Verifying Mutable Systems

by

Joseph Scott

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2017

© Joseph Scott 2017

I hereby declare that I am the sole author of this thesis. This is not a true copy of the thesis, as it is still being reviewed by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Model checking has had much success in the verification of single-process and multi-process programs. However, model checkers assume an immutable topology which limits the verification in several areas. Consider the security domain, model checkers have had success in the verification of unicast structurally static protocols, but struggle to verify dynamic multicast cryptographic protocols.

We give a formulation of dynamic model checking which extends traditional model checking by allowing structural changes, *mutations*, to the topology of multi-process network models. We introduce new mutation models when the structural mutations take either a primitive, non-primitive, or a non-deterministic form, and analyze the general complexities of each. This extends traditional model checking and allows analysis in new areas.

We provide a set of proof rules to verify safety properties on dynamic models and outline its automizability. We relate dynamic models to compositional reasoning, dynamic cutoffs, parametrized analysis, and previously established parametric assertions.

We provide a proof of concept by analyzing a dynamic mutual exclusion protocol and a multicast cryptography protocol.

Acknowledgements

First and foremost, I would like to thank my advisor Richard Trefler for his teaching, endless patience, and encouragement provided throughout my time at the program.

I would like to thank my readers, Dan Berry and Arie Gurfinkel. I would like to thank the entire computer science college for making this experience so enjoyable, and Vijay Ganesh volunteering his mentorship. I would like thank my colleague Nguyen Hoang Linh for his discussion and collaboration.

I would like to thank the Computer Science and Mathematics colleges of Ohio University transforming me into a student ready for graduate school. My mentors, David Juedes, David Chelberg, Avinash Kodi, and Razvan Bunescu, and Sam Merten for encouraging me to pursue graduate studies and research.

I would thank my family for all the support, especially my mother Betty, and my friends back in Ohio.

Dedication

In loving memory of my grandmother, Alice Rebeca Scott, 1921 - 2017.

Table of Contents

List of Figures

Chapter 1

Intro

1.1 Motivation

Model checking is a fundamental approach to autonomously verifying concurrent and non-concurrent systems. However, model checking brings forth several challenging research problems. Defining both a concurrent and non-concurrent mathematical model with correctness criteria for a practical problem is in itself challenging. Furthermore, obtaining a proof of correctness can suffer from impractically high running times, which makes receiving a certificate of correctness a challenging research problem.

To combat the high running times in model checking, researchers use complexity reducing techniques such as compositional analysis, parametric analysis, cutoff analysis, and symmetry detection [65] [61] [63] [12] [42] [27] [15] [49] [41]. Furthermore, the verification of systems that have structurally or programmatically *dynamic* attributes is a challenging research problem. Modern model checkers are capable of verifying a structurally *static* model where a topology is fixed. Model checking has had a major impact in the debugging of concurrent systems, yet it's structurally static formulation makes it challenging to verify several practical models.

A popular application of formal methods and computer aided verification is to verify cryptographic security protocols [68] [58]. Traditional model checking struggles with capturing the *dynamism* of certain cryptographic security protocols. For example, consider a multicast security protocol, which allows for users to join and leave a secured network. The security specification of a multicast protocol is highly dependent on both a programmatic aspect and a topological aspect of the modeled network. Model checking has had positive results in verifying unicast protocols but has had little to no results in multicast protocols.

In this thesis we formulate *dynamic* model checking, in contrast to traditional *static* model checking. *Static* model checking analyzes reachability of a program or network of processes over its state space. We extend static model checking to analyze the reachability of programs under mutating topologies, which opens the door for computer-aided verification techniques in several new challenging domains.

1.2 Contributions and Related Work

This work was motivated by the work of Namjoshi, Trefler, published in TACAS 2015 [63]. This thesis extends their analysis. In their work, they provided a formulation to do dynamic process analysis and compute invariants in networks. They study program structures that can make primitive structural changes. Using parametric assumptions and compositional reasoning, they compute safety invariants on a dining philosophers and AODV protocol.

In this work, we provide a formulation of dynamic models and dynamic model checking. We build on [63] by extending their analysis beyond only primitive structural changes. This allows for the analysis of complex and non-deterministic user oriented Internet protocols by analyzing non-primitive structural mutations and non-deterministic structural mutations.

There have been other attempts at dynamic network analysis. Bouajjani et al. proposed a framework and logic to verify dynamic Petri net models [17]. Delzanno et al. analyzed the complexity and general decidability of types of dynamic ad-hoc network models [26]. Other approaches, include the analysis of families of computational models. Dynamic cutoff detection techniques [42] [4] have been used to find the smallest model that implies the verification of larger models and families of models. The parametrized model checking problem includes a size parameter of the model and asks if the specification holds for all values of the parameter [15].

The analysis of related families of concurrent systems is a growing area of research. However, this problem is undecidable in general [6]. The analysis of certain target models has resulted in several semi-decision procedures or even full decision procedures for specific models [65]. In this work, we provide analysis on a method that can verify safety specifications on concurrent protocol models that can be formulated in our analysis.

The deployment of computer aided verification tools is continuing trend in the domains of cryptography and security. SAT/SMT Solvers have been used to break cryptographic hash functions. Synthesis tools have been used to develop novel protocols. Theorem provers and model checkers have been used to verify small unicast protocols. Recently, the *Inductive Method* by Paulson et al. has been extended beyond unicast protocols to study multicast protocols. In this work, we this as momentum to extend model checking to multicast cryptographic protocols by analyzing models of mutating topologies and general families.

We relate dynamic model checking to other trends in model checking similar problem; the parametrized model checking problem (PMCP) and Cutoff analysis. We set forth criteria to use dynamic modeling to decide PMCP for a specific protocol. We discuss how to find the smallest initial model which implies security for the entire model as seen in cutoff detection.

1.3 Thesis Structure

- Chapter 1 - Introduction and Motivation of this thesis
- Chapter 2 - Discusses the preliminaries of this thesis. Specifically: propositional logic, theoretical computer science, and computer aided verification techniques.
- Chapter 3 - Introduces traditional model checking and common temporal logics. Algorithms used to perform model checking are discussed.
- Chapter 4 - Introduces and defines Concurrent Models and Concurrent Model Checking.
- Chapter 5 - Defines a structured model with a topology, topological mutations, and dynamic models
- Chapter 6 - Defines the dynamic model checking problem, how to prove safety properties, and relates to cut off detection, complexity reducing assertions, and PMCP.
- Chapter 7 - Analysis of a multi critical section mutual exclusion protocol and a multicast cryptographic protocol
- Chapter 8 - Concludes the thesis and discusses some further work to be done.

Chapter 2

Computer Aided Verification

2.1 Propositional Logic

In this section, we will provide the preliminary background of propositional logic that is required for this thesis. An understanding of propositional logic is assumed and for further reference, see [40].

Definition 1. *An* atomic proposition *is boolean variable that can take the value of truth, denoted as $\top$, or false, denoted as $\bot$.*

Definition 2. *A* propositional formula ψ*, over a set of atomic propositions AP, is defined inductively as follows:*

1. $\psi = \top$ *or* $\psi = \bot$.
2. $\psi = p$, *where* $p \in AP$.
3. $\psi = (\neg\alpha)$, *where* α *is a propositional formula.*
4. $\psi = (\alpha * \beta)$, *where* α, β *are propositional formula, and* $* \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$.

