h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Hochschule Darmstadt

- Fachbereich Informatik –

Fast Prototyping of Model Checking Exploration Algorithms

Abschlussarbeit zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

vorgelegt von

Nouri Alnahawi

Referent : Prof. Dr. Bernhard Humm

Korreferent : Prof. Dr. Klaus Frank

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Darmstadt, den

Abstract

In the field of concurrent distributed systems, model checking has been used to verify the correctness of these systems. However, due to state explosion, the exponential growth of a system's state space, it is rather difficult and challenging to apply this method and acquire reliable results regarding a given system. Partial order reduction (POR) simplifies the problem by restricting the verification to a reduced state space while maintaining the soundness of properties of the system being tested.

Research scientists have been pursuing this issue during the last three decades, developing model checking exploration algorithms based on partial order reduction, in order to alleviate the problem as much as possible. These algorithms however, vary in their logic and implementation, which makes it difficult to decide, which algorithms are more efficient without thoroughly testing them. This procedure costs a lot of time and effort, which is not desired while prototyping these algorithms.

In this thesis, we present a specially designed tool that allows developers and researchers to further pursue this issue, yet save time, effort and costs of developing. The tool offers a minimal standard interface for implementing algorithms, and applying them onto simplified programs, that simulate the behavior of concurrent systems.

As a foundation to our tool, we create a multi-threaded-like environment using transition systems, which allows testing the features of the algorithms under development without having to apply them on real existing and functioning concurrent systems.

Table of Contents

Chapter 1

Introduction

The first chapter provides an overview in respect to the background of this thesis, as well as some of the related work in that field. It also lists some of the reasons and motives for the work at hand, as further research in the same area.

1.1. Background & Motivation

Over the last three decades, the usage of computerized and computer based systems has been rapidly increasing all around the globe. Day by day we become more dependent on instruments and functionalities related to digital and computational breakthroughs, which unarguably has made our lives much easier and more efficient in many aspects. Control functions in modern cars, cruise controls, mobile phones, medical devices etc. are almost solely able to function due to technological inventions that are also computer based.

In most cases, these devices utilize artificial intelligence computer based systems that are more often than not connected to each other. It has become nearly impossible for a system to function without being connected to another. Furthermore, a lot of these systems work concurrently together, such as multi-threaded servers, and embedded car control units that perform quasi-parallel tasks (multitasking). These systems are steadily growing in complexity, and are being distributed over several processing units and networks, making them a vital and non-expendable part of many safety-critical systems.

In the light of these clear facts, many aspects regarding the correctness, safety and security of critical information technology systems pose the questions: How do we verify the trustworthiness and reliability of these systems? How can we assure that these constantly growing systems are still able to perform their duties without jeopardizing the security of the systems? Thus a main challenge for the field of computer science is to provide formalisms, techniques, and tools that will enable the efficient design of correct and well-functioning systems despite their complexity [MIT16].

Model checking, also known as property checking, is a technique for verifying finite state concurrent systems. It has a number of advantages over traditional approaches that are based on simulation, testing, and deductive reasoning. In particular, model checking is automatic and usually quite fast. Also, if the design contains an error; model checking will produce a counterexample that can be used to pinpoint the source of the error. The method developed by Edmund M. Clarke, Randal Bryant, E. Allen Emerson, and Kenneth McMillan, which was awarded the 1998 ACM Paris Kanellakis Award for Theory and Practice, has been used successfully in practice to verify real industrial designs [MIT16].

The main challenge in model checking is dealing with the state space explosion problem. This occurs in systems with many components that can interact with each other or systems with data structures that can assume many different values. In such cases the number of global states can be enormous. Researchers have made considerable progress on this problem over the last ten years [MIT16]

Using special algorithms based on partial order reduction, researchers have been able to drastically reduce the size of the state space in a given system, so that the state explosion problem is hopefully no longer an obstacle standing in the way of complete and sound verification of concurrent distributed systems.

Nonetheless, these recently developed algorithms, some of which are still being analyzed, tested and enhanced, may deliver different results regarding efficiency depending on their logic and implementation. Even though they are based on partial order reduction, there is still plenty of room for further research and improvement.

Throughout this thesis, we present a tool that offers a standard interface for implementing and testing these algorithms under development, as well as already existing ones. Making it possible to quickly see the results that a certain algorithm can deliver, when applied to a simplified user defined concurrent system utilizing the concept of transition systems. This tool aims to help researchers fast prototype their own ideas and compare it to other known works in this field without having to design or implement a standalone tool or program just to test one algorithm. Furthermore, this tool provides an environment for executing common instructions that are to be found in almost all concurrent systems, such as simple arithmetic operations that have access to shared resources. Focusing on changing the values of variables used by several components of the systems, e.g., threads, and monitoring how these changes affect the flow of the program running the system, especially when these values are used in control flow statements such as loops and branching.

The end result of this work will demonstrate that this tool is able to read example programs, accordingly build representative transition systems, and apply a model checking exploration algorithm that delivers statistics regarding the state space of the given example program.

1.2. Related Work

One of the most known names in the field of model checking is Edmund M. Clarke, who, with his Ph.D. student E. Allen Emerson, first introduced the idea of using model checking as a verification technique for finite state concurrent systems in the early eighties. His book *Model Checking* [CGP99] is considered to be the first comprehensive presentation of theory and practice of model checking [MIT16]. His opinion on the state-explosion problem being the most important problem in Model Checking, which he stated in his article *The Birth of Model Checking* [Cla08], was also featured in the book *25 Years of Model Checking* [GV08] among other articles from several pioneers in this field, e.g., E. Allen Emerson's *The Beginning of Model Checking: A Personal Perspective*, in which he emphasizes the importance of model checking and verification [Eme08].

Partial order reduction was introduced in the early nineties by Doron A. Peled, who suggested to reduce the state graph used for model checking by applying existing model checking algorithms to the reduced state graph rather than on the larger full state graph of the program [Pel93]. Patrice Godefroid also contributed to the partial-order methods with a new approach to the state-space explosion problem [God94]. The foundation for the design and architecture of our tool is however based on dynamic partial order reduction introduced by Cormac Flanagan and Patrice Godefroid. They presented a new approach based on initially exploring an arbitrary interleaving of the various concurrent processes/threads, and dynamically tracking interactions between these to identify backtracking points where alternative paths in the state space need to be explored. Furthermore, using examples of multi-threaded programs where the new dynamic partial-order reduction technique significantly reduces the search space, even though traditional partial-order algorithms are helpless [FG05].

A survey of model checking tools was published by the University of Virginia, comparing some of the most common tools to date [SA06]. Another relevant work was published by the Université de Sherbrooke in Canada, offering an overview for six different tools in addition to a thorough analysis of each one of them [FF+10].

It is to be taken into consideration that the related works provide the basis for the theoretical understanding of the concepts of model checking, (dynamic) partial order reduction and model checking tools. However, the purpose of the tool presented in this thesis is solely to provide a platform for implementing POR-algorithms and testing their efficiency, not verifying concurrent systems using the model checking and POR methods.

1.3. Outline

This thesis is divided into seven chapters, the first of which being the Introduction. In the second chapter we will focus on the preliminaries and some of the basic concepts necessary for understanding the theory behind model checking and partial order reduction. We will also shed some light on transition systems, and how they are an important corner stone in state exploration, leading us eventually to the idea of fast prototyping.

After getting an elaborate idea of the theory, we move on to Chapter three, where we deal with the requirement analysis of the assignment, i.e., the tool we developed, focusing on the demands and requisitions, and how they were systematically categorized and prioritized for the development.

Chapter four handles the basic concept and the design of the tool. This includes the decisions made regarding the technologies used in the tool such as the programming language and the run time environment. It also describes the chosen architecture and components of the tool.

Having laid down the foundations for the design, we move on to Chapter five, where we take a closer look at the actual implementation and the most important code sections of the system components.

Finally we analyze the results we have achieved and evaluate the tool in Chapter six, moving on to the final summary in Chapter seven.

Chapter 2

Preliminaries

In this chapter, we introduce the basic definitions and concepts of software verification, model checking and state exploring using partial order reduction. Since the algorithms that can be implemented in the tool take as input a transition system, we will also go through the basics of transition systems covering up the most important theory concepts necessary for comprehending, how the above mentioned concepts work together to build up the components of our tool.

2.1 Software Verification

In the field of software engineering, verification and validation (V&V) is the process of checking that a software system meets certain specifications and that it fulfills its intended purpose.

Barry Boehm succinctly expresses the goal of system verification [Boe89], being the answer to the question: Are we building the product right? In other words, software verification is ensuring that the product has been built according to the requirements and design specifications.

There are a lot of different methods for system verification. Some of them are more reliable than others, and are also used in the industry. Model checking has proved to be one of the best general verification techniques that explores all possible system states in a brute-force manner.

2.2 Model Checking

Both the idea and the term model checking were introduced by Clarke and Emerson in 1981 [CE81], proposing a method to establish that a given program meets a given specification where:

- The program defines a finite state graph[1] *M*.
- *M* is searched for elaborate patterns to determine if the specification *f* holds.
- Pattern specification is flexible.
- The method is efficient in the size of *M*, and ideally, *f*.
- The method is algorithmic and practical.

Over the last 30 years, Clarke, Emerson and others in the field managed to get model checking into a concrete state, where model checkers are able to verify protocols with millions of states and hardware circuits with 10^{50} or more states [Eme08].

The work of Baier and Katoen in their book *Principles of Model Checking* [BK08] provides a clear and simple definition of the concept of model checking.

Definition 1 (Model Checking). *Model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model.*

[1] P.11 Definition 3 Finite State Machine

The described model checking process consists of three phases according to Baier and Katoen [BK08]:

- Modeling phase: Modelling the system using a modelling description language and formalizing the property to be checked.
- Running phase: Running the model checker for verification of the system property.
- Analysis phase: Expecting three possible outcomes. The specified property is either valid in the given model or not, or the model turns out to be too large to fit within the physical limits of the computer memory.

In order for us to be able to utilize the concept of model checking, we need a standardized way of modelling a given system. This leads us to a state graph like class of models to represent hardware and software systems, which is known as *Transition System.*

2.3 Transition Systems

The formal definition of a transition system introduced by Baier and Katoen [BK08], although worth mentioning, differs from the one used in this thesis. Our notion of transition systems is borrowed form Saissi et al. [SBM+13]. The main characteristics of this notion are that transitions are deterministic and that there exists a unique initial state for every transition system. As the used algorithms are stateless, we additionally require transition systems to be acyclic [Met14].

Definition 2 (Transition System). *A transition system is a triple TS = (S, s0, T) where S is a finite set of states, s0 ϵ S is the initial state of the system, and T is a finite set of transitions such that for all t ϵ T, t: S ⇀ S, i.e., transitions are partial functions from S to S, and for all s1,..., sn+1 ϵ S and any finite sequence t1…tn ϵ T such that ti(si) = si+1, s1 ≠ sn+1.*

For a transition system *(S, s0, T), t ϵ S, and s, s´ ϵ S,* whenever *t(s) = s´* is defined, *s´* is called the resulting state of *t* at *s* and we write $s \xrightarrow{t1} s´$. A transition *t ϵ T* is *enabled* at a state *s ϵ S* if there exists a resulting state of *t* at *s.* A state for which no transitions are enabled is called a *deadlock.* We write *enabled(s)* for the set of enabled transitions of *s*, i.e. *enabled(s):=* {*t ϵ T:* ∃*s´ ϵ S.s´= t(s)*} [Met14].

In order to be able to apply the designed POR-algorithms onto a transition system, and to optimize the exploration of the state graph, we require that transitions cannot disable other transitions and call transition systems that satisfy this requirement *separated.* Each statement of the program is modelled by a single transition. A transition *t* which corresponds to a statement *stmt* is enabled at a state *s* if and only if the transition corresponding to the preceding statement of *stmt* has *s* as a resulting state, or if *t* models an initial statement of the program and *s* is the initial state of the transition system [Met14].

In chapters four and five, we will take a closer look at the details of modelling a transition system, and how the transitions correspond to the statements of a given program in order to use the transition system for applying an algorithm. It is also to mention, that we are using explicit state model checking, as opposed to symbolic model checking [EP02].

2.4 State Exploration and State Space Explosion

In order to understand, what state exploration means, we need to first understand what a finite state machine (FSM) is, and what it consists of. The formal definition of an FSM by Katrin Erk and Lutz Preise [EP08] offers a very clear and comprehensible explanation.

Definition 3 (Finite State Machine). *A finite state machine, or a finite sequential machine (FSM) is a five-tuple $A = (K, \Sigma, \delta, s0, F)$ where:*

- *K is a finite set of states.*
- *Σ is a finite alphabet.*
- *$\delta : K \times \Sigma \longrightarrow K$ are the transition functions.*
- *$s0 \in K$ is the initial state.*
- *$F \subset K$ is the set of final (accept) states.*

A state describes the status of the machine at a certain point in time, and has information that is defined according to the type of the system.

Considering the similarities between transition systems and state graphs, we can assume that various types of systems can be modelled using transition systems. In this thesis we focus on modelling finite-state systems to use them for the representation as transition systems. However, having a finite number of possible states does not necessarily mean that the number of these states, thus the number of possible states in a transition system that correspond to the program's state graph, is possible to determine using formal methods. For many practical systems, the state space may be extremely large, which is a major limitation for state-space search algorithms such as model checking.

A very important term in the area of state exploration is *traversing*, i.e., given a startup position in a state graph (usually the initial state) we move forward to the next available state while respectively executing the program statements connecting two states. As explained in the last section, transitions correspond to statements, therefore we will be referring only to exploring transition systems from now on, ignoring the small formal differences between state graphs and transition systems. However, bearing in mind that the states in a transition system are dynamically calculated, i.e., a new state results of the execution of an enabled transition at a preceding state. States are not pre-calculated and do not exist at the beginning of the exploration, only the initial state.

Examining the small example (Fig.2.1) of a graphic representation of a transition system, we can easily see how state exploration functions. Starting at *s0* we *apply* (execute) the transition *t1* and therefore get the next state *s1*. When two transitions are *enabled* at a certain state, e.g., *s1* with *t2* and *t3* we logically have to execute both resulting in two new states *s2* and *s3*.

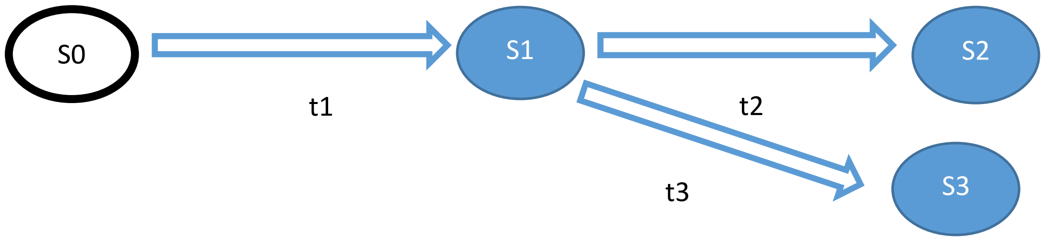


Figure 2.1: Example of a Transition System.

How is state exploration connected to state space explosion? Baier and Katoen [BK08] explained the problem in very comprehensive words using this example:

A program graph with ten locations, three Boolean variables and five bounded integers (with domain in {0... 9}) has $10 \cdot 2^3 \cdot 10^5$=8,000,000 states. If a single bit array of 50 bits is added to this program graph, for example, this bound grows even to $800{,}000 \cdot 2^{50}$! This observation clearly shows why the verification of data-intensive systems (with many variables or complex domains) is extremely hard. Even if there are only a few variables in a program, the state space that must be analyzed may be very large.

Another important aspect discussed by Baier and Katoen [BK08] is *Parallelism.* Considering the previous calculations for a single-threaded system, the state space of a system with n threads where n >= 1, or processes, is the Cartesian product of the state spaces of the single components, i.e., threads, which is more than enough proof that the state explosion problem is not to be taken lightly.

2.5 Partial Order Reduction

In this thesis, we will not attempt to fully understand partial order reduction, nor will we thoroughly discuss the characteristics of this concept. However, we will briefly explain the main idea so that we can have a better understanding of the importance of POR in model checking and system verification.

As we have learned in the former sections, the state-explosion problem is real, and it can be a show stopper when it comes to exploring a transition system, but first, we will take a look at the following example (List.2.1) with two threads *T1* and *T2,* both of which consisting of two program statements with one shared variable *x* and two local variables *y* and *z.*

```
T1{                                     T2{
t1.1 :   x := 2;                        t2.1 :   z := 6;
t1.2 :   y := 3;                        t2.2 :   x := 0;
}                                       }
```

Listing 2.1: Program Order Example with two Threads.

