Master’s thesis

A Framework for Interacting with Parameterized and Infinite State Verification Tools

by

Rizwan Abbasi

2013-05-30
Institutionen för datavetenskap
Department of Computer and Information Science

Master’s Thesis

A Framework for Interacting with Parameterized and Infinite State Verification Tools

by

Rizwan Abbasi

2013-05-07

Supervisor:  Ahmed Rezine
                IDA, Linköpings universitet

Examiner:  Unmesh D. Bordoloi
                IDA, Linköpings universitet

ISRN #: LIU-IDA/LITH-EX-A--13/030--SE

Department of Computer and Information Science
Linköpings universitet
SE-581 83 Linköping, Sweden
Abstract

We develop a tool to explore the behavior of parameterized systems (i.e., systems consisting of an arbitrary number of identical processes that synchronize using shared variables or global communications) and to ease user interaction with tools that verify them. The tool includes a user friendly GUI that allows the user to describe a parameterized system and to perform guided, interactive or random simulation. This tool empowers the user to plug in several independent verifiers to perform verification. A mockup verifier is developed in order to facilitate the development of the tool and testing the required functionalities. The mockup verifier involves parsing descriptions of the parameterized systems to be analyzed. In order to interact with the verifier, the tool is user friendly and flexible in the sense that the user can plug in a verifier developed in any language as long as it allows to perform a number of basic computations on the parameterized system (such as the set of enabled transitions or the set of successor configurations). In order to plug in a new tool, our tool needs to be able to make use of these operations, for instance using a wrapper written for a verifier particular to a class of parameterized systems. Given these operations, our tool enables the user to carry out various types of simulations like random, interactive or guided simulations. Moreover, our tool can submit verification queries to the underlying verifier and walk the user through the generated counter examples as if it was a simulation session.
Acknowledgements

First of all, I am thankful to Allah the All Mighty and the creator of this universe who gave me strength and ability to complete this work. I am also very much thankful to my parents for their continuous encouragement, support and prayers without which it would be impossible to complete this thesis work. It is because of the prayers of my parents that I achieved so much in life. I warmly thank my brother, Zeshan, for his continuous support throughout my studies, especially in Sweden.

I am very much thankful to my supervisor Ahmed Rezine for his continuous help from the first day. He helped me a lot in understanding the requirements, in mathematical modeling and in implementation. Without the continuous help and encouragement of my supervisor, it would be impossible for me to complete this thesis work.

I am Thankful to my friend Usman Shaukat for his great help in technical aspects of this thesis work and for encouraging me. I am also thankful to my friends Syed Asmat Ali Shah, Muhammad Touqir Pasha and Abdul Mateen Malak for their support. I am thankful to all my friends for their help.

I dedicate my thesis work to my family.
## Table of Contents

Abstract ................................................................................................................................. 4
Acknowledgements ............................................................................................................... 5
Table of Contents .................................................................................................................. 6
List of Figures ....................................................................................................................... 7

1 Introduction ..................................................................................................................... 8
  1.1 Background: Testing and Verification ..................................................................... 8
  1.2 Problem statement ................................................................................................. 9
  1.3 Targeted Readers ................................................................................................... 9
  1.4 Objectives ............................................................................................................... 10
  1.5 Development Methodology .................................................................................... 10
  1.6 Organization of the Thesis ..................................................................................... 10

2 Background .................................................................................................................... 11
  2.1 Model Checking .................................................................................................... 11
  2.2 Parameterized Systems ......................................................................................... 11

3 Design ............................................................................................................................. 14
  3.1 Simulation Algorithm ............................................................................................ 15
    3.1.1 Random Simulation ......................................................................................... 15
    3.1.2 Interactive Simulation ..................................................................................... 16
    3.1.3 Guided Simulation .......................................................................................... 16
  3.2 Contract between tool and verifier ......................................................................... 17

4 Mockup design ................................................................................................................. 18

5 Development environment and tools ............................................................................. 21
  5.1 IDE language environment .................................................................................... 21
  5.2 ANTLR ................................................................................................................... 21
  5.3 Documentation development .................................................................................. 21

6 Results ............................................................................................................................. 22
  6.1 Screenshots ............................................................................................................ 22
    6.1.1 Random Simulation ......................................................................................... 22
    6.1.2 Interactive Simulation ..................................................................................... 28
    6.1.3 Guided Simulation .......................................................................................... 31
    6.1.4 Verification ...................................................................................................... 35

7 Future work ..................................................................................................................... 36

8 Bibliography .................................................................................................................... 37
List of Figures