The unary connective, $\neg$, follows the semantics of a logical negation. The binary connectives, $\vee, \wedge, \rightarrow, \leftrightarrow$, follow the semantics of logical inclusive or, logical and, implication, and equivalence, respectfully. A strict use of parenthesis is enforced in Definition 2 to remove any ambiguity in order of operations of the logical connectives. There are a few standardizations of orders of operations over these logical connectives, but none are used in this thesis.

Definition 3 (Valuation and Evaluation)**.** *A* valuation *over AP is a function, $v : AP \to \{\top, \bot\}$, that assigns each atomic proposition a truth value. Furthermore, an evaluation of a propositional formula ψ with valuation v, denoted as $v(\psi)$, is a value of either $\{\top, \bot\}$ obtained by substituting all atomic propositions $p \in \psi$ with $v(p)$ and evaluating over the logical operations.*

In the following sections, we will discuss the problems of finding valuations with certain properties. We will next define types of propositional formula, where the formula takes a specific structure.

Definition 4. *A* literal *ℓ is a propositional formula of the form:*

1. *$\ell = p$, where $p \in AP$.*
2. *$\ell = (\neg p)$.*

Definition 5. *A* clause*, ϕ, is a propositional formula defined inductively as follows:*

1. *$\phi = \top$ or $\phi = \bot$.*
2. *$\phi = \ell$ where ℓ is a literal.*
3. *$\phi = (\phi_1 \vee \phi_2)$, where ϕ_1, ϕ_2 are clauses.*

Definition 6. *A* cube*, ϕ, is a propositional formula defined inductively as follows:*

1. *$\phi = \top$ or $\phi = \bot$.*
2. *$\phi = \ell$, where ℓ is a literal.*
3. *$\phi = (\phi_1 \wedge \phi_2)$, where ϕ_1, ϕ_2 are cubes.*

A literal, cube, and clause are the primitive types used to define the three most well known standard forms of a propositional formula.

Definition 7 (Standard forms)**.**

1. *A propositional formula, ψ, in Negated Normal Form (NNF) is defined inductively as follows:*
   (a) *$\psi = \top, \bot$*
   (b) *$\psi = \ell$, where ℓ is a literal.*

(c) $\psi = (\alpha * \beta)$*, where* α, β *are NNF propositional formula, and* $* \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$

2. *A propositional formula,* ψ*, in Conjunctive Normal Form (CNF) is defined inductively as follows:*

   (a) $\psi = \phi$*, where* ϕ *is a cube*

   (b) $\psi = (\phi_1 \wedge \phi_2)$*, where* ϕ_1, ϕ_2 *are clauses.*

3. *A propositional formula,* ψ*, in Disjunctive Normal Form (DNF) is defined inductively as follows:*

   (a) $\psi = \phi$*, where* ϕ *is a cube*

   (b) $\psi = (\phi_1 \wedge \phi_2)$*, where* ϕ_1, ϕ_2 *are cubes.*

Theorem 1. *Any propositional formula has a logically equivalent formula in NNF, DNF, and CNF.*

The proof is omitted, but we will outline it. The first note in the proof is that if a formula is in either DNF or CNF, then it must be in NNF. From there, the proof is by structural induction by showing that every well-formed formula can be converted to CNF by applying well known axioms, such as Demorgan's laws. However, the size of the resulting formula, in the worst case, exponential with respect to the original formula.

2.2 Logic and Complexity

In this section, we will introduce well known complexity classes, some of which are referenced later. Further problems relating to propositional logic that have impacts in computational complexity theory are discussed. For further reference, see [8], [39].

Definition 8 (Turing Machines)**.** *Formally, a Turing Machine,* T*, can be defined as a 7-tuple with* $T = (Q, \Delta, b, \Sigma, \delta, q_0, F)$*, where:*

1. Q *is a finite set of states.*

2. $q_0 \in Q$*, is the initial or starting state.*

3. F*, the set if final or accepting states* $(F \subseteq Q)$*.*

4. Δ *is a finite alphabet of the tape.*

5. *b is the empty symbol of the tape language ($b \in \Delta$)*

6. *Σ, the input characters on the tape at the start, ($\Sigma \subseteq \Delta \setminus b$)*

7. *δ a transition relation decides how to move along the tape, move in the state space, and write to the tape, ($\delta : (Q \setminus F) \times \Delta \to Q \times \Delta \times \{L, R\}$*

   M is a deterministic if the transition relation δ is a function, otherwise it is nondeterministic.

Definition 9 (Common Complexity Classes)**.** *Informally, the following complexity classes are denoted as follows:*

1. **P/NP** *is the set of decision problems that can be solved in polynomial time by a deterministic/non-deterministic Turing machine.* [1]

2. **EXPTIME/NEXPTIME** *is the set of problems that can be solved in exponential time by a deterministic/non-deterministic Turing machine*

3. **L/NL** *is the set of all problems that can be solved in logarithmic space using a deterministic/non-deterministic Turing machine.*

4. **PSPACE/NPSPACE** *is the set of all problems that can be solved in polynomial space using a deterministic/non-deterministic Turing machine.*

5. **EXPSPACE/NEXPSPACE** *is the set of all problems that can be solved in exponential space using a deterministic/non-deterministic Turing machine.*

Definition 10 (Completeness)**.** *Informally, for a problem A and complexity class* **C** *of all problems with property μ, A is* $\mathbf{C} - \mathbf{Complete}$ *if A is in* **C** *and for every problem in* **C**, *there exists a reduction from it to A that upholds μ.*

For example, to show a problem is $\mathbf{NP} - \mathbf{Complete}$, you need to show that there exists a non-deterministic Turing machine that can solve the problem in polynomial time and show there is a reduction from every problem in **NP** to said problem using a Turing machine in polynomial time. Informally, a problem is complete if its the *hardest* problem in its class.

Problem 1 (Boolean Satisfiability Problem)**.** *For a propositional formula ψ, does there exist a valuation v such that $v(\psi) = \top$.*

[1] The definition of **NP** is not the traditional definition, but an equivalent statement

Problem 2 (3-SAT)**.** *A propositional formula, ψ, is in 3-CNF if ψ is in CNF and the maximum number of atomic propositions in every clause is at most 3. The problem of 3-SAT asks the satisfiability problem over 3-CNF formula.*

Theorem 2 (Cook-Levin Theorem)**.** *3-SAT is* $\mathbf{NP-Complete}$.

The problem of deciding 3-Sat and general boolean propositional formula satisfiability is "the poster child" of $\mathbf{NP-Complete}$ problems. The showing of 3-SAT being in $\mathbf{NP-Complete}$ resulted in several other problems shown to be $\mathbf{NP-Complete}$ as a reduction from 3-SAT to the targeted problem creates a reduction from all problems in **NP** through the concatenation of the two reductions.

Completeness of a complexity class is most used when deciding if two complexity classes are equal. For example, if a $\mathbf{NP-Complete}$ problem is shown to have a polynomial algorithm, then it is in **P**. Furthermore, Since all problems in **NP** have a reduction to the said problem, a polynomial time algorithm for each problem in **NP** can be constructed by reducing it to the $\mathbf{NP-Complete}$ problem in polynomial time and then calling its polynomial time solution. But alas, no algorithm is known and the $\mathbf{P} \overset{?}{=} \mathbf{NP}$ problem remains open.