At first glance, it is clear to see that the possible orderings for the execution of the program statements is as follows:

- Variation A: *t1.1→ t1.2→ t2.1→ t2.2*
- Variation B : *t2.1→ t2.2→ t1.1→ t1.2*

- Variation C : *t1.1→ t2.1→ t1.2→ t2.2*
- Variation D : *t1.1→ t2.1→ t2.2→ t1.2*
- Variation E : *t2.1→ t1.1→ t2.2→ t1.2*
- Variation F : *t2.1→ t1.1→ t1.2→ t2.2*

A total of 6 possible *paths* to reach the end of the program, which is easy to calculate using formal mathematical methods. But we notice that both variables *y* and *z* do not affect the outcome of the whole program, since they are local to their respective threads. On the other hand, variable *x* is changed in both threads, which makes it important to find out, under which circumstances the order of execution is changed, and thus changing the final value in the variable *x*.

The aim of partial order reduction in a nutshell, is to reduce the number of possible orderings that need to be analyzed in a transition system, so that the state space is also reduced. Applied on our small example above, we can omit the interleaving paths, where only the order of the statements affecting *y* and *z* is involved, not that of *x*. In doing so, we can reduce the number of possible paths to only two. One path has *x* changed in *T1* first, and vice versa. Put on a much larger scale, e.g., systems with hundreds of threads, each of which working with shared resources that can definitely affect each other, it becomes clear, why we need POR to overcome the state space explosion problem.

2.6 Fast Prototyping

Fast prototyping is a common term used to describe a certain process, by which one can quickly asses and evaluate a product, which is only a prototype in the first phase of the production. Similarly we will refer to our work as *Fast Prototyping*, since we aim to quickly evaluate the quality of the prototypes of the developed POR-Algorithms.

Chapter 3

Requirements Analysis

DEEDS[2], which stands for *Dependable*, *Embedded Systems and Software*, is a research facility in the department of computer science at the Technical University of Darmstadt. They offer regular study program courses held by their professors and Ph.D. seeking students, as well as research studies in several areas of computer science, focusing on design and analysis, and assessment. The development of a fast prototyping tool was proposed by Patrick Metzler and Habib Saissi in order to support their work on partial order reduction in the field of distributed systems.

In this chapter, we will examine the prerequisites and requirements for developing this tool, as a topic for a Bachelor/Master project and thesis[3]. The process of requirement analysis was conducted according to the rules of modern agile software engineering, carried out in weekly iterations parallel to the actual development of the tool, to satisfy the desired end results requested by the research facility. These requirements divide into two categories, functional and non-functional requirements, the latter being of more significance for both Metzler and Saissi, as we will see in the following sections.

The development was documented using the UML-tool MagicDraw 18.2[4], which utilizes the standards of the *Unified Modelling Language* (UML2)[5].

[2] https://www.deeds.informatik.tu-darmstadt.de/deeds/ [Accessed: 28.02.2016]
[3] https://www.deeds.informatik.tu-darmstadt.de/fileadmin/user_upload/GROUP_DEEDS/teaching/msthesis/DEEDS_ThesisProposal_1_habib_.pdf [Accessed: 28.02.2016]
[4] http://www.nomagic.com/products/magicdraw.html [Accessed: 28.02.2016]
[5] http://www.uml.org/ [Accessed: 28.02.2016]

3.1 Fast Prototyping Tool

The main idea was to develop a framework for implementing, testing, and comparing different POR-algorithms. The first step was establishing a system model (design) focusing on three main components that build up the tool:

- Input: Example programs in a pre-defined input language that are represented as transition systems inside the tool (Chapter 2.3, Definition of *Transition System).*
- Functionality: Implementing state space exploration algorithms, and applying them onto the transition systems (Chapter 2.4, *State Space Exploration*).
- Output: Statistics regarding the execution of the algorithms.

On the one hand, it is important that the tool accepts a diversity of POR-algorithms using a standardized way of implementation; on the other hand, the tool should also be capable of applying them to example programs with different size, complexity, and structure. The result of the tool is an executable program that delivers statistics regarding the state space exploration of the transition systems.

As previously mentioned, the meetings with Metzler and Saissi were held on weekly basis. Due to this weekly iteration concept of developing the tool, the goals set after each iteration were updated to suit the current state of development, and to adapt to change requests. This agile method also allowed coping with unexpected and undesired effects, such as delay in the development or deficient understanding of the problem at hand. This way of work was carried out consistently until the end of the project.

The following state machine analysis diagram[6] (Fig.3.1) represents the first system model describing the basic flow, as a result of the first analysis iteration:

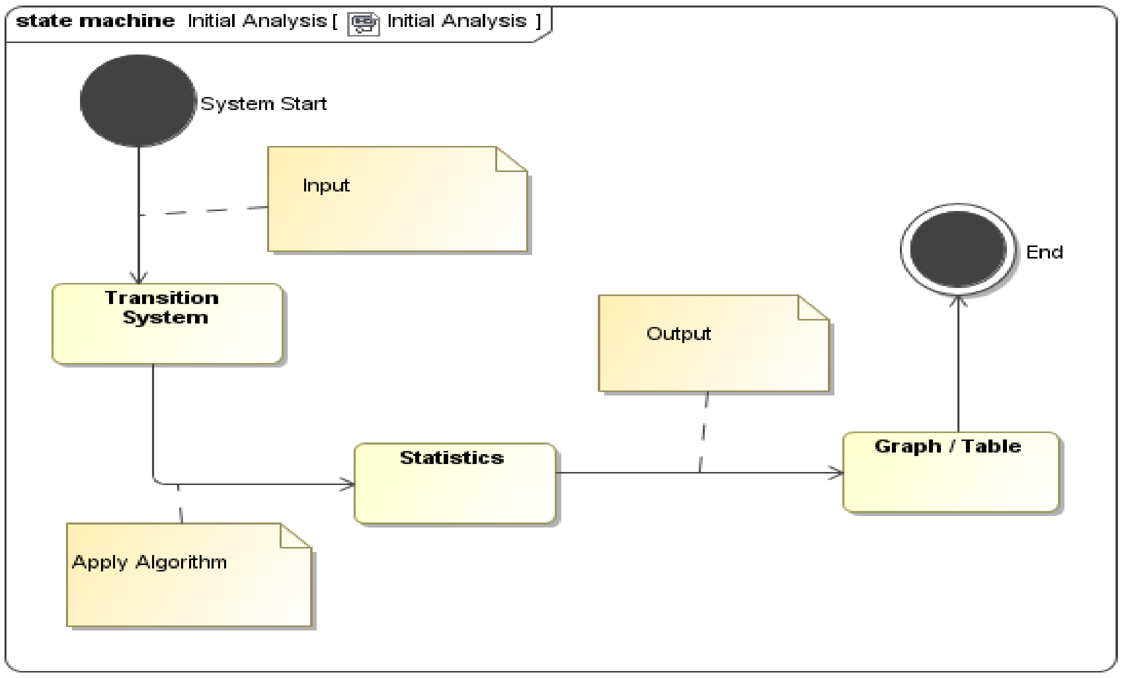


Figure 3.1 Initial Analysis.

It was not necessary to perform a use case analysis applying use case diagrams or story cards etc., since there is only one user related use case, being:

- Start program with specified input file(s) and a specified algorithm.

All other parts in the tool were developed according to the requested specification, which we will discuss in the following sections.

[6] http://www.uml-diagrams.org/state-machine-diagrams.html [Accessed: 29.02.2016]

3.2 Input

The first step in the analysis phase was to determine the type of input to be used for the tool. Metzler and Saissi suggested using standard .txt files to feed the tool with example programs. These example programs should be suitable for representing programs with multiple processes or threads, performing simple arithmetic operations on variables, and also containing common control flow structures, in order to simulate the deterministic and the multithreaded behavior in a distributed system.

The emphasis for the input was laid on the branching and looping concept. The given program should support both sequential and non-sequential execution of the program statements, which allows for different paths leading to the end of the program, depending on the decisions made along the way. These requirements can be summed up as follows:

- Input as .txt files.
- Each .txt file contains one program with one or more threads.
- Each thread contains program statements that are independent of other threads.
- The program is written in a pre-defined programming language, which is to be created by the developer.
- The programming language should support the following aspects:
    - Multi-threaded program code.
    - Local, and constant identifiers with initial value and length, which can be accessed by all threads.
    - The data type *INTEGER* is used for the value and length of identifiers.
    - Identifiers can be simple *Variables* or indexed *Arrays.*

- A threads consists of a main block containing other statement blocks.
- A statement can be either *Atomic, Conditional*, or a *Loop.*
- Atomic statements are assignments.
- Conditionals and loops use *Boolean* expressions to evaluate the condition.
- Boolean expressions support *Equals, Greater, and Less.*
- Arithmetic operations support *Addition, Subtraction, Multiplication, Division, Modulus, Parentheses* and *Precedence Rules.*

Metzler had defined his own input programming language in his Master thesis on partial order reduction [MET14], which was offered as basis for the new language used in this project.

3.3 Functionality

Naturally, having a defined input language does not suffice for the tool to function. Therefore, it was just as important to decide, what the next step is.

First of all, the tool should be able to read the input (example programs), and transform it into a transition system consisting of program statement. The transition system is then saved in the tool memory to be used later on by the state space exploration algorithm. The algorithms on the other hand should be implemented in the tool, so that they can be applied onto the existing transition system. To manage the transition system, we also had to consider saving variables and their values, and maintaining the order of the program statements in addition to all possible branches using a program counter.

The functional requirements were eventually described as follows:

- Tool can read and parse given example programs (.txt files).
- Tool should recognize wrong input and notify user accordingly.
- Tool can extract program statements from the given examples.
- Tool should differentiate between the types of program statements as specified in the input language.
- Tool should differentiate between the types of identifiers as specified in the input language.
- Transition system should be represented identical to its formal definition.
- Transition system offers methods to be used by algorithms.
- Transition system allows algorithms to dynamically execute program statements using suitable methods. The methods *apply* and *isEnabled* must be implemented in transition systems.
- Program memory is simulated within the tool to keep track of changes in variables and program counters.
- Tool can apply any implemented algorithm onto any correct example program.
- Algorithms calculate states dynamically for a transition system, and calculate statistics based on the number of visited states and reached *Deadlocks.*
- The method *runAlgorithm* must be implemented in algorithms.
- Only the algorithm *Full Search (Exhaustive Search)* is to be implemented by the developer.

3.4 Output

The final main component of the tool was requested to be the output. There were several possibilities for representing the output of the tool; however, our biggest concern was the content of the output, and not its final representation on the screen. The most important aspect in regards to the content of the output was a clear and understandable statistic overview regarding the execution of an algorithm onto the given example programs.

The desired output was decided to be as follows:

- Formatted output as text in console (terminal).
- Statistics regarding algorithm execution containing:
    - Number of applied (executed) transitions.
    - Number of visited (dynamically calculated) states.
    - Number of unique states.
    - Number of all deadlocks.
    - Number of unique deadlocks.

3.5 Non-Functional Requirements

Even though the functional requirements are of great importance, in order for the tool to work, the non-functional requirements were actually the key and most important requisites in the development. The tool is supposed to be used by scholars in the research, to test and optimize newly created POR-algorithms, which means, that the tool can, and will be maintained and expanded to suit the needs of the developers. To achieve this goal, certain criteria must be met.

Some of these demands were non-negotiable and had to be accepted as requested. Other demands were taken into consideration, and we payed extra attention to the basic rules of software engineering and design, so that the tool can be managed and maintained in the future.

The most important non-functional requirements are:

- Tool is written in object oriented C++11 and compiled using GNU GCC[7].
- Tool executable runs on Unix/Linux based systems.
- Git[8] repository for version control.
- Input Language is extendable and adaptable in its definition and description.
- All components are implemented as classes to allow easy future extension and adaptation.
- Use of meaningful naming conventions and standard programming guidelines.
- Tool supports changing and extending the input language in its component design, which requires the appliance of *Separation of Concerns* [Ern04] in the design of the classes.
- Each thread generates a unique transition system with its own program counter.
- Transition system consists of a list of lists of transitions, and an initial state with an initial memory.
- Transitions can independently evaluate their program statements.

[7] https://gcc.gnu.org/ [Accessed: 02.03.2016]
[8] https://git-scm.com/ [Accessed: 02.03.2016]

- Memory is simulated using a C++ indexed array containing all program counters respectively, and all the variables in the given program.
- A standard interface for implementing further algorithms with uniformed methods that can be overridden.

Further details regarding the design concept and the actual implementation will be discussed in the following chapters.

Chapter 4

Concept & Design

With the requirement analysis at hand, it was rather simple to find the main components for the tool, and to decide, what the architecture driver is, which was of great assistance for laying down the design of the software.

According to the concept desired by Metzler and Saissi, it was more important to design the tool with the focus on the maintainability and adaptability, yet the core functionality of the tool had to be simple and comprehensible, so that other researchers can understand how it works, without having to dig deep inside the code. Choosing C++ as a programming language for the tool was of great advantage, since C++ supports almost all the modern object oriented programming features, as well as the old procedural aspects of C; as it was necessary at certain points, to have full control of the behavior of each component of the tool. The run time environment was set to be Unix/Linux based systems, as mentioned in Chapter 3.5.

The result of the first iteration was, as expected, a rough design consisting of three main components that actually correspond to the requests of Metzler and Saissi. These three components are:

- Front-end: Component responsible for reading the example programs and extracting the program statements for further use.
- Middleware: Component responsible for generating transition systems using the extracted program statements, and also applying algorithms onto them.
- Back-end: Calculating statistics acquired from the algorithm execution, formatting the statistics and printing them out on the screen, or writing them into a file.

The following component diagram[9] (Fig.4.1) represents the above mentioned components:

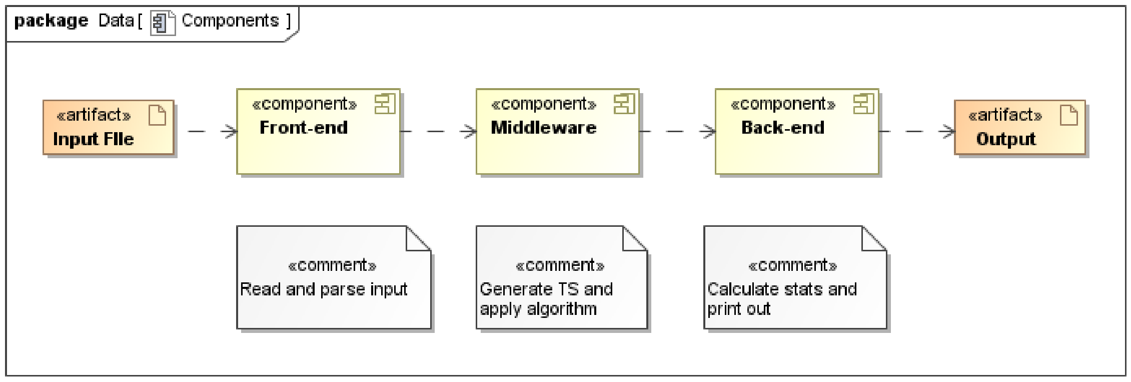

package Data [Components]
«artifact» Input File
«component» Front-end
«component» Middleware
«component» Back-end
«artifact» Output
«comment» Read and parse input
«comment» Generate TS and apply algorithm
«comment» Calculate stats and print out

Figure 4.1 Component Diagram.

4.1 Input Language

Considering the demands on the input language, and the language offered by Metzler from his Master thesis as basis for the new one, we ascertained that the new language must preserve its grammatical characteristics. Most importantly, the grammar had to support recursion and empty rules, in order to allow relatively elaborate and meaningful programming structures.

The theoretical grammar (language description) was defined according to the rules of context free languages[10]. This type of grammar is suitable to our purposes for many reasons, mainly because it allows defining relatively complex structures, and because it's easy to implement using the common parsing concepts.

[9] http://www.sparxsystems.com/resources/uml2_tutorial/uml2_componentdiagram.html [Accessed: 03.03.2016]
[10] https://www.cs.rochester.edu/~nelson/courses/csc_173/grammars/cfg.html [Accessed: 03.03.2016]

A context free grammar consists of four parts:

- Set of terminal symbols (the alphabet of the language).
- Set of non-terminal symbols (place holders).
- Set of productions or rules for replacing the place holders with the actual language.
- A start symbol.

However, it is of utmost importance that the grammar doesn’t contain any *left recursion* or any *ambiguous rules*, so that it could be perfectly utilized by a parsing tool. Fig.A.1 (Appendix A) shows how the grammar of the language is defined. However, this grammar is a representation of the language definition in its final state. Throughout the implementation, the grammar was iteratively extended with more rules and tested accordingly to assure the correctness of its semantics. List.A.3 shows an example of an input file containing two threads (Appendix A).

Logically, the defined input language has to be understood by the tool; In other words, a parsing[11] component is needed. Since there are several parsing tools available for free use, we decided not to implement a parser from scratch, since it would not offer any additional possibilities compared to the already existing tools, which are used in a variety of industrial products.