Figure 2.1 An instance of simple linear parameterized system .................................................. 12
Figure 3.1 Design of the application, how the tool is interacting with verifiers ...................... 14
Figure 3.2 Simulation Algorithm .......................................................................................... 15
Figure 3.3 A simple trail file ............................................................................................... 16
Figure 4.1 UML class diagram of Mockup ........................................................................... 18
Figure 4.2 Abstract class Guard .......................................................................................... 19
Figure 4.3 Abstract class Assignment .................................................................................. 19
Figure 4.4 Abstract class Choice ......................................................................................... 20
Figure 4.5 Abstract class Transition .................................................................................... 20
Figure 6.1 Main interface of the application when launched .................................................. 22
Figure 6.2 ReOpen button error dialog box ........................................................................ 23
Figure 6.3 Open button dialog box when clicked on Open button ....................................... 24
Figure 6.4 After opening file containing description of the system ..................................... 24
Figure 6.5 Find button in action .......................................................................................... 25
Figure 6.6 Example grammar description .......................................................................... 26
Figure 6.7 Random Simulation ........................................................................................... 27
Figure 6.8 Random Simulation first result ......................................................................... 27
Figure 6.9 Single step output ............................................................................................... 28
Figure 6.10 Random simulation results .............................................................................. 28
Figure 6.11 Interactive simulation ....................................................................................... 29
Figure 6.12 Interactive simulation next choice ................................................................... 30
Figure 6.13 Completed interactive simulation ................................................................... 30
Figure 6.14 Interactive simulation out of bound error handling ........................................ 30
Figure 6.15 Interactive simulation string input error handling ........................................... 31
Figure 6.16 A simple trail file .............................................................................................. 32
Figure 6.17 Save Trail, saving the simulation results ........................................................... 32
Figure 6.18 Guided simulation browse button ..................................................................... 33
Figure 6.19 Guided simulation Simulate button clicked ...................................................... 33
Figure 6.20 Guided simulation popup dialog box ............................................................... 34
Figure 6.21 Guided simulation completed .......................................................................... 34
Figure 6.22 Verify button to perform verification ............................................................... 35
1 Introduction

This document reports on 30 credit points worth thesis work. Verification is a process to prove that the software program is error free, thus we can say that program is correct. There are several famous verification tools that come with a friendly user interface (SPIN [1], PRISM [2], and UPPAAL [6]). Several verification tools, more precisely targeting parameterized systems, are developed at ESLAB but lack a user friendly GUI to ease their utilization. The objective of this thesis work is to develop a user friendly tool with an intuitive Graphical User Interface that allows the user to explore the behaviors of the parameterized systems to be analyzed. The tool will enable the interaction with other verification tools for parameterized systems and will make it simple to plug in several independent verification tools. As a result, this work helps in developing a framework to interact with parameterized and infinite state verification tools. This chapter introduces testing and verification, the problem statement, the targeted readers, the objectives and the adopted methodology. The chapter ends with an overview of the rest of the report.

1.1 Background: Testing and Verification

We use computers in our daily life and these are evolving with the passage of time. New efficient and compact sized hardware is arriving in the market. These systems have extended functionalities about which we could just dream some years ago. The evolution and growth in computer hardware industry played an important role in making computer software complex. Computer based systems are used widely in telecommunications, air traffic controllers, multimedia applications and control systems for cars. Without proper checks and efforts, it would be naïve to assume that the computer software monitoring and controlling these systems is free of errors. These errors can result in severe consequences depending on the type of the system. One example is of a PlayStation that automatically restarts. Another example is about an air traffic control system that gives green sign to two airplanes landing on the same track at the same time. We can tolerate a PlayStation with such a problem but the condition is critical in the second example. It is vital to make sure that such situations do not occur and that the computer systems do behave as expected. Therefore, we need to be able to check computer programs against desired properties. A desired property of a PlayStation is that it should not restart itself and a desired property of an air traffic control system is that it should not allow two airplanes to land at the same track on the same time.

Identifying and fixing the faults and malfunctions in computer programs is an important problem in software engineering. The objective is to uncover those computations that do not satisfy the desired properties of the system. Substantial resources are assigned in IT projects to ensure that software behaves as expected. “Software testing is the process to examine a software program in order to find the differences between current conditions and required conditions” [5]. These differences are known as bugs. Software testing is a way to explore the features of the computer program. The main objective of software testing is to reduce the number of errors or failures and is done after the software is programmed. Software testing is carried out by writing test cases, for example Boundary value testing targets to find errors at the boundaries of the input range. Software testing is valuable in finding errors computer programs but it cannot guarantee that all errors are found. On the other hand, verification is a process to prove that the software program is error free, thus we can say that program is correct. Verification is carried out before, during and after the implementation phase of the software development life cycle. Verification aims at mathematically proving the correctness
of the system and is often performed on a mathematical model of the targeted computer program. In this process the desired property is verified against all the possible scenarios. The systems, discussed earlier, should be verified using a verification tool to ensure that these systems are bug free. The tool we develop in this thesis allows interacting with different verifications tools.

