description	Visualize the values of the variables of the last state in the trace. The variables will be organized in groups and these ones in sections.
Precondition	-
Input	Last state values in the trace.
Output	Table displaying the values of the variables of the last state of the trace.
Error	-

Id	2.7
Use Case	Go to a selected state of the trace.
Description	Visualize the values of the variables of the selected state in the trace. The variables will be organized in groups and these ones in sections.
Precondition	The user selects a state to visualize
Input	The state to visualize and its values in the trace.
Output	Table displaying the values of the variables of the selected state.
Error	The selected state does not appear in the trace.

Id	2.8
Use Case	Play steps forward automatically
Description	Visualize the values of the variables of the states automatically stepping forward every second. The variables will be organized in groups and these ones in sections.
Precondition	-
Input	The counterexample trace.
Output	Table displaying the values of the variables of the each state that is visited.
Error	-

3. Trace visualizer settings

Id	3.1
Use Case	Set the size of the interval step.
Description	Set how big the interval step is.
Precondition	-
Input	Size of the step.
Output	-
Error	The step is bigger than the length of the trace.

Id	3.2
Use Case	Save configuration.
Description	Save the settings such as the organization of the variables, groups and sections in the XML trace template..
Precondition	-
Input	-
Output	Modified trace template.
Error	-

General settings

1. Organize the groups in sections

Id	1.1
Use Case	List sections and their corresponding groups.
Description	See the groups contained in each section.
Precondition	-
Input	Groups and sections.
Output	List of groups in each section.
Error	-

Id	1.2
Use Case	Create a new section.
Description	Allow the user to create a new section.
Precondition	-
Input	Name of the section.
Output	A new empty section with the input name.
Error	-

Id	1.3
Use Case	Change section name.
Description	Allow the user to change the name of the section.
Precondition	There must be sections in the list.
Input	A section and a new name for it.
Output	The selected section renamed.
Error	-

Id	1.4
Use Case	Delete a section.
Description	Allow the user to delete a section.
Precondition	There must be sections in the list.
Input	A section to be deleted.
Output	The deleted section will disappear from the sections list. If the section has groups, these ones will be without section when the section is deleted.
Error	-

Id	1.5
Use Case	Change a group to a different section.
Description	Change a group from a section to another one.
Precondition	-
Input	A group to be changed to another section and the destination section.
Output	The group is deleted from its section and added to the destination one.
Error	-

Id	1.6
Use Case	Change visibility of groups.
Description	A group is chosen not to appear in the results tables and viceversa.
Precondition	The group has to be in a section to make it invisible and has to be invisible to make it visible.
Input	A group.
Output	Appear or not to appear in the result table.
Error	-

2. Organize the variables in groups

Id	2.1
Use Case	List groups and their corresponding variables.
Description	See the variables contained in each group.
Precondition	-
Input	Variables and groups.
Output	List of variables in each group.
Error	-

Id	2.2
Use Case	Create a new group.
Description	Allow the user to create a new group.
Precondition	-
Input	Name of the group and section to be assigned to.
Output	A new empty group with the input name.
Error	-

Id	2.3
Use Case	Change group name.
Description	Allow the user to change the name of the group.
Precondition	There must be groups in the list.
Input	A group and a new name for it.
Output	The selected group renamed.
Error	-

Id	2.4
Use Case	Delete a group.
Description	Allow the user to delete a group.
Precondition	There must be group in the list.
Input	A group to be deleted.
Output	The deleted group will disappear from the groups list. If the group has variables, these ones will be without group when the group is deleted.
Error	-

Id	2.5
Use Case	Change a variable to a different group.
Description	Change a variable from a group to another one.
Precondition	-
Input	A variable to be changed to another group and the destination group.
Output	The variable is deleted from its group and added to the destination one.
Error	-

Id	2.6
Use Case	Change visibility of the variables.
Description	A variable is chosen not to appear in the results tables and viceversa.
Precondition	The variable has to be in a group to make it invisible and has to be invisible to make it visible.
Input	A variable.
Output	Appear or not to appear in the result table.
Error	-

3.5.2 Model Generator

Id	1
Use Case	Generate an EVMDD model of the Clock synchronization Protocol.
Description	Generate automatically a EVMDD model of the Clock synchronization Protocol according to the input parameters .
Precondition	The protocol is modeled correctly.
Input	Necessary parameters to custom the model in order to follow the real system as accurately as possible (See Section sec:modelgen).
Output	The model to be checked (see Appendix **??**).
Error	The parameters are not correct.

3.5.3 Result Analyzer

Id	1
Use Case	Generate a PDF report with Clock Synchronization Protocol results.
Description	Creation of a PDF report with the results of model checking in EVMDD-smc the Clock Synchronization Protocol
Precondition	EVMDD-smc has checked the precission convergence properties producing an output file (see Section 5.3.3).
Input	An output file that informs the number of states accomplishing each precission.
Output	PDF report with the information of the chosen topology: related parameters, precission, expected precission, size of state space and time to create state space.
Error	-

3.6 Non-functional requirements

3.6.1 Interface requirements

Both, simulator and trace visualizer, must share the same interface. The GUI must allow easy navigation among the states and, in case of the simulator, to choose clearly the next state to fire.

The Model Generator and the Result Analyzer will be run in the console. They should allow the user to input the parameters following the input data standards of the console.

3.6.2 Performance

The memory usage must be really taken into account, because the quantity of the states in a trace generated by a model checker can be huge.

3.6.3 Maintainability

It must be able to be improved in the future, since it works together with other tools in development.

3.6.4 Portability

The AMC Visualization tool must run in Linux, Mac OS and Windows. The Model Generator and the Trace Visualizer are not necessary to run in Windows, since the EVMDD-smc cannot run in it.

Chapter 4

AMC Visualization tools

4.1 Introduction

As presented in Chapter 1, the AMC Visualization tool is divided into two modules: trace visualizer and the EVMDD simulator. These functionalities share the GUI and part of the object data structure, because both explore states, what makes possible to reuse code.

In order to fit them in common data structures and GUI, it is necessary to have a clearly layered architecture that allows to reuse blocks correctly and to communicate them in an effective and efficient way, saving as much memory as possible, due to the vast quantity of states that can be analyzed. Moreover, the specific blocks of each functionality must be adapted correctly to the common structure.

In this section, the common architecture and specific design points are shown.

4.2 Analysis

The AMC Visualization tool needs a different input depending on the operation that it is going to perform, but its main point is finally displaying the values of the variables organized in sections and groups for each state.

4.2.1 Trace visualizer

In Figure 4.1, it is possible to see two inputs that correspond to the trace visualizer module:

- **.xml file**: It is the structured file that provides the trace output of a model checker to AMC.
- **.out file**: It is the trace output from the EVMDD model checker. As the trace visualizer reads from .xml file, it converts this file to XML in order to be able to work with it.

The tool interprets .xml files in order to display the values of the variables for each state in order.

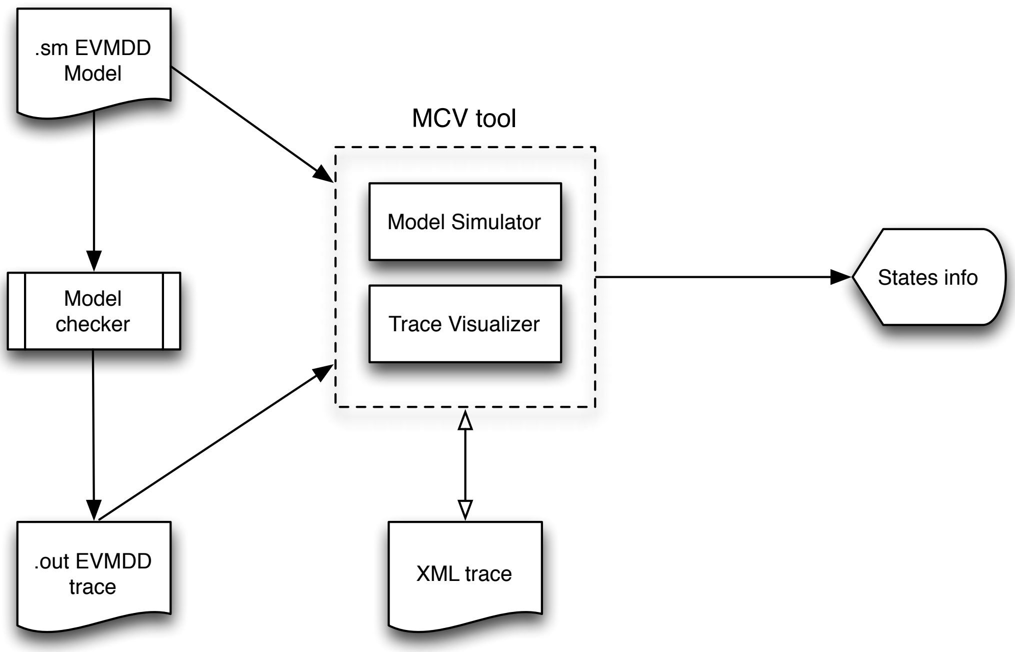


Figure 4.1: Input-Output diagram

4.2.2 EVMDD simulator

The simulator accepts as output EVMDD models, which are .sm files. These models are parsed into objects, so as to let the user simulate the states of the model checker.

4.3 Design

4.3.1 Architecture

The AMC Visualization tool architecture makes a clear separation in three layers, as it is possible to see in Figure 4.2:

1. **Front end**: the GUI shared by the trace visualizer and the model simulator.
2. **Business Layer**: This is the core of the tool. It makes possible the trace visualizer and the simulator share common parts of the data structure and to identify the specific components. The following internal blocks can be identified:
    - Common Manager: contains the shared data structures and methods and communicates with the front end.
    - Trace Manager: interprets the XML and converts it into objects.
    - Model Manager: contains the specific data structures and the methods required by the Parser to convert EVMDD model into objects.

- Beans: the necessary objects in which the information, either from the XML or from the model, is converted.

3. **Persistence Layer**: It is independent for each module:

   - Trace visualizer: XML files that keep track of the organization of the trace information: variables and its values, groups and sections.
   - EVMDD simulator: .sm model files. They are read by the Parser, that is the link to the business layer.
   - EVMDD trace interpreter: External module that interprets the EVMDD trace output and converts it to XML.