Bison[12], a free general purpose parser designed by the GNU project, was the best choice for our tool. *Bison* supports C++ implementation utilizing classes and object oriented concepts. It is also perfect for defining context free grammars, since it supports

[11] http://dictionary.reference.com/browse/parsing [Accessed: 03.03.2016]
[12] https://www.gnu.org/software/bison/ [Accessed: 03.03.2016]

LR-parsing[13]. More precisely, *Bison* can be implemented as an LALR(1)[14] bottom-up parser[15] that uses a *shift/reduce stack*[16] to parse the input files (program code). LALR(1) stands for *Look-Ahead Left-to-Right Rightmost Derivation* where 1 is the number of the look-ahead items.

However, *Bison* is not a stand-alone tool. Surely it has the required logic to parse and check the semantic correctness of a given input, but it needs a so called *Tokenizer,* also known as *Scanner,* for the lexical analysis[17]. *Bison* recommends *Flex*[18], a lexical analyzer that basically goes through the given text, sending pre-defined tokens to *Bison,* which in turn uses these tokens during the semantic analysis phase. *Flex* is thus used as a lexical scanner to find and identify correct lexical terms in the code, and pass them over to *Bison.* The biggest advantage of using *Flex* is its compatibility with both *Bison* and C++. Fig.A.2 (Appendix A) displays the tokens defined to be used in *Flex.*

Now that we have the grammar of the input language, the scanner, and the parser of choice, we need to join them. The examples offered on the *Bison* website, recommended using a driver class to encapsulate the scanner, and to control reading and parsing the input. The class diagram (Fig.4.2) shows the representation of the mentioned components as classes.

[13] https://en.wikipedia.org/wiki/LR_parser [Accessed: 04.03.2016]
[14] https://www.gnu.org/software/bison/manual/html_node/Language-and-Grammar.html#Language-and-Grammar [Accessed: 04.03.2016]
[15] http://www.tutorialspoint.com/compiler_design/compiler_design_bottom_up_parser.htm [Accessed: 04.03.2016]
[16] http://sites.tufts.edu/comp181/2013/10/06/shift-reduce-parsing-bottom-up-parsing/ [Accessed: 04.03.2016]
[17] https://en.wikipedia.org/wiki/Lexical_analysis [Accessed: 04.03.2016]
[18] http://flex.sourceforge.net/ [Accessed: 04.03.2016]

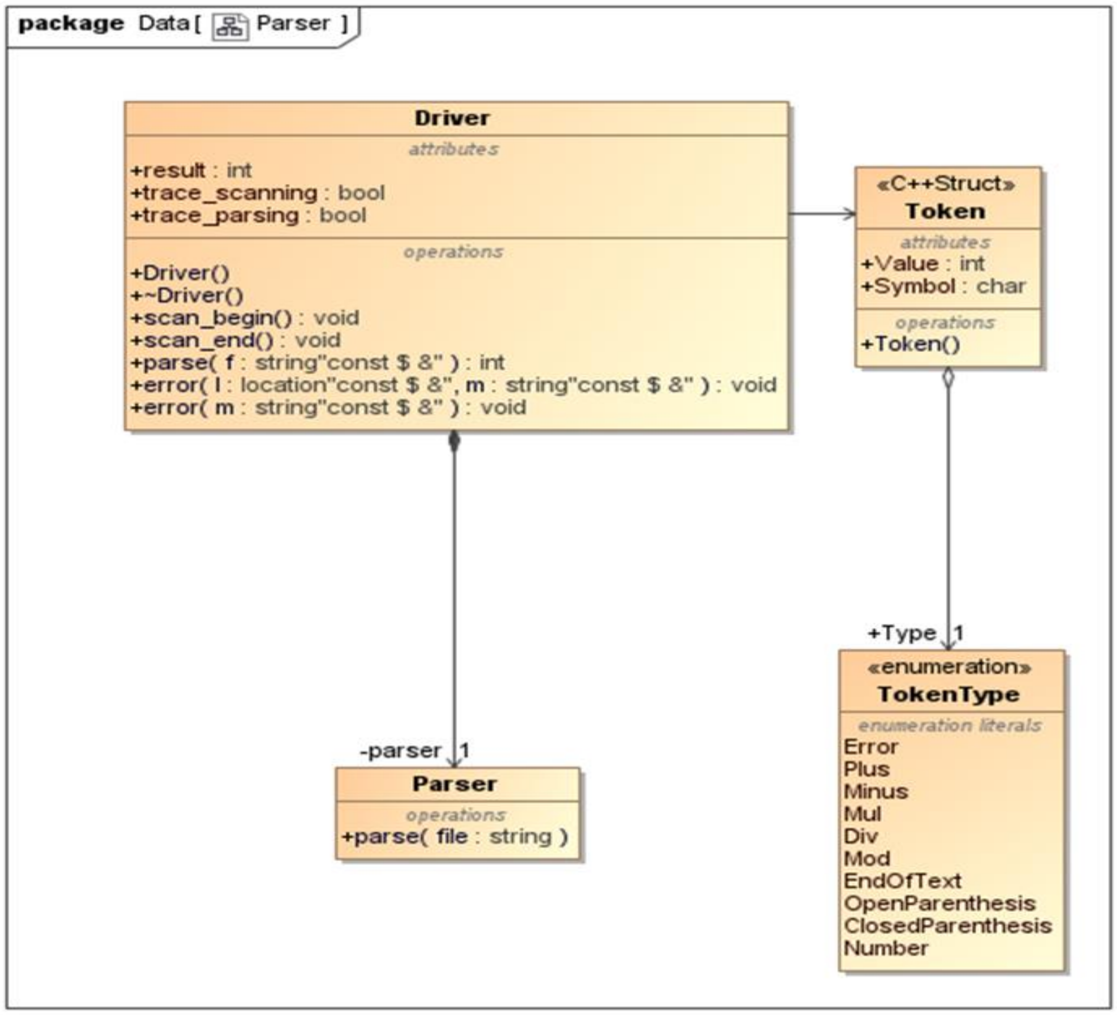


Figure 4.2 Parsing Classes.

Also we had to consider, what to do with the parsed input, and where it goes afterwards, in order to generate transition systems, which is the main purpose of parsing the input. These details will be discussed in chapters 5.1 and 5.2

4.2 Representation of Transition Systems

Since our whole tool uses transition systems as basis for simulating multi-threaded behavior and for representing programs' state graphs, it was necessary to define a suitable representation for transition systems within the tool. The most important aspects in this regard, are the attributes (class members) and the functions (class methods) that a transition system possesses.

Based on the definition of transition systems (Chapter 2.3), and the definition of a finite state machine (Chapter 2.4), we were able to derive several design classes that contain all the needed characteristics and to modulate them according to C++ and to the rules of object oriented design. The class diagram (Fig.4.3) provides an overview of the classes that make up a transition system.

The main class in this group is *TransitionSys,* which contains information regarding the parts that build a transition system. Each instance of this class has a name corresponding to the input program name, and the number of processes (threads) in that program. This class is also responsible for constructing the initial memory and the initial state of the system, therefore, it is also connected to the classes *Memory* and *State,* which are composite to the former. Most importantly, this class is also connected to the class *Transition* via aggregation, as a transition system consists mainly of transitions. Each process in a transition system has its own transitions (refer to input language in Chapter 4.1), which are referenced in a C++ vector of C++ maps in the class *TransitionSys.* Thus, each process owns an index in the vector *transitions,* and at that index exists a map (*int, transition)* containing all the transition objects belonging to that process with their respective program counters. Aside from some utility methods in the class, we have the two most significant public methods of a transition system: *getInitialState* and *getEnabledTransitions,* the role of which will be discussed in Chapter 5.4.

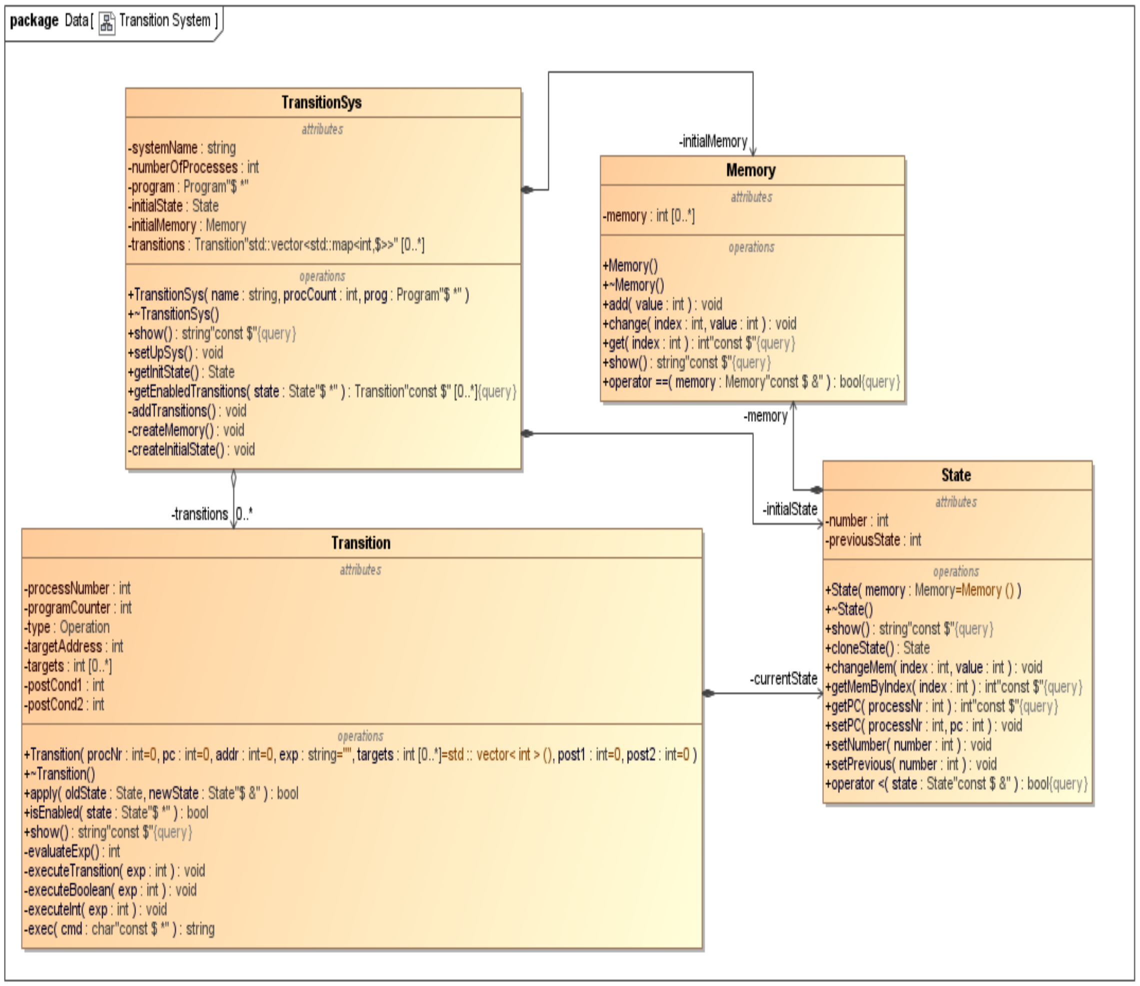

package Data[Transition System]
TransitionSys
attributes
-systemName : string
-numberOfProcesses : int
-program : Program"$ *"
-initialState : State
-initialMemory : Memory
-transitions : Transition"std::vector<std::map<int,$>>" [0..*]
operations
+TransitionSys(name : string, procCount : int, prog : Program"$ *")
+~TransitionSys()
+show() : string"const $"{query}
+setUpSys() : void
+getInitState() : State
+getEnabledTransitions(state : State"$ *") : Transition"const $" [0..*]{query}
-addTransitions() : void
-createMemory() : void
-createInitialState() : void
-initialMemory
Memory
attributes
-memory : int [0..*]
operations
+Memory()
+~Memory()
+add(value : int) : void
+change(index : int, value : int) : void
+get(index : int) : int"const $"{query}
+show() : string"const $"{query}
+operator ==(memory : Memory"const $ &") : bool{query}
-memory
-initialState
State
attributes
-number : int
-previousState : int
operations
+State(memory : Memory=Memory ())
+~State()
+show() : string"const $"{query}
+cloneState() : State
+changeMem(index : int, value : int) : void
+getMemByIndex(index : int) : int"const $"{query}
+getPC(processNr : int) : int"const $"{query}
+setPC(processNr : int, pc : int) : void
+setNumber(number : int) : void
+setPrevious(number : int) : void
+operator <(state : State"const $ &") : bool{query}
-transitions 0..*
-currentState
Transition
attributes
-processNumber : int
-programCounter : int
-type : Operation
-targetAddress : int
-targets : int [0..*]
-postCond1 : int
-postCond2 : int
operations
+Transition(procNr : int=0, pc : int=0, addr : int=0, exp : string="", targets : int [0..*]=std :: vector< int > (), post1 : int=0, post2 : int=0)
+~Transition()
+apply(oldState : State, newState : State"$ &") : bool
+isEnabled(state : State"$ *") : bool
+show() : string"const $"{query}
-evaluateExp() : int
-executeTransition(exp : int) : void
-executeBoolean(exp : int) : void
-executeInt(exp : int) : void
-exec(cmd : char"const $ *") : string

Figure 4.3 Transition System Classes.

As for the class *Transition*, it is of the same importance as the class *TransitionSys*, since it contains, in each instance (object), information regarding a single transition corresponding to the original parsed program statement. These include, in addition to the number of the process, the program counter at the observed transition, and the addresses of the program variables used in the transition. It also disposes of the type of the executable transition (more details in Chapter 5.3). Since a transition is executable, it is the most suitable place for having the logic regarding its own execution. Thus, it offers two public methods to be used by algorithms, namely *isEnabled* and *apply*, the implementation of which will be discussed in chapters 5.3 and 5.4. This class disposes also of several private methods necessary for the execution of a transition, such as *evaluateExp* and *executeTransition.*

The classes *Memory* and *State*, even though most vital for the whole concept, can be viewed as smaller classes holding only the logic needed to perform simple attribute changing actions. However, the class *Memory* provides a place to store and edit the values of the variables and program counters in the system (refer to Chapter 3.5). In addition to the initial memory, each state has an up-to-date snapshot of the whole memory, depending on the changes resulting of the transition execution.

The class *State* was modulated according to the following definition, derived from the definition of a transition system, which we extended to suit our demands of the tool:

A transition system is a five-tuple TS = (S, s0, T, V, P) where S, s0 and T are the same as in the formal definition (Chapter 2.3), V is the set of all variables, and P is the set of all process program counters in the system.

With the variables and the processes included in the definition of a transition system, we can define a state in our system as follows:

The set of states $S\colon V \cup P \longrightarrow \mathbb{N}$ *where* $V \cap P = \phi$ *is the union function of the set of all variables and the set of all process program counters, whose members belongs to the set of natural numbers.*

Additionally we define:

s(v) = value of variable $v \in \mathbb{N}$ *at the state s.*

s(p) = value of program counter $p\ v \in \mathbb{N}$ *at the state s.*

Therefore, an instance of the class *State* disposes of a state number and the number of the preceding state. In addition, a state object offers public methods for changing the values in the memory snapshot that belongs to it. The method *cloneState* was explicitly requested, as it is a part of POR-algorithms, and the functionality of transitions systems transition system.

Having all the above mentioned classes provides us with a logical and realistic representation of a transition system with all the required attributes and functions.

4.3 Algorithm Interface

The final design decision that had to be made, was the representation of algorithms. Bearing in mind that all sorts of POR-algorithm will be implemented using the tool, the design of a standard interface for algorithms was pretty logical and also presupposed by Metzler and Saissi, as mentioned in Chapter 3.5.

The conceptualized design for the interface is rather simple, yet highly effective for our purposes; based on the prerequisites of a POR-algorithm. With a transition system offering public methods to be used by an algorithm throughout its execution (Chapter 4.2), we already have a great deal of what we actually need, since all POR-algorithms require the earlier mentioned methods to explore a transition system, and dynamically calculate the states. The other aspect that all POR-algorithm also have in common, is computing statistics regarding the traversal and the calculated states. The class diagram (Fig.4.4) offers an insight to the layout of the class *Algorithm.*

The algorithm interface class *Algorithm* is a pure virtual class[19], which is the C++ equivalent of an abstract class according to the rules of polymorphism[20] in object oriented

[19] http://en.cppreference.com/w/cpp/language/abstract_class [Accessed: 06.03.2016]
[20] http://www.webopedia.com/TERM/P/polymorphism.html [Accessed: 06.03.2016]

programming. The class contains the pure virtual method *runAlgorithm*, which automatically renders the whole class pure virtual, i.e. no objects can be instantiated using this class. In order to create an instance of *Algorithm*, one has to define a sub-class that implements (inherits from) *Algorithm*, which is the main reason for this decision; to force developers to implement their algorithms according to the pre-given interface.