1.2 Problem statement

Today, we are using many computer based systems in our daily life activities. These computer based systems are operated using software that can give rise to systems with complex behaviors. An important family of such systems consists in the family of parameterized systems, i.e., systems consisting of an arbitrary number of identical processes that synchronize using shared variables or global communications. Such systems are widely used in cache coherence protocols, mutual exclusion algorithms, sensor systems, telecommunication protocols and bus protocols. It can be easy to introduce errors and bugs in such systems. Such errors can be benign or can result in high material and even severe human costs. Here comes the role of software verification in general, and of model checking in particular. Model checking is a technique that ensures that (a model of a) software is bug free with respect to a given property. There are many verification tools available. Among these, Spin [1] for the verification of communication protocols and distributed systems, PRISM [2] for the verification of probabilistic systems and UPPAAL [6] for the verification of timed systems are quite popular. Some parameterized verification tools are developed at the Embedded Systems Laboratory (ESLAB) in Linköping University, but these are command line and not user friendly. Interaction with these tools using command line is not easy. Many of these tools share common “interaction behaviors”. A user edits a description of a system, explores some executions, sends verification requests to the verifier and explores the returned result. One approach is to build a rich GUI for each such tool. Here, we propose to build a common intuitive and user friendly Graphical User Interface that allows exploring behaviors of parameterized systems while interacting with various verification tools in such a way that one can plug in several independent verification tools. These verification tools can be developed in any language. In this tool, the user should be able to write and modify the description of the system to be used by the verifier. The user should be able to verify the description of the system using the verifier. In addition, the user should be able to perform various types of simulations (random, interactive and guided). This tool should have various independent verifiers to be used for verification purposes. The graphical user interface (GUI) should allow the user to give the description of the system and to verify/simulate it in a uniform and intuitive way. This work is the first step in building a framework with a friendly graphical user interface to ease interacting with verification tools targeting parameterized systems.

1.3 Targeted Readers

Our tool is helpful for the developers of parameterized verification tools. Our tool can also help researchers in the field of parameterized systems. The developers of such tools can plug in a verification tool developed in any language as long as it allows performing a number of basic computations on the parameterized system (such as the set of enabled transitions or the set of successor configurations). In order to plug in a new tool, our tool needs to be able to make use of these operations, for instance using a wrapper written for a verifier particular to a
class of parameterized systems. Given these operations, our tool enables the user to carry out various types of simulations like random, interactive or guided simulations.

1.4 Objectives

The main objective of this thesis work is to develop a user friendly tool with an intuitive Graphical User Interface that allows exploring behaviors of a parameterized system to be analyzed. It should also be simple to plug in different verification tools independently of the language or the algorithms they use. In our tool, the user should be able to write and modify the description of the system to be sent to the verifier and obtain as a result either that the parameterized system is correct or a sequence that violates the desired property. The user should also be able to verify the description of the parameterized system, with the help of a verifier to compute simple functions for the particular class of parameterized systems, to perform various types of simulations (random, interactive and guided). Our tool will therefore simplify the interaction with various independent verifiers, targeting different classes of parameterized systems, with an intuitive and simple to use graphical user interface (GUI) to verify and simulate these systems.

1.5 Development Methodology

First, we performed a literature review about model checking, parameterized systems and their verification [3, 4, 10]. Then I tried several tools [1, 2, 6] in order to get an idea of how to specify a system and a property to be analyzed and about how to perform random, guided and interactive simulation. Instead of directly using a verification tool, we preferred to write a simple tool to perform the functions that will be required by our tool from a verification tool. We therefore implemented two tools: the tool that is the objective of this work, and a mock up verifier tool to experiment with the different interaction possibilities. For this, we wrote several prototypes as they help in getting concrete feedback about the product and make it possible to implement the changes in the next prototype. Small prototypes of the tool were developed and demonstrated to get valuable feedback. This methodology helped greatly in understanding the exact requirements for the tool. Various iterations have been performed for this development methodology that lead to the current version of the tool.

1.6 Organization of the Thesis

In the next chapter we introduce important concepts on which this work relies. In chapter 3, we will focus on the design aspects of the tool. In chapter 4, we will explain in detail that a mockup verifier was developed. Instead of developing a new verifier, a mockup verifier was developed that saved a lot of time and helped in implementing as well as testing all the functionality that was expected from this tool. We will also define contract between the tool and the verifier. In chapter 5, we will talk about the development environment and tools used during this thesis work. In chapter 6, we will show and explain the results and in chapter 7, future work will be discussed.
2 Background

Our tool will simplify the interaction with verification tools that perform model checking on parameterized systems. We introduce in this chapter the model checking approach and what we mean by parameterized systems.

2.1 Model Checking

Model Checking is a verification technique that is used to check that a model (of the system to be verified) complies with, or implements a, desired behavior. For computer programs, model checking takes as input a description of the computer program and a desired property to be checked. The model checking tool can result in two kinds of outputs. It can guarantee that the desired property holds or it can give a counter example or trace if the desired property is not satisfied. Returning a counter example when the desired property is not satisfied is an appealing feature of model checking that helps a lot in identifying as well as in fixing errors. In computer systems, the correctness of a software application can be represented in the form of various properties. These properties can be divided into two main categories so that most interesting properties can be classified in one of these two categories. The first category of properties is the one of safety properties where nothing bad ever happens. For example, that air traffic control never allows two airplanes to land at the same track at the same time. Another example is that a coffee machine pours coffee when there is no cup. A safety property is violated whenever a bad configuration is reached. The second category of properties is the one of liveness properties specifying that good things ultimately happen. For example, when the appropriate button of a coffee machine is pressed then coffee is eventually produced. A liveness property is violated when the system is running and it does not come across a good configuration. The verification tools implemented at ESLAB perform model checking on over-approximations of parameterized systems to check safety properties. These tools are quite complex to manipulate. Our verifier that we implemented in this thesis to experiment with interacting with a verification tool, pretends to perform model checking of a parameterized system.

