AMERICAN UNIVERSITY OF BEIRUT

BEHAVIOURAL MODELING AND ABSTRACTION OF CONCURRENT PROGRAMS

by
CHUKRI ALBERT SOUEIDI

A thesis
submitted in partial fulfillment of the requirements
for the degree of Master of Science
to the Department of Computer Science
of the Faculty of Arts and Science
at the American University of Beirut

Beirut, Lebanon
April 2019

AMERICAN UNIVERSITY OF BEIRUT

BEHAVIOURAL MODELING AND ABSTRACTION OF CONCURRENT PROGRAMS

by

CHUKRI ALBERT SOUEIDI

Approved by: [illegible]

Dr. Paul Attie, Professor — Advisor
Computer Science

[illegible] for Mohamad Jaber

Dr. Mohamad Jaber, Assistant Professor — Member of Committee
Computer Science

[illegible]

Dr. Fadi Zaraket, Associate Professor — Member of Committee
Electrical and Computer Engineering

Date of thesis defense: April 30, 2019

AMERICAN UNIVERSITY OF BEIRUT

THESIS, DISSERTATION, PROJECT RELEASE FORM

Student Name: Soueidi Chuki Albert

Last First Middle

[✓ Master's Thesis] Master's Project Doctoral Dissertation

☐ I authorize the American University of Beirut to: (a) reproduce hard or electronic copies of my thesis, dissertation, or project; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes.

☒ I authorize the American University of Beirut, to: (a) reproduce hard or electronic copies of it; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes after:

One ___ year from the date of submission of my thesis, dissertation or project.

Two ___ years from the date of submission of my thesis , dissertation or project.

Three ✓ years from the date of submission of my thesis , dissertation or project.

[illegible] May 3, 2019

Signature Date

Acknowledgements

"If you wish to make an apple pie from scratch, you must first invent the universe" said Carl Sagan. To solve our problems, that is what mathematicians, logicians, engineers and pioneers of computer science have been doing for the past century. We owe them a universe they created for us that has changed humanity forever and is at the core of our daily lives and future.

I was granted a first-class ticket to this universe by Professor Paul C. Attie. I am forever grateful for his exceptional pedagogical style, immense knowledge, and careful explanations. It was a privilege working with him and being his student. I would also like to thank Dr. Mohamad Jaber and Dr. Fadi Zaraket for following up on my work. I am grateful for Dr. Wassim El Hajj for all his support and for being the perfect academic advisor.

I would also like to thank the Office of Information Technology at AUB for their constant patience and support. Finally, I thank my friends and family, especially Lena, for their motivation, and for everyone who provided me with helpful feedback during my work.

An Abstract of the Thesis of

Chukri Albert Soueidi for Master of Science
Major: Computer Science

Title: Behavioural Modeling and Abstraction of Concurrent Programs

We address the problem of modeling, analyzing, and repairing finite-state and infinite-state concurrent programs. We define a textual notation for concurrent programs and implement it in the Eshmun tool. For finite-state programs, we automatically generate Kripke structures (transition diagrams) from the program text. This structure can then be model checked and repaired using Eshmun facilities. The resulting repair can then be used to guide the designer in fixing the program itself.

For infinite-state programs, we define the notion of a finitely-representable infinite-state Kripke structure, and we provide a semi-automatic method for generating such a structure from an infinite-state concurrent program. This structure models the behavior of the infinite state concurrent program. We label the states of the Kripke structure with state predicates, and the transitions with preconditions P and postconditions Q. Each transition τ then generates a Hoare triple $\{P\}\ \tau\ \{Q\}$ which we verify using the Z3 SMT solver. Hoare triples that are not valid must be repaired. When all triples are valid, we model check to determine if the required properties hold. If the model check fails, more repair is needed. If the model check succeeds, we can semi-automatically extract a correct infinite state concurrent program.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

There is an obvious advantage in verifying that a computer program behaves correctly according to its intended purpose. Today programs are the most complex machines built by humans, and have huge responsibilies for human safety, security, health, and well-being [5]. As the ubiquity and complexity of software systems continue to grow, there is a clear growing demand for formal software verification. This demand is vital in the case of safety-critical systems, for example, where software bugs may possibly lead to catastrophic results. Two main classes of properties, we are mostly interested in verifying, that are usually required in a software: safety properties to assert that nothing bad happens, and liveness properties to assert that something good will eventually happen.

Verifying software correctness often requires to show that a bad state cannot be reached from a good state. That is because the detection of bugs in programs is often easier than proving their absence due to the undecidable nature of the latter problem. For example, if your program has an array of integers of fixed length 10, and you have an integer expression calculating an index onto the array. A good state may be one where the integer expression evaluates into a safe index, while a bad state is a one that evaluates to an out of bound index causing your program to halt.

Verification techniques mainly compromise of three parts: a framework for modeling and describing the behaviour of systems, a specification language for describing what properties i.e., specification need to be verified, and a verficiation method to establish if a model satisfies that specification . The methods vary based on the many different critieria, some of them are: proof-based vs. model-based, degree of automation, full vs. property verification, pre vs. post development [11]. In this paper, we take a model-based property-verification approach to deal with concurrent programs. We seperate between two types of concurrent programs: finite state programs, by which we mean programs that upon execution produce a finite number of states, and infinite state programs by which upon execution produce a state space that is not finite e.g., programs that may contain numeric variables that are being unboundedly incremented and decremented.

A formula φ often written in temporal logic represents a specification i.e., one or many properties. A program model M can be described in terms of a directed graph representing a transition relation R over a concrete set of "states". The model checking task is then to say if $M \models \varphi$. If not, model checkers output counterexamples that witnesses the violation of φ by M. These counterexamples prove the existence of bugs since they identify behaviours that violate the formula being checked. However, there could be many counterexamples, and they may have to be dealt with by making different fixes manually, thus increasing debugging effort. Attie et al [2] deal with all counterexamples at once, by automatically repairing a model with respect to CTL specs. Attie et al present an algorithm for repairing finite state Kripke structures and programs, so that they satisfy a specification written as a formula of the temporal logic CTL (see Section 2.1 and [8]). The repair algorithm is subtractive in nature; it fixes the model by removing transitions and states. Given a finite state Kripke structure M and a CTL formula η, their repair algorithm generates a boolean formula, which is then sent to a SAT solver. A satisfying assignment, if it exists, gives a solution to the repair problem: all states and transition receiving an assignment of "false" must be deleted, and the resulting structure will then satisfy η. Since the repair problem is mapped to a boolean formula, this approach can handle only finite state programs and structures.

When analyzing a shared memory concurrent program, running on multiple processes, we are only interested in the parts of this program that are significant to the synchrnoization between these processes. Hence, we reduce it by abstracting away parts that are only performing local computations. In our model, we only represent parts that we see important and that can result in unwanted behaviour. A challenge that emerges immediately when trying to model infinite state concurrent programs, is how to represent an infinite number of states explicitly. That is why we define the notion of a finitely-representable infinite-state Kripke structure, and we provide a semi-automatic method for generating such a structure from an infinite-state concurrent program. Thus, the objectives of the current work are:

1. Implementing a general text notation to represent finite and infinite state concurrent programs.

2. Devising the algorithms needed to generate the Kripke structures out of finite and infinite state text programs.

3. Extending the model checking algorithm to handle our new finitely-representable infinite-state Kripke structure.

4. Extending the model repair algorithm to a semi-automatic method for repairing infinite state structures and programs.

5. Extracing back the text notation, this is needed after we do model checking and repairing.

The rest of this thesis proceeds as follows: In Chapter 2, we give a preliminary description of the theoretical framework of our work. In Chapter 3, we describe our work on finite state concurrent programs: text based notation, Kripke Structure generation, and program extraction. In Chapter 4, we describe our work on infinite state concurrent programs. In brief, text based notation, our newly introduced finitely-representable infinite-state Kripke structure generation, and how we model check these structures. In Chapter 5, we undertake different case studies to show the how our work can be used. In Chapter 6, we carefully describe our implementation in Eshmun. In Chapter 7, we conclude and discuss our future work.

Chapter 2

Preliminaries

2.1 CTL Syntax and Semantics

Let $\mathcal{AP}$ be a set of atomic propositions, including the constants true and false. We use true, false as "constant" propositions whose interpretation is always the semantic truth values *tt*, *ff*, respectively. The propositional branching-time temporal logic CTL [7, 8] is given by the following grammar:

$$\varphi ::= \mathsf{true} \mid \mathsf{false} \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \mathsf{AX}\varphi \mid \mathsf{EX}\varphi \mid \mathsf{A}[\varphi \mathsf{R} \varphi] \mid \mathsf{E}[\varphi \mathsf{R} \varphi]$$

where $p \in \mathcal{AP}$, and $\mathsf{true}, \mathsf{false}$ are constant propositions with interpretation tt, ff respectively (i.e., "syntactic" true, false respectively).

The semantics of CTL formulae are defined with respect to a Kripke structure.

Definition 1. *A Kripke structure is a tuple $M = (S_0, S, R, L)$ where S is a finite state of states, $S_0 \subseteq S$ is a set of initial states, $R \subseteq S \times S$ is a transition relation, and $L : S \mapsto 2^{\mathcal{AP}}$ is a labeling function that associates each state $s \in S$ with a subset of atomic propositions, namely those that hold in the state. State t is a* sucessor *of state s in M iff $s, t \in R$.*

We assume that a Kripke structure $M = (S_0, S, R, L)$ is total, i.e., $\forall s \in S, \exists s' \in S : (s, s') \in R$. A path in M is a (finite or infinite) sequence of states, $\pi = s_0, s_1, \ldots$ such that $\forall i \geq 0 : (s_i, s_{i+1}) \in R$. A fullpath is an infinite path. A state is reachable iff it lies on a path that starts in an initial state. Without loss of generality, we assume in the sequel that the Kripke structure M that is to be repaired does not contain any unreachable states, i.e., every $s \in S$ is reachable.

Definition 2. *$M, s \models \varphi$ means that formula φ is true in state s of structure M and $M, s \not\models \varphi$ means that formula φ is false in state s of structure M. We define $\models$ inductively as usual:*

1. $M, s \models \mathsf{true}$

2. $M, s \not\models \mathsf{false}$

3. $M, s \models p$ *iff* $p \in L(s)$ *where atomic proposition* $p \in \mathcal{AP}$

4. $M, s \models \neg\varphi$ *iff* $M, s \not\models \varphi$

5. $M, s \models \varphi \wedge \psi$ *iff* $M, s \models \varphi$ *and* $M, s \models \psi$

6. $M, s \models \varphi \vee \psi$ *iff* $M, s \models \varphi$ *or* $M, s \models \psi$

7. $M, s \models \mathsf{AX}\varphi$ *iff for all* t *such that* $(s, t) \in R : (M, t) \models \varphi$

8. $M, s \models \mathsf{EX}\varphi$ *iff there exists* t *such that* $(s, t) \in R$ *and* $(M, t) \models \varphi$

9. $M, s \models \mathsf{A}[\varphi \mathsf{R} \psi]$ *iff for all fullpaths* $\pi = s_0, s_1, \ldots$ *starting from* $s = s_0$*:*
   $\forall k \geq 0 : (\forall j < k : (M, s_j \not\models \varphi)$ *implies* $M, s_k \models \psi$

10. $M, s \models \mathsf{E}[\varphi \mathsf{R} \psi]$ *iff for some fullpath* $\pi = s_0, s_1, \ldots$ *starting from* $s = s_0$*:*
    $\forall k \geq 0 : (\forall j < k : (M, s_j \not\models \varphi)$ *implies* $M, s_k \models \psi$

We use $M \models \varphi$ to abbreviate $M, S_0 \models \varphi$, i.e., for all $s \in S_0$, $M, s \models \varphi$. We introduce the abbreviations $\mathsf{A}[\phi \mathsf{U} \psi]$ for $\neg\mathsf{E}[\neg\varphi \mathsf{R} \neg\psi]$, $\mathsf{E}[\phi \mathsf{U} \psi]$ for $\neg\mathsf{A}[\neg\varphi \mathsf{R} \neg\psi]$, $\mathsf{AF}\varphi$ for $\mathsf{A}[\mathsf{true}\mathsf{U}\varphi]$, $\mathsf{EF}\varphi$ for $\mathsf{E}[\mathsf{true}\mathsf{U}\varphi]$, $\mathsf{AG}\varphi$ for $\mathsf{A}[\mathsf{false}\mathsf{R}\varphi]$, $\mathsf{EG}\varphi$ for $\mathsf{E}[\mathsf{false}\mathsf{R}\varphi]$.