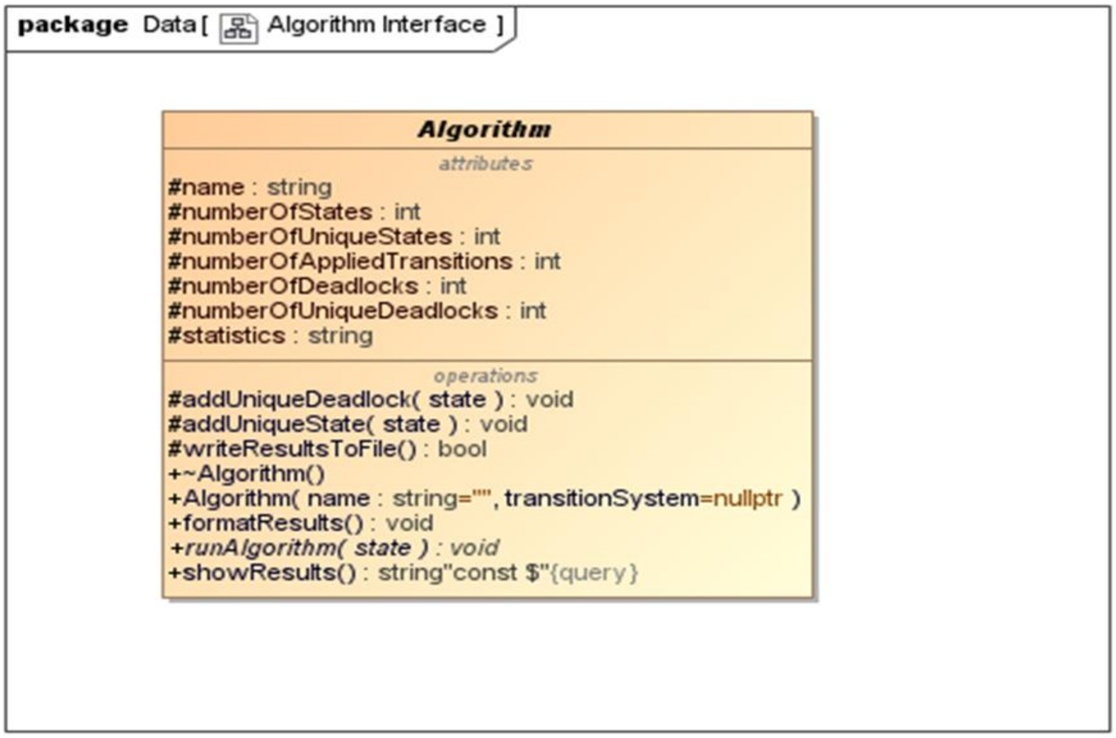

package Data [Algorithm Interface]
Algorithm
attributes
#name : string
#numberOfStates : int
#numberOfUniqueStates : int
#numberOfAppliedTransitions : int
#numberOfDeadlocks : int
#numberOfUniqueDeadlocks : int
#statistics : string
operations
#addUniqueDeadlock(state) : void
#addUniqueState(state) : void
#writeResultsToFile() : bool
+~Algorithm()
+Algorithm(name : string="", transitionSystem=nullptr)
+formatResults() : void
+runAlgorithm(state) : void
+showResults() : string"const $"{query}

Figure 4.4 Algorithm Interface.

Moreover, the class contains several members and methods, essential for keeping track of, and storing, the calculated statistics, as well as formatting the results, so that they can be displayed on screen, or written into a file. The most important statistics are held in the class members *numberOfStates*, *numberOfDeadlocks* etc. complying with the requirement analysis (Chapter 3.4).

Chapter 5

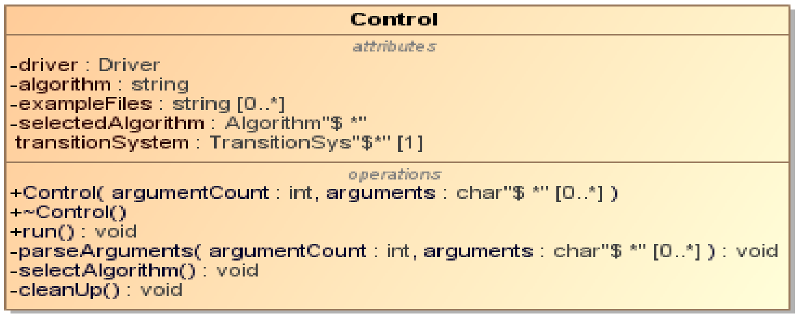
Implementation

This chapter provides detailed insights regarding the actual implementation of the classes in the tool, as well as the other utility classes that were iteratively added to the design to support the functionality within the course of the program. Additionally, we will examine the algorithms implemented, according to the tasks they carry out, and to the component they belong to. The most important aspects of the implementation can be categorized into five sections; start-up, parsing the input, generating transition systems, applying algorithms and formatting the output statistics.

5.1 Start-Up

The tool was set to be a console-application with one executable file that takes one or more input files as program arguments, in addition to the algorithm that should be applied on the given input, which will be further examined in the evaluation chapter (chapters 6.1 and 6.2).

The first class to be initialized, is the class *Control* (Fig.5.1).



```
Control
attributes
-driver : Driver
-algorithm : string
-exampleFiles : string [0..*]
-selectedAlgorithm : Algorithm"$ *"
 transitionSystem : TransitionSys"$*" [1]
operations
+Control( argumentCount : int, arguments : char"$ *" [0..*] )
+~Control()
+run() : void
-parseArguments( argumentCount : int, arguments : char"$ *" [0..*] ) : void
-selectAlgorithm() : void
-cleanUp() : void
```

Figure 5.1 Class *Control*.

The following sequence diagram[21] (Fig.5.2) demonstrates the program course in a simplified manner, yet sufficient for having an idea regarding the interaction between the class objects, and the chronological order of that interaction. It also provides information regarding the call order of the functions.

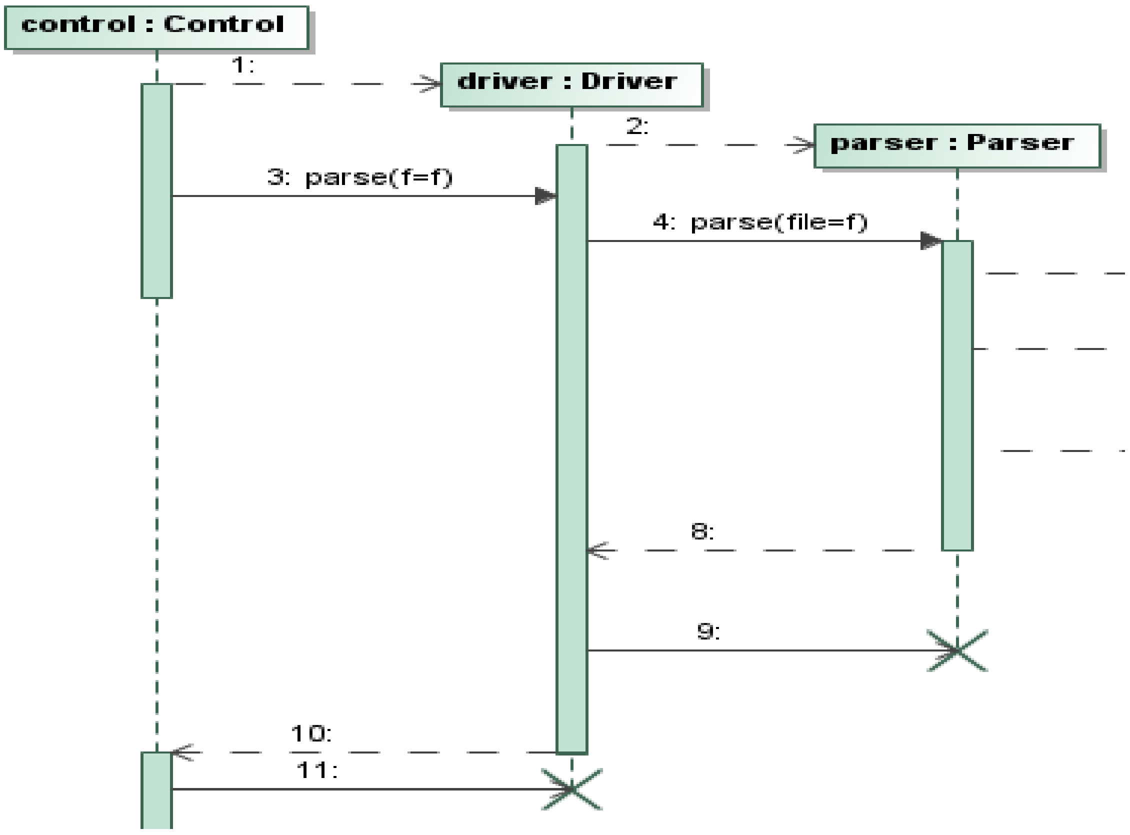


Figure 5.2 Parsing

[21] http://www.sparxsystems.com/resources/uml2_tutorial/uml2_sequencediagram.html [Accessed 11.03.2016]

Its only object instance is dynamically instantiated in the *main* function of the executable *FastPrototyper.cc* (Appendix B.1), which passes the program arguments as parameters into the class constructor. The constructor calls the method *parseArguments* in order to extract the given parameters and store them in their respective data containers. The program is then started with the method *run* (List.5.1), which takes care of the rest of the program course as follows:

```
void Control::run()
{
   for(unsigned int files = 0; files < exampleFiles.size(); ++files)
   {
      selectAlgorithm();

      if(!driver.parse (exampleFiles[files]))
      {
         transitionSystem->setUpSys();
         selectedAlgorithm->runAlgorithm(transitionSystem->getInitState());
         std::cout << formatter->showStatistics() << std::endl;
      }
      else
      {
         std::cerr << "Failed to parse input!\n";
         exit(1);
      }

      cleanUp();
   }
}
```

Listing 5.1 Method *run* of Class *Control*.

The *for* loop iterates over all given input files, doing the same for each one of them. However, we can choose only one algorithm per program-run as we can only pass one algorithm argument at program-start. The structure chosen can be extended with a *switch/case statement* or similar constructs when more algorithms are implemented.

The class control calls the *driver*, which in turn reads the input files (Chapter 5.2), parses them, and stores the extracted information temporarily until the method *setUpSys* from the class *TransitionSys* is called to generate a transition system (Chapter 5.3) for the currently active input file in the *for* loop. Afterwards, the method *runAlgorithm* of the selected algorithm is called to execute the algorithm onto the generated transition system. Finally, the *formatter* is called to display the calculated statistics. The method *cleanup* deletes all dynamically created objects, so that next input file can also be processed.

At this point, it is worth mentioning, that all of these objects (*selectedAlgorithm, transitionSystem* and *formatter*) are dynamically initiated, where the object *transitionSystem* is declared in the class *control* as follows (List.5.2):

```
#ifndef CONTROL_HH
#define CONTROL_HH
.
.
.
class Control
{
.
.
.
};

extern TransitionSys* transitionSystem;

#endif // ! CONTROL_HH
```

Listing 5.2 *TransitionSys* Dynamic Object *extern* Declaration.

The object is declared as an *extern pointer* object, as its actual initialization takes place in the *parser* (Chapter 5.2), during the parsing process. This is mainly because we need to pass a reference to this object into the *selectedAlgorithm* constructor (List.5.3), before the object is even initialized, which usually leads to a core dump.

As we can see, the passed *TransitionSys* object, is a pointer to a pointer. More details in that regard are provided in chapters 5.3 and 5.4.

```
#ifndef Algorithm_HH
#define Algorithm_HH
.
.
.
class Algorithm
{
.
public:
   Algorithm(std::string name = "", TransitionSys** transitionSystem = nullptr);
.
.
};

#endif // ! Algorithm_HH
```

Listing 5.3 Constructor of Class *Algorithm*.

5.2 Reading and Parsing Input

As demonstrated in the previous section, the method *run* of the class *Control* calls the class member *driver*, an object of the class *Driver*, for each input file, passing a single file as parameter into the member function *parse*. At this point the driver object takes over, and starts parsing the text.

The class *Driver* has one significant method, namely *parse* (List.5.4)

```
int Driver::parse (const std::string &f)
{
   file = f;
   scan_begin ();
   yy::Parser* parser = new yy::Parser(*this);
   parser->set_debug_level (trace_parsing);
   int res = parser->parse ();
   scan_end ();
   delete parser;
   return res;
}
```

Listing 5.4 Method *parse* of Class *Driver*.

The method takes a constant reference to the file (the file's name) that should be parsed. It utilizes the methods *scan_begin* and *scan_end* declared in the scanner (*Flex*) to start and stop the tokenizing. It also initializes the parser object, which in turn takes a pointer to the parsing context (the driver). The parser starts with the call of the method *parse*, which does the actual parsing. When the parsing is done, the driver ends the scanning and deletes the parser object, returning to the caller in the class *Control.*

As for the parser and scanner (*Bison* and *Flex*), we will not go into full detail on how they work. For the structure, compiling and classes generated by *Bison* and *Flex* please refer to Chapter 6.1. We will however examine the configuration files of the parser and scanner, since we have to define them according to our needs. List.B.2 and List.B.3 in Appendix B display the complete files.

The parser configuration file tells the parser which tokens it has to expect in which grammar rules, in order to perform a successful parsing. It also tells it which other classes it communicates with, such as *Driver*, *TransitionSys*, etc. (List.5.5)

```
%code
{
# include "Driver.hh"
# include "TransitionSys.hh"
# include "SymbolTable.hh"
# include "Program.hh"
Program* program;
TransitionSys* transitionSystem;
SymbolTable* symbolTable;
}
```

Listing 5.5 Code Section in the Parser Configuration File.

This code section also allows declaring objects of these classes, so that they can be used in the rules section. An example showing how the pointer object *transitionSystem* is initialized (List.5.6), shows how to use grammar rules.

```
declaration_list:   "<" system_name proc_count {symbolTable = new
SymbolTable;}
                declarations ">"
                {
                program = new Program(symbolTable, $3);
                transitionSystem = new TransitionSys($2, $3, program);};
```

Listing 5.6 Initializing a Class Object in the Rules Section of the Parser Configuration.

Whenever a grammar rule is matched by the parser, or a part of it, we can apply grammar rule actions, which are program code. In this case C++ code, since we integrated *Bison* in a C++ application. The code above shows how two class objects are initialized, whereas they also take parameters defined or initialized in other grammar rules, e.g., *symbolTable*. The *$* symbol refers to the single parts of a rule i.e. the first part on the left is *$$*, and the other parts are numbered respectively, e.g., *$1, $2* and so on. These symbols carry a pre-defined semantic value, which gets transferred from the leaves of a metaphoric parsing tree, all the way up, or down, to the roots. Thus we can access all the elements we need from the text in the input files, and use it in our tool. These elements will be further described, when we examine the classes *Program* and *SymbolTable*.

The scanner configuration file is very similar to that of the parser. It tells the scanner which lexical instances are expected in an input text (Fig.A.2 Appendix A), and how to handle them (List.5.7). For each matched lexical unit, the scanner creates a suitable token and sends it to the parser.

```
// Rules
{blank}+        loc.step ();
[\n]+           loc.lines (yyleng); loc.step ();
"-"             return yy::Parser::make_MINUS(loc);
"+"             return yy::Parser::make_PLUS(loc);
…
```

Listing 5.7 Token Rules Section in the Scanner Configuration.

The token rules, as in the parser configuration, also allow rule actions, which in turn are pre-defined *Flex* functions or value return statements going to the parser. The scanner also forward declares the methods *scan_begin* and *scan_end*, which, as discussed, belong to the driver to access a file and start scanning it, and at the end close it.