2.2 Parameterized Systems

The tool developed in this thesis work allows exploring behaviors of parameterized systems and can interact with verification tools in such a way that one can plug in several independent verification tools for different classes of parameterized systems. “Parameterized systems are composed of an arbitrary number of identical state machines running in parallel” [3, 4]. Each state machine has local variables and can have access to a number of shared variables. Transitions of a state machine can also depend on the state of other state machines. State machines are also referred to as processes or components. Parameterized systems are widely used in cache coherence protocols, mutual exclusion algorithms, sensor systems, telecommunication protocols and bus protocols.

More formally, a parameterized system P is a triple (Q, X, T) where Q represents the states of one process, X is the set of local variables and T represents the transitions. A particular transition can be of the following form.
In this transition rule $t$, $q$ is the current state, $q'$ is the next state and both $q, q' \in Q$. A guard is a predicate that depends on the values of the current states and variables. If all the guards are satisfied, the assignments to the variables will be applied and the state machine will go from the current state $q$ to the next state $q'$. A configuration is a snapshot of the parameterized system at some time that has information about the states and values of all variables and processes in the system. A component can have local and global transitions. A local transition does not depend on or change the state and variables of the other processes in the configuration. A global transition depends on, and may change, the states and variables of the other processes in the configuration. In the beginning, we initialize all the processes and that state of the system is called an initial configuration. A transition can change the state of a single process or many other processes. One objective of this work is to simplify the interaction with tools that verify the correctness of such systems irrespective of the size of the considered configurations. The size of an instance of the parameterized system, or of a configuration, is the number of processes in the instance, or configuration.

**Example:**

Parameterized systems can be organized into trees, linear array structures or unstructured. They can involve variables of different types: Booleans, integers or arrays. They can communicate using different kinds of transitions. For example, linear parameterized systems organize there processes in a linear array. Consider the instance of a linear parameterized system depicted in Figure 2.1. This instance has five processes, each with only two possible states, idle and critical. In the beginning all state machines are in the idle state. A state machine can change the state by following one of the two transition rules. First one is the global transition that says a state machine can move from the idle state to the critical state if and only if all other state machines are in their idle state. This condition is denoted by $\forall_{LR}$ (idle) that says that transition can only occur if all the state machines on the left of the current state machine and all the state machines on the right of the current state machine are in idle state. The second transition is a local transition rule that says a state machine can also come to idle from critical state at any time. A state machine in this linear parameterized system can only be in one state at a time either idle or critical.

![Figure 2.1 An instance of simple linear parameterized system.](image)
An example of a safety property for such a simple parameterized system is mutual exclusion. It specifies that, in any case, no two state machines can be simultaneously in their critical section. A state machine can access shared resource only when it is in critical section. There are many mutual exclusion algorithms (Burns [3, 4], Dijkstra [3, 4] and Szymanski [4]) that can be modeled as linear parameterized systems.
3 Design

In this chapter, we will focus on the design aspects of the tool developed in this project. First we will introduce our design choices. Then we will explain how we designed a mockup verifier that allowed us to experiment with different strategies to plug in verifier. This mockup verifier allowed us to focus on how to plugin general verifier instead of focusing on particular ones. Also this saved a lot of time and helped in implementing as well as testing several of the functionalities that were expected from this tool. We will also define the communication between the tool and the verifier. At the end of this chapter we will talk about the development environment and tools used in this project.

The objective of this thesis work is to develop a user friendly tool with an intuitive Graphical User Interface that allows exploring behaviors of parameterized systems and should allow to interact with various verification tools in such a way that one can plug in several independent verification tools.

![Diagram](image)

*Figure 3.1 Design of the application, how the tool is interacting with verifiers*

Figure 3.1 gives an overview of the design of this framework. It can be seen in the figure that there are several verifiers interacting with the tool. The tool developed in this thesis work asks the verifier for different operations to perform various types of simulation. It is designed to
ensure that it is independent of the verifier. The intuition behind separating the interface and the verifier is that our interface doesn’t need to know anything about the verifier. Anything in the sense that, interface does not care in which programming language the verifier is implemented. These verification tools can be written in any language. For example, the tool is designed to work perfectly even if verifier 1 is implemented in java, verifier 2 is implemented in C++. It is the responsibility of the verifier to parse the input given by the user, so that the verifier knows everything about the description of the system. For now, the interface is totally unaware about the class of parameterized systems being handled. The interface will perform three different types of simulations: random simulation, interactive simulation and guided simulation. The disadvantage of separating description from the tool is that we cannot have pretty nice printing for configurations and transitions. As the tool is unaware about the description so the transitions cannot be visually seen using arrows and ovals in a graphical manner.

