

GUI for model checkers

by

Bo Wang

THESIS

MASTER OF SCIENCE

Department of Computer Science

Faculty of EEMCS

Delft University of Technology

June, 2006

Colophon

Author: Bo Wang

Student id: 1235931

E-mail: wangbo_hl@hotmail.com

Title

GUI for model checkers

Graduation Committee

Chair: Prof. Dr. A. van Deursen Faculty EEMCS, TU Delft

University supervisor: Ir. C. (Kees) Pronk Faculty EEMCS, TU Delft

Committee member: Drs. J.W.J.Heijnsdijk Faculty EEMCS, TU Delft

Abstract

System validation, the process of checking the correctness of specifications, designs and products, is an important technique to guarantee the safety and quality of a system. There are several validation techniques, such as peer reviewing, simulation, formal verification, and model checking, etc.

Given a finite-state model of a system and a property stated in some appropriate logical formalism, model checking technique can systematically check the validity of this property. Nowadays, model checking is becoming a general and popular approach and is applied in areas like hardware verification and software engineering. Due to its success in several projects and the high degree of support from tools, there are many model checkers available now. From the view of graphical user interface (GUI), some model checkers are primitive, such as TLC [15] which is used to validate a system written in TLA+ specifications [1], which are highly abstract and give the user a complete view of the system; and some do have a good graphical user interface, such as UPPAAL [16] which lets the user validate a system through a GUI, which provides the user more convenience and gives the user a more direct and clearer view of the system, especially in the process of simulation it can help the user understand how the system runs. Another model checker that has a GUI is Xspin [17] which is the graphical interface of Spin [18], a popular open source model checker, which can be used for the formal verification of distributed software systems. Xspin runs independently from Spin and helps by generating the proper Spin commands based on menu selection.

Is it possible to develop a GUI for the TLC model checker? How to design this GUI? What problems will be encountered? The goal of the literature survey is to find answers to these questions.

The purpose of the final part of the project is to develop the graphical interface of the TLC model checker (the GTLA system) according to the design from the literature survey. The TLA+ user can use the GTLA system to validate a system written in TLA+ specifications through a graphical user interface.

This report consists of two parts. The first part is the literature survey, which will discuss the concepts of model checking and introduce three model checkers: TLC, UPPAAL and Xspin; additionally, if it is good to make specifications executable and how to animate specifications will be discussed; finally, a case study, GUI design for the TLC model checker will be presented. The second part is the final thesis, which will introduce how the GTLA system is designed and implemented; the testing result, conclusion and future work will be given at last.

This report has been written as the results of the literature survey and the final thesis, two parts of the graduation at Delft Computer Science, Delft University of Technology.

Acknowledgement

My foremost thank goes to my thesis supervisor C. (Kees) Pronk. Without him, this dissertation would not have been possible. I thank him for his patience and encouragement that carried me on through difficult times, and for his insights and suggestions that helped to shape my research skills. His valuable feedback contributed greatly to this dissertation.

I would like to thank Prof. Dr. A. van Deursen and Drs. J.W.J.Heijnsdijk for their suggestions on various aspects through this work.

I thank all the students and staffs in the Computer Science department, who taught me knowledge and helped me with my study and life in TU Delft. Especially I thank several good friends of mine, who always encouraged and supported me. They are Li Bei, Yuan Xu, Zhang Sheng, Zheng Renliang, Zhu Yetao, Maarten Wit.

Last but not least, I thank my grandmother, my parents and all other family members for always being there when I needed them most, and for supporting me through all these years.

Bo Wang

Delft, The Netherlands

June 2006

CONTENTS

ABSTRACT	I
ACKNOWLEDGEMENT	II
CONTENTS	III
CHAPTER 1 INTRODUCTION	1
1.1 PROBLEM STATEMENT	1
1.2 OUTLINE OF THIS THESIS	2
PART I LITERATURE SURVEY	4
CHAPTER 2 MODEL CHECKING	5
2.1 SYSTEMS AND THEIR SPECIFICATIONS	5
2.2 MODEL CHECKING	6
2.3 MODEL CHECKERS	10
2.3.1 <i>The TLC model checker</i>	10
2.3.2 <i>The UPPAAL model checker</i>	14
2.3.3 <i>The Xspin model checker</i>	17
2.3.4 <i>Comparison of TLC, UPPAAL and Xspin</i>	21
CHAPTER 3 MAKING SPECIFICATIONS EXECUTABLE	22
3.1 SPECIFICATIONS ARE HARMFULLY EXECUTABLE	22
3.2 SPECIFICATIONS ARE PREFERABLY EXECUTABLE	23
3.3 ANIMATING FORMAL SPECIFICATIONS	23
CHAPTER 4 GUI DESIGN FOR TLC	25
4.1 TLA+ SPECIFICATIONS	25
4.2 ANIMATING TLA+ SPECIFICATIONS	26
4.2.1 <i>Translating TLA+ language into a target language</i>	26
4.2.2 <i>Animator design</i>	33

4.3 GUI DESIGN.....	37
4.4 PROJECT DEVELOPMENT SCHEDULE	39
CHAPTER 5 CONCLUSIONS OF LITERATURE SURVEY	41
PART II FINAL THESIS	42
CHAPTER 6 TRANSLATING TLA+ INTO SML	43
6.1 INTRODUCTION	43
6.1.1 <i>The TLA+ Language</i>	43
6.1.2 <i>The SML Language</i>	44
6.1.3 <i>Overview of the GTLA Translator's design</i>	45
6.2 LEXICAL ANALYSIS	46
6.3 SYNTAX ANALYSIS	48
6.4 CODE GENERATION	50
6.4.1 <i>Declarations</i>	50
6.4.2 <i>Expressions</i>	56
6.4.3 <i>AST Traverse</i>	61
6.4.4 <i>Types</i>	62
6.5 EXAMPLES	64
6.6 IMPROVED SML FORMS	66
6.7 CONCLUSION	68
CHAPTER 7 DESIGNING THE GTLA SYSTEM	70
7.1 OVERVIEW OF THE GTLA SYSTEM.....	70
7.2 DESIGNING THE GUI.....	72
7.2.1 <i>The Menu and Tabbed Folder</i>	72
7.2.2 <i>The System Editor</i>	73
7.2.3 <i>The Simulator</i>	75
7.2.4 <i>The Verifier</i>	77
7.2.5 <i>The Development Tool</i>	78

7.3 CONCLUSION	79
CHAPTER 8 DEVELOPING THE GUI	80
8.1 INTRODUCTION	80
8.2 THE MENU AND TABBED FOLDER	80
8.3 THE SYSTEM EDITOR	84
8.4 THE SIMULATOR	90
8.5 THE VERIFIER	94
8.6 INTEGRATION	95
8.7 CONCLUSION	96
CHAPTER 9 DEVELOPING THE SIMULATOR.....	98
9.1 ARCHITECTURE.....	98
9.2 EXECUTABLE SML FILE.....	100
9.3 GRAPH GENERATOR.....	103
9.3.1 <i>GraphViz and Dot</i>	103
9.3.2 <i>MSC Algorithm</i>	104
9.3.3 <i>Action Graph Algorithm</i>	105
9.4 CONCLUSION	105
CHAPTER 10 TESTING	106
10.1 UNIT TESTING.....	106
10.1.1 <i>Translator Testing</i>	106
10.1.2 <i>Simulator Testing</i>	109
10.1.3 <i>GUI Testing</i>	110
10.2 INTEGRATION TESTING	112
10.3 CONCLUSION	113
CHAPTER 11 CONCLUSION AND FUTURE WORK	115
11.1 PROJECT CONCLUSION.....	115
11.2 PERSONAL CONCLUSION	116

11.3 FUTURE WORK	117
BIBLIOGRAPHY	118
APPENDIX A GTLA USER’S GUIDE	120
APPENDIX B LEXICAL CONVENTIONS	132
APPENDIX C GTLA GRAMMAR.....	137
APPENDIX D TESTING EXAMPLES	146
APPENDIX E TESTING EXAMPLES 2	148

Chapter 1

Introduction

1.1 Problem statement

A system specification is a description of what a system is supposed to do. It helps us understand the system. To guarantee the safety and quality of a system, validation techniques are used to check the correctness of system specifications. Model checking is a general and popular kind of validation technique. Given a finite state model of a system and a property stated in some appropriate logical formalism, model checking techniques could systematically check the validity of this property.

There are various model checkers applied in various areas. TLC is a famous model checker used to find errors in Lamport's TLA+ specifications, which provide a mathematical foundation for describing systems. Although TLC is powerful for finding errors in TLA+ specifications, it is rather primitive with respect to the user interface. Some model checkers have a good graphical user interface, which provides the user more convenience and makes the user easier to locate errors by running a simulator. Like UPPAAL, a model checker that is used especially for real-time system specifications; and Xspin, a model checker that is used for distributed software systems.

To give users immediate feedback of the behavior of the future software, it was suggested to make specifications executable. However, Hayes and Jones argue that "executable specifications should be avoided because executability can restrict the expressiveness of specification languages and can adversely affect implementations" [3]. At the meantime, some others argue that non-executable formal specifications can be made executable on almost the same level of abstraction and without essentially changing their structure. Can non-executable TLA+ specifications be executable?

In this research project the graphical user interface (GUI) for model checkers will be investigated. The attention will be focused on the TLC model checker. A proper GUI design for the TLC model checker will be developed and obtained results will be validated in practice.

According to the results of the literature survey, the implementation project will develop a system (the GTLA system) that provides system editing, simulation and verification mechanisms to the TLA+ user. The user can use the GTLA system to validate a system written in TLA+ specifications through a graphical user interface.

The GTLA system will provide the user many benefits:

- The most important benefit is that the GTLA system provides the TLA+ user a GUI. This provides the TLA+ user more convenience with operations on the

validation of TLA+ specifications.

- The GTLA system provides the user a system editor. The user can both edit TLA+ specifications and verify them through the GUI. This provides the user more convenience.
- The GTLA simulator displays the dynamical simulation process with information about the current value of variables, enabled actions, action trace, and two graphical views of the simulation process to the user.
- The GTLA system provides the user much other functionality, such as syntax checking, verification (model checking with TLC), etc.

The GTLA system has mainly four parts which need to be constructed: a GUI, a system editor, a simulator and a verifier. Each part has its own requirements and is designed and implemented separately. After that they are integrated into the GTLA environment.

In this thesis, the literature survey, the requirements, design, implementation, testing, refinement, conclusion and future work of the GTLA system will be presented.

1.2 Outline of this thesis

This report mainly consists of two parts. Chapter 2, 3, 4, 5 are the literature survey. Chapter 6, 7, 8, 9, 10, 11 are the final thesis.

In chapter 2 the concept of systems and their specifications and the basic theory of model checking technique will be presented. Three model checkers will be discussed: TLC model checker for TLA+ specifications, UPPAAL model checker for real time systems, and Xspin model checker for distributed systems.

The discussion of the subject of making non-executable specifications executable can be found in chapter 3. Two different opinions will be presented and discussed. Then the animating approaches which are used to make non-executable specifications executable will be introduced.

Chapter 4 will state some GUI design plans for the TLC model checker, including making TLA+ specifications executable and simulator design. The related development tools will be discussed too.

Chapter 5 will summarize the possibilities and problems of developing the GUI for the TLC model checker. It will conclude with recommendations for further research work.

In chapter 6 the design and implementation of translating TLA+ into SML will be presented. This translation makes the TLA+ specifications executable. Some examples will be given to describe the translation process. Finally the improvement work and conclusion of this chapter will be presented.

Chapter 7 will discuss the design of the whole GTLA system. The requirements of each part of the GTLA system will be stated. The development tool will be introduced and a short conclusion of this chapter will be given.

Chapter 8 and chapter 9 will discuss the implementation of the GUI and Simulator parts of the GTLA system respectively. A conclusion of each chapter will be presented finally.

Chapter 10 will discuss the testing of the GUI, the simulator and the integrated GTLA system. Methods, results, refinements will be presented.

Chapter 11 will summarize the construction work of the GTLA system. It will conclude with recommendations for further construction work.

Finally, the last part of this thesis will provide the annexes, including user guide, lexical conventions, grammar, testing examples, etc.

Part I

Literature Survey

by

Bo Wang

RESEARCH

MASTER OF SCIENCE

Department of Computer Science

Faculty of EEMCS

Delft University of Technology

January, 2006

Chapter 2

Model checking

2.1 Systems and their specifications

A specification is a description of what a system is supposed to do. Specifying a system helps us understand it and find errors, so writing a specification of a system before implementation is a good idea to think clearly about the design and improve it. The properties of a system to be specified could be its functional properties that specify what the system is supposed to do or its non-functional properties, such as performance properties. The specification is traditionally written in natural language, augmented as necessary with tables, diagrams, etc, but today more and more specifications are written in formal specification languages. Compared with natural languages, formal specification languages have a well-defined syntax and semantics and they are possible to be verified and validated with respect to the requirements. Additionally, the specification should be as abstract as the requirements permit, thus it requires the specification language to be sufficiently powerful to express the required behavior adequately.

The System specification, or functional specification, is generally written in a highly abstract manner, constraining the implementation as little as possible. The objective is to develop an explicit model of the system that is clear, precise and as unambiguous as possible, which is hard to be achieved by a natural language. To achieve this goal mathematics is used to write specifications as the basic tool. But mathematics written by most mathematicians is still not precise enough. Although each equation is a precise assertion, people have to read the accompanying words to understand the relationship between those equations and the meaning of the theorems. Fortunately, logicians have developed ways of eliminate those words, thus we could get precise and completely formal mathematics. Temporal logic is such a way that can be used for describing system behaviors.

Temporal logic assumes an underlying logic for expressing mathematics. It supports the formulation of properties of system behaviors over time. Different interpretations of temporal logics exist depending on how one considers the system to change with time. There are different types of temporal logic. Linear temporal logic and branching temporal logic are two kinds of temporal logic used to specify reactive systems, which are characterized by a continuous interaction with the environment. Real-time (branching) temporal logic is another kind of temporal logic used to specify time-critical systems, which are characterized by quantitative timing properties relating occurrences of events. Linear temporal logic allows the statement of properties of execution sequences of a system, while branching temporal logic allows the user to write formulas which include some sensitivity to the choices available to a system during its execution. It also allows the statement of properties of about possible execution sequences that start in a state. Real-time temporal logic allows

statement of properties of multiple concurrent processes and supports relative time references.

Specification is written to help avoid errors. System validation techniques are used to check the correctness of the specifications, that is means the systems.

2.2 Model checking

Model checking is a technique for system validation. To build a correct system, a design process and a validation process are needed. The validation process is to check if the design can satisfy all the requirements, see Figure 2.1.

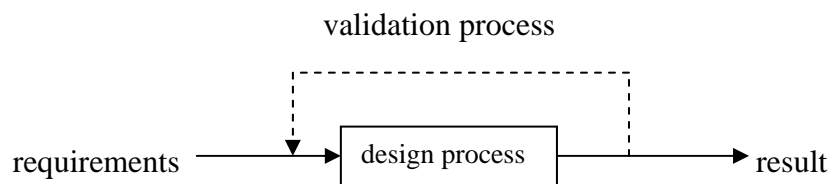


Figure 2.1: System validation.

System validation is a very important activity to determine the correctness of specifications, designs and products. There are two kinds of validation techniques: the informal ones are testing and peer reviewing, the formal ones are formal verification, simulation and model checking. Informal techniques are based upon “some” understanding of the system under test. Among them, testing is an operational way to check whether a given system realization conforms to an abstract specification and could be only applied after a prototype implementation of the system has been realized. As opposed to testing, formal verification works on models and amounts to a mathematical proof of the correctness of a system. This report will focus on a formal validation technique -- model-checking.

Given a finite-state model of a system and properties stated in some appropriate logical formalism (such as temporal logic), model checking technique uses algorithms, executed by computer tools, to systematically check the validity of these properties, that is if the model satisfies these properties. Two kinds of files should be input into the model checker: a description of the model of the real system and a description of the properties to be checked. After that the model checker does the verification. If an error is found the model checker provides a counter-example showing under which circumstances the error can be generated, otherwise the user can refine its model description. With the counter-example the user could locate the error and repair the model specification. See Figure 2.2.

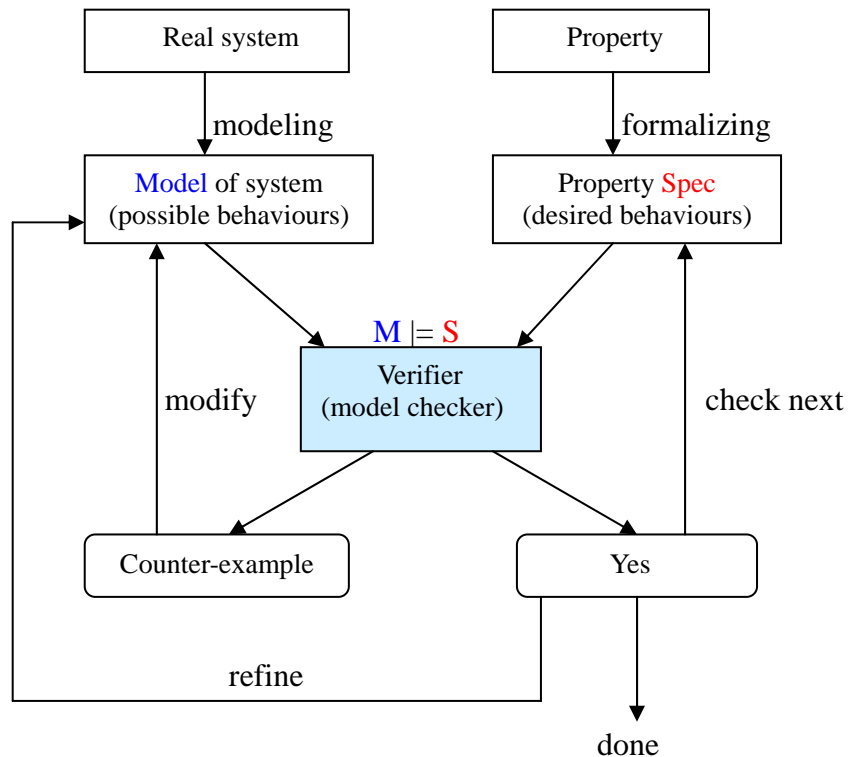


Figure 2.2: Verification methodology of model checking.

“Properties to be specified”

The properties to be checked here is behavior properties, such as liveness, fairness, safeness, deadlock, state properties, time requirement, etc.

- Liveness properties are those properties can't be violated at any particular instant, such as that one can't tell that the clock has stopped ticking before examining an entire infinite behavior.
- Fairness properties are those properties assert that a step must eventually occur if this step is continuously or continually enabled.
- Safeness properties are requirements that the system never do anything, such as the clock need never tick.
- Deadlock property is to check if the system is deadlock-free.
- State properties are used when model checking is about state machine and they are properties to examine if the model can (never) get some state.
- Time requirement is used in a real-time system and it is one property which specifies time requirements of the system.

“Model checking algorithm”

The algorithm for model checking is typically based on an exhaustive state space search of the system model: for each state of the model, it checks whether the state satisfies the desired properties. The most important property is known as reachability: to determine whether a system can reach a state in which no further progress is possible (deadlock), to determine all reachable states, etc. Reachability analysis technique is used to check reachability property. Automated reachability analysis starts from the initial system state and determines all system states that can be reached. From this point of view model checking is about state machines. For example, given a traffic light model, there are three states “Yellow”, “Green” and “Red”, the idea of model checking is to prove if the model can get some state, if the model can get all states eventually, or if the model could never get some state.

“Temporal logics and model checking techniques”

Logics extended by operators that allow the expression of properties about executions, in particular those that can express properties about the relative order between events, are called temporal logics. Temporal logic is a well-accepted and commonly used specification technique for expressing properties of computations (of reactive systems) at a rather high level of abstraction. According to different temporal logics there are three types of model checking techniques: model checking of linear, branching, and real-time (branching) temporal logic. The first two types concentrate on the functional or qualitative aspects of systems, whereas the latter type of model checking allows in addition some quantitative analysis.

- Linear temporal logic allows statements of properties of execution sequences of a system. For example, P (LTL) is a kind of linear temporal logic, which is used by the SPIN model checker.
- Branching temporal logic allows the user to write formulas which include some sensitivity to the choices available to a system during its execution. It allows statements of properties of about possible execution sequences that start in a state. CTL is a kind of branching temporal logic, which is used by the SMV model checker.
- Real-time temporal logic is extended with a quantitative notion of time. The former two temporal logics concentrate on the notion of states while the real-time temporal logic concentrates on the notion of time. Timed CTL is a kind of real-time temporal logic, which is used by the UPPAAL model checker.

“Approaches of model checking”

There are basically two methods of model checking: logic-based or heterogeneous approach and behavior-based or homogeneous approach.

- In logic-based approach a system is modeled as a finite-state automaton, where states represent the values of variables and control locations, and transitions

represent how a system changes from one state to another. Given a set of initial states, if the system satisfies the properties (the desired system behavior) which are given in some logic, the system is regarded to be correct.

- In behavior-based approach both the desired and the possible system behavior are given in the same notation, such as an automaton, the equivalence relations are used as a correctness criterion. A system is considered to be correct if the desired and the possible system behavior are equivalent with the equivalence attribute under investigation.

This research work focus entirely on the logic-based approach.

“Benefits of model checking”

There are many benefits of model checking:

- It can be applied to many areas, such as hardware verification, software engineering, and so forth.
- It supports partial verification. The user can consider only a subset of system requirements. This can result in improved efficiency.
- Many case studies show that using model checking may lead to shorter times of verification than using simulation and testing. Additionally, it can also deal with rather large state spaces.

“Limitations of model checking”

The limitations of model checking are:

- It is not suited to those systems that have no communications between components or those that introduce infinite state spaces.
- It only verifies the model of a system but not the real system itself, so the fact that a model possesses some properties doesn't guarantee that the implementation of the system possesses the same properties.
- It checks only stated properties, no guarantee of other properties.
- It is impossible to check generalizations since it only focuses on finite state space.
- It needs some expertise to find appropriate abstraction.
- As any other tool, model checking software might be unreliable.

2.3 Model checkers

Model checkers are tools using the model checking technique to verify a system. Due to the success of model checking in many case studies, industry became building their own model checkers. There are various model checkers available now. The model checker SPIN is a representative model checker used for model checking of linear temporal logic. It allows the simulation of a specification written in the language PROMELA and the verification of several types of properties, such as verification of state properties, unreachable code, and so on. The model checker SMV is a representative model checker used for model checking of branching temporal logic. It is very useful for verifying hardware circuits and communication protocols. The model checker UPPAAL is a representative model checker used for model checking of real-time temporal logic. Besides model checking, it also supports simulation of timed automata and has some facilities to detect deadlocks.

There are many kinds of model checkers, see Figure 2.3. In this section only three model checkers will be introduced, one is TLC, which is rather primitive with respect of user interface; the other two are UPPAAL and SPIN, which do have a nice graphical user interface.

Logics	Language	Model checker	Application area
(P)LTL	Promela	SPIN	Hardware, protocols
CTL-variant	TLA+	TLC	Hardware, protocols, real-time
Timed CTL	Graphical interface	UPPAAL	Embedded real-time systems
CSP	FDR2	FDR2	Concurrent systems
?	Java->Promela	Java Path Finder	Java programs
?	?	stEAM	C++ programs/assembler-level

Figure 2.3: Overview of model checkers.

2.3.1 The TLC model checker

TLC is a program for finding errors in TLA+ specifications. TLA+ (Temporal Logic

of Actions) language is invented by Lamport. It provides a mathematical foundation for describing systems and it is quite good for specifying a wide class of systems—from program interfaces (APIs) to distributed systems. It can be used to write a precise, formal description of almost any sort of discrete system. It's especially well suited to describing asynchronous systems—that is, systems with components that do not operate in strict lock-step. The TLA tool [15] is used for verifying TLA+ specifications. It consists of three parts: syntactic analyzer, TLATEX typesetter and TLC model checker. The syntactic Analyzer is a Java program that parses a TLA+ specification and checks it for errors. It also serves as a front end for the TLC model checker. The TLATEX is a Java program for typesetting TLA+ modules. It calls the LATEX program to do the actual typesetting and can create a PostScript or PDF file. Figure 2.4 is the ASCII version of an actual TLA+ specification, the way the user type it; while Figure 2.5 is the typeset version that the TLATEX program might produce.

```

----- MODULE HourClock -----
EXTENDS Naturals
VARIABLE hr
HCini == hr \in (1 .. 12)
HCnxt == hr' = IF hr # 12 THEN hr + 1 ELSE 1
HC == HCini /\ [] [HCnxt]_hr
-----
THEOREM HC => []HCini
=====

```

Figure 2.4: A TLA+ specification for clock system.

----- MODULE <i>HourClock</i> -----
EXTENDS <i>Naturals</i> VARIABLE <i>hr</i> $HCini \triangleq hr \in (1 .. 12)$ $HCnxt \triangleq hr' = \text{IF } hr \neq 12 \text{ THEN } hr + 1 \text{ ELSE } 1$ $HC \triangleq HCini \wedge \Box [HCnxt]_{hr}$
THEOREM $HC \Rightarrow \Box HCini$

Figure 2.5: A TLA+ specification after typesetting.

The TLC model checker can check properties of a system in two ways: in model-checking mode (by default) and in simulation mode. In model-checking mode, TLC keeps two data structures: a directed graph G whose nodes are states, and a sequence U of states. A state in G means a state that is a node of the graph G . The graph G is the part of the state reachability graph that TLC has found so far, and U contains all states in G whose successors TLC has not yet computed. The computation of G terminates only if the set of reachable states is finite. Otherwise, TLC will run forever—that is, until it runs out of resources or is stopped. While in simulation mode,

TLC repeatedly constructs and checks individual behaviors of a fixed maximum length. The maximum length can be specified with the depth option. Compared to model-checking mode, after computing the set of initial states and the set T of successors for a state s , TLC randomly chooses an element of that set instead of all the elements. If the element does not satisfy the constraint, then the computation of G stops. Otherwise, TLC puts only that state in G and U , and checks the Invariant formula for it. TLC's choices are not strictly random, but are generated using a pseudo-random number generator from a randomly chosen seed. In both two modes, TLC checks a sequence of invariants at each state, including the predicates Invariant, ImpliedInit (the conjunction of every property conjunct that is a state predicate) and predicate Constraint (the conjunction of all state predicates named by CONSTRAINT statements), etc., if either is false it stops and report an error.

Figure 2.4 shows the TLA+ specification of a trivial system model -- a clock that displays only the hour with the values from 1 to 12. Besides this specification file, TLC also needs a configuration file to tell TLC the names of the specification and of the properties to be checked. For example, the configuration file for the above clock specification will contain the declaration

```
SPECIFICATION    HC
```

telling TLC to take HC as the specification. If the specification has the form $Init \wedge [[Next]_{var\ s}]$, then instead of using a SPECIFICATION statement, the user can declare the initial predicate and next-state action by putting the following two statements in the configuration file:

```
INIT    HCini
```

```
NEXT    HCnxt
```

Invariance checking is so common that TLC allows the user instead to put the following statement in the configuration file:

```
INVARIANT HCini
```

Similarly, properties to be checked are specified with a PROPERTY statement and constant parameters are instantiated with a CONSTANT statement in the configuration file, see the following examples.

```
PROPERTY    [] TypeInvariant
```

```
CONSTANT    Data = {1, 2}
```

Given the TLA+ specification and configuration file of the clock system, Figure 2.6 shows TLC's normal output, including its version number and creation date, the model in which it's being run (here is model-checking), and information about states.

```

命令提示符

STATE 2: <Action line 5, col 12 to line 5, col 46 of module HourClock>
hr = 13

24 states generated, 13 distinct states found, 1 states left on queue.
The depth of the complete state graph search is 2.

C:\TLA\tla>java tlc.TLC HourClock.tla
TLC Version 2.0 of August 18, 2005
Model-checking
Parsing file HourClock.tla
Parsing file C:\TLA\tla\tlasany\StandardModules\Naturals.tla
Semantic processing of module Naturals
Semantic processing of module HourClock
Finished computing initial states: 12 distinct states generated.
Model checking completed. No error has been found.
  Estimates of the probability that TLC did not check all reachable states
  because two distinct states had the same fingerprint:
    calculated <optimistic>: 7.806255641895632E-18
    based on the actual fingerprints: 1.6262447381494354E-18
24 states generated, 12 distinct states found, 0 states left on queue.
The depth of the complete state graph search is 1.

C:\TLA\tla>

```

Figure 2.6: TLC's normal output.

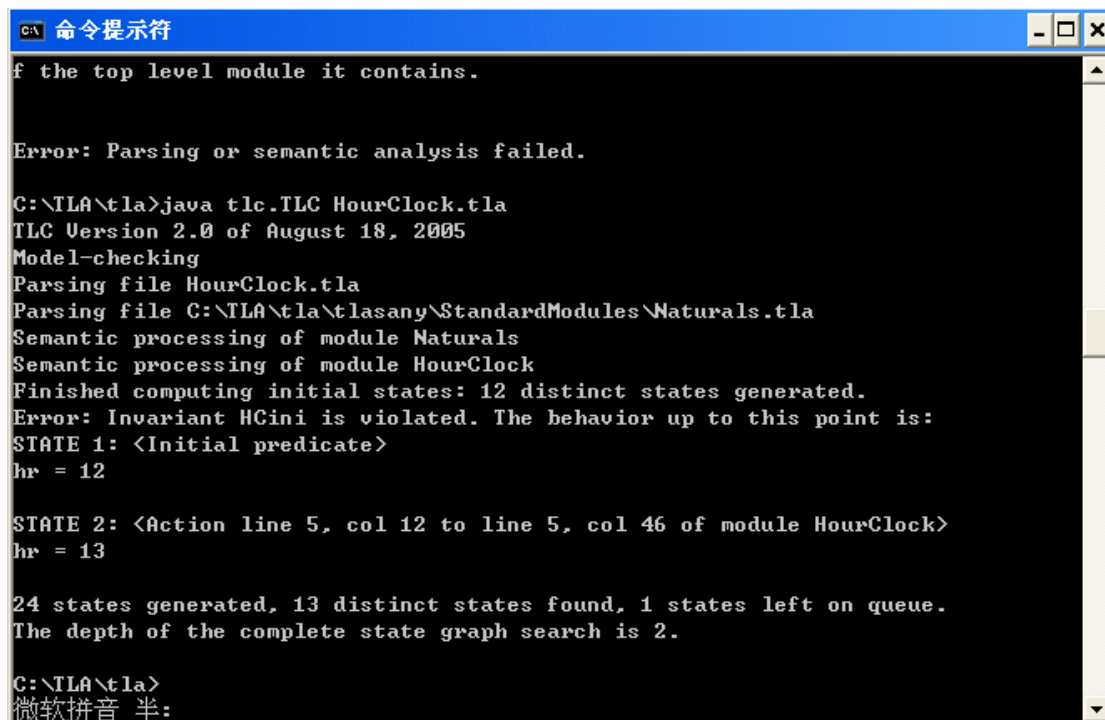
Figure 2.7 shows a wrong example of TLA+ specification for clock system discussed above. This specification only changes the guard of predicate HC_{nxt} to be hr is not equal to 13. Figure 2.8 shows its verification result generated by TLC. It gives out a counter-example that shows when invariant HC_{ini} is violated: with the initial state $hr = 12$, the next state becomes $hr = 13$. This violates invariant HC_{ini} (the value range of hr should be between 1 and 12).

```

----- MODULE HourClock -----
EXTENDS Naturals
VARIABLE hr
 $HC_{ini} \triangleq hr \in (1..12)$ 
 $HC_{nxt} \triangleq hr' = \Pi \text{ } hr \neq 13 \text{ THEN } hr + 1 \text{ ELSE } 1$ 
 $HC \triangleq HC_{ini} \wedge \Box [HC_{nxt}]_{hr}$ 
-----
THEOREM  $HC \Rightarrow \Box HC_{ini}$ 
-----

```

Figure 2.7: A wrong example of TLA+ specification for clock system.



```
f the top level module it contains.

Error: Parsing or semantic analysis failed.

C:\TLA\tla>java tlc.TLC HourClock.tla
TLC Version 2.0 of August 18, 2005
Model-checking
Parsing file HourClock.tla
Parsing file C:\TLA\tla\tlasany\StandardModules\Naturals.tla
Semantic processing of module Naturals
Semantic processing of module HourClock
Finished computing initial states: 12 distinct states generated.
Error: Invariant HCini is violated. The behavior up to this point is:
STATE 1: <Initial predicate>
hr = 12

STATE 2: <Action line 5, col 12 to line 5, col 46 of module HourClock>
hr = 13

24 states generated, 13 distinct states found, 1 states left on queue.
The depth of the complete state graph search is 2.

C:\TLA\tla>
微软拼音 半:
```

Figure 2.8: TLC error report.

All in all, TLC model checker focuses on TLA+ specifications. It can find errors and provide counter-examples for TLA+ specifications. But TLC model checker is rather primitive with respect to user interface and it lacks communications with the user. A graphical user interface will make it better.

2.3.2 The UPPAAL model checker

The model checker UPPAAL is a toolbox for verification of real-time systems. It is designed to verify systems that can be modeled as networks of timed automata extended with integer variables, structured data types, and channel synchronization. It has been applied successfully in case studies ranging from communication protocols to multimedia applications.

In UPPAAL, a system is modeled as a network of several timed automata in parallel, which is a finite-state machine extended with clock variables. A state of the system is defined by the locations of all automata, the clock constraints, and the values of the discrete variables. Every automaton may fire an edge separately or synchronize with another automaton, which leads to a new state.

UPPAAL contains two parts: a graphical user interface and a model checking engine. The GUI is divided into three main parts: the editor, the simulator, and the verifier. Let's investigate UPPAAL thoroughly with the following example.

Figure 2.9 (a) shows a timed automaton modeling a simple lamp. The lamp has three locations: off, low, and bright. If the user presses a button, synchronizes with `press?`, then the lamp is turned on. If the user presses the button again, the lamp is turned off. However, if the user is fast and rapidly presses the button twice, the lamp is turned on and becomes bright. The user model is shown in (b). The user can press the button randomly at any time or even not press the button at all. The clock `y` of the lamp is used to detect if the user was fast ($y < 5$) or slow ($y \geq 5$).

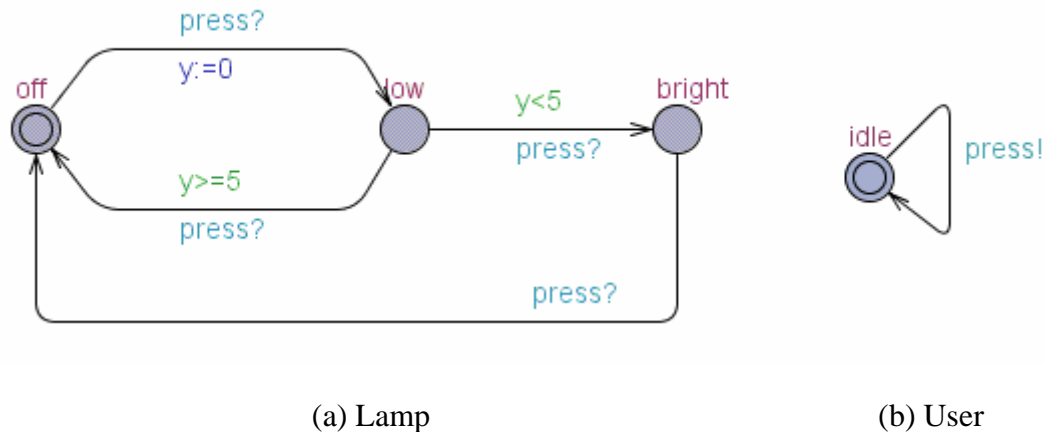


Figure 2.9: The simple lamp example.

A system is defined as a network of timed automata, called processes in the tool, put in parallel. A process is instantiated from a parameterized template. The editor is divided into two parts: a tree pane on the left to access the different templates (there are two templates: Lamp and User in Figure 2.10) and declarations (including global declarations under project and local declarations under each template) and a drawing canvas/text editor on the right. Figure 2.10 shows the editor with the Lamp example mentioned above. The Lamp system is represented with two templates: Lamp and User with one local declaration of clock `y` in Lamp and one global declaration of synchronization channel `press`. “Process assignments” contains declarations for the instances of templates and “System definition” shows the list of processes in the system, in the Lamp system it is “system Lamp, User;”.

The simulator can be used in three ways: the user can run the system manually and choose which transition to take; the random mode can be toggled to let the system run on its own, or the user can go through a trace (saved or imported from the verifier) to see how certain states are reachable. Figure 2.11 shows the simulator with the Lamp example. It is divided into four parts. The “control part” is used to choose and fire enabled transitions, go through a trace, and toggle the random simulation. The “variable view” shows the values of variables in the current state. The “system view” shows all instantiated automata and active locations of the current state. The “message sequence chart” shows the synchronizations between different processes as well as the active locations at every step.

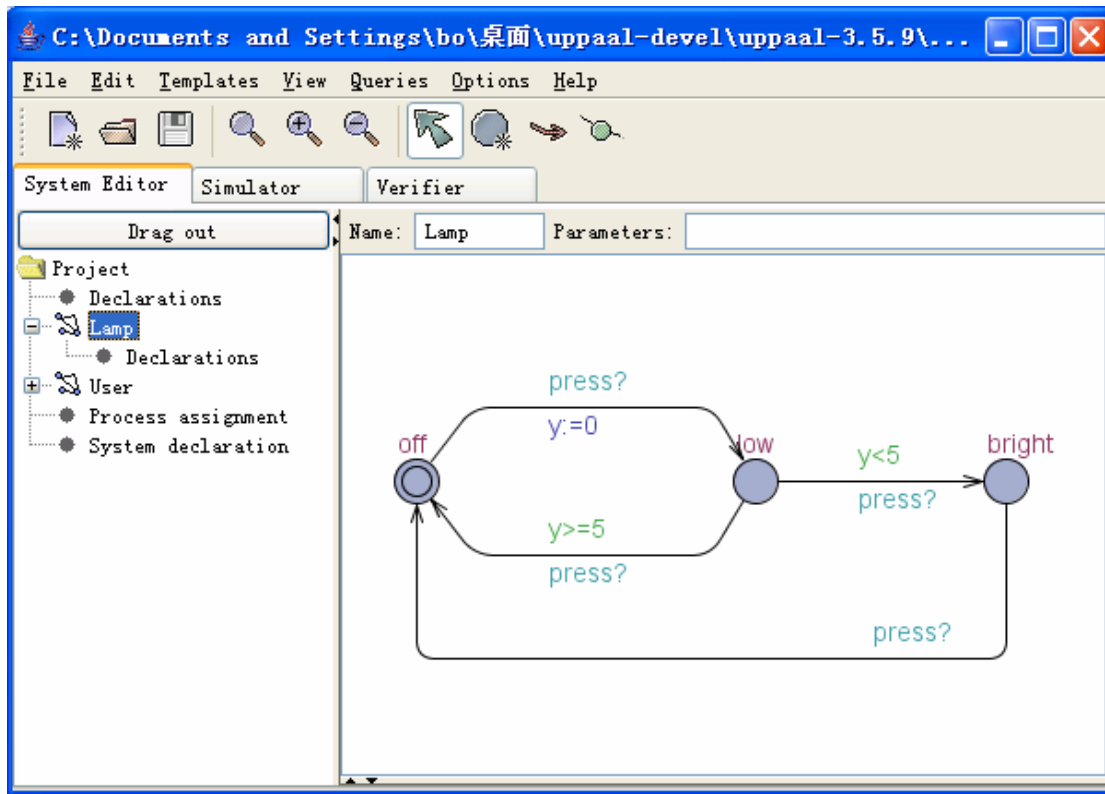


Figure 2.10: The editor of UPPAAL.