There are several other well known complete problems relating to propositional or extended logics. The problem of deciding $2-SAT$ is known to be $\mathbf{NL-Complete}$. Computing the evaluation of a propositional logic formula over a specific valuation is known to be $\mathbf{P-Complete}$. Deciding if a quantified boolean formula is satisfiable is known to be $\mathbf{PSPACE-Complete}$.

2.3 SAT Solving

In the last section, we discussed how there is no known asymptotically efficient algorithm for the SAT problem. Clever brute force implementations in SAT Solving engines are being developed and studied by researchers. These smart brute force engines are capable of deciding the problem of boolean satisfiability for formulas with tens of millions of atomic propositions, all while running an exponential worst case time algorithm.

The Davis Putnam Logemann Loveland algorithm (DPLL) is a sound and complete algorithm for the SAT problem. DPLL is a backtracking search based algorithm. DPLL is presented in Figure 2.1. DPLL builds all possible valuations by assigning one literal at a time and checks to see if the resulting formula is satisfied. If the partial valuation or partial assignment results in a conflict then go up a level of depth in the search tree.

```
procedure DPLL($\phi, V$)
    UnitPropogate($\phi, V$)
    if $\phi$ contains an empty clause then
        return FALSE
    else if $\phi = \top$ then
        return TRUE
    end if
    $p$ = PickProp($\phi, V$)
    return $DPLL(\phi, \{V \cup p\})$ or $DPLL(\phi, \{V \cup \neg p\})$
end procedure
```

```
procedure $UnitPropogate(\phi, V)$
    while $\phi$ has a clause with a single literal $\ell$ do
        Set every clause with $\ell$ to $\top$
        Delete $\neg \ell$ from every clause to which it appears
    end while
end procedure
```

Figure 2.1: The DPLL Algorithm

DPLL works with formulas that are in CNF. As seen in Theorem 1, every propositional formula can be converted to CNF but may be of exponential length. To get around this, the Tseytin Encoding is used to construct a formula that is equisatisfiable to the original formula that is in CNF. Two formula are equisatisfiable one is satisfiable if and only if the other is satisfiable.

Conflict-Driven Clause Learning (CDCL) is an extension of DPLL. CDCL includes several empirically shown optimizations to the DPLL algorithm. Figure 2.2 demonstrates the general architecture of a CDCL SAT Solver. The CDCL algorithm can be quite vague in the literature as to which extensions to DPLL are included, except for conflict clause learning.

The name driven change, conflict clause learning, is when a partial assignment $v = \ell_1 \wedge \ell_2 \wedge \ell_3 ...$ is reached while traversing but is found to contradict the input propositional formula. That is, $\psi \wedge v \rightarrow \bot$. We can then extend ψ equisatisfiably by considering a CNF formula ψ' where $\psi' = (\psi \wedge \neg v)$. Note that $(\neg v)$ is a clause as $\neg(\ell_1 \wedge \ell_2 \wedge ...) = (\neg \ell_1 \vee \neg \ell_2 ...)$.

Heuristic based literal selection is used to choose literals that result in a quicker termination. The most popular of which is the *VSIDS* heuristic which is a tally of appearances in ψ and the set of conflict clauses with an exponential decay. Some SAT Solvers deploy search space restarts while keeping learned conflicts which results in a more informed literal selection heuristic used when selecting the first literals to branch on. Upon finding conflicts, CDCL does back tracking analysis to skip up several levels of depth to greatly prune the search space using the first UIP algorithm [13] [56] [14].

Practical SAT Solving is one of the most interesting phenomena in mathematics in computer science. CDCL and DPLL algorithms have exponential running times of $\mathcal{O}(2^n)$

```
procedure CDCL($\phi, V$)
    if UnitPropogate($\phi, V$) == CONFLICT then
        return FALSE
    end if
    $depth = 0$
    while $V \neq n$ do
        $\ell = PickLiteral()$
        $depth = depth + 1$
        $V = V \cup \{\ell\}$
        if (UnitPropogate($\phi, V$)) == CONFLICT then
            $ConflictDepth = ConflictAnalysis(\phi, V)$
            if $ConflictDepth < 0$ then
                return $FALSE$
            else
                DepthLevel = BackTrack($\phi, V, ConflictDepth$)
            end if
        end if
    end while
    return TRUE
end procedure
```

Figure 2.2: The architecture of a CDCL Solver

with n being the length of the propositional formula. Formulas with tens of millions of variables can be solved in seconds. In fact, constructing problems that SAT Solvers struggle to solve is an active research area. Proof complexity theorists try to find lower bounds on the size of the proof that a SAT solver produces. One of the most fundamental examples is proofs of the unsatisfiability of the pigeon hole principle in which SAT Solvers struggle with instances of tens of variables, while software and hardware instances can handle tens of millions of variables. [1]

2.4 Model Checking

Model checking is a fully automatized method for the verification of finite state systems. The model checking problem asks, given a transition graph, a starting state, and a property, compute the set of reachable states to which the property is true. This section introduces model checking and outlines it informally. Model checking is the main topic of this thesis and will be discussed in greater detail in later chapters.

In Figure 2.3, the overall pipeline of the model checking process is presented. A target model is selected and extracted to a transition system. For common applications such as software model checking and hardware model checking, there are tools that can do this automatically.

Model checking suffers from high running times, and large models can be impractical to verify. Thus abstraction on the target model is often performed. Abstraction needs to be done carefully as an excessive abstraction can result in a coarse model such that a correctness certificate is generated but the original model is bug ridden. The property that is verified in the model must be properly translated into logic. The desired property, or *specification*, of the model is translated into some logic most often a temporal logic.

To illustrate model checking, I will attempt to model my daily life as a graduate student and examine my study habits. My life as a graduate student has had a ton of detail, so a high amount of abstraction needs to take place to avoid an unreasonably large model. As we are dealing with study habits, we will consider the propositions *working*, *sleeping* and *playing* to denote what I am doing in a given state. Studying, writing, attending meetings, and going to talks are all being abstracted into the *working* proposition while all my hobbies such as games, television, or recreational web browsing are abstracted as the *playing* proposition. To denote the location of where I am, I use the propositions of *AtSchool* and *AtHome*.