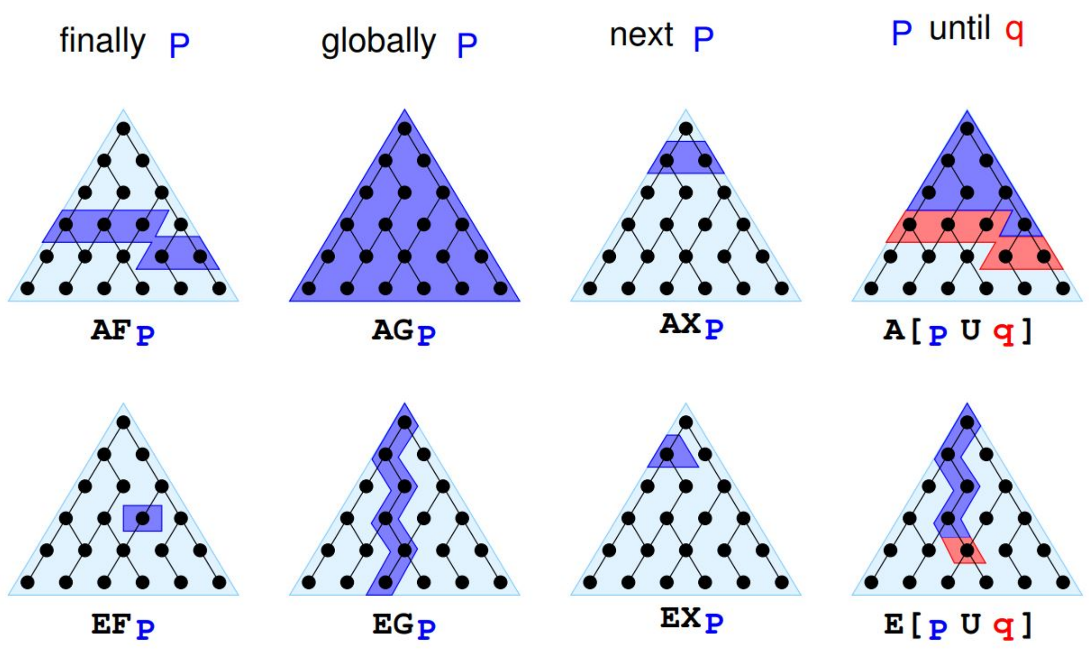


Figure 2.1: CTL semantic visualization taken from CTL Teaching slides [1]

Figure 2.1 shows a graphical visualization of the different CTL modalities. Program execution is represented as computation tree, where each node is a single program state and each edge is a transition to the next possible state. For each modality, it highlights the states that satisfy the CTL formula φ starting from the root node.

2.2 Model of Concurrent Programs

We consider shared-memory concurrent programs $P = (\mathcal{S}t_P, P_1 \parallel \cdots \parallel P_K)$ consisting of K sequential processes $P_1, \ldots, P_K$ running in parallel, together with a set $\mathcal{S}t_P$ of starting global states. For each P_i, there is a finite set $\mathcal{AP}_i$ of *atomic propositions* that are *local to* P_i: only P_i can change the value of atomic propositions in $\mathcal{AP}_i$. Other processes can read, but not change, these values. Local atomic propositions are not shared: $\mathcal{AP}_i \cap \mathcal{AP}_j = \emptyset$ when $i \neq j$. We also admit a set $\mathcal{SH} = \{x_1, \ldots, x_m\}$ of shared variables. These can be read/written by all checks that processes, and have values from domains $D_1 \ldots D_m$ respectively. When domains $D_1 \ldots D_m$ are finite, the program is finite state.

We define the set of all atomic propositions $\mathcal{AP} = \mathcal{AP}_1 \cup \cdots \cup \mathcal{AP}_K$.

Each P_i is a *synchronization skeleton* [8], that is, a directed multigraph where each node is a *local state* of P_i, which is labeled by a unique name s_i, and where each arc is labeled with a *guarded command* [6] $B_i \to A_i$ consisting of a guard B_i and corresponding action A_i. We write such an arc as the tuple $(s_i, B_i \to A_i, s'_i)$, where s_i is the source node and s'_i is the target node. Each node must have at least one outgoing arc, i.e., a synchronization skeleton contains no "dead ends."

The read/write restrictions on atomic propositions are reflected in the syntax of processes: for an arc $(s_i, B_i \to A_i, s'_i)$ of P_i, the guard B_i is a boolean formula over $\mathcal{AP} - \mathcal{AP}_i$, and the action A_i is any piece of terminating pseudocode that updates only the shared variables $\mathcal{SH}$.

Let S_i denote the set of local states of P_i. boolean valuations over $\mathcal{AP}_i$: for $p_i \in \mathcal{AP}_i$, $V_i(s_i)(p_i)$ is the value of atomic proposition p_i in s_i. Hence, as P_i executes transitions and changes its local state, the atomic propositions in $\mathcal{AP}_i$ are updated, since $V_i(s_i) \neq V_i(s'_i)$ in general.

A *global state* is a tuple $(s_1, \ldots, s_K, v_1, \ldots, v_m)$ where s_i is the current local state of P_i and $v_1, \ldots, v_m$ is a list giving the current values of the shared variables in $\mathcal{SH}$.

Let $s = (s_1, \ldots, s_i, \ldots, s_K, v_1, \ldots, v_m)$ be the current global state, and let P_i contain an arc from node s_i to node s'_i labeled with $B_i \to A_i$. If B_i holds in s, then a possible next state is $s' = (s_1, \ldots, s'_i, \ldots, s_K, v'_1, \ldots, v'_m)$ where $v'_1, \ldots, v'_m$ are the new values, respectively, for the shared variables $x_1, \ldots, x_m$ resulting from the execution of action A_i. The set of all (and only) such triples (s, i, s') constitutes the *next-state relation* of program P. As stated above, local atomic propositions $\mathcal{AP}_i$ are implicitly updated, since P_i changed its local state from s_i to s'_i.

The appropriate semantic model for a concurrent program is a *multiprocess Kripke structure*, which is a Kripke structure that has its set $\mathcal{AP}$ of atomic propositions partitioned into $\mathcal{AP}_1 \cup \cdots \cup \mathcal{AP}_K$, and every transition is labeled with the index of a single process, which executes the transition. Only atomic propositions belonging to the executing process can be changed by a transition. Shared variables may also be present. The semantics of a concurrent program $P = (\mathcal{S}t_P, P_1 \parallel \cdots \parallel P_K)$ is then given by its global state transition digram (GSTD): the smallest multiprocess Kripke structure M such that (1) the start states of M are $\mathcal{S}t_P$, and (2) M is closed under the next state relation of P. Effectively, M is obtained by "simulating" all possible executions of P from its start states $\mathcal{S}t_P$. A program satisfies a CTL formula η iff its GSTD does.

For example, Figure 2.2 gives a synchronization skeleton concurrent program consisting of two processes, P_1 and P_2. Each process has three local states, so e.g., P_1 has local states labeled with **N1**, **T1**, and **C1**, respectively. Similarly for P_2. In P_1, the transition from **T1** to **C1** has guard **N2**$\vee$**T2**. There are no shared variables. Figure 2.3 gives a Kripke structure that is generated by the execution of this concurrent program. The transitions of each process are colored in the same color.

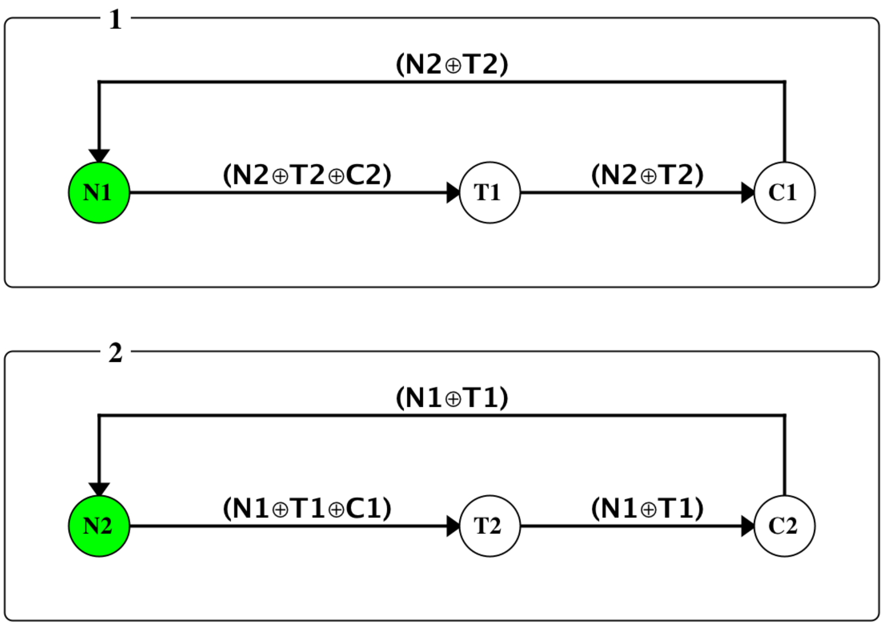


Figure 2.2: Synchronization skeletons for two-process mutual exclusion

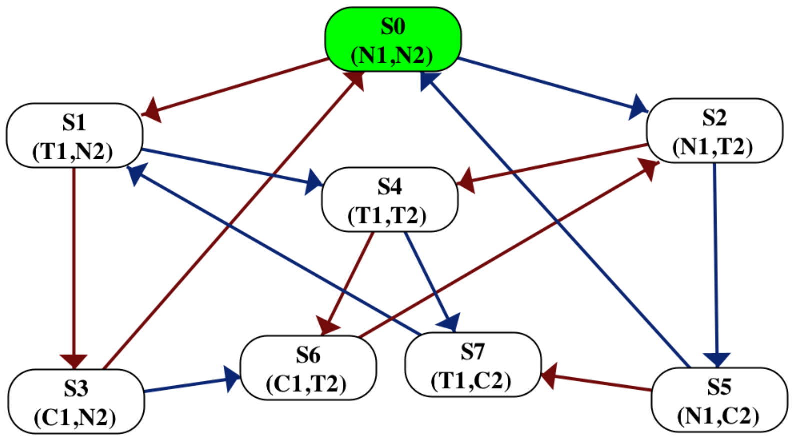


Figure 2.3: Kripke structure for two-process mutual exclusion

2.3 First Order Logic

We use standard first order logic as described in Enderton "A Mathematical Introduction to Logic" [9] Chapter 2, Sections 2.1 and 2.2. We omit the standard definitions of validity and truth in a structure and state, for which we use the following notations:

$s \models \varphi$	means that φ is true in s
$\dashv \ \varphi$	means that φ is satisfiable i.e., $\exists s : s \models \varphi$

In all cases, the underlying first order structure is the standard model of arithmetic.

Chapter 3

Finite-State Concurrent Programs

A finite state concurrent program is a program that produces a finite number of different states upon execution. Our work on finite state allows us to describe such programs in a text-based format. We deal with concurrent programs by keeping the details only relevant to synchronization between different processess. Specifying a program as text is often more simple that creating a state-transition diagram for it. The text program can then be imported into Eshmun as a Kripke structure where model checking and repair will be performed. After interacting with the model, the user can then export the system again into a text program, this facilitates and speeds up in general the process of model-checking and saves a lot of time for the user.

3.1 Text-based Notation

Our text-based program is constructed by specifying synchronization skeletons for its processes. The synchronization skeleton is an abstraction of the actual program where detail irrelevant to synchronization is suppressed [8]. In the following sections, we shall see how these programs can be constructed. The model of a program P is discussed in the preliminaries section 2.

Our program consists of the following declarations:

1. initial states: a boolean formula over AP and shared variables,
2. the domains of shared variables (which must be finite), and
3. the set of processes.

For each process, we define a collection of actions such that each action contains of the following definitions:

1. local guard: a boolean formula over AP_i (atomic propositions of P_i).
2. global guard: a boolean formula over $AP - AP_i$ and shared variables.

3. local effect: assignment statements to update AP_i.

4. global effect: assignment statements to update shared variables.

3.2 Kripke Structure Generation

To generate a Kripke structure that represents the behavior of a finite state concurrent program, we simply simulate all possible behaviors of the program until no new states and transitions are generated. The text based notation, described in Section 3.1, allows us to define for each process P_i a set of arcs i.e., actions. We need to simulate every action on the set of all possible states, the state space is generated by doing a cartesian product of n sets of labels, each set belongs to a process and n is the number of processes. If there are shared variables, then we add to the product all possible values of shared variables. Then, for each action a belonging to P_i, we check if the local guard and global guard hold in every state s of the state space. If the guard holds, the state s is enabled, thus we apply the effects to get the resulting state t.

Algorithm 1: Generate Finite State Kripke Structure

```
let $K$ be the currently generated structure
$K.states$ = states in $K$, starting with initial state
$K.steps$ = transitions in $K$
$Actions$ = set of all parsed program actions

while new states or transitions are generated do
    foreach $a$ in $Actions$ do
        $E :=$ states that satisfy guards of $a$
        foreach $s$ in $E$ do
            $t :=$ simulate action $a$ on $s$ to get a new state $t$
            $K.states += t$
            $K.steps += (s, t)$
        end
    end
end
```

We implement two variations on this basic idea, one produces conrete Kripke structures and the second produces structures with shared variable abstractions.

3.2.1 Concrete Kripke Structure

The most concrete structure possible is obtained when all the global states are differentiated, i.e., if two states differ in even a single shared variable or atomic proposition, then they are rendered as different states in the Kripke stucture.

This gives the most faithful representation of the program's behavior, but also the largest.

3.2.2 With Shared Variable Abstractions

A shorter representation of the program's behavior can be obtained by using abstraction, i.e., by considering some global states to be equivalent. Specifically, we consider some global states that differ in the value of shared variables to be equivalent. These states are indicated by user annotations, which have the form of setting a shared varaible x to `null`. This means that the current value of the shared variable should be "forgotten", so that subsequent global states that differ only in the value of x are considered equivalent, and are merged into a single global state.

3.3 Program Extraction from Kripke

We implemented the program extraction such that: Given a multiprocess Kripke structure M, we can extract a concurrent program by projecting onto the individual process indices [8]. If M contains a transition from $s = (s_1, \ldots, s_i, \ldots, s_K, v_1, \ldots, v_m)$ to $s' = (s_1, \ldots, s'_i, \ \ldots, s_K, v'_1, \ldots, v'_m)$, then we can project this onto P_i as the arc $(s_i, B_i \to A_i, s'_i)$, where B_i checks that the current global state is $(s_1, \ldots, s_i, \ \ldots, s_K, v_1, \ldots, v_m)$, and A_i is the multiple assignment $x_1, \ldots, x_m := v'_1, \ldots, v'_m$, i.e., it assigns v'_ℓ to x_ℓ, $\ell = 1, \ldots, m$.