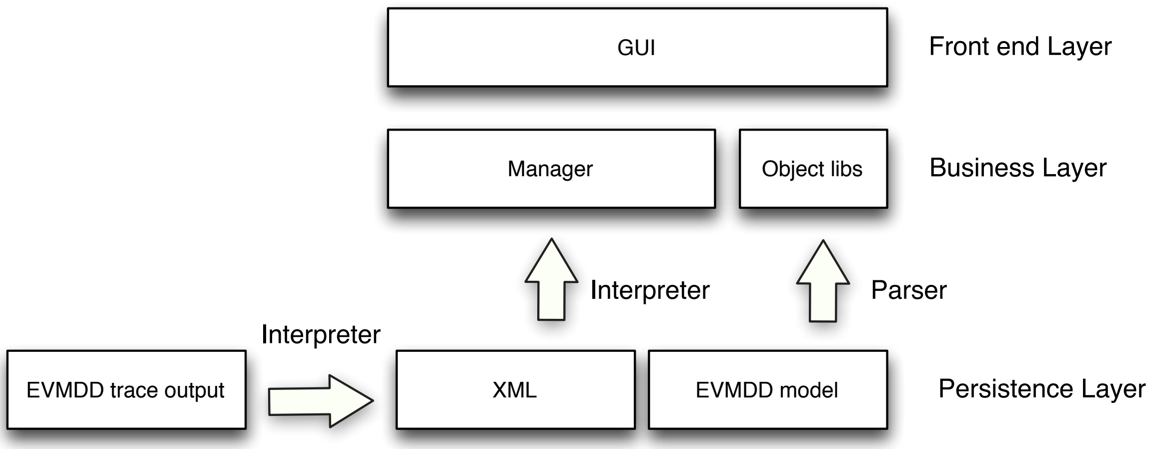


Figure 4.2: Architecture diagram

4.3.2 GUI

The GUI of the AMC Visualization tool is adapted to fit the trace visualizer and the EVMDD simulator in the same environment. It is built to allow easy navigation among the states and it clearly shows each state information in a table, accomplishing the requirements listed in the requirements specification in Section 3.5.

The main goal is to enable the analisis of the information of every state as best as possible. In order to do that, two states are displayed every time to let the user compare between the current state and the previous one. Moreover, the user has the possibility of changing the view of the data tables, either aligned vertically or horizontally, what is helpful when there is a big number of variables to display. In Figures 4.3 and 4.4 it is possible to see both types of alignments of the main window of the visualization tool. Besides, using the control buttons placed on the bottom (or on the right for vertical table alignment), the application accomplishes the navigation requirements.

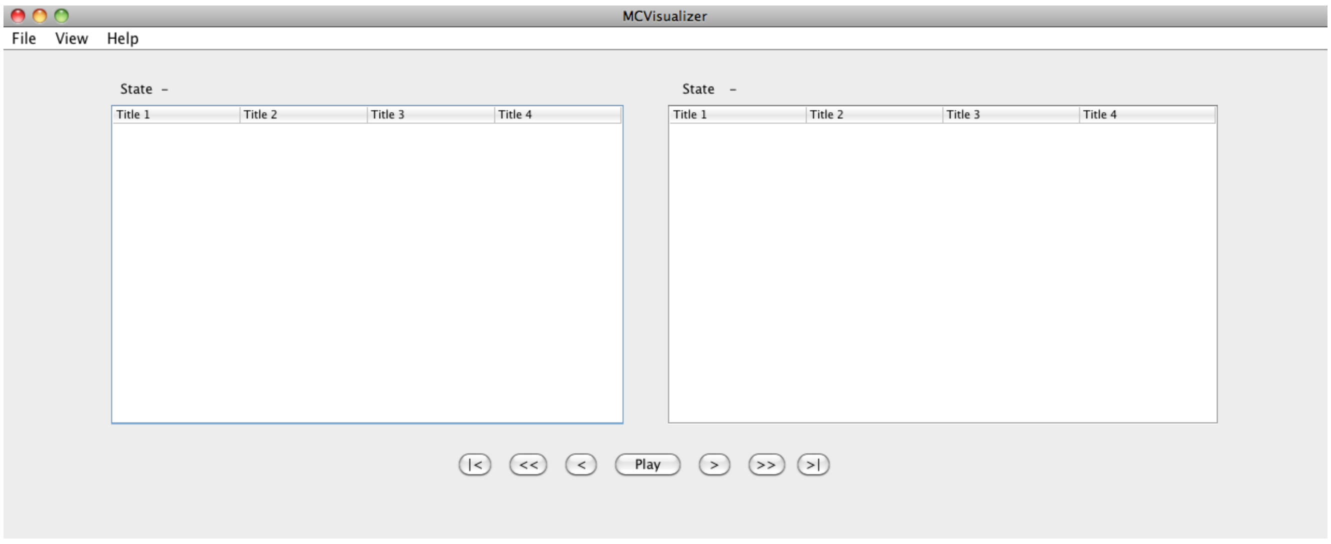

MCVisualizer
File View Help
State -
Title 1 Title 2 Title 3 Title 4
State -
Title 1 Title 2 Title 3 Title 4
|< << < Play > >> >|

Figure 4.3: Main GUI. Horizontal layout

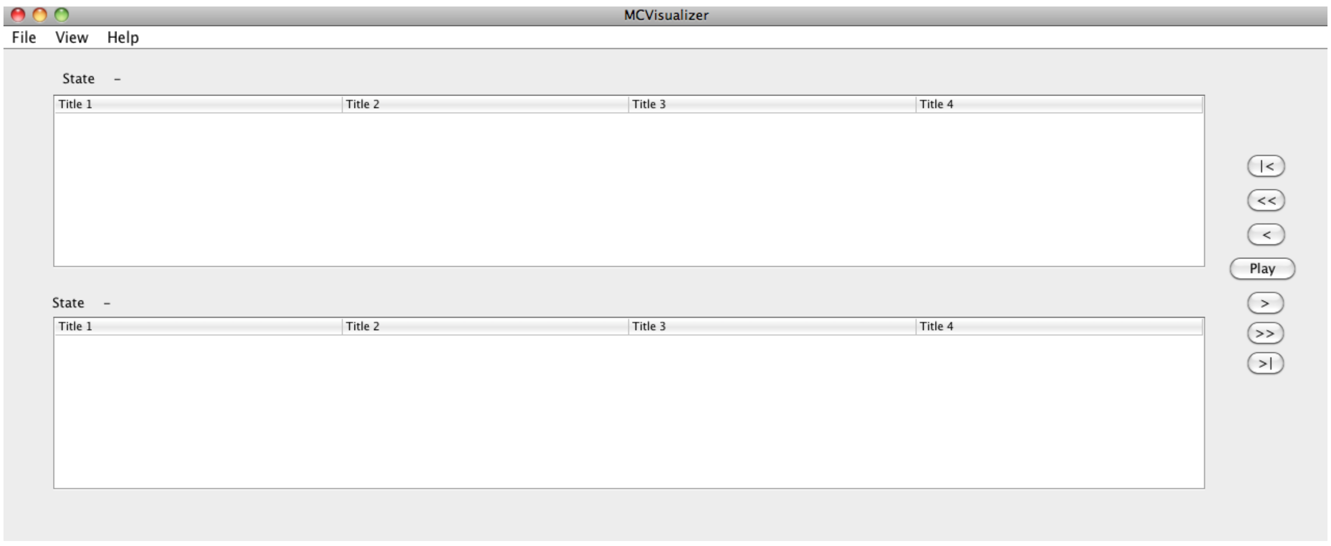
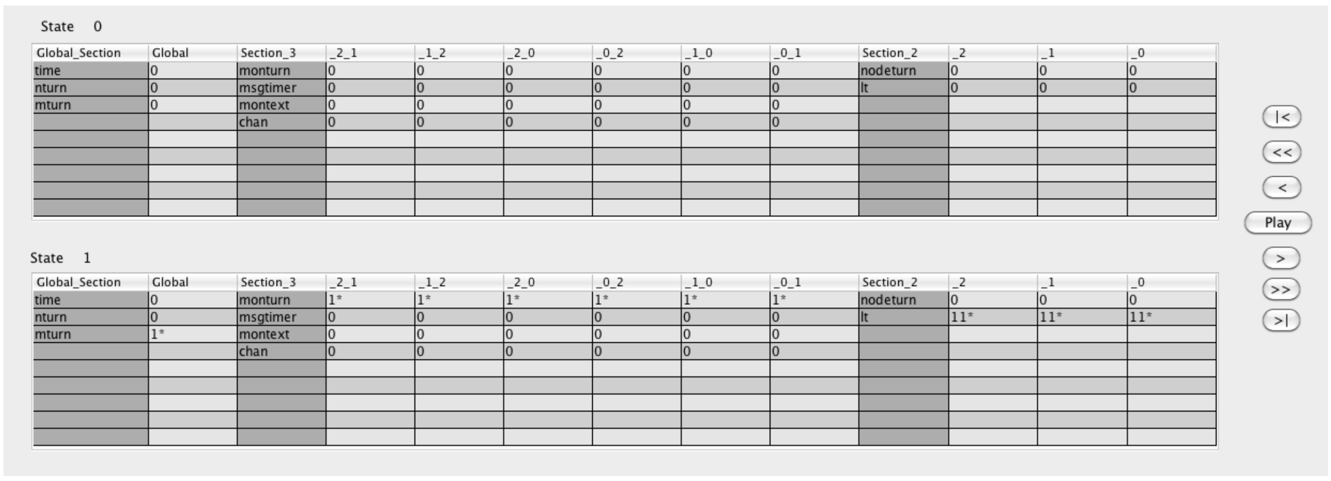

MCVisualizer
File View Help
State -
Title 1 Title 2 Title 3 Title 4
State -
Title 1 Title 2 Title 3 Title 4
|< << < Play > >> >|

Figure 4.4: Main GUI. Vertical layout


State 0

Global_Section	Global	Section_3	_2_1	_1_2	_2_0	_0_2	_1_0	_0_1	Section_2	_2	_1	_0
time	0	monturn	0	0	0	0	0	0	nodeturn	0	0	0
nturn	0	msgtimer	0	0	0	0	0	0	lt	0	0	0
mturn	0	montext	0	0	0	0	0	0				
		chan	0	0	0	0	0	0				

State 1

Global_Section	Global	Section_3	_2_1	_1_2	_2_0	_0_2	_1_0	_0_1	Section_2	_2	_1	_0
time	0	monturn	1*	1*	1*	1*	1*	1*	nodeturn	0	0	0
nturn	0	msgtimer	0	0	0	0	0	0	lt	11*	11*	11*
mturn	1*	montext	0	0	0	0	0	0				
		chan	0	0	0	0	0	0				

Figure 4.5: GUI Trace Visualization

On the other hand, the data table that displays the information of every state is organized as shown in Figure 4.5 in order to accomplish the visualization requirements.

- Every section corresponds to a subtable inside.
  - Each column corresponds to a group.
  - Each row corresponds to a variable that is common for every group of the section.

Moreover, other GUI decisions have been followed to make this information more readable

- The columns with the names of the variables are colored differently than the ones that show the values. Besides, odd and even rows have different colors, to ease the reading.
- The values that have changed with respect to the state before are marked with a star (see Section 3.5 - Req 1.3).
- The navigation controls allow to follow the states easily (see Section 3.5 - Req 2.1-2.8).
- In the settings option, it is possible to set up the organization of the variables and groups between lists (Figure 4.6 and Section 3.5 - Req. General Settings).

For the EVMDD simulator, when the user wants to simulate one step, the transitions enabled to be fired are shown in a dialog. There, the user is able to fire the one he wants to (see Figure 4.7).