In the figure, the top left state of the figure denotes a state of sleeping at home, the

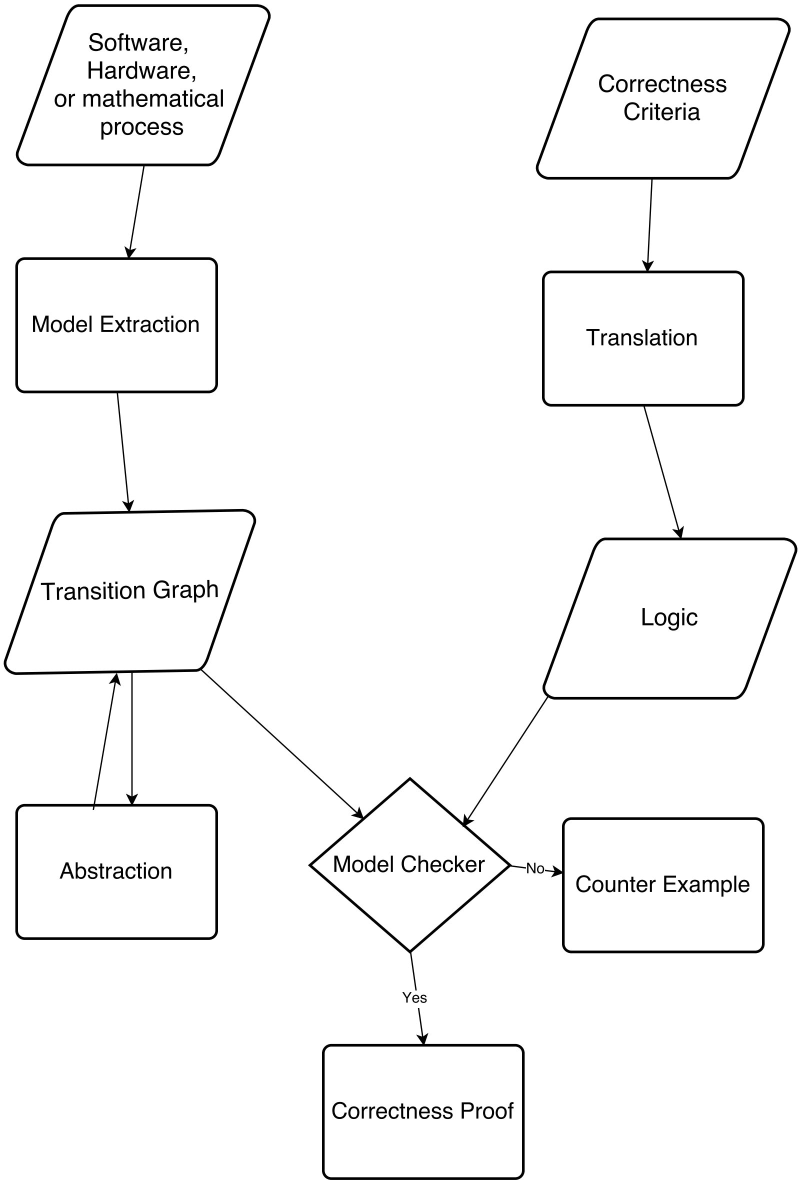

Figure 2.3: The Pipeline of Model Checking

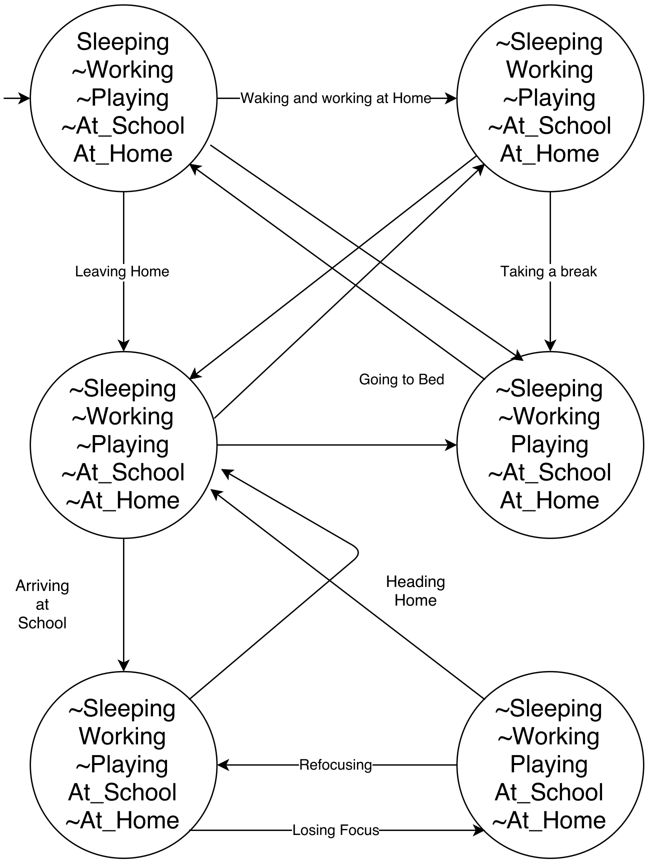

Figure 2.4: A transition graph of the life of a graduate student

top right node denotes a state of working at home, the middle left state denotes a state of traveling to work from home or to home from work, the middle right state denotes playing at home, the bottom left state denotes working at school, and the bottom right state denotes playing at school.

A graduate student with good study habits should study every day. To properly capture this formally, we will use a temporal logic in a later chapter, but informally, this would mean that there is no cyclic path from the state of sleeping where *Working* is always true. However, this is not true in the transition system! A model checker would return the bug that I can transition from sleeping to playing at home all day back to a sleeping state. One example of a fix would be to reformulate my daily routine by removing the transition from sleeping to playing at home.

Chapter 3

Model Checking

3.1 Overview

In Chapter 2.4 we outlined the concept of model checking. In this chapter, we will more formally redefine the concepts presented. First and foremost, we will formally define the previously denoted transition graph as a Kripke Structure.

Definition 11 (Kripke Structure)**.** *Let AP be a set of atomic propositions. A* Kripke Structure*, M over AP is a four tuple with $M = (S, S_0, R, L)$, where:*

1. *S is a finite set of states*
2. *$S_0 \subseteq S$ is the set of initial states*
3. *$R \subseteq S \times S$ is a transition relation that must be total, that is, for every state $s \in S$ there is a state $s' \in S$ such that $R(s, s')$.*
4. *$L : S \to 2^{AP}$ is a function that labels each state with the set of atomic propositions that are true in that particular state.*

We will provide an example by using the transition system in Figure 2.4 discussed in Chapter 2.4 which denoted the daily routine of a graduate student. Our set of atomic propositions $AP = \{Sleeping, Working, Playing, AtSchool, AtHome\}$. We will label each state in the figure as s_1 to s_6 starting from the top to bottom, going from left to right. This would result as:

1. $S = \{s_1, s_1, s_2, s_3, s_4, s_5, s_6\}$

2. $S_0 = \{s_1\}$

3. $R = \{(s_1, s_2), (s_1, s_3,), (s_1, s_4), (s_2, s_3), (s_2, s_4), (s_3, s_2), (s_3, s_4), (s_3, s_5),$
   $(s_4, s_1), (s_4, s_2), (s_4, s_3), (s_5, s_3), (s_5, s_6), (s_6, s_3), (s_6, s_5)\}$

4. $L = \{(s_1, \{Sleeping, AtHome\}), (s_2, \{Working, AtHome\}), (s_3, \{\}),$
   $(s_4, \{Playing, AtHome\}), (s_5, \{Working, AtSchool\}),$
   $(s_6, \{Playing, AtSchool\})\}$

To reason about how a Kripke structure transitions we will define a path. In the next section we will introduce temporal logic that extends the propositional logic to further reason about a path.

Definition 12 (Paths)**.** *A* path *in a Kripke structure $\mathcal{M}$ from a state $s \in S$ is a infinite sequence of states, $\pi = s_0 s_1 s_2 ...$ such that $s_0 = s$ and $R(s_i, s_i + 1)$ holds for all $i \geq 0$.*

The reason for the total requirement in definition 11 is to enforce infinite paths. In our current example, we considered the specification that every day a graduate student does some work. We syntactically index a state along a path π using a superscript and an integer. For example, π^3 denotes the 3rd state in the path. The existence of the path $\pi = s_1 s_3 s_1 s_3 s_1 s_3 ....$ is a bug in the daily routine. To write this specification formally, we will use temporal logic.