Our implementation is formally described by Attie and Emerson [3] (see Section 2.6 Synthesis of Programs), after extracting all transitions we group them by families that can collapse into one single action. This works for finite-state Kripke structures. For infinite-state structures, we will implement a semi-automatic method, i.e., based on interaction with the user.

3.4 CTL Model Checking

To verify properties of Kripke structures, we use the CTL model checking algorithm of Clarke, Emerson, and Sistla [4, 7]. Given a Kripke strcuture $M = (S_0, S, R, L)$ and CTL formula φ, the model checking algorithm determines whether $M, S_0 \models \varphi$, in time linear in the size of M and the size of φ. If the

Chapter 4

Infinite-State Concurrent Programs

Infinite-state programs are programs that produce an infinte state space upon execution. Examples of these programs are such with unbounded arithmetic expressions. The transition diagram i.e., concrete Kripke strucutre of an infinite-state concurrent program is, in general, infinite.

4.1 Finitely Representable Kripke Structure

We seek a finite representation of this infinite Kripke structure, which will call a *finitely-representable Kripke structure (FRKS)*. An FRKS is, of necessity, an abstraction of the corresponding infinite concrete Kripke structure.

To construct an FRKS, we extend finite-state Kripke structures with variables and statements, as follows:

1. Each state s is labeled with a state *predicate*, given by a formula of first order logic

2. Each transition is labeled with

    (a) a block of statements, each of which is either an assignment or an if statement, i.e., straight-line code.

    (b) a predondition

    (c) a postcondition

s now represents the set of concrete states that satisfy its state predicate $s.p$. Transition τ from s generates Hoare triple verification condition usign strongest postconditions sp (see Section 4.4.2). Figure 4.1, shows a visual representation of a transition τ from state s to state t.

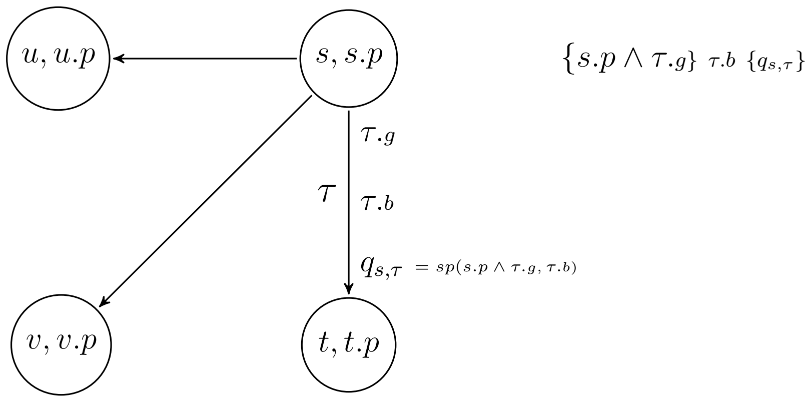

$u, u.p$
$s, s.p$
$\{s.p \wedge \tau.g\}\ \tau.b\ \{q_{s,\tau}\}$
$\tau.g$
τ
$\tau.b$
$q_{s,\tau} = sp(s.p \wedge \tau.g, \tau.b)$
$v, v.p$
$t, t.p$

Figure 4.1: Transition in finitely-representable Kripke structure (FRKS)

τ : $\tau.g \rightarrow \tau.b$(transition)	$\tau.g$: precondition (guard) of τ
$s.p$: state predicate of s	$\tau.b$: body of τ
$t.p$: state predicate of t	$q_{s,\tau}$: strongest postcondition for τ from s

4.2 Text-based Notation

An infinite-sate program is constructed similarly to the finite-state program with two additions: optional local variables for each process and first-order logic state predicates. In addition, the domains of all variables can now be infinite. For each process, we define a collection of actions such that each action contains of the following definitions:

1. local guard: a first-order formula over AP_i (atomic propositions of P_i) and process local variables of P_i.
2. global guard: a first-order formula over $AP - AP_i$, shared variables, and all local variables.
3. local effect: boolean value assignments to update AP_i. Straight-line code blocks of assignments, if and if-else statements over process local variables.
4. global effect: straight-line code blocks of assignments, if and if-else statements over shared variables and process local variables.

4.3 Kripke Structure Generation

We implemented the generation algorithm that converts the text-based program to a FRKS. The algorithm starts from the declared initial state and uses forward reasoning to generate new states by calculating the strongest postconditions of states on actions. The algorithm runs until no new states and transitions are generated. The key idea is that the resulting structure represents all possbile behaviors of the program faithfully.

4.3.1 Hoare Logic and Strongest Postconditions

Our algorithm starts from an initial state labelled with a predicate and will proceed to produce new states and transitions between these states. Each transition in our structure is defined by a Hoare triple of the form $\{P\}S\{Q\}$. P and Q are predicates representing the precondition and postcondition i.e., the predicate of the starting state and the resulting state respectively. S is a straight line code statement. The conditional correctness of a triple $\{P\}S\{Q\}$ (assuming termination [1]) means that if we start in a state where P is true and execute S, it terminates in a state where Q is true. For example, starting from a state labelled with predicate P, a program perfroming an assignment statement $x := e$, will generate a new resulting state as follow:

$$\{P\}\ x := e\ \{\exists x_0 : (P[x_0\backslash x] \wedge x = e[x_0\backslash x]\}$$

We can use predicate transformer semantics to produce these new states by deducing the predicate Q_t that represents the resulting state. We do this by finding the strongest postcondition $sp(P_s, S) = Q_t$. The function sp takes a precondition P and a statement S and returns a postcondition. sp is the strongest such postcondition i.e., take any P, Q such that $\{P\}S\{Q\}$, then $sp(P, S) \Rightarrow Q$.

In our implementation an action can have effects which are statments of the form of:

- a skip operation
- a single assignment statement
- an if-then statement
- an if-else statement
- a block of sequential statements

In our notation, a transition $\tau : \tau.g\!\rightarrow \tau.b$ will have $\tau.b$ as effect i.e., statement.

[1] Termination is guaranteed since s is straight-line code

Definition 3. *The strongest postcondition for each statement type is as follows:*

$$\begin{aligned}
&sp(P, skip) = P \\
&sp(P, x := e) = \exists x_0 : (P[x_0 \backslash x] \wedge x = e[x_0 \backslash x]) \\
&sp(P, \textit{if } B \textit{ then } S) = (B \Rightarrow sp(P, S)) \;\wedge\; (\neg B \Rightarrow P) \\
&sp(P, \textit{if } B \textit{ then } S_1 \textit{ else } S_2) = (B \Rightarrow sp(P, S_1)) \;\wedge\; (\neg B \Rightarrow sp(P, S_2)) \\
&sp(P, S_1; S_2) = sp(sp(P, S_1), S_2)
\end{aligned}$$

The predicate $P[e \backslash x]$ is equivalent to P where every free occurrence of x is replaced by e. x denotes a variable and e is an expression.

4.3.2 Kripke Structure Generation Algorithm

The Kripke stucture generation algorithm will aim to generate a Kripke structure for an infinite program with a finite number of nodes. In general, we will not be able to do that automatically for all cases. User intervention may be needed and we will discuss this after decribing the main ideas of the algorithm.

At the beginning, the Kripke structure contains one initial state which is represented by a formula of first order logic declared in the program text. We start simulating all actions on the initial state, and when the algorithm generates new states, we need to simulate all actions again on exisiting and new states.

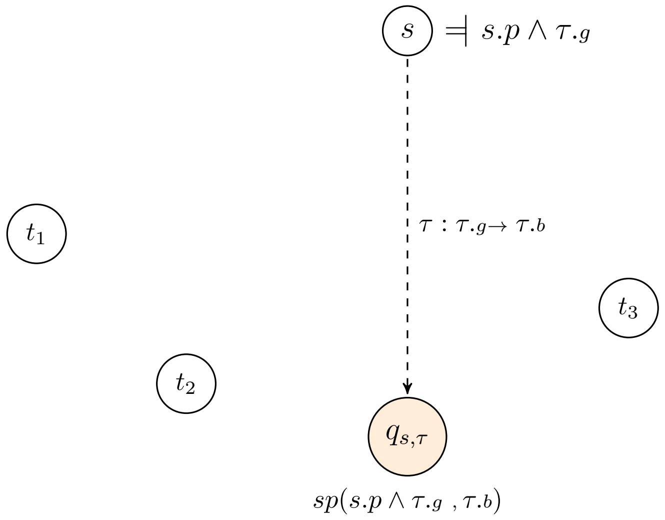

s
$\dashv s.p \wedge \tau.g$
$\tau : \tau.g \to \tau.b$
t_1
t_2
t_3
$q_{s,\tau}$
$sp(s.p \wedge \tau.g \;, \tau.b)$

Figure 4.2: Generating temporary state $q_{s,\tau}$. Step 1

Step 1: For each state s, on each action $\tau : \tau.g \to \tau.b$, we check if $\dashv s.p \wedge \tau_g$, we calculate the postcondition $q_{s,\tau} := sp((s.p \wedge \tau.g), \tau.b)$ by using the τ_b as the body and the conjunction of guard and state predicate as precondition[2]. We

[2]By $\dashv \varphi$ we mean that φ is satisfiable i.e., $\exists s : s \models \varphi$

then create a new temporary state with a state predicate $q_{s,\tau}$ with a temporary transition from s to $q_{s,\tau}$.

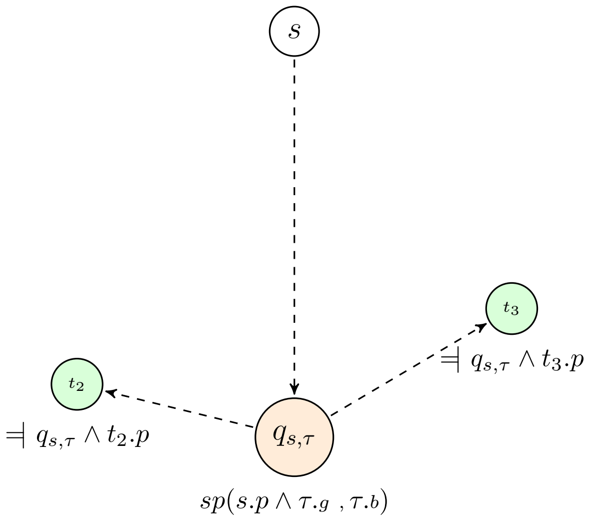

s
t_1
t_3
$\dashv q_{s,\tau} \wedge t_3.p$
t_2
$\dashv q_{s,\tau} \wedge t_2.p$
$q_{s,\tau}$
$sp(s.p \wedge \tau.g\ , \tau.b)$

Figure 4.3: Finding states that intersect with $q_{s,\tau}$. Step 2

Step 2: From the temporary state $q_{s,\tau}$, we find all states $(t_1...t_n)$ in our structure satisfying $\dashv q_{s,\tau} \wedge t.p$. We add a temporary transitions from $q_{s,\tau}$ to those $(t_1...t_n)$, labelled in green in Figure 4.3.

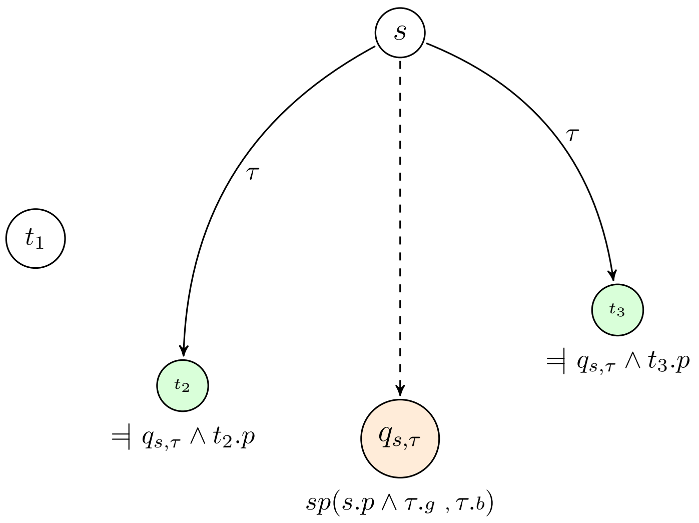
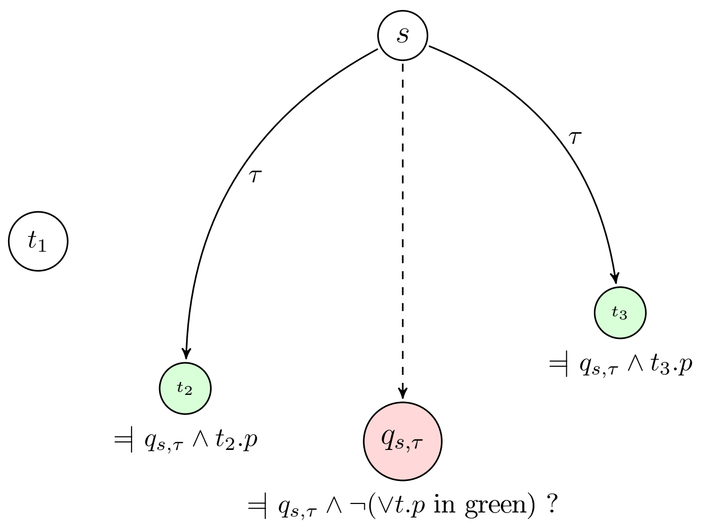

s
τ
τ
t_1
t_3
$\dashv q_{s,\tau} \wedge t_3.p$
t_2
$\dashv q_{s,\tau} \wedge t_2.p$
$q_{s,\tau}$
$sp(s.p \wedge \tau.g\ , \tau.b)$

Figure 4.4: Adding transitions from s. Step 3

Step 3: For all states labeled in green from Step 2, we add a transition from s. These states will be the successors of s. If there are no states satisfying $\dashv q_{s,\tau} \wedge t.p$, we will add a new state u with $u.p = Q$ and add a transition labelled

τ from s to u. If u is added, we are done and we move to the next state starting from Step 1.