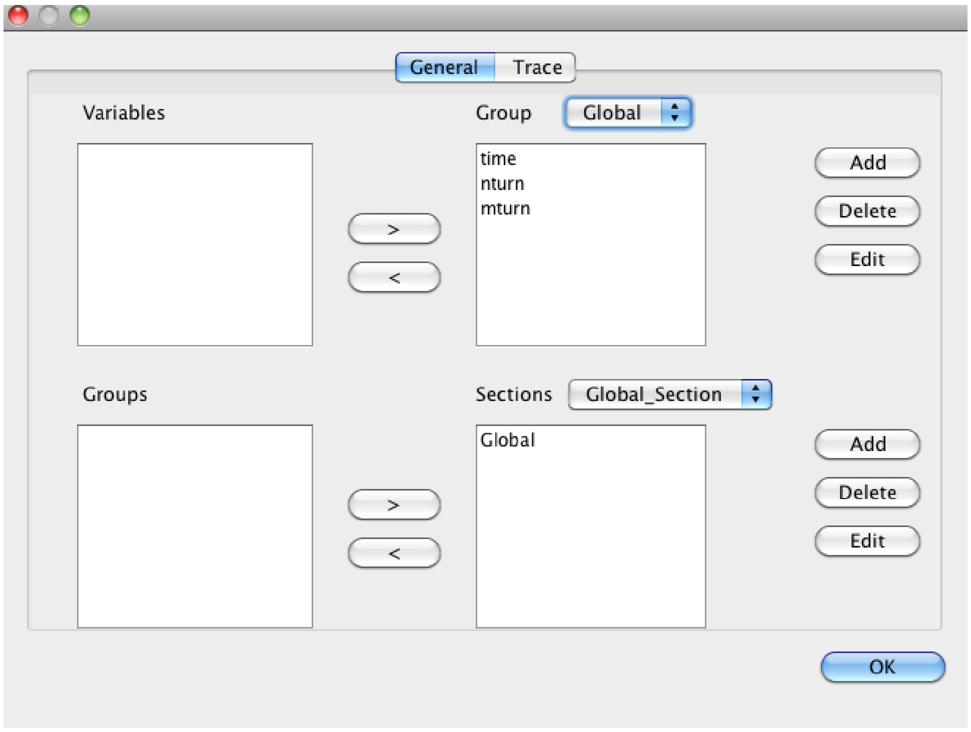
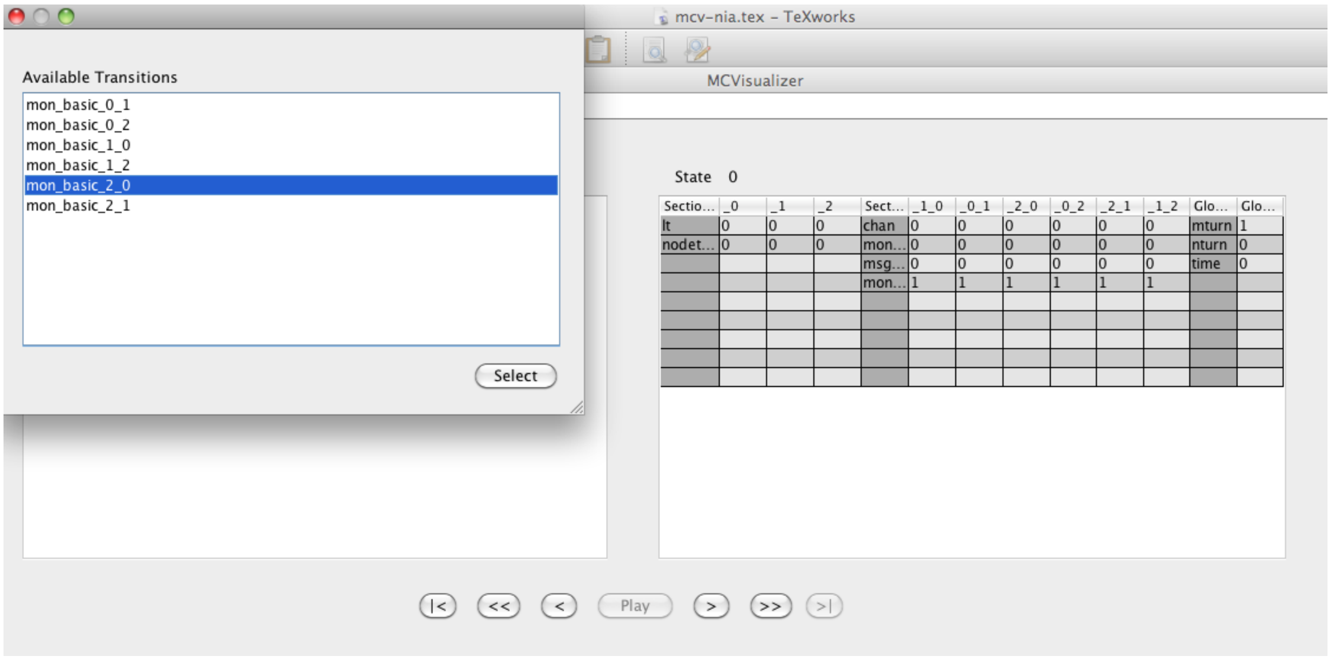


Figure 4.6: GUI Settings



mon_basic_0_1
mon_basic_0_2
mon_basic_1_0
mon_basic_1_2
mon_basic_2_0
mon_basic_2_1

State 0

Sectio...	_0	_1	_2	Sect...	_1_0	_0_1	_2_0	_0_2	_2_1	_1_2	Glo...	Glo...
lt	0	0	0	chan	0	0	0	0	0	0	mturn	1
nodet...	0	0	0	mon...	0	0	0	0	0	0	nturn	0
				msg...	0	0	0	0	0	0	time	0
				mon...	1	1	1	1	1	1		

Figure 4.7: GUI Available Transitions Dialog

4.4 Implementation

This section goes one step lower in the architecture and explains in detail the implementation decisions and the organization.

4.4.1 Environment and platform

The AMC Visualization tool has been developed in Java, so it is flexible to be run in every operating system with a Java Virtual Machine. The environment was NetBeans, since it makes easier to build GUI, using Java Swing libraries. On the other hand, Java CC [8] was used in the simulator module to parse EVMDD to Java objects.

4.4.2 Organization

In order to implement the architecture of the system, the package organization of the code aims to be equivalent to the blocks defined in the architecture. Besides, it takes advantage of the object oriented programming in Java.

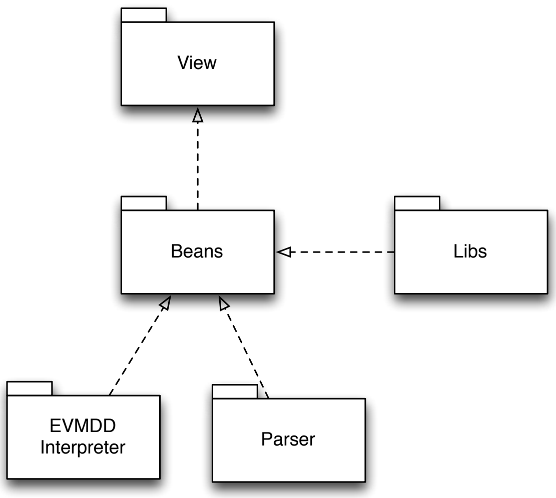


Figure 4.8: Packages organization diagram

In Figure 4.8, which shows the organization of the packages, it is possible to identify the relations with the architecture showed in Figure 4.2:

- **libs** is equivalent to the Objects library block
- **Beans** is equivalent to Manager block
- **View** is the front end presentation block.
- **Parser and EVMDD Interpreter** are the links to the Persistence layer.

Data Structures

The information contained in the trace output and the model is converted into objects, either by the XML interpreter or by the EVMDD model parser.

Figure 4.9 shows the relations between the object library classes to express the information of a state.

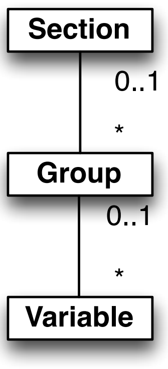


Figure 4.9: Objects library class diagram (General)

EVMDD model beans

In the EVMDD model, it is also necessary to keep track of the transitions that make possible to go from one state to another. In order to fire a transition, its condition must be checked before. Thus, the tool uses a binary tree node structure, that allows to carry out this process. Figure 4.10 looks inside these classes.

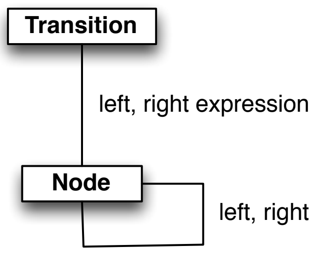


Figure 4.10: Objects library class diagram (Model)

4.4.3 Manager classes

The Manager classes control the logic of the tool. They organize the data structures and the methods separating the especific ones for both modules, placed in the *XMLManager* and *ModelManager* and the common ones, in the Manager class, that is the parent class (see Figure 4.11). This organization permits them to provide the information to the View classes.

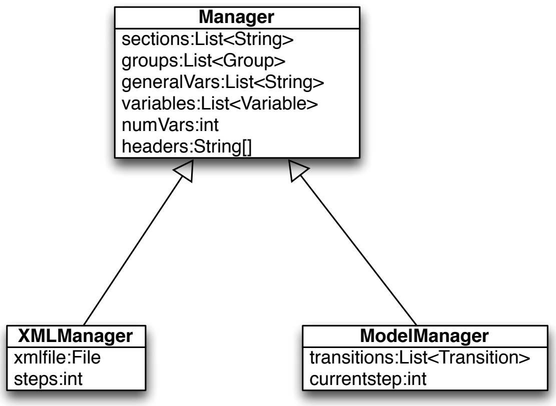


Figure 4.11: Managers class diagram (General)

4.4.4 Parser

In order to turn the EVMDD model into Java Objects, it is necessary to build a parser. This parser analyze lexically the tokens of the model, sintactically identifies that the model structure is correct and fills up the data objects using the methods available in the ModelManager class.

As the code is in Java, it was decided to choose Java CC [8] to build the parser. JavaCC is a parser generator and a lexical analyzer generator. Parsers and lexical analysers are software components for dealing with an input of character sequences. Lexical analysers can break a sequence of characters into a subsequences called tokens and it also classifies the tokens [9]. The JavaCC lexic analyser is LL(n) [10], what means that does not accept left recursion. To select the n tokens that have to be looked at to be analized, it is used LOOKAHEAD(n) method, when there is the same begining in different rules of the same production.

The ModelParser.jj file is the editable one that contains all the analyzers. So, the lexic tokens, the grammar and the used methods are specified in this file. See Appendix C to look at the code.

Having constructed ModelParser.jj, JavaCC is invoked, what generates the following classes:

- **TokenMgrError** is a simple error class; it is used for errors detected by the lexical analyser and is a subclass of Throwable.
- **ParseException** is another error class; it is used for errors detected by the parser and is a subclass of Exception and hence of Throwable.