3.2 Linear Temporal Logic

We are about to introduce LTL, the first of three temporal logics discussed in this thesis. All formula of LTL are *path properties*, the formula's semantics reason about a path in a Kripke structure. The other type of formula we will consider are *state properties*, in which the formula's semantics reason about the properties of a specific state. In LTL, every formula is a path formula.

Definition 13 (Linear Temporal Logic)**.** *We will define LTL inductively,*

- $\top, \bot$ *are LTL formula.*

- *If $p \in AP$ (Atomic Propositions) then p is an LTL formula.*

- *If ψ, ϕ are LTL formula then the following are also LTL formula*
  - $\psi \vee \phi$
  - $\psi \wedge \phi$
  - $\neg\psi$
  - $\psi \implies \phi$
  - $\psi \iff \phi$
- *If ψ, ϕ are LTL formula then the following are LTL formula*
  - $\mathbf{G}\phi$*, ϕ is "**g**lobally" true along path π.*
  - $\mathbf{F}\phi$*, ϕ is eventually or in the "**f**uture" true along path π.*
  - $\mathbf{X}\phi$*, ϕ is true on the "ne**x**t" state on π.*
  - $\phi\mathbf{U}\psi$*, ϕ is true "**u**ntil" ψ is true, along path π.*
  - $\phi\mathbf{R}\psi$*, ψ is is true up until ϕ comes true (on **r**elease" of), along path π.*

The tokens $\vee, \wedge, \neg, \implies, \iff$ have the same semantics as seen in propositional logic. Let ϕ, ψ be an LTL formula; then the following is the formal semantics of the temporal operators.

- $\mathcal{M}, \pi \models \mathbf{G}\psi \iff$ for all $i \geq 0, \mathcal{M}, \pi^i \models \psi$
- $\mathcal{M}, \pi \models \mathbf{X}\psi \iff \mathcal{M}, \pi^1 \models \psi$
- $\mathcal{M}, \pi \models \mathbf{F}\psi \iff$ there exists a $k \geq 0$ such that $\mathcal{M}, \pi^k \models \psi$.
- $\mathcal{M}, \pi \models \psi\mathbf{U}\phi \iff$ there exists a $k \geq 0$ such that $\mathcal{M}, \pi^k \models \phi$ and forall $0 \leq j < k, \mathcal{M}, \pi^j \models \psi$
- $\mathcal{M}, \pi \models \psi\mathrm{R}\phi \iff$ for all $j \geq 0$, if for every $i < j$ $\mathcal{M}, \pi^i \not\models \psi$ then $\mathcal{M}, \pi^j \models \phi$

Theorem 3. *$\{\mathbf{U}, \mathbf{X}, \neg, \vee\}$, $\{\mathbf{R}, \mathbf{X}, \neg, \vee\}$, $\{\mathbf{W}, \mathbf{X}, \neg, \vee\}$ are adaquete LTL sets.*

Proof. First note that instead of $\neg, \vee$ any other adequate set of propositional logic can be used instead. The result then comes as

1. $\psi\mathbf{R}\phi \equiv \neg(\neg\psi\mathbf{U}\neg\phi)$

2. $\psi \mathbf{W} \phi \equiv \phi \mathbf{R}(\psi \vee \phi) \equiv \neg(\neg\phi \mathbf{U} \neg(\psi \vee \phi))$

3. $\psi \mathbf{U} \phi \equiv \neg(\neg\psi \mathbf{R} \neg\phi)$

4. $\psi \mathbf{W} \phi \equiv \phi \mathbf{R}(\psi \vee \phi)$

5. $\psi \mathbf{U} \phi \equiv \neg(\neg\psi \mathbf{R} \neg\phi)$

6. $\psi \mathbf{R} \phi \equiv \phi \mathbf{W}(\psi \wedge \phi)$

□

Theorem 4 ([73])**.** *LTL Model Checking is* **PSPACE – Complete**

The proof of Theorem 4 is omitted from this thesis, but can be found in [73].

3.3 Computation Tree Logic

In LTL, all logical formula were *path* formula. In CTL, all logical formula are *state formula*. A *state formula* is a formula to which its semantics reason about properties of states. CTL includes the temporal operators of LTL but restricts that they must be appended to a path quantifier. In CTL there are two path quantifiers:

- **A**, along all paths
- **E**, there exists a path

Let $\mathcal{M}$ be a Kripke structure with path π and state $s \in S$. Let ϕ, ψ be *path formula* Formally, the semantics of the path quantifiers are as follows:

- $\mathcal{M}, s \models \mathbf{A}\psi \iff$ for every path π from s, $\mathcal{M}, \pi \models \psi$
- $\mathcal{M}, s \models \mathbf{E}\psi \iff$ there is a path π from s such that $\mathcal{M}, \pi \models \psi$.

Definition 14. *We can define CTL inductively*

1. $\top, \bot$ *are CTL Formula*

2. If ψ, ϕ *are CTL formula then the following are CTL formula*

- $\psi \vee \phi$
- $\psi \wedge \phi$
- $\neg\psi$
- $\psi \implies \phi$
- $\psi \iff \phi$

3. *If ϕ, ψ are CTL formula then the following are CTL formula*
   - $\mathbf{AX}\phi$
   - $\mathbf{EX}\phi$
   - $\mathbf{AF}\phi$
   - $\mathbf{EF}\phi$
   - $\mathbf{AG}\phi$
   - $\mathbf{EG}\phi$
   - $\mathbf{AU}\phi, \psi$
   - $\mathbf{EU}\phi, \psi$
   - $\mathbf{AR}\phi, \psi$
   - $\mathbf{ER}\phi, \psi$

Theorem 5. $\neg, \vee, \mathbf{EX}, \mathbf{EU}, \mathbf{EG}$ *is an adequate set over CTL.*

Proof. As before any adequate set of propositional logic can be used instead of $\neg, \vee$. The result then comes as the following are equivalent:

1. $\mathbf{AX}\psi \equiv \neg\mathbf{EX}(\neg\psi)$
2. $\mathbf{EF}\psi \equiv \mathbf{EU}\top, \psi$
3. $\mathbf{AG}\psi \equiv \neg\mathbf{EF}(\neg\psi)$
4. $\mathbf{AU}\psi, \phi \equiv \neg\mathbf{EU}(\neg\phi), (\neg\psi \wedge \neg\phi) \wedge \neg\mathbf{EG}\neg\phi$
5. $\mathbf{AR}\psi, \phi \equiv \neg\mathbf{EU}(\neg\psi), (\neg\phi)$
6. $\mathbf{ER}\psi, \phi \equiv \neg\mathbf{AU}(\neg\psi), (\neg\phi)$

□

Theorem 6 ([72],[23],[7])**.** *CTL model checking can be done in* $\mathcal{O}(|\mathcal{M}| \cdot |\psi|)$ *and is in the* $\mathbf{P}-\mathbf{complete}$ *complexity class.*

The proof of the first part of the theorem is by the construction of an algorithm. The algorithm is discussed in Chapter 3.6. $\mathbf{P}-\mathbf{Complete}$is analogous to $\mathbf{NP}-\mathbf{Complete}$, in that every problem in class P can be polynomially reduced to it and it has a polynomial time solution. Some of the open questions of complexity theory deal with efficiently parallelizing P-Complete problems and solving P-Complete problems with logarithmic space. The proof of CTL being $\mathbf{P}-\mathbf{Complete}$ goes by a polynomial reduction to the Circuit Value Problem, which asks given a boolean circuit and input to the circuit compute the output value of the circuit.