3.1 Simulation Algorithm

Figure 3.2 shows the simulation algorithm that takes parameterized system P(Q, X, T) plus trace for guided simulation as input. Here, Q represents the states of one process, X is the set of local variables and T represents the transitions.

| Input: A parameterized system P(Q,X,T) (+ trace for guided simulation) |
| conf = get initial(P) |
| while enabled do |
| enabled = get_enabled(P, conf) |
| if not enabled.empty() |
| T = choose(enabled) |
| conf = fire (T, conf) |
| return |

Figure 3.2 Simulation Algorithm

3.1.1 Random Simulation

The algorithm in figure 3.2 takes the description of the parameterized systems as input. Then, it will get initial configuration and will store it. The simulation loop will start and will continue execution until there is no enabled transition. We will pass that configuration and parameterized system to a function get_enabled that will return us all the enabled transitions. After getting enabled transitions, we will check if enabled set is not empty. In random simulation, simulation choices are randomly selected using a random number generator. We will randomly choose one enabled transition and will fire it on the configuration to get the new configuration. Now, the simulation process will be repeated within the loop with new configuration.
3.1.2 Interactive Simulation

The algorithm shown in figure 3.2 takes the description of the system as input. After having description, it gets initial configuration and stores it. Simulation loop will start execution until there are enabled transitions. The configuration and the parameterized system is passed as an argument to a function get_enabled and this function will return all the enabled transitions. We will check that if we have enabled transitions and then we will ask from the user to choose one transition from these enabled transitions. Once the user has chosen one transition, we will fire that transition on the configuration and will get next configuration. Now, the simulation loop will be executed using this updated configuration. Figure 6.11 shows you the choices printed in the popup dialog box titled Interactive Simulator.

3.1.3 Guided Simulation

Guided Simulation requires a trail to carry out simulation. Whenever you perform random or interactive simulation, you can save the trail by pressing ‘Save Trail’ button and this will store simulation results in a file. This trail file can be used later to perform guided simulation. An example of a very small trail file can be seen below in figure 3.3.

<table>
<thead>
<tr>
<th>Transition#1, 0, [condition=false, pc=1] : [flag=up, door=unlocked] &lt;flag=down, door=locked]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transition#2, 1, [condition=true, pc=7] : [flag=up, door=unlocked] &lt;flag=down, door=locked]</td>
</tr>
</tbody>
</table>

*Figure 3.3 A simple trail file*

Each row contains a record and this trail file has only two records. The verifier will first parse this trail file and then for each record it will fire transition on the configuration using the choice and will match the resulting configuration. Like interactive simulation, you can also control the speed of guided simulation by entering iterations in a popup dialog box that appears when you click simulate button after choosing guided simulation.

We considered two design choices for our tool. In the first choice the tool parses the description of the system and thus knows everything about the system. The advantage is that the tool would be rich enough to print the transitions in a nice way. For example, use of different colors to highlight various type of transitions. Moreover, have ovals to represent local variables and squares to represent shared variables. The disadvantage of this design choice is that the tool would require an important adaptation effort in order to plug in a new verifier targeting a new class of parameterized systems. In the second design choice, it is the responsibility of the verifier to parse the description of the system and it is only the verifier who knows everything about the system, thus our tool is unaware about the description of the system. Whenever the tool requires successors, transitions or initial configuration, it requests the verifier and the verifier provides the required output in desired format. We have adopted this design choice because in this way we can plug in more than one independent verifier. Moreover, we have made communication between the tool and the verifier clear. For instance, observe that the simulation algorithm given in figure 3.2 has various functions. During simulation, the tool will ask the verifier to give back initial configuration and transition pattern and these values will be printed in the tool. Now the tool will ask the verifier
(currently our mockup verifier) to give back enabled transitions and one of the transitions is fired on the configuration to get next configuration (computation of successors). Choosing one transition depends upon the mode of the simulation, in random simulation a random number generator is used to randomly choose one transition while in interactive simulation all the choices are printed to the user and user chooses one transition. All the core functionality are requested from the verifier and the tool is unaware about the implementation details. This next configuration is used to repeat the whole process and the tool will stop until there is no enabled transition on the given configuration.

We can plug in any type of the verifier with the tool but the verifier has to follow the communication protocol and should provide the required functionality as explained in next section.

3.2 Contract between tool and verifier

As explained in previous section that only the verifier knows all the description about the system and the tool is unaware about the description. Thus the tool will request verifier for various functionalities and the verifier has to provide these functionalities. In order to plug in a verifier, it has to respect the contract and has to provide the functionalities mentioned in the contract. Interaction between the tool and verifier is based upon a contract. Verifier is required to have some functions to be used by the tool. Following are the functions this tool expects from the verifier.

1. getChoices(String configurationString)

This function takes a configuration in the form of string and returns a list of choices in the form of string

2. fire(String stepString)

This function fires the transition on the configuration to get next configuration. In firing, it applies all the assignments.

3. getNextConfiguration(String stepString)

This function returns next configuration. The function fires the transition on the Configuration and returns the new configuration in the form of string.

4. getInitialConfiguration()

This function returns initial configuration.

5. showStringTransitions()

This function returns all the transitions (present in the description of the system).
4 Mockup design

A mockup verifier is developed to perform the functionality of the verifier and to test the application. Developing a mockup verifier saved a lot of time as developing a new verifier can take much time. Mockup verifier knows all about the system description and has the appropriate functionality for carrying out random, interactive or guided simulation. It first parses the description of the system and then interacts with the tool using the protocol introduced in the previous section.

Figure 4.1 UML class diagram of Mockup

Figure 4.1 shows a simplified UML class diagram of the mockup verifier. The proces class has a HashMap of strings called localVariables and has a public member function andProces(p1, p2). The configuration class has a list of Proces objects and a HashMap of strings used to store sharedVariables. Proces and Configuration class have composition relationship means Configuration contains proces. Transition class has list of guards and assignments. It has two important functions, enabled() and fire() which are used in random, interactive and guided simulation. Enabled function will take configuration as a parameter and will return all the choices (Choice is an index number on which the transition is enabled). Fire function will fire choice on the respective configuration and will return list of configurations.