- **Token** is a class representing tokens. Each Token object has an integer field kind that represents the kind of the token (PLUS, NUMBER, or EOF) and a String field image, which represents the sequence of characters from the input file that the token represents.
- **SimpleCharStream** is an adapter class that delivers characters to the lexical analyser.
- **AdderConstants** is an interface that defines a number of classes used in both the lexical analyser and the parser.
- **ModelTokenManager** is the lexical analyser.
- **Model** is the parser.

4.4.5 View

The View package contains the visual components that encode the GUI. It uses Java.swing libraries [11] that structures the graphical components in containers. Figure 4.12 shows the organization of the graphical components classes.

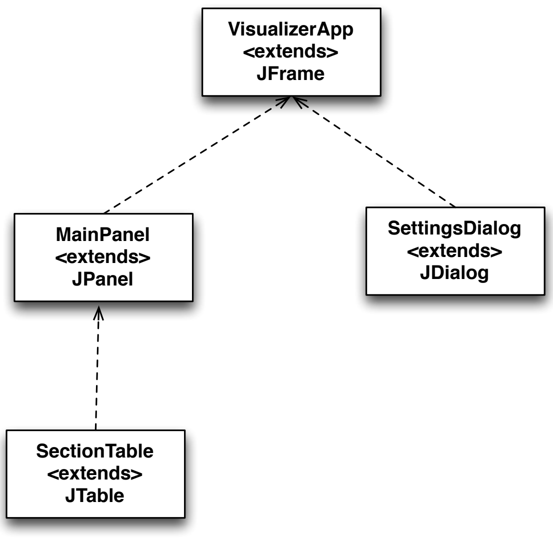

VisualizerApp <extends> JFrame
MainPanel <extends> JPanel
SettingsDialog <extends> JDialog
SectionTable <extends> JTable

Figure 4.12: View components class diagram (General)

Apart from this, a special table called *SectionTable* is created , that extends from *JTable*, to add color when displaying the data.

Chapter 5

Modeling the ”Clock Synchronization Protocol for Arbitrary Digraphs” in EVMDD

5.1 Introduction

In the case study, the Distributed clock Synchronization Protocol for Arbitrary Digraphs is model checked using the EVMDD model checker. In this process, the generation of the model was automated using a Model Generator. Moreover, the model was validated and verified using the trace visualizer and the simulator and, finally, the results were presented in a report using the output analyzer. This section explains in detail the process, paying special attention to the model, the model generator and the result analyzer.

5.2 The protocol

5.2.1 Description

In this section, we provide a description of the protocol [3] and give an intuitive depiction of its behavior. The system has two synchronization states: synchronized and unsynchronized. It is in the unsynchronized state when it starts up and in the synchronized state when the nodes are within an expected bounded precision. The system transitions go from the unsynchronized state to the synchronized state after the execution of a synchronization protocol. Therefore, the clock synchronization protocol is expected to enable the system to transition to the synchronized state and remain there.

When a system reaches and operates in the synchronized state, it is said to be synchronous or in synchrony. Due to the inherent drift in the local times, a synchronization protocol must be re-executed at regular intervals to ensure that the local times are kept synchronized. The rate of resynchronization is constrained by physical parameters of the design (e.g., oscillator drift rates) as well as precision and accuracy goals. Thus, the Clock Synchronization Protocol for Arbitrary Digraphs addresses achieving and maintaining the precision goal of the system. The protocol enables the system to achieve and maintain synchrony among distributed local logical clocks, i.e., LocalTimers (not local physical oscillators).

The clocks need to be periodically synchronized due to their inherent drift with respect to each other. In order to achieve synchronization, the nodes communicate by exchanging Sync messages. So, the periodic synchronization after achieving the initial synchrony is referred to as the resynchronization process whereby all nodes reengage in the synchronization process. A node is said to time-out when its LocalTimer reaches its maximum value and the resynchronization process begins when the first node (fastest node) times-out. At that moment, it transmits a Sync message and ends after the last node (slowest node) transmits a Sync message. A node is said to be interrupted when it accepts an incoming Sync message before its LocalTimer reaches its maximum value, i.e., before it times-out.

A node consists of a synchronizer and a set of monitors. A Sync message is transmitted either as a result of a resynchronization timeout, or when a node receives Sync message(s) indicative of other nodes engaging in the resynchronization process. The messages to be delivered to the destination nodes are deposited on communication channels.

All protocol parameters have discrete values with the time-based terms having units of real time clock ticks. The discretization is for practical purposes in implementing and model checking of the protocol. Although, the network level measurements are real values, locally and at the node level, all parameters are discrete.

The **resynchronization period**, denoted **P**, has units of real time clock ticks and is defined as the upper bound on the time interval between any two consecutive resets of the LocalTimer by a node.

Drift per t, denoted δ**(t)**, has units of real time clock ticks and is defined as the maximum amount of drift between any two nodes for the duration of t, δ(t) $>$ 0.

The **graph threshold**, T_s, is based on a specified graph topology and has units of real time clock ticks. The guaranteed precision or simply precision of the network, denoted π, $0 \leq \pi <$ P, has units of real time clock ticks and is defined as the guaranteed achievable precision among all nodes.

The **convergence time**, denoted **C**, has units of real time clock ticks and is defined as the bound on the maximum time it takes for the network to converge, i.e., to achieve synchrony. Precision between LocalTimers of any two adjacent nodes N_i and N_j at time t has units of real time clock ticks.

The **initial synchrony** is a state of the network and the earliest time when the precision among all nodes, upon convergence, is within π. The initial synchrony occurs at time C_{Init}.

The **initial precision** among LocalTimers of all nodes at time t has units of real time clock ticks and is defined as a measure of the precision of the network after elapse time of C_{Init}.

The **initial guaranteed precision** among LocalTimers of all nodes at time t is denoted by **InitGuaranteed(t)**, has units of real time clock ticks and is a measure of the precision of the network after elapse time of C.

Delay D, is the time that takes to the Sync message to arrive to another node (see Figure 5.1).

Network imprecission d, is the maximum time difference among all receivers of a message from a transmitting node with respect to real time (see Figure 5.1).

The **communication latency**, denoted γ, is expressed in terms of D and d, and is constrained by γ = (D+d) and so has units of real time clock ticks.

The maximum number of **faulty nodes** is denoted as **F**.

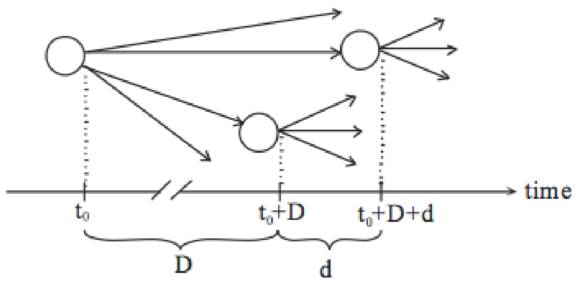


Figure 5.1: Delay and network imprecission explanation

5.2.2 How Does The Protocol Work?

A node periodically undergoes a resynchronization process either when its LocalTimer times out or when it receives a Sync message. If it times out, it broadcasts a Sync message and so initiates a new round of the resynchronization process. However, since we are assuming that there are no faults present, i.e., F = 0, when a node receives a Sync message, except during a predefined window, it accepts the Sync message and undergoes the resynchronization process where it resets its LocalTimer and relays the Sync message to others. This process continues until all nodes participate in the resynchronization process and converge to a guaranteed precision. The predefined window where the node ignores all incoming Sync messages, referred to as ignore window, provides a means for the protocol to stop the vicious cycle of resynchronization processes triggered by the follow up Sync messages.

5.2.3 Assumptions

The protocol is based on the following assumptions:

- All nodes correctly execute the protocol.
- All channels correctly transmit data from their sources to their destinations.
- K $\geq$ 1.
- T = strongly connected digraph.
- A message sent by a node will be received and processed by all other nodes within, where γ = (D + d).

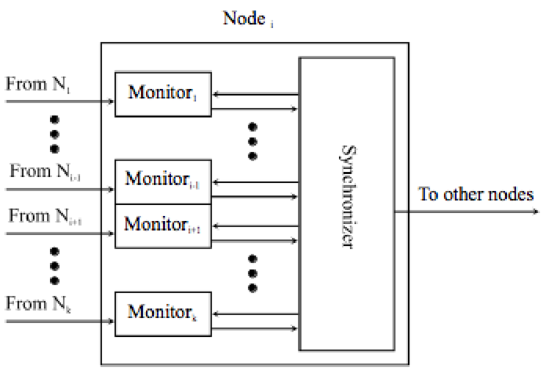
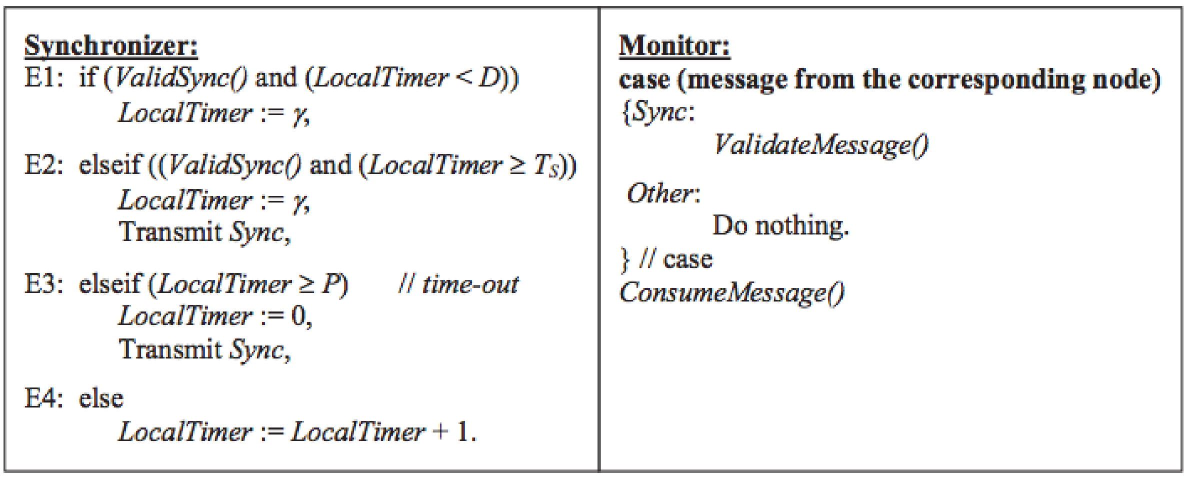


Figure 5.2: Detailed view of a node

- $0 \leq \rho < 1$.
- Absence of faults in the links and nodes, i.e., F = 0.
- The initial values of the variables of a node are within their corresponding data-type range, although possibly with arbitrary values.