3.4 LTL vs. CTL Expressibility and CTL*

In the previous sections, LTL and CTL were defined and discussed. In LTL, we reason about properties of a path, while in CTL we reason about state and path properties. CTL does not have an exponential running time as it is linear in the size of the model, but the size of the model is often exponential in the number of atomic propositions.

There are formula that can be expressed in both LTL and CTL, for example, a single atomic proposition. Furthermore, if $\psi_{CTL} = \psi_{LTL}$ then $\mathbf{AG}\psi_{CTL} = \mathbf{G}_{LTL}$. But there are also examples of formula that are in one but not the other. For example, $\mathbf{A}(\mathbf{FG}p)$ is a valid LTL formula but not expressible in CTL.

The next logic is a superset of both LTL and CTL that has both state formula and path formula with the union of the previously discussed state operators and path quantifiers.

Definition 15 (CTL*)**.**

- $\top, \bot$ *are CTL* formula.*
- *If* $p \in AP$ *(Atomic Propositions) then* p *is an CTL* formula.*
- *If* ψ, ϕ *are CTL** state *formula then the following are also CTL* formula*
    - $\psi \vee \phi$

- $\psi \wedge \phi$
- $\neg\psi$
- $\psi \implies \phi$
- $\psi \iff \phi$

- *If ψ, ϕ are CTL* formula such that ψ, ϕ are* path formula *then the following are CTL* formula*

  - $\mathbf{G}\phi$, ϕ
  - $\mathbf{F}\phi$, ϕ
  - $\mathbf{X}\phi$, ϕ
  - $\phi\mathbf{U}\psi$,
  - $\phi\mathbf{R}\psi$,
  - $\mathbf{A}\phi$
  - $\mathbf{E}\phi$

Theorem 7 ([72]). *CTL* model checking is is* **PSPACE – Complete**

This result is seen as unsurprising by some as CTL* resembles the union of both LTL and CTL, but also surprising to some as CTL* is much more expressible than both LTL and CTL.

3.5 Ordered Binary Decision Diagrams

In the remaining sections of this chapter we will outline a few approaches of model checking that are implemented in modern model checkers.

An ordered binary decision diagram (OBDD) is a data structure for the representation of a propositional formula and its evaluation for any interpretation. They resemble a binary decision tree but mainly differ by its canonical form.

Definition 16. *Let ψ be a propositional formula. A binary decision diagram G for ψ is a rooted directed acyclic graph such that every non-terminal node denotes is labeled with an atomic proposition of ψ with exactly two outgoing edges labeled as either $\top$ or $\bot$. Every terminal node is labeled with either $\top, \bot$.*

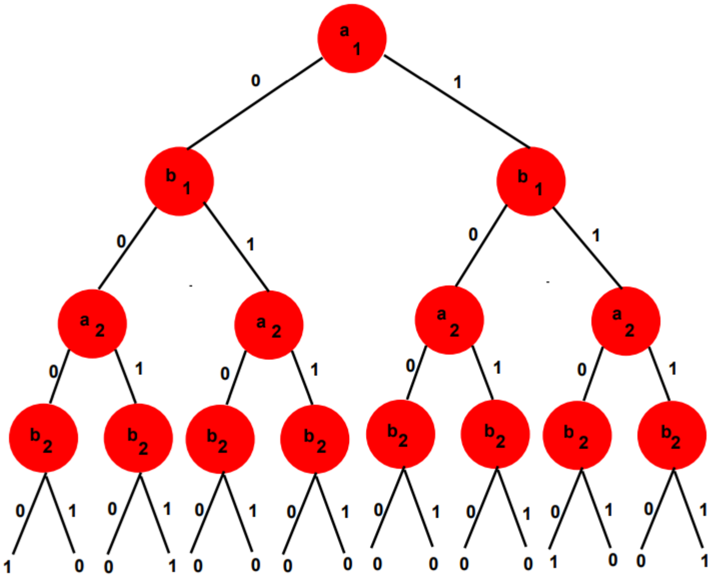

Figure 3.1: A binary decision diagram for $\psi = (a_1 \iff b_1) \wedge (a_2 \iff b_2)$

An example of a binary decision diagram is given in Figure 3.1 for the formula $\psi = (a_1 \leftrightarrow b_1) \wedge (a_2 \leftrightarrow b_2)$. The structure of a binary decision diagram very much resembles the process that is done by a DPLL SAT Solver. The DPLL SAT solver performs the depth first search until it finds the terminal $\top$ node or returns no satisfying assignment exists after traversing the entire tree.

As seen when considering the CDCL SAT solving algorithm compared to the DPLL SAT solving algorithm notable improvements can be made which results in a much smaller search. The (perhaps less clear) analogy to CDCL SAT Solving is with Ordered Binary Decision Diagrams.

Definition 17. *Let G be a binary decision diagram of formula ψ. Let $ord(p, q)$ be a total ordering relation of every proposition $p, q \in \psi$. G is an ordered binary decision diagram if it has the following properties:*

1. *(Redundant Termainals) There are exactly two terminal verticies.*
2. *(Ordered) For propositions $p, q \in \psi$, with labeled nodes $u, v \in G$ respectively. If there is an outgoing edge from u to v then $ord(p, q)$.*

3. *(Redundant Nodes) For every two nonterminal nodes with label $p, p' \in \psi$, and children nodes with label $q, q', r, r' \in \psi$ respectively, then $p \neq p'$ or $q \neq q'$ or $r \neq r'$.*

4. *(Redundant Tests) There does not exists a nonterminal node with children node lables $p, q \in \psi$ where $p = q$.*

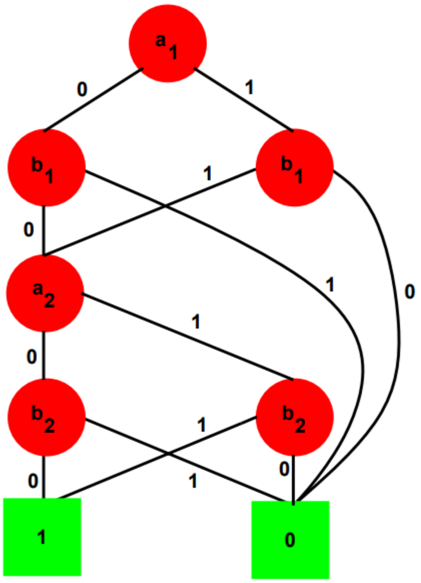

Figure 3.2: The ordered binary decision diagram from Figure 3.1

The conversion of a binary decision diagram to an ordered binary diagram is fairly straightforward and has been shown to be done in linear time in size of the binary decision diagram [19]. The binary decision diagram is shown in Figure 3.1 is converted to a ordered binary decision diagram in figure 3.2. The ordering is of $a_1 < b_1 < a_2 < b_2$.