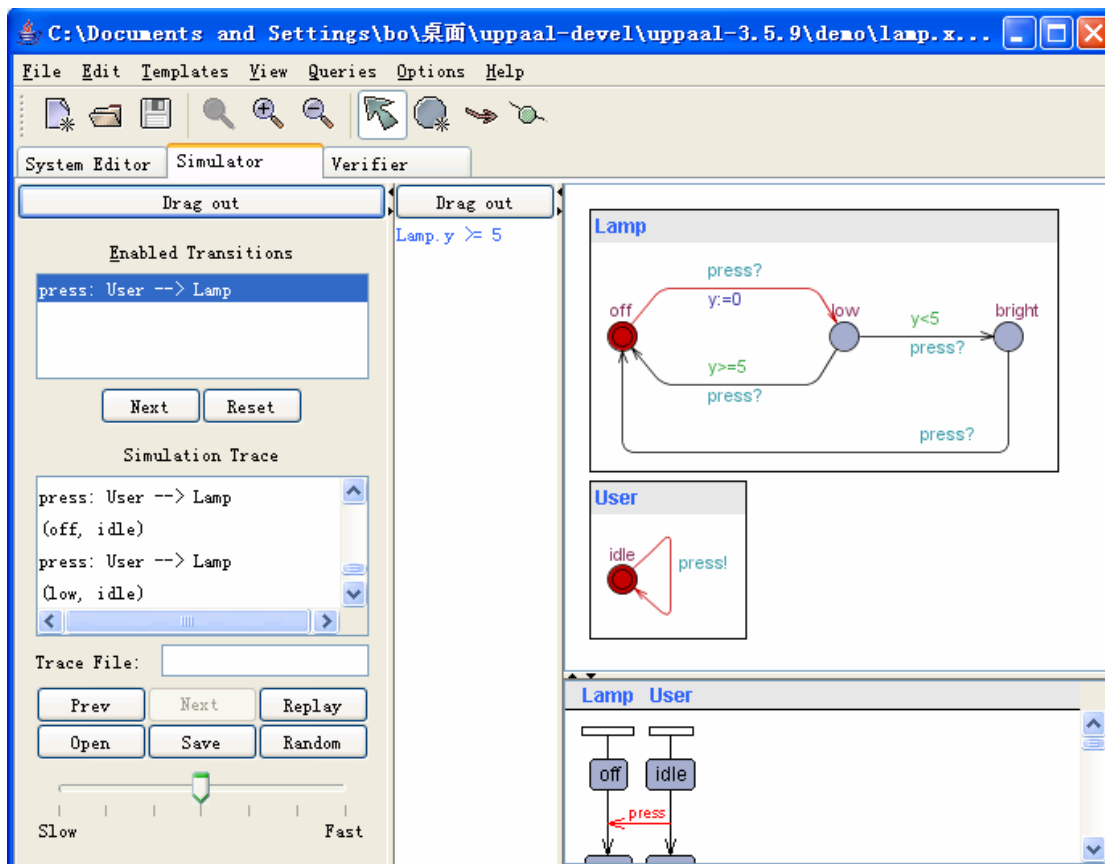


Figure 2.11: The simulator of UPPAAL.

The verifier is used to check properties. Satisfied properties are marked green and violated ones red, the properties are marked yellow in case that the verification is inconclusive with the approximation used. Figure 2.12 shows the verifier of UPPAAL.

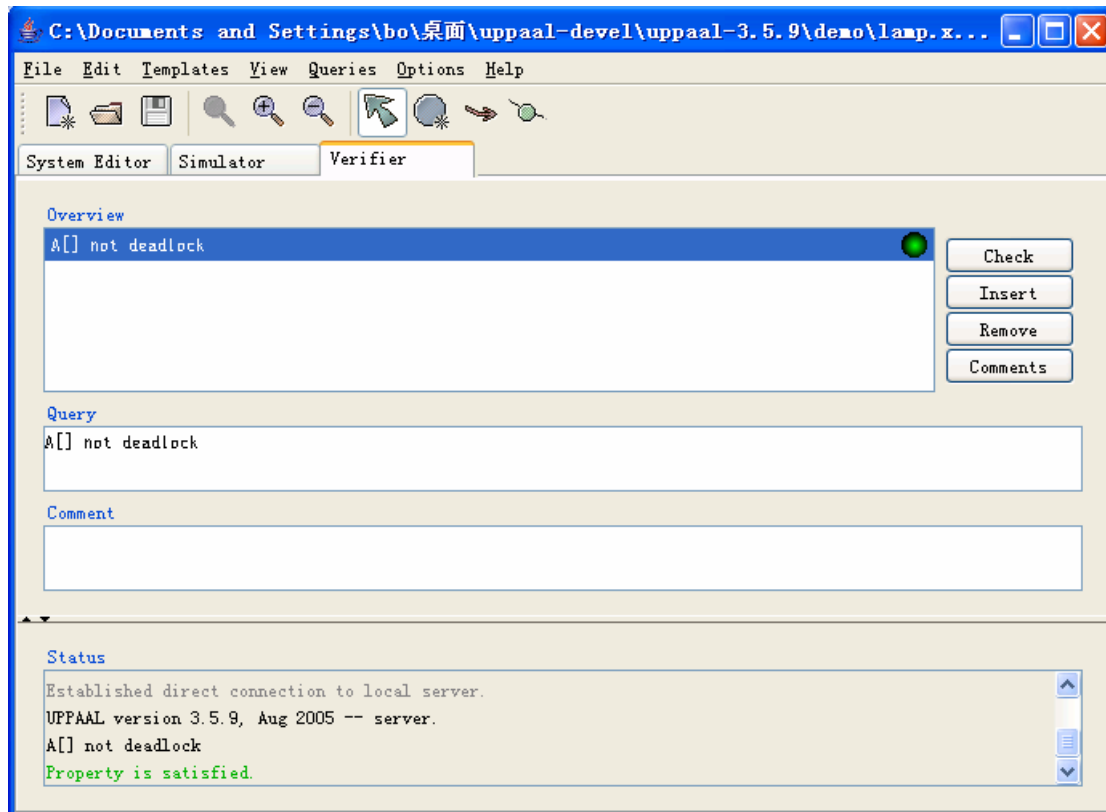


Figure 2.12: The verifier of UPPAAL.

Compared to the TLC model checker, UPPAAL does have a nice graphical user interface which makes a better communication between the user and the model checking process. The system is represented by state graphs, which can also be used in the simulation. The user can run the simulator manually which makes the user easier to locate the errors. Properties are easily checked with verifier. But UPPAAL also has its limited parts. It entirely focuses on real-time systems. Without concerning the real-time systems, the language used in UPPAAL is not expressive enough. TLA+ language and parts of functional and imperative languages can't be expressed correctly in UPPAAL, such as logic, sets, records, tuples types and their operators.

All in all, UPPAAL is a success model checker with a nice graphical user interface. It is available for free at <http://www.uppaal.com/> [16]. It has been ported to different platforms and it is in constant development. About how to use UPPAAL you can refer to UPPAAL tutorial [7].

2.3.3 The Xspin model checker

Spin (Simple Promela Interpreter) is a tool for analyzing the logical consistency of concurrent systems, in particular of data communication protocols. The system is described in a high level language called PROMELA (Process Meta Language). Given a model system specified in Promela, Spin can either perform random simulations of the system's execution or it can generate a C program that performs an efficient online verification of the system's correctness properties. During simulation and verification, Spin checks for the absence of deadlocks, unspecified receptions and an unexecutable code. The verifier can also be used to verify the correctness of system invariants, it can find non-progress execution cycles, and it can verify correctness properties expressed in next-time free linear temporal logic (LTL) formulae.

Spin can be used in three basic modes:

- As a simulator, allowing for rapid prototyping with a random, guided, or interactive simulations
- As an exhaustive verifier, capable of rigorously proving the validity of user specified correctness requirements (using partial order reduction theory to optimize the search)
- As proof approximation system that can validate even very large system models with maximal coverage of the state space

Xspin is the graphical interface of Spin. It runs independently from Spin itself, and helps by generating the proper Spin commands based on menu selections. Xspin runs Spin in the background to obtain the desired output, and wherever possible it will attempt to generate a graphical checkers that Spin can generate, and it knows when and how to execute it, so there is less to remember.

- Figure 2.13 shows the graphical interface of Xspin. It consists of a system editor which is used to edit PROMELA specifications. Xspin also provides the user facilities of simulation and verification.
- Figure 2.14 shows the simulation options window. The simulator has three modes: random, guided and interactive mode corresponding to Spin. Additionally, it provides the user several views of the simulation process: MSC (message sequence chart) which describes interaction (communication) between system components, Time Sequence panel which describes every step of simulation process with time, Data values panel which describes data values at each step, and Execution Bar panel which describes how many processes are used for executing the simulation and how many steps each process has taken. Examples of them are shown in Figure 2.15.
- The verifier provided by Xspin is shown in Figure 2.16. It can verify correctness properties expressed in next-time free linear temporal logic (LTL) formulae. In Figure 2.16 the property that “variable mutex is always less than 3” is expressed in LTL formula “[\square] mutex \leq 3” and it is verified by XSpin's verifier. The result shows it is not valid and a trace is generated containing counter examples. This

trace can be reloaded into the simulator in the guided mode.

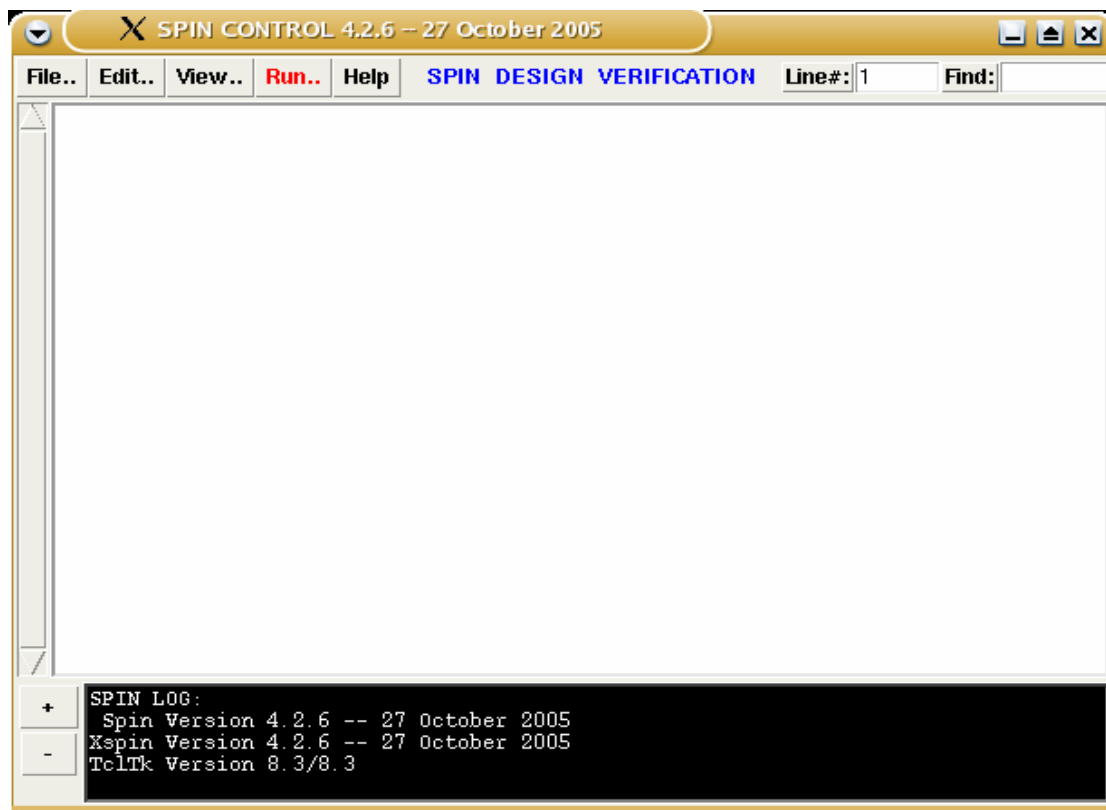


Figure 2.13: GUI of Xspin.

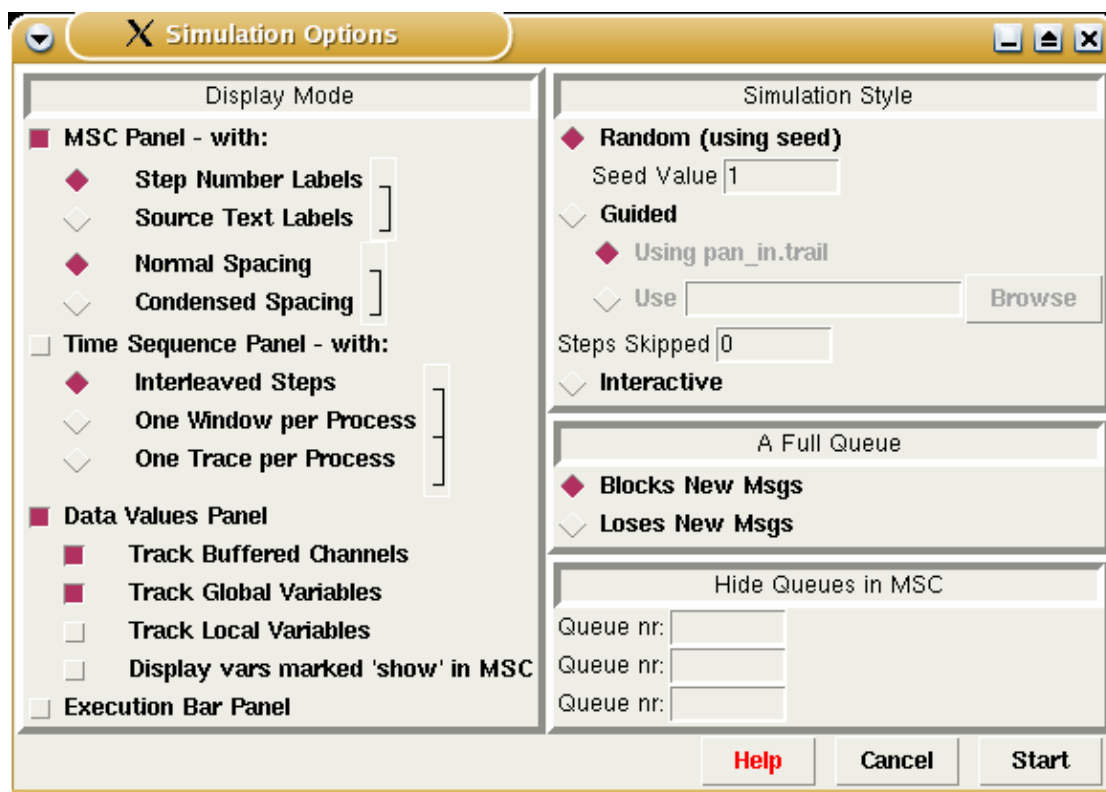


Figure 2.14: Simulation options of Xspin.

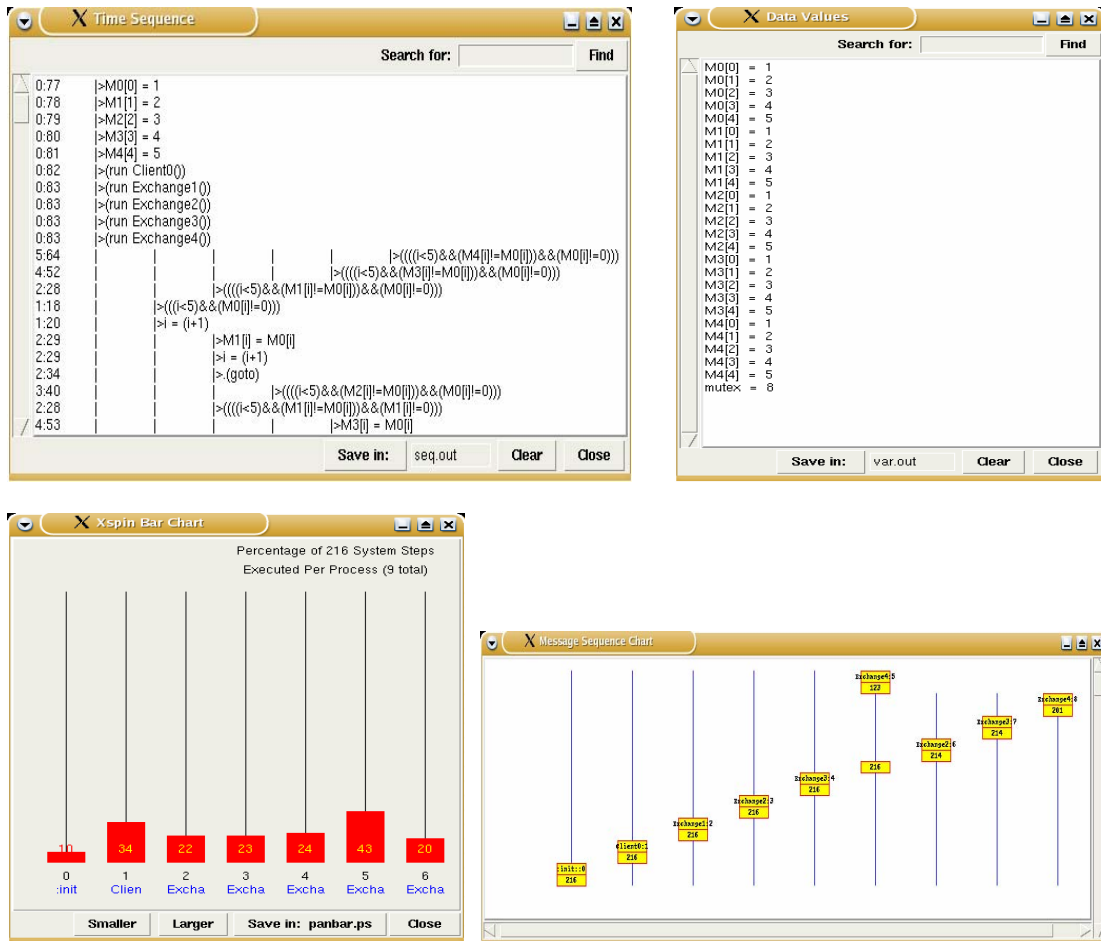


Figure 2.15: Various views provided by Xspin's simulator.

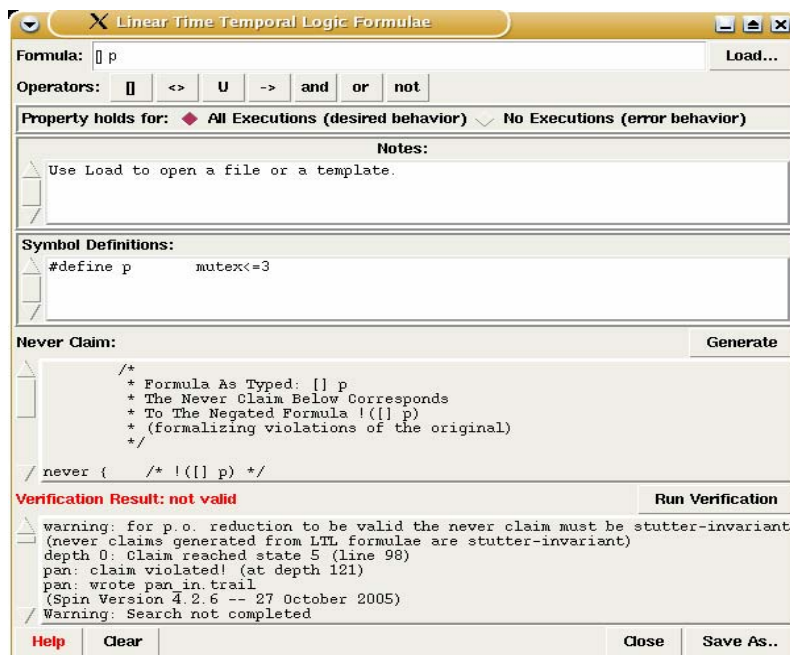


Figure 2.16: Xspin's simulator.

All in all, Spin model checker and Xspin focus on PROMELA specifications. They can find errors and provide counter examples. They can be used as a simulator, a verifier and a LTL prover. Especially Xspin lets the PROMELA user acquire Spin's facilities through a GUI, which provides the user more convenience and improves the communication between Spin and the user.

2.3.4 Comparison of TLC, UPPAAL and Xpin

From the introduction above we know that TLC, UPPAAL, Xspin are very different model checkers.

- The TLC model checker is used to check TLA+ specifications, which could specify a wide range of systems; while the UPPAAL model checker is especially used for specifying and checking real-time systems; and Spin (Xspin) model checker is used to check concurrent systems specified in PROMELA, especially distributed systems.
- The TLC model checker needs two input files, one is the TLA file that specifies the model of the real system and the other is the configuration file that specifies the properties and specifications to be checked, both of them are of text form and written in TLA+ language. The UPPAAL model checker uses a graph to specify the system combined with a set of variables or parameters, the properties to be checked are input into the verifier using special UPPAAL language. The Spin (Xspin) model checker uses only one PROMELA specification to specify the system and the properties to be checked are written in LTL formulae and are input into the verifier.
- UPPAAL specifications give the user a direct and clear view of the system and how it runs, while TLA+ specifications and PROMELA specifications give the user a complete view of the system in a text form.
- In terms of the user interface, both UPPAAL and Xspin have a nice GUI, which consists of system editor, simulator and verifier. Through the GUI the user can well communicate with the model checker; while TLC is oppositely primitive.
- In terms of language, although UPPAAL language can better describe real-time systems, to specify asynchronous and concurrent systems, TLA+ language and PROMELA language are much more expressive and highly abstract than UPPAAL language. For example, TLA language has more data structures, such as set, record, tuple, etc.

Chapter 3

Making specifications executable

The validation of a system with specifications and user requirements is rather difficult. Making specifications executable can ease the validation task, give the user immediate feedback of the behavior of the future software. However, some people argue that executable specifications should be avoided because many problems may arise. In Chapter 2 we mentioned TLA+ specifications, if we can make it executable, it could both keep the benefit of good understandability and readability and gain new benefit of good communication with the user. Can a specification be executable? How to make a specification executable? These questions will be discussed in this chapter.

3.1 Specifications are harmfully executable

A non-executable formal specification can be made executable by translating it into a functional language or an imperative language. The cost of this refinement is that each version of the specification is longer, further from the original requirements, and harder to understand. The benefit is the improvement in executability.

There are various arguments against the use of executable specifications. The most important problem is the tradeoff between expressiveness of the formal specification and its executability. Hayes' and Jones' [3] critique of executable specifications is most representative. They argue that the demands for high expressiveness and executability exclude each other, and that executable specifications should be avoided with many cases as examples. Their arguments can be summarized in the following statements.

- Executability inevitably limits the expressive power of a specification language and restricts the forms of specifications that can be used. Specifications should not contain the algorithmic details necessary to make them directly executable.
- The early validation with executable specifications is through executing individual test cases, which is less powerful than the general proving of properties.
- Executable specifications can unnecessarily constrain the choice of possible implementations. Executable specifications can produce particular results in cases where a more implicit specification may allow a number of different results.
- It is much easier to verify if an implementation meets an abstract specification than to verify if it meets an executable specification, which may have different data and program structures.

For the above reasons, Hayes and Jones suggest that to determine the validity of a specification one must examine the specification itself instead of the results of executing it. The executable specifications should be avoided.

3.2 Specifications are preferably executable

As opposed, Norbert E Fuchs argues that “specifications are (preferably) executable” [2] and A. Gravell argues that “executing formal specifications need not be harmful” [4]. Norbert E Fuchs takes examples from Hayes and Jones and translates them into a logic specification language (LSL) which is more expressive than Prolog with constructive elements and proves LSL is powerful and expressive enough to represent those examples in executable form. It is claimed that the translations are made “on almost the same level of abstraction and without essentially altering their structure”. He argues that “high expressiveness and executability need not exclude each other if specifications are written in declarative languages”.

An executable specification gives many benefits.

- It represents both a conceptual and a behavioral model of the system.
- The behavior of the system interacting with its environment can be demonstrated and observed before it actually exists in its final form.
- It allows early validation on an abstract level which can increase the correctness and the reliability of the system and reduce development costs and time.
- It allows to experiment with different requirements. This help to make a complete and precise requirement for the system.
- It helps with the communication between users and developers and it helps people, even those who are unfamiliar with it, to better understand the system.
- It can be embedded in various software development paradigms.

Thus declarative languages, especially logic languages, which combine high expressiveness with executability, can be used to write executable specifications on the required abstraction level by adding a small number of constructive elements.

3.3 Animating formal specifications

The validation of formal specifications is mainly based on either proof or on animation, which is execution. Just as Fuchs’s opinion, animation is the most important vehicle of communication between users and designers. In “animating TLA

specifications” [5], Y. mokhtari proposes two approaches to animate formal specifications, shown in Figure 3.1. The first approach is to translate formal specifications into an executable target language. The second approach is to define an operational semantics using a standard tool for the formal specification. The first approach is a programming activity and proving properties of the translation is often nontrivial.

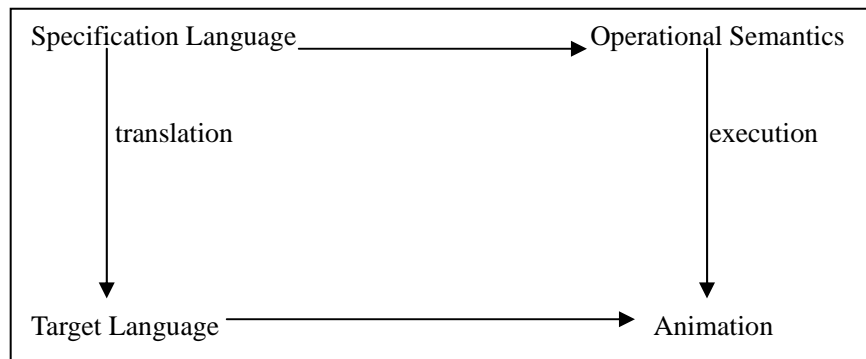


Figure 3.1 Animation approaches

There are many criteria to evaluate animation systems, such as the expressiveness of the animation language, efficiency, correctness which must be preserved and completeness, etc.

Chapter 4

GUI design for TLC

4.1 TLA+ specifications

TLA+ (temporal logic action) language makes it practical to describe a system by a single formula. It provides a mathematical foundation for describing systems.

TLC handles TLA+ specifications that have the standard form:

$$Init \wedge \Box_{\text{vars}} [Next] \wedge Temporal$$

where *Init* is the initial predicate, *Next* is the next-state action, *vars* is the tuple of all variables, and *Temporal* is a temporal formula that usually specifies a liveness condition. The *Temporal* part is optional.

TLA+ language specifies a system in the following ways:

- Firstly, deciding which parts of the system need to be specified. Generally speaking, the part that a specification can reveal most errors should be chosen because the goal is to find errors.
- Secondly, choosing the specification's level of abstraction. How to represent a system, with coarser-grained specification or finer-grained specification depends on the particular system need to be specified.
- Lastly, writing specifications. A specification consists of four parts: description of the specification, declaration of variables, the type invariant and initial predicate, a set of next-state actions and theorems.

The TLA+ specification provides lots of benefits.

- The TLA+ specification can help the design process. Finding and correcting errors can improve the design.
- The TLA+ specification can provide a clear, concise way of communicating a design. It may also help the user understand the system.
- Tools (such as TLC model checker) can be applied to use TLA+ specifications to find errors in the design and to help in testing the system.

TLA+ is quite good for specifying a wide class of systems—from program interfaces (APIs) to distributed systems. It can be used to write a precise, formal description of almost any sort of discrete system. As opposed to UPPAAL, it is especially well suited to describing asynchronous systems due to its high expressive language.

4.2 Animating TLA+ specifications

Chapter 3 mentioned two approaches to animating a formal specification. This research work focuses on animating TLA+ specifications with the first approach: translating the specification language into a target language.

4.2.1 Translating TLA+ language into a target language

In terms of animating a formal specification by translating the specification language into a target language, there are two major kinds of language could be considered as the target language. One is imperative languages, such as Java and C; the other is functional languages, such as Haskell and SML, etc. This research work investigates three languages: Java, Haskell and SML. A subset of first-order TLA+ is chosen as our specification language because TLA+ is a highly abstract and expressive language, thus not all of them could be translated into an imperative or a functional language. For example, the unbounded data structures can not be expressed well in an imperative language, which will be described in detail in the following sections. Here only the constant operators and miscellaneous constructs are considered, without temporal operators.

“Using Java as the target language”

Java as an imperative language is very powerful [10]. How each kind of TLA+ operators could be translated into Java is described below.

- The logic operators. The basic logic operators can be easily mapped into Java, see table 4.1. But Java doesn't support the predicate logic operators, thus they are not easily to be mapped into Java.

Firstly, the unbounded quantifiers $\forall x: p$, $\exists x: p$, *CHOOSE* $x: p$ could not be mapped into Java since the domain and type of x are unknown, and the predicate p could be in various forms which is rather difficult to handle.

Secondly, the bounded quantifiers $\forall x \in S: p$, $\exists x \in S: p$, *CHOOSE* $x \in S: p$ could be mapped into Java in the following forms.

(1) TLA+: $\forall x \in S: p$, *CHOOSE* $x \in S: p$

TLATEX: $\exists x \in S: p$, *CHOOSE* $x \in S: p$

Java: `Random gen = new Random ();`

`Object x = S.toArray () [gen.nextInt (S.size ())];`

(2) TLA+: $\forall x \in S: p$

TLATEX: $\forall x \in S: p$

Java: `for (int i=0; i<S.size ();i++)`

`Object x = S.toArray ()[i];`

TLA+	TLATEX	Java
<code>/\</code> or <code>\land</code>	\wedge	<code>&&</code>
<code>\ /</code> or <code>\lor</code>	\vee	<code> </code>
<code>~</code> or <code>\not</code> or <code>\neg</code>	\neg	<code>!</code>
<code>F => G</code>	$F \Rightarrow G$	<code>!F G</code>
<code><=></code> or <code>\equiv</code>	\equiv	<code>==</code>
TRUE	TRUE	<code>true</code>
FALSE	FALSE	<code>false</code>
BOOLEAN	BOOLEAN	<code>boolean</code>

Table 4.1: Translate logic operators from TLA+ to Java.

- **Sets.** Java supports the Set type but it doesn't provide enough operators on Set. Thus all the operators on Set defined in TLA+ should be mapped into self-defined functions in Java. Table 4.2 shows a subset of Sets operators in TLA+ which could be translated into Java. The other operators are $=$, \neq , \in , \notin , \cup , \cap , \subseteq , \setminus , SUBSET, UNION.
- **Records.** There are two ways to map records into Java, one is to map it into an array and the other is to map it into a class. Table 4.3 shows the former mapping (to array), the field names of a record is mapped to index numbers. A transform function between string names and index numbers should be constructed.

TLA+	TLATEX	Java
$\{e_1, \dots, e_n\}$	$\{e_1, \dots, e_n\}$	<pre>Object obj={e₁,...,e_n}; Set M=new HashSet(); for(int i=0;i<obj.length;i++) M.add(obj[i]);</pre>
$M=\{x \mid x \in S : p\}$	$M=\{x \in S : p\}$	<pre>Set M=new HashSet(); for(int i=0;i<S.size();i++){ Object obj=S.toArray()[i]; If(p) M.add(obj);}</pre>
$\{e : x \mid x \in S\}$	$\{e : x \in S\}$	<pre>for(int i=0;i<S.size();i++){ Object obj=S.toArray()[i]; M.add(obj);}</pre>

Table 4.2: Translate Sets constructors from TLA+ to Java.

TLA+	TLATEX	Java
$e.h$	$e.h$	$e[\text{transform}(h)]$
$[h_1 \mapsto e_1, \dots, h_n \mapsto e_n]$	$[h_1 \mapsto e_1, \dots, h_n \mapsto e_n]$	$e[\text{transform}(h_i)] = e_i$
$[h_1 : S_1, \dots, h_n : S_n]$	$[h_1 : S_1, \dots, h_n : S_n]$	$e[\text{transform}(h_i)] \in S_i$
$[r \text{ Except } !.h=e]$	$[r \text{ Except } !.h=e]$	$r[\text{transform}(h)] = e$

Table 4.3: Translate Records from TLA+ to Java.

- Other operators. Tuples in TLA+ can be mapped into (multidimensional) arrays in

Java. Strings and numbers, most of functions and miscellaneous constructs operators are similar in Java except function operators DOMAIN f and $[S \rightarrow T]$. Additionally the LETIN operator “ $LET\ d_1 \triangleq e_1 \dots d_n \triangleq e_n\ IN\ e$ ” should be mapped into such a form: the definition of $d_1 \dots d_n$, followed with the statement e .

Java is possible to substitute a reasonable subset of TLA+ language, but there are two problems difficult for Java to solve. One is the type problem. TLA+ is a type-free language while Java has strict requirements of types, so it is hard for Java to give the correct type for an object. The other problem is about the large or infinite set. Java doesn't have a lazy-evaluation mechanism, so it is a huge cost to compute a large set during the compile time.

“Using Haskell as the target language”

Haskell [6, 8, 9, 14], as a functional language, has many similar mechanisms as TLA+ language, thus it is much easier to map TLA+ language into a functional language than an imperative language. Table 4.4 shows the translation from TLA+ into Haskell. Like Java, there are some operators which can't be directly translated into Haskell, such as unbounded quantifiers, function operators DOMAIN f and $[S \rightarrow T]$, etc. Additionally it supports lazy-evaluation. But there is a serious problem Haskell could not solve, that is Haskell could not assign new values to variables, while in TLA+ specifications assigning new values to global variables is very common and important. Due to this reason, Haskell can't be chose as a target language.

The Logic operator

TLA+	TLATEX	Haskell
\wedge or \backslash land	\wedge	$\&\&$
\vee or \backslash lor	\vee	$\ \$
\neg or \backslash not or \backslash neg	\neg	not
$F \Rightarrow G$	$F \Rightarrow G$	not $F \ \ G$
\Leftrightarrow or \backslash equiv	\equiv	$==$
TRUE	TRUE	True
FALSE	FALSE	False
BOOLEAN	BOOLEAN	Bool
$\forall x \in S: p$	$\forall x \in S: p$	filter (p) S
$\exists x \in S: p$	$\exists x \in S: p$	$t = \text{filter } (p) S$ $y = t!!(\text{randomR } (0, (\text{length } t)-1))$
CHOOSE $x \in S: p$	$CHOOSE\ x \in S: p$	$t = \text{filter } (p) S$ $y = t!!(\text{randomR } (0, (\text{length } t)-1))$

Sets

TLA+	TLATEX	Haskell
$=$	$=$	$==$
$\#$ or \neq	\neq	\neq

$\backslash\text{in}$	\in	elem
$\backslash\text{notin}$	\notin	notElem
$\backslash\text{cup}$ or $\backslash\text{union}$	\cup	++
$\backslash\text{cap}$ or $\backslash\text{intersect}$	\cap	intersect
\backslash [set difference]	\backslash	$\backslash\backslash$
$\{e_1, \dots, e_n\}$	$\{e_1, \dots, e_n\}$	$[e_1, \dots, e_n]$
$\{x \backslash\text{in } S : p\}$	$\{x \in S : p\}$	filter (p) S eg, filter (>6) [3,7]
$\{e : x \backslash\text{in } S\}$	$\{e : x \in S\}$	map e S eg, map (+6) [1,2,3]
UNION S	UNION S	union

Functions

TLA+	TLATEX	Java
$f[e]$	$f[e]$	$f \ e$
DOMAIN f	DOMAIN f	
$[x \backslash\text{in } S \mid \rightarrow e]$	$[x \in S \mapsto e]$	$\text{map}(\backslash x \mapsto 4) S$

Records

TLA+	TLATEX	Java
e.h	e.h	h
$[h_1 \mid \rightarrow e_1, \dots, h_n \mid \rightarrow e_n]$	$[h_1 \mapsto e_1, \dots, h_n \mapsto e_n]$	$x = (e_1, \dots, e_n)$ $(h_1, \dots, h_n) = x$
$[h_1 : S_1, \dots, h_n : S_n]$	$[h_1 : S_1, \dots, h_n : S_n]$	$x = (S_1 \text{!!}(\text{mod } r(\text{length } S_1))), \dots,$ $(S_n \text{!!}(\text{mod } r(\text{length } S_n)))$ $(h_1, \dots, h_n) = x$

Tuples

TLA+	TLATEX	Java
e[i]	e[i]	fst e, snd e (only used in 2-tuples)
$\langle e_1, \dots, e_n \rangle$	$\langle e_1, \dots, e_n \rangle$	(e_1, \dots, e_n)

String and Numbers

TLA+	TLATEX	Java
"c1...cn"	"c ₁ ...c _n "	"c1...cn"
STRING	STRING	String
$d_1 \dots d_n \ d_1 \dots d_n \cdot d_{n+1} \dots d_m$	$d_1 \dots d_n \ d_1 \dots d_n \cdot d_{n+1} \dots d_m$	$d_1 \dots d_n \ d_1 \dots d_n \cdot d_{n+1} \dots d_m$

Miscellaneous constructs

IF p THEN e ₁ ELSE e ₂	IF p THEN e ₁ ELSE e ₂	if p then e1 else e2
CASE p ₁ ->e ₁ []...[] p _n ->e _n []OTHER->e	CASE p ₁ → e ₁ □ ... □ p _n → e _n □ OTHER → e	case x of p ₁ → e ₁ ... p _n → e _n
LET d ₁ =e ₁ ...d _n =e _n IN e	LET d ₁ ≜ e ₁ ...d _n ≜ e _n IN e	let d1=e1 ... dn=en in e
/\	∧	&&
\/	∨	

Table 4.4: Translate from TLA+ to Haskell.

“Using SML as the target language”

SML is a functional language but supports some mechanisms of imperative languages [11]. The translation from TLA+ into SML is described below.

- The logical operators. Table 4.5 shows the mapping of basic operators and bounded operators.

TLA+	TLATEX	SML
/\ or \land	∧	andalso
\/ or \lor	∨	orelse
~ or \lnot or \neg	¬	not
F => G	F ⇒ G	not F orelse G
<=> or \equiv	≡	=
TRUE	TRUE	true
FALSE	FALSE	false

BOOLEAN	BOOLEAN	Bool
$\backslash A\ x\ \backslash in\ S: p$	$\forall x \in S : p$	<i>List.all</i> (<i>fn</i> $x \Rightarrow p$) <i>S</i>
$\backslash E\ x\ \backslash in\ S: p$	$\exists x \in S : p$	<i>List.exists</i> (<i>fn</i> $x \Rightarrow p$) <i>S</i>

Table 4.5: Translate Logic operators from TLA+ to SML.

- Sets. See table 4.6. Some operators should be user-defined, like SUBSET.