5.2.4 The protocol

Figure 5.3 shows the protocol itself. It separates synchronizer and monitor actions.



```
Synchronizer:
E1: if (ValidSync() and (LocalTimer < D))
        LocalTimer := γ,
E2: elseif ((ValidSync() and (LocalTimer ≥ T_S))
        LocalTimer := γ,
        Transmit Sync,
E3: elseif (LocalTimer ≥ P)        // time-out
        LocalTimer := 0,
        Transmit Sync,
E4: else
        LocalTimer := LocalTimer + 1.
```

```
Monitor:
case (message from the corresponding node)
{Sync:
        ValidateMessage()
 Other:
        Do nothing.
} // case
ConsumeMessage()
```

Figure 5.3: The self-stabilizing clock synchronization protocol for arbitrary digraphs.

In the synchronizer actions:

E1: Any message in the node and LT less than the Delay? Reset LT to gamma, what means that the message is discarded.

E2: Any message in the node and LT greater or equal than the ignore window? Reset LT to gamma and retransmit the message.

E3: LT reaches the resynchronization period?(time-out) Reset LT and transmit a synchronization message.

E4: Rest of the cases? Increase LT.

The monitor only validates and consumes a message when this one has been received.

5.3 Modeling the protocol

There are two general formal methods approaches for the verification of the correctness of a protocol; theorem proving and model checking. Proof via theorem proving requires a deductive proof of the protocol, whereas proof via model checking is based on specific scenarios and generally limited to a subset of the problem space. The election of model checking approach was because of its ease, feasibility, and quick examination of a subset of the problem space [2]. As it was explained in Section 2.2, the selected model checker to verify the system was EVMDD-smc. The model aims to represent the protocol as accurately as possible according to the limitations of:

- State explosion (memory consumption)
- EVMDD representation

Moreover, some assumptions are taken into account in order to model the system:

- Discrete variables: it is not feasible that the model checker accepts continues variables. Therefore, the clock ticks are discretized.
- Asynchronous system.
- For every tick, components are executed in order: Monitors - Nodes

Since EVMDD-smc cannot model synchronous systems, the protocol is adapted to be accepted by the model checker. So as to do it, the transition order is managed by turns, where there is a turn for each component.

Firstly, it was decided to model the system without taking into account if the monitor actions were taken first than the node actions. However, the results showed that it was not possible to achieve synchronization in that way and global turns were introduced to manage the preference between monitors and nodes. Consequently, monitor transitions are executed before node ones in every tick. Anyway, it is still possible to check the results produced by not giving preference to Monitor actions by selecting this option in the model generator (see Section 5.4).

In the following subsections, the parts of the model are explained using an example of the EVMDD model for the protocol with a two node, complete topology (all nodes connected between them), delay=2 and d=1 (network inprecisson).

5.3.1 Variables

The variables that the model contains are:

- A local timer per node. Range: [0. . .P-1] It can be checked one or several values of the range for the initial state.

  ```
  lt_0             [0,23]
  lt_1             [0,23]
  ```

- A channel for every connection. It shows if a message is being sent. Range: [0. . .1]

  ```
  chan_1_0        [0,1]
  chan_1_0        [0,1]
  chan_1_0        [0,1]
  ```

- A counter for the received messages in the monitor (monitor_text). It is necessary one for every connection. Range: [0. . .1]

  ```
  montext_0_1     [0,1]
  montext_1_0     [0,1]
  ```

- A message timer [0. . .γ +1] to count the Delay, network imprecission and the tick that is kept in the monitor.

  ```
  msgtimer_0_1    [0,4]
  msgtimer_1_0    [0,4]
  ```

- Drift modeled by turns [0. . .1]. There is a turn for each component, two to manage the preferences (one for the nodes and one for the monitors) and another one for the global tick. Transitions are fired if the corresponding turn is active. In order to be active, all the turns must be inactive. Consequently, all the turns are turned on in the next step. That makes an interleaving of as many states as components are for every global tick plus one for the preference between monitors and node transitions .

  ```
  mturn           [0,1]
  nturn           [0,1]
  nodeturn_1      [0,1]
  monturn_1_0     [0,1]
  monturn_0_1     [0,1]
  ```

- Global timer. [0. . .Convergence time]

  ```
  time            [0,54]
  ```

5.3.2 Transitions

Here it is the explanation of the transitions that model the system. In the Figure 5.4, it is possible to see how it works graphically.

1. Global transitions

   (a) global_time_inc: increments the global clock in one tick when all the actions have been done.

   ```
   time<54  /\ mturn=0 /\ nturn=0 -> time'=time+1  /\ mturn'=1
   /\ nturn'=0 /\ nodeturn_0'=0 /\ monturn_0_1'=1
   /\ nodeturn_1'=0 /\ monturn_1_0'=1
   ```

   (b) global_last_turn: represents the moment when the global clock has reached the convergence time and has no actions to do. Consequently, there is no tick increment.

   ```
   time=54 /\ mturn=0 /\ nturn=0 ->  mturn'=1 /\ nturn'=0
   /\ nodeturn_0'=0 /\ monturn_0_1'=1
   /\ nodeturn_1'=0 /\ monturn_1_0'=1
   ```

   (c) turn_to_nodes: when every action related with the monitors is done, the turn goes to nodes transitions.

   ```
   time<=54 /\ mturn=1 /\ nturn=0
   /\ nodeturn_0=0 /\ monturn_0_1=0
   /\ nodeturn_1=0 /\ monturn_1_0=0 ->
   mturn'=0 /\ nturn'=1
   /\ nodeturn_0'=1 /\ monturn_0_1'=0
   /\ nodeturn_1'=1
   ```

   (d) end_node_turn: when all nodes actions are executed, it goes to the state that enables global tick increment.

   ```
   time<=54 /\ mturn=0 /\ nturn=1
   /\ nodeturn_0=0 /\ monturn_0_1=0
   /\ nodeturn_1=0 /\ monturn_1_0=0 ->
   mturn'=0 /\ nturn'=0 /\ nodeturn_0'=0
   /\ monturn_0_1'=0 /\ nodeturn_1'=0
   ```

2. Monitor transitions: The actions done by the monitor where x is the receiver node id and y is the sender node id.

   (a) mon_basic_x_y: trivial case. For each monitor, if there is no message to manage, consume the turn.

   ```
   monturn_0_1=1 /\ chan_1_0=0 /\ montext_0_1=0 ->
   monturn_0_1'=0
   ```

   (b) mon_x_y_msg_delay: For the Delay modeling, the message resides in the channel for D-1 ticks until it may processed. For the network imprecision, the message is received sometime in the interval [D,D+d] ticks. The model checker will decide a value in this range. Non-determinism in the left side condition between the related transitions.

   ```
   monturn_0_1=1 /\ chan_1_0=1 /\ msgtimer_0_1<2 ->
   monturn_0_1'=0  /\ msgtimer_0_1'=msgtimer_0_1+1
   ```

(c) mon_x_y_msg_receive: Process the message of the channel and put it in the monitor after D-1 ticks.

```
monturn_0_1=1 /\ chan_1_0=1 /\ msgtimer_0_1>=1 ->
monturn_0_1'=0  /\ chan_1_0'=0 /\ montext_0_1'=1
```

3. Node transitions (Synchronizer): The transitions correspond to the protocol actions.

(a) trans_e1: Is there any message in the node (method ValidSync(); see 5.2.4) and timer is under the communication latency(γ)? Set the local timer to γ, remove the message(put mon_text=0) and reset the corresponding msg_timer (consume message).

```
nodeturn_0=1 /\ lt_0<1 /\(montext_0_1>0) ->
nodeturn_0'=0 /\ lt_0'=3 /\ montext_0_1'=0
/\ msgtimer_0_1'=(1-montext_0_1)*msgtimer_0_1
```

(b) trans_e2: Is there any message in the node (ValidSync()) and local timer is bigger than the ignore window (T_s)? Set the local timer to γ, consume the message and put it in the channel that goes to the connections (retransmition).

```
nodeturn_0=1 /\ lt_0>=11 /\ (montext_0_1>0) ->
nodeturn_0'=0 /\ montext_0_1'=0 /\ chan_0_1'=1 /\ lt_0'=3
/\ msgtimer_0_1'=(1-montext_0_1)*msgtimer_0_1
```

(c) trans_e3: Are there no messages in the monitor and local timer is bigger or equal than the synchronization period (P), i.e time out? Reset local timer to 0 and transmit a message to the output channels.

```
nodeturn_0=1 /\ montext_0_1=0 /\ lt_0>=23 ->
nodeturn_0'=0 /\ lt_0'=0 /\ chan_0_1'=1
```

(d) trans_e4_1: Is there any message in the node (ValidSync()), but the local timer is under the ignore window? Consume the message and increment the global clock

```
nodeturn_0=1 /\ lt_0>=1 /\ lt_0<11 /\ (montext_0_1>0) ->
nodeturn_0'=0 /\ montext_0_1'=0 /\ lt_0'=lt_0+1
/\ msgtimer_0_1'=(1-montext_0_1)*msgtimer_0_1
```

(e) trans_e4_2: Are there no messages and the local timer is less than P? Increment the global clock

```
nodeturn_0=1 /\ lt_0<23 /\ montext_0_1=0 ->
nodeturn_0'=0 /\ lt_0'=lt_0+1
```

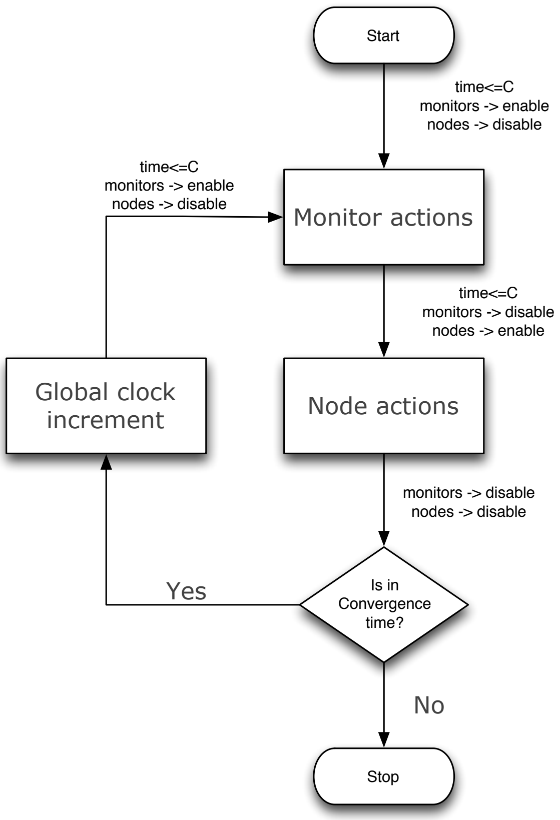


Figure 5.4: Diagram of the transition flow

5.3.3 Properties

Three properties must be checked to see the correctness of the protocol:

Liveness: AF(ElapsedTime) : *time* $\geq$ *Convergence time*

```
  ((time>=54) /\ !((((lt_0-lt_1 >=-2) /\ (lt_0-lt_1 <=2))
\/ ((lt_0-lt_1 <=-22) \/ (lt_0-lt_1 >=22))))) /\ mturn=0
/\ nturn=0
```

Convergence and closure:

AF(time $\geq$ ConvergeTime) /\AG(time $\geq$ ConvergeTime –>AllWithinPrecision) /\AG((time $\geq$ ConvergeTime –>AllWithinPrecision) –>EX(!AllWithinPrecision))

This property must be false, because it is necessary to check that for all the states when time $\geq$ ConvergenceTime, they must be AllWithinPrecission. Then, the property will be verified if the model checker does not find a counterexample of the following:

EF(time $\geq$ ConvergeTime /\!AllWithinPrecision)

To express AllWithinPrecision, it is necessary to measure the difference between the maximum and minimum values of the LocalTimers of all the nodes for the current tick, together with the result with the previous (W+1)γ ticks. However, this is not an efficient way to express it, since it is necessary to keep track of the (W+1)γ previous ticks. Therefore, we check for every node if it is less or equal than the given precision (π) or bigger than the maximum bound (wrap around).