Guard

Guard is an abstract class having two subclasses EqualGuard and DifferenceGuard as shown in figure 4.2. It has the functions enabled_local(int index, Configuration configuration) that checks if guard of local transition is enabled on a particular index in the provided configuration, enabled_exists(int index, Configuration configuration, Range range) will only deal with other modifiers, show() to print the guard. For local and shared variables, enabled_local function in Transition class is used. Both the subclasses override these
functions to perform their required functionalities. In future, if we need another type of guard, we just need to add a subclass from the abstract class Guard.

![Diagram of Guard class hierarchy]

**Figure 4.2 Abstract class Guard**

**Assignment**

Assignment is an abstract class having only one subclass called SimpleAssignment as shown in figure 4.3. This abstract class have functions apply(int index, Configuration configuration, Range range) to apply assignment on the provided process index in the configuration and show() to print the assignment. Adding new type of assignment is simple by adding a subclass to this abstract class.

![Diagram of Assignment class hierarchy]

**Figure 4.3 Abstract class Assignment**

**Choice**

Choice is an abstract class having three sub classes LocalChoice, ExistsChoice and Forall Choice used for Local, Existential and Universal transitions respectively as shown in figure 4.4. It has print function overridden in all subclasses. These classes have index/indices on which the transition will be fired or assignments will be applied. It is easy to add a new choice by adding a new subclass for the abstract class Choice.
Abstract class Choice

Diagram:

```
+-----------------+
|                  |
|   Choice        |
|                  |
+-----------------+
    |              |
| LocalChoice     |
| ExistsChoice    |
| ForallChoice    |
```

Figure 4.4 Abstract class Choice

Transition

Transition is an abstract class having three subclasses LocalTransition, ExistsTransition and ForallTransition as shown in figure 4.5. It has functions enabled(Configuration c) that checks set of guards on configuration and returns an array of choices that satisfies set of guards, fire(Configuration c, Choice choice) that fires transition on a particular choice in the given configuration and showTransition() function to print the transition. Three subclasses have set of guards and assignments which are used in enabled and fire functions. Adding a new type of transitions is easy by adding a new subclass to the abstract class Transition.

Diagram:

```
+-----------------+
|                  |
|   Transition    |
|                  |
+-----------------+
    |              |
| LocalTransition |
| ExistsTransition|
| ForallTransition|
```

Figure 4.5 Abstract class Transition

A local transition can only deal with current process’s local variables as well as shared variables. This type of transition cannot change the states of other variables present in the configuration. An existential transition can deal with current process’s local variables, other processes and shared variables. This type of transition can affect current process’s states as well as another single process’s states in the configuration. A universal transition can deal with current process’s local variables, all other processes as well as shared variables. Universal transition, when applied, can change current process’s states, shared variables’ states and all the other processes in the current configuration.
5 Development environment and tools

In this project, Java was used as the development language and it is natural to use eclipse for java development. There are various parser generator tools like ANTLR, bison and Yaac. We have chosen ANTLR as parser generator because it supports java as the target language.

5.1 IDE language environment

The primary reason behind choosing java as programming language is that it is portable and platform independent and a natural choice to develop a Graphical User Interface (GUI), an important part of this project. Eclipse IDE was used while developing framework for interacting with parameterized and infinite state verification tools. Eclipse IDE is a rich development tool with plenty of good features that distinguishes it from other available tools. Some of the features include code refactoring/renaming, automatic comments generation using java doc and code hints. One of the appealing features of Eclipse IDE is plugins so that you can install required plugin to extend the functionality. Two plugins are used in this project, first one is for ANTLR (parser generator) and second one is for GUI development.

5.2 ANTLR

ANTLR (Another Tool for Language Recognition) is used to parse the input from the user. ANTLR is a parser generator to read, process and execute structured input [11]. Many software tools are using ANLR for building compilers, Domain Specific Languages (DSL) and language interpreters. A grammar file is used to generate AST (Abstract Syntax Tree) and validate the input. ANTLR automatically generates Lexer and Parser for this grammar file. Tree grammar is used to process the input after validation and ANTLR generates respect target language file for that tree grammar. Java is used as target language and it has generated a java file for that tree grammar. ANTLRWorks is a graphical tool for ANTLR that is quite helpful while building parsers using ANTLR [12]. It is also helpful in pointing out the grammar errors. It shows you syntax diagram of the grammar you have written so that you can have better understanding as well as control over the flow of the grammar. In ANTLRWorks you can modify the grammar at run time and can validate it by giving input. You can also view AST at run time. ANTLRWorks 2 (the latest version) is equipped with grammar editor and interpreter to test your grammar on the go. Moreover, it is also equipped with debugger that helps in identifying grammar errors, thus saving a lot of time.