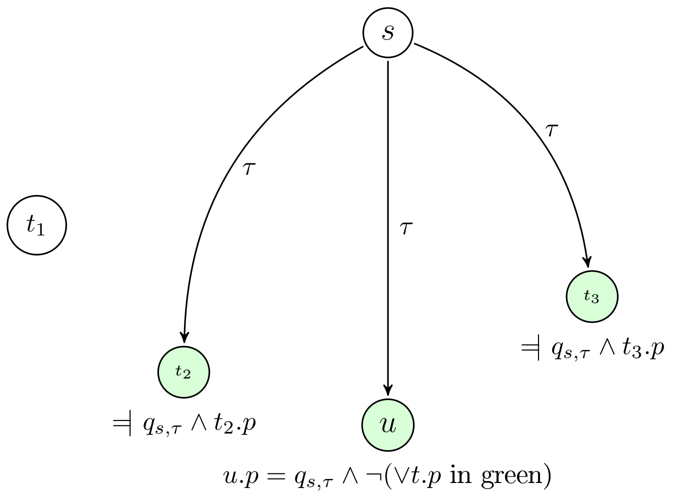


s
τ
τ
t_1
t_3
$\dashv q_{s,\tau} \wedge t_3.p$
t_2
$\dashv q_{s,\tau} \wedge t_2.p$
$q_{s,\tau}$
$\dashv q_{s,\tau} \wedge \neg(\vee t.p$ in green) ?

Figure 4.5: Check if $q_{s,\tau}$ is contained. Step 4

Step 4: If we have successors for s at Step 3, we need to check that $q_{s,\tau}$ is fully contained by these states. We do this by checking if $\dashv q_{s,\tau} \wedge \neg(\vee t.p$ in green). If no then $q_{s,\tau}$ is contained and we delete Q and move to next state at Step 1. If yes, we move to Step 5.

Step 5: If $q_{s,\tau}$ is not contained in all successros of s in Step 4. We then create a new state u with a state predicate $u.p = q_{s,\tau} \wedge \neg(\vee t.p$ in green) and transition from s to u labelled with τ.



s
τ
τ
τ
t_1
t_3
$\dashv q_{s,\tau} \wedge t_3.p$
t_2
$\dashv q_{s,\tau} \wedge t_2.p$
u
$u.p = q_{s,\tau} \wedge \neg(\vee t.p$ in green)

Figure 4.6: Create a new state. Step 5

We now give the pseudocode for the generation algorithm described above.

Algorithm 2: Generate Finitely Representable Kripke Structure

```
let $K$ be the currently generated structure
$K.states$ = states currently in $K$ // Initially, $K$ consists of a single
  initial state
$K.steps$ = transitions currently in $K$

for $s$ in $K.states$ do
    $s.succ$ = the states in $K$ that are reachable from $s$ by a single
      transition, i.e., $(s, t)$ in $K.steps$
    $s.newsucc$ = successors to be added in the current iteration of the
      main loop
    $s.pred$ = the state predicate of $s$
end

//Now, the main loop
for $a$ in $Actions$ do
    $B := a.localGuard \wedge a.globalGuard$
    for $s$ in $K.states$ do
        $s.newsucc := \phi$
        if $\dashv B \wedge s.pred$ then
            $Q := strongestPostCondition((B \wedge s.pred), a.Action)$
            for $t$ in $K.states$ do
                if $\dashv q_{s,\tau} \wedge t.pred$ then
                    $s.newsucc += t$
            end
            if $s.newsucc = \phi$ then
                $s.newsucc :=$ new $t$ where $t.pred = Q$
            else if $\dashv q_{s,\tau} \wedge \neg(\vee t \text{ in } s.succ : t.pred)$ then
                $s.newsucc +=$ new $t$ such that $t.pred = q_{s,\tau} \wedge \neg(\vee t$ in
                  $s.succ : t.pred)$
            end
            for $t$ in $s.newsucc$ do
                $K.states += t$
                $K.steps += (s, t)$
            end
    end
end
```

Our algorithm is not guaranteed to terminate at all times. That is why we introduced, the Interactive mode where we can configure the number of iterations to be perfromed before asking for user interaction. Then the user can intervene at the stage of adding new states, by weakening the new state predicate carefully. This will lead to an abstract representation, and will allow the algorithm to terminate. In other cases, mentioned in Chapter 5, our algorithm terminates due to the bounded domain or to the bounded value a shared variable can have.

4.4 Model Checking

We model check an FRKS on two steps. The first step checks that every Hoare triple is valid using Z3. The second step then treats the FRKS as finite state, i.e., only uses the atomic propositions, and model checks it using the standard CTL model checking algorithm, which is implemented in Eshmun.

4.4.1 Weakest Preconditions

To reason about our program correctness, we use Hoare logic to verify the Kripke structure transitions.To build a valid deduction, we use predicate semantic transformers namely weakest-preconditions to reduce the problem of verifying a triple to a problem of proving a first-order formula. We can then use a SMT solver to prove the formula's validity.

If $\{P\}\ S\ \{Q\}$ and for all P' such that $\{P\}\ S\ \{Q\}$, $P \Rightarrow P'$, then P' is the weakest precondition $wp(S, Q)$ of S with respect to Q.

Definition 4. *We define a function yielding the weakest precondition with respect to some postcondition Q for statement S as follows:*

$$wp(skip, Q) = Q$$
$$wp(x := e, Q) = Q[e\backslash x]$$
$$wp(S_1; S_2, Q) = wp(S_1, wp(S_2, Q))$$
$$wp(\mathit{if}\ B\ \mathit{then}\ S, Q) = (B \Rightarrow wp(S, Q))\ \wedge\ (\neg B \Rightarrow Q)$$
$$wp(\mathit{if}\ B\ \mathit{then}\ S_1\ \mathit{else}\ S_2, Q) = (B \Rightarrow wp(S_1, Q))\ \wedge\ (\neg B \Rightarrow wp(S_2, Q))$$

The predicate $Q[e\backslash x]$ is equivalent to Q where every free occurrence of x is replaced by e. x denotes a variable and e is an expression.

4.4.2 Hoare Triples Check

The first step in model checking the introduced finitely-representable Kripke Structure (FRKS) is Hoare Triples Check. In FRKS, each state is an abstraction of concrete states represented by a first-order logic predicate which labels the state. Recall, from Section 4.1, that we added for each transition: a precondition P and postcondition Q predicate, and a statement block S. An empty predicate is interpreted as $true$, and an empty statment is iterpreted as $skip$.

Figure 4.7, shows how to perform a Hoare triple check on transition τ from state s to state t. All transitions labelled with τ from s should be considered for

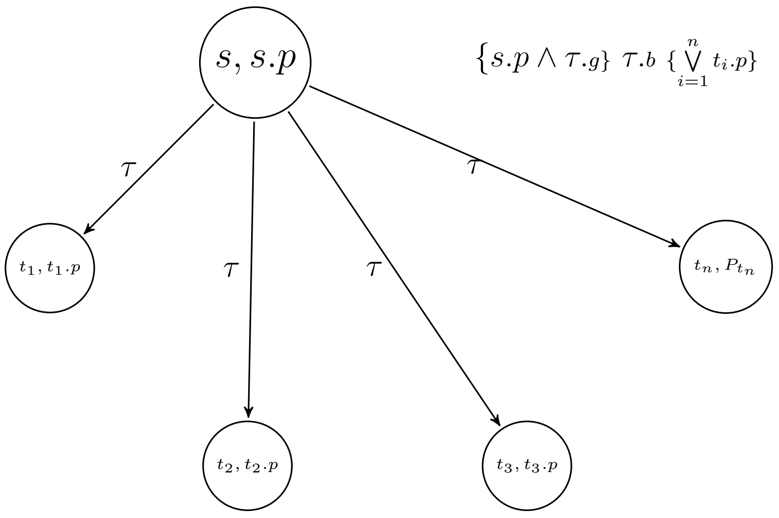


Figure 4.7: Hoare Triple Check

each transition from s. We get the successors of s by τ and do a disjunction of their states predicates. This disjunction will be our postcondition.

Definition 5. *(Hoare Triples Check) For each transition* $(s, t)\ :\ \tau$

1. $s.p \Rightarrow \tau.g$ *is satisfiable, else warning*
2. $\{s.p \wedge \tau.g\}\ \tau.b\ \{\bigvee_{i=1}^{n} t_i.p\}$ *triple is valid.*
   - $t_1 ... t_n$ *are all successors of* s *by transition* $\tau : \tau.g \rightarrow \tau.b$
   - $s.p \wedge \tau.g \Rightarrow wp(\tau.b, \bigvee_{i=1}^{n} t_i.p)$ *(weakest precondition)*
3. $q_{s,\tau} \Rightarrow \bigvee_{i=1}^{n} t_i.p$ *is valid.*

Each check from above is sent to Z3 SMT solver and the transition is labeled valid if and only if Step 2 and Step 3 are valid.

Chapter 5

Case Studies

5.1 Finite State

5.1.1 2-process Mutex with a shared variable

In concurrent programming, access to a shared resource by mulitple process can lead to a erroneous behaviour, thus exclusive access of each process to the shared resource is required. This resource can be a shared memory variable, a peripheral device, or a network connection. We call the block of code accessing this shared resource, the critical section. Mutual exclusion (Mutex) means that at any point of time only one process should be present inside its critical section. Since our programs are abstractions of real programs, we intend to only represent states that are important for the synchronization of processes. For mutex programs we define 3 main sections for process P_i, these will be the labels for a process local state:

- N_i : a process is in a Neutral state performing local computations
- T_i : a process is Trying state, requested access to critical section
- C_i : a process is inside the Critical section

Listing 5.1 shows a text-based two-process mutual exclusion example program. We added a shared memory variable x with a domain $[1, 2]$ that can be read by both processes, this variable will help decide which process enters the critical section if both both processes are at a Trying state.

Concrete Kripke Structure

Figure 5.1 shows the concrete structure generated from the mutex program listed in page 22. Process 1 in blue and Process 2 in red.

```
program  {
  initial   : N1 & N2
  sharedvariables   :  x  : {  1, 2 }
  process  1 {
    action {
      l_grd : N1
      g_grd : N2 | C2
      l_eff : N1,T1 := ff, tt
      g_eff : x := null
    }
    action {
      l_grd : N1
      g_grd : T2
      l_eff : N1,T1 := ff, tt
      g_eff : x := 2
    }
    action {
      l_grd : T1
      g_grd : N2 | (T2 & x=2 )
      l_eff : T1,C1 := ff, tt
      g_eff : x := null
    }
    action {
      l_grd : C1
      g_grd : N2 | T2
      l_eff : C1,N1 := ff, tt
      g_eff : x := null
    }
  }
  process  2 {
    action {
      l_grd : N2
      g_grd : N1 | C1
      l_eff : N2,T2 := ff, tt
      g_eff : x := null
    }
    action {
      l_grd : N2
      g_grd : T1
      l_eff : N2,T2 := ff, tt
      g_eff : x := 1
    }
    action {
      l_grd : T2
      g_grd : N1 | (T1 & x=1 )
      l_eff : T2,C2 := ff, tt
      g_eff : x := null
    }
    action {
      l_grd : C2
      g_grd : N1 | T1
      l_eff : C2,N2 := ff, tt
      g_eff : x := null
    }
 }
 specifications : { AG( !( C1 & C2 ) ) }
}
```

Listing 5.1: Text-based two-process mutual exclusion program

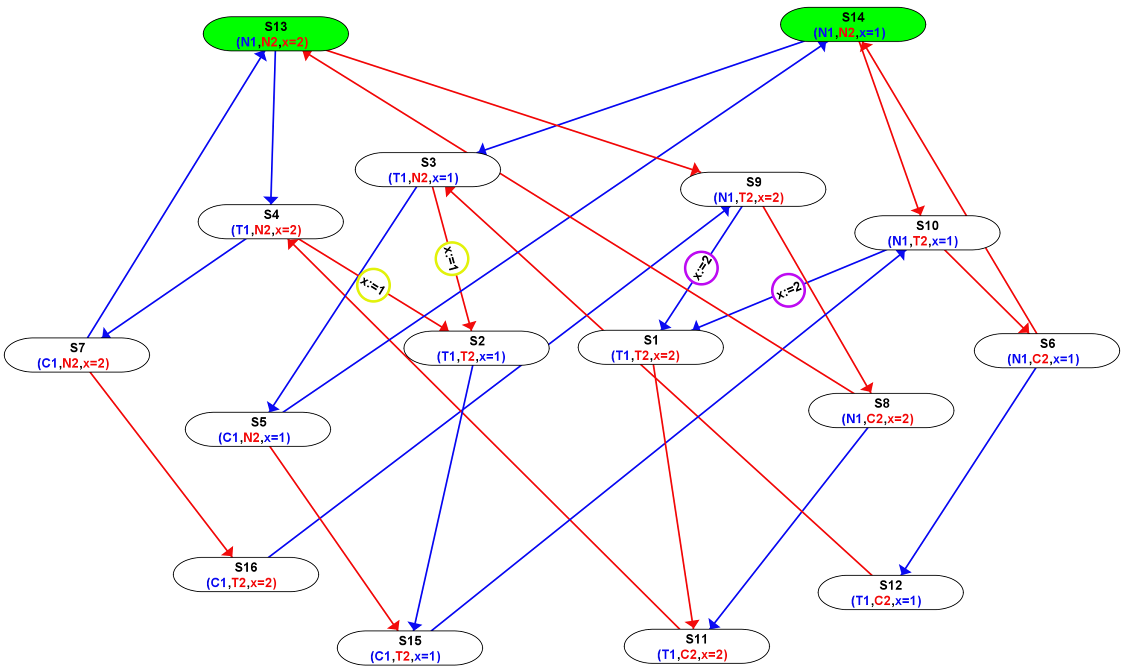


Figure 5.1: Two-process mutex

With Shared Variable Abstraction

Figure 5.2 shows the structure generated with shared variable abstraction, from the mutex program listed in page 22. Process 1 in blue and Process 2 in red. We can see how the number of states was reduced from 16 states to 9 states by collapsing together states where the value of the shared variable is not important, i.e., it does not affect future execution (indicated by annotations in the text program).