$(\bigwedge |lt\_i - lt\_j| \leq \pi)\ [\ \bigvee |lt\_i - lt\_j| \geq P\text{-}\pi]$

For a two nodes topology in EVMDD would be:

$((time \geq C) \bigwedge !((((lt\_0 - lt\_1 \geq -\pi) \bigwedge (lt\_0 - lt\_1 \leq \pi)) \bigvee ((lt\_0 - lt\_1 \leq -(P-\pi))\ \ (lt\_0 - lt\_1 >= (P-\pi))))))$

Congruence: All the nodes must have the same behavior:

AF(time $\geq$ ConvergeTime) /\AG(!((time $\geq$ ConvergeTime) /\(lt_i =γ)) \/ ((time $\geq$ C) /\(\/ |lt_i lt_j|$\leq\pi$) [/\|lt_i lt_j|$\geq$ P-π] $(((time \geq C\ /\ lt\_0=1)) \bigwedge !((time \geq C\ /\ (((lt\_0 - lt\_1 \geq -\pi) \bigwedge (lt\_1 - lt\_0 \leq \pi)) \bigvee ((lt\_0 - lt\_1 \geq -(P-\pi))\ /\ (lt\_1 - lt\_0 \leq (P-\pi))))$

To simplify this and run the models faster, we check convergence and closure and we split this property to know what achieved precision is exactly.

```
states_with_precision_1_after_54_ticks: (time >= 54) /\
((lt_0 - lt_1 = 1) \/ (lt_1 - lt_0 = 1) \/ (lt_0 - lt_1 = 23)
\/ (lt_1 - lt_0 = 23))
```

5.3.4 Model example

The code in Appendix D is an example of the EVMDD model for the protocol with a two node, complete topology (all nodes connected between them), delay=2 and d=1 (network imprecision).

5.4 Model Generator

As it is explained in Section 2.2, EVMDD does not allow types, arrays or other complex structures; only global, integer and simple variables. So, the bigger the model is, the longer and more difficult to create. Therefore, it is necessary the support of a tool written in a procedural language like Java, C... This tool is a model generator that is specific for each system that is modeled in EVMDD. In this case study, the model generator has been implemented in Java. It receives the information about the protocol in parameters, calculates the rest of the variable values following the protocol formulas and it generates the corresponding model to be analized by the EVMDD-smc. The user can input the following parameters via the command line:

```
-D, --delay=num
        Set event response delay time.

-d, --imprecision=num
        Set network imprecision.

-P, --period=num
        Set timeout period
```

```
-p, --precision=num
        Set desired synchronization precision for the results.

-I, --interleaving=num
        Set interleaving type.
                1: single phase
                2: biphase

-c, --config=filename
        Set path to parameter configuration file.

-T, --topology=filename
        Set path to network topology description file.

-G, --graph=code
        Set network topology to standard formats.
        complete-n      complete graph with n nodes.
        ring-n          ring with n nodes.
        linear-n        linear with n nodes.
        star-n          star with n nodes.
        grid-n          grid of nxn nodes.

-o, --output=filename
        Set path to output file.

-i, --init_lt=<i> [x,y]
        Set initial value of local timer i
        in the interval [x,y].
```

5.4.1 Config File

The configuration file allows the user to specify a custom topology. The following template must be followed:

```
Constants
D= n
d= m
P= x
pi= y
Topology
input->output
```

Implementation In order to define and initialize the variables, and set up the transitions and the properties, the model generator looks at the topology defined by the user. This topology is expressed as a List of Nodes. In the Figure 5.5 , it is possible to see this class in detail.

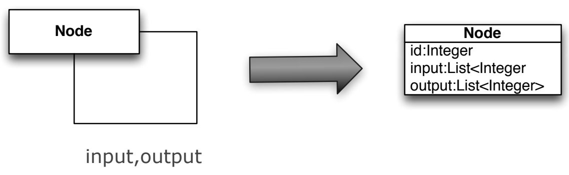


Figure 5.5: Node class in the Model Generator

5.5 Results

It is really important to interpret and to express clearly the results given by the EVMDD-smc when the models of the protocol have been executed. We remind the reader that the model checker can have two types of output: a counter example trace to a state that accomplishes the property or the number of states that match with the property. The trace can be analized by the trace visualizer in our AMC Visualization tool, but it is necessary to obtain the precission achieved by the protocol for every type of topology with different values in the parameters. This is what shows that the model and the system are correct.

In order to do this, the property Convergence and Closure is split in the model to know how many states have a specific clock precission at the convergence time (see Section 5.3.3) Therefore, it is possible to know the maximum achieved precission for each case.

5.5.1 Scripts

As we have already explained, it is necessary to obtain the maximum precission for different types of topologies with different delay and network imprecission. So, there is a need to create a bunch of models and to execute them in EVMDD-smc in order to obtain the mentioned results. Indeed, this has to be done automatically to save time and to enable the user to get these results as easy as possible.

Thus, we test the model by running a script that examines a series of selected topologies for a given number of nodes, delay and network imprecission.

```
for (the choice of parameters)
do
  generate-model
  run-model > output
done
extract-data-from-output    (Report Analyzer)
generate-pdf-report
```

Running this, it is expected not to have any counterexample and obtain a maximum precission ($\pi = 1$) for every case. Actually, the real π would be equal to 0, but due to the asynchronous interleaving, every π in the results will be, in fact, (real π) + 1. If we encountered states over the expected precission, then we would have to look into where the problem is, with the model or even with the protocol.

5.5.2 Result Analyzer

Yet generated the models and obtained the outputs, the results are interpreted and shown in pdfLaTeX tables. This is done by a Result Analyzer that has been developed in Java. It is basically a class that:

- Reads the EVMDD output: in this case, it is not a trace, it is the number of states accomplishing the property, to see the final precision.
- Produces PDF reports from this output with a table for each type of topology including the following information:
  - K - Number of nodes in the topology
  - D - Delay
  - d - Network imprecission
  - C - Convergence time
  - P - Synchronization period
  - T_s - Graph threshold
  - π - Clock precision at C time
  - Expected π - It is the maximum expected precission when the global clock is C_{init}: (K-1)*γ
  - Number of states
  - Time to create the state space

So far, it has been possible to check topologies up to five nodes, due to the huge number of states produced and memory issues in the EVMDD-smc. In the Appendix E, the reports showing the results for a complete topology (where all the nodes are connected among them) of 3 and 4 nodes are presented. In them, it is possible to see that for all the cases, the synchronization is achieved.

Chapter 6

Conclusion and future work

Along the document it has been possible to see how the AMC tools support the automation of the model checking process and increase efficiency in:

- Testing exhaustively the system
- Validating models
- Verifying models

Thus, supporting tools are necessary to identify errors, both in the model and in the system. Moreover, it is really important to have flexibility when organizing data in visualization tools, because it is the key that makes the user analyze the output clearly. In our case, the grouping goes through sections, groups and variables, allowing a hierarchy of three "generations". So, it would be desirable to extend it to wider hierarchies, to enable long nested properties in the variables. Including a tree structure in the tables could be a solution.

On the other side, the simulator is a really useful tool to check the behavior of EVMDD models. Since often it takes a long time to obtain an output from them, because of the vast number of states, using the simulator allows the user to keep track of the actions and, also to identify possible errors.

Nevertheless, the biggest problem would be the necessity of having a different simulator for each model checker, due to:

- Different behavior: model checkers allowing synchronous or asynchronous behavior.
- Different parser for the syntax: each model checker has a different code, what implies a different parser.
- Different type of logic: depending on the model checker, if it accepts CTL or LTL

Because of this generalization problem, this simulator could be generalized creating a XML template format that standarized the model checker input. This way would have the objective of simulate as much model checkers as possible. Obviously, creating XML standard formats still would not be enough to make these tools as completely universal and useful for other cases. It would also be necessary an adaptation library that made easier the transformation from different model checkers to the XML templates.

Summarizing, although there is a big "fragmentation" in the model checking field, where there are different types of model checkers, outputs, behaviors... We have been able to support the whole process, identifying automation points and helping the user to check the system in a much easier way. Furthermore, we have include visualization tools to analyze outputs, where there is still a big lack of them. Anyway, the next step in the work to support the model checking process is to make the tools more adaptable and standard to different model checkers and, indeed, make them as flexible as possible.

Chapter 7

Aknowledgements

Thank you to Radu Siminiceanu and Santiago Escobar for the opportunity and mentoring the project and to Mahyar Malekpour for his explanations about the protocol.

Appendix A

User manual: How to generate, run and analyze an EVMDD model for the clock synchronization protocol

1. Create the model using the model generator:
   (a) Open the terminal
   (b) Change the directory where Generator.jar file is.
   (c) Run the following command in the terminal:
   ```
   java    jar  Generator.jar  [parameters]
   ```
   (d) This will create an output file *.sm. This is the EVMDD model, that checks if there are nodes that converge within more than the given precision pi.
2. Execute the model:
   (a) Download EVMDD library and model checker in: ''http://research.nianet.org/~radu/evmdd/''
      i. Choose the last version 0.0.8b
   (b) Extract both of them:
   ```
   tar -xf evmdd-0.0.8.tar.gz
   tar -xf evmdd-smc-0.0.8.tar.gz
   ```
   (c) Compile the libraries:
      i. Go to the extracted folder of the library
      ii. Execute ./configure and follow the steps.
   (d) Compile the model checker:
      i. Go to the extracted folder of the model checker
      ii. Execute ./configure and follow the steps.
      iii. If the process is correct, it generates the binary file in the src folder. It is recommended to copy it in a workspace, where the models are going to be executed.

(e) Execute the model that you have generated before with the generator.

   i. Go to the src folder where evmdd-mcc is or workspace where you copied it.

   ii. Obtain a counterexample trace

   ```
   ./evmdd-smc --trace --dist -q < model.sm > trace.out
   ```

   iii. You can also know what states accomplish the property

   ```
   ./evmdd-smc  witnesses < model.sm
   ```

3. Simulate the model

   (a) Open the MCVisualizer.jar

   (b) Go to File ->Open Model.

      i. Select the model.sm created before.