TLA+	TLATEX	SML
=	=	=
# or /=	\neq	$\langle \rangle$
$\backslash in$	\in	<i>List.exists</i> (<i>fn</i> $x \Rightarrow x = y$) <i>S</i>
$\backslash notin$	\notin	<i>List.all</i> (<i>fn</i> $x \Rightarrow x \langle \rangle y$) <i>S</i>
$\backslash cup$ or $\backslash union$	\cup	@
$\{e_1,...,e_n\}$	$\{e_1,...,e_n\}$	$[e_1,...,e_n]$
$\{x\ \backslash in\ S: p\}$	$\{x \in S : p\}$	<i>List.filter</i> (<i>fn</i> $x \Rightarrow p$) <i>S</i>
$\{e: x\ \backslash in\ S\}$	$\{e : x \in S\}$	<i>map</i> (<i>fn</i> $x \Rightarrow e$) <i>S</i>

Table 4.6: Translate Sets operators from TLA+ to SML.

- Records. See table 4.7.

TLA+	TLATEX	SML
e.h	e.h	#h(e)
$[h_1 : S_1, \dots, h_n : S_n]$	$[h_1 : S_1, \dots, h_n : S_n]$	$\{ \quad h_1 = \text{ref } \text{List.nth}(S_1$ Random.randRange $(0, \text{length } S_1 - 1) \text{ rand})$ $\dots, h_n = \text{ref } \dots \}$
$[h_1 \mapsto e_1, \dots, h_n \mapsto e_n]$	$[h_1 \mapsto e_1, \dots, h_n \mapsto e_n]$	$\{h_1 = \text{ref } e_1, \dots, h_n = \text{ref } e_n\}$
$[r \text{ Except } !.h=e]$	$[r \text{ EXCEPT } !.h=e]$	$\#h \ r := e$

Table 4.7: Translate Records operators from TLA+ to SML.

- SML has similar forms of most operators of functions, tuples, string and numbers and miscellaneous constructs, except function operators DOMAIN f and $[S \rightarrow T]$.

SML could solve the assigning problem with reference. Concluding, SML seems the most ideal language to be used as the target language.

4.2.2 Animator design

The animator provides a way to interact with the user. The goal of the animator design is to provide as much convenience as possible for the user to help them communicate with the model checker and find errors in the specifications. The following are some animator designs for TLA+ specifications.

“Using the state graph to represent executable TLA+ specifications”

The most direct way is to show the state space with all the reachable states, the values of all the variables in each state, and the transitions from one state to another. Figure 4.1 shows the complete state graph for the clock specification described in section 2.3.1. The state graph is consists of 12 states with different values of variable hr. The directed edges represent the possible transitions from one state to another state. This gives the user a clear and direct view of the clock system.

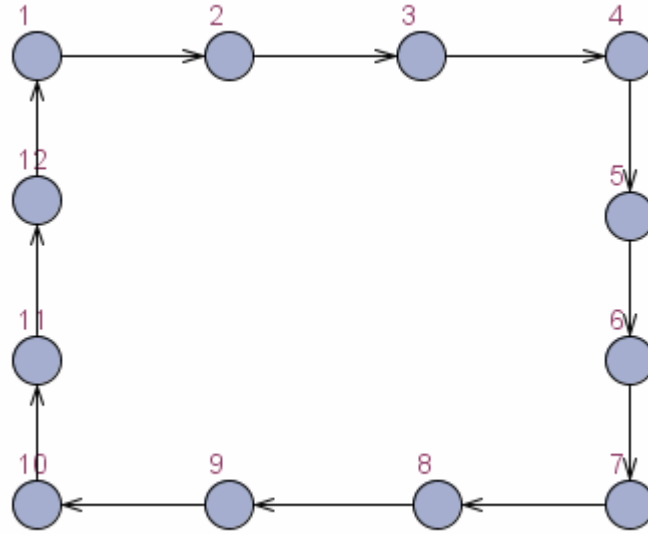


Figure 4.1: State graph for the clock specification.

Unfortunately this is impossible for those specifications with a large state space. One reason is that it is hard to show a complete state graph for a specification that generates thousands of states totally and such a big state graph will make it be more difficult to communicate with the user. Another reason is that the state space is closely related to the domain of the variables such as constants. The initial states and distinct states have some obvious proportional relationship with the number of variables, but not with the total states and depth of the state graph. A specification having many variables is really complex.

“Using the action graph to represent executable TLA+ specifications”

The UPPAAL model checker has been introduced in section 2.3.2. It models the system to a network of several timed automata in parallel and gives a nice system view and good communication with the user. Can TLA+ specifications be mapped to a similar model? Examine TLA+ specifications, similar to the UPPAAL model which consists of states, a TLA+ specification consists of actions. I tried to map a TLA+ specification into an action graph in such a way: each action (including the initial predicate and all the next actions) in TLA+ specification is represented with one node; there is an directed edge from the initial node to every next action; there are directed edges from every next action to all the other next actions and these next actions make a circulation. Here the directed edge means a transition from one action to another action with the guard and new values accord with TLA+ specifications.

Many TLA+ specifications are modeled to such an action graph and run under the UPPAAL system. Figure 4.2 shows an example of the clock specification. It consists of two actions: HCini is the initial predicate and HCnxt is the next action. There is a transition from the initial action to the next action. Because there is only one next action here, HCnxt makes a circulation itself.

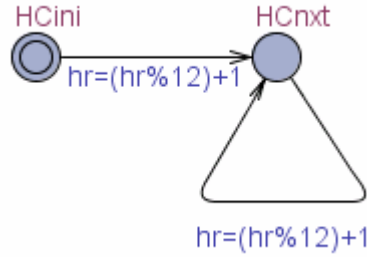


Figure 4.2: Action graph for the clock specification.

This action graph model can be run with the UPPAAL simulator and verifier. Figure 4.3 gives a nice view of the simulation. The simulator starts with the initial predicate with $hr = 4$ and repeats the next action and generates new values.

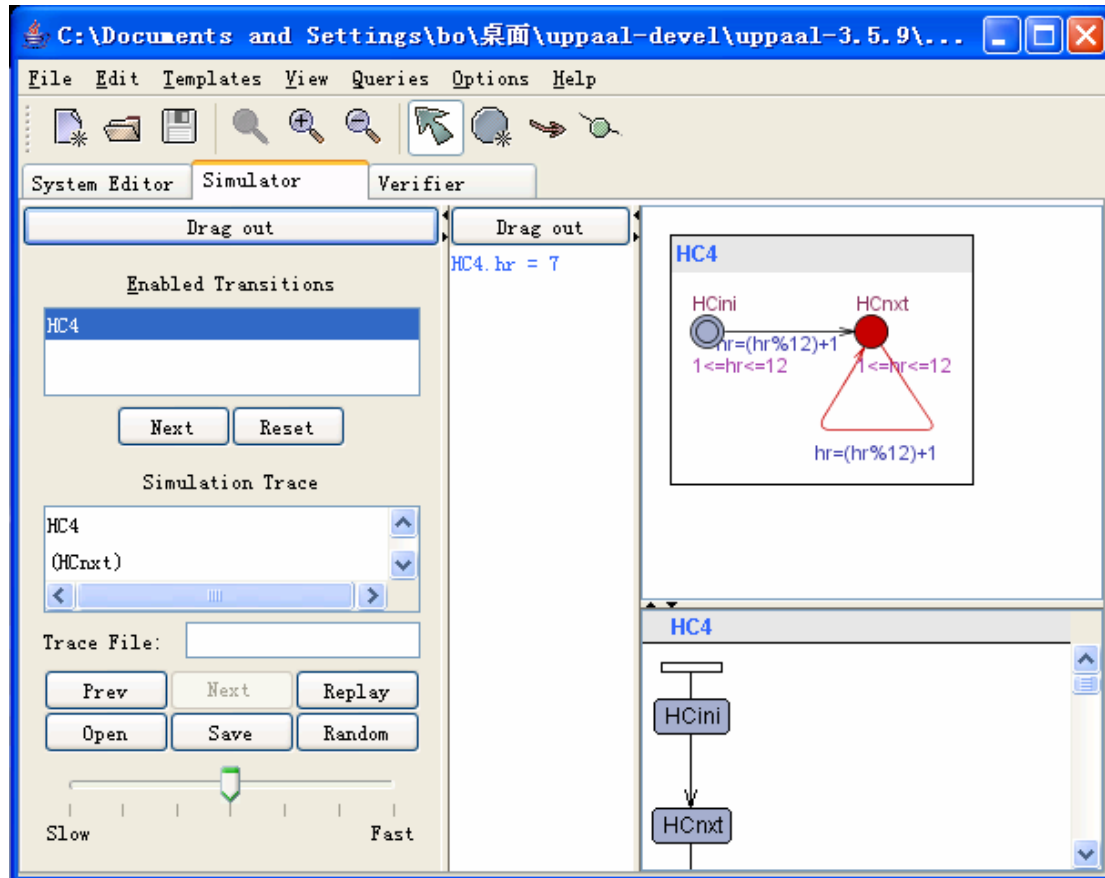


Figure 4.3: Simulation in UPPAAL with an action graph model of a TLA+ specification.

The verifier of UPPAAL can be also applied to such a model. The following statements check the properties of the clock specification: the first statement checks if there is a deadlock, the second statement checks if all the next actions are reachable,

the last statement checks the theorem of the specification “ $HC \Rightarrow []HCini$ ”. And all of these properties are satisfied with the action graph model.

$A[]$ not deadlock

$E \triangleleft HC4.HCnxt$

$A[] HC4.HCini$ and $HC4.HCnxt$ imply $1 \leq HC4.hr \leq 12$

It seems that this could be a possible and nice animator design for TLA+ specifications. Further more, many more complex cases show a good result too. But further examined this method, it reveals some problems:

- The network could be rather complex. To preserve the completeness, there should be possible transitions from the initial action to all the next actions and from each next action to all the other next actions as specified in the TLA+ specifications. Such a network is really complex, especially to those big systems with lots of actions. Further more, only a small part of the transitions are useful due to the guard. One way to solve this problem is to optimize the model by detecting and deleting the useless transitions which will never be reached. But such an optimization algorithm will make the animator design more complex.
- The correctness is hard to be preserved. As we know TLA+ is a rather expressive language, using the action graph to represent TLA+ specifications require the same expressive language to guarantee the correctness of the specifications. This is related to the translation work mentioned above. In fact the translation is not an easy work and only a subset of the TLA+ specification could be animated.

“Using the message sequence chart to represent executable TLA+ specifications”

From section 2.3.2 we know both the UPPAAL model checker and the XSpin model checker use a “message sequence chart” to show the synchronizations between the different processes and the active locations at every step of the simulation. This chart can well describe TLA+ specifications too. Figure 4.4 shows an example of the “message sequence chart” of the clock specification. Here we don’t concern the synchronization situation. This chart consists of two active locations (actions in TLA+ specifications): HCini and HCnxt. Running the simulator the chart should show the transitions between these actions, as shown in the figure. It starts from the initial predicate HCini to the next action HCnxt, and then stays in the HCnxt action but with many transitions from one state to another state. This “message sequence chart” can both interact with the user and guarantee the completeness and correctness of the specification.

“Using the state tree to represent executable TLA+ specifications”

Another nice design is to use the state tree. In the simulation or model-checking mode a sequence of states will be generated, we establish a state tree with all the variables to represent such a state sequence as shown in Figure 4.5. It is also a good way to both

interact with the user and guarantee the completeness and correctness of the specification.

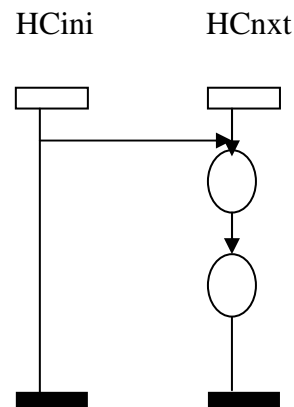


Figure 4.4: Message sequence chart for clock specification.

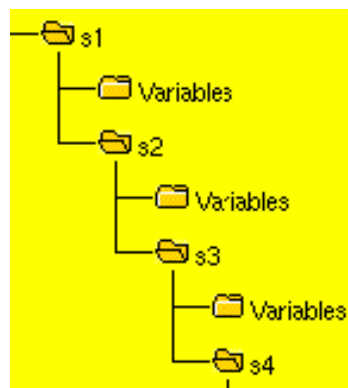


Figure 4.5: A state tree example.

Concluding, I mentioned four animator designs and related problems. The message sequence chart and the state tree should be the best choices and could be well applied to the animation of TLA+ specifications.

4.3 GUI design

The GUI for the TLC model checker should include the following components:

- Source Editor: edit source files using a point-and click editor. Here a TLA+ parser is needed to check the syntax errors.

- Animator (Simulator): animate TLA+ specifications. The animator consists of at least three components: an “action list” which shows all the actions in a TLA+ specification and it should allow the user to choose the next action they expect to execute; a “state tree” which shows a sequence of states generated by the animator with the values of all variables; a “message sequence chart” which shows the action transitions.
- Model checker (Verifier): model checker for TLA+ specifications, the TLC model checker is expected to be used here to check the correctness of the input specification and generate a counter example when an error occurs.

Figure 4.6 shows the architecture diagram for this GUI.

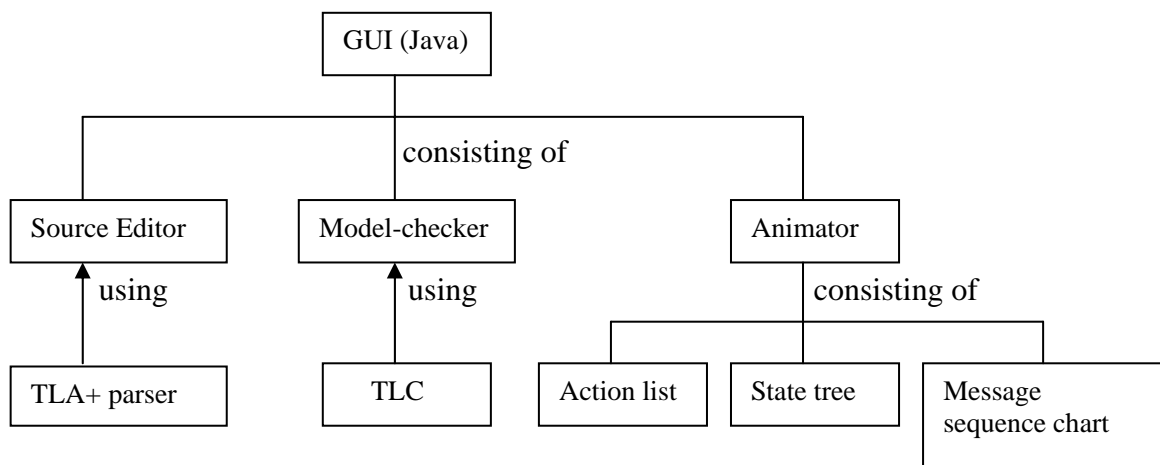


Figure 4.6: Architecture diagram of GUI for TLC.

There are two kinds of high level Java IDE (integrated development environment) can be used to develop this GUI. One is Netbean and the other is Eclipse. Eclipse has many advantages over Netbean: better looking, higher speed, far more stable, more flexible, support more plug-ins, while Netbean is more user friendly, J2EE support, better development. Eclipse is an extensible, open source IDE. Its goal is to develop a robust, full-featured, commercial-quality industry platform for the development of highly integrated tools. Additionally, the Eclipse IDE is extended with tools for development in functional programming languages (currently Haskell and OCaml), providing support for a wide range of tools (compilers, interpreters, doc tools etc.) and a convenient and configurable environment. We can get it and related information from the website <http://eclipsefp.sourceforge.net/> [13].

A parser is needed to generate AST for the given TLA+ specifications, there are three choices. The first one is the syntactic analyzer, written by Jean-Charles Gregoire and David Jefferson that parses a TLA+ specification and checks it for errors. It also serves as a front end for other tools, such as TLC. Because only a subset of TLA+ language could be translated into the target language, using the analyzer costs lot of work to do with the code generation. The second choice is to create such an appropriate parser by hand, although there are much work to be done, it could be more

suitable. Both these two seem not very good. Fortunately we found the third choice which is also the best choice, a free TLA+ Front-End tool which can be used for the TLA+ syntactic analyzer. This tool attempts to follow as closely as possible the TLA+ language. It includes a parser and a semantic checker, which resolves symbols across a full specification. It is possible to use it to write TLA+ tools, such as animators, model checkers, provers, pretty printers, etc. To each TLA+ specification, the TLA+ Front-End gives out a nice flattened parse tree.

The construction work will focus more on the animator and graphical user interface.

4.4 Project development schedule

This section will give an overview of how this project will be constructed and how to manage this project.

Figure 4.7 shows the structure of this project. The two grey blocks are available and the blocks encircled by the red line belong to GUI design parts; the TLA+ Front-End tool serves as a syntactic analyzer; TLA+ and SML translator will translate the flattened parse tree generated by the TLA+ Front-End tool into SML code; a SML parser is used to parse the generated SML file, give out syntax error reports and support the simulator and verifier. Not all the work could be done, some functions are additional.

Project development schedule is as following.

- Step 1—Translators: translate a subset of TLA+ language into compiled SML language. From the TLA+ parser provided by the free TLA+ Front-End tool [12], we could get a nice flattened parse tree. The result is to translate TLA+ specifications into SML specifications through this flattened parse tree generated by the TLA+ front end.
- Step 2—Simulator: if the translated SML specifications can be successfully compiled by a SML parser, we can make the TLA+ specifications executable with the simulator.
- Step 3—Integrated GUI: a friendly graphical user interface should be developed to integrate all the parts of the project, including the TLA+ Front-End tool, the translators, the SML parser, the simulator, the TLC model checker, etc.
- Step 4—Additional work: if time is allowed, a sequence of additional work could be done: making the project deal with fairness properties of TLA+ specifications; adding more functionalities into the simulator, such as using a state tree to describe the simulation process; develop a verifier, make it executable and call the simulator with a counter example to show the errors.
- Step 5—Testing: test the system with various TLA+ specifications and evaluate the developed system.

- Step 6—Writing thesis: prepare the thesis and presentation.

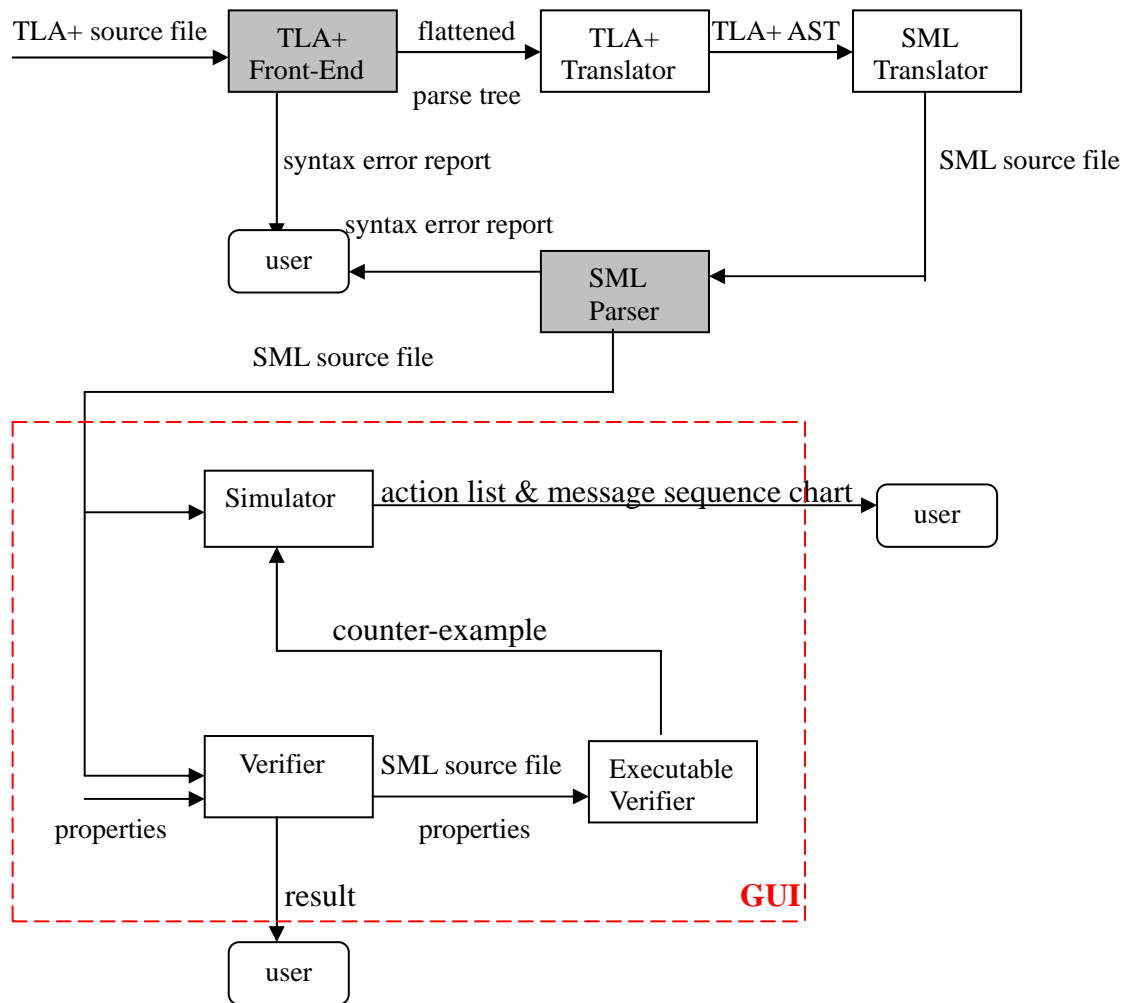


Figure 4.7: Structure of this project.

Chapter 5

Conclusion of literature survey

In the above literature survey part of this thesis, the model checking technique, theory of executable specifications and a case study with TLA+ specifications were presented. The objective of this research project is to develop a GUI design for the TLC model checker, which is described in chapter 4. Firstly, the TLA+ specification should be animated by translating the specification language into a target language. The SML language is much more suitable than Java and Haskell to be the target language. Secondly, the subset of TLA+ language which will be animated was chosen. The message sequence chart and state tree are good forms for animator. Lastly, the development tools and the architecture of graphical user interface were decided.

I intend this work as the first step towards the development of an animator with a graphical user interface for TLC.

Finally, there may be other languages more suitable to be the target language for animating TLA+ specifications. This could be viewed as a future work.

Part II

Final Thesis

by

Bo Wang

THESIS

MASTER OF SCIENCE

Department of Computer Science

Faculty of EEMCS

Delft University of Technology

June, 2006

Chapter 6

Translating TLA+ into SML

Before constructing the GUI for the TLC model checker, TLA+ specifications should be made executable. This is implemented by translating TLA+ specifications into SML code.

6.1 Introduction

6.1.1 The TLA+ Language

The TLA+ language of Lamport is a well-accepted specification technique for describing systems at a rather high level of abstraction. It is based on TLA, which is built on a logic of actions – a language for writing predicates, state functions, actions and a logic for reasoning about them. A predicate is a Boolean expression containing constants and variables; a state function is a non-Boolean expression containing constants and variables; and an action is a Boolean expression containing constants, variables, and primed variables. The complete specification language TLA+ includes such a language (logic of actions).

A TLA+ formula is true or false on a behavior, a sequence of states, where a state is an assignment of values to variables. Syntactically, a TLA+ formula may have one of the following basic forms:

P : a predicate, satisfied by a behavior iff P is true for the initial state.

$\Box[A]_f$: satisfied by a behavior iff every step satisfies A (an action) or leaves f (a state function) unchanged. Symbol \Box means “always true”.

$\Box F$: F is always true; satisfied by a behavior iff F is true for all suffixes of the behavior. Symbol \Box means “always true”.

$\neg F, F \equiv G, F \Rightarrow G, F \wedge G, F \vee G$: F and G are TLA formulas; the Boolean

operators $\neg, \equiv, \Rightarrow, \wedge, \vee$ have their usual meanings.

TLA+ is quite good for specifying a wide class of systems – from program interfaces (APIs) to distributed systems. It can be used to write a precise, formal description of almost any sort of discrete system.

6.1.2 The SML Language

Standard ML (SML) is a safe, modular, strict, functional, polymorphic programming language with compile-time type checking and type inference, garbage collection, exception handling, immutable data types and updatable references, abstract data types, and parametric modules. It has efficient implementations and a formal definition with a proof of soundness. Some important features are introduced as following.

ML has higher-order functions: functions can be passed as arguments, stored in data structures, and returned as results of function calls. Functions can be statically nested within other functions; this lexical scoping mechanism gives the ability to create “new” functions at run time.

Function calls in ML, like those of C, Pascal, C++, Java, etc., evaluate their arguments before entering the body of the function. Such a language is called strict or call-by-value, in contrast to some functional programming languages that are lazy or call-by-need. Strict evaluation makes it easier for the programmer to reason about the execution of the program.

The ML programmer need not write down the type of every variable and function-parameter: the compiler can usually infer the type from context. This makes programs more concise and easier to write.

Automatic de-allocation of unreachable data makes programs simpler, cleaner, and more reliable.

In ML, most variables and data structures – once created and initialized – are immutable, meaning that they are never changed, updated, or stored into. This leads to some powerful guarantees of noninterference by different parts of the program operating on those data structures. In a functional language such as ML, one tends to build new data structures (and let the old ones be garbage collected) instead of modifying old ones.

ML has updatable (assignable) reference types, so that in those cases where destructive update is the most natural way to express an algorithm, one can express it directly.

There are several implementations of Standard ML available for a variety of hardware and software platforms. Standard ML of New Jersey (SML/NJ) is a comprehensive implementation and is the most widely used. SML/NJ extends the SML '97 Basis Library with several top-level structures. It was developed jointly by Bell Laboratories and Princeton University.

6.1.3 Overview of the GTLA Translator's design

The objective of this section's work is to implement the translation from TLA+ specifications to SML code. Given a TLA+ specification, it is first input into the TLA+ Front-End tool, which includes a parser and a semantic checker. After that a flattened parse tree is generated. The GTLA compiler is constructed to generate the SML file from this flattened parse tree and the configuration file automatically. The LLgen parser generator and the GNU compiler collection are used as development tools. Figure 6.1 describes the work to be done.

- Lexical analysis is a process that converts a stream of characters to a stream of tokens. The purpose of producing these tokens is usually to forward them as input to another program, such as a parser. Lex is a lexical analyzer generator. A Lex source is a table of regular expressions and corresponding program fragments (e.g. the token description). The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is recognized, the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by Lex.
- Syntax analysis is a process in compilers that recognizes the structure of programming languages. It is known as parsing. After lexical analysis, it is much easier to write and use a parser, as the language is far simpler. Context-free grammar is usually used for describing the structure of languages. Programs that do parsing are called parsers. LLgen is a program that generates parsers in the C programming language.
- Code generation is a process to generate the concrete code from the AST. The general idea behind code generation is to decompose the tree structure of the syntax tree (AST) into a sequence of instructions of the target machine.

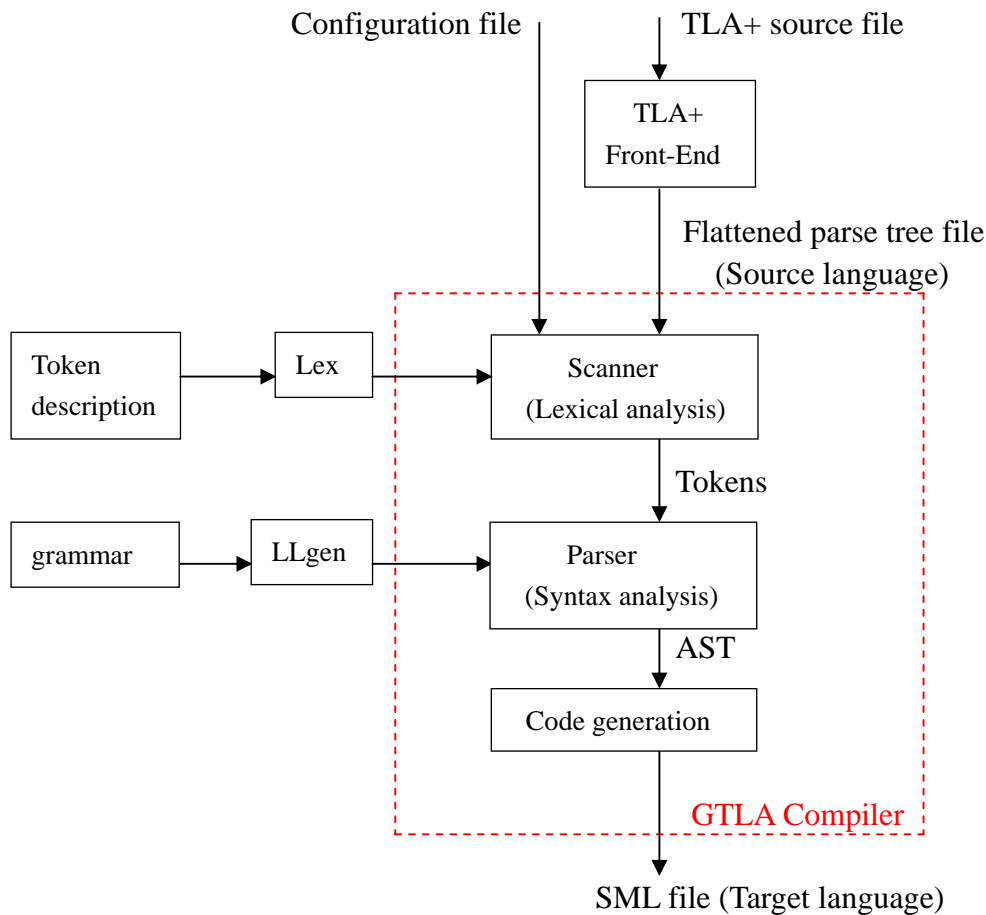


Figure 6.1: Construction architecture.

The details of the GTLA compiler construction work are described below.

6.2 Lexical analysis

Lexical analysis is the name given to the processing of an input sequence of characters (such as the source code of a computer program) to produce, as output, a sequence of symbols called “lexical tokens”, or just “tokens”. For instance, lexers for many programming languages will convert the character sequence 123 abc into the two tokens 123 and abc. The purpose of producing these tokens is usually to forward them as input to another program, such as a parser.

The lexical analyzer generator Lex is used to generate a lexical analyzer given a token description, which contains regular expressions that describe the tokens allowed in the input stream. Regular expressions compactly represent patterns that the characters in lexemes might follow. For example, a NAME token might be any alphabetical

character or an underscore, followed by any number of instances of any alphanumeric character or an underscore. This could be represented compactly by the string `[a-zA-Z_][a-zA-z_0-9]*`. This means “any character a-z, A-Z, or `_`, then 0 or more of a-z, A-Z, `_` or 0-9”.

An overview of the token description is introduced below. The complete version is in Appendix B Lexical Conventions.

- White space: blanks, tabs, new-lines, and comments form white space. White space is ignored except when it separates adjacent tokens. All input between `(*` and `*)` or behind `*` is considered a comment.
- Identifier: an identifier consists of a letter or underscore, followed by any number of letters, underscores, and digits. Identifiers are case sensitive.
- Keywords: the following tokens are keywords and are reserved. They cannot be used as regular identifiers.

N_Module	N_Instantiation	N_SubsetOf
N_BeginModule	N_IdentLHS	N_SetOfAll
N_Extends	N_IdPrefix	N_FcnAppl
N_Body	N_IdPrefixElement	N_FcnConst
N_EndModule	N_OpArgs	N_SetOfFcns
N_VariableDeclaration	N_GenInfixOp	N_RcdConstructor
N_ParamDeclaration	N_GenPrefixOp	N_FieldVal
N_OperatorDefinition	N_GenPostfixOp	N_SetOfRcds
N_FunctionDefinition	N_GeneralId	N_FieldSet
N_Instance	N_OpApplication	N_Except
N_Assumption	N_PrefixExpr	N_ExceptSpec
N_ModuleDefinition	N_InfixExpr	N_ExceptComponent
N_Theorem	N_PostfixExpr	N_ActionExpr
N_ConsDecl	N_ParenExpr	N_IfThenElse
N_IdentDecl	N_BoundedQuant	N_Case
N_QuantBound	N_SetEnumerate	N_CaseArm
N_OtherArm	N_DisjItem	N_MaybeBound
N_LetIn	N_Number	N_ConjList
N_Tuple	N_RecordComponent	N_ConjItem
N_LetDefinitions	N_UnboundedOrBoundedChoose	
N_DisjList		

MODULE	ELSE	ASSUME
EXTENDS	CASE	ASSUMPTION
CONSTANT	OTHER	AXIOM
CONSTANTS	CHOOSE	LOCAL
VARIABLE	INSTANCE	DOMAIN
VARIABLES	WITH	ENABLED
THEOREM	LET	SUBSET
IF	IN	UNCHANGED
THEN	EXCEPT	UNION

- Number: a number can be an integer constant (including octal and hexadecimal constants) or a real constant. They can't be used as lvalue.
- Character constants: a character constant is defined exactly as in ANSI-C. It is of type char and cannot be used as lvalue.
- String constants: a string constant is defined exactly as in ANSI-C. It is of type string and cannot be used as lvalue.
- The null constant: the value null is of the implicit null type and denotes a null reference. null cannot be used as lvalue.

6.3 Syntax analysis

Given a TLA+ specification, it is first input into the TLA+ Front-End tool. After that, a flattened parse tree is generated. Figure 6.2 gives some examples of the TLA+ formulae and the corresponding flattened parse tree generated by the TLA+ Front-End tool. Because the parse tree is in the flattened text form and it is rather complicated, instead of translating directly from the flattened parse tree it is better to construct an AST and translate from the AST to SML automatically.

TLA+ expressions	Flattened parse tree
----- MODULE HourClock -----	N_BeginModule { ----- MODULE [1 1] HourClock [1 31] ----- [1 41] }
EXTENDS Naturals	N_Extends { EXTENDS [2 1] Naturals [2 9] }
VARIABLE hr	N_VariableDeclaration { VARIABLE [3 1] hr [3 10] }

HCini == hr \in (1 .. 12)	<pre> N_OperatorDefinition { N_IdentLHS { HCini [4 1]} == [4 8] N_InfixExpr { N_GeneralId { N_IdPrefix { } hr [4 12]} N_GenInfixOp { N_IdPrefix { } \in [4 15]} N_ParenExpr { ([4 19] N_InfixExpr { N_Number { 1 [4 20]} N_GenInfixOp { N_IdPrefix { } .. [4 22]} N_Number { 12 [4 25]}}) [4 27] } } } </pre>
THEOREM HC => []HCini	<pre> N_Theorem { THEOREM [8 1] N_InfixExpr { N_GeneralId { N_IdPrefix { } HC [8 10]} N_GenInfixOp { N_IdPrefix { } => [8 13]} N_PrefixExpr { N_GenPrefixOp { N_IdPrefix { } [] [8 16]} N_GeneralId { N_IdPrefix { } HCini [8 18] } } } } } </pre>

Figure 6.2: Flattened parse tree examples.

Writing the grammar rule (it is called GTLA grammar in this thesis) is the first step of AST construction. Because the grammar of the generated parse tree file is unavailable, some research work should be done here. Many TLA+ source files that contain almost all the TLA+ language grammar are input into the TLA+ Front-End tool; the corresponding flattened parse tree files are generated and analyzed to contribute to the GTLA grammar. Compare with the TLA+ language grammar, the GTLA language grammar is similar to it. This makes the research work much easier. The full grammar is given in Appendix C GTLA Grammar.

Based on the grammar, the AST is constructed with declarations, expressions and tokens. There are two files are input into the GTLA compiler: a TLA+ specification and a configuration file, so the AST consists of two declaration branches: one for the flattened parse tree generated from the TLA+ Front-End tool and the other for the configuration file. The flattened parse tree branch consists of an `s_beginmodule` declaration, an `s_extends` declaration and arbitrary `s_body` declarations, which consists of arbitrary other declarations. These declarations might contain expressions. The configuration branch consists of a `c_specdeclaration` which specifies the name of the specification and a `c_paramdeclaration` which specifies the instantiation of constants. Figure 6.3 gives a view of the structure of the AST. The arrow means “consisting of”; the broken arrow means “may consist of”.

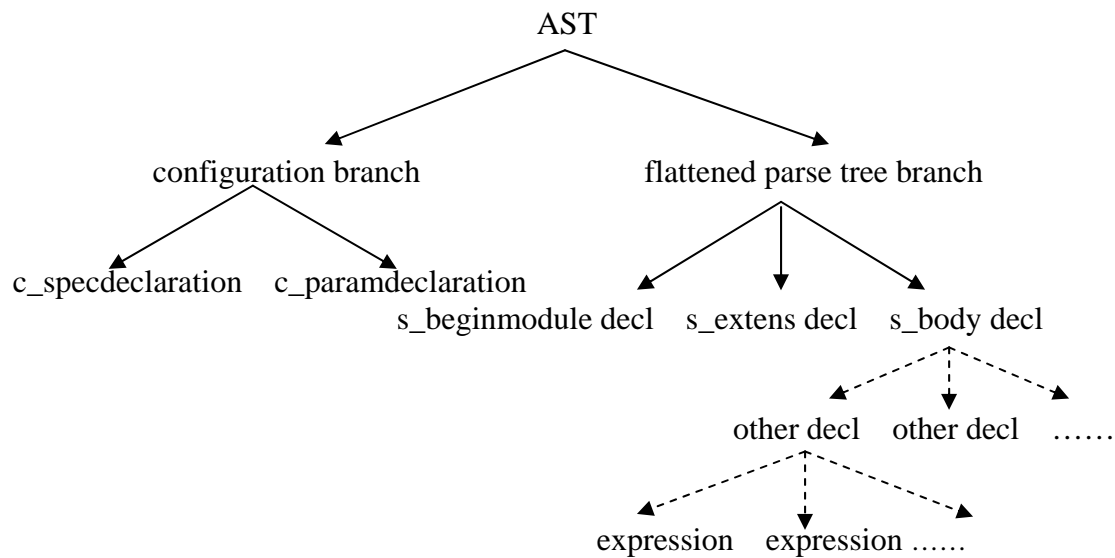


Figure 6.3: AST structure.

6.4 Code generation

Code generation generates the output SML file by specifying how each declaration, expression and token should be translated into SML code.

6.4.1 Declarations

A specification (ASTs) consists of declarations, which consists of a `sepc_declarations` (specification declaration) and a `conf_declarations` (configuration declaration). The `spec_declaration` consist of an `s_beginmodule` declaration, an `s_extends` declaration and arbitrary `s_body` declarations. Only the order of the `s_body` declarations is not important.

specification \rightarrow *declarations*
declarations \rightarrow *spec_declarations*
 conf_declarations
spec_declarations \rightarrow *N_Module* '{'
 s_begin module()
 s_extends()
 N_Body '{'
 [*s_body*
 | *AtLeast4*(' ') '[' *Number Number* '']*
 ']}'
 N_EndModule '{'
 AtLeast4('=') '[' *Number Number* '']
 ']}'
 ']}'
s_body \rightarrow *s_variabledeclaration*
 | *s_paramdeclaration*
 | [*local*]? [*s_operatordefinition*
 | *s_funtiondefinition*
 | *s_instance*
 | *s_moduledefinition*]
 | *s_assumption*
 | *s_theorem*
 | *declarations*

➤ Configuration declarations

conf_declarations \rightarrow [*c_specdeclaration*]? [*c_paramdeclaration*]*
c_paramdeclaration \rightarrow *CONSTANT*
 IDENTIFIER
 '='
 {'
 [*IDENTIFIER* | *NUMBER*]
 [, [*IDENTIFIER* | *NUMBER*]]*
 }'
c_specdeclaration \rightarrow *SPECIFICATION*
 IDENTIFIER

The *conf_declarations* consists of arbitrary *c_paramdeclaration* and an optional *c_specdeclaration* declaration. The *c_paramdeclaration* declaration states the instantiation of constants and the *c_specdeclaration* states the name of the

specification formula in TLA+ specifications. These declarations are only stored in the AST to provide additional information for GTLA compiler. They are not needed to be translated into SML code.