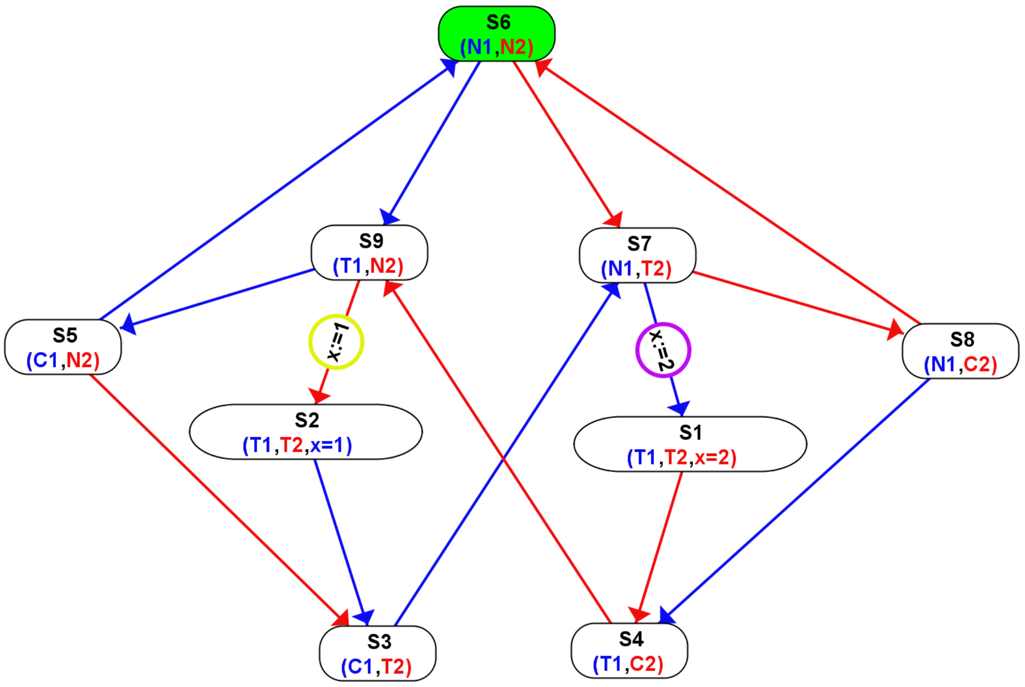


Figure 5.2: Two-process mutex with shared variable abstraction

5.1.2 Peterson's Algorithm

Listing 5.2 shows a text representation of Peterson's algorithm, a concurrent programming algorithm for mutual exclusion. The local states for each process are N_i, T_i, and C_i similar to the example at Section 5.1.1, an new label SP_i was introduced and indicates that P_i is in spinning mode i.e., busy-waiting. This program uses 3 shared memory variables, f_1 (a flag of Process 1), f_2 (a flag of Process 1), and t. $f_i = 1$ indicates that the P_i wants to enter the critical section. Entrance to the critical section is granted for process P_0 if P_1 does not want to enter its critical section or if P_1 has given priority to P_0 by setting turn to 0

Figure 5.3 shows the concrete structure generated from the Peterson program text program above.

```
program  {
 initial   : N1 & N2 & (f1=0) & (f2=0) & (t=0)
 sharedvariables   :  f1 : {  0, 1 } , f2 : {  0, 1 } , t : {  0, 1 }
 process  1 {
  action {
   l_grd : N1
   g_grd : true
   l_eff : N1,T1 := ff, tt
   g_eff : f1 := 1
  }
  action {
   l_grd : T1
   g_grd : true
   l_eff : T1,SP1 := ff, tt
   g_eff : t := 1
  }
  action {
   l_grd : SP1
   g_grd : (f2=0 | t=0)
   l_eff : SP1,C1 := ff, tt
   g_eff : skip
  }
  action {
   l_grd : C1
   g_grd : true
   l_eff : C1,N1 := ff, tt
   g_eff : f1 := 0
  } }
 process  2 {
  action {
   l_grd : N2
   g_grd : true
   l_eff : N2,T2 := ff, tt
   g_eff : f2 := 1
  }
  action {
   l_grd : T2
   g_grd : true
   l_eff : T2,SP2 := ff, tt
   g_eff : t := 0
  }
  action {
   l_grd : SP2
   g_grd : (f1=0 | t=1)
   l_eff : SP2,C2 := ff, tt
   g_eff : skip
  }
  action {
   l_grd : C2
   g_grd : true
   l_eff : C2,N2 := ff, tt
   g_eff : f2 := 0
  } }
 specifications : { ( AG( !( C1 & C2 ) ) ) & ( AG( T1 => ( AF( C1 ) ) ) )
    & ( AG( T2 => ( AF( C2 ) ) ) ) }
}
```
Listing 5.2: Peterson's algorithm two-process mutual exclusion

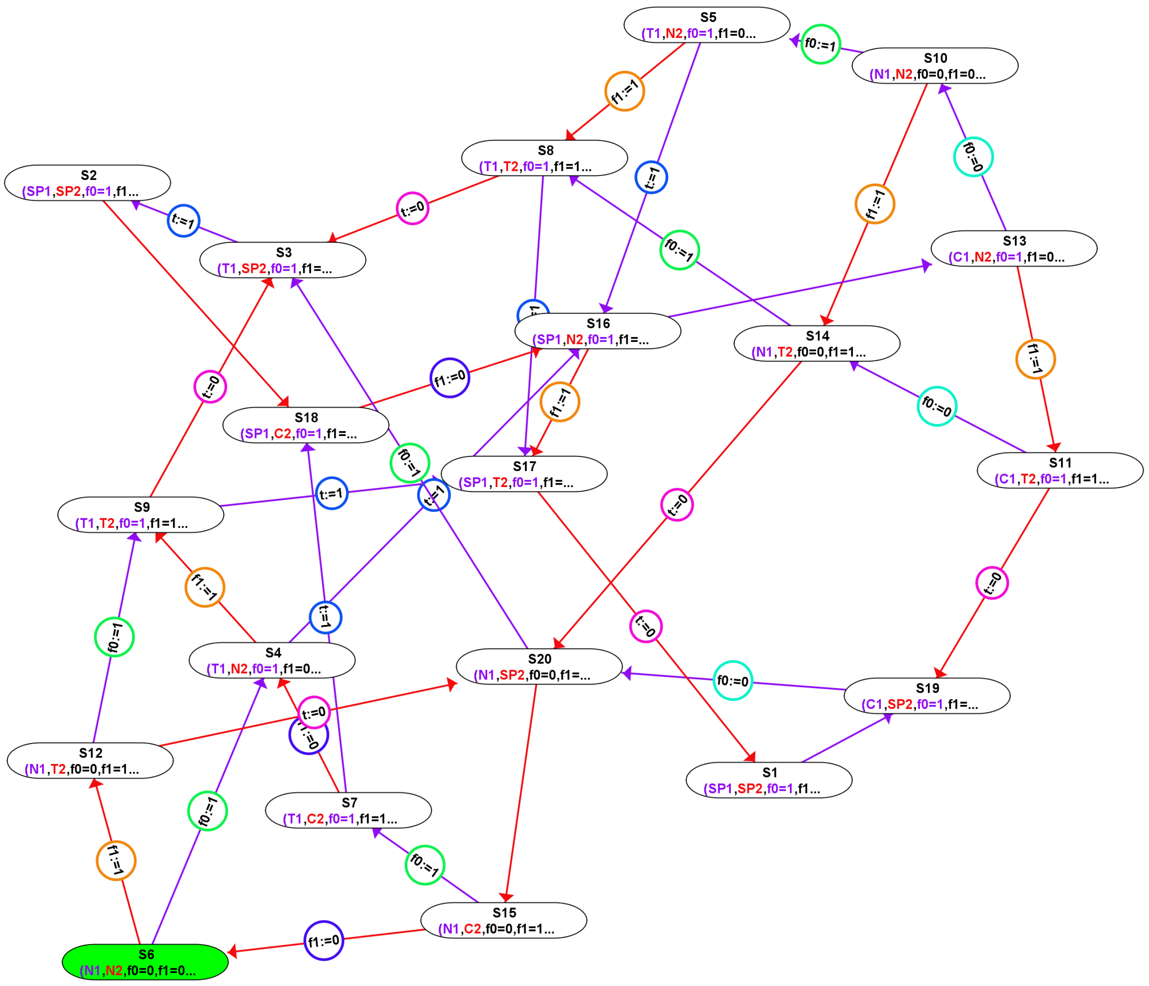


Figure 5.3: Peterson Algorithm Kripke Structure

5.1.3 Producer-Consumer Example

The producer consumer problem is a classic example of conncurent programming synchronization problem. In this probelm we have two processes, a producer and a consumer. They both share a fixed-size buffer. The producer adds data to the buffer and the consumer will consumer this data. The problem is to make sure that the producer does not add data to the buffer if it is full, and the consumer does not consume data from an empty buffer. A_i represents the local state where P_i is in sleeping mode, B_i represents the local state where P_i is awake. x represents the item count in the buffer.

```
  program  {
 initial   : A1 & A2 & (x=0) & !B1 & !B2
 process  1 {
  action   {
   l_grd : A1
   g_grd : (x<3)
   l_eff : { A1 :=ff; B1 :=tt;  }
   g_eff : { }
  }
  action   {
   l_grd : B1
   g_grd : true
   l_eff : { A1 :=tt ; B1 := ff;  }
   g_eff :{  x := x + 1; }
  }
 }
 process  2 {
  action   {
   l_grd : A2
   g_grd : (x>0)
   l_eff : { B2 := tt ; A2 := ff;  }
   g_eff : { }
   }
   action   {
   l_grd : B2
   g_grd : true
   l_eff : {  B2 := ff ; A2 := tt;    }
   g_eff : {x := x - 1; }
  }
 }
}