5.3 Documentation development

Code documentation plays a key role during maintenance or future development. It is important to properly document code so that programmers working on the same projects in future can easily understand it. The code is properly documented and each java file has an overview about its functionality. Each function is documented to understand its flow and description about the parameters and return type of the function. Documentation for this tool is also available in HTML form so that user can easily go through all the classes and all the functions within a class. HTML documentation for the code is generated using javadoc that empowers us to create user friendly manual for the whole application code.
6 Results

In this chapter we will run the application using a simple example. During this process, snapshots are captured in order to present a typical use of the tool.

6.1 Screenshots

A small example will be used to test drive our framework for interacting with parameterized and infinite state verification tools. First we will give a quick overview about the tool by explaining various control buttons then we will use a description of a simple parameterized system to perform various types of simulations (Random, interactive and guided) and to carry out verification. Later while simulating the simple example and during snapshots session, we will also explain the input this tool will be expecting from the user and how it will behave when undesired input is given.

6.1.1 Random Simulation

In random simulation, simulation choices are randomly selected using a random number generator. Random simulator will first plug in a verifier (in these experiments the mock up verifier) to give back an initial configuration and a transition pattern and these values are printed in the tool. We can enter the description of the system in the tool and then pressing simulate button after choosing random simulation, a popup window will appear that will ask the user about the number of iterations for the simulation. If there are total 10 iterations and you enter 2 each time then popup window will appear a total of five times. This gives you control over the simulation by speeding up or slowing down the simulation.

![Main interface of the application when launched](image)

*Figure 6.1 Main interface of the application when launched*
Figure 6.1 above shows the main interface of the application when it is launched. The Graphical User Interface is quite simple. Basically, the interface is divided into three main parts.

1. Control panel area on the very top and is subdivided into two parts
   a. Code editor buttons (Open, ReOpen, Save, Save As, Find, Clear)
   b. Simulate and verify buttons (Simulate, Save Trail, Verify)
2. Input area on the bottom left (White background)
3. Output area on the bottom right (Grey background)

![Figure 6.2 ReOpen button error dialog box](image)

So it’s time to start using this tool. In order to operate the tool user must provide the description of the system in order to simulate or verify. User can input the description of the system in two possible ways.

1. Write the description in the input text area (White color on the bottom left)
2. Load the file with the description stored in it.

In order to load description from the file, user will have to click the ‘Open’ button and a dialog box will open to select the file, as shown in figure 6.3
Once user had chosen the file, it will be loaded in the input panel as shown in figure 6.4. Here user can perform basic text editing like Find a string and user can also save the modified description.
After loading the file with description, type a word ‘transitions’ and press find button. The tool finds all the occurrences of the specified word and will highlight these in red color as shown in figure 6.5 above.

Figure 6.6 shows you the complete contents of simple grammar file accepted by the verifier (in this case the mock up verifier) that is loaded into the tool. The syntax is quite simple and is as follow

1. Define domains
   Domains are defined by typing keyword ‘domains’ followed by colon symbol once.

2. Define and initialize local variables
   Local variables are defined and initialized by typing keyword ‘locals’ followed by colon symbol once.

3. Define and initialize shared variables
   Shared variables are defined and initialized by typing keyword ‘shared’ followed by colon symbol once.

4. Define the number of processes to be in the configuration
   Number of processes are defined by typing keyword ‘processes’ followed by colon symbol and then typing ‘processes =’ followed by desired number of processes.

5. Define the transitions
   Transitions are defined by typing keyword ‘transitions’ followed by colon symbol once. This tool supports three types of transitions, local, existential and universal. All transitions end with a semicolon symbol. For existential, existential left, existential right and universal type of transitions, user has to type ‘exists:’, ‘exists left:’, ‘exists right:’, ‘forall:’ in the beginning of the transition.
domains:
isLocked [locked, unlocked]
possibleStates [0..7]

locals:
resource in isLocked init locked

shared:
pc in possibleStates init 1

processes:
processes = 4

transitions:
this.resource=locked this.resource:=unlocked;
this.resource=locked shared.pc:=1 shared.pc:=2 this.resource:=unlocked;
exists: this.resource=locked other.resource=locked this.resource:=unlocked
other.resource:=unlocked;

Now it is easy to understand that how description of the grammar works and what syntax should be used while writing the description. For the sake of simplicity and for demonstration purposes, existential left, existential right and universal type of transitions are not used in this example but you can write a simple description to test these types of transitions.

As the description is loaded in the tool, we can perform simulation and verification. Moreover, we can also store the result of the simulation. We have to click on radio button ‘Random’ and then click the simulate button. A popup dialog box will appear as shown in figure 6.7. The dialog box asks you about the number of steps you want to randomly simulate. Enter 1 and then pressed the ‘Nexts’ button.
As the ‘Nexts’ button is pressed in the popup dialog box, the tool will perform a single iteration, will fire the transition on the configuration and will show you results for a single step as shown in figure 7.8. This single step result can be seen under ‘Simulation Results’ in the output window. Now let’s analyze the output. You can see the output for single step in Figure 6.9
This output tells you that Transition #1 is fired on index number zero in the configuration given at the end. Please note that in the configuration all the states before colon symbol(:) are shared and all the states after colon symbol(:) are local.

In the popup dialog box, you can specify the number of steps to be performed for random simulation. You can see the simulation results in figure 6.10 once random simulation is completed. Random simulation chooses one step randomly among all the possible steps. Interactive simulation gives user some freedom and user can choose one step among all the calculated steps.