   (c) Simulate the states.

   (d) Go to File ->Settings to set up the organization of the variables and groups.

4. Analyze the trace:

   (a) Open the MCVisualizer.jar

   (b) Go to File ->EVMDD trace: transforms the .out trace into a XML trace.

      i. Select the trace.out with the counterexample trace that you have obtained executing the model

   (c) Go to File ->Open Trace

   (d) Explore the trace

   (e) Go to File ->Settings to set up the organization of the variables and groups.

5. Analyze the model checker output

   (a) Open the terminal

   (b) Change the directory where ResultAnalyzer.jar file is.

   (c) Run the following command in the terminal:

   ```
   java  jar ResultAnalyzer.jar m n > results.tex
   ```

   [m,n] is the range of topologies whose results are going to be displayed.

   (d) Compile results.tex in pdfLaTeX to obtain the .pdf file and see the results clearly. ''http://www.latex-project.org/ftp.html''

Appendix B

XML output template

```
<?xml version="1.0"?>
<trace>
<section name="Global_Section">
        <group name="General">
                <variable name="time">
                <values>
                        <val change="0">0</val>
                        <val change="1">1</val>
                        <val change="0">1</val>
                </values>
                </variable>
        </group>
</section>
<section name="Section_monitors">
        <group name="0_1">
                <variable name="monturn_0_1">
                <values>
                        <val change="0">0</val>
                        <val change="1">1</val>
                        <val change="0">1</val>
                </values>
                </variable>
                <variable name="montext_0_1">
                <values>
                        <val change="0">0</val>
                        <val change="1">1</val>
                        <val change="0">1</val>
                </values>
                </variable>
        </group>
        <group name="1_0">
                <variable name="monturn_1_0">
                <values>
```

```
                        <val change="0">0</val>
                        <val change="1">1</val>
                        <val change="0">1</val>
                </values>
                </variable>
                <variable name="montext_1_0">
                <values>
                        <val change="0">0</val>
                        <val change="1">1</val>
                        <val change="0">1</val>
                </values>
                </variable>
        </group>
</section>
</trace>
```

Appendix C

Parser in JavaCC

```
PARSER_BEGIN(Model)
public class Model
{
Model parser = new Model(System.in);
parser.input();
}
PARSER_END(Model)
/**
 * LEXIC PART
 */
SKIP :
{
  " "
| "\t"
| "\n"
| "\r"
}

TOKEN : /* OPERATORS */
{
  < TPLUS : "+" >
| < TMIN : "-" >
| < TMUL : "*" >
| < TDIV : "/" >
| < TMOD : "%" >
| < TPOW : "^" >
| < TLE : "<=" >
| < TGE : ">=" >
| < TLT : "<" >
| < TGT : ">" >
| < TEQ : "=" >
| < TNEQ : "!=" >
| < TNOT : "!" >
| < TOR : "\\/" >
| < TAND : "/\\" >
| < TAX : "AX" >
| < TAG : "AG" >
| < TAF : "AF" >
| < TA : "A" >
| < TEG : "EG" >
| < TEX : "EX" >
| < TEF : "EF" >
| < TE : "E" >
| < TX : "X" >
| < TF : "F" >
| < TG : "G" >
| < TU : "U" >
| < TR : "R" >
}

TOKEN :
{

  < #DIGIT : [ "0"-"9" ] >
| < LETTER : (["a"-"z"]) >
| < TDECL : "Declarations" >
| < TINIT : "Initial states" >
| < TTRANS : "Transitions" >
| < TPROPS : "Properties" >
| < TNUM : (< DIGIT >)+ >
| < TIDENT: ((<LETTER>|< DIGIT >|"_"|"-"))+ >
| < TTT : ("true"|"True") >
| < TFF : ("FALSE"|"False"|"false") >
| < TLPAR : "(" >
| < TRPAR : ")" >
| < TLBRA : "[" >
| < TRBRA : "]" >
| < TCOMMA : "," >
| < TPOINTS : ":" >
| < TARROW : "->" >
| < TPRIME : "\'" >
}


SPECIAL_TOKEN:
{
  <MULTI_LINE_COMMENT: "/""*" (~["*"])* "*" ("*"
| (~["*","/"] (~["*"])* "*"))* "/" >
```

```
}

/**
 * SINTACTIC and SEMANTIC PART
 */
boolean input():
{}
{
  LOOKAHEAD(4) decls() inits() events() properties()< EOF >{return true;}
|
  LOOKAHEAD(4) decls()  inits()  events() < EOF > { return true; }
}

void decls():
{
}
{
  < TDECL > decls2()
}
void decls2():
{
}
{
          decl() decls2()
|{}
}

void inits():
{
}
{
  < TINIT >  inits2()
}
void inits2():
{
  Node ininode;
}
{
          ininode = bool_exp()
          {
            mm.initialize(ininode);
          } inits2()
|
          {}
}
```

```
void events():
{
}
{
  < TTRANS >   events2()
}
void events2():
{
}
{
        event() events2()
|
        {}
}
void event():
{
  Transition trans;
  Token name;
}
{
  LOOKAHEAD(2) name = < TIDENT > <TPOINTS > trans = event_body()
  {
    trans.setName(name.image);
    mm.addTransition(trans);
  }
|
  trans = event_body()
  {
    mm.addTransition(trans);
  }
}
Transition event_body():
{
  Node leftexp, rightexp;
  Transition trans;
}
{
        leftexp = bool_exp() <TARROW > rightexp = bool_exp()
        {
          trans = new Transition();
          trans.setLeftSide(leftexp);
          trans.setRightSide(rightexp);
          return trans;
        }
}
```

```
void decl():
{
    Token varname;
}
{
  varname = < TIDENT > < TLBRA > < TNUM > < TCOMMA > < TNUM > < TRBRA >
                                   {
                                     mm.declareVariable(varname.image);
                                   }
}

Node bool_exp():
{
  Node resnode,rightnode ;
}
{
  LOOKAHEAD(4) resnode = atom_bool_exp()
  (< TOR > rightnode = bool_exp()
  {
        resnode = new Node("or",resnode,rightnode);
  }

  |< TAND > rightnode = bool_exp()
  {
        resnode = new Node("and",resnode,rightnode);
  })*
  {
    return resnode;
  }

| LOOKAHEAD(4) < TNOT > resnode= bool_exp()
{
  resnode = new Node("!",resnode,null);
}
 (< TOR > rightnode = bool_exp()
  {
        resnode = new Node("or",resnode,rightnode);
  }

  |< TAND > rightnode = bool_exp()
  {
        resnode = new Node("and",resnode,rightnode);
  })*
  {
    return resnode;
  }
```

```
|LOOKAHEAD(4) < TLPAR > resnode = bool_exp() < TRPAR >{
  resnode = new Node("()",resnode,null);
}
        (< TOR > rightnode = bool_exp()
  {
        resnode = new Node("or",resnode,rightnode);
  }

  |< TAND > rightnode = bool_exp()
  {
        resnode = new Node("and",resnode,rightnode);
  })*
  {
    return resnode;
  }
}

Node atom_bool_exp(): //boolean
{
  Node resnode, leftnode, rightnode;
}
{
  < TTT >
  {
    resnode = new Node("true",null,null);
    return resnode;
  }
| < TFF >
{
        resnode = new Node("false",null,null);
        return resnode;
}
| LOOKAHEAD(2) leftnode = int_exp() < TEQ > rightnode = int_exp()
{
  resnode = new Node("=", leftnode,rightnode);
  return resnode;
}
| LOOKAHEAD(2) leftnode = int_exp() < TNEQ > rightnode = int_exp()
{
  resnode = new Node("!=", leftnode,rightnode);
  return resnode;
}
| LOOKAHEAD(2) leftnode = int_exp() < TLT > rightnode = int_exp()
{
  resnode = new Node("<", leftnode,rightnode);
  return resnode;
```

```
}
| LOOKAHEAD(2) leftnode = int_exp() < TLE > rightnode = int_exp()
{
  resnode = new Node("<=", leftnode,rightnode);
  return resnode;
}
| LOOKAHEAD(2) leftnode = int_exp() < TGT > rightnode = int_exp()
{
  resnode = new Node(">", leftnode,rightnode);
  return resnode;
}
| LOOKAHEAD(2) leftnode = int_exp() < TGE > rightnode = int_exp()
{
  resnode = new Node(">=", leftnode,rightnode);
  return resnode;
}
}
Node int_exp():
{
  String variable, cnst;
  Node resnode, rightnode, value;
}
{
  LOOKAHEAD(2) variable = var()
  {
    resnode = new Node(variable,null,null);
  }
  ( < TPLUS > rightnode = int_exp()
{
  resnode = new Node("+",resnode,rightnode);
}

 | < TMIN > rightnode = int_exp()
 {
  resnode = new Node("-",resnode,rightnode);
}
 | < TMUL > rightnode = int_exp()
 {
  resnode = new Node("*",resnode,rightnode);
 }
 | < TDIV > rightnode = int_exp()
 {
  resnode = new Node("/",resnode,rightnode);
 }
 | < TMOD > rightnode = int_exp()
 {
    resnode = new Node("%",resnode,rightnode);
  }
  )*
  {
    return resnode;
  }

| LOOKAHEAD(2) cnst = constant()
  {
    resnode = new Node(cnst,null,null);
  }
  ( < TPLUS > rightnode = int_exp()
{
  resnode = new Node("+",resnode,rightnode);
}

  | < TMIN > rightnode = int_exp()
  {
  resnode = new Node("-",resnode,rightnode);
}
  | < TMUL > rightnode = int_exp()
  {
  resnode = new Node("*",resnode,rightnode);
  }
  | < TDIV > rightnode = int_exp()
  {
  resnode = new Node("/",resnode,rightnode);
  }
  | < TMOD > rightnode = int_exp()
  {
  resnode = new Node("%",resnode,rightnode);
  }
  )*
  {
    return resnode;
  }

| LOOKAHEAD(3) < TMIN > resnode = int_exp()
{
  resnode = new Node("-", resnode, null);
}

( < TPLUS > rightnode = int_exp()
{
  resnode = new Node("+",resnode,rightnode);
}
```

```
 | < TMIN > rightnode = int_exp()
 {
  resnode = new Node("-",resnode,rightnode);
}
 | < TMUL > rightnode = int_exp()
 {
  resnode = new Node("*",resnode,rightnode);
 }
 | < TDIV > rightnode = int_exp()
 {
  resnode = new Node("/",resnode,rightnode);
 }
 | < TMOD > rightnode = int_exp()
 {
  resnode = new Node("%",resnode,rightnode);
 }
 )*
 {
   return resnode;
 }


| LOOKAHEAD(4) < TLPAR > resnode = int_exp() < TRPAR >{
  resnode = new Node("()",resnode,null);
}

( < TPLUS > rightnode = int_exp()
{
  resnode = new Node("+",resnode,rightnode);
}
 | < TMIN > rightnode = int_exp()
 {
  resnode = new Node("-",resnode,rightnode);
}
 | < TMUL > rightnode = int_exp()
 {
  resnode = new Node("*",resnode,rightnode);
 }
 | < TDIV > rightnode = int_exp()
 {
  resnode = new Node("/",resnode,rightnode);
 }
 | < TMOD > rightnode = int_exp()
 {
  resnode = new Node("%",resnode,rightnode);
 }
 )*
 {
   return resnode;
 }

}

String constant():
{
  Token t;
}
{
 t = < TNUM > {return t.image;}
}

String var():
{
  Token variable;
}
{
        LOOKAHEAD(2) variable = < TIDENT > < TPRIME >
        { return variable.image+"'"; }
|
        variable = < TIDENT >
        {return variable.image; }
}
```