```

Listing 5.3: A producer-consumer example with a buffer size=3

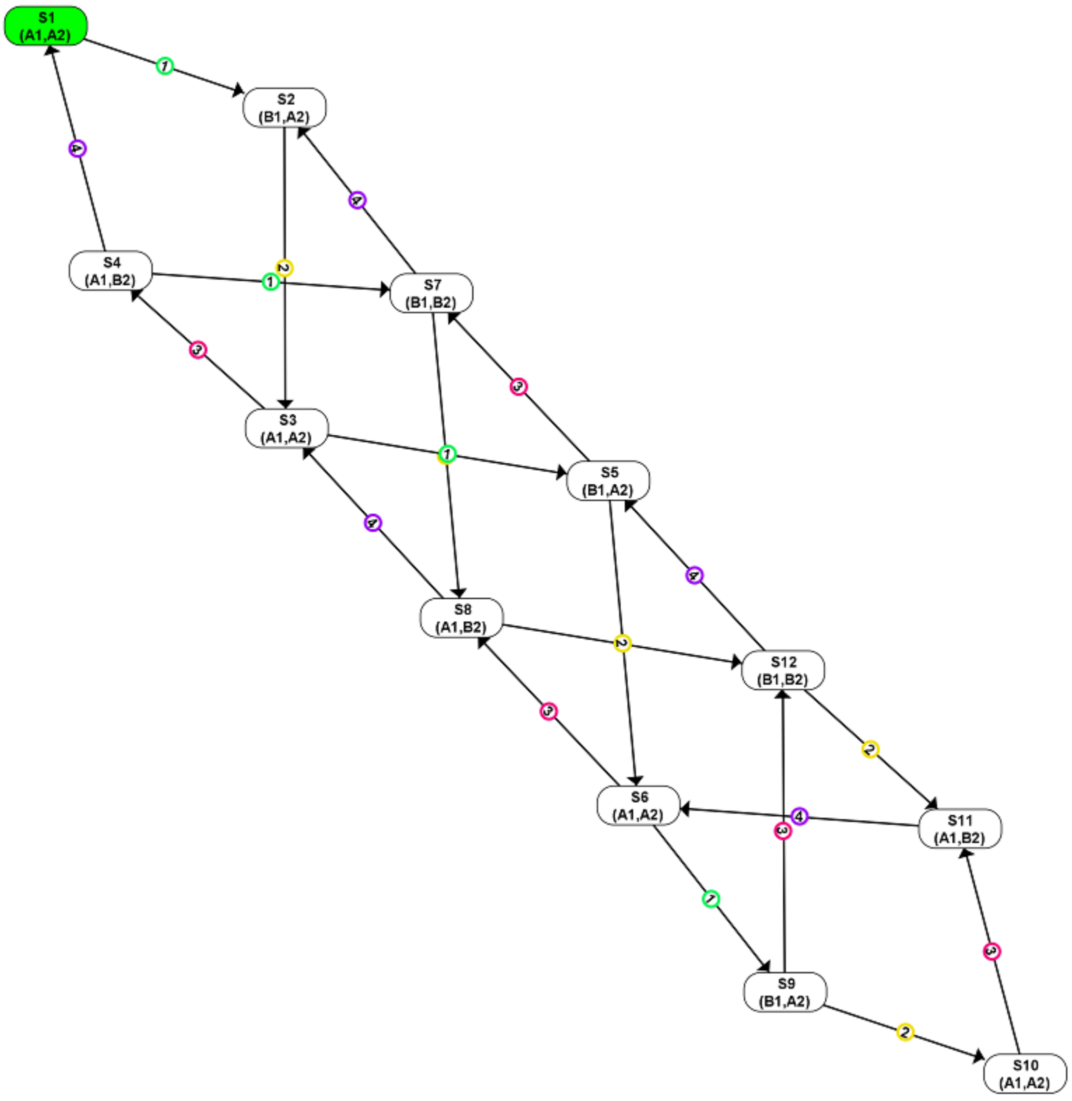


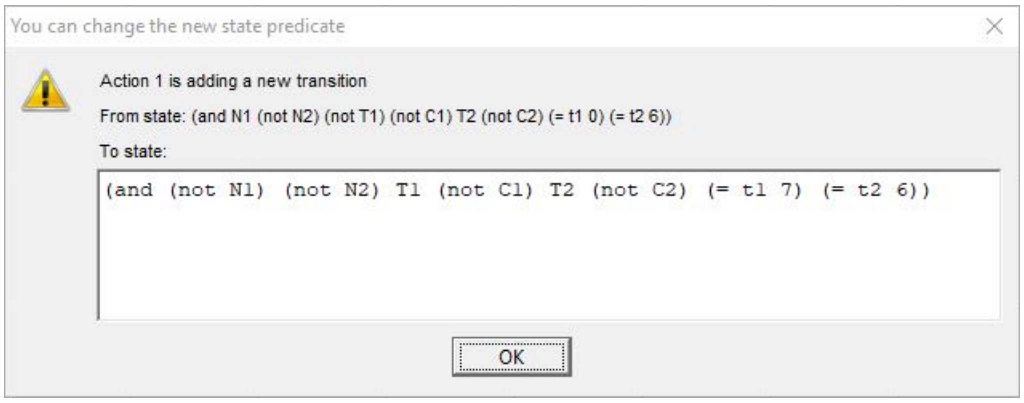
Figure 5.4: Three buffered size producer consumer example

5.2 Infinite State

5.2.1 Simplified Bakery Example

The Bakery Algorithm was devised by Leslie Lamport and it is a solution to the mutual exclusion problem for 2 or more processes. The local states for each process are N_i, T_i, and C_i similar to the previous examples. This algorithm preserves the first come first serve property. Whenever a process wants to enters the trying section, they receive a ticket number which is the max ticket number plus one, just like in a bakery shop. The holder of the smallest number will be eligible to enter its critical section. We have create 2 local variables t_1 for P_1 and $t2$ for P_2. These local variables can be read by both processes. Listing 5.4 shows the program text we created for this example.

The ticket number in this program is unbounded and can keep on incrementing till infinity. Generating a Kripke structure automatically will not terminate and here is where the user is prompted to interact with the algorithm. Figure 5.5 shows such an interaction.



Action 1 is adding a new transition

From state: (and N1 (not N2) (not T1) (not C1) T2 (not C2) (= t1 0) (= t2 6))

To state:

```
(and (not N1) (not N2) T1 (not C1) T2 (not C2) (= t1 7) (= t2 6))
```

Figure 5.5: Weakening state predicate interaction

Below is our interaction for program in Listing 5.4, on first line is the generated transition and the second line is our modification:

$$N_1,\ T_2,\ t_1 = 0,\ t_2 = 6 \to T_1,\ T_2,\ t_1 = 7,\ t_2 = 6$$
$$N_1,\ T_2,\ t_1 = 0,\ t_2 = 6 \to T_1,\ T_2,\ t_1 = 1,\ t_2 = 6$$

$$T_1,\ N_2,\ t_1 = 6,\ t_2 = 0 \to C_1,\ N_2,\ t_1 = 6,\ t_2 = 0$$
$$T_1,\ N_2,\ t_1 = 6,\ t_2 = 0 \to C_1,\ N_2,\ t_1 = 1,\ t_2 = 0$$

$$T_1,\ N_2,\ t_1 = 6,\ t_2 = 0 \to T_1,\ T_2,\ t_1 = 6,\ t_2 = 7$$
$$T_1,\ N_2,\ t_1 = 6,\ t_2 = 0 \to T_1,\ N_2,\ t_1 = 0,\ t_2 = 0$$

```
program  {
 initial   : N1 & N2 & (t1=0) & (t2=0) & !C2 & !C1 &!T1 & !T2
 process  1 {
  action   {
   l_grd : N1
   g_grd : N2 | T2 | C2
   l_eff : { N1 :=ff; T1 :=tt;  t1 := t2 + 1; }
   g_eff : {}
  }
  action   {
   l_grd : T1
   g_grd : (t1 < t2) | N2
   l_eff : { T1 :=ff; C1 :=tt;  }
   g_eff : {}
  }
  action   {
   l_grd : C1
   g_grd : N2 | T2 | C2
   l_eff : { N1 :=tt ; C1 := ff;  t1 :=  0; }
   g_eff : {}
  }
  }
 process  2 {
  action   {
   l_grd : N2
   g_grd : N1 | T1 | C1
   l_eff : { N2 :=ff; T2 :=tt;  t2 := t1 + 1; }
   g_eff : {}
   }
  action   {
   l_grd : T2
   g_grd : (t2 < t1) | N1
   l_eff : { T2 :=ff; C2 :=tt;  }
   g_eff : {}
  }
  action   {
   l_grd : C2
   g_grd : N1 | T1 | C1
```

```
    l_eff : { N2 :=tt ; C2 := ff;  t2 :=  0; }
    g_eff :{}
   }
   }
 }

```
Listing 5.4: Simplified Bakery Example

5.2.2 Bounded Bakery Example

This is an example to show the expressive power of our language, we added conditional statements which gives an upper bound on the the ticket number in the Bakery algorithm. Listing 5.5, shows at Lines 7 and 27 the if-else statement. Our Kripke generation algorithm will terminate when running this program and Kripke structure will be automaticaly generated.

Figure 5.6 shows the concrete structure generated from the simplified bakery example.

```
program  {
 initial   : N1 & N2 & (t1=0) & (t2=0) & !C2 & !C1 &!T1 & !T2
 process  1 {
 action   {
  l_grd : N1
  g_grd : N2 | T2 | C2
  l_eff : { N1 :=ff; T1 :=tt;  if(t2 < 5){ t1 := t2 + 1; } else { t1:= 5;
    }}
  g_eff : {}
 }
 action   {
  l_grd : T1
  g_grd : (t1 <= t2) | N2
  l_eff : { T1 :=ff; C1 :=tt;  }
  g_eff : {}
 }
 action   {
  l_grd : C1
  g_grd : N2 | T2 | C2
  l_eff : { N1 :=tt ; C1 := ff;  t1 :=  0; }
  g_eff :{  }
 }
 }
 process  2 {
  action   {
   l_grd : N2
   g_grd : N1 | T1 | C1
   l_eff : { N2 :=ff; T2 :=tt;  if(t1 < 5){ t2 := t1 + 1; } else { t2:= 5;
     }}
   g_eff : {}
  }
  action   {
   l_grd : T2
   g_grd : (t2 < t1) | N1
   l_eff : { T2 :=ff; C2 :=tt;  }
   g_eff : {}
  }
  action   {
   l_grd : C2
   g_grd : N1 | T1 | C1
   l_eff : { N2 :=tt ; C2 := ff;  t2 :=  0; }
   g_eff :{  }
  }
  }
}
```

Listing 5.5: Simplified Bounded Bakery Example Program with a max ticket = 5

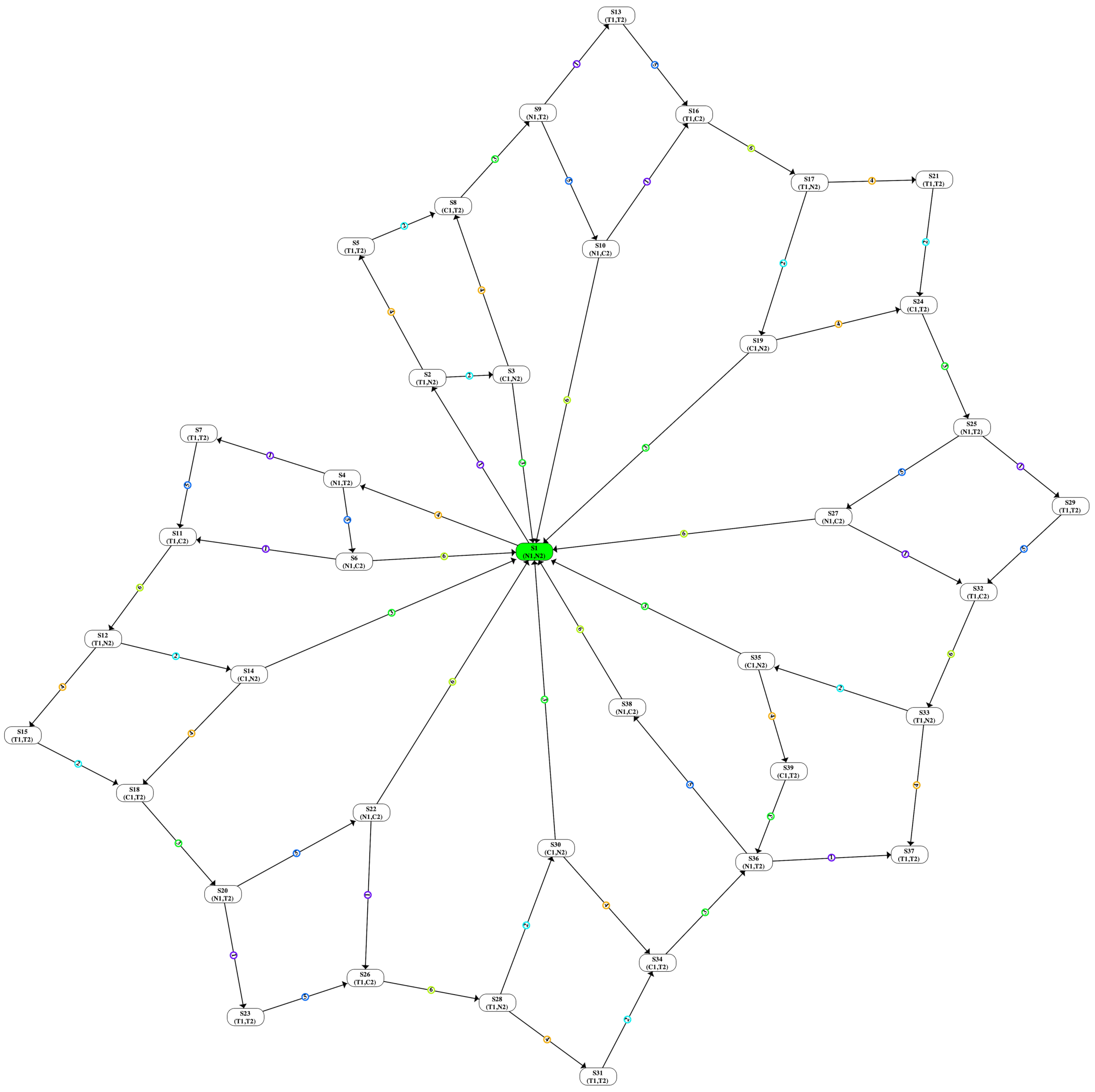


Figure 5.6: A simplified Bakery example with upper bound ticket number of 5

Chapter 6

Implementation

In this chapter we will detail our implementation and put all theoretical work into action. The text program semantics, Kripke structure generation and extraction algorithms, and model checking has been discussed in Chapters 3 and 4. Here we will focus more on syntax and Java implementation. We start with a brief summary of Eshmun, then we proceed by describing our work for finite and infinite state programs by topic.

6.1 Eshmun

We have implemented all of our work in Eshmun, a GUI-based tool available at `http://eshmuntool.blogspot.com/`. Eshmun is an interactive GUI tool created by Attie et al [2]; it allows users to create a Kripke structure M by adding states and transitions, do model checking and repair, export structures and import them as needed. It has many other facilities that are outside our current scope. Below is a brief description of modules we used from Eshmun in our current owrk.

1. *CTL Parser*: parses a CTL formula φ to generate a CTLParsedTree object which is a tree data structure representing φ .

2. *UI*: implements GUI interface between user and the other modules.

3. *Model Checker*: takes as input a Kripke structure $M = (S_0, S, R, L, AP)$, and a CTL formula φ and verifies if M satisfies φ.

4. *Model Repairer*: takes as input a Kripke structure M and a CTL formulae φ and return a repaired model with respect to φ.

5. *SAT Solver*: specifies whether a CNF formulae is satisfiable or not. In case it is satisfiable it also returns the satisfying valuation.

6.1.1 Changes on Existing Modules

We only modified one pre-existing module in Eshmun which is the *UI module.* We needed to extend a Kripke structure GUI component to be able to represent a FRKS. Under *eshmun.gui.utils.models.vanillakripke* we added the following:

1. **State** : we added a property to hold the state predicate in string format

2. **Transition** : we added 3 string properties for: precondition, postcondition and statement block.