➤ **Begin module declaration**

$$s_begin\ module \rightarrow N_BeginModule \{ ' \\ \quad \textit{AtLeast4}(' - ') \ \textit{MODULE} \ [' \ \textit{Number} \ \textit{Number} \ '] ' \\ \quad \textit{IDENTIFIER} \ [' \ \textit{Number} \ \textit{Number} \ '] ' \\ \quad \textit{AtLeast4}(' - ') \ [' \ \textit{Number} \ \textit{Number} \ '] ' \\ \quad \} ' }$$

A specification contains only one begin module declaration, which declare the name of the specification. This name is stored in the global variable `modulename`, which is used for the output file's name.

➤ **Extends declaration**

$$s_extends \rightarrow N_Extends \{ ' \\ \quad \textit{EXTENDS} \ [' \ \textit{Number} \ \textit{Number} \ '] ' \\ \quad \textit{IDENTIFIER} \ [' \ \textit{Number} \ \textit{Number} \ '] ' \\ \quad [, \textit{IDENTIFIER} \ [' \ \textit{Number} \ \textit{Number} \ '] \ '] * \] ? \\ \quad \} ' }$$

A specification contains only one extends declaration, which declares the names of the modules that the specification extends. These names are stored in the global variable “`extend_args`”, which is used for the module inheritance.

➤ **Variable declarations**

$$s_variabledeclaration \rightarrow N_VariableDeclaration \{ ' \\ \quad \textit{[VARIABLE | VARIABLES]} \ [' \ \textit{Number} \ \textit{Number} \ '] ' \\ \quad \textit{IDENTIFIER} \ [' \ \textit{Number} \ \textit{Number} \ '] ' \\ \quad [, \textit{IDENTIFIER} \ [' \ \textit{Number} \ \textit{Number} \ '] \ '] * \\ \quad \} ' }$$

An `s_variabledeclaration` is used to declare a variable in the current scope. Multiple variables can be declared in one declaration. In the code generation phase, each variable is translated into the SML code “`val variable_name = ref initial_value;`”. Declaring a variable with a type operator `ref` is to create an updatable storage cell for the variable for the future update of it. The `initial_value` cannot get directly from the context due to the fact that TLA+ language is type-free and no initialization is needed

for a variable declaration. But SML language needs initialization of variable declarations. To solve this problem, the compiler must traverse the input file or the AST to get information for variables' initialization from the context. The AST traverse approach is used, see section 6.4.3. In a TLA+ specification, a variable is initialized directly in the Initial predicate or indirectly by calling a TypeInvariant predicate. In either one, a variable is initialized in such a form: “variable \in initialization”. Using AST traverse function to find such an expression for a variable in the AST and assign the initial value from the initialization. If the initialization is a number type, 0 is generated, if it is a string type, “ ” is generated, etc.

➤ Constant declarations

$$s_paramdeclaration \rightarrow N_ParamDeclaration \{ ' \\ N_ConsDecl \{ ' \\ [CONSTANT | CONSTANTS] \text{ 'Number Number' } ' \\ ' \\ s_idnetdecl[, s_identdecl]^* \\ ' \}$$

$$s_identdecl \rightarrow N_IdentDecl \{ ' \\ [IDENTIFIER \\ | IDENTIFIER \text{ '([, ' -]^*)' } ' \\ | PrefixOp \text{ '-' } ' \\ | \text{ '-' } InfixOp \text{ '-' } ' \\ | \text{ '-' } PostfixOp \\] \text{ 'Number Number' } ' \\ ' \}$$

An s_paramdeclaration is used to declare a constant (constants). Multiple constants can be declared in one declaration. In the code generation phase, each constant is translated into the SML code “val constant_name = initial_value;”. The initial_value should be got from the configuration file. The configuration file provides extra information of the specification, properties to be checked (such as invariants), constants declarations, etc. All the information is contained in the configuration branch of the AST. The AST traverse function is called to find the constant declaration in the AST and the corresponding initial value given a constant.

➤ Operator declaration

$$s_operatordefinition \rightarrow N_OperatorDefinition \{ ' \\ [s_identlhs \\ | s_prefixop IDENTIFIER$$

```

| IDENTIFIER s_inf ixop IDENTIFIER
| IDENTIFIER s_postfixop ]
==
Expressions
}'

```

The operator declaration is the most important declaration that forms the body of a specification. It is usually used to specify various TLA+ formulae. The left side of the operator “==” declares the name and parameter(s) of a formula, the right side declares its body, which is made up of various expressions. The operator declaration is also used as part of some expressions as a variable definition, for example “x == y + 1”. In the code generation phase, because a formula is a Boolean expression in TLA+ language, if an operator declaration specifies an action, it is translated into a Boolean function in SML of the form “fun operatordef_name (parameters...) == expressions”, where the operatordef_name and parameters can be got directly from the specification while the expressions are generated in the expression layer of AST with a Boolean value returned. If an operator declaration specifies a variable, it is translated into a variable in SML of the form “val operatordef_name == expressions”.

Whether an operator declaration specifies an action or a variable is differentiated by two points. If an operator declaration is the initial predicate or a (sub) next action, it is translated into a function; otherwise if the left side of “==” operator contains parameter(s), it is translated into a function; otherwise it is translated into a variable.

➤ Function declaration

```

s_functiondefinition → N_FunctionDefinition '{'
    IDENTIFIER
    '[' s_quantbound
    [, s_quantbound ] * ']'
    ==
    Expressions
    }'

s_quantbound → N_QuantBound '{'
    Tuple
    | IDENTIFIER [, IDENTIFIER ] *
    "\" in "
    Expressions
    }'

```

A function declaration declares a function in the specification. The left side of the operator “==” declares the name, parameter(s) and its (their) domain(s) of the function, the right side declared the body of the function. Because the domain of the parameter is concerned, this is translated in a special way. The following gives a

typical example of the translation.

The declaration written in TLA+:

$$f[t \text{ in } Nat] == t + 1$$

The translated SML code:

$$\text{fun } f(t) = \text{if } (List.exists(fn x \Rightarrow t) Nat) \text{ then } (t + 1) \text{ else } (t);$$

Only when the given parameter t belongs to the domain Nat , its value can be updated by the right side expression “ $t+1$ ”.

➤ Theorem declaration

$$\begin{aligned} s_theorem &\rightarrow N_Theorem'\{ \\ &\quad THEOREM \text{ '[' } Number \text{ Number ']' } \\ &\quad Expressions \\ &\quad \}' \end{aligned}$$

A theorem declaration declares a formula that must be true if the module is correct. It can be viewed as a Boolean function that if the expression is true then returns true else returns false. In SML it can be directly translated into a function “fun theorem () = Expressions”.

➤ Assumption declaration

$$\begin{aligned} s_assumption &\rightarrow N_Assumption'\{ \\ &\quad [ASSUME | ASSUMPTION | AXIOM] \\ &\quad Expressions \text{ '[' } Number \text{ Number ']' } \\ &\quad \}' \end{aligned}$$

An assumption declaration is similar to a theorem declaration. It has no effect on any definitions made in the module, but it may be taken as a hypothesis when proving any theorems asserted in the module. In other words, a module asserts that its assumptions imply its theorems. Additionally an assume statement in a TLA+ specification should be used only to make assumptions about constants. For example, “ $ASSUME(N \in Nat) \wedge (N > 0)$ ” is an assumption about constant N . As the theorem declaration, this can be translated into a Boolean function “fun assume () = Expressions”.

➤ Instance declaration and Module declaration

Instance declaration and Module declaration are used to state an object in another module when the current module inherits from other modules. The inheritance is a

difficult problem to be solved in the future work. It is described in chapter 11.

6.4.2 Expressions

A declaration may contain arbitrarily various expressions. There are in total 29 expressions in the TLA+ grammar. This number excludes a couple of expressions describing unbounded cases, such as “ $\backslash A \ x: x > 8$ ”. Other expressions are explained below.

GTLA grammar	TLA+ examples	SML translation
$s_generalid \rightarrow$ $N_GeneralId \{'$ $s_idprefix$ $IDENTIFIER$ $\}'$	id (identifier) $A ! id$ (instance identifier, excluded)	id or !id (with reference) Refer to A after this table
$s_opapplication \rightarrow$ $N_OpApplication \{'$ $s_generalid$ $s_op \ arg \ s$ $\}'$	$A!Op(x+1,y)$ (instance application, excluded)	Refer to A after this table
$s_prefix \ expr \rightarrow$ $N_PrefixExpr \{'$ $s_genprefixop$ $s_expression$ $\}'$	\sim or $\backslash not$ or $\backslash neg \ P$ $SUBSET \ S$ (unfinished)	not P Refer to B after this table
$s_infix \ expr \rightarrow$ $N_InfixExpr \{'$ $s_expression$ $s_geninfixop$ $s_expression$ $\}'$	a+b	a+b
$s_postfix \ expr \rightarrow$ $N_PostfixExpr \{'$ $s_expression$ $s_genpostfixop$ $\}'$	x'	x

$s_paren\ expr \rightarrow$ $N_ParenExpr\{'$ $\('s_expression')'$ $\}'$	$(x+1)$ $(1..3)$	$(x+1)$ $[1,2,3]$
$s_boundedquant \rightarrow$ $N_BoundedQuant\{'$ $["\ A" "\ E"]$ $s_quantbound$ $[,s_quantbound]^*$ $':s_expression$ $\}'$	$\forall x \in S : F(x)$ $\forall x \in s, < y, z > \in T$ $: F(x, y, z)$ (unfinished)	$List.all(fn\ x=>F(x))\ S$ Refer to C after this table
Unbounded Quantifier	$\exists x, y : x + y > 0$ (cannot be solved)	Refer to D after this table
$s_unboundedorboundedchoose \rightarrow$ $N_UnBoundedOrBoundedChoose$ $CHOOSE$ $[IDENTIFIER Tuple]$ $["\ in" s_expression]?$ $':s_expression$ $\}'$	$x \in S : F(x)$	$List.nth(List.filter(fn$ $x \Rightarrow F(x))S$ $Random.randRange(0,$ $length(List.filter(fn$ $x \Rightarrow F(x))S)-1\ rand)$
$s_setenumerate \rightarrow$ $N_SetEnumerate\{'$ $[s_expression$ $[,s_expression]^*]?$ $\}'$	$\{1,2,2+2\}$	$[1,2,2+2]$
$s_subsetof \rightarrow$ $N_SubsetOf\{'$ $[IDENTIFIER Tuple]$ $"\ in" s_expression$ $':s_expression$ $\}'$	$\{x \in Nat : x > 0\}$	$List.filter(fn$ $x \Rightarrow x > 0)Nat$
$s_setofall \rightarrow$ $N_SetOfAll\{'$ $\{'s_expression':'$ $s_quantbound$ $[,s_quantbound]^*\}'$ $\}'$	$e : x \in S$ $F(x, y, z) : x, y \in S, z \in T$ (unfinished)	$map(fn\ x=>e)\ S$ Refer to C after this table

$s_fcnappl \rightarrow$ $N_FcnAppl \{'$ $s_expression$ $'[s_expression$ $[,s_expression]^*']'$ $\}'$	$f[i+1, j]$	$f(i+1, j)$
$s_fcnconst \rightarrow$ $N_FcnConst \{'$ $'[s_quantbound$ $[,s_quantbound]^*$ $" ->" s_expression']'$ $\}'$	$[i \in S \mapsto F(i)]$ $[i, j \in S, < p, q > \in T$ $\mapsto F(i, j, p, q)]$	<p>Fun f(i) = if (List.exists (fn x=>x=i) S) then F(i) else ?</p> <p>Fun f(i,j,(p,q)) = if (List.exists (fn x=>x=i) S) andalso (List.exists (fn x=>x=j) S) andalso (List.exists (fn x=>x={p,q}) T) then F(i,j,p,q) else ?</p> <p>(? decides on the type)</p>
$s_setoffcns \rightarrow$ $N_SetOfFcns \{'$ $'[s_expression$ $" \rightarrow " s_expression']'$ $\}'$	$[S \rightarrow T]$ (unfinished due to the domain of f is used.)	Refer to E after this table
$s_rcdconstructor \rightarrow$ $N_RcdConstructor \{'$ $'[s_fieldval$ $[,s_fieldval]^*']'$ $\}'$	$[a \mapsto x+1, b \mapsto y]$	$\{a = ref \quad x+1, b = ref \quad y\}$
$s_setofrcds \rightarrow$ $N_SetOfRcds \{'$ $'[s_fieldset$ $[,s_fieldset]^*']'$ $\}'$	$[a : Nat, b : S \cup T]$	$\{a = ref \quad List.nth(Nat, Random.randRange (0, length Nat - 1) rand) , b = ref \dots\}$
$s_except \rightarrow$ $N_Except \{'$ $'[s_expression$ $EXCEPT$ $s_exceptspec']'$ $\}'$	$[f$ $EXCEPT$ $!.r = 4, !.y = 5]$	$(\#r \ f) := 4;$ $(\#y \ f) := 5$

<i>N_Tuple</i> '{' " << " <i>s_expression</i> [, <i>s_expression</i>] * " >> " ' }	< 1, 2, 3 >	(1, 2, 3)
<i>s_actionexpr</i> → <i>N_ActionExpr</i> '{' [' <i>s_expression</i> '] ' _ <i>s_expression</i> ' }	[<i>A ∨ B</i>] _ < <i>x</i> , <i>y</i> >	<i>A</i> () orelse <i>B</i> ()
<i>s_ifthenelse</i> → <i>N_IfThenElse</i> '{' <i>IF</i> <i>s_expression</i> <i>THEN</i> <i>s_expression</i> <i>ELSE</i> <i>s_expression</i> ' }	IF <i>p</i> THEN <i>A</i> else <i>B</i>	If <i>p</i> then <i>A</i> else <i>B</i>
<i>N_Case</i> '{' <i>CASE</i> <i>s_casearm</i> [[<i>s_casearm</i>] * [<i>s_otherarm</i> ' }	<i>CASE</i> <i>p1</i> → <i>e1</i> [<i>p2</i> → <i>e2</i> [<i>OTHER</i> → <i>e3</i>	<i>if</i> (<i>p1</i>) <i>then</i> (<i>e1</i>) <i>else if</i> (<i>p2</i>) <i>then</i> (<i>e2</i>) <i>else</i> (<i>e3</i>)
<i>s_letin</i> → <i>N_LetIn</i> '{' <i>LET</i> <i>s_letdefinitions</i> <i>IN</i> <i>s_expression</i> ' }	<i>LET</i> <i>x</i> == <i>y</i> + 1 <i>IN</i> <i>x</i> + 1	<i>let</i> <i>x</i> == <i>y</i> + 1 <i>in</i> <i>x</i> + 1 <i>end</i>
<i>s_conjlist</i> → <i>N_ConjList</i> '{' [' ^ ' <i>s_expression</i>] + ' }	$\wedge x = 1$ $\wedge y = 2$	<i>x</i> = 1 <i>andalso</i> <i>y</i> = 2 (not so simple in action definition)
<i>s_disjlist</i> → <i>N_DisjList</i> '{' [' ∨ ' <i>s_expression</i>] + ' }	$\vee x = 1$ $\vee y = 2$	<i>x</i> = 1 <i>orelse</i> <i>y</i> = 2
Number	09001	09001
String	“foo”	“foo”
@	@	Using AST traverse

The above table introduced various expressions of the GTLA grammar and the corresponding SML code expected. From the table we can see some expressions can't be translated into the SML code successfully. These are classified and explained below.

- A. The translation of two expressions that are used to specify instance identifiers and instance applications can't be solved. The instance declaration is viewed as part of the future work and excluded in this project.
- B. The prefix expression is of the form "prefix_operator expression". There are several kinds of prefix operators and not all of them can be translated into SML code.

What can: " \sim ", " \backslash not", " \backslash neg" expr are translated into "not expr";

"[]", "UNCHANGED" expr are translated into expr;

What can't: "SUBSET" expr, which specifies the set of subsets of expr;

"UNION" expr, which specifies the union of all elements of expr;

"DOMAIN" expr, which specifies the domain of expr;

These three expressions can't be translated into SML code directly, which were also stated in the literature survey.

Similarly, there are various infix operators and postfix operators (refer to Appendix C GTLA Grammar) in the infix and postfix expressions and not all of them are translated into SML code. The basic operators like +, -, *, /, %, etc. have the same meaning in SML language and can be directly translated to SML code; others like :>, <:, \$\$, etc. are belong to special modules and excluded in this project.

- C. There are three kinds of expressions can't be translated to SML code completely. Take the s_boundedquant expression as an example, in terms of the expression that contains only one s_quantbound expression, like $\forall x \in S : F(x)$, it can be translated into SML code of the form "List.all(fn x=>F(x)) S".

But in terms of the expression contains multiple s_quantbound expressions, like $\forall x \in S, < y, z > \in T : F(x, y, z)$, due to the limitation of SML language, it can't be directly translated into SML code.

- D. The unbounded quantifier (such as " \backslash E x: P") is excluded in this project.
- E. The meaning of $[S \rightarrow T]$ is:

IsAFcn (f) \wedge (S = DOMAIN f) \wedge ($\forall x \in S : f[x] \in T$). Because the DOMAIN f can't be solved in SML code, this kind of expression can't be translated into SML code.

6.4.3 AST Traverse

In the code generation phase, some AST node can't be translated into the target language directly due to lack of information, so an extra special AST traverse is needed in order to gather information.

The AST traverse is constructed with similar structures as the AST. The traverse walks along the declarations and expressions. While looking for a given token, it traverses the whole AST to find all declarations containing this token, and it will return a declaration list. Figure 6.4 shows how the AST traverse works.

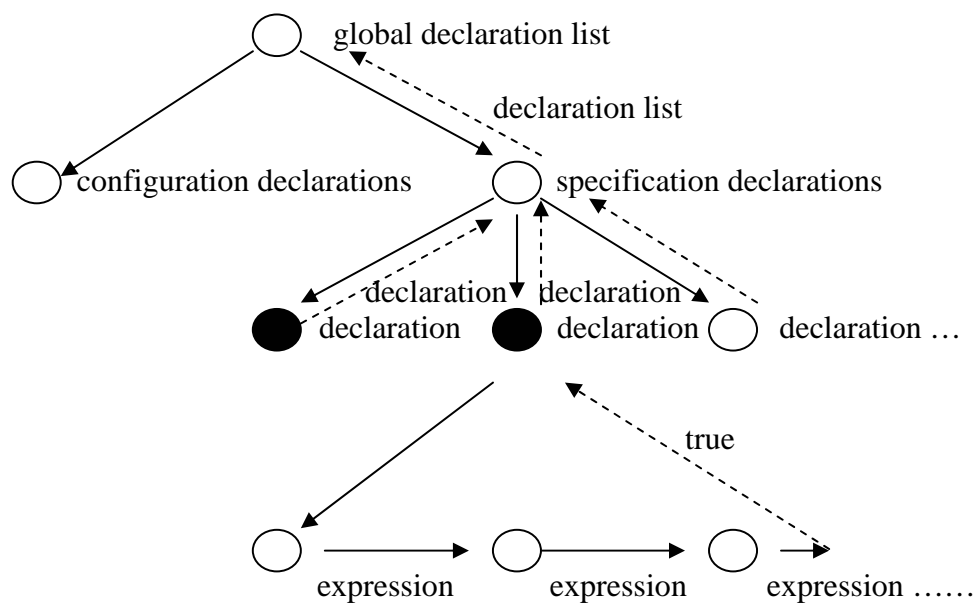


Figure 6.4: How AST Traverse works.

The arrow means traverse path, the broken arrow means return path, the black node means the node receives a true value from its children nodes. The AST Traverse starts at the global declaration list, which has two branches, configuration declarations that provide extra information of constants and properties, etc. and specification declarations. The AST traverse can choose either part to traverse. If it wants to find a token in the specification, it traverses the specification declaration branch processing all the declaration sequentially. In each declaration, if the declaration contains this token, the declaration is returned, otherwise if the declaration has expressions, the expressions are traversed and if any expression contains this token, the Boolean value true is returned to this declaration, otherwise false is returned. If a declaration receives a true value from any expression, the declaration is returned, otherwise, nothing is returned. After traversing all the declarations, a declaration list is returned that contains all the declarations containing the given token. This declaration list will contain all the missing information.

Normally the AST traverse is used in three cases.

- Looking for a constant

When translate a constant declaration in the TLA+ specification into SML code, in order to initialize this constant, the AST Traverse is used to find the initial value of it from the configuration declarations branch in the AST. Because normally in a TLA+ system, the concrete value of the constants are given in the configuration specifications.

- Looking for a variable

Similarly, when translate a variable declaration in the TLA+ specification into SML code, in order to initialize this variable; the AST Traverse is used to find the type of it from the specification declarations branch in the AST. The initial value is determined according to its type.

- Looking for '@'

Symbol '@' always appears in a record expression to represent the corresponding part in the original variable. For example, in the expression [chan EXCEPT !.val = 1 - @], '@' represents chan.val. In the SML code it should be translated to # val chan. The AST Traverse is used to find information about it.

6.4.4 Types

Type checking is an important part of semantic analysis. In the GTLA compiler construction part, a TLA+ specification is first input into the TLA+ Front-End tool. After that the generated JCF file (flattened parse tree) is input into the GTLA compiler as the source file, which is assumed absolutely correct after first parsing and semantic checking by the TLA+ Front-End tool. But compared with TLA+, a type-free language, SML has strict requirement of types. This is a problem for code generation. Given a variable, how to recognize the type of it and initialize it? Is type checking technique needed to keep the correctness of the types of the source file?

To solve this problem, we must understand the TLA+ grammar and the specification rules well. A TLA+ specification declares variables in the form "VARIABLE name" or "VARIABLES name1, name2, ...". These variables are initialized in the initial predicate formula. There are two forms of the initial predicate formula. In terms of a single variable, the initial predicate formula is in the form as following:

Init == Name \in Set or Init == Name == Expression or Init == TypeInvariant

"Name" represents the name of the variable and "Set" represents its domain. The variable Set can be either a constant or an expression. TypeInvariant is another action formula that declares the domain of all variables, in this case, we should traverse the AST to find TypeInvariant formula and do initialization from it.

In terms of many variables, the initial predicate formula is in the form as following:

$\text{Init} == \wedge \text{Name1} \setminus \text{in Set1}$

$\wedge \text{Name2} == \text{Set2}$

$\wedge \text{TypeInvariant}$

$\wedge \dots$

All the initializations of variables are combined in a conjunction list formula, whose items could be of the form “Name \setminus in Set” or of the form “TypeInvariant” or of the form “Name j = Expr”, where “Set” could be either a constant or an expression and “Expr” could be either a variable that has been already initialized or an expression (for example, Name j = 0).

So given a variable, we first find the initial predicate in the AST, then for each variable we find the correct type of it according to the set or other variable (expression) in the right side of some operator (“ \setminus in” or “=”). There are in total four cases, see the following table.

Initial predicate	How to find the type
$V_j == V_i$	according to V_i 's initialization
$V \setminus \text{in ConstantSet}$	according to the declaration of ConstantSet by traversing the AST
$V \setminus \text{in Expression}$	according to the expression
$V == \text{Expression}$	According to the expression
TypeInvariant	Traverse the AST to get TypeInvariant formula, according to which do initialization

According to the GTLA grammar, we can find all the expression cases in order to find expected initialization of a variable, see the following table.

Expression example	Expected initialization
$hr \in (1..12)$	0
$hr \in \{0,1\}$	0
$x \in \{y \in \text{Nat} : y > 0\}$	0
$chan \in [\text{val} : \text{Data}, \text{rdy} : \{0,1\}, \text{ack} : \{0,1\}]$	$[\text{val} := \text{ref } ?, \text{rdy} := \text{ref } 0, \text{ack} := \text{ref } 0]$
$chan \in [\text{val} \mapsto 0, \text{rdy} \mapsto 1, \text{ack} \mapsto 0]$	$[\text{val} := \text{ref } 0, \text{rdy} := \text{ref } 0, \text{ack} := \text{ref } 0]$
$mem \in [\text{Adr} \rightarrow \text{Val}]$	0!mem = List.nth(Val, Random.randRange(0,(length Val)-1) rand);), etc... (mem [0..Adr] = any of Val)
$\text{ctl} == [p \in \text{Proc} \mapsto \text{"rdy"}]$	$p ! \text{ctl} = \text{"rdy"}, \text{etc.}$ (ctl[p] = “rdy” for any $p \in \text{Proc}$)

$x == \langle 0, "bo" \rangle$	$(0, "")$
String	“ ”
Number	0

In the above table, the initialization value is found and assigned to a variable just as a variable declaration. In the later initial predicate action, a variable is re-assigned a value according to the initial predicate. For example, from the initial predicate “ $hr \in (1..12)$ ” the variable *hr* is declared as “*val hr = ref 0;*”, while in the initial predicate variable *hr* is re-assigned as “*hr := List.nth([1,2,3,4,5,6,7,8,9,10,11,12], Random.randRange(0,(length [1,2,3,4,5,6,7,8,9,10,11,12])-1) rand);*” (a random number between 1 and 12).

Additionally, SML doesn’t require type declaration of a variable, but initialization must be provided to serve as a type declaration.

Only if we can guarantee the correctness of this rule, the correctness of the type and initialization of a variable can be guaranteed and type checking can be ignored in this project. Even though, just like any compiler, a type checking mechanism can improve the security and correctness of the compilation. Considering of the limited time and huge work needed to be done in type checking, it is viewed as a future work after examining the test result using the current project. Refer to chapter 6 and 7.

6.5 Examples

In this section, two examples will be presented to show what kind of SML file will be generated by the GTLA translator given a TLA+ source file. The first example “HourClock” specifies a digital clock that displays hour cycles through the values 1 through 12. The second example “AsynchInterface” specifies an interface for transmitting data between asynchronous devices: a sender and a receiver. These examples can be found in Lamport’s book “Specifying Systems” [1].

Example 1 – HourClock

- TLA+ source file:

```

----- MODULE HourClock -----
EXTENDS Naturals
VARIABLE hr
HCini == hr \in (1 .. 12)
HCnxt == hr' = IF hr # 13 THEN hr + 1 ELSE 1
HC == HCini /\ [][HCnxt]_hr
-----
THEOREM HC => []HCini
=====

```


- SML file:

```

val rand = Random.rand(0,1);
val hr = ref 0;
fun HCini() = (List.exists(fn listexist => listexist = (!hr))
[1,2,3,4,5,6,7,8,9,10,11,12];true);
fun HCnxt() = (hr := (if(!hr)<>13) then ((!hr)+1) else (1));true);
fun HC() = ((HCini() andalso HCnxt()));
fun theorem() = not(HC()) orelse HCini();

```

Example 2 – AsyncnInterface

- TLA+ source file:

```

----- MODULE AsyncnInterface -----
EXTENDS Naturals
CONSTANT Data
VARIABLES val, rdy, ack

TypeInvariant ==  $\wedge$  val  $\in$  Data
                 $\wedge$  rdy  $\in$  {0, 1}
                 $\wedge$  ack  $\in$  {0, 1}

-----

Init ==  $\wedge$  val  $\in$  Data
         $\wedge$  rdy  $\in$  {0, 1}
         $\wedge$  ack = rdy

Send ==  $\wedge$  rdy = ack
         $\wedge$  val'  $\in$  Data
         $\wedge$  rdy' = 1 - rdy
         $\wedge$  UNCHANGED ack

Rcv ==  $\wedge$  rdy  $\neq$  ack
       $\wedge$  ack' = 1 - ack
       $\wedge$  UNCHANGED <<val, rdy>>

Next == Send  $\vee$  Rcv

Spec == Init  $\wedge$   $\square$ [Next]-<<val, rdy, ack>>

-----
THEOREM Spec  $\Rightarrow$   $\square$ TypeInvariant
=====

```

- SML file:

```

val rand = Random.rand(0,1);
val Data = [1];
val val = ref 0;

```

```

val rdy = ref 0;
val ack = ref 0;
fun TypeInvariant() = ((List.exists(fn listexist => listexist = (!val)) Data andalso
List.exists(fn listexist => listexist = (!rdy)) [0,1] andalso
List.exists(fn listexist => listexist = (!ack)) [0,1]));
fun Init() = ((List.exists(fn listexist => listexist = (!val)) Data andalso
List.exists(fn listexist => listexist = (!rdy)) [0,1] andalso
(!ack) = (!rdy)));
fun Send() = (if (!rdy)=(!ack) then(val :=
List.nth(Data,Random.randRange(0,(length Data)-1) rand);
rdy := (1-(!rdy));true) else (false));
fun Rcv() = (if (!rdy)<>(!ack) then(ack := (1-(!ack));true) else (false));
fun Next() = (Send() orelse Rcv());
fun Spec() = ((Init() andalso Next()));
fun theorem() = not(Spec()) orelse TypeInvariant();

```

6.6 Improved SML forms

In the above paragraphs, the GTLA compiler construction work has been introduced. Now we have got a general idea about what kind of SML code will be generated given some declarations or expressions, even some TLA+ files. But in terms of this project (the GTLA system), some additional work should be done with the code generation phase in order to assist the simulation mechanism. In this section the expected SML code will be discussed.

Besides the direct translation from the TLA+ specification to SML code, there are some extra variables and functions need to be added into the generated SML file to assist the execution of the simulator. They are introduced below.

The additional variables are:

```

val rand = Random.rand(0,1);

val nextactionnum = [0,1];

val counter = ref 0;

```

The variable rand is used as a parameter to implement random selection. For example, see the following code in section 6.5 Example 2 – AsyncnInterface. The variable val is assigned a value that is randomly chosen from the set Data.

```

fun Send() = (if (!rdy)=(!ack) then(val := List.nth(Data,Random.randRange(0,(length
Data)-1) rand);

```

The variable nextactionnum is initialized according to the number of next actions; it is used to randomly choose one next action to execute in the random mode. While the variable counter is used to different between the initial predicate and next actions. Refer to the example at the end of this section.

The additional functions are:

The writeV() function: being used to write the current variables' values into the Variables.txt file.

The writeAT() function: being used to add the next action to be executed into the ActionTrace.txt file.

The writeEA() function: being used to write the enabled actions of the next step into the EnabledAction.txt file.

The VariablesOutput() function: being used to write all the variables' values in with their names into the Variables.txt file by calling writeV() function.

Additionally, what kind of forms does the initial predicate, next actions, the specification predicate have in the SML file?

- The initial predicate: a “VariablesOutput ();” statement is added into the initial predicate function to generate initial variables' values.
- The next action: each next action is translated into two functions, one Guard function that specifies the guard to execute the action, if the guard is satisfied, a true value is returned, otherwise a false value is returned. The other one is the action itself, but in this function two extra statements are added. One is the “VariablesOutput ()”, the other is “writeAT “Send”” for example, to add the name to the action trace file if this action is executed.
- An extra function GuidNext () is defined to use the Guard function of each next action to estimate which action is enabled in the current state and calls the writeEA () function to write them into the enabled action file.
- An extra function RandNext () is defined to simulate the random mode. In this function, a random action among the enabled actions is chosen.
- The specification predicate: the specification function execute the initial predicate at the first step and calls the RandNext () function to execute a random chosen enabled action at the following steps.

Given the source TLA+ specification in section 6.5 Example 2 – AsyncnInterface. The expected SML code is as below.

```
val rand = Random.rand(0,1);
val nanum = [0,1];
val counter = ref 0;
val Data = [0,1];
val Val = ref 0;
val rdy = ref 0;
val ack = ref 0;
fun writeV w = let val text = TextIO.openOut ("c:/Workspace/Simulator/
                Variables.txt")
                in (TextIO.output(text,w^"\n"); TextIO.closeOut text)
```

```

end;
fun writeAT w = let val text = TextIO.openAppend("c:/WorkSpace/Simulator/
              ActionTrace.dot ")
              in (TextIO.output(text,w^"\n"); TextIO.closeOut text)
              end;
fun writeEA w = let val text = TextIO.openOut( "c:/WorkSpace/Simulator/
              EnabledAction.txt")
              in (TextIO.output(text,w^"\n"); TextIO.closeOut text)
              end;
fun VariablesOutput() = let val v = ref ""
              in (v := (!v) ^ "Val = "^(Int.toString(!Val))^"\n";
                  v := (!v) ^ "ack = "^(Int.toString(!ack))^"\n";
                  v := (!v) ^ "rdy = "^(Int.toString(!rdy))^"\n";
                  writeV (!v))
              end;
val TypeInvariant = ((List.exists(fn listexist => listexist = (!Val)) Data andalso
              List.exists(fn listexist => listexist = (!rdy)) [0,1] andalso
              List.exists(fn listexist => listexist = (!ack)) [0,1]));
fun Init() = (VariablesOutput();(List.exists(fn listexist => listexist = (!Val)) Data
              andalso List.exists(fn listexist => listexist = (!rdy)) [0,1] andalso
              (!ack) = (!rdy)));
fun GSend() = if (!rdy)=(!ack) then true else false;
fun Send() = (if GSend() then(Val := List.nth(Data,Random.randRange(0,(length
              Data)-1) rand); rdy := (1-(!rdy)); writeAT "Send"; VariablesOutput())
              else ());
fun GRcv() = if (!rdy)<>(!ack) then true else false;
fun Rcv() = (if GRcv() then(ack := (1-(!ack)));writeAT "Rcv";VariablesOutput())
              else ());
fun GuidNext() = let val ea = ref ""
              in (if GSend() then ea := (!ea) ^ "Send " else ();
                  if GRcv() then ea := (!ea) ^ "Rcv " else ();
                  writeEA (!ea))
              end;
fun RandNext() = let val index = List.nth(nanum,Random.randRange(0,(length
              nanum)-1) rand)
              in if index = 0 andalso GSend() then (Send();true)
                  else if index = 1 andalso GRcv() then (Rcv();true)
                  else RandNext()
              end;
fun RandSpec() = if (!counter)=0 then (counter:=(!counter)+1;Init()) else
              (counter:=(!counter)+1;RandNext());
fun theorem() = not(RandSpec()) orelse TypeInvariant;

```

6.7 Conclusion

This chapter discussed how to translate the TLA+ specification into SML code

automatically. Only a subset of the TLA+ language-- the elementary operators are considered, other modules and some unsolved work are treated as part of the future work. They are:

- Liveness and fairness
- Parameterized instantiation (instance and module declarations)
- Unbounded expressions and several other expressions (see page 56-61)

After introducing the GTLA compiler construction work, two examples were presented to give out an overview of the translation. After that, the expected SML code is discussed. To assist the simulator of the GTLA system, some additional information (variables and functions) need to be added to the SML code and an example is given.

After constructing the translator, the TLA+ specification can be translated into the SML code automatically. With this executable SML code, the GTLA system can make the TLA+ specification being executed through the simulator.

Chapter 7

Designing the GTLA system

The first step of any development process is to design the application. This application (GTLA) consists of three important mechanisms: a system editor, a simulator and a verifier. They are integrated into a graphical user interface, which provides all the functionality the GTLA system requires. An overview of the design of the GTLA system will be introduced in section 7.1. The functionality required for the GTLA system will be described in the following paragraphs. It serves as a complete guide to exactly what functionality will be included in the GTLA system.

7.1 Overview of the GTLA system

The GTLA system is a tool providing system editing, simulation and verification functionality for the TLA+ user through a graphical user interface. The GUI, the TLA Front-End and the translator, the GraphViz tool and the graph generator algorithms, the SML/NJ compiler, and the TLC model checker are integrated into the GTLA environment by using the Java runtime package in order to implement all the functionality the GTLA system required.

The GUI consists of two SWT widgets: a menu that contains all the functionality and a tabbed folder that provides the three important mechanisms: system editor, simulator and verifier. Several other tools are called by the simulator and verifier to implement the functionality they required. The relationship among these tools is described below, see Figure 7.1.

- The user loads the TLA specification file and configuration file into the GTLA system by opening files from the GUI (Menu→File→"Open TLA Spec..."/"Open Configuration Spec..."). These two files are stored into a temporal directory called "WorkSpace" for later use.
- When the user presses the "Simulator" tab item to enter the simulator page, the GTLA system calls the TLA Front-End and translator to translate the source files in the temporal directory into the SML file.
- Then the GTLA system calls the SML/NJ compiler and loads in the generated SML file.

- When the user runs the simulator manually (by clicking the “Next” button) or automatically (by clicking the “Random” button), the GTLA system writes corresponding commands to the SML/NJ compiler that executes the SML file and generate some useful output files (like current variable values, action trace, etc.) into the temporal directory. For example, at one step, the user wants the simulator execute the “Send” action in the next step, after the user chose the “Send” action in the enabled action list and pressed the “Next” button, the GTLA system writes the command “Send();” to the SML/NJ compiler which is running the SML file.
- At the same time, the graph generators are called to generate the simulation dot files given some of those information files (those contains information of action trace and initial actions) and the GraphViz tool is called to generate simulation graphs from the dot files. These generated graphs and some information files are fed back and displayed in the simulator page of the GUI.
- When the user presses the “Verifier” tab item, the GTLA system calls the TLC model checker to check the source files that are stored in the temp directory in the model-checking mode, and display the generated message of TLC model checker in the verifier window.

The details of the implementation and structures of the GUI and simulator are discussed in chapter 4 and 5. The TLC model checker is used as the verifier, which is integrated into the GTLA system.

7.2 Designing the GUI

The graphical user interface (GUI) consists of two parts: a menu and a tabbed folder. The menu consists of a “File” list, an “Edit” list and a “Help” list. The tabbed folder consists of three pages: a system editor, a simulator and a verifier. It is developed by a Java GUI builder “Swing Designer”, which is a plug-in of Eclipse. Besides the GUI construction, the TLC model checker, the SML/NJ compiler, the translator that translates TLA+ language into SML language, and the simulator are integrated into the GTLA environment. With this GUI, the user can run the TLC model checker through a graphical user interface. The requirements and design of the GUI are introduced below.

7.2.1 The Menu and Tabbed Folder

A menu bar will be created to allow the user to access all functionality. Most applications will have a minimum of File, Edit and Help as high-level menu choices, with submenu items varying depending upon the application’s functionality. The menu will have the following functionality:

- It will have three high-level menu items: File, Edit and Help
- The menu item File will contain the menu items related to some of the GTLA system's functionality, like "open a file", "save a file", "check syntax", "exit the GTLA system", etc.
- The menu item Edit will contain the menu items Undo and Redo, which are related to the undo/redo mechanism
- The menu item Help will contain the menu item About, which provides information about the GTLA system

A tabbed folder object will be created to display three mechanisms (system editor, simulator and verifier) in one window using a notebook format, with tabs allowing easy access to widgets on each page of the notebook. It should provide the following functionality:

- It will have three tabs: System Editor, Simulator and Verifier
- Each page contains the widgets each mechanism requires
- By clicking each tab the user can access the widgets on each page

The windows of the menu should look like Figure 7.2. They are the File menu, the Edit menu and the Help menu. The window of the tabbed folder is contained in the main window, see Figure 7.3.