**6.1.2 Interactive Simulation**

In interactive simulation, user is asked to choose a simulation choice. Interactive simulation gives more control to user during simulation. During interactive simulation, first initial configuration and transitions are printed and these values are acquired from the verifier. Simulation choices are also obtained from the verifier and are printed to the user in a popup dialog box. Simulation choices are list of Step objects and each object contains a configuration, an enabled transition and a choice object. After the user has selected a choice, the transition is fired on the configuration by using the choice object. Thus, we get the next configuration and the process is repeated until there is no transition enabled.
button Interactive and then click simulate button. A popup dialog box will be opened as shown in figure 6.11. Popup dialog box for interactive simulation will print you all the choices for steps. Each step has a unique identification number printed in the beginning enclosed in square brackets. User has to choose one step by entering desired identification number in the given range. As shown in figure 6.11 identification number zero (0) is entered by user.

![Interactive simulation](image)

**Figure 6.11 Interactive simulation**

After entering step identification number (choosing the step) user has to press the simulate button and results will start showing in output windows below ‘Simulation Results’.
When there is no enabled transition, no choice will be printed in the interactive simulator dialog box.
A confirmation dialog box will be opened when interactive simulation will be completed as shown in figure 6.14.

If user enters a number that doesn’t belong to the printed indices in the interactive or random simulation popup dialog box, an input error message dialog box will be opened stating that only numeric values are allowed (Enter index enclosed in []) as shown in figure 6.15.

![Interactive simulation string input error handling](image)

If user enters a string instead of a valid (desired) integer, user will also get an input error dialog box as shown in figure 6.15 above.

Random and interactive simulations are somehow same besides the fact that user has some freedom in choosing the enabled transitions in interactive simulation. Guided simulation is bit different from both where user has to provide a trail (a path) upon which guided simulation will be carried out. This tool also facilitates the use to save trail when random or interactive simulation is completed. User has to press ‘Save Trail’ button and a dialog box will be opened asking the write the name with which user would like to save the trail as shown in figure 6.18.

6.1.3 Guided Simulation

Guided Simulation requires a trail to carry out simulation. Whenever you perform random or interactive simulation, you can save the trail by pressing ‘Save Trail’ button and this will store simulation results in a file. This trail file can be used later to perform guided simulation. SimulatorGuided class is responsible to perform guided simulation. This class will first get initial configuration and transitions from the verifier and will print them in the tool. An example of a very small trail file can be seen below in figure 6.16.
Each row has a record containing the transition number, choice and the configuration. This trail file has only two records. The verifier will first parse this trail file and then for each record it will fire transition on the configuration using the choice and will match the resulting configuration. Like interactive simulation, you can also control the speed of guided simulation by entering iterations in a popup dialog box that appears when you click simulate button after choosing guided simulation.

Now we can perform guided simulation once a trail is stored in a file. Click the radio button ‘Guided with trail:’ and then click ‘Browse’ button. A dialog box will be opened that will ask you to enter the name of the trail file. In this example, trail file has the name ‘simpleAlgo_trace.txt’ as shown in figure 6.16. User can also select trail file from another directory.

| Transition#1, 0, [<condition=false, pc=1> : <flag=up, door=unlocked> <flag=down, door=locked>] |
| Transition#2, 1, [<condition=true, pc=7> : <flag=up, door=unlocked> <flag=down, door=locked>] |

**Figure 6.16 A simple trail file**

**Figure 6.17 Save Trail, saving the simulation results**
Figure 6.18 Guided simulation browse button

Figure 6.19 Guided simulation Simulate button clicked
After selecting the trail file, click on ‘Simulate’ button. Name of the trail file will be printed in the text box in front of radio button ‘Guided with trail:’ and a dialog box will opened automatically asking number of steps from the user as shown in figure 6.19. Type 1 in the steps text field and press ‘Nexts’ button, one result will be printed in the output panel below ‘Simulation Results’.

Figure 6.20 Guided simulation popup dialog box

Figure 6.21 Guided simulation completed
Results are shown in a step wise manner depending upon the number of steps selected by user in the output panel. Figure 6.20 shows the simulation results in output windows when guided simulation is completed.

6.1.4 Verification
User can also do verification by using this tool. After writing the description of the system in the input panel or feeding the description through a file, user can press ‘Verify’ button to perform verification process. ‘Property holds’ message will be printed in the output window if the if the property holds otherwise guided simulation will be started, radio button guided will be selected automatically and a dialog box will be opened asking user to input number of steps as shown in figure 6.21. After this all the steps will be same as in guided simulation.

![Figure 6.22 Verify button to perform verification](image)
7 Future work

One direction of future work for this project is to develop a language for printing configurations. Currently, we are printing configurations in simple textual form. One can develop such a language that updated states in the configuration during transition are highlighted. It would be great to enhance that language in such way that transitions can be visually seen using arrows and ovals in a graphical manner. This will give user a visual interface to understand clearly that what is happening during transition or during whole simulation. Moreover, one can add the possibility of having arrays in the verifier as currently this tool only supports variables.
8 Bibliography

Copyright Information

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© [Författarens för- och efternamn]