Under *eshmun.gui.kripke.dialogs* we modified **StateDialog** to allow the user to modify the above added properties. Under *eshmun.gui.kripke.bars* we modified **EshmunMenuBar**, we added the following buttons:

1. *Import Program*: to import a finite program

2. *Import Program (Abstract View)*: to import a finite program with shared variable abstraction

3. *Import Infinite Program*: to import an infinite state program

4. *To Program (Text)*: to convert a Kripke Structure back to porgram text

6.1.2 New Modules Added

Our addition to Eshmun comes in 2 new modules that were added to it. Also, we integrated with Z3 SMT Solver to solve formulas of first order logic.

1. *Text Representation*: this module contains finite and infinite state implementation. It includes parsing text programs, Kripke structure generation, and program extraction.

2. *Hoare Triple Check*: implements verifying the validitiy of Hoare Triples found on Kripke structure transitions.

3. *Z3 Solver*: a state-of-the art theorem prover from Microsoft Research. Used by both modules above.

Figure 6.1 shows in blue the previously existing modules and in green the new modules we added. Below is a brief description of the modules that already existed in Eshmun and used in our implementation:

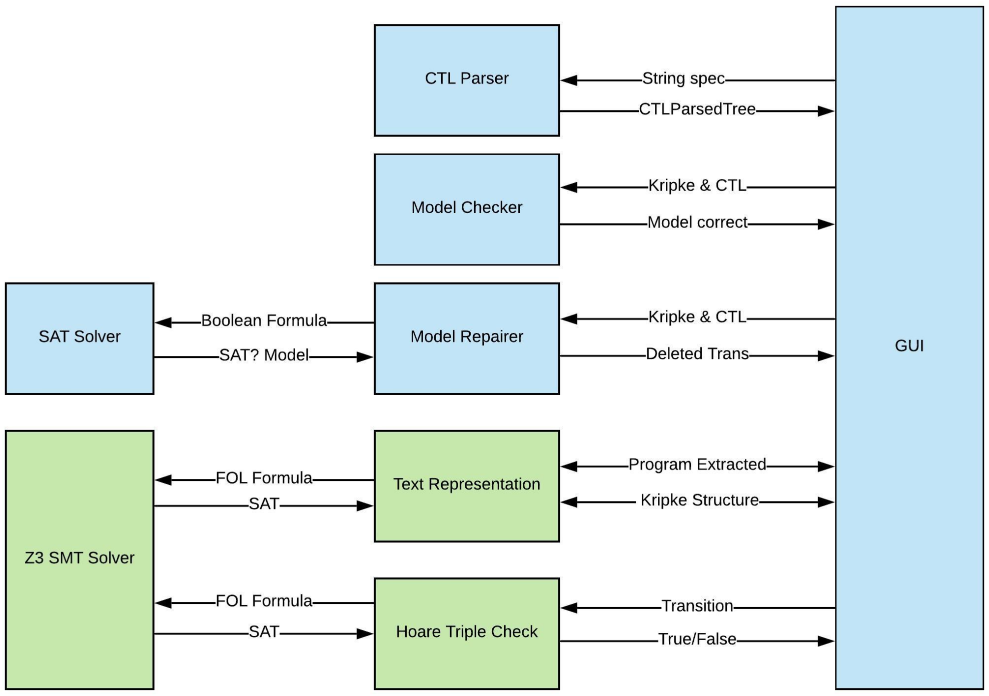


Figure 6.1: Eshmun Modules

6.2 Parsing Programs

Our text notation allows users to define a multi-process program in a text file and then import to Eshmun to generate the representing Kripke structure. Import can be chosen from File → Import → Program | Infinite Program. We have implemented 2 different parsers: **FiniteProgramParser** for parsing finite state, and **InfiniteProgramParser** for infinite state programs, both under *package eshmun.skeletontextrepresentation*. We start with parsing the basic structure which is shared between both types of programs and proceed with sections that distinct rules.

Each program starts with the keyword *program* and must declare an initial state, which is a logic formula, and a collection of *processes*. For this part, parsing the program skeleton, we are only concerned in 2 things: saving the initial state, and storing the action definitions. Parsing stores the *initial state* and keeps it in a static variable to be shared by within the scope of our operation. An action will be parsed and stored in a Type we created, named **SingleAction** defined under *package eshmun.skeletontextrepresentation.actions*. Each instance will contain local and global guards, and local and global effects definitions. The "?" in ANTLR means that this rule is optional, you can see that guards and actions are optional; which means that ommitting them will be the equivalence of *true* for guards and *skip* for effects. CTL will be handled to CTL Parser module.

```
prog: 'program' LCURLY intial processes RCURLY;
intial: 'initial' COLON formula;
processes: (process)+;

process: 'process' name?  LCURLY (action)* RCURLY;
name: IDENT | NUM_INT;
action: 'action'  name? LCURLY  localGuard?
                                globalGuard?
                                localEffect?
                                globalEffect?
                        RCURLY;

localGuard: 'l_grd' COLON formula SEMI?;
globalGuard: 'g_grd' COLON formula SEMI?;
localEffect:  'l_eff' COLON statement SEMI?;
globalEffect: 'g_eff' COLON statement SEMI?;
ctlFormula:   a valid CTL formula
```

Listing 6.1: Program structure ANTLR

Listing 6.1, shows the ANTLR grammar rules to declare a program skeleton. For finite state programs, we need to declare the finite domains of shared variables. This is achieved by introducting a new section, **sharedvariables**, to be added before declaring processess. These domains will be parsed and stored in a **Map<String, HashSet<String»** dictionary. Listing 6.2, shows the ANTLR grammar rules for adding shared variables.

```
prog: 'program' LCURLY intial sharedvariables?  processes RCURLY;
...
sharedvariables:  'sharedvariables'  COLON vardomainassignment (COMMA?
   vardomainassignment)* ;
vardomainassignment: variablename COLON vardomain;
vardomain: LEFT_CURL variablevalue  (COMMA variablevalue)* RIGHT_CURL;
...
```

Listing 6.2: Shared Variables in Finite State

6.3 Parsing Guards

A **SingleAction** defines a transition between two local states for process P_i and is guarded by local and global conditions. Our program is defined to abstract away any sequential executions not related to interprocess synchronization. Synchronization happens by evaluating the guards on each state in our model to check if it is enabled by a specific action. We will need to parse these guards and save them within each SingleAction.

6.3.1 Finite State Guards

Recall from Section 3.1, guards in finite programs are boolean formulas defined over the set of atomic propositions AP_i and shared variables. Below is the grammar rule for such formulas.

$$
\begin{aligned}
\textbf{formula} \longrightarrow\ & (\ G\ \&\ G\ ) \\
& \mid (\ G \mid G\ ) \\
& \mid (\ x\ =\ c\ ) \\
& \mid\ !\ G \\
& \mid booleanLiteral \\
& \mid atomicProp
\end{aligned}
$$

$$
\begin{aligned}
\textbf{atomicProp} \longrightarrow\ & AP\ starts\ in\ a\ capital,\ ends\ with\ integer\ representing \\
& proccess\ id \\
\textbf{booleanLiteral} \longrightarrow\ & \ tt\ \ true\ \ ff\ \ false
\end{aligned}
$$

We use ! for the negation, & for conjunction and | for disjunction of boolean expressions. Literals true and false can be replaced by tt and ff respectively. Shared variables x can be any valid identifier and can be checked for equality with any constant value c from the set of finite domain. In local guards, AP can only range on local atomic propositions (labels) of process i, and any shared variable test for equality (x=c) is discarded. In global guards, AP ranges on all AP_j where $j \neq i$, and shared variable equality checks are allowed.

The guards are parsed to a tree data structure where each node is of the abstract type **Guard** defined under *package eshmun.skeletontextrepresentation.guards.* Concrete types implementing Guard are:

1. **AndGuard**: stores a conjunction of guards and can have a collection of children.

2. **OrGuard**: stores a disjunction of guards and can have a collection of children.

3. **NotGuard**: stores a negation of a guard.

4. **AtomicGuard**: stores AP or shared variable equality checks.

5. **LiteralGuard**: stores a boolean literal, true or false.

6.3.2 Infinite State Guards

Recall from Section 4.2, in infinite state programs, states are represented by predicate logic formulas. Hence, our guards need to be more expressive thus we implement first-order logic syntax and semantics.

Below is the grammar rules for first-order logic guards:

$$
\begin{aligned}
\textbf{guard} &\longrightarrow formula \\[1em]
\textbf{formula} &\longrightarrow\ primitiveFormula \\
&\quad \mid formula\ connective\ formula \\
&\quad \mid quantifier\ variable\ formula \\
&\quad \mid !\ formula \\[1em]
\textbf{primitiveFormula} &\longrightarrow\ term\ relOp\ term \\
&\quad \mid atomicProp \\
&\quad \mid booleanLiteral \\[1em]
\textbf{term} &\longrightarrow\ (term\ arithmeticOp\ term) \\
&\quad \mid variable \\
&\quad \mid constant \\[1em]
\textbf{connective} &\longrightarrow\ \&\quad \mid \quad = \\
\textbf{quantifier} &\longrightarrow\ forall\quad exists \\
\textbf{relOp} &\longrightarrow\ =\quad !=\quad <\quad <=\quad >\quad >= \\
\textbf{arithmeticOp} &\longrightarrow\ -\quad +\quad *\quad / \\
\textbf{variable} &\longrightarrow\ any\ valid\ identifier\ string \\
\textbf{constant} &\longrightarrow\ any\ numeric\ value\ from\ \mathbb{N}
\end{aligned}
$$

A formula states a property that will eventually be interpreted as true or false. Formulas are defined recursively as:

- primitive formulas or atoms
- a logical combination of formulas
- a quantified formula using the universal or existential quantifiers

Logical connectives (| , & , and =) will be used to recursively build complex formulas, connectives like $\rightarrow$ can be constructed using | and ! . Terms are used to reference concrete objects: a constant, a variable, or an arithmetic operation in our case. Our implementation will parse guards to a Z3 **BoolExpr** object and save it in local or global guarded respectively. BoolExpr is a flexible tree data structure defined under defined under *com.microsoft.z3.BoolExpr*, it alows us to easily to manipulate and evaluate such expressions.

6.4 Parsing Effects

An action defines 2 types of effects: local effect and global effect. Local effects alter only the local state of a process owning the transition, while global effects alter globally shared variables. An effect is a command that needs to be executed during a transition between 2 states.

6.4.1 Finite State Syntax

Recall from Section 3.1 how a local effect for finite state programs will assign a true or false value for AP_i; which means it only have a local effect on the process perfroming an action. A global effect will only effect shared variables, it assigns values to shared variables from a predefined finite domain. We show the syntax for local effects, then for global effects.

$$\begin{aligned}
\textbf{localEffect} &\longrightarrow \mathit{atomicProp}\ (,\ \mathit{atomicProp})* \\
&\quad := \mathit{booleanLiteral}\ (,\mathit{booleanLiteral}) * \ SEMI \\
\textbf{globalEffect} &\longrightarrow \mathit{variable}\ (,\ \mathit{variable})*\ :=\ \mathit{constant}\ (,\ \mathit{constant}) * \ SEMI
\end{aligned}$$

Both effects will be parsed to type **AssignmentStatement**, a subtype of **Statement**, and stored within a SingleAction as **LocalEffect** and **GlobalEffect** of super type **Effect**. This statement contains an *emum* to define the assignment type as either: **BoolAssignment** for local effects , or **VariableAssignment** global effects.

It also contains a dictionary that keeps the values of each variable.

6.4.2 Infinite State Syntax

In finite state, effects had only assignments over AP and shared variables . For infinite state, recall from Section 4.2, we will add integer expressions, boolean expressions, and commands in the form of if-statements and sequential compositions. A command **C** will be parsed into a type **Statement**. A statement can be of subtype **AssignmentStatement**, **IfStatement**, and **BlockStatement**. These subtypes are all defined under package *eshmun.skeletontextrepresentation.commands* Below is the grammar, disregarding left-recurion to simplify the syntax.

$$
\begin{aligned}
\mathbf{C} \longrightarrow\ & if\ (\mathbf{B})\ \{\ C\ \}\ (\ \mathbf{else}\ \{\ C\ \}\ )\ ?\\
& \mid C; C\\
& \mid x\ :=\ E
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{B} \longrightarrow\ & B\ connective\ B\\
& \mid (E\ relop\ E)\\
& \mid !\ B\\
& \mid true\\
& \mid false
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{E} \longrightarrow\ & (\ E\ arithmeticOp\ E\ )\\
& \mid (-\ E\ )\\
& \mid x\\
& \mid n
\end{aligned}
$$

In grammar rule **E**, n is any numeral in $\{..., -2, -1, 0, 1, 2, ...\}$ and x is any variable. E represents integer expressions and supports basic operations like multiplication and addition. For command conditions, we introduced a syntactic rule for Boolean expressions **B**, which is freely expanded by relational operators and logical connectives. Now that we have expressions and boolean conditions, we define commands C, built from assignments and control structures.

6.5 Kripke Structure Generation

The purpose of defining text programs is to convert them to Kripke Structures where we can do CTL Model Checking. Generation is directly executed when a new program is imported to Eshmun. At this stage, the parsers has already saved the collection of Actions, initial state, and CTL specification formula.

We introduce 2 new types to define states and transitions:

1. **State** : a data structure for a Kripke stucture states. A state has a name, and a label which is a comma seperated string over AP defining true propositions in state. For infinite state, we introduce a subtype **InfiniteState** which has a BoolExpr to hold the state predicate.