Appendix D

EVMDD Model

The following code is an example of the EVMDD model for the protocol with a two node, complete topology (all nodes connected between them), delay=2 and d=1 (network imprecisson).

```
Declarations
   lt_0              [0,23]
   nodeturn_0        [0,1]
   chan_1_0          [0,1]
   montext_0_1       [0,1]
   msgtimer_0_1      [0,4]
   monturn_0_1       [0,1]
   lt_1              [0,23]
   nodeturn_1        [0,1]
   chan_0_1          [0,1]
   montext_1_0       [0,1]
   msgtimer_1_0      [0,4]
   monturn_1_0       [0,1]
   mturn             [0,1]
   nturn             [0,1]
   time              [0,54]
Initial states
   time             = 0
   mturn            = 1
   nturn            = 0
   lt_0 >=0
   nodeturn_0       = 0
   chan_1_0         = 0
   montext_0_1      = 0
   msgtimer_0_1     = 0
   monturn_0_1      = 1
   lt_1 >=0
   nodeturn_1       = 0
   chan_0_1         = 0
   montext_1_0      = 0
```

```
  msgtimer_1_0    = 0
  monturn_1_0     = 1
Transitions
global_time_inc:  time < 54 /\ mturn = 0 /\ nturn = 0 ->
time' = time + 1  /\ mturn' = 1 /\ nturn' = 0 /\ nodeturn_0' = 0
/\ monturn_0_1' = 1 /\ nodeturn_1' = 0 /\ monturn_1_0' = 1
global_last_turn:  time = 54 /\ mturn = 0 /\ nturn = 0 ->  mturn' = 1
/\ nturn' = 0 /\ nodeturn_0' = 0 /\ monturn_0_1' = 1 /\ nodeturn_1' = 0
/\ monturn_1_0' = 1
turn_to_nodes:  time <= 54 /\ mturn = 1 /\ nturn = 0 /\ nodeturn_0 = 0
/\ monturn_0_1 = 0 /\ nodeturn_1 = 0 /\ monturn_1_0 = 0 ->
mturn' = 0 /\ nturn' = 1 /\ nodeturn_0' = 1 /\ monturn_0_1' = 0
/\ nodeturn_1'=1
end_node_turn:  time <= 54 /\ mturn = 0 /\ nturn = 1 /\ nodeturn_0 = 0
/\ monturn_0_1 = 0 /\ nodeturn_1 = 0 /\ monturn_1_0 = 0 ->
mturn' = 0 /\ nturn' = 0 /\ nodeturn_0' = 0 /\ monturn_0_1' = 0
/\ nodeturn_1'=0
mon_basic_0_1:  monturn_0_1 = 1 /\ chan_1_0 = 0 /\ montext_0_1 = 0 ->
monturn_0_1' = 0
mon_basic_1_0:  monturn_1_0 = 1 /\ chan_0_1 = 0 /\ montext_1_0 = 0 ->
monturn_1_0' = 0
mon_0_1msg_delay:  monturn_0_1=1 /\ chan_1_0=1 /\ msgtimer_0_1<2 ->
monturn_0_1' = 0  /\ msgtimer_0_1' = msgtimer_0_1 + 1
mon_1_0msg_delay:  monturn_1_0=1 /\ chan_0_1=1 /\ msgtimer_1_0<2 ->
monturn_1_0' = 0  /\ msgtimer_1_0' = msgtimer_1_0 + 1
mon_0_1msg_receive:  monturn_0_1=1 /\ chan_1_0=1 /\ msgtimer_0_1>=1 ->
monturn_0_1' = 0  /\ chan_1_0' = 0 /\ montext_0_1' = 1
mon_1_0msg_receive:  monturn_1_0=1 /\ chan_0_1=1 /\ msgtimer_1_0>=1 ->
monturn_1_0' = 0  /\ chan_0_1' = 0 /\ montext_1_0' = 1
trans_0_e1:  nodeturn_0 = 1 /\ lt_0 < 1 /\ ( montext_0_1 > 0 ) ->
nodeturn_0' = 0 /\ lt_0' = 3 /\ montext_0_1' = 0
/\ msgtimer_0_1' = (1 - montext_0_1)*msgtimer_0_1
trans_1_e1:  nodeturn_1 = 1 /\ lt_1 < 1 /\ ( montext_1_0 > 0 ) ->
nodeturn_1' = 0 /\ lt_1' = 3 /\ montext_1_0' = 0
/\ msgtimer_1_0' = (1 - montext_1_0)*msgtimer_1_0
trans_0_e2:  nodeturn_0 = 1 /\ lt_0 >= 11 /\ ( montext_0_1 > 0 ) ->
nodeturn_0' = 0 /\ montext_0_1' = 0 /\ chan_0_1' = 1 /\ lt_0' = 3
/\ msgtimer_0_1' = (1 - montext_0_1)*msgtimer_0_1
trans_1_e2:  nodeturn_1 = 1 /\ lt_1 >= 11 /\ ( montext_1_0 > 0 ) ->
nodeturn_1' = 0 /\ montext_1_0' = 0 /\ chan_1_0' = 1 /\ lt_1' = 3
/\ msgtimer_1_0' = (1 - montext_1_0)*msgtimer_1_0
trans_0_e3:  nodeturn_0 = 1 /\ montext_0_1 = 0 /\ lt_0 >= 23 ->
nodeturn_0' = 0 /\ lt_0' = 0 /\ chan_0_1' = 1
trans_1_e3:  nodeturn_1 = 1 /\ montext_1_0 = 0 /\ lt_1 >= 23 ->
nodeturn_1' = 0 /\ lt_1' = 0 /\ chan_1_0' = 1
trans_0_e4_1:  nodeturn_0 = 1 /\ lt_0 >= 1 /\ lt_0 < 11
```

```
/\ ( montext_0_1 > 0 ) -> nodeturn_0' = 0 /\ montext_0_1' = 0
/\ lt_0' = lt_0 + 1
/\ msgtimer_0_1' = (1 - montext_0_1)*msgtimer_0_1
trans_1_e4_1: nodeturn_1 = 1 /\ lt_1 >= 1 /\ lt_1 < 11
/\ ( montext_1_0 > 0 ) -> nodeturn_1' = 0 /\ montext_1_0' = 0
/\ lt_1' = lt_1 + 1
/\ msgtimer_1_0' = (1 - montext_1_0)*msgtimer_1_0
trans_0_e4_2: nodeturn_0 = 1 /\ lt_0 < 23 /\ montext_0_1 = 0 ->
nodeturn_0' = 0 /\ lt_0' = lt_0 + 1
trans_1_e4_2: nodeturn_1 = 1 /\ lt_1 < 23 /\ montext_1_0 = 0 ->
nodeturn_1' = 0 /\ lt_1' = lt_1 + 1
Properties
  convandclos: ((time>=54) /\ !((((lt_0-lt_1 >=-2) /\ (lt_0-lt_1 <=2))
\/ ((lt_0-lt_1 <=-22) \/ (lt_0-lt_1 >=22))))) /\ mturn=0 /\ nturn=0
  states_with_precision_1_after_54_ticks: (time >= 54)
/\ ((lt_0 - lt_1 = 1) \/ (lt_1 - lt_0 = 1)
\/ (lt_0 - lt_1 = 23) \/ (lt_1 - lt_0 = 23))
  states_with_precision_2_after_54_ticks: (time >= 54)
/\ ((lt_0 - lt_1 = 2) \/ (lt_1 - lt_0 = 2)
\/ (lt_0 - lt_1 = 22) \/ (lt_1 - lt_0 = 22))
  states_with_precision_3_after_54_ticks: (time >= 54)
/\ ((lt_0 - lt_1 = 3) \/ (lt_1 - lt_0 = 3)
\/ (lt_0 - lt_1 = 21) \/ (lt_1 - lt_0 = 21))
  states_with_precision_4_after_54_ticks: (time >= 54)
/\ ((lt_0 - lt_1 = 4) \/ (lt_1 - lt_0 = 4)
\/ (lt_0 - lt_1 = 20) \/ (lt_1 - lt_0 = 20))
```

Appendix E

Result Reports

E.1 Complete Topology

K	D	d	γ	C	P	Ts	Max π	Expected π	State space	Time(sec)
3	1	0	1	39	18	5	1	3	1×10^6	0.07
3	1	1	2	72	33	10	1	5	3×10^7	0.78
3	2	0	2	72	33	10	1	5	2×10^7	0.47
3	2	1	3	105	48	15	1	7	1×10^8	2.12
3	3	0	3	105	48	15	1	7	1×10^8	1.39
3	3	1	4	138	63	20	1	9	3×10^8	4.46
3	4	0	4	171	63	20	1	9	3×10^8	2.94

K	D	d	γ	C	P	Ts	Max π	Expected π	State space	Time(sec)
4	1	0	1	60	28	6	1	4	8×10^9	1.29
4	1	1	2	112	52	12	1	7	3×10^{11}	20.29
4	3	0	3	164	76	18	1	10	1×10^{12}	36.26
4	4	0	4	216	100	24	1	13	8×10^{12}	94.45

List of Figures

Bibliography

[1] Ken Tindell. Real time systems by fixed priority scheduling. Technical report, Uppsala University, 1997.

[2] John Wordsworth. *Software Development With Z: A Practical Approach to Formal Methods in Software Engineering*. Addison-Wesley Pub, 1992.

[3] Mahyar R. Malekpour. A self-stabilizing distributed clock synchronization protocol for arbitrary digraphs. Technical report, NASA, 2011.

[4] Doron A. Peled Edmund M. Clarke Jr, Orna Grumberg. *Model Checking*. MIT Press, 1999.

[5] Pierre Roux Radu I. Siminiceanu. Model-checking with edge-valued decision diagrams. Technical report, NASA, 2010.

[6] Dijkstra. Dinning philosophers problem. Technical report, TUe, 1965.

[7] IEEE Computer Society, editor. *IEEE Recommended Practice for Software Requirements Specifications*, 1998.

[8] Java CC documentation.

[9] Java. *Introduction to Java CC*.

[10] Emilio Vivancos. *Introduction to Language Processors*. Universitat Politècnica de València, 2011.

[11] Oracle. What is java swing?