This sums up the first parsing pass using the parser and the scanner. If the input file is lexically or syntactically incorrect, the parser and the scanner return a message referring to the error. Otherwise, we move on to the second pass of the parsing done by the class *Program*, as demonstrated in the sequence diagram below (Fig.5.3)

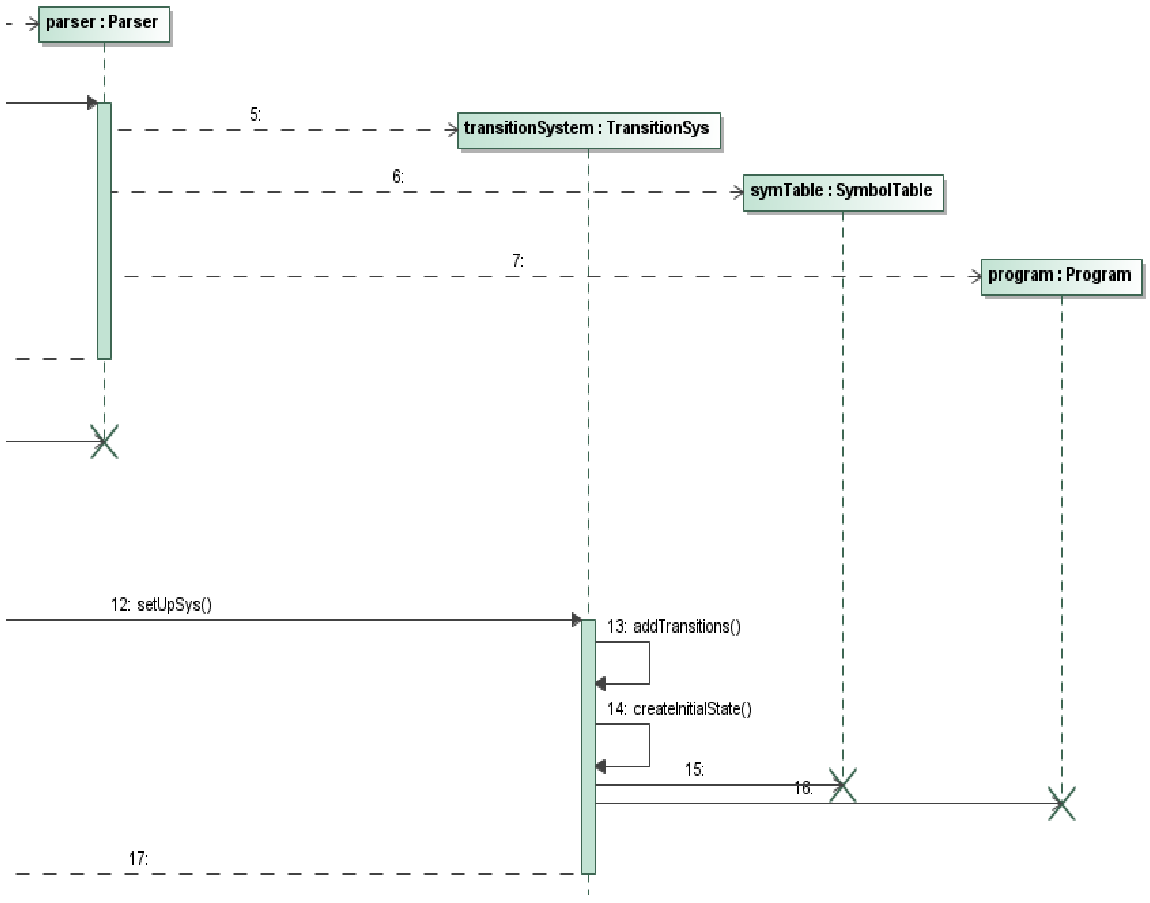


Figure 5.3 Second Parsing Pass

Even though the class *Program* (Fig.5.4) is only a secondary class compared to *TransitionSys*, it plays a big role in completing the parsing of the input file, so that we can generate a transition system.

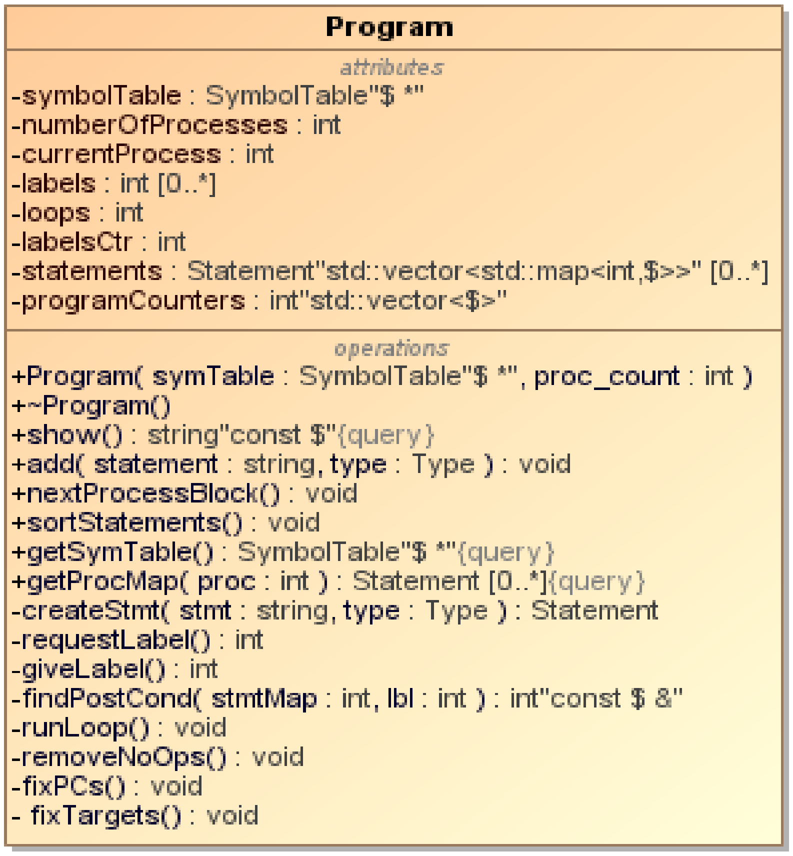

Program
attributes
-symbolTable : SymbolTable"$ *"
-numberOfProcesses : int
-currentProcess : int
-labels : int [0..*]
-loops : int
-labelsCtr : int
-statements : Statement"std::vector<std::map<int,$>>" [0..*]
-programCounters : int"std::vector<$>"
operations
+Program(symTable : SymbolTable"$ *", proc_count : int)
+~Program()
+show() : string"const $"{query}
+add(statement : string, type : Type) : void
+nextProcessBlock() : void
+sortStatements() : void
+getSymTable() : SymbolTable"$ *"{query}
+getProcMap(proc : int) : Statement [0..*]{query}
-createStmt(stmt : string, type : Type) : Statement
-requestLabel() : int
-giveLabel() : int
-findPostCond(stmtMap : int, lbl : int) : int"const $ &"
-runLoop() : void
-removeNoOps() : void
-fixPCs() : void
- fixTargets() : void

Figure 5.4 Class Program

Each program class object, corresponds to one input file. The method *add* is called by the parser as a grammar rule action whenever a grammar rule is matched, whose parts make up a program statement (a string representation of the statement and a type). These statements are then stored in a vector of maps. The indexes of the vector correspond to the program processes, and the map keys correspond to the program counter values of a process. The differentiation between process blocks is done by the parser, based on the grammar rules. The suitable action when parsing a new process block is to call the method *nextProcessBlock*, which triggers incrementing the number of process found, and thus the number of maps stored in the vector.

Internally, the class uses its private methods to complete the parsing. The method *createStmt* (List.5.8) is called by *add* using the same parameters, calling the constructor of the class *Statement* to create a statement object. It also assigns reference labels to statement, whose types are conditional, loop or returns of those, so that it can calculate the branching addresses in further steps.

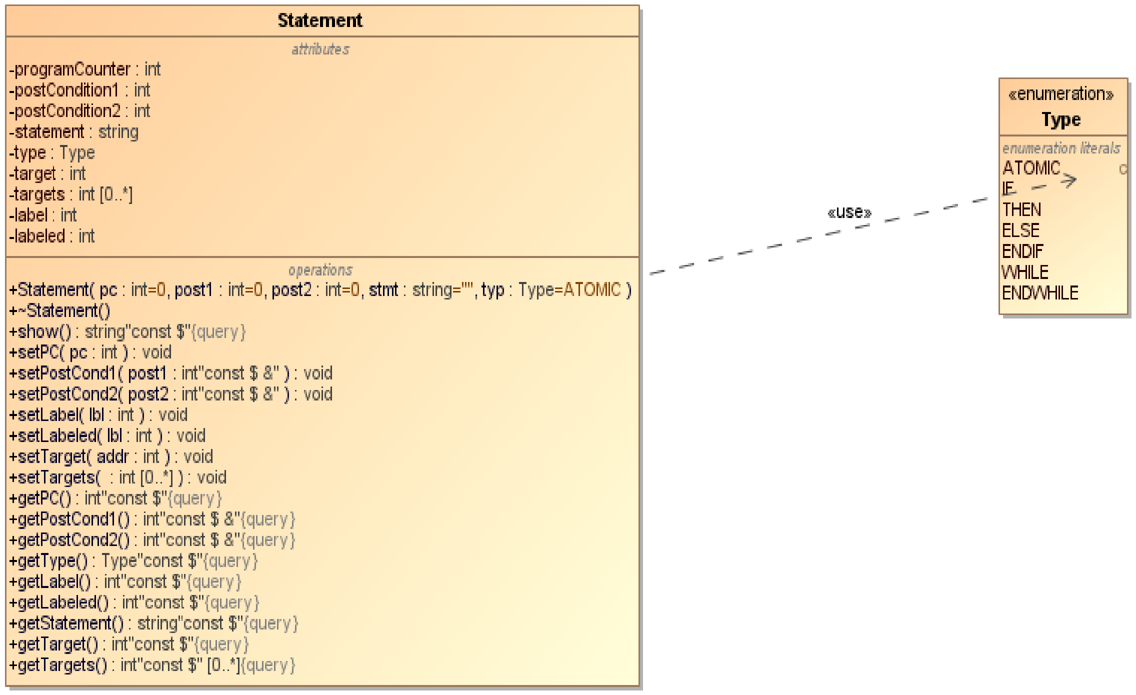
```
Statement Program::createStmt(std::string stmt, Type type)
{
   int pc = programCounters[currentProcess];
   int lbl = 0;

   Statement statement(pc, pc+1, pc+1, stmt, type);

   if(ENDIF == type || ENDWHILE == type || ELSE == type)
   {
      lbl = giveLabel();
      statement.setLabeled(lbl);
   }
   if(IF == type || ELSE == type || WHILE == type )
   {
      lbl = requestLabel();
      statement.setLabel(lbl);
   }

   return statement;
}
```

Listing 5.8 Method *createStmt* of Class *Program*.

After the whole input file is parsed, the function *parse* called by the class *Control* returns to the function *run* (List.5.1), which triggers generating a transition system. The method *setUpSys* calls the method *sortStatements*, which performs a series of private function calls, in order to set the correct branching addresses, remove redundant statements that were added to assist the labeling, set the correct program counters for the statements, and finally set the correct variable addresses in the symbol table.

Three other classes that we have not examined yet, are *Statement* (Fig.5.5), *SymbolTable* and *Variable* (Fig.5.6). Due to the complexity of the program statements, it was necessary to encapsulate the knowledge regarding the attributes a statement can carry.

The most important attributes in the class *Statement* that are not necessarily self-explanatory, are *postCondition1*, *postCondition2*, *target* and *targets*. *postCondition1* holds the value of the program counter of the directly following statement. *postCondition2* is only relevant to statements of special type (conditionals and loops), where it holds the value of the corresponding branch address; in most cases when the *boolean* condition is not met. *target* is the address of the variable on the left side of a statement (if the statement is an assignment), and *targets* is a vector with the respective addresses of the right side variables.



Statement
attributes
-programCounter : int
-postCondition1 : int
-postCondition2 : int
-statement : string
-type : Type
-target : int
-targets : int [0..*]
-label : int
-labeled : int
operations
+Statement(pc : int=0, post1 : int=0, post2 : int=0, stmt : string="", typ : Type=ATOMIC)
+~Statement()
+show() : string"const $"{query}
+setPC(pc : int) : void
+setPostCond1(post1 : int"const $ &") : void
+setPostCond2(post2 : int"const $ &") : void
+setLabel(lbl : int) : void
+setLabeled(lbl : int) : void
+setTarget(addr : int) : void
+setTargets(: int [0..*]) : void
+getPC() : int"const $"{query}
+getPostCond1() : int"const $ &"{query}
+getPostCond2() : int"const $ &"{query}
+getType() : Type"const $"{query}
+getLabel() : int"const $"{query}
+getLabeled() : int"const $"{query}
+getStatement() : string"const $"{query}
+getTarget() : int"const $"{query}
+getTargets() : int"const $" [0..*]{query}
«use»
«enumeration»
Type
enumeration literals
ATOMIC
IF
THEN
ELSE
ENDIF
WHILE
ENDWHILE

Figure 5.5 Class *Statement*

The class *SymbolTable* has the job of, managing and storing objects from the class *Variable* and offering methods to access them and their attributes. The most important of which being, *initialValue*, *length* (for arrays), and *address* (their address in the memory of the tool).

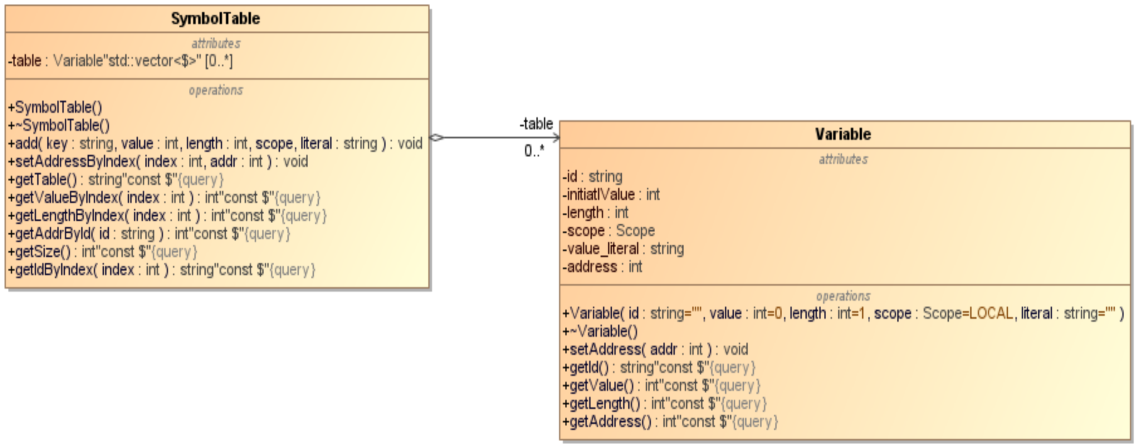

SymbolTable
attributes
-table : Variable"std::vector<$>" [0..*]
operations
+SymbolTable()
+~SymbolTable()
+add(key : string, value : int, length : int, scope, literal : string) : void
+setAddressByIndex(index : int, addr : int) : void
+getTable() : string"const $"{query}
+getValueByIndex(index : int) : int"const $"{query}
+getLengthByIndex(index : int) : int"const $"{query}
+getAddrById(id : string) : int"const $"{query}
+getSize() : int"const $"{query}
+getIdByIndex(index : int) : string"const $"{query}
-table
0..*
Variable
attributes
-id : string
-initiatlValue : int
-length : int
-scope : Scope
-value_literal : string
-address : int
operations
+Variable(id : string="", value : int=0, length : int=1, scope : Scope=LOCAL, literal : string="")
+~Variable()
+setAddress(addr : int) : void
+getId() : string"const $"{query}
+getValue() : int"const $"{query}
+getLength() : int"const $"{query}
+getAddress() : int"const $"{query}

Figure 5.6 Classes *SymbolTable* and *Variable*

5.3 Building Transition Systems

After all the parsing passes are complete, i.e., after the method *sortStatements* of the class *Program* has returned (Chapter 5.2), the class *TransitionSys* (Fig.4.3) continues building the transition system of the parsed input file (example program) in its method *setUpSys*. At this point, we have to take a step back in the order of function calls in that method, as the method creates the initial memory and the initial state using the private methods *createMemory* (List.5.9) and *createInitialState*.

The initial state is simply created by calling the constructor of the class *State*, into which we pass the initial memory object as a function parameter. The initial memory is built using the variables stored in the symbol table, utilizing their attributes (*address*, *length*, etc.), in order to create the memory array (Chapter 3.5).

```
void TransitionSys::createMemory()
{

   SymbolTable* table = program->getSymTable();
   int num = numberOfProcesses;
   int size = table->getSize() + num;
   int value = 0;
   int index = 0;

   for(int i = 0; i < num; ++i)
   {
      initialMemory.add(0);
   }

   while(num < size)
   {
      value = table->getValueByIndex(index);
      table->setAddressByIndex(index, num);

      for(int i = 0; i < table->getLengthByIndex(index); ++i)
      {
         initialMemory.add(value);
         num++;
      }
      index++;

   }
}
```

Listing 5.9 Method *createMemory* of Class *TransitionSys.*

The last step in building a transition system is adding the program statements by calling the method *addTransitions* (List.5.10), whereas they are now referred to as transitions.

As demonstrated, the method calls the constructor of the class *Transition* in each iteration going over the map of statements, creating a corresponding transition object with the needed parameters, which are the attributes of a transition object (Fig.4.3). In the end, all program statements are stored as transition objects in the class *TransitionSys*, in a vector of maps (Chapter 4.2). Finally, the program object is deleted, as it is no longer needed for the following steps; thus we have a complete transition system object, containing all the needed information for applying an algorithm.

```
void TransitionSys::addTransitions()
{

   for(int proc = 0; proc < numberOfProcesses; proc++)
   {

      std::map<int, Statement> stmtMap = program->getProcMap(proc);
      std::map<int, Transition> tranMap;
      int pc = 0;
      int addr = 0;

      for(auto a : stmtMap)
      {
         addr = a.second.getTarget();
         std::vector<int> targets = a.second.getTargets();
         std::string exp = a.second.getStatement();
         if("" != exp)
         {
            Transition t(proc, a.second.getPC(), addr, exp, targets,
a.second.getPostCond1(), a.second.getPostCond2());
            tranMap[pc] = t;
            pc++;
         }
      }

      transitions.push_back(tranMap);

   }
}
```

Listing 5.10 Method *addTransitions* of Class *TransitionSys*.