2. **Transition** : a data structure that represents a single transition containing a *From* state and *To* state. It also stores a reference to the **SingleAction** responsible of the transition.

6.5.1 Finite State

Recall from Section 3.2 the description of constructing the Kripke structure of a program. We start by listing the most important functions and classes from parsing the program text to Kripke generation:

GuardVisitor

In order to evaluate if a state is enabled by a guard, we implemented a visitor pattern that traverses the Guard tree structure to evaluate a specific state on the guard. This visitor will help us in getting all states satisfying a guard. A state satisfies a guard by simply inserting the values of the APs and shared variables from the state and checking if a guard evaluates to true or false. Note that if a state does not contain an AP in its labels, then the value of this AP is evaluated to false. The visitor visiting an OrGuard, will evaluate all children gaurds and return the disjunction of the guard evaluations. For AndGuard, the result will be a conjunction of the children visiting result. The NotGuard, does a negation to the single child it has. The AtomicGuard, will check if the state has that same label or same variable equality check and return True, else returns False. The LiteralGuard will evaluate true or tt as True, and ff or false as False.

```
public static interface GuardVisitor {
/* Defines the interface for any visitor operating on Guard types */
    public boolean visit(OrGuard or, State state);
    public boolean visit(AndGuard and, State state);
    public boolean visit(NotGuard not, State state);
    public boolean visit(AtomicGuard label, State state);
    public boolean visit(LiteralGuard literalGuard, State state);

}
```

Listing 6.3: Guard Visitor

EffectsFiniteStateVisitor

In order to find the list of successor states of a specific state enabled by a guard, we implemented a visitor pattern to traverse the effects. Note that, in finite state implementation, the effects are not composed structures but we still implemented a visitor to apply the same design on infinite state effects later, which are tree like structures. On an action enabled state s, the LocalEffect visitor is invoked first and will produce a new state t with different labels, note that this new state already exists in the state space. After that, the GlobalEffect visitor is invoked on T and will apply all changes in the shared variables values. Thus the result will be to find the state t that satisfies the new values indicating that a transition to this state is valid starting from state s.

```
public static interface EffectsFiniteStateVisitor {
```

```
/* Defines the interface for any visitor operating on Finite State Effects
    */
    public State visit(GlobalEffect effect, State state);
    public State visit(LocalEffect effect, State state);

}
```

Listing 6.4: Effects Visitor

Our implementation allows the user to import a program in **Abstract** mode, which is reflected in visiting global effects. While in Abstract mode, if a user annotates the action by setting a variable to null in a global effect e.g., x:=null, the visitor will discard the variable from the state. In all other cases, setting the variable to null does nothing.

ProgramToKripkeConverter

ProgramToKripkeConverter handles dealing with UI and is called by **EshmunMenuBar** for converting an imported text program to a Kripke Structure. The main method is *convert(String args, boolean isAbstractView)*, it first calls the FiniteProgramParser to parse the program and then calls *generateFiniteStateKripke()*, defined below. The latter produces a set of transitions that will be passed to the **KripkeGenerator** class that contains the method *generateKripkeUIDefinition* to convert a Kripke to a string format to be sent to Eshmun UI module. This string format is defined by Eshmun help pages under Scripting → Structure Definition.

GenerateFiniteStateKripke()

This is the main method in generating the Kripke Structure, described in eSection 3.2. Its final result is a collection of all the valid transitions of the Kripke structure. Below is the pseudo code for the algorithm:

It includes a method *a.getEnabledStatesByGuards(allStates)* loops on all the state space and foreach state invokes the GuardVisitor on the action local guard first, if a state is enabled then it checks the global guard. If the local guard and global guard are satisfied together, then a state is enabled by an action, meaning this state has a transition going out of it. The method *a.generateTransitions(s)* will then invoke the local effect and then the global effect visitor to produce the resulting state t, and create the transition (s, t)

DeleteNodesWithNoIncomingEdges(T)

This is the final method before converting the generated structure to the string format the UI understands. Deletion runs recursively on the set of all generated transitions and delete all nodes i.e., states with no incoming edges. If a node is deleted, by default all its transitions are deleted. The method will keep running until no states are being deleted.

6.5.2 Infinite State

We now move to detail the implementation of generating the Kripke structure for infinite state programs. The main difference between the finite and infinite algorithm is that in finite state we can generate all thestate space by simulating the program actions and produce a finite number of the transitions. In infinite state, this may not be the case. Starting from the initial state, we start generating new states and transitions. These states will be represented by predicates and if our algorithms finds that it is not terminating, it might alert the user to enter weaker predicates for new states. In Section 4, we defined the theory behind the work. We now proceed to give brief a description of the main methods and classes involved.

StrongestPostConditionVisitor

This visitor implements Hoare logic predicate semantics to calculate a new predicate out of a post condition and a Command i.e., Statement. Recall our semantics from Section4.3.1, that action effects are more than just assignment statements. We add 2 new Commands: if-else statements, and sequential compoisitions of other statements. To generate the new predicates we traverse these Statements by calling method *getStrongestPostCondition()* using **StrongestPostConditionVisitor** visitor we implemented which implements **CommandLogicVisitor**.

```
public interface CommandLogicVisitor {
/* Defines the interface for any visitor operating on Infinite State
   Effects */
 BoolExpr visit(IfStatement statement, BoolExpr p, Context ctx );
 BoolExpr visit(BlockStatement statement, BoolExpr p, Context ctx);
 BoolExpr visit(AssignmentStatement statement, BoolExpr p, Context ctx);
}
```

Listing 6.5: Effects Visitor

BoolExpr is the data structure holding the state precondition. Context object is a Z3 type. The main interaction with Z3 happens via the Context. This object maintain all data structures related to objects and formulas that are created within the scope of our operation.

SolveAndSimplify()

The generation of the strongest postconditions introduces an existential quantifier on each assignment statement. This leads to very large formulas after few iterations of our Kripke generation algorithm. We implemented this method to simplify the produced formulas. The method tries to solve the formula first, it does so by asserting it to a the Z3 solver and then checking for its satisifability. If it is satisfiable, we get the model and rebuild a simpler formula by creating a conjunction of all values. If not, we simply call Z3 simplify which is just a bottom-up rewriter.

Algorithm 3: Solve And Simplify

```
input   : A predicate logic formula $Q$
output  : A simplified $Q$

solver := new Z3 Solver instance
solver.assert(Q)
if solver is SAT then
    Q := True
    for m in Model do
        Q := Q $\wedge$ (m.variable = m.value)
    end
    return Q
else
    return Q with Z3 default simplification
end
```

GenerateFRKS()

The details of generation algorithm are described in Section 4.3.2, and all previously mentioned methods, in this section, are used in the its main loop. For optimization, all satisfiability results are stored in memory to be used in later iterations. The algorithm uses Z3 Java API whenever it is checking for satisfiability, solving, or simplifying formulas.

We mentioned earlier that this algorithm is semi-automatic, meaning that item may not terminate, and this depends on the program we are trying to represent. The interactive mode will be using *JOptionPane.showMessageDialog* to allow users to weaken the predicates being generated on new states and possibly leading to termination.

6.6 Program Extraction

We only implemented program extraction for finite state programs, for inifinte state this will be in our future work.

6.6.1 Finite State

The aim of model checking a repair is to verify to the user wether a program is correct or not. After generating the Kripke Structure and interacting with, by model checking and model repairing, we provide the feature to extract back the program into text format. This will guide the user in implementing the program.

Recall from Section 3.3, the Kripke structure is a global-state transition diagram of a program, and this program can be extracted from M by "projecting" onto the individual processes. We do this by first projecting all transitions and then grouping the transitions by process families.

$s \uparrow i$ = start	$t \uparrow i$ = end	label = i,A	f.assign = A	B
N_1	T_1	1	skip	N_2
N_1	T_1	1	skip	C_2
N_1	T_1	1,x:=2	x=2	T_2 & $x = 2$
...	...	...	...	...

Table 6.1: Sample transitions from Kripke Structure in Figure 5.2

Table 6.1 shows the transitions of Process 1 from $N_1 \rightarrow T_1$ in the Kripke Structure of Figure 5.2. This gives rise to 2 actions in our text program, since by definition an action is a family of transitions having $F.start = s_i$, $F.finish = t_i$, $F.assign = A$, $F.guard = B$. The 2 action produces hence are:

$$N_1\ ,\ N_2 \vee C_2 \longrightarrow\ skip,\ T_1$$
$$N_1\ ,\ T_2 \longrightarrow\ x := 2,\ T_1$$

This work is implemented in class **KripkeToProgramConverter** under *package eshmun.skeletontextrepresentation.*

6.7 Hoare Triples Check for Infinite State

Recall from Section 4.3.1 how each transition, in FRKS, is a Hoare Triple of the form $\{P\}S\{Q\}$.

In Seciton 4.4.1, we described how weakest preconditions gets generated. For this purpose we implemented a new visitor **WeakestPreConditionVisitor** that implements **CommandLogicVisitor** show in Listing 6.5. This visitor traverses the command S of each transition carrying with it the postcondition Q and produces the weakest preconiditon.

For checking validity and satisfiability as described in 4.4.2 , we use Z3 SMT solver to and the above results to flag that a transition is valid or not valid. This result is retuned to the UI for the user to see it.

Chapter 7

Related Work

Our work combines deductive and model checking approaches to achieve better expressivity and handle infinite state programs while providing a high level of automation. Several abstraction methods have been devised to approximate infinite state spaces by models of finite state spaces.

7.1 Predicate Abstraction

In predicate abstraction [10], a concrete transition system is approximated by an abstract transition system. Properties can be verified by observing specific predicates over the concrete transition system. In the case of Lamport's Bakery algorithm, for example, we know that it is an infinite-state program since the ticket value may grow without any bound, e.g., when two processes alternately enter and exit the critical section. Predicates like $t1 < t2$ (t1 and t2 are ticket numbers of process 1 and 2), $t1 = 0$, $t2 = 0$ can be abstracted as boolean variables and used to build the abstract transition system. Also, states can be represented using these predicates. Properties verified of the abstract system can then be "concretized" by replacing the abstract booleans with their corresponding concrete predicates, and these concrete properties will hold of the concrete system.

7.2 Alloy Model Checker

Alloy [12] is a modelling notation that lets you enter specifications in restricted first order logic. The associated Alloy tool can then check validity by allowing the user to restrict the range of all variables to finite domains. It offers a decalaration syntax that can express complex constraints. It uses SAT solvers to verify the satisfiability of axioms defined in a model and to find counter examples.

7.3 Temporal Verification Diagrams

A verification diagram [13] is a way to visualize transitions between nodes labelled with assertions. A verification diagram is a directed labeled graph constructed as follows:

- Nodes: each node is labeled by an assertion.
- Edges: a directed edge between two nodes represents a transition between those nodes.
- Terminal nodes: a terminal node has no outgoing edges, and is used, e.g., for liveness properties.

Each directed edge generates a Hoare-triple, which must be proven valid, e.g., by proving the associated verificaiton condition, this can be done manually or with the aid of a theorem prover or SMT solver.

Chapter 8

Conclusions and Future Work

8.1 Summary of Contributions

The major results of this thesis are:

1. We implemented a textual notation for finite and infinite state concurrent programs. This is more flexible and easier to edit than the previous graphical synchronization skeleton notation. In the infinite state case, it is also much more expressive.

2. We implemented shared variable abstraction for finite state concurrent programs. This allows a user to indicate (using `x := null`) that the value of a shared variable can be subsequently ignored. This enables the use of significantly smaller Kripke structures to represent the behavior of a concurrent program.

3. We implemented finitely representable (infinite state) Kripke structures, along with Hoare-triple checking and model checking. We also implemented a semi-automatic method for generating a finitely representable Kripke structure from an infinite-state concurrent program.

4. We presented several case studies in finite state and infinite state to illustrate our methods.

8.2 Future Work

Future work includes:

1. Undertake more case studies.

2. Developing theoretical results on the correctness of the semi-automatic method for generating a finitely representable Kripke structure from an infinite-state concurrent program.

3. Semi-automatically repairing a finitely representable Kripke structure and extracting back the infinite state program from the resulting Kripke.

4. Extracting low atomicity and distributed concurrent programs from finite and infinite-state Kripke structures.

Bibliography

[1] Alessandro Artale. Formal methods lecture iv: Computation tree logic (ctl), 2010.

[2] Paul C. Attie, Kinan Dak Al Bab, and Mouhammad Sakr. Model and program repair via sat solving. *ACM Trans. Embed. Comput. Syst.*, 17(2):32:1–32:25, Dec. 2017.

[3] P.C. Attie and E.A. Emerson. Synthesis of concurrent programs for an atomic read/write model of computation. *TOPLAS*, 23(2):187–242, 2001.

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

[5] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking*. Springer, 2018.

[6] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall Inc., Englewood Cliffs, N.J., 1976.

[7] E. A. Emerson. Temporal and modal logic. *Handbook of Theoretical Computer Science*, B:997–1072, 1990.

[8] E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.

[9] Herbert Enderton. *A mathematical introduction to logic*. Academic Press, San Diego, Calif, 2001.

[10] S. Graf and H. Saidi. Construction of abstract state graphs with pvs. In *CAV*, volume 1254 of *LNCS*, pages 72–83. Springer, London, UK, 1997.

[11] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2012.

[12] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, 2002.

[13] Zohar Manna and Amir Pnueli. Temporal verification diagrams. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software, International Conference TACS '94, Sendai, Japan, April 19-22, 1994, Proceedings*, volume 789 of *Lecture Notes in Computer Science*, pages 726–765. Springer, 1994.