Building upon the analogy of binary decision diagrams is to DPLL as ordered binary decision diagrams are to CDCL, the process of removing redundancies of a binary decision diagram is analogous to the learning of conflict clauses when searching through the state space of a formula. Furthermore, the selection of the total ordering relation resembles that of branching heuristics deployed in CDCL SAT solvers.

OBDD's are used in practice for model checking. A kripke structure and its specification is translated into a single logical equation and represented as an OBDD and queried. A formal description of how to translate a kripke structure and the CTL adaquete set in Theorem 5 is provided in [24] and with LTL in [21].

3.6 Buchi Automata

In the last section, we demonstrated a data structure approach to CTL model checking. In this section, an automata theoretic approach using Buchi Automata is demonstrated to perform model checking on LTL specifications.

Definition 18. *A Buchi Automata is a $\omega-$automaton over infinite languages*

A deterministic Buchi Automata $\mathcal{B} = (Q, \Sigma, \delta, q_0, F)$ is a five tuple where:

1. *Q is a finite set of states*
2. *Σ is a finite alphabet*
3. *$\delta : Q \times \Sigma \rightarrow Q$ is the transition function*
4. *$q_0 \in Q$ is the initial state*
5. *$F \subseteq Q$ is the accepting states.*

A non-deterministic Buchi Automata $\mathcal{N} = (Q, \Sigma, \Delta, I, F)$

1. *Q is a finite set of states*
2. *Σ is a finite alphabet*
3. *$\Delta : Q \times \Sigma \rightarrow \mathcal{P}(Q)$ is the transition function, with $\mathcal{P}$ denoting the power set.*
4. *$I \subseteq Q$ is the initial state*
5. *$F \subseteq Q$ is the accepting states.*

$\mathcal{B}, \mathcal{N}$ accepts a run if and only there exists a state $s \in F$ which occurs infinitely often throughout the run.

To perform LTL model checking, the Kripke structure is translated into a Buchi Automaton. The LTL specification is also translated to a Buchi Automaton. Let $\mathcal{L}(\mathcal{M})$ denote the language of the Buchi automaton formed by converting a Kripke structure $\mathcal{M}$ into its Buchi automaton representation. Let $\mathcal{L}(\psi)$ denote the language of the Buchi automaton by converting the LTL specification into its Buchi Automaton. We can then decide the model checking problem by deciding if

$$\mathcal{L}(\mathcal{M}) \subseteq \mathcal{L}(\psi)$$

or equivalently if

$$\mathcal{L}(\mathcal{M}) \cap \overline{\mathcal{L}(\psi)} = \emptyset$$

where the over-line denotes the compliment of the language. Buchi Automaton are known to be closed under intersection and compliment, and deciding subsets intersections and compliments are decidable problems. []

Converting a Kripke Structure can be done as follows. Consider Kripke structure $\mathcal{M} = (S, R, S_0, L)$. Then we will construct Buchi Automaton $\mathcal{B}$ as follows:

1. $\Sigma = \mathcal{P}(AP)$
2. $Q = S \cup \{i\}$
3. For $s, s' \in S$, $\alpha \in L(s')$, $((s, \alpha), s') \in \delta$ if and only if $(s, s') \in R$ and $\alpha = L(s')$. Furthermore, $((i, \alpha), s) \in \delta$ if and only if $s \in S_0$ and $\alpha = L(s)$.
4. $q_0 = i$
5. $F = \{i\} \cup S$

There are several approaches to converting the LTL formula to Buchi Automaton. The most well known are in: [32] [44] [78], which are omitted from this thesis. Furthermore, any other logic that can be converted to a Buchi Automata can also use the same procedure.

3.7 IC3

IC3, Incremental Construction of Inductive Clauses for Indubitable Correctness, was invented proposed by Aaron Bradley in 2011 in [18]. IC3 is a symbolic model checking algorithm that makes several calls to a SAT Solving subroutine. IC3 was not the first algorithm that used a SAT solver. However, it was the first model checking algorithm that did not *unroll* the transition system of the model fully. As a consequence, the formulas sent to the SAT solver are much smaller in comparison to other SAT based techniques.

The lack of transition system *unrolling* makes the use of IC3 very appealing instead of other model checking algorithms. However, the IC3 Model checking algorithm as originally presented checks for invariants properties, commonly known as *safety* properties. Informally, a *safety* property says that something bad will never happen. Another common type of property is a *liveness* property which IC3 can not analyze. Informally, a *liveness* property says that something good will eventually happen.

Consider a Kripke structure $\mathcal{M} = (S, R, S_0, L)$ with a safety property ψ. IC3 computes an *over approximation* of reachable states where ψ is satisfied. The semantics of a "state" can be a bit confusing in the literature. Traditionally, a state is just a node in a transition graph. However, it can be commonly referred to as an element of the power set of the atomic propositions of the Kripke structure. The remaining details of IC3 are cut from the thesis but can be found in [18].

3.8 Symmetry in Model Checking

In this section we will introduce how to incorporate symmetry in model checking. While the main results in this work do not directly relate with symmetric model checking, it is later hypothesized and reserved for future works.

Definition 19. *A group G is a set elements with a binary operation, notated as multiplication, such that:*

1. *Multiplication is associative, that is, $a(bc) = (ab)c$*
2. *There is a element $e \in G$, the identity, such that forall $a \in G$, $ae = a$.*
3. *For each element $a \in G$, there exists an inverse of the element, notated as $a^{-1} \in G$, such that $aa^{-1} = e$*

Definition 20. *A subset $H \subseteq G$ of a group G, if H forms a group under the multiplicative operator of G.*

Definition 21. *A permutation of a set S is a one-to-one onto function $f : S \to S$*

Theorem 8. *Let S be a set, and G be the set of all permutations of S. Then G is a group under function composition. This is commonly notated as $G = \mathrm{Sym}(S)$ read as the symmetry group.*

Definition 22. *Let S be a set, and $H \subseteq \mathrm{Sym}(S)$ be a subgroup. H is permutation group of S.*

Definition 23. *Let G be a permutation group of set S. Let $s \in S$. The orbit of s on G, is $\theta(s) = \{t | \exists f \in G, f(s) = t\}$*

Definition 24. *Let $\theta(s)$ be an orbit under permutation group G, set S, and permutation f. Let the function rep() map an orbit to an element of the orbit.*

Definition 25. *Let $\mathcal{M} = (S_0, S, R, L)$ be a Kripke Structue. Then G be a permutation group over S under permutation f. G is an automorphism of the $\mathcal{M}$ if G preserves R. Formally,*

$$\forall s_1 \in S, \forall s_2 \in S, ((s_1, s_2) \in R \implies (f(s_1, f(s_2)) \in R)$$

Definition 26. *Let $\mathcal{M} = ()$ be a kripke structure and G be an automorphism group of $\mathcal{M}$. The* quotient structure *$\mathcal{M}_G = (S_G, R_G, L_G)$ is defined as follows:*

1. *$S_G + \{f(s) | s \in S\}$ be the set of orbits over the states in S*
2. *$R_G = \{(f(s_1), f(s_2)) | (s_1, s_2) \in R\}$*
3. *$L_G(f(s)) = L(rep(f(s)))$*

Theorem 9. *For any formula in $f \in CTL*$, $\mathcal{M}, s \models f \iff \mathcal{M}_G, f(s) \models f$.*

```
$reached = \emptyset$
$unexplored = \emptyset$
for $s \in S_0$ do
    $reached = reached \cup \xi(s)$
    $unexplored = unexplored \cup \xi(s)$
end for
while $unexplored \neq \emptyset$ do
    Pick $s \in unexplored$
    if $s \not\models \psi$ then
        return $False$
    end if
    $unexplored = unexplored \setminus s$
    for Successor state $q$ of $s$ in $T$ do
        if $\xi(q) \notin reached$ then
            $reached = reached \cup \xi(s)$
            $unexplored = unexplored \cup \xi(s)$
        end if
    end for
end while
return $True$
```

Figure 3.3: The above algorithm uses 9 is an extension of a state space exploration algorithm introduced in [41] for model checking

Theorem 10. *Computing $\xi(s)$ is hard as the Graph Isomorphism Problem.*

This proof is omitted from the thesis, but can be found in [24].