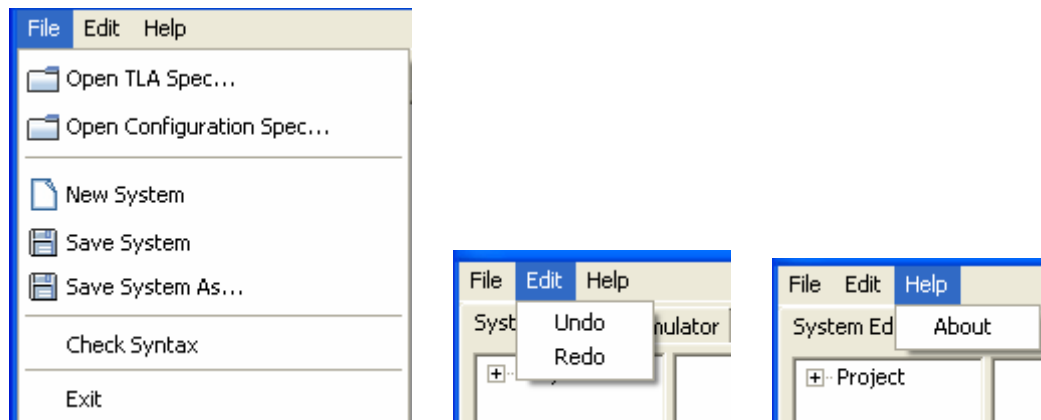


Figure 7.2: The Menu Bar.

7.2.2 The System Editor

The System Editor will have the following functionality:

- The ability to edit text

- The ability to open files, with the contents being displayed in the text-editing area of the GUI
- The ability to edit both the TLA specification file and the configuration file and switch between them
- The ability to save the contents of the text-editing area into a file for later use
- The ability to check the syntax of the opened TLA system
- The ability to provide undo/redo mechanism
- The ability to display an About window with information about the GTLA system

The main window for the system editor should look like Figure 7.3. The left window is a project tree with two children: a “TLA Spec” and a “CFG Spec”. Due to the fact that a TLA system must have two files – one TLA specification file (“TLA Spec”) and one configuration file (“CFG Spec”), to run a TLA system the user must load both files into the GTLA system. With this project tree, the user can switch between the TLA specification file and the configuration file. The right window is the text editor that is used to display and edit the contents of either the TLA specification file or the configuration file.

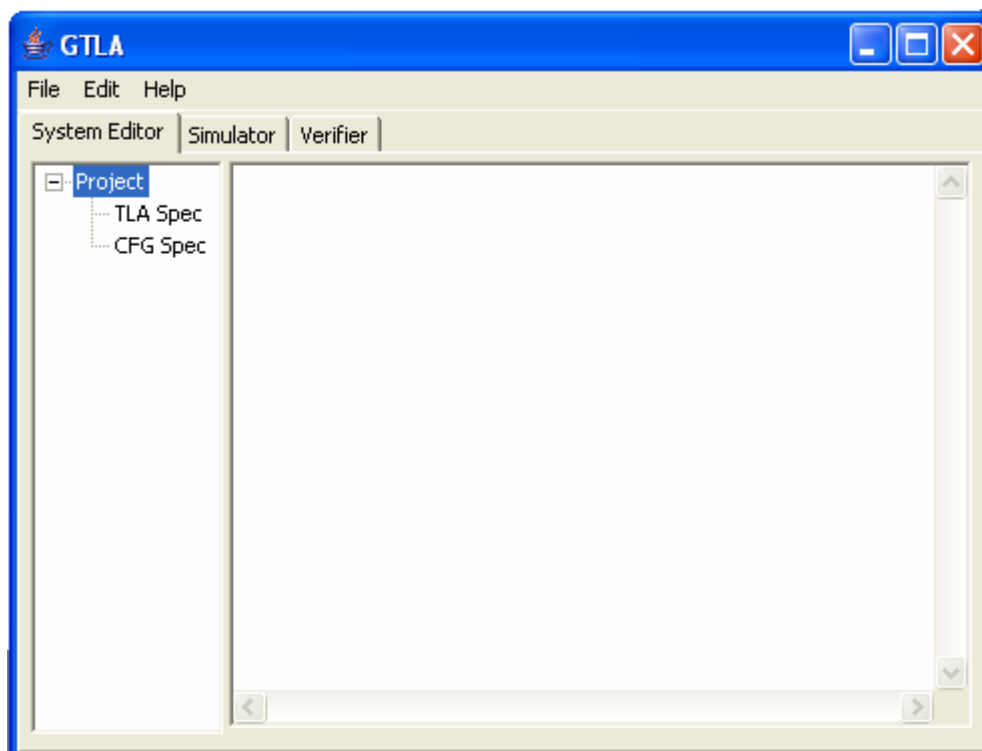


Figure 7.3: The main window (the System Editor window).

After loading the TLA system into the GTLA system, the user can use the “Check

Syntax” functionality to check its syntax. A new window “Check Syntax” (as shown in Figure 7.4) is displayed to show the syntax checking information generated from the TLC’s syntactic analyzer.

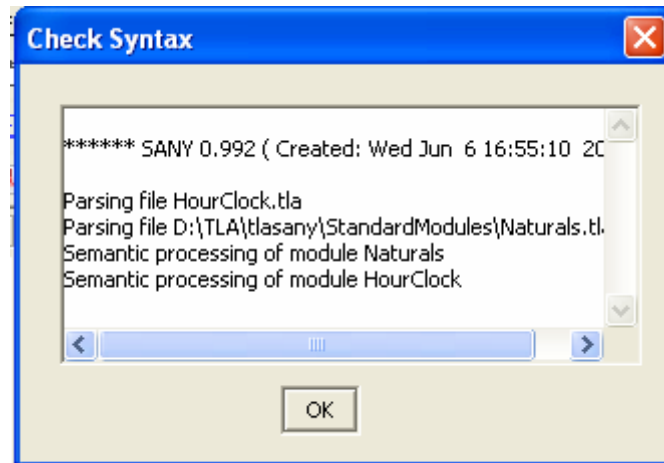


Figure 7.4: The Syntax Checker window.

When the user selects the About functionality to require more about the GTLA system, a new window “About” with information about the GTLA system should be displayed, as shown in Figure 7.5.



Figure 7.5: The About window.

7.2.3 The Simulator

The simulator is used to run the simulation manually by the user or automatically by the application. This can help the user find and locate errors of the given TLA system much easier. The simulator will have the following functionality:

- The ability to run the simulation in two different modes: manually by the user and automatically by the GTLA system

- The ability to display all the enabled actions at every step
- The ability to display the action trace at every step
- The ability to display the current value of all the variables at every step
- The ability to display the Action graph and the MSC graph, which are used to describe the simulation result at every step
- The ability to choose which enabled action to be executed by the user
- The ability to reset the simulation
- The ability to control the speed of the random simulation

The main window for the simulator should look like Figure 7.6. The two windows at the right side are used to display the Action graph and the MSC graph separately.

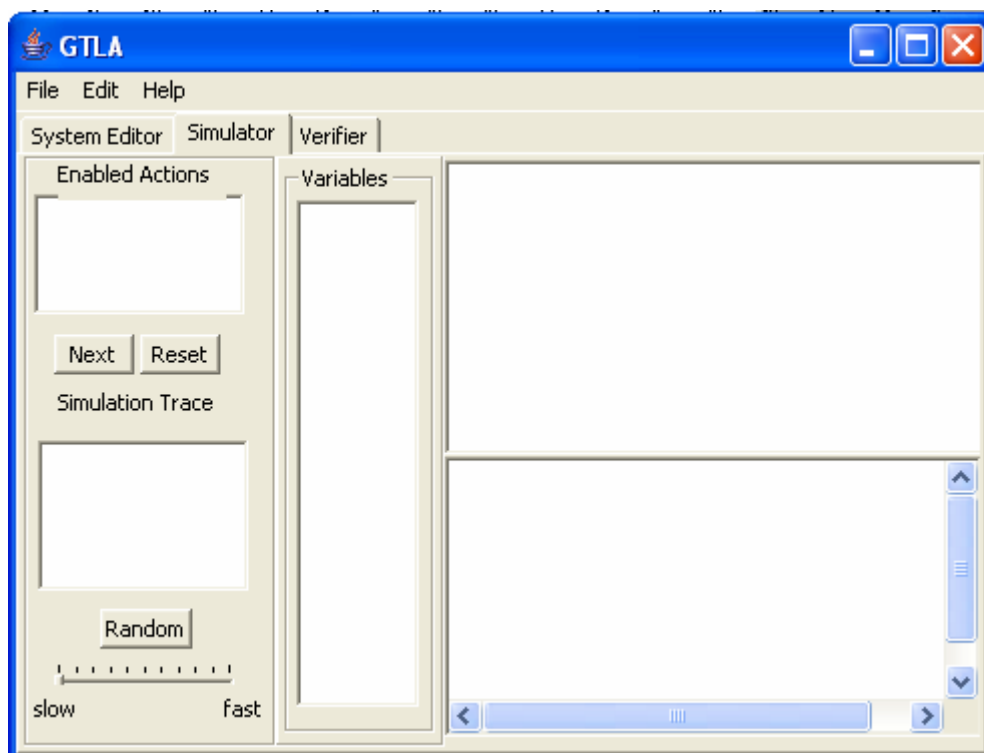


Figure 7.6: The Simulator window.

The simulator could be used in two ways:

1. In the user-guided mode the user can run the system manually and choose which enabled action to take if there is more than one action that is enabled. The “Next” button is used to execute one step. It executes either the action chosen by the user in the “Enabled Actions” list or a default chosen action. The “Reset” button is used to restart the simulator.

2. The random mode can be toggled to let the system run on its own for some fixed steps, randomly choosing an enabled action to take. The “Random” button is used to enter the random mode. The scale bar is used to set the execution speed of the random simulation.

7.2.4 The Verifier

Given a TLA system, the verifier uses the model checking technique to automatically check the validity of the properties provided by the configuration file. The verifier will have the following functionality:

- The ability to verify the given TLA system
- The ability to display the result of the verification

The main window for the verifier should look like Figure 7.7. The verifier calls the TLC model checker to do the verification work. It consists of only one editor, which displays the model-checking (verification) information generated by the TLC model checker. It doesn't allow the user to edit.

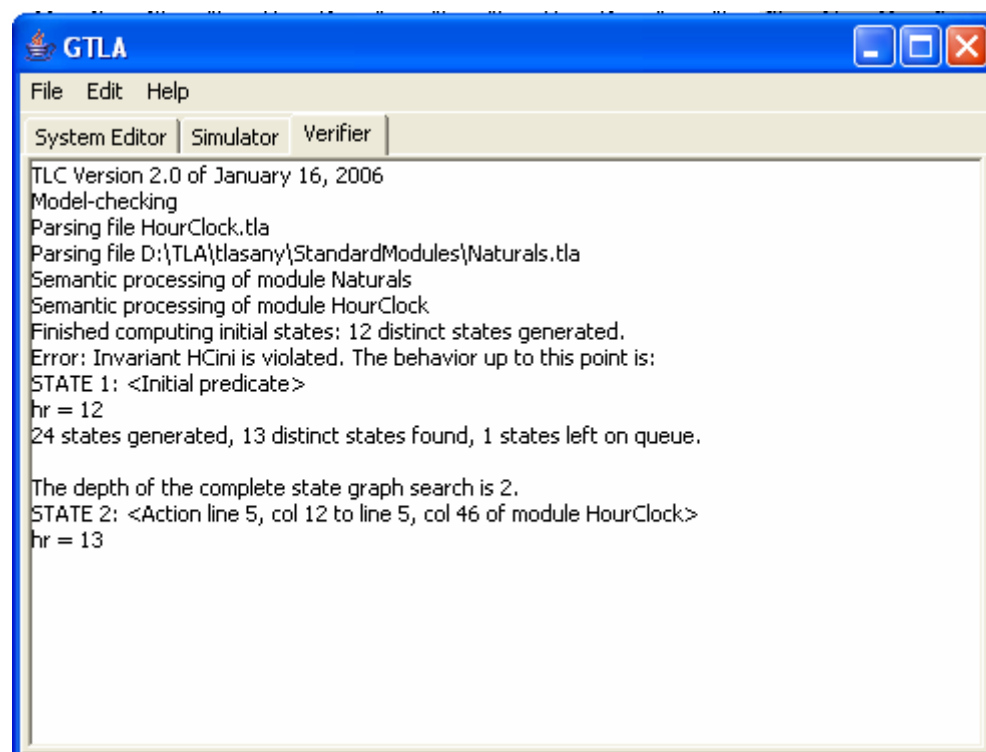


Figure 7.7: The Verifier window.

7.2.5 The Development Tool

There are different types of GUI builders: Full IDE, IDE plug-in, WYSIWYG editor, Non-WYSIWYG editor and pure code.

- A full IDE is a full integrated development environment for building the GUI, developing the supporting code, and “wiring together” the GUI with event handling and other related tasks. Examples include the Eclipse Visual Editor, NetBeans, JBuilder, and JDeveloper.
- An IDE plug-in is a GUI editor designed specially to work within an IDE. Examples include Swing Designer, Jigloo, and Jvider.
- A WYSIWYG editor is a panel builder that gives a “what you see is what you get” editing view. Examples include JFormDesigner and Foam.
- A non-WYSIWYG editor is a panel builder that helps the user build panels by grouping components and setting properties, but without the immediate visual feedback the user gets with a WYSIWYG editor.
- The pure code is a library intended to ease or improve GUI development, without providing a visual tool.

In terms of this project, several tools (like the TLC model checker, GraphViz, etc.) are needed to be integrated into the GTLA environment. So a full IDE or an IDE plug-in that could provide integration functionality is needed. Among multiple available choices, Swing Designer, an Eclipse plug-in with tight IDE integration, is chosen because of the following reasons:

- It provides integration functionality to integrate the GUI and other software into a unique environment.
- Compared with other full IDE or IDE plug-in tools, it is much easier to start and use. With it the user can construct a GUI within a shorter time.
- The best visual designer (fast, flexible, and full-featured) of most IDE or IDE plug-in.
- It supports many different layouts and components, which leads to higher development productivity and better user interfaces.

Swing Designer supports both Swing component and SWT component. Swing is the “built-in” GUI component technology of the Java platform. SWT (the Standard Widget Toolkit) is a GUI component technology that is part of the Eclipse project. There is a large and growing community that advocates SWT over Swing for Java GUI programming. In this project, the SWT GUI component technology is used.

7.3 Conclusion

This chapter stated the design and requirements of the GTLA system. The GTLA system can be mainly divided into three blocks: the GUI, the simulator and the verifier. The TLC model checker is used as the verifier of the GTLA system, so the only work to do with the verifier is to integrate it into the GTLA environment. The GUI and the simulator should be implemented by the author. They are introduced in chapter 8 and chapter 9 respectively.

Chapter 8

Developing the GUI

8.1 Introduction

In the GTLA system, the GUI mainly consists of two parts: a menu and a tabbed folder. So they are developed first. The other three important mechanisms, the system editor, the simulator and the verifier are added into the tabbed folder and the functionality of them is developed next. See Figure 8.1. In the following paragraph, the development of the application is described in four parts: the menu and tabbed folder, the system editor, the simulator and the verifier.

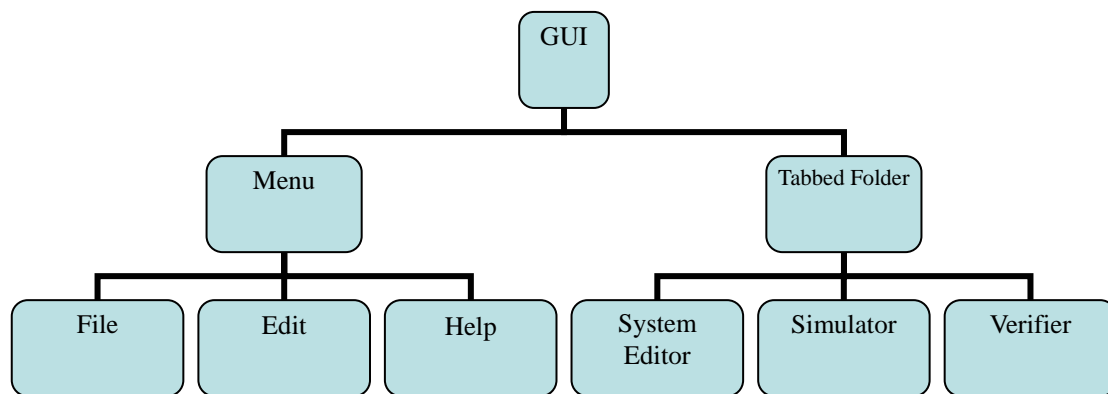


Figure 8.1: Overview of the GUI structure.

8.2 The Menu and Tabbed Folder

The foundation from which a GUI is built is the window. Two SWT classes are used to create windows: Display and Shell. Display is the class responsible for managing the interaction between all SWT widgets and the underlying operating system. Instances of Shell represent windows which are currently being managed by the desktop (on MS Windows) or the windows manager (on Unix or Linux systems). Two coding steps are always needed to create a window:

- Create an instance of the Display class:

```
Display d = new Display();
```

- Pass the instance of Display to the Shell constructor to create an instance of the Shell class:


```
Shell s = new Shell(d);
```

After creating a window, other SWT widgets can be added to it. Menus are the primary application navigation tool that an application should provide for the users. In SWT, the classes provided to create menus are `Menu` and `MenuItem`, both located in the `org.eclipse.swt.widgets` package. The following steps show how to construct the Menu, see Figure 8.2.

- The first step in creating a menu system is to create an instance of `Menu` to serve as the menu bar and attach it to an instance of the `Shell` class. The following code creates a menu bar attached to the window shell.

```
Menu menu = new Menu(shell, SWT.BAR);  
shell.setMenuBar(menu);
```

- The second step in creating a menu system is to create instances of the `MenuItem` class using the `SWT.CASCADE` style and set the text attribute of each such instance to represent the clickable text the user will see. These instances of `MenuItem` are what the user will see running across the menu bar (eg. File). Instances of `MenuItem` are attached directly to the menu bar by passing a reference to the menu bar in the `MenuItem` constructor. These `MenuItem` objects must be given the `SWT.CASCADE` style in order to add more individual menu items in each such object. The `setText ()` method is called to specify the text to appear on the menu. The following code creates an instance of `MenuItem` `fileMenuItem` and sets its text attribute to be “File”.

```
final MenuItem fileMenuItem = new MenuItem(menu, SWT.CASCADE);  
fileMenuItem.setText("File");
```

- The third step in creating a menu system is to create `Menu` instances to attach to the cascading menu item. These instances of `Menu` are the containers for the individual menu items that appear when the user clicks the cascading menu (i.e., when the menu drops down). The following code is an example.

```
Menu filemenu = new Menu(s, SWT.DROP_DOWN);  
fileMenuItem.setMenu(filemenu);
```

- The last step in creating a menu system is to create instances of `MenuItem` to add to the `DROP_DOWN` style menu, passing them a reference to the containing `Menu` instance and specifying the `SWT.PUSH` style (one of the other available `MenuItem` styles). The following code adds a `MenuItem` `openProjectMenuItem` to the `DROP_DOWN` style menu `filemenu` and sets its text attribute to be “Open TLA Spec...”.

```
MenuItem openProjectMenuItem = new MenuItem(filemenu, SWT.PUSH);  
openProjectMenuItem.setText("Open TLA Spec...");
```

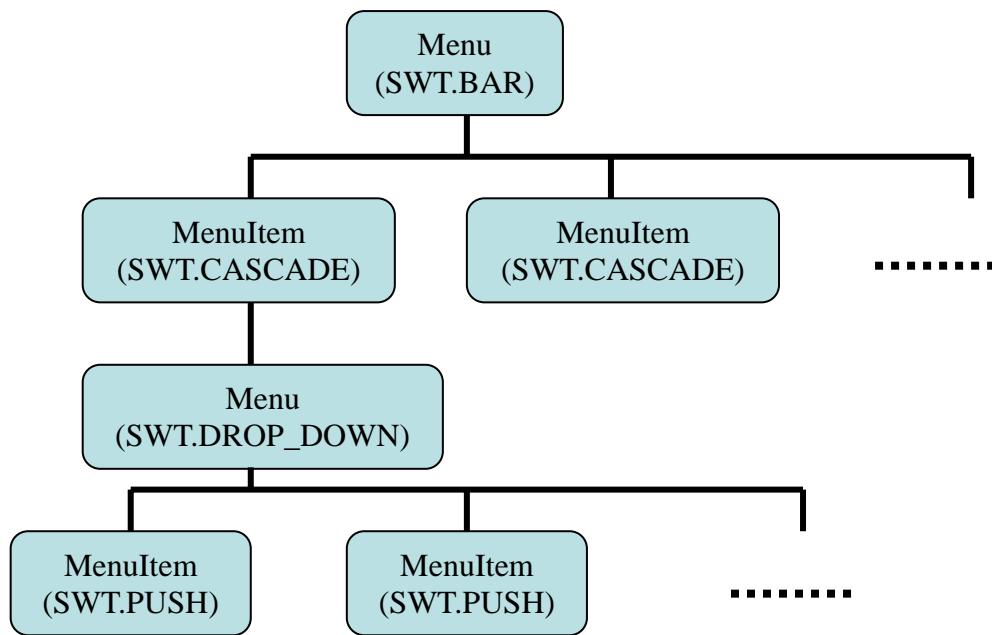


Figure 8.2: Structure of the Menu system.

Using the above techniques, the menu bar for the GTLA system can be created, as shown in Figure 8.3. The menu bar consists of three MenuItem objects: File, Edit and Help. When File is clicked by the user, it cascades to reveal some choices (Open TLA Spec... and Open Configuration Spec..., etc.).

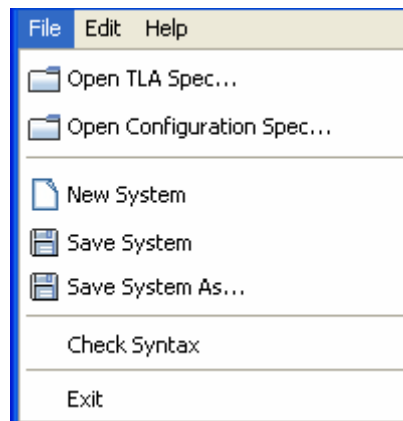


Figure 8.3: The Menu Bar and MenuItem File.

The setImage () method is called to specify the image to appear ahead of a menu item. For example, the following code adds the directory image to the “Open TLA Spec...” menu item, as shown in Figure 8.3.

```

openProjectMenuItem.setImage(SWTResourceManager.getImage(GTLAUI.class
    ,"/javax/swing/plaf/metal/icons/ocean/directory.gif"));
  
```

To divide the menu items in a cascading menu into functional groups, separator bars can be added. For example, from Figure 8.3, we can see the menu “File” is divided into four groups by three separators. A separator is simply an instance of MenuItem that has been given the SWT.SEPARATOR style:

```
MenuItem separator = new MenuItem (filemenu, SWT.SEPARATOR);
```

A separator menu item will take no action and cannot be clicked.

Using the same method, the other two menu items “Edit” and “Help” can be created. The cascading menu item Edit and Help are shown in Figure 8.4 and Figure 8.5.

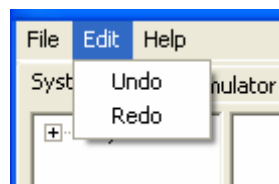


Figure 8.4: The MenuItem Edit.

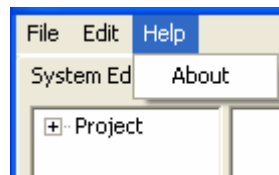


Figure 8.5: The MenuItem Help.

A very common user interface construct is one in which a window is divided using a notebook format, with tabs allowing easy access to widgets on each page of the notebook, as shown in Figure 7.3. Clicking each tab (System Editor, Simulator or Verifier) reveals a separate page of the notebook, each with its own set of widgets. This construct is usually referred to as a tabbed folder interface. The SWT provides two classes TabFolder and TabItem from the org.eclipse.swt.widgets package to create such an interface.

- First, create an instance of the TabFolder class that will act as a container for the tabs. It is added to another container class, a Shell. TabFolder has no styles intrinsic to itself, but SWT.NONE is still needed to be passed in a placeholder for a style:

```
TabFolder tf = new TabFolder (shell, SWT.NONE);
```

- Second, create individual tabs for each folder and add it to the TabFolder instance previously created, the setText() method is used to set its text attribute:

```
TabItem ti = new TabItem (tf, SWT.NONE);  
ti.setText ("System Editor");
```

- Finally, use `setControl ()` to add controls to the tab pages. Each `TabItem` page can be assigned only one control. This means that the user almost certainly will create a `Composite` for each page, add widgets to that `Composite` to achieve the desired user interface, and then add the `Composite` to the page. The following code adds a control `c` to the tab page `ti`.

```
ti.setControl(c);
```

Using the above methods, a friendly user interface for the GTLA system was created. The following work is to construct each tab page that corresponds to one mechanism and implement the functionalities.

8.3 The System Editor

The System Editor consists of two SWT widgets: a `Tree` instance and a `StyledText` instance, see Figure 7.3. The construction of it are introduced below, see Figure 8.6.

➤ Create an instance of SWT Composite

To display more than one SWT widgets in a tab page, a `Composite` object is needed to be created. `Composite` is an ancestor of every SWT class to which one can add other widgets, including `Shell`. The following code creates a `Composite` object `c` on a `tabbed Folder` object `tabFolder`, which uses `RowLayout` layout to display multiple widgets in a row.

```
Composite c = new Composite (tabFolder, SWT.NONE);  
c.setLayout (new RowLayout ());
```

➤ Use the Styled Text Area to edit text

A styled text widget can display and edit text using more than one font. Styled text components are powerful and standard part of any GUI—used either to present information to the user, or to enable the user to edit or create information. The following code creates a `StyledText` object with the horizontal and vertical scroll bars in `Composite c`, as shown in Figure 7.3.

```
StyledText styledText = new StyledText(c, SWT.V_SCROLL | SWT.BORDER |  
SWT.H_SCROLL);
```

➤ **Use the SWT Tree to control multiple files**

An instance of Tree can be created to display all the files in a given TLA system and switch between them. The Tree class represents the trunk of the tree, to which other items will be attached. TreeItem objects represent individual items (branches) of the tree. The following code creates a Tree object and adds it to Composite c.

```
Tree t = new Tree(c, SWT.BORDER);
```

Items within the tree are represented by instances of the TreeItem class. To add one item to the tree, one TreeItem object must be created.

```
TreeItem parent = new TreeItem (t, SWT.NONE);  
parent.setText ("Project");
```

➤ **Add control to the tab page**

At last, to add the composite c created above to the “System Editor” page (the tab item systemEditorTabItem), the setControl () method is used:

```
systemEditorTabItem.setControl(c);
```

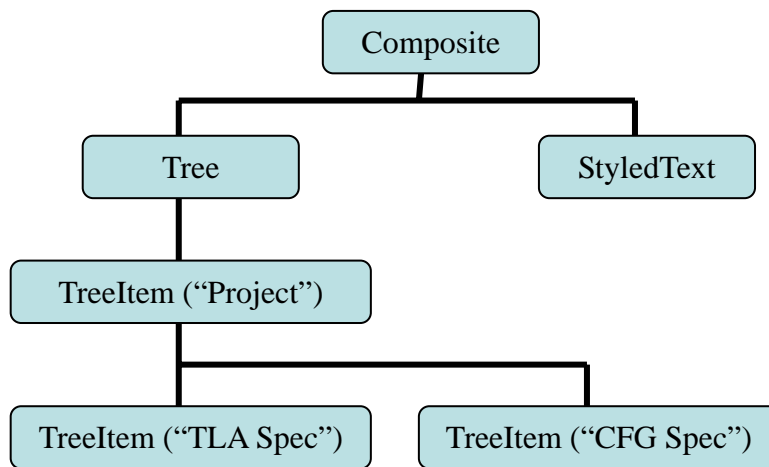


Figure 8.6: Structure of the System Editor.

So far, the user interface of the System Editor has been created. The menu bar and text editor (styled text area) has been described in the above paragraph. In the following, the functionality the system editor required will be implemented.

➤ **Use the SWT FileDialog to open/save a file**

The application needs a mechanism to ask the user for a filename to open or save files.

A file open dialog will contain many buttons, and a list box or tree view (or both) that enables the user to navigate through the file system. Developing such a dialog from scratch would take many hours of programming time. Fortunately, the SWT `FileDialog` class does all the heavy lifting for the user, who just needs to create an instance of `FileDialog` and specify a few parameters. The following code creates an instance of `FileDialog`, as shown in Figure 8.7. The `setFilterExtensions ()` method specifies what files within the directory will be displayed, in the following example only the files with the “.tla” postfix are displayed. The `setFilterPath ()` method is called to specify which directory to display initially.

```
FileDialog dialog = new FileDialog (shell, SWT.OPEN);  
dialog.setFilterExtensions (new String [] { "*.tla" });  
dialog.setFilterPath ("c:\\");
```

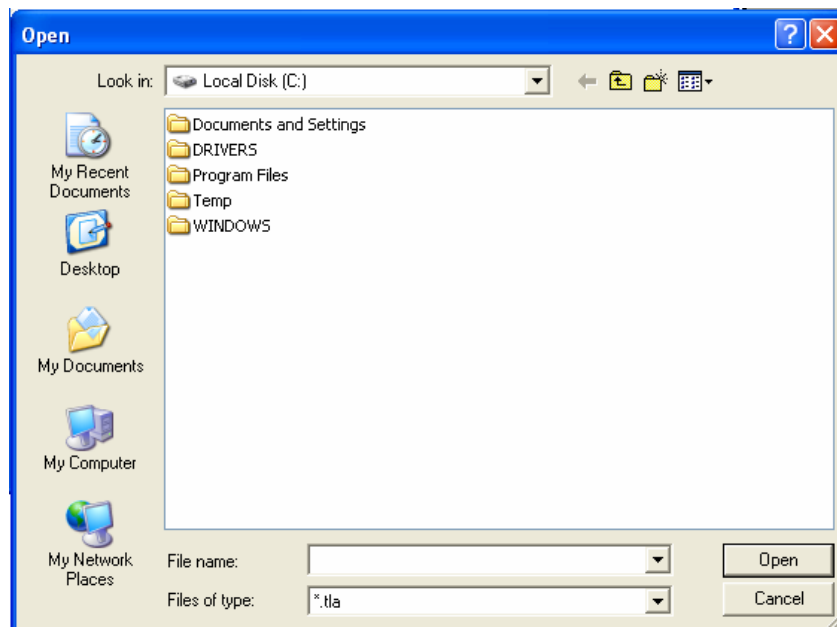


Figure 8.7: The file chooser dialog.

Finally, the dialog is opened by calling the `open ()` method and the current selection of the dialog will be returned to the caller:

```
String tlafilename = dialog.open ();
```

The variable `tlafilename` contains the file that was selected when the dialog was closed. If no file was selected, the variable is null.

The following code assigns the contents of the selected file to the variable `tlaText`, which serves as a buffer storing the contents of the TLA file.

```
String tlaText = (loadTextFromFile (tlafilename)).toString ();
```

When the contents stored in the `tlaText` is needed to be displayed in the Styled Text area `styledText`, the `setText ()` method is used.

```
styledText.setText (tlaText);
```

There are two ways to save a TLA system: either in the original place from which the system is loaded (“Save”) or in another place in the operating system (“Save As”). We mentioned above when open a file from the file chooser dialog, the file is stored in a global variable `tlafilename`. To save the system in the original place the following code is used. First create a `File` object and a `FileWriter` object given the variable `tlafilename`. Second write the contents stored in the variable `tlaText` to the file. Finally close the `FileWriter`.

```
File tlafile = new File (tlafilename);
FileWriter fileWriter = new FileWriter (tlafile);
fileWriter.write (tlaText);
fileWriter.close ();
```

To save the system in another place the file chooser dialog is used. The same code opens the `FileDialog` in save mode, except that the style passed to the constructor is `SWT.SAVE`. And instead of using the variable `tlafilename`, the new file chosen by the user is used.

➤ Control SWT widgets using SWT Listeners

To use the `Tree` object to control the contents of the Style Text area, a `Tree Listener` is needed to be created. There are two children in the tree, one “TLA Spec” and one “CFG Spec”, which represent the TLA specification file and configuration file respectively. When the user presses the “TLA Spec”, the contents of the specification file should be displayed in the Styled Text area, while when the user presses the “CFG Spec”, the contents of the configuration file should be displayed in the Styled Text area. The following code implements this functionality through adding a `Listener`.

```
t.addSelectionListener (new SelectionAdapter () {
    public void widgetSelected (SelectionEvent e) {
        TreeItem ti = (TreeItem) e.item;
        if (ti.getText ().equals ("TLA Spec")) {
            styledText.setText (tlaText);
        }
        if (ti.getText ().equals ("CFG Spec")) {
            styledText.setText (cfgText);
        }
    }
});
```

➤ **About dialog**

When the user clicks Help→About, the “About” dialog window (as shown in Figure 7.5) is displayed. That is to say, open a new window within the main window (as shown in Figure 7.3). To create an instance of Shell that as a child window of a parent shell, the following code is needed.

```
Shell child = new Shell (parent);
```

Two SWT widgets are added into this child shell. One is a label that is used to display the information.

```
Label l = new Label (about, SWT.NONE);  
l.setText ("GTLA version 0.0.1\n...");
```

The other is a button that is used to close the window. To close the window, the following listener is created for the button b.

```
b.addSelectionListener (new SelectionAdapter () {  
    public void widgetSelected (SelectionEvent e) {  
        child.dispose ();  
    }  
});
```

➤ **Check Syntax**

When the user clicks File→Check Syntax, the “Check Syntax” window (as shown in Figure 7.4) is displayed. The way to create such a window is the same as that to create the “About” window. The two widgets to be added to the child window are a Styled Text area and a button. The Styled Text area is used to display the syntax checking information generated from the TLC model checker. How it communicates with the TLC model checker will be described in Chapter 8.6.

➤ **Undo/Redo**

The submenu list “Edit” (see Figure 8.4) provides the undo/redo mechanism, which allows users to correct their mistakes and also to try out different aspects of the application without risk of repercussions.

- “Undo”: unexecuted the last action the user just performed with StyledText.
- “Redo”: re-execute the last undone action with StyledText.

Swing Designer doesn’t provide any class that can be used to implement this functionality easily. The implementation of this functionality is described below.

- First, create two LinkedList objects to store the text caused by the last undo and

redo action the user just performed.

```
private LinkedList undoStack = new LinkedList ();  
private LinkedList redoStack = new LinkedList ();
```

- Second, add a Listener to the StyledText object. When the user edits the text in the Styled Text area, add the new text changed by the user (add or delete) to the undoStack list.

```
styledText.addExtendedModifyListener (new ExtendedModifyListener () {  
    public void modifyText (ExtendedModifyEvent event) {  
        String currText = styledText.getText ();  
        String newText = currText.substring (event.start, event.start  
            + event.length);  
        if (newText != null && newText.length () > 0) {  
            if (undoStack.size () == MAX_STACK_SIZE) {  
                undoStack.remove (undoStack.size () - 1);  
            }  
            undoStack.add (0, newText);  
        }  
    }  
});
```

- Third, create two user-defined methods undo () and redo (), which unexecuted and re-execute the last action the user performed.

```
private void undo () {  
    if (undoStack.size () > 0) {  
        String lastEdit = (String) undoStack.remove (0);  
        int editLength = lastEdit.length ();  
        String currText = styledText.getText ();  
        int startReplaceIndex = currText.length () - editLength;  
        styledText.replaceTextRange (startReplaceIndex, editLength, "");  
        redoStack.add (0, lastEdit);  
    }  
}  
  
private void redo () {  
    if (redoStack.size () > 0) {  
        String text = (String) redoStack.remove (0);  
        styledText.setCaretOffset (styledText.getText ().length ());  
        styledText.append (text);  
        styledText.setCaretOffset (styledText.getText ().length ());  
    }  
}
```

- Finally, add the Listener to the “Undo” and “Redo” menu item. When the user clicks these items, the undo and redo methods are called.

8.4 The Simulator

There are multiple widgets in the simulator tab item. As the System Editor a Composite object is created to display these widgets in one page. There are some differences. The structure of the simulator is shown in Figure 8.8.

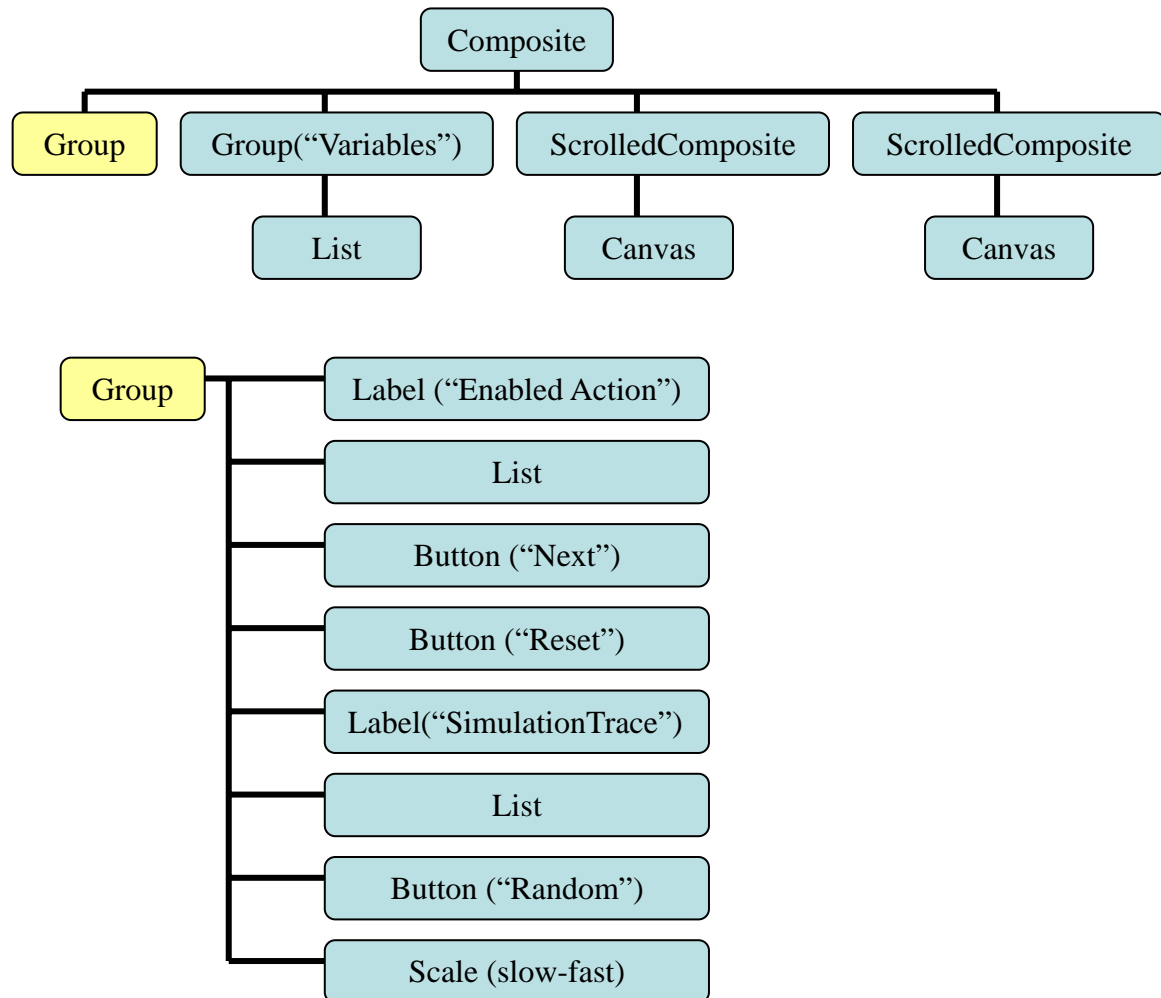


Figure 8.8: Structure of the Simulator.