5.4 Applying Algorithms

Now that the transition system is set up, we can apply the algorithm chosen at the program start; in this case the algorithm *FullSearch* (List 5.18), which implements the interface

Algorithm (Fig.4.4), and whose functionality will be demonstrated, as all other potential algorithms have to implement the same methods.

We left off at the return call of the method *setUpSys* in the method *run* (List.5.1) of the class *Control* (Fig.5.1). Now we will examine the implementation behind the function call: *selectedAlgorithm->runAlgorithm(transitionSystem->getInitState());*

But first, we will take a look at the constructor (List.5.11) of the interface class *Algorithm*.

```
Algorithm::Algorithm(std::string n, TransitionSys** t):
   name(n),
.
.
.
```

Listing 5.11 Constructor of Class *Algorithm*

As we notice in the head, the constructor takes two arguments, the name of the transition system, and a pointer object to a pointer to the transition system object. In Chapter 5.2 (List.5.4), we have examined how a transition system pointer object is declared for the first time, and then initialized in the grammar rule actions. However, it is also declared using the *extern* directive in the class *Control* (List.5.2). An algorithm on the other hand, expects a pointer to a transition system object, which in our case, is already a pointer to an object that cannot be directly dereferenced; therefore, the class member *transitionSystem* has to be defined as a pointer to a pointer.

The method *runAlgorithm* (List.5.12) takes the initial state of a transition system as an argument, in order to start the execution. The method, as already mentioned in Chapter 4.3, is a pure virtual method that must be implemented by any sub-class inheriting from *Algorithm.* The following sequence diagram (Fig.5.7) demonstrates the implementation of the algorithm.

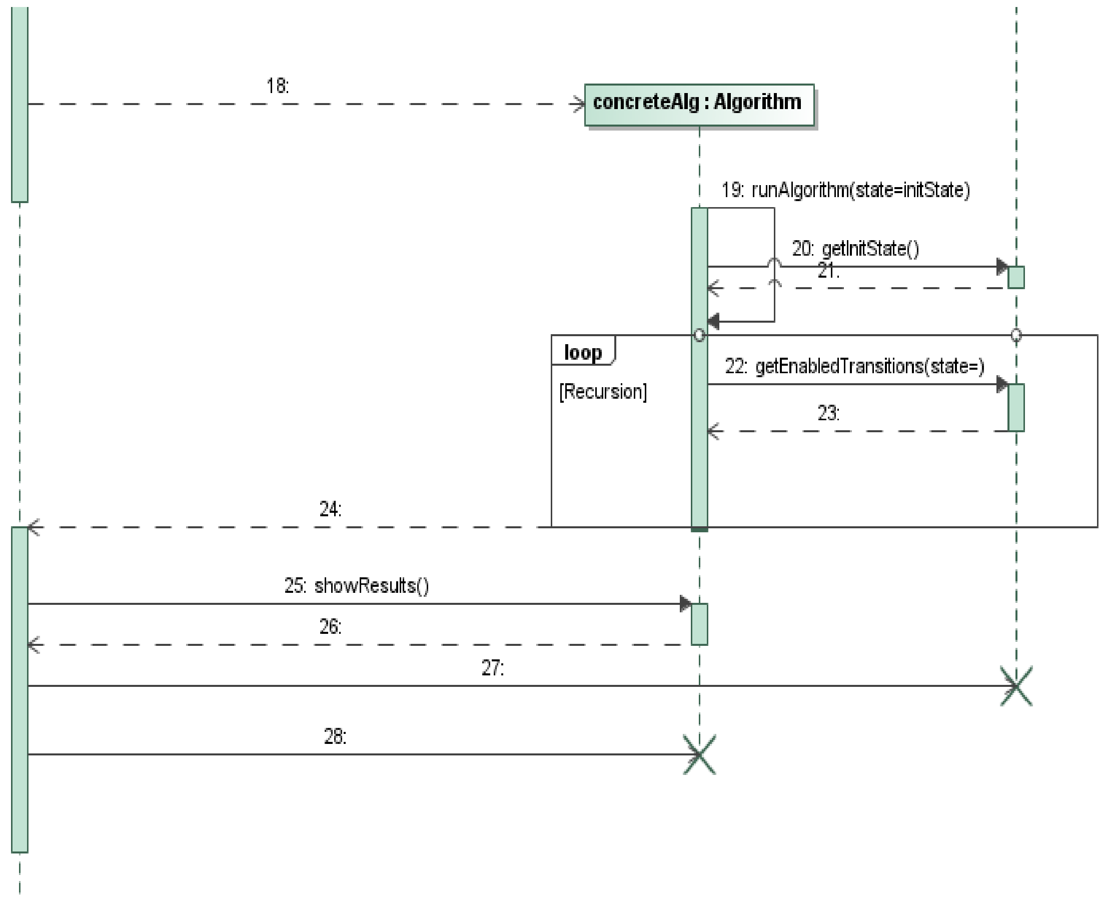


Figure 5.7 Algorithm Implementation

```
void FullSearch::runAlgorithm(State oldState)
{

   std::vector<Transition> enabledTransitions = (*Algorithm::transitionSystem)-
>getEnabledTransitions(&oldState);

   if(enabledTransitions.empty())
   {
      Algorithm::numberOfDeadlocks++;
      addUniqueDeadlock(oldState);
   }

   for(auto transition : enabledTransitions)
   {

      State newState;

      if(transition.apply(oldState, newState))
      {
         addUniqueState(newState);
         newState.setPrevious(Algorithm::numberOfStates);
         Algorithm::numberOfStates++;
         Algorithm::numberOfAppliedTransitions++;
         newState.setNumber(Algorithm::numberOfStates);
         runAlgorithm(newState);
      }

   }
}
```

Listing 5.12 Method *runAlgorithm* as Implemented in *FullSearch*.

This method contains all the logic required for the algorithm to run its course. First, it calls the method *getEnabledTransitions* (List.5.13) passing the initial state as parameter, in order to obtain the transitions that can be applied (executed) at the current state, which is the initial state at the start of the algorithm. These transitions are temporarily stored in the vector *enabledTransitions.* If there are not any enabled transitions, the method detects a deadlock, the state of which is automatically added to the set of all deadlocks; otherwise the algorithm iterates through all enabled transitions, respectively applying (List.5.16) them onto the current state (*oldState*), and creating a new resulting state (*newState*). The algorithm contributes to the information required for the statistics by also adding the new dynamically calculated states to the set of all states.

The key feature in this algorithm is the recursive call of the method *runAlgorithm* demonstrated in this line:

```
runAlgorithm(newState);
```

For each new calculated state, the algorithm starts from the top, repeating all the steps in order to complete a *Full Depth Search*, which is a specific characteristic of the full search algorithm.

Determining, whether a transition is enabled or not at a certain state, is done partly in the method *getEnabledTransitions*.

```
const std::vector<Transition> TransitionSys::getEnabledTransitions(State *state)
const
{
   std::vector<Transition> enabledTransitions;

   for(auto a : transitions)
   {
      for(auto b : a)
      {
         if(b.second.isEnabled(state))
         {
            enabledTransitions.push_back(b.second);
         }
      }
   }

   return enabledTransitions;
}
```

Listing 5.13 Method *getEnabledTransitions* of Class *TransitionSys*.

The method's task is to go through all transitions existing in the transition system, where it calls the method *isEnabled* (List.5.14), respectively pushing the enabled transitions into the vector *enabledTransitions*, which in turn is the return value of the function.

```
bool Transition::isEnabled(State *state)
{
    bool enabled = false;

    if(programCounter == state->getPC(processNumber))
    {
       enabled = true;
    }
    return enabled;
}
```

Listing 5.14 Method *isEnabled* if Class *Transition.*

The criterion that determines, whether a transition is enabled or not, is the program counter value of the transition at hand. This value is compared to the value in the memory snapshot at the state passed into the method (Chapter 4.2). If both values are equal to each other, the transition is enabled (executable) at this state, otherwise it is not. If no transitions are enabled at a certain state, the algorithm has either reached a deadlock, or a final state, which is also considered a deadlock in regards to the statistics.

The class *Algorithm* has two class members dedicated for keeping track of the unique states and unique deadlocks (List.5.15). Their utility methods, try to add each calculated state, depending on its type (*state* or *deadlock*), to the corresponding data container (C++ *set*). A C++ set does not allow inserting duplicates i.e. objects that are identical according to pre-defined criteria. It utilizes the default *less* operator to determine which object goes before the other, and thus deciding if two objects are the same[22]. To achieve this, we overloaded the *less* operator in the class *State* (List.B.4), and the *equal to* operator in class *Memory* (List.B.5), since states differ only in the content of the memory (Chapter 4.2).

```
    std::set<State> uniqueStates;
    std::set<State> uniqueDeadlocks;
```

Listing 5.15 Private Data Containers for States and Deadlocks in Class *Algorithm.*

As for the core of the whole process, the method *apply* offered by the class *Transition* performs a series of function calls as follows:

[22] http://www.cplusplus.com/reference/set/set/set/ [Accessed: 11.03.2016]

```
bool Transition::apply(State oldState, State &newState)
{
   currentState = oldState.cloneState();

   if(isEnabled(&currentState))
   {
      int result = evaluateExp();
      executeTransition(result);
      newState = currentState.cloneState();
      return true;
   }
   else
   {
      return false;
   }
}
```

Listing 5.16 Method *apply* of Class *Transition*.

The method takes two arguments, a copy of the current state, and a reference to the new state, which isn't initialized yet. It checks again, if the transition is enabled at the given state, and evaluates the expression (program statement) by calling the private method *evaluateExp*, which dynamically builds an abstract syntax tree for the expression, as expressions are stored as their string representation.

The tree is then evaluated using an abstract syntax tree evaluator (Fig.5.8), which returns an integer value of the evaluated expression. Afterwards, this value is used to apply the transition to the state by calling the private method *executeTransition*, which differentiates between two types of statements, assignments and conditionals (*intExp* and *BooleanExp*). The later causing changes only to the program counter values in the state memory, whereas the former actually changes the values of the variables in the state memory.

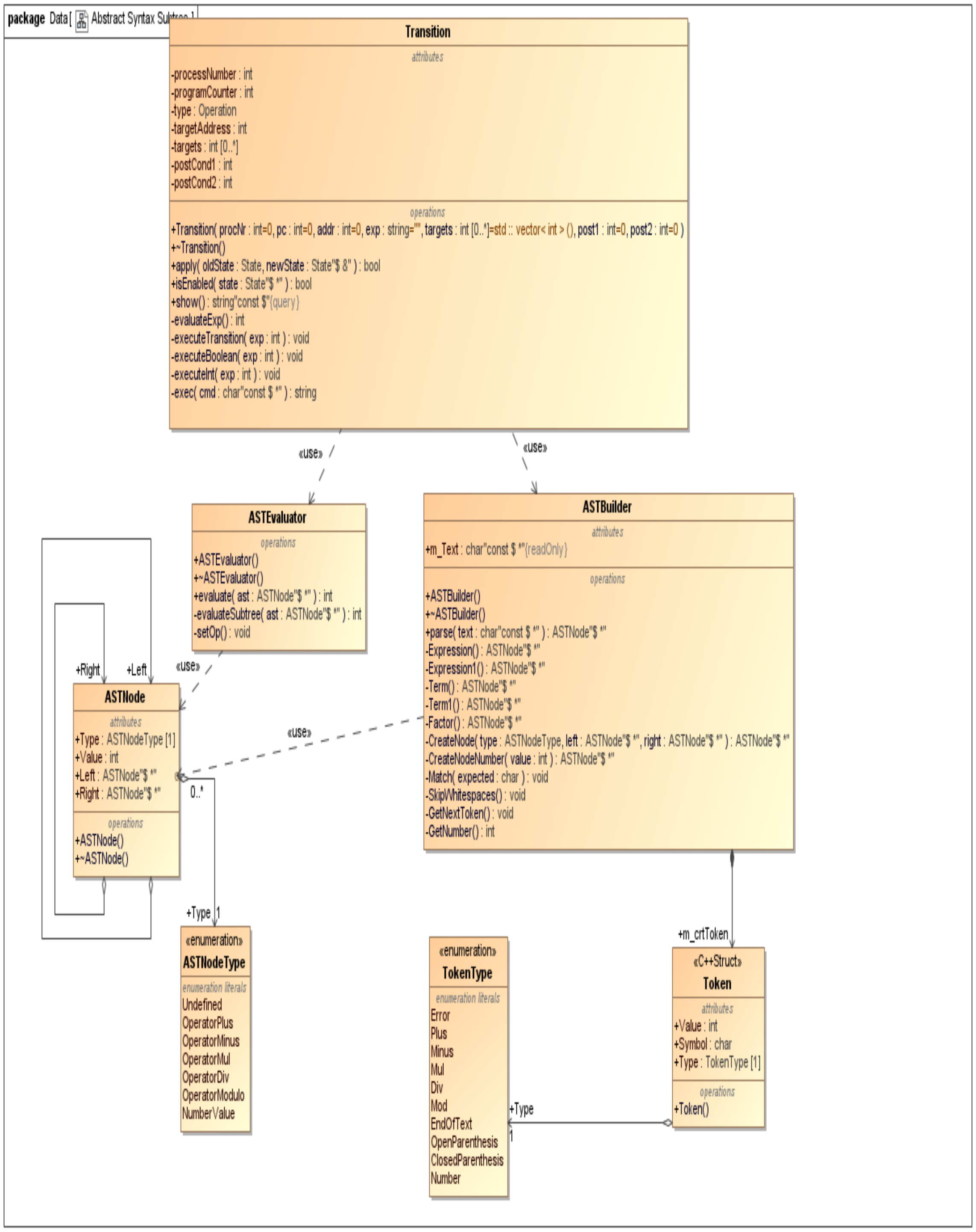


Figure 5.8 Abstract Syntax Subtree

After a transition is successfully applied, the function call returns with the now initialized object *newState* using the method *cloneState*, which is further used in the implementation of the algorithm, as demonstrated in the previous sections (List.5.12). The algorithm increments the number of acquired states, the number of transitions, and starts over from the top by recursively calling itself again. A unique state is also identified using the data container in the class *Algorithm*.

5.5 Formatting Output

The last order of business after the completion of the algorithm execution, is to display the statistics on the screen. The class *Formatter* (List.5.17) manages collecting the statistics from the class *Algorithm*, and constructs a formatted string representation that can be used as output in the console, which we will examine further in the evaluation chapter.

```
class Formatter
{
public:
   Formatter(Algorithm *selectedAlgorithm);
   virtual ~Formatter();
   const std::string showStatistics();
   void formatStatistics();

private:
   Algorithm* selectedAlgorithm;
   std::string statistics;
};
```

Listing 5.17 Class *Formatter*

This class is declared as *friend* in the algorithm interface (List.5.18), so that it can access the protected class members of *Algorithm*, without having to define utility functions for that purpose.

```
class Algorithm
{
   friend class Formatter;
.
.
.
};
```

Listing 5.18 *friend* Class Declaration of *Formatter* in *Algorithm*

Chapter 6

Evaluation

This chapter demonstrates the settings and conditions required, in order to run the tool. It also shows how to start the tool with the program arguments on the target system, in addition to the final output displayed in the console. Finally, this chapter examines the achieved result, showing how the accomplished work meets the development requests.

6.1 Setup

As stated earlier in Chapter 3.5, the source code of the tool was set to be written in C++11, and compiled using GNU GCC. However, two components in the tool, namely the parser and the scanner, are external tools (Chapter 5.2 *Bison* and *Flex*), which are integrated within our tool to assist reading and parsing the input. These tools are managed by the developer only through their configuration files, *Parser.yy* and *Scanner.ll* (List.B.2 and List.B.3 Appendix B). Nevertheless, the two configuration files are mainly used to generate the parser and scanner C++ header and implementation files, which is done by the *Flex* and *Bison* compilers. To insure a standardized way of compiling the source code, and linking the object files, we decided to use a *Makefile*. All the compilation options regarding the GCC, *Flex* and *Bison* compilers are in that Makefile.

On a different note, we also provided a *README* file where we describe the system requirements, and how to compile and run the tool properly, with the desired input files and the desired algorithm.

Naturally, the *Bison* and *Flex* compilers can be run separately, and their generated files can be linked to our code by including the needed files of those in the class *Driver*.

The classes generated are: *location.hh, Parser.cc, Parser.hh, position.hh, Scanner.cc* and *stack.hh*.

6.2 Experiments

The first example program used in the first experiment is rather compact, containing only two process blocks with one program statement each (List.6.1), performing a simple assignment to a variable and an array variable.

```
<readers_writers 2, x 1 1, l 2 2>
[
   x := 42
]
[
  l[1] := 42
]
```

Listing 6.1 Example Input File with two Process Blocks

The first line in the example program is the declaration block, which contains the name of the example program, the number of process blocks, and the initialization of the variables used; where the variables are declared with their respective lengths and values. The square brackets indicate the start and end of a process block and enclose its statements.