➤ Group and List

The SWT Group class extends the Composite class and provides some additional functionality related primarily to the appearance of the group. Group permits the user to add a text label as a prompt to the user of group contents and to change the border style of the resulting object, yielding more of a visual indication to the user that the widgets are part of a functional area of the application.

Because the Group class cannot be subclassed, it must be used in a manner similar to Shell—the user creates an instance of Group within another container class (such as

Composite) and then add widgets to that instance.

```
Group group= new Group (composite, SWT.NONE);
```

The simulator contains two groups, as shown in Figure 7.6. One group contains the “Enabled Actions” and “Simulation Trace” labels, two list objects, the “Next”, “Reset”, “Random” buttons and one scale. The other “Variables” group contains one list to display all the variables’ information.

List widgets are used to present information to the user and allow selection of an item or items for further action (when processing is initiated by a command button, for example).

As with all widgets, the first step in using a list is to create the List object – an instance of the List class – then position the list within the container as desired. Once a list object has been created, it can be populated with items. For example, the following code adds a list object to an instance of group. The add method is used to add items to the list.

```
List list = new List (group, SWT.BORDER);  
list.add (“Item One”);
```

When the simulator runs, the information of enabled actions, action trace and variables are generated and stored in the corresponding files. To display all the useful information from those files to the list, the following user-defined method is created. Given the filename, from which the contents of the file can be read into the application, and the list, to which the contents are displayed, this method reads in the contents from the file, removes the old items in the list and adds new items to the list. Using this method it is easy to display various kinds of information in the list.

```
void WriteTextToList (String filename, List list) {  
    File file = new File (filename);  
    try {  
        BufferedReader reader = new BufferedReader (new FileReader (file));  
        String line = null;  
        list.removeAll ();  
        while ((line = reader.readLine ()) != null) {  
            list.add (line);  
        }  
    }  
    catch (IOException e) {  
        System.out.println ("Failed to load text from file!");  
    }  
}
```

➤ ScrolledComposite and Canvas

The simulator automatically loads two graphs to describe the simulation. These two graphs could be very big, so a ScrolledComposite object is needed to be created to

display the graphs. Actually the graph is contained in a Canvas object, which is contained in the ScrolledComposite object. SWT Canvas can be used to handle loading, scrolling, painting and zooming of large images. The following code is used to create a canvas object in a ScroledComposite object:

```
Canvas canvas = new Canvas (scrolledComposite, SWT.NONE);
scrolledComposite.setContent (canvas);
```

The setBackground () method can be used to set the background color of the canvas. For example, the following code sets the background color to be white:

```
canvas.setBackground (display.getSystemColor (SWT.COLOR_WHITE));
```

To load images to the canvas, a new user defined method setImage is defined as following, which load image in the path img to the Canvas object. The Image (Display, String) method is used to generate an image under the given path. The setSize () method is used to set the canvas' size the same as the image. This made the image be loaded into the canvas with 1:1 scale.

The default behavior for a Canvas is that before it paints itself the entire client area is filled with the current background color. This can create screen flicker, because if the paintEvent also draws onto the GC, then the user sees a flash between the original background being filled, and the drawing occurring. One way to reduce flicker is to batch up the drawing that occurs on the display by double buffering. Double buffering is a technique where the drawing occurs onto a GC that is not he one used on the paintEvent, and the contents are then copied across. To do this you can create an Image with the same size as the Canvas' client area and then draw onto this with a GC (Image); The final image is then transferred across to the paint event's GC with a single drawImage (Image image, int x, int y) method call. When using this technique be aware that some platforms perform native double buffering for you already, so you may in fact be triple buffering.

The old image is needed to be disposed before loading the new image into the canvas. This is implemented with a Dispose Listener.

```
public void setImage (Canvas c, String img) {
    final Image image = new Image (display, img);
    final Canvas canvas = c;
    canvas.setSize (image.getBounds ().width, image.getBounds ().height);
    canvas.addPaintListener (new PaintListener () {
        public void paintControl (PaintEvent e) {
            GC gc = e.gc;
            gc.drawImage(image,0,0,image.getBounds().width,
            image.getBounds().height,0,0,canvas.getSize().x,canvas.getSize().y)
            ;
        }
    });
    canvas.addDisposeListener (new DisposeListener () {
        public void widgetDisposed (DisposeEvent e) {
            image.dispose ();
        }
    });
}
```

```

    }
  });
  canvas.redraw ();
}

```

With the ScrolledComposite and Canvas, the graphs used to describe the simulation can be loaded and displayed in the simulator page.

How to display the useful information and graphs is described above. The other functionalities of the simulator are described below.

➤ **Run the simulator manually**

The “Next” button is used to control the simulation manually by the user. When the user presses the “Next” button, the application communicates with other software (like the SML compiler, the GraphViz tool, etc) to run one step of simulation, and then reloads all the corresponding information and graphs into the simulator page to display for the user. This is implemented by using a Listener to respond to the user having clicked the button, see the following code. At every step, when the user presses the “Next” button, the simulator chooses one enabled action to execute in the next step. There maybe multiple enabled actions, by default the first item in the “Enabled Action” list is chosen to be executed in the next step. The statement “list.select (0);” is used for this purpose. Of course the user can choose which enabled action to take from the list. This is what we called User-Guide mode. The method getSelection () is called to return the chosen item to the caller. In the future, the selected item is used as a reference to be passed to the simulation commands. The communication that is used to execute the simulation is described in chapter 7.3. After running the simulation, the new information of enabled actions, action trace, and values of variables, MSC and Action graphs are reloaded into the list or the canvas in the simulator page to display to the user.

```

nextButton.addSelectionListener (new SelectionAdapter () {
    public void widgetSelected (final SelectionEvent e) {
        list.select (0);
        String selected [] = list.getSelection ();
        .....
        /*communicate with other software by using runtime commands*/
        .....
        /*reload information and graphs to the simulator page*/
    }
});

```

➤ **Run the simulator randomly**

The “Random” button is used to run the simulation randomly by the simulator. Similar as in the manual mode, the random mode runs the simulation step by step. A

for-loop statement is used here to set how many steps to run at each time.

➤ **Reset the simulator**

The “Reset” button is used to restart the simulator. A listener is added to respond to the user. When the user clicks this button, it calls the CreateNewFile program to clear all the old temp files and resets the runtime process to generate new files.

➤ **Control the speed of random simulation**

To control the speed of the random simulation, in the random mode, the following statement is used within the for-loop to make the thread sleep for a while at each step in order to provide the user a clear view in the GUI.

```
try {Thread.sleep (RandomTime); } catch(Exception ee) {};
```

The variable RandomTime is initialized to be 3000. A Scale object is created to change this variable in order to control the sleep time between two steps. That is the speed of the simulation. The maximum and minimum are set to be 3000 and 0. The page increment is set to be 500. A Listener is created to respond to the user. When the user adjusts the scale, the corresponding number is returned to the caller, which reset the value of variable RandomTime. In this way, the user can control the speed of the random simulation.

```
Scale scale = new Scale (group, SWT.NONE);
scale.setBounds (10, 255, 100, 15);
scale.setMaximum (3000);
scale.setMinimum (0);
scale.setIncrement (100);
scale.setPageIncrement (500);
scale.addListener (SWT.Selection, new Listener () {
    public void handleEvent (Event event) {
        int perspectiveValue = scale.getSelection ();
        RandomTime = 3000 - perspectiveValue;
    }
});
```

8.5 The Verifier

An instance of Styled Text area is created to display the verification (model-checking) information generated by the TLC model checker. It doesn't allow the user to edit the text area. How the verifier calls the TLC model checker to verify the given TLA system is described in chapter 8.6.

8.6 Integration

The GUI, the TLA Front-End and the translator, the GraphViz tool and the graph generator algorithms, the SML/NJ compiler and the TLC model checker are integrated into the GTLA environment by using the Java runtime package. The relationship among these tools is described in Figure 7.1.

To implement the integration task, the Java runtime package is used. The class `java.lang.Runtime` features a static method called `getRuntime ()`, which retrieves the current Java Runtime Environment. That is the only way to obtain a reference to the `Runtime` object. With that reference, the user can run external programs by invoking the `Runtime` class's `exec ()` method. The following example shows how to execute “javac” command:

```
Runtime rt = Runtime.getRuntime ();
Process proc = rt.exec ("javac");
```

So, in terms of the “Check Syntax” functionality, the TLA+ syntactic analyzer is called by using the following commands:

```
File file = new File (sourcefile);
Runtime rt = Runtime.getRuntime ();
Process proc = rt.exec ("java -cp "+ tlapath +" tlasany.SANY "+ tla, null, file);
```

The “`tlapath`” variable is the installed path of the TLA tool, for example, it could be “`d:\tla`”; the “`tla`” variable is the given TLA file’s name; the file is the directory the TLA source files located in.

In terms of the “Verifier” functionality, the TLC model checker is called by using similar commands:

```
File file = new File (sourcefile);
Runtime rt = Runtime.getRuntime ();
Process proc = rt.exec ("java -cp "+ tlapath +" tlc.TLC "+ tla, null, file);
```

There are some points need to be paid attention to.

- Because some native platforms only provide limited buffer size for standard input and output streams, failure to promptly write the input stream or read the output stream of the sub process may cause the sub process to block, and even deadlock. To solve this problem, two threads are added to read the output into a buffer. One deals with the normal output stream, the other deals with the error output stream. The following is an example of a thread dealing with the error output stream.

```
new Thread (new Runnable () {
    public void run () {
        try {
            BufferedReader br_err = new BufferedReader (new
                InputStreamReader (proc.getErrorStream ()));
```

```

String buff = null;
while ((buff = br_err.readLine ()) != null) {
    System.out.println ("Process err :" + buff);
    try { Thread.sleep (100);} catch (Exception e) {}
}
br_err.close ();
} catch (IOException ioe) {
    System.out.println ("Exception caught printing javac result");
    ioe.printStackTrace ();
}
}
}).start ();

```

- In terms of the simulator, there are multiple commands which need to be executed. After calling the SML/NJ compiler to load the SML file, other commands need to be write into the output stream to execute the simulation step by step. The following is an example.

```

try {
    BufferedOutputStream  bufferout  =  new  BufferedOutputStream
                                   (m_process.getOutputStream ());
    PrintWriter commandInput = new PrintWriter ((new OutputStreamWriter
                                   (bufferout)), true);
    commandInput.println ("Spec (");
    commandInput.close ();
} catch (Exception e) {
    System.err.println (e);
}

```

8.7 Conclusion

After constructing the GUI and integrating all the tools into the GTLA environment, the user can run the TLC model checker from a graphical user interface. The GTLA system can provide the most important three mechanisms as all the other model checkers that have a GUI: the system editor, the simulator and the verifier. The system editor can provide basic functionality. The simulator provides several views of the simulation, as MSC graph, variable list, etc. Although these are less-developed compared with some model checkers (as Xspin), they provide all the useful information and two graphs to the user. Especially the action graph, it provides the user with a new view of the simulation. The verifier can verify the given TLA system and display the result to the user. But the result can't be loaded into the simulator. This functionality is called the executable verifier, which is part of the future work described in chapter 11.

The Swing Designer tool works pretty well in constructing the GUI. It is easy to use and made me finish the construction work within a short time. All the related tools are well integrated into the GTLA environment. An example is provided in the Appendix A– GTLA User’s Guide, which shows how to use the GTLA system in detail. The performance, correctness, functionality of the GUI is tested and the result is provided in chapter 10.

Chapter 9

Developing the Simulator

The simulator is an important functionality of the GTLA system. With the simulator the user can use the simulation technique to verify a TLA+ system. This makes the user easier locate errors in the TLA+ system.

The simulator can be used in two ways:

- In the user-guided mode the user can run the system manually and choose which enabled action to take at the next step.
- The random mode can be toggled to let the system run on its own, randomly choosing an enabled action to take.

The simulation is described by five parts.

- The “Enabled Actions” list displays all the available transitions in the next step and can be chosen by the user.
- The “Simulation Trace” list displays the action trace after each step.
- The “Variables” list shows the values of variables in the current state.
- The “Action graph” shows all the actions and active actions or transitions of the current state.
- The “MSC graph” shows the transitions between the different actions at every step.

This chapter will clarify how these kinds of functionality are implemented and how the simulator runs in detail.

9.1 Architecture

The simulator is implemented by communication among the GUI and other tools like SML compiler, Graph generators and GraphViz tool [19] using Java Runtime package. How it works is described below, see Figure 9.1.

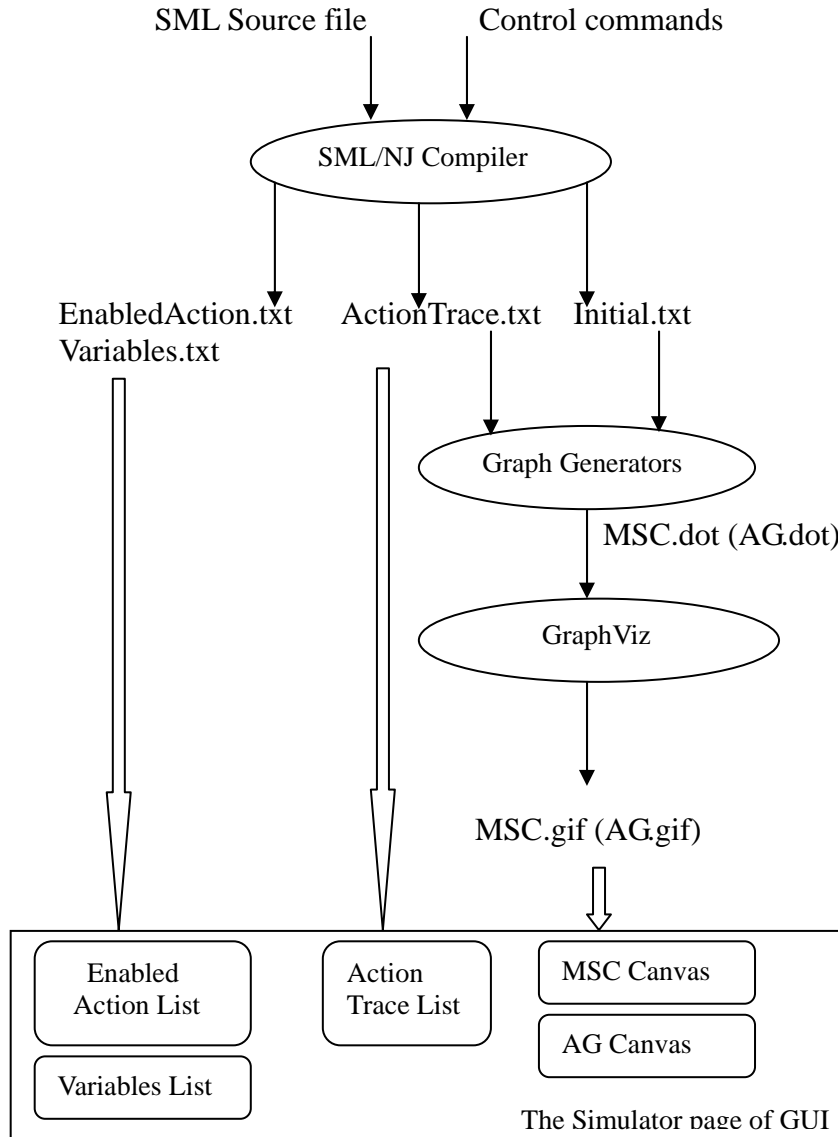


Figure 9.1: The simulator’s architecture.

- Firstly, the simulator is started when the user enters the simulator page in the GUI. At the same time, the simulator calls the TLA Front-End tool and the translator to translate the given TLA+ system into a SML file. Then it calls the SML/NJ compiler running this SML source file. The initial information and graphs are generated and loaded into the GUI. Among them, the simulation trace list is empty because no action is ever taken; the enabled action list contains all the enabled actions to start with; the variable list contains all the variables with initial value set by the system; the MSC and AG graphs are described in section 9.3 in detail.
- Secondly, after the simulator is ready to use, the user can either run the simulator manually by pressing the “Next” button or automatically by pressing the “Random” button from the GUI. No matter in which mode, the simulation is executed step by step. At every step, the corresponding executable SML commands (Control commands) are written into the SML/NJ compiler which is

running the SML source file. For example, in the user guide mode, the enabled action “Send” is chosen to be executed in the next step, the SML command “Send();” will be written to the SML/NJ compiler. More information is given in section 9.2. After executing one step, the new information and graphs are generated to the corresponding files. Among them, the EnabledAction.txt, EnabledAction.txt and Variables.txt are used to display the enabled actions, the simulation trace and the current value of variables at every step in the corresponding list windows of the GUI; the Initial.txt and ActionTrace.txt are used as the input files of the Graph Generators.

- Thirdly, the simulator calls the graph generators to generate the MSC.dot and AG.dot files and calls the GraphViz to generate the message sequence chart (MSC.gif) and action graph (AG.gif) given the MSC.dot and AG.dot as the input files.
- Finally, the useful information in the EnabledAction.txt, Variables.txt and ActionTrace.txt, and the two graphs MSC.gif and AG.gif are loaded into the enabled action’s list, the variables’ list, the action trace’s list, the MSC canvas, and the AG canvas of the GUI respectively to describe the simulation after one step.

9.2 Executable SML file

The above paragraph provides a general view of how the simulator works. To know more details about it, we should examine the executable SML source file. In the following paragraph, an example is used to clarify what kinds of SML source file is expected to be generated from a TLA+ specification and how this SML source file is executed to implement the simulator functionality.

➤ The TLA+ specification file and configuration file

A TLA+ system consists of two files: a specification file and a configuration file. Normally, a TLA+ specification file consists of four parts:

- Constant and variable declarations
- Initial predicate and next actions’ declarations.
- Next (optional) and Specification declarations
- Theorem declaration

The following is an example.

```

----- MODULE AsynchInterface -----
EXTENDS Naturals
CONSTANT Data
VARIABLES val, rdy, ack
TypeInvariant ==  $\wedge$  val  $\backslash$ in Data
                $\wedge$  rdy  $\backslash$ in {0, 1}
                $\wedge$  ack  $\backslash$ in {0, 1}

-----
Init ==  $\wedge$  val  $\backslash$ in Data
        $\wedge$  rdy  $\backslash$ in {0, 1}
        $\wedge$  ack = rdy
Send ==  $\wedge$  rdy = ack
        $\wedge$  val'  $\backslash$ in Data
        $\wedge$  rdy' = 1 - rdy
        $\wedge$  UNCHANGED ack
Rcv  ==  $\wedge$  rdy  $\neq$  ack
        $\wedge$  ack' = 1 - ack
        $\wedge$  UNCHANGED <<val, rdy>>
Next == Send  $\vee$  Rcv
Spec == Init  $\wedge$  [][Next]_<<val, rdy, ack>>

-----
THEOREM Spec => []TypeInvariant
=====

```

Normally, a configuration file specifies the specification's name in the corresponding TLA+ specification, the properties to be verified and the instantiation of the constants. The following is the configuration file of the above example.

```

SPECIFICATION Spec
INVARIANT TypeInvariant
CONSTANT Data = {0,1,2}

```

➤ The generated SML source file

Given such a TLA+ system, what kind of SML file should be generated to make it executable? Here only the expected form of the SML file is discussed. In terms of how the translation work is implemented, please refer to chapter 6.

Normally, the expected SML file should fulfill two purposes. One is that the correctness and completeness of the given TLA+ system should be guaranteed after it being translated into the SML file. The other is that some extra functionality should be provided to assist the execution of the simulator.

By referring to the configuration file and context of the TLA+ specification file, the constants and variables are declared and initialized in the SML file. The other predicates besides the initial predicate and next actions, like Theorem, are directly translated to functions in the SML file.

There are some extra variables and functions are added into the generated SML file to assist the execution of the simulator.

The variables are:

```
val rand = Random.rand(0,1);  
val nextactionnum = [0,1];  
val counter = ref 0;
```

The variable rand is used as a parameter to randomly choose one action or one value to be assigned to a variable; the variable nextactionnum is initialized according to the number of next actions, it is used to randomly choose one next action to execute in the random mode; the variable counter is used to different between the initial predicate and next actions.

The functions are:

The writeV () function: being used to write the current variables' values into the Variables.txt file.

The writeAT () function: being used to add the next action to be executed into the ActionTrace.txt file.

The writeEA () function: being used to write the enabled actions of the next step into the EnabledAction.txt file.

The VariablesOutput () function: being used to write all the variables' values in with their names into the Variables.txt file by calling writeV () function.

What kind of forms does the initial predicate, next actions, the specification predicate have in the SML file.

- The initial predicate: a “VariablesOutput ();” statement is added into the initial predicate function to generate initial variables' values.
- The next action: each next action is translated into two functions, one Guard function that specifies the guard to execute the action, if the guard is satisfied, a true value is returned, otherwise a false value is returned. The other one is the action itself, but in this function two extra statements are added. One is the “VariablesOutput ()”, the other is “writeAT “Send”” for example, to add the name to the action trace file if this action is executed.
- An extra function GuidNext () is defined to use the Guard function of each next action to estimate which action is enabled in the current state and calls the writeEA () function to write them into the enabled action file.
- An extra function RandNext () is defined to simulate the random mode. In this function, a random action among the enabled actions is chosen.
- The specification predicate: the specification function first execute the initial predicate and later calls the RandNext () function to execute a random chosen

enabled action.

➤ Execution

In the User-Guide mode, for example, the user chooses the Send action to be executed in the next step, then just write command “Send();” into the SML/NJ compiler, the Send function is called and the corresponding information about variables values, action trace are generated. After that the “GuidNext ();” command is also need to be written into the SML/NJ compiler to get the enabled actions in the next step.

In the random mode, only a for-loop is needed to be created to call the RandNext () and GuidNext () functions continuously for amount of steps, which is predefined for 100 steps.

In such a way, the simulator is implemented and the information about variables, enabled actions, action trace can be got during the execution of the simulator.

9.3 Graph Generator

In the following paragraph, it is discussed how to generate some graphs to describe the simulation.

9.3.1 GraphViz and Dot

GraphViz is an open source graph visualization software package. It contains several main graph layout programs, which take descriptions of graphs in a simple text language, and construct diagrams in several useful formats such as images and SVG for web pages, Postscript for inclusion in PDF or other documents; or for display in an interactive graph browser. GraphViz has many useful features for diagrams, such as options for colors, fonts, tabular node layouts, line styles, hyperlinks, and custom shapes.

Dot [20] is one graph layout program provided by GraphViz. It creates “hierarchical” or layered drawings of directed graphs. It runs as a command line program, a web visualization service, or with a compatible graphical interface. It reads attributed graph text files and writes drawings, either as graph files or in a graphics format such as GIF, PNG, SVG or PostScript (which can be converted to PDF). Dot accepts input in the DOT language. This language describes three kinds of objects: graphs, nodes, and edges.

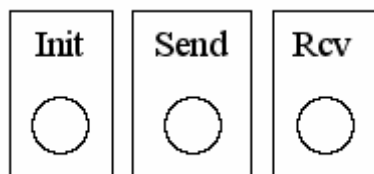
GraphViz and Dot are used to generate some simulation graphs. First some algorithms are created to generate the corresponding dot files given the action transitions and

other information from each step of the simulation. Then the GraphViz and Dot tool are called to generate graphs from these dot files.

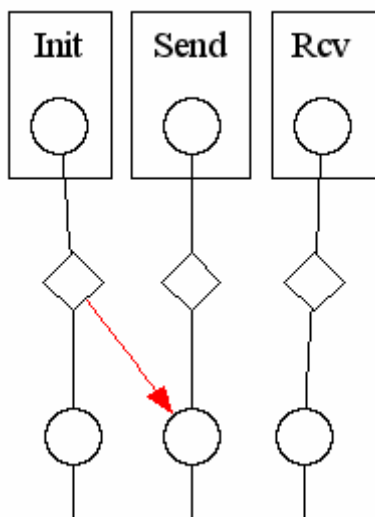
9.3.2 MSC Algorithm

The MSC algorithm generates the message sequence chart in the following steps:

- According to the file Initial.txt, which shows all the actions in the given TLA specification, generate the initial graph. For example, a TLA specification has three actions in total: Init, Send, Rcv, the initial graph is as follow:



- According to the file ActionTrace.txt, which shows the action trace generated by running the responding SML file, generate the message sequence chart in every step. This MSC graph shows the transitions between these actions. The following graph is an example.



This algorithm is a Java program, which generates the corresponding DOT language from the files Initial.txt and ActionTrace.txt. Given action list in the ActionTrace.txt, for each action, a switch and a node, which represent the new state, are generated. Additionally an arrow which shows the transition from the last action to the current action is added to the graph. By calling the tool GraphViz and Dot, the graph is generated given the dot file.

9.3.3 Action Graph Algorithm

The AG (Action Graph) algorithm generates the action graph in the following steps:

- From the file Initial.txt, which shows all the actions in the given TLA specification, the initial graph is generated. For example, a TLA specification has three actions in total: Init, Send, Rcv, the initial graph is as follow:



- From the file ActionTrace.txt, which shows the action trace generated by running the responding SML file, generate the action graph in every step. This action graph shows the transitions between these actions with the red edge representing the current transition, the black edge representing others. The following graph is an example with an action sequence:

“Send→Rcv→Send→Rcv→Send→Rcv→Send→Rcv”.



This algorithm is a Java program, which generate the corresponding DOT language given the Initial.txt and ActionTrace.txt. Given action list in the ActionTrace.txt, for each action, a transition between two nodes is generated if it doesn't exist in the graph. Additionally the current transition is represented with red color, while other transitions are represented with black color. By calling the tool GraphViz and Dot, the graph is generated given the dot file.

9.4 Conclusion

After constructing the simulator and integrating all the tools into the GTLA environment, the user can use the simulation technique to validate a TLA system. The simulator provides the user two ways to run the simulation: the user-guide mode and the random mode. At each step the simulator provides the user five views of the simulation: enabled actions, action trace, current variables' value, a message sequence chart and an action graph. The simulator is successfully constructed and it fulfils all functionality the GTLA system required. The performance, correctness, functionality of the GUI is tested and the result is provided in chapter 10.

Chapter 10

Testing

Chapter 6, 7, 8, 9 introduced the design and implementation of the GTLA system. In this chapter, each unit and the integrated system will be tested. The methods, results and refinements will be presented.

10.1 Unit Testing

The GTLA system mainly consists of five units: the translator, the system editor, the simulator, the verifier, and the GUI. Because the TLC model checker is used as the verifier directly, we don't need to test the verifier unit. Additionally, the system editor is part of the GUI, so the system editor testing is included in the GUI testing. In this section, the translator unit, the simulator unit and the GUI will be tested.

10.1.1 Translator Testing

The GTLA Translator, as a unit of the GTLA system, provides the basic executable SML file for the simulator. Given a TLA+ file and a configuration file, the GTLA translator translates them into an executable SML file. The testing of this translator is divided into two phases.

First, systematically test every statement (expression) using various examples in order to keep the correctness and completeness of the testing.

To write such an example and test it (for example, to test the let... in... statement):

- Write a basic TLA+ specification:

```
----- MODULE TEST LETIN -----  
  
EXTENDS Naturals  
  
/*add testing statement*/  
  
-----
```

- Add various testing statement into the specification, such as:

```
TestA == LET p == 6 IN p+1
```

TestB == LET x == 1

f [t \in Data] == t + 6

IN x + f [1]

- Run the TLA+ Front-End tool to generate the JCG file (flattened parse tree) given the TLA+ testing specification.
- Run the Translator to generate the SML file from the generated JCG file.
- Use the SML/NJ compiler to execute the generated SML file and check the correctness of it.

Several examples used for the systematic testing are given in Appendix E – Testing Examples 2. After testing, most TLA+ operators can be correctly translated into the SML code. They are given in the following table.

Logic operators	$\wedge, \vee, \neg, \Rightarrow, \equiv, TRUE, FALSE, \forall x \in S : p, \exists x \in S : p,$ $CHOOSE x \in S : p$
Set operators	$=, \neq, \in, \notin, \cup, \{e_1, \dots, e_n\}, \{x \in S : p\}, \{e : x \in S\}$
Function operators	$f[e], [x \in S \mapsto e], [f \text{ EXCEPT } ![e_1] = e_2]$
Record operators	$e.h, [h_1 \mapsto e_1, \dots, h_n \mapsto e_n], [r \text{ EXCEPT } !h = e], [h_1 : S_1, \dots, h_n : S_n]$
Tuple operators	$e[i], \langle e_1, \dots, e_n \rangle$
Strings and Numbers	$"c_1, \dots, c_n", STRING, d_1, \dots, d_n, d_1, \dots, d_n.d_{n+1}, \dots, d_m$
Miscellaneous constructs	IF p THEN e1 ELSE e2, CASE $p_1 \rightarrow e_1 \square \dots \square p_n \rightarrow e_n \square OTHER \rightarrow e$ $LET d_1 = e_1 \dots d_n = e_n \text{ IN } e$

	$\wedge p_1 \dots \wedge p_n$ $\vee p_1 \dots \vee p_n$
--	--

Even though, in some special cases, bugs occur. They are stated below.

➤ Logic operators

- 1) Unbounded quantifiers are excluded. They are:

$$\forall x : p, \exists x : p, \text{CHOOSE } x : p$$

- 2) Bounded quantifiers are partly tested because in those cases containing multiple parameters, they can't be translated correctly into SML code, such as:

$$\forall x \in S1, y \in S2 : F(x, y), \exists x \in S1, y \in S2 : F(x, y),$$

$$\text{CHOOSE } x \in S1, y \in S2 : F(x, y)$$

- 3) BOOLEAN [the set {TRUE, FALSE}]. For example, given a TLA+ expression $x \in \text{BOOLEAN}$, SML doesn't provide a corresponding BOOLEAN set. But this can be solved in such a way: whenever encounter token "BOOLEAN" in the source file, it is translated directly into "[true, false]" in the SML file.

➤ Set operators

Operators $\cap, \subseteq, \setminus, \text{SUBSET}, \text{UNION}$ specify intersection, subset of, set difference, set of subsets of, union of all elements of. SML doesn't provide corresponding operators. In the literature survey, these operators are suggested to be replaced by corresponding user-defined operator mechanisms in SML. They are not included in this project, as part of future work.

➤ Function operators

- 1) DOMAIN f. This can't be translated into SML code directly.
- 2) $[S \rightarrow T]$ specifies the set of functions f with $f[x] \in T$ for $x \in S$. This complete operator can't be translated into SML code.

These were stated also in Chapter 6 Translating TLA+ into SML.

➤ Record and Tuple

$S_1 \times \dots \times S_n$ specifies the set of all n-tuples with i^{th} component in S_i .

This was excluded in the literature survey.

➤ Miscellaneous constructs

CASE $p_1 \rightarrow e_1 \ [] \dots [] p_n \rightarrow e_n$. This should be translated into the If expression in SML, but in SML the If expression should be of the form “if...then...else if...then...else...”. What should be generated after “else”? In the TLA+, when p_i is false for all i , the value of this case expression is unspecified. If this expression is used, the value of the expression should matter only when there is some i that p_i is true. So instead of translating the last case “ $p_n \rightarrow e_n$ ” into “else if p_n then e_n ”, we translate it into “else e_n ”, assuming if all the earlier $n-1$ condition p are false then the last condition p must be true.

Second, further test the GTLA translator using several concrete TLA+ examples to check the performance and correctness of the translator. The examples used to test the translator are in Appendix D – Testing Examples. Excluding all the restrictions of the GTLA translator, given a TLA+ specification and a configuration specification, the GTLA translator can translate them into the executable SML code correctly. But still more testing should be taken.

10.1.2 Simulator Testing

The Simulator, a unit of the GTLA system, uses the simulation technique to validate a given TLA+ specification. Given a TLA+ file and a CFG (configuration) file, the GTLA translator is used to translate them into an executable SML file. The simulator executes this SML file step by step manually or automatically and generates some useful information, which is fed back to the simulator page of the GUI. The purpose of the simulator testing is to check:

- In the user-guide mode, let the simulator execute one action (among all the enabled actions), check if the output information is correct.
- In the random mode, let the simulator execute actions for amount time period, check if the simulator works well and the output information is correct.
- At every step of the simulation, check if the MSC and AG graphs are generated correctly.

Several examples are used to test the simulator. The testing is distributed into three phases.

- Test random mode: given a SML file, after running this file using the SML/NJ compiler, input the RandSpec() command into the SML/NJ compiler for several times, check the correctness of the output information of the variables, enabled actions, simulation trace.
- Test user-guide mode: given a SML file, after running this file using the SML/NJ compiler, input one of all the actions command (for example, Send()) into the SML/NJ compiler, check the correctness of the output information of the variables, enabled actions, simulation trace.

- Test the generated graph: given the Initial.txt file that contains all the action names and the SimulationTrace.txt file that contains a serials of actions, run the graph generator algorithms using the Java JDK to generate a dot file. After that, run the GraphViz tool to generate GIF file given the dot file. Check the correctness of this graph. Make sure it corresponds with the Initial.txt file and the SimulationTrace.txt file.

Using the above method to systematically test the simulator with several examples, the simulator works well. The current values of variables, the simulation trace, and the enabled actions generated at every step are correct. The MSC and AG graphs are generated correctly.

The MSC graph can be improved further. More information can be added into the graph at every step, such as transition variables, some label, etc.

The simulator can be further test using the GUI after integrating it into the GUI. This is described in the Integration testing.

10.1.3 GUI Testing

The GUI is an important unit of the GTLA system. It provides the user much functionality. The purpose of the GUI testing is to systematically check if all the functionality works well.

- Open a file:

Select File>Open TLA Spec... (Open Configuration Spec...), the file chooser dialog displays; choose the desired TLA (CFG) file through the file chooser dialog and press the button Open, the desired file is displayed in the system editor of the GUI. Make sure the desired file is correctly stored in the temporary directory "...Workspace/Sourcefiles" for future use.

- Save/Save as:

Select File>Save System, the TLA and CFG files are saved in the original directory. Make sure the TLA and CFG files in this directory are updated.

Select File>Save System As..., the file chooser dialog displays; choose the directory to save the TLA and CFG files and press the button save. Make sure these files are saved in the chosen directory.

- Create a new file

Select File>New System, if there are some unsaved TLA and CFG files opened in the system editor, a query dialog displayed to ask the user whether save those files and clear the contents in the system editor; otherwise the contents in the system editor are cleared in order to let the user edit new files.

- Check Syntax

Select File>Check Syntax, the check syntax window displayed and the syntax checking results are displayed in this window; when the user clicks the button OK, the window is closed. Make sure the syntax checking result is the one of the current TLA file.

- Exit

Select File>Exit, if there are some unsaved TLA and CFG files opened in the system editor, a query dialog displayed to ask the user whether save those files and close the GTLA system; otherwise the GTLA system is closed.

- Undo/Redo

Select Edit>Undo (Redo), the users are allowed to correct their mistakes. Make sure the last (undo) action the user just performed is unexecuted (re-executed).

- About

Select Help>About, the about dialog displayed to show information about the GTLA system; click the button OK, the dialog is closed.

- System editor

Select System Editor, make sure the styled text area can be used to edit the current file displayed in it, including add, delete, cut, copy text; make sure the project tree can be used to switch the contents of the styled text area between the TLA file and the CFG file.

- Simulator

Select Simulator, run several examples to make sure the information of enabled actions, variables and simulation trace can be displayed and updated in the corresponding lists correctly; make sure the MSC and AG graphs can be displayed and updated in the corresponding canvases correctly; click the next button for several times and make sure the simulator runs one step every time; click the reset button and make sure the contents of all the lists and canvases are reinitialized; click the random button and make sure the simulator runs automatically and correctly for amount time period; change the scale bar from slow to fast and make sure in the random mode the time period between two steps changes from a small time period to a large time period.

- Verifier

Select Verifier; make sure the verification information displayed in the verifier page correctly.

Using the above methods to systematically test the GUI of the GTLA system, almost all the functionality works well. But there are still some points need further refinements:

- Every time when the user clicks the Simulator tab item, the simulator starts and the initial information is loaded into the simulator page. It should be made sure that the initial information can be loaded into the simulator page only after all the initial information is generated, otherwise missing parts of information may occur. To make sure this, before loading new information into the simulator page the thread used to load information into the GUI should be set to sleep for a feasible time period, 10000 millisecond is set as below. It need further adjust.

Thread.sleep (10000);

- A scale bar is used to adjust the random simulation speed, that is the time period between two steps. This is implemented by setting the sleeping time period of the thread used to load information into the GUI. The time period is set to be between 0 and 3000 millisecond. A variable RandomTime is updated respond to the user, as below. It need further adjust.

Thread.sleep (RandomTime);

10.2 Integration Testing

To test a system correctly and completely, unit testing is the first step. The following step is to test the integrated system. The purpose is to check if all the units are integrated into the GTLA system and work well.

The method for the integration testing is that running the GTLA system with several examples. There are many tools integrated into the GTLA system: a syntactic analyzer used for syntax check, the TLA+ Front-End tool, GNU compiler, SML/NJ compiler, GraphViz tool, Java SDK used for the simulator, the TLC model checker used for the verifier. These three functionalities will be tested after loading a TLA+ system into the GTLA system:

1. If the check syntax functionality works correctly
2. If the simulator works correctly
3. If the verifier works correctly

Several examples are used for the integration testing; refer to appendix D – Testing Examples. Results are given below.

➤ **Syntax check**

After clicking File→Check Syntax, the check syntax window displayed and the corresponding information generated from the syntactic analyzer displayed in this window correctly and synchronously.

➤ Simulator

In the unit testing section, the simulator has been already tested and worked fine. After integrating it into the GUI, we should test if it works fine.

- Select Simulator to start running the simulator; make sure the initial information displayed in the simulator page is correct.
- Choose any enabled actions at each step, press the button Next, check the correctness of the information displayed in the simulator page.
- Press the button Random to run the simulator for several time periods; check the correctness of the information displayed in the simulator page.
- Run the simulator in the random mode, change the speed of the simulation by updating the scale bar from slow to fast, check if the speed of the simulation changed.

Basically, the simulator works fine, but there are still some details need to be improved:

- At every step of the simulation, the information displayed slowly in the simulator page because it takes a long time period the simulator to generate all the information.
- When the user clicks the simulator tab item again, the simulator should start again and the initial information should be reloaded into the simulator page again. But sometimes the simulator page is not re-initialized completely. This may be caused by the long time initialization of the simulator to generate all the information again.

➤ Verifier

After clicking the Verifier tab to enter the verifier page, the TLC model checker starts to validate the given TLA+ system in the model-checking mode; the results are displayed in the verifier page correctly and synchronously to the user.

10.3 Conclusion

After developing the GTLA system, each unit and the integrated system were tested. The GTLA translator is responsible for translating a given TLA+ specification and a configuration specification into an executable SML file. Excluded the restrictions of the TLA+ language, the left subset of TLA+ language can be successfully translated into SML language. The simulator is responsible for providing information of variables, enabled actions, etc. to the user. The GUI provides the user all the

functionality. They work fine, but there are some details can be improved, such as the MSC graph generated by the simulator, the scale bar that controls the simulation speed in the GUI, etc. All the unit and related tools are integrated into the GTLA environment, basically the whole GTLA system woks fine. Although, the simulator communicates with so many tools, its performance is not perfect.

Chapter 11

Conclusion and Future Work

11.1 Project Conclusion

This thesis consists of two parts. The literature survey part discussed the graphical user interface (GUI) for model checkers and how to develop a GUI for the TLC model checker. The final thesis part presented the design, implementation, testing of the GTLA system, with which the user can run the TLC model checker through a GUI. Except the TLC model checker, the GTLA system also provided other fundamental functionality, such as a system editor, a simulator, etc.

Compared with other model checkers for TLA+ specification (like the TLC model checker), the GTLA system has many benefits. It provides the user a GUI, which bring the user much convenience; it provides the user more functionality, like system editor, simulator, etc.. Even though, it has some insufficiency. The GTLA system can only deal with a subset of the TLA+ language. Although the TLC model checker can't deal with the complete TLA+ language, the GTLA system's TLA+ subset is less than the TLC model checker.

Compared with other model checkers with a GUI (like UPPAAL, Xspin), the GTLA system provides the basic three mechanisms: a system editor, a simulator and a verifier. In terms of the simulator, the GTLA system provides almost the same functionality as the UPPAAL system; but it is less-developed than the Xspin system, which provides many simulation algorithms (like depth-first, shortest path, etc.) and more simulation views to the user. In terms of the verifier, the GTLA system directly calls the TLC model checker to validate the given TLA+ specification, but the generated counter-example can't be re-loaded into the simulator, this is called executable verifier, which is part of the future work.

The implementation of the GTLA system achieves most requirements of the design. The testing shows it works well. But there are still some insufficiencies which need to be improved, like the language translation problems of some expressions encountered in the GTLA translator, the performance of the simulator, etc.

The total work I did in this project is described below. The source code of them can be got from the CD/Source code.

- The GTLA Translator is developed using GNU compiler collection. Around 5500 lines code which is written in C contributes to the GTLA Translator's construction.
- The GUI is developed using Swing Designer tool. Around 1000 lines code which

is written in Java contributes to the GUI's construction.

- The Graph Generators are two algorithms written in Java, totally around 270 lines code are contained.

11.2 Personal Conclusion

This project combines the elementary software developing processes together. From the literature survey and system design to the implementation, testing and refinement, I learned lot from it.

The first step of this project is the literature survey. When I was in the university in China, a project is given with clear system requirements, developing tools and programming language, I can start design and implementation directly. But in terms of this MSc project, I only got a short description of the project at first. I need to find answers of concepts, system requirements, developing tools, programming language, etc. by myself. In fact, this really brought me troubles and sometimes I lost my way and didn't know how to go on. I realized my ability of literature survey was really low, but I also realized literature survey is the most basic ability a software engineer should have. There are some reasons for my low survey ability. First, I lack the experience of survey, so I always don't know how and what to do. Second, I lack a broad knowledge. Without much big project experience and broad knowledge outside the schoolbooks, I felt hard to start when encountered a topic totally new for me. For example, to develop a simulator, I don't know how and what tools to be used. The literature survey improved my ability and I learned how and in which way to do a research. Additionally, I realized one insufficiency of myself, lack of a broad technical knowledge. I decide to improve it by reading more technical papers and books in my daily life.

During the two years study in Computer Science department of TU Delft, I have learned many fundamental courses such as Compiler Construction, Advanced Software Engineering, Distributed, Parallel, Optimization algorithms, etc. Those kinds of knowledge are not only fundamental but also very important for me in future development. In this final project, the GTLA Translator is developed by constructing a compiler using the compiler construction knowledge; the model checking knowledge introduced in Advanced Software Engineering helped me understand and design this project better, and the graph generator algorithms of the GTLA simulator is developed using Java language, which I learned from the Advanced Software Engineering course too. To be a software engineer, although the ability of survey, system design is very important, familiar with basic implementation knowledge is even more important. I still need to improve my skills, ability and understanding of those kinds of knowledge.

Although the development of this project (the GTLA system) can implement the fundamental functionality of a model checker, it would have been much better. The GTLA translator can be improved to translate a bigger subset of the TLA+ language

to SML and get better performance, but because the TLA+ grammar rules is really different from other imperial and functional language and it is a bit complex, in the implementation process sometimes I ignore and miss some details. The GTLA simulator uses too many other tools to assist, so the performance is not good and fluent enough, there must be some better way can be used to make it better. Because in the literature survey part, I didn't find too much possible ways and take all the situations into consideration, some details of the GTLA system doesn't perform well enough.

The Swing Designer of Eclipse tool is really a good and fast GUI builder and it helped me develop the GUI in a short time and got a good performance and successful integration result. The GraphViz and Dot are also very good graph tools. They can generate a graph in a very fast speed, which makes the simulator possible to provide the user some graphs views.

Concluding, the ten months that I spent in this project was really a nice experience for me. Not only did I improve my software engineering skills and ability, but also I found insufficiency of myself and realized how to be a software engineer. I think in my future career or study I must get much better performance.

11.3 Future Work

The elementary GTLA system has been developed. There is still much future work need to be done.

- The GTLA translator translates a subset of the TLA+ language into the SML language. The excluded restrictions and unsolved problems will be examined further and solved, including inheritance, unbounded quant, etc. The details were stated in Chapter 6.
- Type checking and error report will be added into the GTLA Translator to further guarantee the correctness and security of the translation from TLA+ into SML.
- The simulator of the GTLA system needs to be improved. The MSC graph and Action graph generator will be improved to provide the user more information. More simulator views will be examined and provided to the user to describe the simulation process.
- Some facilities will be added into the simulator, such as checking deadlock, checking properties and reporting errors, etc.
- The executable verifier will be developed to load the counter-example into the simulator.

Bibliography

- [1] L. Lamport. (2002). *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley.
- [2] N. E. Fuchs. *Specifications are (preferably) executable*. Software Engineering Journal, 7(5), pages 323-334, 1992.
- [3] I. J. Hayes and C.L B. Jones. *Specifications are not (necessarily) executable*. Software Engineering Journal, 4(6), pages 320-338, 1989.
- [4] A. Gravell. *Executing Formal Specifications Need Not Be Harmful*. Available on the WWW at URL <http://dsse.ecs.soton.ac.uk/amg/papers.html>
- [5] Y. Mokhtari, S. M. (1999). *Animating TLA specifications*. In *LPAR*, pages 92-110.
- [6] P. Hudak. (2000). *The Haskell School of Expression: Learning Functional Programming through Multi-media*. Cambridge University Press, New York, NY.
- [7] B.Gerd, D.Alexandre, G. L. (2004). *A tutorial on uppaal*. In *SFM*, pages 200-236.
- [8] Davie, A. T. J. (1992). *An Introduction to Functional Programming Systems Using Haskell*. Cambridge University Press, Cambridge.
- [9] G.Navratil, D. M. (2003). *Haskell tutorial*.
- [10] J.R. Hubbard, A. H. (2004). *Data structures with Java*. Pearson Prentice Hall.
- [11] Bird, R. and Wadler, P. (1988). *Introduction to Functional Programming*. Series in Computer Science. Prentice-Hall International, Englewood Cliffs, NJ.
- [12] TLA+ Front-End Home Page:
<http://externe.inrs-emt.quebec.ca/users/gregoire/tla-page.html>.
- [13] Eclipse Home Page: <http://eclipsefp.sourceforge.net/>.
- [14] The Haskell 98 Language Report: <http://www.haskell.org/onlinereport/>.
- [15] TLA Web page: <http://research.microsoft.com/users/lamport/tla/tla.html>.
- [16] UPPAAL Home Page: <http://www.uppaal.com/>.
- [17] XSpin Home Page: <http://spinroot.com/spin/Man/GettingStarted.html>.
- [18] Spin Home Page: <http://www.spinroot.com/spin/whatispin.html>

[19] GraphViz Home Page: <http://www.graphviz.org/>.

[20] Emden Gansner and Eleftherios Koutsofios and Stephen North (2002). Drawing graphs with dot.

Appendix A

GTLA User's Guide

Table of Contents

1. Introduction
2. How GTLA Works
3. How to use the System Editor
4. How to use the Simulator
5. How to use the Verifier
6. Conclusion
7. An Example

1. Introduction

GTLA is a graphical user interface for the TLC tool – the TLA+ Model Checker. It provides the standard functionality people expect of a model checker: system editor, simulator, model checker, and so on.

GTLA is a GUI special for the TLC tool, so it is only suitable for the users who want to validate their systems specified by using the TLA+ language.

Through this graphical user interface, the user can:

- Edit the model of system and property specification written in TLA+ language
- Validate the model using the simulation technique
- Validate the model using the model-checking technique provided by the TLC model checker

2. How GTLA Works

To forestall confusion in getting started with GTLA, it's important to know a little bit about how it works.

- Once the user opens the GTLA, he or she can either create new TLA+ (or configuration) specification or load a TLA+ (or configuration) specification from the local disk into the system editor. This specification is stored into a temp directory called “WorkSpace” in the local disk for future use.
- Given the TLA+ system, the user must make sure its correctness before running the simulator or verifier. The “Check Syntax” functionality in the menu File→Check Syntax is used for this purpose. It calls the syntactic analyzer of the TLC model checker to implement this functionality.
- Now the user can run the simulator and the verifier. When the user enters the simulator page, the simulator starts. It first calls the TLA Front-End tool and the Translator to translate the given TLA+ system in the temp directory into the SML executable file. Second it starts the SML/NJ compiler and loads the SML file into the compiler. Now the simulator is ready to run. The user can run the simulator manually or automatically. No matter in which mode, at each step of the simulation, the simulator inputs executable commands into the SML/NJ compiler to execute one step and output useful information files into the temp directory, such as variables’ value, action trace, etc.. It also calls the Graph Generator algorithms and GraphViz tool to generate the MSC and AG graphs which provide different views of the simulation to the user. The generated information and graphs are loaded into the GUI and displayed to the user.
- When the user enters the verifier page, the verifier calls the TLC model checker to automatically verify the TLA+ system in the temp directory and display the results in the GUI for the user.

The detailed information about how GTLA works can be gained from the following paragraph and the example.

3. How to use the System Editor

The system editor consists of two windows, see Figure 1. In the left window there is a project tree with two children: a “TLA Spec” and a “CFG Spec”. Due to the fact that a TLA system must have two files, one specification file and one configuration file, to run a TLA system the user must load both two files into the software. With this project tree, the user can switch between the specification file and the configuration file. In the right window there is an editor, which is used to display the contents of either the specification file or the configuration file to the user. It allows the user to edit these files, including add, delete, cut, copy, etc.

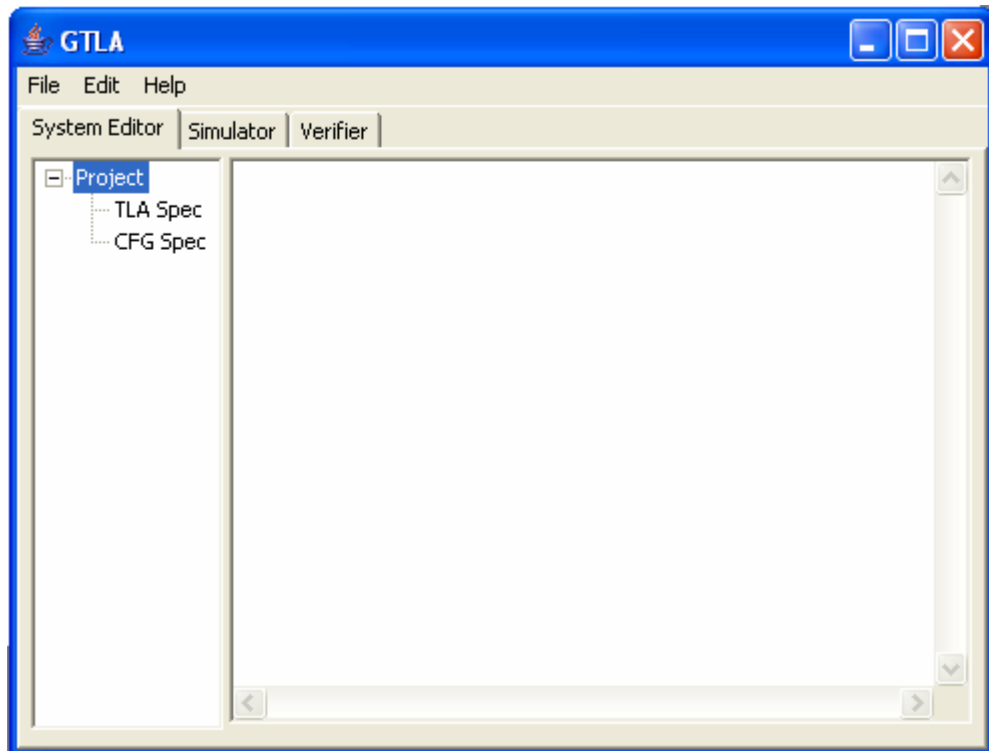


Figure 1: The System Editor window.

The system editor provides some functionality, which can be gained from the menu, see Figure 2. The menu consists of a “File” list, an “Edit” list and a “Help” list. Each submenu provides some functionality.

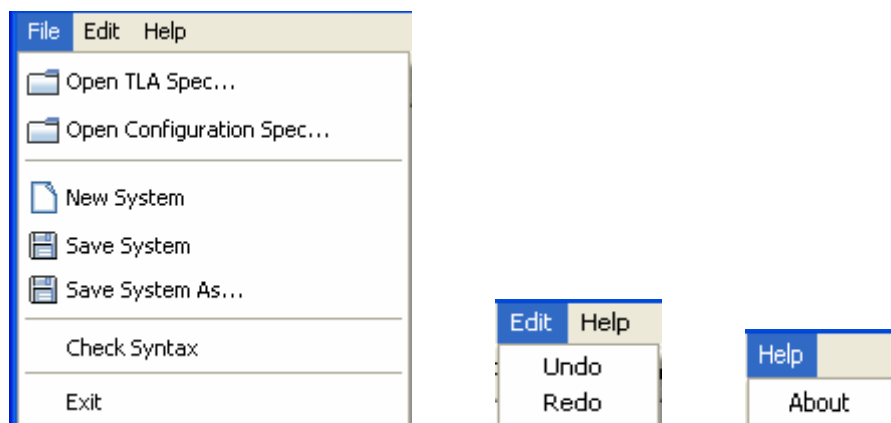


Figure 2: The Menu.

The submenu “File” provides the following functionality.

- “Open TLA Spec...” opens a .tla file (TLA+ specification), with the contents

being displayed in the system editor's text-editing area of the application. It filters non-TLA files, only displaying directories and .tla files in a file chooser dialog, see Figure 3.

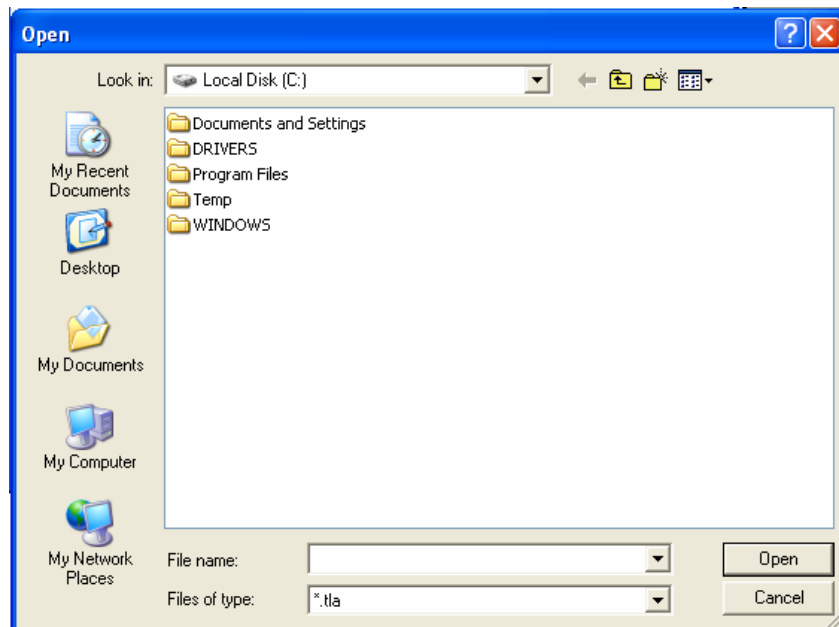


Figure 3: The file chooser dialog.

- “Open Configuration Spec...” opens a .cfg file (configuration file), with the contents being displayed in the system editor's text-editing area of the application. It filters non-CFG files, only displaying directories and .cfg files in a file chooser dialog.
- “New System” let the user to create a new TLA system in the text editing area.
- “Save System” saves the contents of the text-editing area into the original place for later use.
- “Save System As...” saves the contents of the text-editing area into a new place in the local disk for later use.
- “Check Syntax” calls the syntax analyzer to find the syntax error of the given TLA specification and returns the syntax analyzer report in a new dialog, see Figure 4. After the user pressing the “OK” button, the dialog is closed.

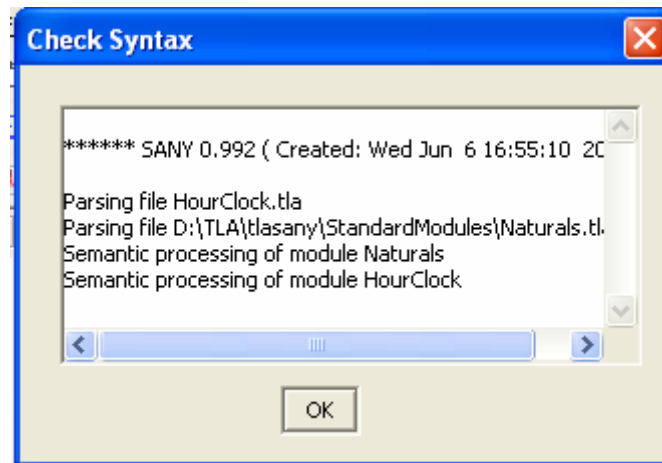


Figure 4: The syntax checker dialog.

- “Exit” exits the application.

The submenu “Edit” provides the undo/redo mechanism, which allows users to correct their mistakes and also to try out different aspects of the application without risk of repercussions.

- “Undo”: unexecuted the last action the user just performed.
- “Redo”: re-execute the last undone action.

When the user selects Help → About from the submenu list “Help”, the “About” dialog is displayed with information about the application, see Figure 5.



Figure 5: The “About” dialog.

4. How to use the Simulator

The simulator consists of five parts, see Figure 6. It provides the following functionality:

- The “Enabled Actions” displays all the available transitions in the next step and can be chosen by the user.

- The “Simulation Trace” displays the action trace at every step.
- The “Variables” shows the values of variables at every step.
- The “Action graph” (the window at the top right) shows all the actions and the active actions or transitions at every step.
- The “MSC graph” (the window at the bottom right) shows the transitions between the different actions at every step.

The simulator can be used in two ways:

- In the user-guided mode the user can run the system manually and choose which available transition to take. The “Next” button is used to execute one step. It executes either the action chosen by the user in the “Enabled Actions” list or a randomly chosen action. The “Reset” button is used to restart the simulator.
- The random mode can be toggled to let the system run on its own, randomly choosing an available transition to take. The “Random” button is used to enter the random mode. The scale bar is used to set the execution speed of the random mode.

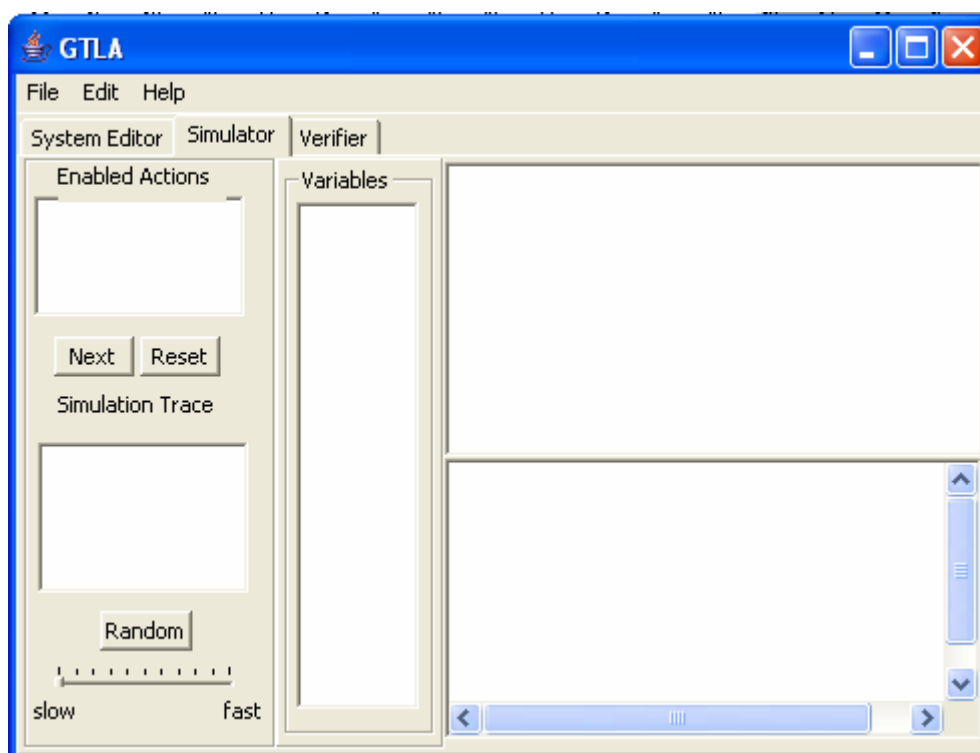


Figure 6: The simulator window.

5. How to use the Verifier

The Verifier consists of only one SWT widget, a StyledText object which is used to display the model-checking information generated by the TLC model checker. It doesn't allow the user to edit the text. When the user presses the "Verifier" tab item, the software calls the TLC model checker the systematically finds errors in the current TLA system and displays the message generated by the TLC model checker on the verifier window synchronously. See Figure 7.

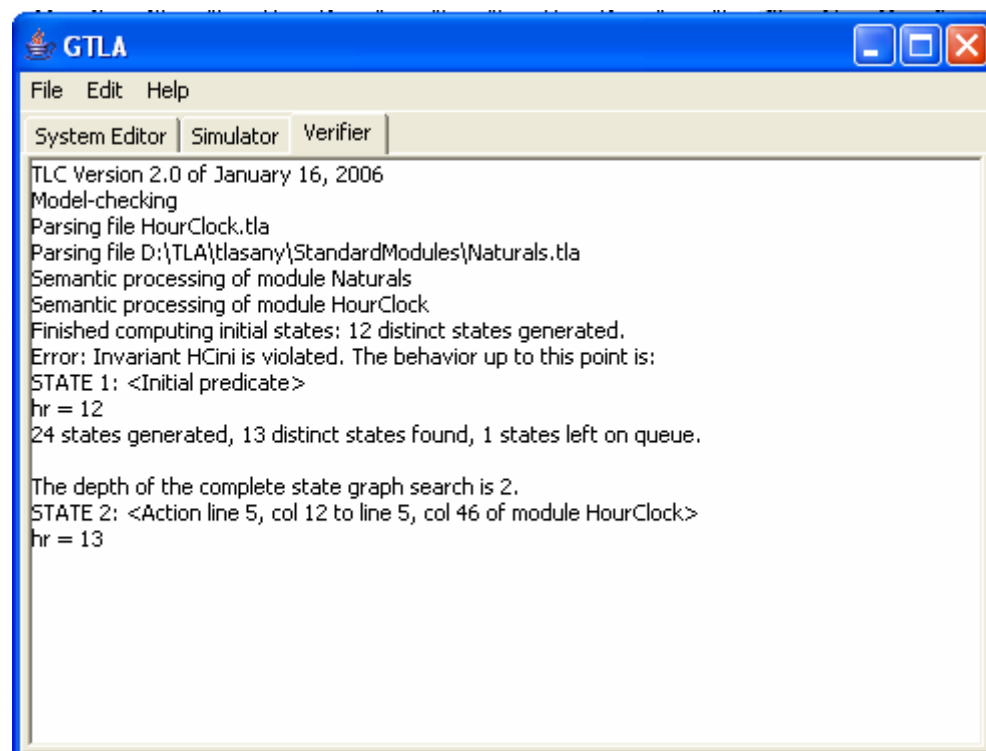


Figure 7: The verifier window.

6. Conclusion

The GTLA system, examples and supporting tools are provided in the CD. To run the GTLA system, you should install all the following tools correctly in your local disk. All these tools are provided in CD/Tools.

- Install GraphViz and Dot tool.
- Install Java JDK 1.5.
- Install TLA+ Front-End tool under local disk (d:).
- Install the TLC model checker under local disk (d:).

- Install the GNU compiler collection under local disk (d:), and put the src directory under the path ../home/.
- Put the Workspace directory which includes graph generators and temporary directories under local disk (c:).

After all the tools installed correctly, you can start using the GTLA system with the given examples in CD/Examples.

Finally, thank you for trying GTLA! I hope this GTLA User Guide has helped you get started with GTLA. If you have questions that are not answered here, send an email to the author: wanbo_hl@hotmail.com.

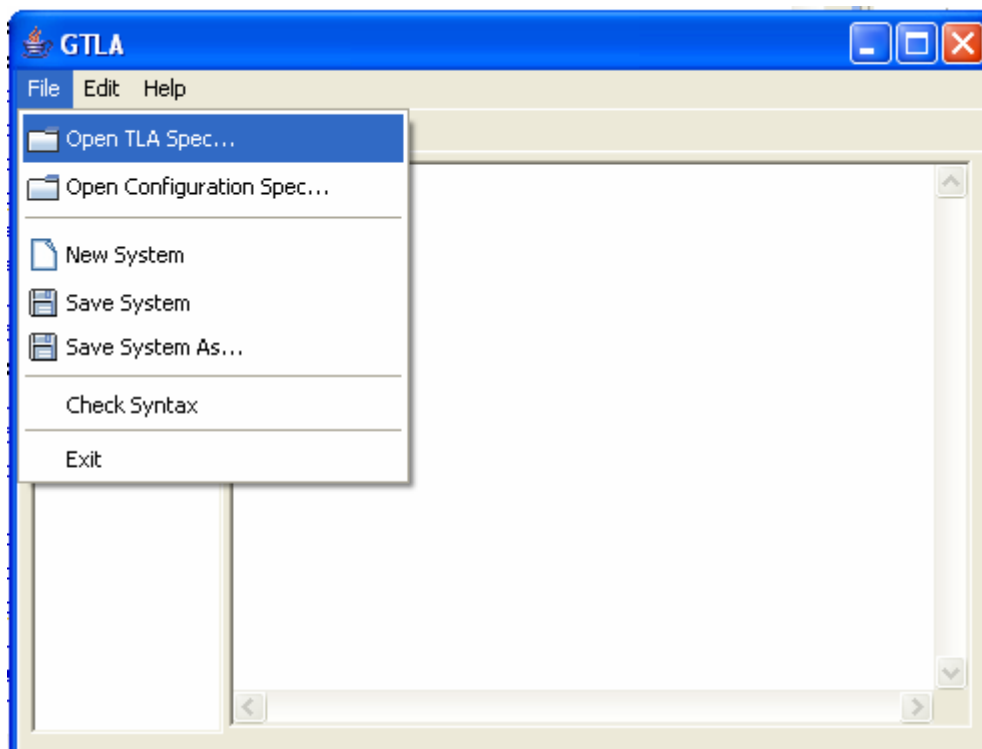
Happy GTLA!

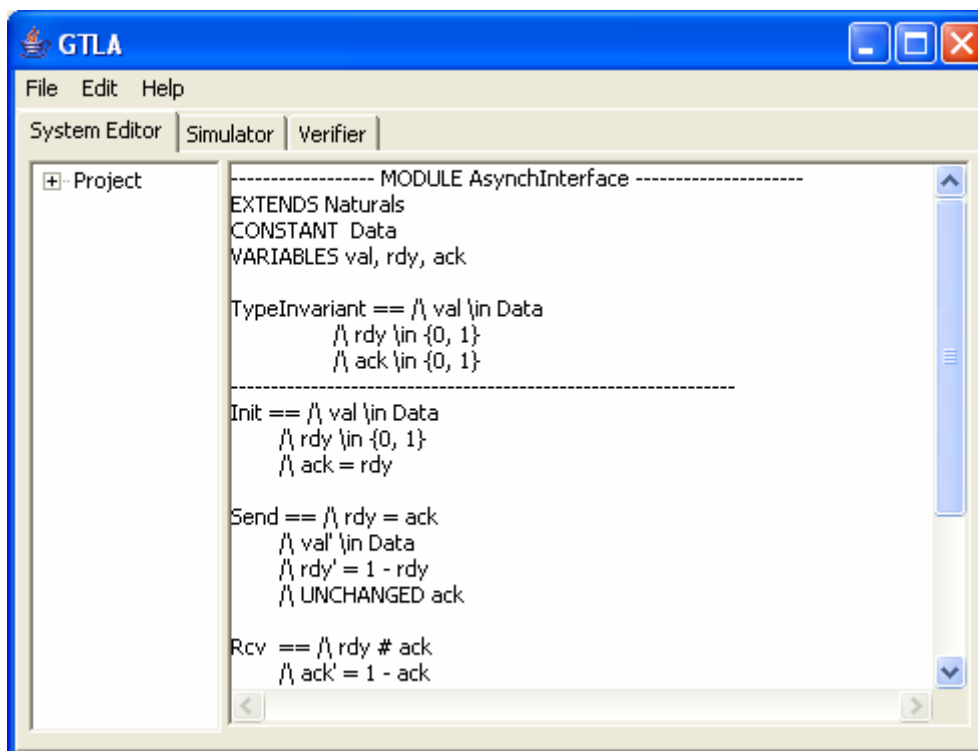
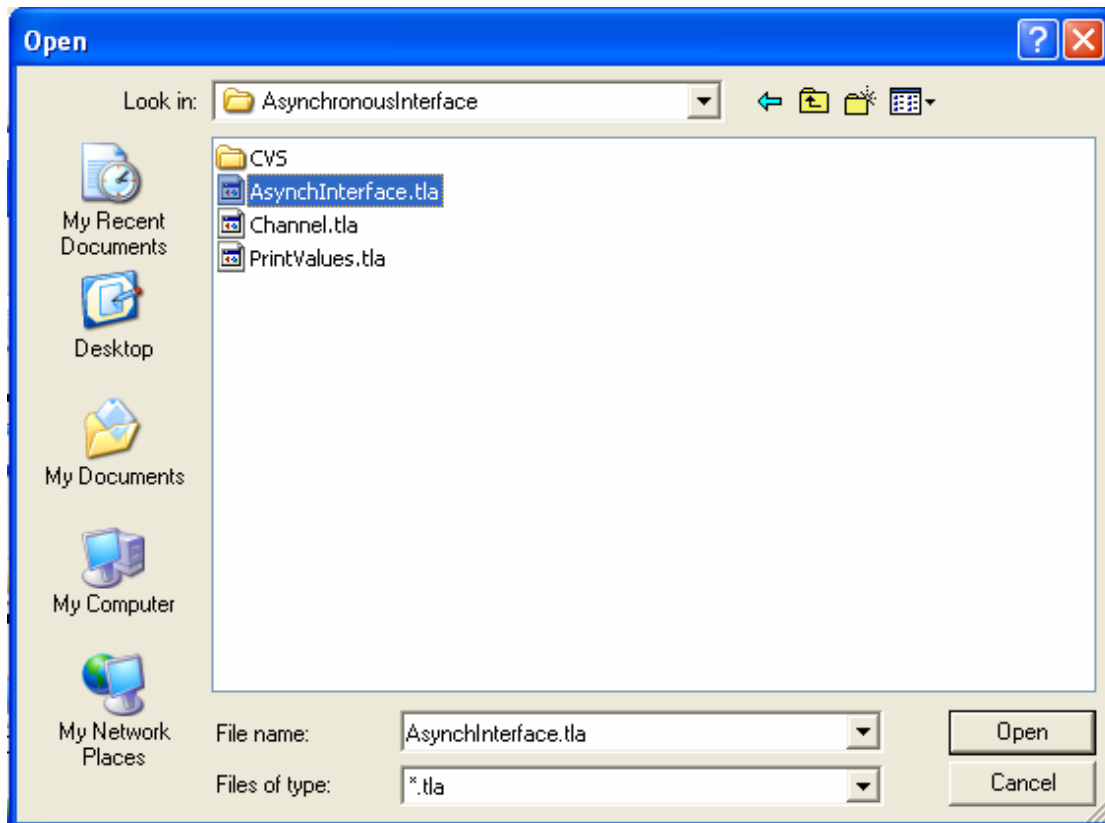
7. An Example

7.1 Load a TLA+ system into GTLA through “open file” functionality.

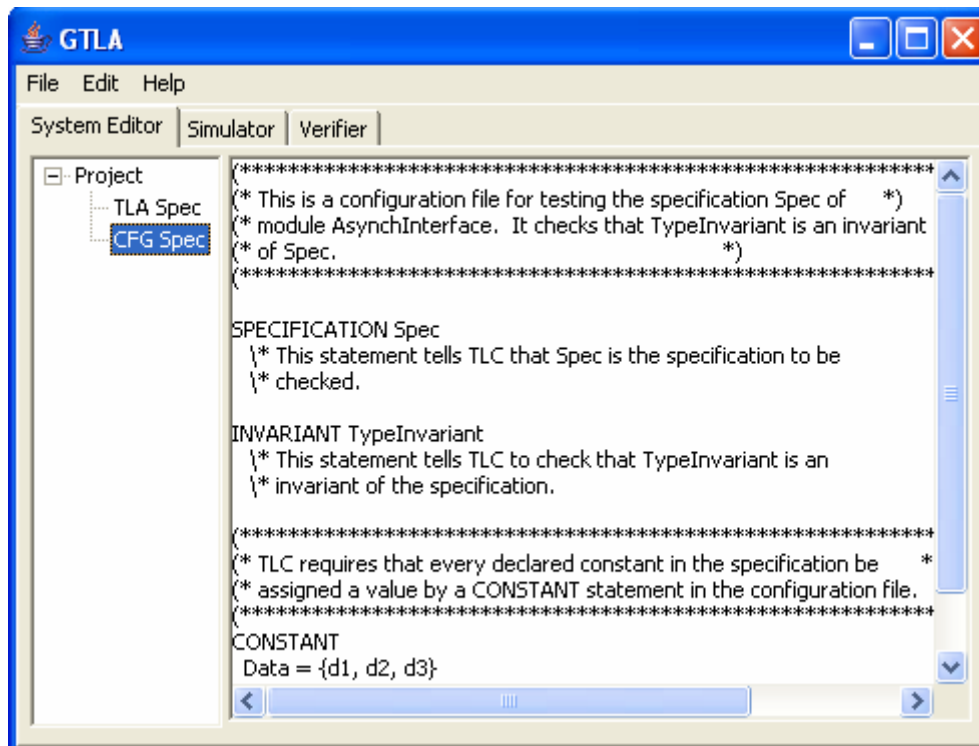
Press the “Open TLA Spec...” item in the submenu “File”, the file chooser dialog shows. From this dialog choose the TLA file desired to load into the application. After that the contents of the TLA file is displayed in the text editing area.

Use the same way to load the configuration file into the application by pressing the “Open Configuration Spec...” from the submenu “File”.



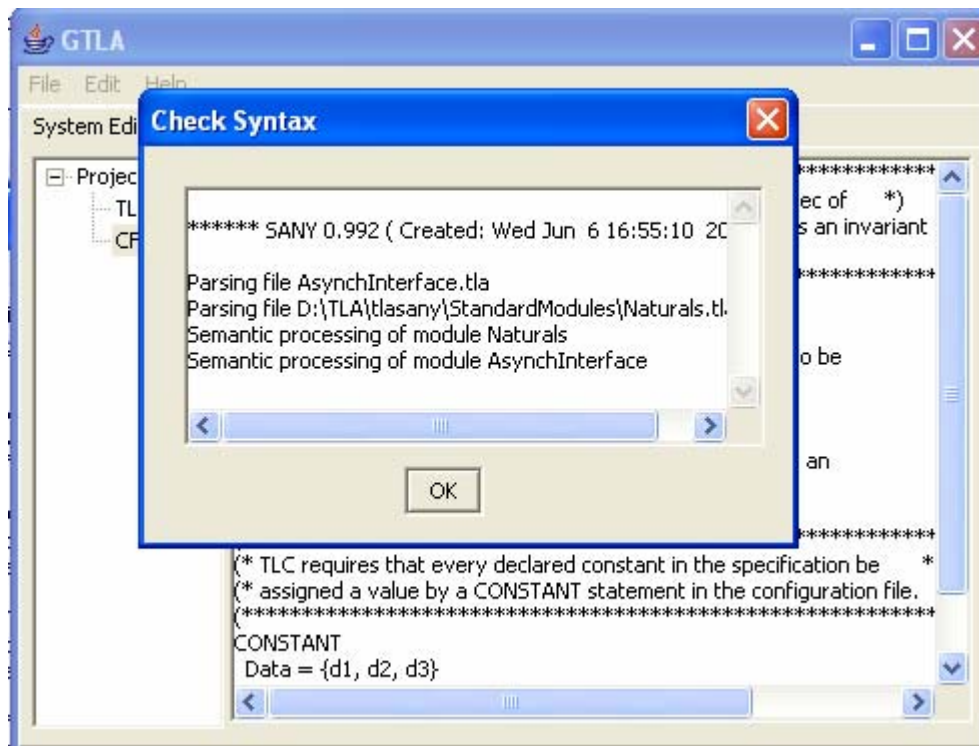


By using the project tree, the user can switch between the TLA and CFG files. The user can edit these files in the text editing area.