To run the tool with this example file as input, and the algorithm *FullSearch*, we type the following command in the console:

```
./FastPrototyper -f readers_writers
```

In this case, the input file is located in the same directory as the tool executable *FastPrototyper.* Afterwards, the tool starts the execution and the output is displayed in the console (List.6.2).

```
Execution of Full Search on:
The Transition System: readers_writers - Number of Processes:
2
---------------------------------------------------
Process: 0
---------------------------------------------
Program Counter: 0 - Expression: x:=42
Type: Assignment

-------------------------------------
Process: 1
---------------------------------------------
Program Counter: 0 - Expression: l[1]:=42
Type: Assignment

-------------------------------------

The Memory:
Index: 0 => Value: 0 - Index: 1 => Value: 0 - Index: 2 =>
Value: 1 - Index: 3 => Value: 2 - Index: 4 => Value: 2 -

--------------------------------------
Number of visited states w/out the initial state: 4
Number of Unique states: 3
Number of applied transitions: 4
Number of deadlocks: 2
Number of unique deadlocks: 1
---------------------------------------------------
---------------------------------------------------
```
Listing 6.2 Tool Output for Example Program

The output of the tool includes the conditions, under which the tool was run. It also shows the process blocks found in the input file, to insure the correctness of the generated transition system. A simple overview of the values stored in the tool memory is included as well. Lastly, the statistics regarding the execution are displayed.

Since the example program is relatively simple, we can verify the quality of the execution of the algorithm manually, as shown in the state graph (Fig.6.1). We will call the transitions corresponding to the program statements *t1.1* and *t2.1*. The black filled states are accept (final) states.

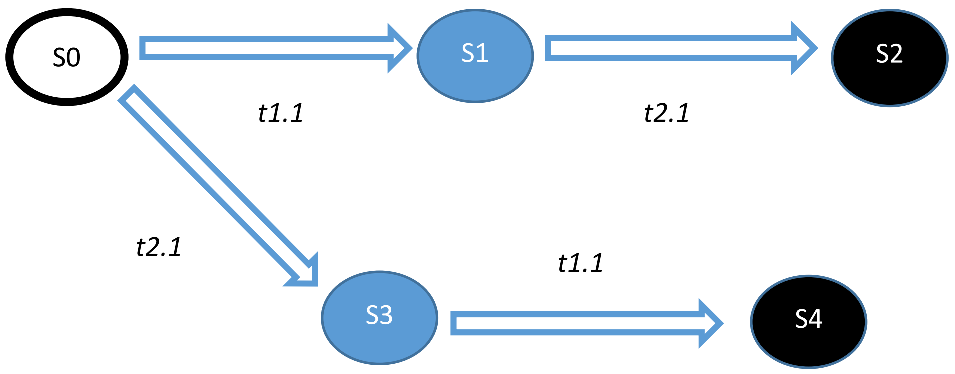


Figure 6.1 State Graph of Example Program

As demonstrated, the statistics calculated by the tool are correct. The number of dynamically calculated states, same as the number of applied transitions, is four; there exist two deadlocks, of which only one is unique, since *s2* = *s4*, and therefore three unique states, namely *s1*, *s2* and *s3*.

However, we can also formally verify the correctness of the statistics by using this mathematical formula: *Number of States* = *i*t+1* where *i* is the number of interleavings (paths leading to an accept state) and *t* is the number of transitions in one interleaving. The formula calculates the number of all visited states (not only unique states) plus the initial state.

To calculate the number of interleavings we use this formula:

Number of Interleavings = $(pt)! / (t!)^p$, where *p* is the number of process blocks and *t* is the number of transitions in each process block, which must be the same in each process block to apply this formula.

This means:

Number of Interleavings i = $(2*1)! / (1!)^2 = 2 / 1 = 2$

Number of States Including the Initial State = *2*2+1 = 5*

The second example program has three process blocks with different numbers of program statements (List.6.3). The first and third processes have one program statement each, whereas the second has two.

```
< readers_writers 3, x 1 2, y 1 2>
[
   x := 1
]
[
   y := 5
   x := 4
]
[
   y := 6
]
```

Listing 6.3 Example Input File with three Process Blocks

The tool execution is done the same way as in the first experiment. The generated output (List.6.4) shows the information regarding the example program and the calculated statistics.

In this case, it is not possible to verify the correctness of the number of visited states, as the process blocks have different number of program statements. However, we can verify the number of interleavings corresponding to the number of all deadlocks in the state graph using the formula:

Number of Interleavings = $(\sum_{i=1}^{p} ti)! \,/\, \prod_{i=1}^{p}(ti!)$, where *p* is the number of all process blocks, and each process *i* has *ti* transitions.

```
Execution of Full Search on:
The Transition System: readers_writers - Number of
Processes: 3
---------------------------------------------------
Process: 0
---------------------------------------------
Program Counter: 0 - Expression: x:=1
Type: Assignment
-------------------------------------
Process: 1
---------------------------------------------
Program Counter: 0 - Expression: y:=5
Type: Assignment

Program Counter: 1 - Expression: x:=4
Type: Assignment
-------------------------------------
Process: 2
---------------------------------------------
Program Counter: 0 - Expression: y:=6
Type: Assignment
-------------------------------------
The Memory:
Index: 0 => Value: 0 - Index: 1 => Value: 0 - Index: 2 =>
Value: 0 - Index: 3 => Value: 2 - Index: 4 => Value: 2 -
Index: 5 => Value: 2 -
--------------------------------------
Number of visited states w/out the initial state: 34
Number of Unique states: 24
Number of applied transitions: 34
Number of deadlocks: 12
Number of unique deadlocks: 5
---------------------------------------------------
```

Listing 6.4 Tool Output for Example Program

Applying the values at hand we calculate:

Number of Interleavings i $= (\sum_{i=1}^{3} ti)! / \prod_{i=1}^{3}(ti!) = (1+2+1)! / (1!*2!*1!) = 4! / 2$

$= 24 / 2 = 12.$

Our last example program (List.A.4 Appendix A) is relatively more complex than the first and second examples, containing several conditional statements and a loop. The calculated statistics (List.6.5) upon the execution of the tool display the degree of complexity that can be realized by the tool. Unfortunately, it is not possible to verify the correctness of the statistics within this thesis, as the resulting state graph cannot to be drawn manually within human capabilities. The mathematical formulas introduced earlier also fail to provide sufficient proof, as they are not suitable for working with conditional statements, as they require that all transitions are enabled, which depends on the values in the Boolean expressions.

```
--------------------------------------
Number of visited states w/out the initial state: 1133
Number of Unique states: 542
Number of applied transitions: 1133
Number of deadlocks: 157
Number of unique deadlocks: 5
---------------------------------------------------
---------------------------------------------------
```

Listing 6.5 Tool Output for Example Program (Incomplete)

6.3 Extending and Maintaining The Tool

One of the most important aspects regarding the tool, is its ability for extension and maintenance, so that it can be used by developers and scholars in further research. Mainly to allow implementing more elaborate POR-Algorithms, and to apply them on more complex example programs; as well as adjusting and extending the grammar of the input.

The grammar definition at hand (List.A.1 Appendix A), in its abstract form, can be extended according to the features demanded by developers. The following example demonstrates, how additional syntactic and semantic rules can be added to the grammar definition, and how the responsible tool components can be extended in regards to the new added features.

Considering the following section of the grammar:

```
statement:      atomic_statement
            {program->add($1, ATOMIC);}
            | conditional
            {program->add("endif", ENDIF);}
            | loop
            {program->add("endwhile", ENDWHILE);};
```

The grammar rule defines three types of statements; loops, conditionals and atomic statements, whereas atomic statements are assignmens.

We can extend the types of accepted statements, adding a *switch/case* statement, defined as follows:

```
statement:      atomic_statement
            {program->add($1, ATOMIC);}
            | conditional
            {program->add("endif", ENDIF);}
            | loop
            {program->add("endwhile", ENDWHILE);}
            | switch
            {program->add(“endswitch”, ENDSWITCH);};
```

Of course, we have to further define the *switch* non-terminal symbol. For example, the new switch statement accepts integer expressions:

```
switch:         “switch” int_expr
              {program->add(“switch” + $2, SWITCH);}
              “{“ case_block
              default_case “}”
              {};
```

The case block must also be defined. We will restrict the case code to atomic statements for simplicity reasons, however allowing more than one case, and not allowing a missing default case:

```
case_block:          case case_block       {}
                     | case                {};
case:          "case" int_exp ":"    {program->add($2, CASE)}
               atomic_statement      { program->add($4, ATOMIC);}
               "break"               { program->add("endcase", BREAK);};
default_case: atomic_statement       { program->add($1, ATOMIC);}
               "break"               { program->add("endcase", BREAK);};
```

The scanner also needs to be informed about the new tokens:

```
// Rules
{blank}+       loc.step ();
[\n]+          loc.lines (yyleng); loc.step ();
"if"           return yy::Parser::make_IF(loc);
.
.
.
 "switch"      return yy::Parser::make_SWITCH(loc);
"case"          return yy::Parser::make_CASE(loc);
"break"        return yy::Parser::make_BREAK(loc);
```

The responsible class (*Program*) uses the same method *add* without having to change its prototype. However, we have to define the new statement types (*SWITH, CASE* and *BREAK*), which is done in the class *Statement*:

```
enum Type { ATOMIC = 0, IF, THEN, ELSE, ENDIF, WHILE, ENDWHILE,
SWITCH, ENDSWITCH, CASE, ENDCASE = 10 };
```

And in the method *createStatement* of class *Program*:

```
Statement Program::createStmt(std::string stmt, Type type)
{
   int pc = programCounters[currentProcess];
   int lbl = 0;

   Statement statement(pc, pc+1, pc+1, stmt, type);

   if(ENDIF == type || ENDWHILE == type || ELSE == type || ENDSWITCH
== type || ENDCASE == type)
   {
      lbl = giveLabel();
      statement.setLabeled(lbl);
   }
   if(IF == type || ELSE == type || WHILE == type || SWITCH == type || CASE
== type )
   {
      lbl = requestLabel();
      statement.setLabel(lbl);
   }

   return statement;
}
```

Any other aspects in the grammar can be similarly changed, added or extended, offering developers the freedom to adjust the grammar of the input language to suit their needs.

The second key aspect in regards to maintenance, is the algorithm interface, which offers a standardized skeleton for implementing state exploration algorithms, including ones based on partial order reduction.

A developer only needs to create a new class that implements (inherits from) the interface *Algorithm* in order to apply their algorithms onto transition systems. For example, if we want to implement the algorithm *Dynamic Partial Order Reduction*, as introduced by

Flanagan and Godefroid[23], we create a class e.g. *DPOR* and have it inherit from *Algorithm* as follows:

```
#ifndef DPOR_HH
#define DPOR _HH

#include "Algorithm.hh"

class DPOR: public Algorithm
{
public:
   DPOR (std::string name = "", TransitionSys** transitionSystem = nullptr);
   ~ DPOR ();
   void runAlgorithm(State state);

private:
   void addUniqueState(State state);
   void addUniqueDeadlock(State state);

};
#endif // ! DPOR _HH
```

The implementation logic of the algorithm is encapsulated in the implementation file *DPOR.cc*. More precisely, the logic is implemented solely inside the method *runAlgorithm*. Thus, we don't need to worry about connecting our new algorithms to the entire system, since the methods *addUniqueState* and *addUniqueDeadlocks* are implemented in the class *Algorithm*.

[23] https://users.soe.ucsc.edu/~cormac/papers/popl05.pdf [Accessed: 03.03.2016]

Chapter 7

Conclusion

This chapter provides a brief summary of the topics and ideas discussed in the thesis, as well as the most important findings. Moreover, it offers some final thoughts and perspectives with regard to the achieved work at hand.

7.1 Summary

Throughout this thesis, we examined the theoretical aspects of model checking and model checking exploration algorithms based on partial order reduction in the field of distributed systems. We also demonstrated, the important role of transition systems in representing state space graphs of distributed systems, scaling it down to simulating the unpredictability of system behavior in multi-threaded environments; emphasizing on the gravity of the state space explosion problem, and the appliance of the above mentioned theory as means for alleviating this problem.

Furthermore, we presented an adequate, yet simple solution for aiding scholars and developers in their research for constructing the most fitting and appropriate POR-algorithms, in addition to quickly implementing and applying those, onto transition systems that meet the requirements and wishes of developers.

The exhibited process of analyzing, designing and implementing the tool, shows a systematic approach to putting the theoretical knowledge into practice, connecting the abstract theory of computer science with pragmatic exercise of the concepts of modern and agile software engineering and object oriented programming; however, maintaining a sense of control over the software using fairly acceptable procedural techniques.

The result of this work is thus, a moderately complex tool that can be utilized in developing and inspecting POR-algorithms, which also possesses a sufficient degree of capability for maintenance and extension.

7.2 Perspective

As proven by the performed experiments, the tool is capable of reading and parsing simple example programs, as well as more elaborate ones. It can transform these programs into transition systems that perfectly correspond to the introduced formal theory without dismissing any practical details. Moreover, the tool complies with the presented requests and prerequisites in regards to both of the functional and non-functional aspects, as it clearly delivers reliable statistics in respect to applying algorithms as means for exploring transition systems, while preserving the ability to be extended and adapted to suit more intricate demands.

In addition to the accomplished work, the tool has the potential for further implementation and upgrade. It can be extended into a test suite, or test framework, that checks and compares several algorithms, when applied onto the same system, so that it can produce detailed statistics regarding the quality and efficiency of POR-algorithms.

Moreover, the tool can be provided with a graphical user interface (GUI) that allows developers to manage the available algorithms and example programs. The visualization can also be applied to the generated transition systems to represent them using state graphs, and to visualize the execution of algorithms and allow step-by-step inspection (debugger view) e.g. marking visited states with different colors, highlighting enabled transitions etc.

Finally, as any other proposed work, this thesis is also subject to praise and critique. The suggested solution to the observed problem may carry some limitation in respect to the complexity of the systems it simulates in its current state. On the other hand, it surely offers a great value of contribution in this field of study; and provides a foundation, to which many useful ideas can be added.

Bibliography

[AW04] Attyia, Hagit; Welch, Jennifer: *Distributed Computing : Fundamentals, Simulations, and Advanced Topics*.volume 19. John Wiley & Sons, s.l., 2004.

[BK08] Baier, Christel; Katoen, Joost-Pieter: *Principles of Model Checking : Representation and Mind Series*. The MIT Press, s.l., 2008, pp.1-16.

[Boe89] Boehm, Barry: *Software Risk Management*. IEEE Press Piscataway, NJ, USA, 1989.

[Cla08] Clarke, Edmund M, Jr.: *The Birth of Model Checking*. In: 25 Years of Model Checking, Springer Science & Business Media, 2008, pp.1-26.

[CGP99] Clarke, Edmund M, Jr.; Grumberg, Orna; Peled, Doron A: *Model Checking*. MIT Press Cambridge, MA, USA, 1999.

[CE81] Clarke, Edmund M.; Emerson, E. Allen: *Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic*. In: Proceeding Logic of Programs, Springer Lecture Notes in Computer Science, 131, 1981, pp.52-71.

[FG05] Flanagan, Cormac; Godefroid, Patrice: *Dynamic partial-order reduction for model checking software*. ACM New York, NY, USA, 2005.

[EP02] Eisner, Cindy,; Peled, Doron: *Comparing Symbolic and Explicit Model Checking of a Software System*. In: Proceedings of the 9th International SPIN Workshop on Model Checking of Software. Springer-Verlag London, UK, 2002, pp.230-239.

[Eme08] Emerson, E.Allen: *The Beginning of Model Checking: A Personal Perspective*. In: 25 Years of Model Checking, Springer Science & Business Media, 2008, pp.27-45.

[EP08] Erk, Katrin; Preise, Lutz: *Theoretische Informatik: Eine umfassende Einführung, 3.* Springer Berlin Heidelberg, 2008.

[Ern04] Ernst, Erik: *Separation of Concerns*. Dept. of Computer Science, University of Aarhus, Denmark, 2004.

[FF+10] Frappier, Marc; Fraikin, Benoît et al.: *Comparison of Model Checking Tools for Information Systems*. http://www.dmi.usherb.ca/~frappier/Papers/TR-GRIL-1006-29.pdf, 2010. [Cited: 29.02.2016.].

[God94] Godefroid, Patrice: *Partial-Order Methods for the Verification of Concurrent Systems : An Approach to the State-Explosion Problem.* University of Liege, Computer Science Department, 1994.

[GV08] Grunberg, Orna; Veith, Helmut: *25 Years of Model Checking : History, Achievements, Perspectives*. Springer-Verlag Berlin Heidelberg, 2008.

[Met14] Metzler, Patrick: *Lazy POR for the Verification of Distributed Programs*. Technical University of Darmstadt, 2014.

[Pel93] Peled, Doron: *All from one, one for all : On model checking using representatives*. In: Computer Aided Verification. Springer Berlin Heidelberg, 1993, pp.409-423.

[SBM+13] Saissi, Habib et al.: *Efficient Verification of Distributed Protocols Using Stateful Model Checking*. In: *SRDS*, IEEE, 2013, pp.133-142.

[SA06] Strunk, Elisabeth A.; Aiello, M. Anthony: *A Survey of Tools for Model Checking and Model-Based Development*. http://www.cs.virginia.edu/~eas9d/papers/CS-TR-2006-17.pdf, 2006, [Cited: 29.02.2016.].

[MIT16] The MIT Press: *Model Checking | The MIT Press*. https://mitpress.mit.edu/books/model-checking, 2016, [Cited: 29.02.2016.].

Appendix A

Input Language

This chapter provides details of the input language, including the grammar definition, the tokenizer (Flex) and the parser (Bison), as well as an example of an input file.

A.1 The Grammar

<u>Legend:</u>

Italic = Non-terminal.

“Plain” = Terminal.

Non-terminal : *non-terminal* « terminal »… = Rule.

***Program* =** Start symbol.

***program*:** *declaration_list process_blocks*

***declaration_list*:** "<" *system_name proc_count declarations* ">"

***system_name*:** "identifier" | %empty

***proc_count*:** "number"

declarations: *declarations declaration* | %empty

***declaration*:** "," "local" "identifier" *length init_values*

| "," "const" "identifier" *length init_values*

| "," "identifier" *length init_values*

| "," "identifier" *init_values*

***length*:** "number"

init_values**:** "number"

process_blocks**:** *process_blocks* "[" *block* "]" | %empty

block**:** *statement block* | *statement*

statement**:** *atomic_statement* | *conditional* | *loop*

atomic_statement**:** *assignment*

assignment**:** *asgn_target* ":=" *int_expr*

conditional: "if" *bool_expr then_block else_block*

then_block**:** "then" "{" *block* "}" | %empty

else_block**:** "else" "{" *block* "}" | %empty

loop**:** "while" *bool_expr* "{" *block* "}"

bool_expr**:** *int_expr* "==" *int_expr*

| *int_expr* "<" *int_expr*

| *int_expr* ">" *int_expr*

int_expr**:** *int_expr* "+" *int_expr*

| *int_expr* "-" *int_expr*

| *int_expr* "*" *int_expr*

| *int_expr* "/" *int_expr*

| *int_expr* "%" *int_expr*

| "(" *int_expr* ")"

| *var*

| *array*

| "number"

asgn_target**:** *var* | *array*
var**:** "identifier"
array**:** *var* "[" i*nt_expr* "]"

Figure A.1 Grammar Description

A.2 The Tokens

"=="	*EQ*
"<"	*LT*
">"	*GT*
"-"	*MINUS*
"+"	*PLUS*
""*	*STAR*
"/"	*SLASH*
"%"	*MOD*
"("	*LPAREN*
")"	*RPAREN*
"["	*LBRACKET*
"]"	*RBRACKET*
"{"	*LBRACE*
"}"	*RBRACE*
":="	*ASSIGN*
","	*COMMA*
"local"	*LOCAL*
"const"	*CONST*
"if"	*IF*
"then"	*THEN*
"else"	*ELSE*
"while"	*WHILE*
{int}	*NUMBER*
{id}	*IDENTIFIER*
<<EOF>>	*END*

Figure A.2 The Tokens

A.3 Example Program

Please refer to the grammar definition (Fig.A.1) to understand the syntax of the example.

```
< Branching_Example 2, x 1 2, y 1 2>
[
  if x == 2 then {
    x := 1
  }
]
[
  x := 0
  y := 4
]
```

Lisiting A.3 Example Program Written in the Defined Input Language

A.4 Example Program

Please refer to the grammar definition (Fig.A.1) to understand the syntax of the example.

```
< Branching_Looping 2, local num 1 2, x 1 0, l 2 4, y 2 4>
[
  if l[0] == 4 then {
      l := 24
      if x == 0 then {
         num := l[1]

         if y[0] == 4 then {
            num := 81
            x := 45
         }
      }
      else {
         num := 6
      }
      x := 10
   }
 while x == 10{
   l := l+1
   x := l
   if x > 4 then {
      l := 45
      y[0] := 34

      if x > 4 then {
         y[1] := 1
      }
      else {
         y[1] := 4
      }
   }
 }
 num := x * l + ( x + l )
]
[
   l := 45
   y[0] := 34
]
```

Listing A.4 Example Program Written in the Defined Input Language

Appendix B
Code Segments

This chapter provides code segments from the source code of the tool that are however too large or unsuitable for placing in the text.

B.1 FastPrototyper.cc

```
#include "Control.hh"


int main (int argc, char *argv[])
{
   Control *control = new Control(argc, argv);
   control->run();

   return 0;
}
```

Listing B.1 FastPrototper.cc with the Program’s Main Function

B.2 The Parser Configuration File

```
// Use c++ with version 3.0.2 of Bison
%skeleton "lalr1.cc" /* -*- C++ -*- */
%require "3.0.2"
// Define the name of the Parser class to be generated
%defines
%define parser_class_name {Parser}
```

```
// Tell Bison how to handle tokens i.e. generate them using a constructor
%define api.token.constructor
%define api.value.type variant
%define parse.assert

// Tell Bison what to use.
%code requires
{
# include <string>
class Driver;
}

// The parsing context.
%param { Driver& driver }

%locations
%initial-action
{
  // Initialize the initial location.
  @$.begin.filename = @$.end.filename = &driver.file;
};

%define parse.trace
%define parse.error verbose

// Include the Driver and other classes needed while parsing
// Declare pointer objects to use while parsing. Initialization takes place during the
parsing to avoid segmentation faults.
```

```
%code
{
# include "Driver.hh"
# include "TransitionSys.hh"
# include "SymbolTable.hh"
# include "Program.hh"
Program* program;
TransitionSys* transitionSystem;
SymbolTable* symbolTable;
}
// Error Handling
void yy::Parser::error (const location_type& l, const std::string& m)
{
  driver.error (l, m);
}
```

Listing B.2 Parser Configuration

In the code segment above, three sections are left out, due to their extreme length. These sections are the token section, token types and grammar rules, which are already mentioned in Appendix A (Fig.A.1 and Fig.A.2).

B.3 The Scanner Configuration File

```
%{ /* -*- C++ -*- */
# include <cerrno>
# include <climits>
# include <cstdlib>
# include <string>

# include "Driver.hh"
# include "Parser.hh"
```

```
// Work around an incompatibility in flex
# undef yywrap
# define yywrap() 1

// The location of the current token.
static yy::location loc;
%}
 // options
%option noyywrap nounput batch debug noinput

 // Definitions
id    [a-zA-Z][a-zA-Z_0-9]*
int   [0-9]+
blank [ \t\r]

%{
  // Code run each time a pattern is matched.
  # define YY_USER_ACTION  loc.columns (yyleng);
%}

%%

%{
  // Code run each time yylex is called.
  loc.step ();
%}

 // Rules
{blank}+   loc.step ();
[\n]+      loc.lines (yyleng); loc.step ();

"=="   return yy::Parser::make_EQ(loc);
"<"        return yy::Parser::make_LT(loc);
">"        return yy::Parser::make_GT(loc);

"-"     return yy::Parser::make_MINUS(loc);
"+"     return yy::Parser::make_PLUS(loc);
"*"     return yy::Parser::make_STAR(loc);
"/"     return yy::Parser::make_SLASH(loc);
"%"     return yy::Parser::make_MOD(loc);

"("     return yy::Parser::make_LPAREN(loc);
")"     return yy::Parser::make_RPAREN(loc);
"["     return yy::Parser::make_LBRACKET(loc);
"]"     return yy::Parser::make_RBRACKET(loc);
"{"     return yy::Parser::make_LBRACE(loc);
```

```
"}"     return yy::Parser::make_RBRACE(loc);

":="    return yy::Parser::make_ASSIGN(loc);
","     return yy::Parser::make_COMMA(loc);

"local" return yy::Parser::make_LOCAL(loc);
"const" return yy::Parser::make_CONST(loc);

"if"    return yy::Parser::make_IF(loc);
"then"  return yy::Parser::make_THEN(loc);
"else"  return yy::Parser::make_ELSE(loc);
"while" return yy::Parser::make_WHILE(loc);


{int}      {
  errno = 0;
  long n = strtol (yytext, NULL, 10);
  if (! (INT_MIN <= n && n <= INT_MAX && errno != ERANGE))
    driver.error (loc, "integer is out of range");
  return yy::Parser::make_NUMBER(n, loc);
}


{id}    return yy::Parser::make_IDENTIFIER(yytext, loc);

.       driver.error (loc, "invalid character");
<<EOF>> return yy::Parser::make_END(loc);

%%
```

Listing B.3 Scanner Configuration

B.4 Overloaded Less Operator

```
bool State::operator <(const State& state) const
{
    return !(memory == state.memory);
}
```

Listing B.4 Overloaded *less* Operator in Class State

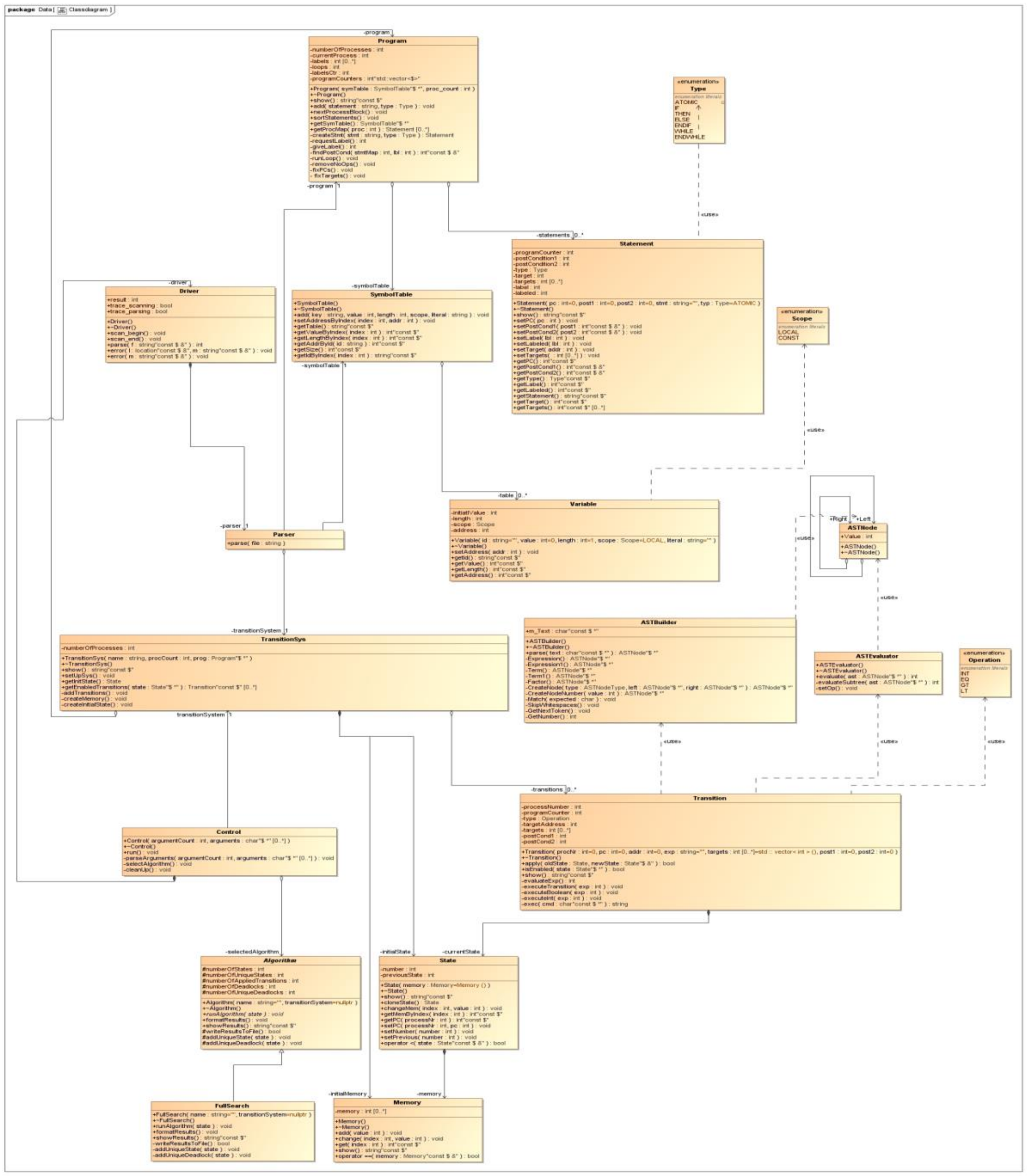
B.5 Overloaded Equal To Operator

```
bool Memory::operator ==(const Memory& mem) const
{

   bool result = true;

   for(unsigned int index = 0; index < memory.size(); ++index)
   {
      if(get(index) != mem.get(index))
      {
         result = false;
         break;
      }
   }

   return result;
}
```

Listing B.5 Overloaded *equal to* Operator in Class Memory

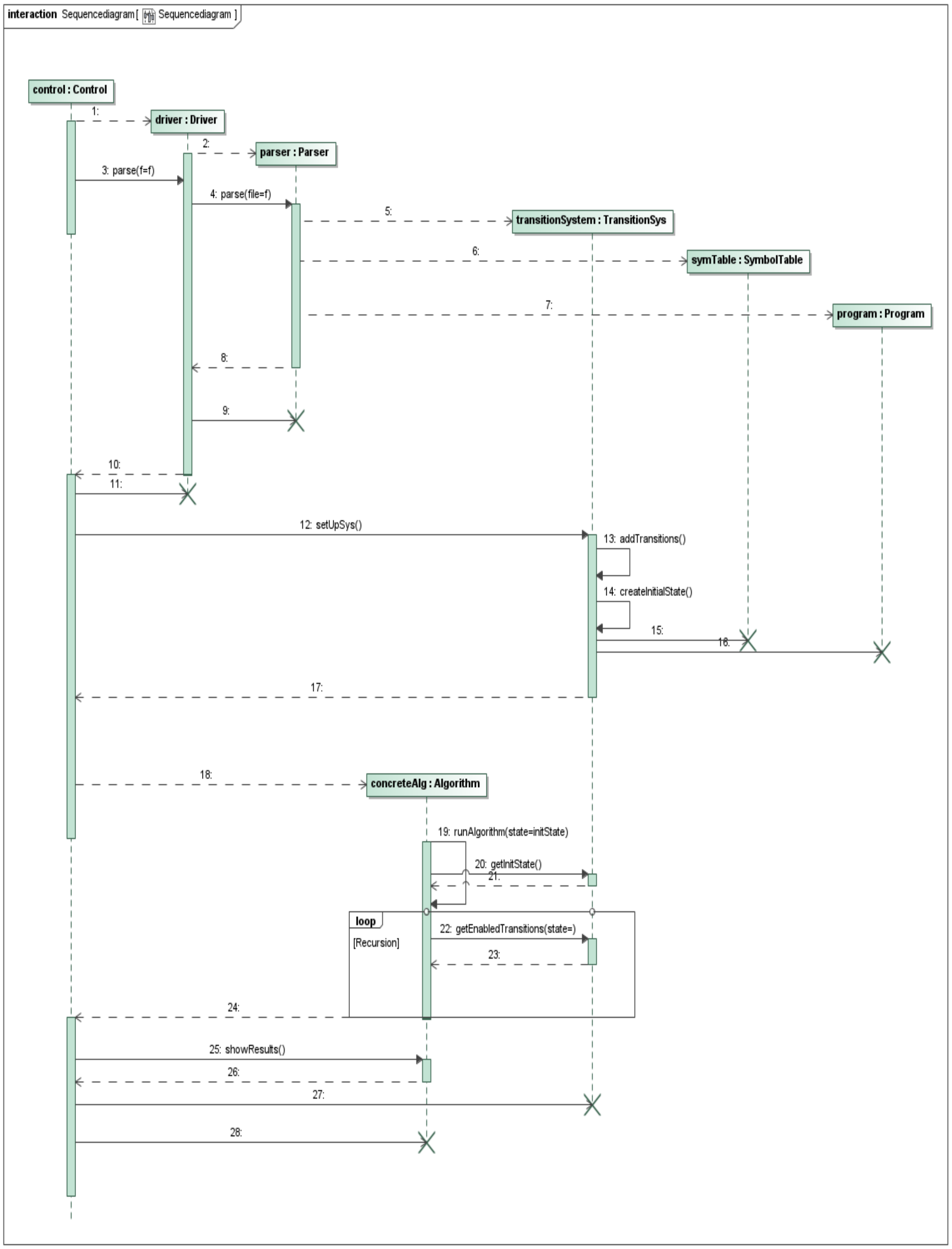
Appendix C
Design Diagrams

C.1 Complete Class Diagram



C.1 Complete Class Diagram

C.1 Complete Sequence Diagram



C.2 Complete Sequence Diagram