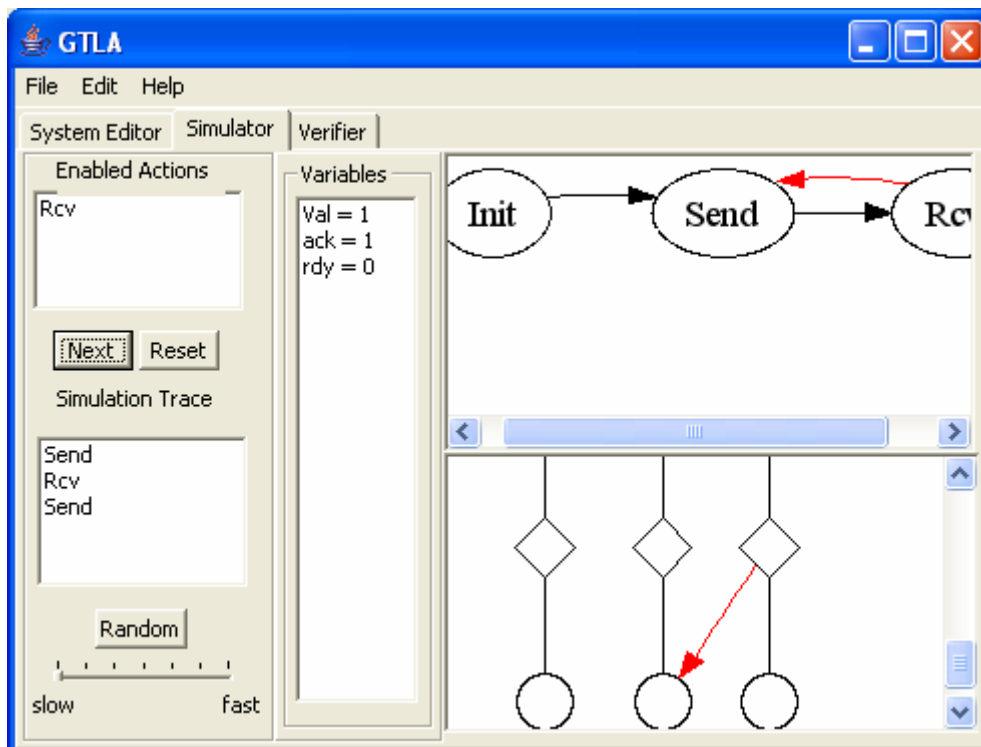
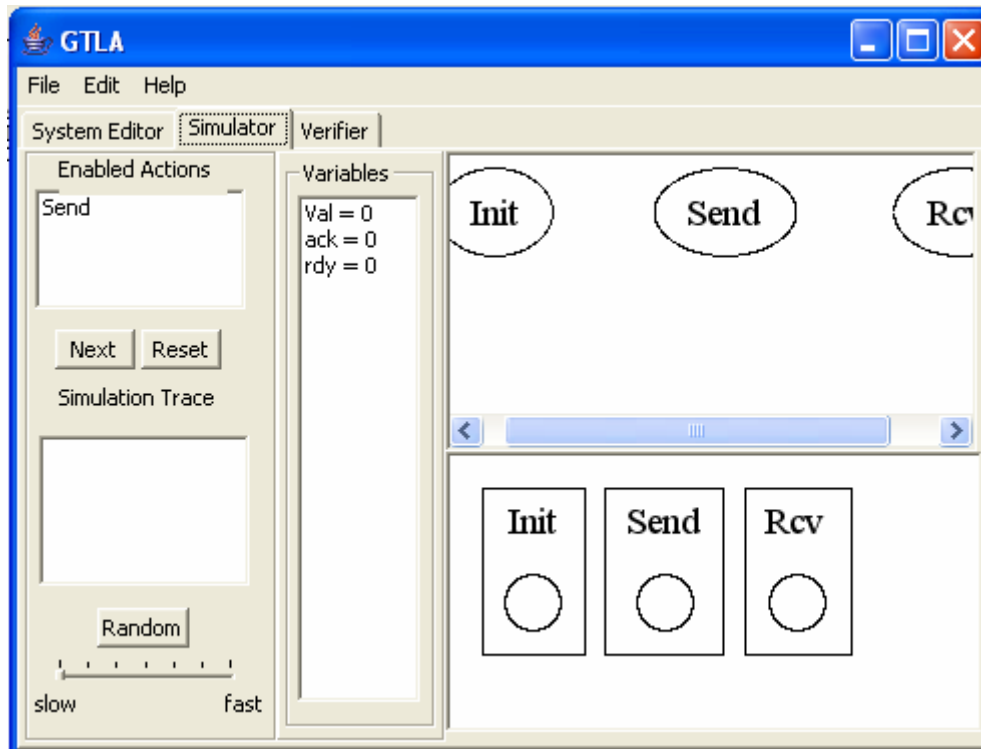
7.2 Check Syntax

When the user clicks File→Check Syntax, the following window is displayed to show the checking result to the user.



7.3 Run the Simulator

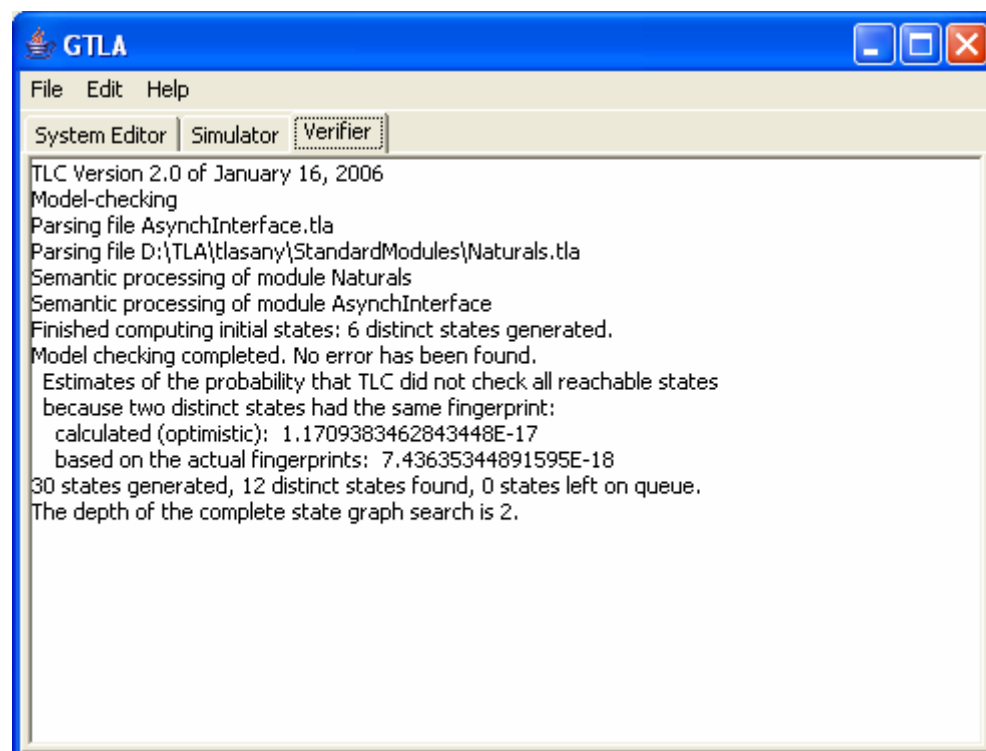
When the user clicks the Simulator tab item, the simulator starts. The following graph shows the initial window of the simulator to the user.



The user can run the simulator either manually by using the “Next” button or automatically by using the “Random” button. The following graph is an example after the user running several steps manually.

7.4 Run the Verifier

When the user clicks the Verifier tab item, the verifier starts. It calls the TLC model checker to verify the given TLA+ system and the verification result is displayed in the verifier window, as shown below.



Appendix B

Lexical Conventions

The complete token descriptions are given below, which provides the token table for the other parts of the GTLA Translator, such as the lexical and syntax analysis, etc. It is part of the lex.l file of the GTLA Translator's source files, which can be found in CD/Source code.

```
IDENTIFIER    [A-Za-z_0-9]*[A-Za-z][A-Za-z_0-9]*
DEC_CONST     [0-9]+
HEX_CONST     \\[Hh][0-9A-Fa-f]+
OCT_CONST     \\[Oo][0-7]+
BOOL_CONST    \\[Bb][01]+
REAL_CONST    [0-9]*\\.[0-9]+
NUMBER {DEC_CONST}| {HEX_CONST}| {OCT_CONST}| {BOOL_CONST}|
        {REAL_CONST}
NON_ESCAPED    [^\\n\\"]
ESCAPED    [ntvbrfa\\\\"?\\"]|[0-7]{1,3}|x[0-9A-Fa-f]+
CHAR_SYMBOL    {NON_ESCAPED}|\\{ESCAPED}
STRING    ({CHAR_SYMBOL}|\\)*
%x           COMMENT
%%
[ \\t\\r]+
\\n           ++ current_line;
"(*"         comment_level = 1; comment_start = current_line; BEGIN(COMMENT);
<COMMENT>
{
    "(*"      ++ comment_level;
    "*)"      if (-- comment_level == 0) BEGIN(INITIAL);
    "\\n"      ++ current_line;
    <<EOF>>    { error(0, "unterminated comment starting at line %u",
                    comment_start); return 0; }
}

"% % Parse Tree"
"N_Module"      return START;
"N_BeginModule" return N_Module;
"N_Extends"     return N_BeginModule;
"N_Body"        return N_Extends;
"N_EndModule"   return N_Body;
"N_VariableDeclaration" return N_EndModule;
"N_ParamDeclaration" return N_VariableDeclaration;
"N_OperatorDefinition" return N_ParamDeclaration;
"N_FunctionDefinition" return N_OperatorDefinition;
"N_Instance"    return N_FunctionDefinition;
return N_Instance;
```

"N_Assumption"	return N_Assumption;
"N_ModuleDefinition"	return N_ModuleDefinition;
"N_Theorem"	return N_Theorem;
"N_ConsDecl"	return N_ConsDecl;
"N_IdentDecl"	return N_IdentDecl;
"N_QuantBound"	return N_QuantBound;
"N_Instantiation"	return N_Instantiation;
"N_IdentLHS"	return N_IdentLHS;
"N_IdPrefix"	return N_IdPrefix;
"N_IdPrefixElement"	return N_IdPrefixElement;
"N_OpArgs"	return N_OpArgs;
"N_GenInfixOp"	return N_GenInfixOp;
"N_GenPrefixOp"	return N_GenPrefixOp;
"N_GenPostfixOp"	return N_GenPostfixOp;
"N_GeneralId"	return N_GeneralId;
"N_OpApplication"	return N_OpApplication;
"N_PrefixExpr"	return N_PrefixExpr;
"N_InfixExpr"	return N_InfixExpr;
"N_PostfixExpr"	return N_PostfixExpr;
"N_ParenExpr"	return N_ParenExpr;
"N_BoundedQuant"	return N_BoundedQuant;
"N_UnboundedQuant"	return N_UnboundedQuant;
"N_UnBoundedOrBoundedChoose"	return N_UnBoundedOrBoundedChoose;
"N_SetEnumerate"	return N_SetEnumerate;
"N_SubsetOf"	return N_SubsetOf;
"N_SetOfAll"	return N_SetOfAll;
"N_FcnAppl"	return N_FcnAppl;
"N_FcnConst"	return N_FcnConst;
"N_SetOfFcns"	return N_SetOfFcns;
"N_RcdConstructor"	return N_RcdConstructor;
"N_FieldVal"	return N_FieldVal;
"N_SetOfRcds"	return N_SetOfRcds;
"N_FieldSet"	return N_FieldSet;
"N_Except"	return N_Except;
"N_ExceptSpec"	return N_ExceptSpec;
"N_ExceptComponent"	return N_ExceptComponent;
"N_ActionExpr"	return N_ActionExpr;
"N_IfThenElse"	return N_IfThenElse;
"N_Case"	return N_Case;
"N_CaseArm"	return N_CaseArm;
"N_OtherArm"	return N_OtherArm;
"N_LetIn"	return N_LetIn;
"N_ConjList"	return N_ConjList;
"N_ConjItem"	return N_ConjItem;
"N_DisjList"	return N_DisjList;
"N_DisjItem"	return N_DisjItem;
"N_Number"	return N_Number;
"N_Tuple"	return N_Tuple;
"N_RecordComponent"	return N_RecordComponent;

"N_LetDefinitions"	return N_LetDefinitions;
"N_MaybeBound"	return N_MaybeBound;
"MODULE"	return MODULE;
"EXTENDS"	return EXTENDS;
"CONSTANT"	return CONSTANT;
"CONSTANTS"	return CONSTANTS;
"VARIABLE"	return VARIABLE;
"VARIABLES"	return VARIABLES;
"THEOREM"	return THEOREM;
"IF"	return IF;
"THEN"	return THEN;
"ELSE"	return ELSE;
"CASE"	return CASE;
"OTHER"	return OTHER;
"CHOOSE"	return CHOOSE;
"INSTANCE"	return INSTANCE;
"WITH"	return WITH;
"LET"	return LET;
"IN"	return IN;
"EXCEPT"	return EXCEPT;
"ASSUME"	return ASSUME;
"ASSUMPTION"	return ASSUMPTION;
"AXIOM"	return AXIOM;
"LOCAL"	return LOCAL;
"SPECIFICATION"	return SPECIFICATION;
"\E"	yylval.token = new_token();return EXIST;
"\A"	yylval.token = new_token();return FORALL;
"\EE"	yylval.token = new_token();return EEXIST;
"\AA"	yylval.token = new_token();return FFORALL;
"<-"	yylval.token = new_token();return INST;
"->"	yylval.token = new_token();return CASEOP;
" ->"	yylval.token = new_token();return FIELDDOP;
"_ "	yylval.token = new_token();return Actionunderline;
"!!"	yylval.token = new_token();return DCLAIM;
"##"	yylval.token = new_token();return DWELL;
"\$\$"	yylval.token = new_token();return DDOLLAR;
"%%"	yylval.token = new_token();return DDIV;
"&&"	yylval.token = new_token();return DAND;
"(+)"	yylval.token = new_token();return BRPLUS;
"(-)"	yylval.token = new_token();return BRSUB;
"(.)"	yylval.token = new_token();return BRDOT;
"(/)"	yylval.token = new_token();return BRDIV;
"(\X)"	yylval.token = new_token();return BRTIMES;
"**"	yylval.token = new_token();return DSTAR;
"++"	yylval.token = new_token();return DPLUS;
"-+>"	yylval.token = new_token();return TEMPORAL;
"--"	yylval.token = new_token();return DSUB;

"- "	yylval.token = new_token();return OVEROR;
".."	yylval.token = new_token();return TWODOT;
"..."	yylval.token = new_token();return THREEDOT;
"//"	yylval.token = new_token();return DIFFERENT;
"/="	yylval.token = new_token();return NOTEQUAL;
"\\\""	yylval.token = new_token();return CONJ;
"::="	yylval.token = new_token();return BNFDEFINE;
":=\""	yylval.token = new_token();return ASSIGN;
yylval.token = new_token();return FCNCONSTRUCTOR;	
"<:\""	yylval.token = new_token();return OPFC;
"<=>\""	yylval.token = new_token();return EQUAL;
"<=\""	yylval.token = new_token();return SMALLERTHAN;
">=\""	yylval.token = new_token();return IMPLY;
"= \""	yylval.token = new_token();return OPBA;
">=\""	yylval.token = new_token();return LARGER THAN;
"??\""	yylval.token = new_token();return DQUE;
"@\""	yylval.token = new_token();return ET;
"@@\""	yylval.token = new_token();return DET;
"\\\""	yylval.token = new_token();return DISJ;
"^^\""	yylval.token = new_token();return DEX;
" -\""	yylval.token = new_token();return OROVER;
" =\""	yylval.token = new_token();return BOOLASSIGN;
" \""	yylval.token = new_token();return DOR;
"~>\""	yylval.token = new_token();return LEADSTO;
"\approx\""	yylval.token = new_token();return Approx;
"\geq\""	yylval.token = new_token();return Geq;
"\oslash\""	yylval.token = new_token();return Oslash;
"\sqsupseteq\""	yylval.token = new_token();return Sqsupseteq;
"\asymp\""	yylval.token = new_token();return Asymp;
"\gg\""	yylval.token = new_token();return Gg;
"\otimes\""	yylval.token = new_token();return Otimes;
"\star\""	yylval.token = new_token();return Star;
"\bigcirc\""	yylval.token = new_token();return Bigcirc;
"\in\""	yylval.token = new_token();return Belong;
"\prec\""	yylval.token = new_token();return Prec;
"\subset\""	yylval.token = new_token();return Subset;
"\bullet\""	yylval.token = new_token();return Bullet;
"\intersect\""	yylval.token = new_token();return Intersect;
"\preceq\""	yylval.token = new_token();return Preceq;
"\subseteq\""	yylval.token = new_token();return Subseteq;
"\cap\""	yylval.token = new_token();return Cap;
"\land\""	yylval.token = new_token();return Land;
"\propto\""	yylval.token = new_token();return Propto;
"\succ\""	yylval.token = new_token();return Succ;
"\cdot\""	yylval.token = new_token();return Cdot;
"\leq\""	yylval.token = new_token();return Leq;
"\sim\""	yylval.token = new_token();return Sim;
"\succeq\""	yylval.token = new_token();return Succeq;
"\circ\""	yylval.token = new_token();return Circ;

"ll"	yylval.token = new_token();return Ll;
"\simeq"	yylval.token = new_token();return Simeq;
"\supset"	yylval.token = new_token();return Supset;
"\cong"	yylval.token = new_token();return Cong;
"\lor"	yylval.token = new_token();return Lor;
"\sqcap"	yylval.token = new_token();return Sqcap;
"\supseteq"	yylval.token = new_token();return Supseteq;
"\cup"	yylval.token = new_token();return Cup;
"\circ"	yylval.token = new_token();return O;
"\sqcup"	yylval.token = new_token();return Sqcup;
"\union"	yylval.token = new_token();return Union;
"\div"	yylval.token = new_token();return Div;
"\odot"	yylval.token = new_token();return Odot;
"\sqsubset"	yylval.token = new_token();return Sqsubset;
"\uplus"	yylval.token = new_token();return Uplus;
"\doteq"	yylval.token = new_token();return Doteq;
"\ominus"	yylval.token = new_token();return Ominus;
"\sqsubseteq"	yylval.token = new_token();return Sqsubseteq;
"\wr"	yylval.token = new_token();return Wr;
"\equiv"	yylval.token = new_token();return Equiv;
"\oplus"	yylval.token = new_token();return Oplus;
"\sqsupset"	yylval.token = new_token();return Sqsupset;
"\lnot"	yylval.token = new_token();return LNOT;
"\neg"	yylval.token = new_token();return NEG;
"[]"	yylval.token = new_token();return ALWAYS;
"<>"	yylval.token = new_token();return NALWAYS;
"DOMAIN"	yylval.token = new_token();return DOMAIN;
"ENABLED"	yylval.token = new_token();return ENABLED;
"SUBSET"	yylval.token = new_token();return SUBSET;
"UNCHANGED"	yylval.token = new_token();return UNCHANGED;
"UNION"	yylval.token = new_token();return UNION;
"^_"	yylval.token = new_token();return EXPLUS;
"^*"	yylval.token = new_token();return EXSTAR;
"^#"	yylval.token = new_token();return EXWELL;
{ IDENTIFIER }	yylval.token = new_token(); return IDENTIFIER;
{ NUMBER }	yylval.token = new_token(); return NUMBER;
{ STRING }	yylval.token = new_token(); return STRING;
[{\}\[\]\- =,\(\)\'\!:]	yylval.token = new_token(); return *yytext;
[#\\$\%\&* \+ \- \. = < > \^ /~]	yylval.token = new_token(); return *yytext;

Appendix C

GTLA Grammar

The GTLA grammar is the grammar of the flattened parse tree generated from the TLA+ Front-End tool given a TLA+ specification and a configuration file. The following is the main part of the grammar rule, which is part of the grammar.g file of the GTLA Translator's source files, which can be found in CD/Source code.

```
specification → declarations
declarations → spec_declarations
               conf_declarations
conf_declarations → [c_specdeclaration]? [c_paramdeclaration]*
c_paramdeclaration → CONSTANT
                   IDENTIFIER
                   '='
                   '{'
                   [ IDENTIFIER | NUMBER ] {
                   [ ',' [ IDENTIFIER | NUMBER ] }*
c_specdeclaration → SPECIFICATION
                   IDENTIFIER
spec_declarations → START
                  N_Module '{'
                  s_beginmodule
                  s_extends
                  N_Body '{'
                  [ s_body | AtLeast4('-') LCNumber ]*
                  '}'
                  N_EndModule '{'
                  AtLeast4('=') LCNumber
                  '}'
                  '}'
s_beginmodule → N_BeginModule '{'
               AtLeast4('-') MODULE LCNumber
               IDENTIFIER LCNumber
               AtLeast4('-') LCNumber
               '}'
s_extends → N_Extends '{'
           [ EXTENDS LCNumber
           IDENTIFIER LCNumber
           [ ',' LCNumber
           IDENTIFIER LCNumber
           ]*
           ]?
           '}'
```

$s_body \rightarrow s_variabledeclaration \mid s_paramdeclaration \mid [LOCAL\{\}] s_local \mid$
 $s_assumption(list) \mid s_theorem(list) \mid spec_declarations(list);$
 $s_local \rightarrow s_operatordefinition(list) \mid s_functiondefinition(list) \mid s_instance(list)$
 $\mid s_moduledefinition(list);$
 $s_variabledeclaration \rightarrow N_VariableDeclaration \{'$
 $\quad [VARIABLE \mid VARIABLES] LCNumber$
 $\quad IDENTIFIER LCNumber$
 $\quad [',' LCNumber$
 $\quad \quad IDENTIFIER LCNumber$
 $\quad \quad]^*$
 $\quad \}'$
 $s_paramdeclaration \rightarrow N_ParamDeclaration \{'$
 $\quad N_ConsDecl \{'$
 $\quad [CONSTANT \mid CONSTANTS] LCNumber$
 $\quad \}'$
 $\quad s_identdecl$
 $\quad [',' LCNumber$
 $\quad \quad s_identdecl]^*$
 $\quad \}'$
 $s_identdecl \rightarrow N_IdentDecl \{'$
 $\quad [IDENTIFIER ['(' '_' [',' '_']^* ')'] ?$
 $\quad \mid s_prefixop _'$
 $\quad \mid _ '[s_infixop _ ' \mid s_postfixop]$
 $\quad \quad] LCNumber \}'$
 $s_operatordefinition \rightarrow N_OperatorDefinition \{'$
 $\quad [s_identlhs$
 $\quad \mid s_prefixop LCNumber IDENTIFIER LCNumber$
 $\quad \mid IDENTIFIER LCNumber [$
 $\quad \quad s_infixop LCNumber IDENTIFIER LCNumber$
 $\quad \quad \mid s_postfixop LCNumber]$
 $\quad]$
 $\quad '=' LCNumber$
 $\quad s_expression$
 $\quad \}'$
 $s_identlhs \rightarrow N_IdentLHS \{'$
 $\quad IDENTIFIER LCNumber$
 $\quad ['(' LCNumber$
 $\quad \quad s_identdecl [',' LCNumber$
 $\quad \quad s_identdecl]^*$
 $\quad \quad ')' LCNumber$
 $\quad] ?$
 $\quad \}'$
 $s_functiondefinition \rightarrow N_FunctionDefinition \{'$
 $\quad IDENTIFIER LCNumber$
 $\quad '[' LCNumber$
 $\quad s_quantbound$
 $\quad [',' LCNumber s_quantbound]^*$
 $\quad \quad ']' LCNumber$
 $\quad \quad '=' LCNumber$

```

s_expression
  '}'
s_quantbound → N_QuantBound '{'
               [ s_idelltuple
               | [ IDENTIFIER LCNumber
                 [ ',' LCNumber IDENTIFIER LCNumber]* ]
               ]
               Belong LCNumber
               s_expression
               '}'
s_idelltuple → N_Tuple '{'
               '<'<' LCNumber
               IDENTIFIER LCNumber
               [ ',' LCNumber IDENTIFIER LCNumber ]*
               '>'>' LCNumber
               '}'
s_instance → N_Instance '{'
             INSTANCE LCNumber
             s_instantiation
             '}'
s_instantiation → N_Instantiation '{'
                 IDENTIFIER LCNumber
                 [ WITH LCNumber
                   s_substitution [ ',' LCNumber s_substitution]*
                 ]?
                 '}'
s_substitution → [ IDENTIFIER
                  | %prefer
                    s_infixop
                  | s_prefixop
                  | s_postfixop] LCNumber
                  INST LCNumber
                  s_argument
s_argument → s_expression | s_genprefixop | s_geninfixop | s_genpostfixop
s_moduledefinition → N_ModuleDefinition '{'
                    s_idelltls
                    '=' LCNumber
                    s_instance
                    '}'

s_assumption → N_Assumption '{'

               [ ASSUME | ASSUMPTION | AXIOM ] LCNumber
               s_expression
               '}'
s_theorem → N_Theorem '{'
            THEOREM LCNumber
            s_expression
            '}'

```

```

s_expression → s_generalid | s_opapplication | s_prefixexpr | s_infixexpr |
               s_postfixexpr | s_parenexpr | s_boundedquant | s_unboundedquant |
               s_unboundedorboundedchoose | s_setenumerate | s_subsetof |
               s_setofall | s_fcnappl | s_fcncnst | s_setoffcns | s_rcdconstructor |
               s_setofrcds | s_except | s_tuple | s_actionexpr | s_ifthenelse | s_case |
               s_letin | s_conjlist | s_disjlist | s_recordcomponent | s_number |
               IDENTIFIER LCNumber
s_generalid → N_GeneralId '{'
              s_idprefix
              [ IDENTIFIER | ET ] LCNumber
              '}'
s_opapplication → N_OpApplication '{'
                  s_generalid
                  s_opargs
                  '}'
s_prefixexpr → N_PrefixExpr '{'
               s_genprefixop
               s_expression
               '}'
s_infixexpr → N_InfixExpr '{'
              s_expression
              s_geninfixop
              s_expression
              '}'
s_postfixexpr → N_PostfixExpr '{'
                s_expression
                s_genpostfixop
                '}'
s_parenexpr → N_ParenExpr '{'
              '(' LCNumber
              s_expression
              ')' LCNumber
              '}'
s_boundedquant → N_BoundedQuant '{'
                 [ FORALL | EXIST ] LCNumber
                 s_quantbound
                 [ ',' LCNumber s_quantbound ]*
                 ':' LCNumber
                 s_expression
                 '}'
s_unboundedquant → N_UnboundedQuant '{'
                   [ FORALL | EXIST | FFORALL | EEXIST ] LCNumber
                   IDENTIFIER LCNumber
                   [ ',' LCNumber IDENTIFIER LCNumber ]*
                   ':'
                   s_expression
                   '}'
s_unboundedorboundedchoose → N_UnBoundedOrBoundedChoose '{'
                              CHOOSE LCNumber

```

```

[ IDENTIFIER LCNumber | s_idelltuple]
[ N_MaybeBound '{'
  Belong LCNumber
  s_expression
  '}' ]?
':' LCNumber
s_expression
'}'

s_setenumerate → N_SetEnumerate '{'
  '{' LCNumber
  [ s_expression [ ',' LCNumber s_expression ]* ]?
  '}' LCNumber
  '}'

s_subsetof → N_SubsetOf '{'
  '{' LCNumber
  [ IDENTIFIER LCNumber | s_tuple]
  Belong LCNumber
  s_expression
  ':' LCNumber
  s_expression
  '}' LCNumber
  '}'

s_setofall → N_SetOfAll '{'
  '{' LCNumber
  s_expression
  ':' LCNumber
  s_quantbound [ ',' LCNumber s_quantbound ]*
  '}' LCNumber
  '}'

s_fcnappl → N_FcnAppl '{'
  s_expression
  '[' LCNumber
  s_expression [ ',' LCNumber s_expression ]*
  ']' LCNumber
  '}'

s_fcnconst → N_FcnConst '{'
  '[' LCNumber
  s_quantbound
  [ ',' LCNumber s_quantbound ]*
  '[' '-'> LCNumber
  s_expression
  ']' LCNumber
  '}'

s_setoffcns → N_SetOfFcns '{'
  '[' LCNumber
  s_expression
  '-'> LCNumber
  s_expression
  ']' LCNumber

```

```

    '}'
s_rcdconstructor → N_RcdConstructor '{'
                  '[' LCNumber
                  s_fieldval
                  [ ',' LCNumber s_fieldval ]*
                  ']' LCNumber
                  '}'
s_fieldval → N_FieldVal '{'
            IDENTIFIER LCNumber
            FIELDOP LCNumber
            s_expression
            '}'
s_setofrcds → N_SetOfRcds '{'
             '[' LCNumber
             s_fieldset
             [ ',' LCNumber s_fieldset ]*
             ']' LCNumber
             '}'
s_fieldset → N_FieldSet '{'
            IDENTIFIER LCNumber
            ':' LCNumber
            s_expression
            '}'
s_except → N_Except '{'
          '[' LCNumber
          s_expression
          EXCEPT LCNumber
          s_exceptspec
          ']' LCNumber
          '}'
s_exceptspec → N_ExceptSpec '{'
              '!' LCNumber
              [ s_exceptcomponent | s_otherexceptcomponent ]
              '=' LCNumber
              s_expression
              '}'
              [ ',' LCNumber
              N_ExceptSpec '{'
              '!' LCNumber
              [ s_exceptcomponent | s_otherexceptcomponent ]
              '=' LCNumber
              s_expression
              '}'
              ]*
s_otherexceptcomponent → '[' LCNumber
                       s_expression
                       [ ',' LCNumber s_expression ]*
                       ']' LCNumber
s_exceptcomponent → N_ExceptComponent '{'

```

```

                                '.' LCNumber
                                IDENTIFIER LCNumber
                                '}'
s_tuple → N_Tuple '{'
          '<"<' LCNumber
          s_expression
          [ ';' LCNumber s_expression]*
          '>">' LCNumber
          '}'
s_actionexpr → N_ActionExpr '{'
              [ '[' LCNumber
                s_expression
                Actionunderline LCNumber
                | '<"<' LCNumber
                  s_expression
                  '>">"_' LCNumber
                ]
              s_expression
              '}'
s_ifthenelse → N_IfThenElse '{'
              IF LCNumber
              s_expression
              THEN LCNumber
              s_expression
              ELSE LCNumber
              s_expression
              '}'
s_case → N_Case '{'
        CASE LCNumber
        s_casearm
        [ ALWAYS LCNumber
          [ s_casearm | s_otherarm ]
        ]*
        '}'
s_casearm → N_CaseArm '{'
           s_expression
           CASEOP LCNumber
           s_expression
           '}'
s_otherarm → N_OtherArm '{'
            OTHER LCNumber
            CASEOP LCNumber
            s_expression
s_letin → N_LetIn '{'
         LET LCNumber
         N_LetDefinitions '{'
         s_letdefinitions
         '}'
         IN LCNumber

```

```

s_expression
'}'
s_letdefinitions → [
    [s_operatordefinition | s_functiondefinition | s_moduledefinition]
]+
s_conjlist → N_ConjList '{'
    [ N_ConjItem '{'
        CONJ LCNumber
        s_expression
        '}'
    ]+
    '}'
s_disjlist → N_DisjList '{'
    [ N_DisjItem '{'
        DISJ LCNumber
        s_expression
        '}'
    ]+
    '}'
s_recordcomponent → N_RecordComponent '{'
    s_expression
    '.' LCNumber
    IDENTIFIER LCNumber
    '}'
s_number → N_Number '{'
    NUMBER LCNumber
    '}'
s_genprefixop → N_GenPrefixOp '{'
    s_idprefix
    s_prefixop
    LCNumber
    '}'
s_geninfixop → N_GenInfixOp '{'
    s_idprefix
    s_infixop
    LCNumber
    '}'
s_genpostfixop → N_GenPostfixOp '{'
    s_idprefix
    s_postfixop
    LCNumber
    '}'
s_idprefix → N_IdPrefix '{'
    [ s_idprefixelement ]*
    '}'
s_idprefixelement → N_IdPrefixElement '{'
    IDENTIFIER LCNumber
    [ s_opargs ]?
    '!' LCNumber

```



```

s_opargs → N_OpArgs '{'
          '(' LCNumber
            s_expression
          [ ',' LCNumber
            s_expression ]*
          ')' LCNumber
        '}'

s_prefixop → ['- | '~ | ALWAYS | NALWAYS | LNOT | NEG | DOMAIN |
              ENABLED | SUBSET | UNCHANGED | UNION]

s_infixop → ['# | '$' | '%' | '&' | '*' | '+' | '-' | '.' | '/' | '<' | '=' | '>' | '?' | '\\' | '^' | '|' |
              DCLAIM | DWELL | DDOLLAR | DDIV | DAND | BRPLUS |
              BRSUB | BRDOT | BRDIV | BRTIMES | DSTAR | DPLUS |
              TEMPORAL | DSUB | OVEROR | TWODOT | THREEDOT |
              DIFFERENT | NOTEQUAL | CONJ | BNFDEFINE | ASSIGN |
              FCNCONSTRUCTOR | OPFC | EQUAL | SMALLERTHAN |
              IMPLY | OPBA | LARGER THAN | DQUE | DET | DISJ | DEX |
              OROVER | BOOLASSIGN | DOR | LEADSTO | Approx | Geq |
              Oslash | Sqsupseteq | Asymp | Gg | Otimes | Star | Bigcirc | Belong |
              Prec | Subset | Bullet | Intersect | Preceq | Subseteq | Cap | Land |
              Propto | Succ | Cdot | Leq | Sim | Succeq | Circ | Ll | Simeq | Supset |
              Cong | Lor | Sqcap | Supseteq | Cup | O | Sqcup | Union | Div | Odot |
              Sqsubset | Uplus | Doteq | Ominus | Sqsubseteq | Wr | Equiv | Oplus |
              Sqsupset ]

s_postfixop → [ '\" | EXPLUS | EXSTAR | EXWELL ]

```

Appendix D

Testing Examples

The following TLA+ formulas are some examples which are used for systematic testing of the GTLA translator. For each prototype various TLA+ formulas which represent various situations were used for testing. The results showed a successful translation from TLA+ to SML for each prototype. The examples are included in the testing files under CD/Testing Examples.

➤ Logic operators

TestA == $\neg (\text{TRUE} \wedge (\text{FALSE} \vee \text{TRUE})) \Leftrightarrow \text{FALSE}$

TestB == $(\text{TRUE} \wedge \text{FALSE}) \Rightarrow \text{TRUE}$

TestC == $\forall x \in \{1,2,3\}: x > 2$

TestD == $\exists x \in \{1,2,3\}: x > 2$

TestE == $\text{CHOOSE } x \in \{1,2,3\}: x < 3$

TestF == $\exists x \in \text{BOOLEAN}: x \Leftrightarrow \text{TRUE}$

➤ Set operators

TestA == $\{1,2,3\} = \{1,2,3\}$

TestB == $\{ "a", "b" \} \# \{ "c" \}$

TestC == $1 \in \{1,2,3\}$

TestD == $1 \notin \{2,3,4\}$

TestE == $\{1,5\} \cup \{2,3,4\}$

TestF == $\{x \in \{1,2,3\}: x > 2\}$

TestG == $\{x+5: x \in \{1,2,3\}\}$

➤ Function operators

$f[x \in \{1,2,3\}] = x+3$

TestA == $f[3]$

TestB == $[x \in \{1,2,3\}] \rightarrow 1$

TestC == [f EXCEPT ![3]=4]

➤ **Record operators**

TestA == chan \in [Val:{ 1,2,3}, rdy:{0,1}, ack:{0,1}]

TestB == r = [Val |-> "d1", rdy |-> 1, ack |-> 0]

TestC == chan.Val

TestD == [r EXCEPT !.rdy = 0]

➤ **Tuple, Strings and Numbers**

Tuple == <<1,"rdy",TRUE>>

Tupleelem == Tuple[1]

TestNum == 1.34 + 20.9 - 0.56

TestStr(x) == IF x = "bo" THEN TRUE ELSE FALSE

➤ **Miscellaneous constructs**

✓ **if...then...else...**

TestA(x) == IF x>=0 THEN 1 ELSE 0

TestB(x) == IF x<=0 THEN TRUE ELSE FALSE

TestC(x) == IF x=0 THEN "equal" ELSE "notequal"

TestD(x) == IF x>0 THEN x + 100 ELSE 0

TestE(x) == IF x<0 THEN 5 ELSE 0

✓ **case...other...**

TestA(x) == CASE x = 0 -> 0 [] x = 1 -> 1 [] x = 2 -> 2

TestB(x) == CASE x = 0 -> x [] x = 1 -> x+1 [] OTHER -> x+2

✓ **let...in...**

TestA == LET p == 6 IN p+1

TestB == LET x == 1

f[t \in Data] == t + 6

IN x + f[1]

Appendix E

Testing Examples 2

The following concrete examples are used for the translator, simulator and integration testing of the GTLA system and got successful results. These examples are included in the testing files under CD/Testing Examples.

➤ Example 1 – HourClock

- **TLA+ specification file**

```
----- MODULE HourClock -----  
  
EXTENDS Naturals  
  
VARIABLE hr  
  
HCini == hr \in (1 .. 12)  
  
HCnxt == hr' = IF hr # 12 THEN hr + 1 ELSE 1  
  
HC == HCini /\ [][HCnxt]_hr  
  
-----  
  
THEOREM HC => []HCini  
  
=====
```

- **Configuration file**

```
SPECIFICATION HC
```

➤ Example 2 – AsyncInterface

- **TLA+ specification file**

```
----- MODULE AsyncInterface -----  
  
EXTENDS Naturals  
  
CONSTANT Data
```

VARIABLES val, rdy, ack

TypeInvariant == \wedge val \in Data

\wedge rdy \in {0, 1}

\wedge ack \in {0, 1}

Init == \wedge val \in Data

\wedge rdy \in {0, 1}

\wedge ack = rdy

Send == \wedge rdy = ack

\wedge val' \in Data

\wedge rdy' = 1 - rdy

\wedge UNCHANGED ack

Rcv == \wedge rdy \neq ack

\wedge ack' = 1 - ack

\wedge UNCHANGED $\langle\langle$ val, rdy $\rangle\rangle$

Next == Send \vee Rcv \vee Init

Spec == Init \wedge \square [Next]₋ $\langle\langle$ val, rdy, ack $\rangle\rangle$

THEOREM Spec \Rightarrow \square TypeInvariant
=====

● **Configuration file**

SPECIFICATION Spec

CONSTANT Data = {1, 2, 3}

➤ **Example 3 – Channel**

● **TLA+ specification file**

----- MODULE Channel -----

EXTENDS Naturals

CONSTANT Data

VARIABLE chan, d

TypeInvariant == chan \in [val : Data, rdy : {0, 1}, ack : {0, 1}]

Init == \wedge chan \in [val : Data, rdy : {0, 1}, ack : {0, 1}]

\wedge chan.ack = chan.rdy

\wedge d \in Data

Send == \wedge chan.rdy = chan.ack

\wedge chan' = [chan EXCEPT !.val = d, !.rdy = 1 - @]

\wedge d' \in Data

Rcv == \wedge chan.rdy # chan.ack

\wedge chan' = [chan EXCEPT !.ack = 1 - @]

Next == Send \vee Rcv

Spec == Init \wedge [][Next]_chan

THEOREM Spec => []TypeInvariant

● Configuration file

SPECIFICATION Spec

CONSTANT Data = {1, 2, 3}

➤ Example 4 – Counter

● TLA+ specification file

----- MODULE Counter -----

EXTENDS Naturals

VARIABLE d

```

Gini == d \in (1 .. 10)

Ginc1 ==  \ d <= 9
          \ d' = d + 1

Ginc2 ==  \ d <= 8
          \ d' = d + 2

Ginc3 ==  \ d <= 7
          \ d' = d + 3

Gdec1 ==  \ d >= 1
          \ d' = d - 1

Gdec2 ==  \ d >= 2
          \ d' = d - 2

Gdec3 ==  \ d >= 3
          \ d' = d - 3

Gnxt == Ginc1 \vee Ginc2 \vee Ginc3 \vee Gdec1 \vee Gdec2 \vee Gdec3

G == Gini \wedge [][Gnxt]_d

```

```

-----

THEOREM  G => []Gini

```

● Configuration file

SPECIFICATION G