Chapter 4

Network Models and Parametrized Models

4.1 Network Transition Systems

Up to this point in the thesis, we have discussed model checking of a single transition system that resembles a single computational structure. This modeling style fails to apply to several practical models that are in need of automated verification. In this section, we will formulate a transition system such that it is representative of several transition systems running concurrently with each other.

Definition 27. *A Process Template $p = (S_0, S, T, L, AP)$ is defined as:*

1. *S_0 is the set of initial states*
2. *S is the set states*
3. *$T \subseteq S \times S$ is the set of transitions.*
4. *$L : S \to AP$ is the labeling function*
5. *AP is the set of atomic propositions.*

A process template has a nearly identical formulation to a Kripke Structure. Conceptually, two process templates are unique if the semantics of the modeled transition systems differ in functionality. For example, consider the concurrent algorithm paradigm where there are several *worker* nodes performing a task using the same algorithm and a single *master* who organizes and gathers the work of the *worker* nodes all running a shared piece of computer code. The worker nodes would all have equivalent process templates, and the master node would have a unique process template. In the case of a mutual exclusion protocol, all the process templates would be equivalent.

We will next formulate a concurrent transition system, called a process. As it is a concurrent process, it is formulated with respect to other concurrent processes that further dictates the transition set. We will use the *ordered list* mathematical structure to represent multiple process templates. An ordered list is used instead of a set as it is indexable and allows for multiple equivalent processes. Notationally, for an ordered list $\overline{P}$ of processes, we will use a subscript, $\overline{P}_i$, to denote the ith process in the list. A superscript over a member of the tuple in a process's definition, for example, $S^{\overline{P}_3}$, will denote the set of states in the third process in $\overline{P}$.

Definition 28. *A (Concurrent) Process within an ordered list of other process $\overline{PP}$ is a six tuple $p = (S_0, S, T, L, AP, \overline{P})$ where:*

1. *S_0, S, L, AP are the same as in Definition 27*
2. *$\overline{P} \subseteq \overline{PP}$ is an ordered list of process that effect the transition*
3. *$T \subseteq (S, (S^{\overline{P}_1} \times S^{\overline{P}_2} \times ... \times S^{\overline{P}_n}), S)$ is a total transition set* [1]

The notable change in the process transition system compared to the previously studied transition systems is that T is now a three tuple instead of a two tuple. The first and last elements are, as before, the starting point of the transition and where it transitions too. The second element denotes the current states of $\overline{P}$ and will make the transition as before but with the condition that all processes in $\overline{P}$ are in the states as defined by the middle element. An example is provided in the following section.

We will next define a network over several processes. We will define it in two different ways for two different models of process synchronization. The first will be the *synchronous* model of process synchronization where each transition system makes a transition simultaneously. The second is the *asynchronous* model of process synchronization where any nonempty subset of the processes make a transition while the compliment does not.

Definition 29. *A Synchronous Network $N = (\overline{P}, S_0, S, T_{Sync}, L, AP)$ is a six tuple where:*

[1] As total is a property of a function and this is a set of three tuples and are abusing semantics. We infer that the first two elements of the tuple, $S \times S^{\overline{P}_1} \times S^{\overline{P}_2} \times ... \times S^{\overline{P}_n}$ and S, has the total property.

1. $\overline{P}$ *is an ordered list of processes*

2. $S_0 = S_0^{\overline{P}_1} \times S_0^{\overline{P}_2} \times ... \times S_0^{\overline{P}_n}$

3. $S = S^{\overline{P}_1} \times S^{\overline{P}_2} \times ... \times S^{\overline{P}_n}$

4. $T \subseteq ((S^{\overline{P}_1} \times S^{\overline{P}_2} \times ... \times S^{\overline{P}_n}), (S^{\overline{P}_1} \times S^{\overline{P}_2} \times ... \times S^{\overline{P}_n}))$ *is the transition set of the network, where* $((s_1, s_2, ...), (s'_1, s'_2, ..)) \in T$ *if and only if for all* $i \in [n]$*, there exists* $(s_i, \alpha, s'_i) \in T^{\overline{P}_i}$*, where* $\alpha = (\tau_1, \tau_2, ..., \tau_m)$ *and for each* $j \in [m]$ $\tau_j \in S^{\overline{P}_k}, \tau_j = s_k$ *for some* k*.*

5. $L(S^{\overline{P}_1} \times S^{\overline{P}_2} \times ... \times S^{\overline{P}_n}) = \bigcup\limits_{i \in [n]} L^{\overline{P}_i}(S^{\overline{P}_i})$

6. $AP = \bigcup_i AP^{\overline{P}_i}$

To elaborate on the notation, n is the size of $\overline{P}$, m is the size of $\overline{PP}$ for a specific process, $\alpha = (\tau_1, \tau_2, ...)$ is the tuple of current states of the neighboring processes to make the transition and each τ_i denotes the specific state.

In a network, there is a state for every element in the Cartesian product over each process's state. The transition set, T, is from state to state, like the Kripke structure but unlike the process, such that the requirement of a concurrent process in Definition 28 by only including the appropriate transitions where the neighboring processes are in the expected state.

Theorem 11. *A Synchronous Network transition system is total. Furthermore, it is deterministic if and only if every process in* $\overline{P}$ *is deterministic*

Proof. The total requirement is persistent on all seen transition systems as it enforces infinite paths. A network is is a total transition set as it implied from the result that each process has a total transition set. If it were not total, then there would exists $s \in S^{\overline{P}_1} \times S^{\overline{P}_2} \times ... \times S^{\overline{P}_n}$ where a transition is not defined. This would mean that for $\overline{P}_1$, there is no $(s_1, \alpha, *) \in T$, for any $* \in S^{\overline{P}_1}$, which is total. If $\overline{P}_i$ is deterministic, then $T^{\overline{P}_i}$ forms a function by $S \times S^{\overline{P}_1} \times S^{\overline{P}_2} \times ... \times S^{\overline{P}_n} \to S$. Consider the mapping of T where $T(s_1, s_2, ..., s_n)$ maps to (s'_1, s'_2, s'_n) and (s''_1, s''_2, s''_n) where $(s'_1, s'_2, s'_n) \neq (s''_1, s''_2, s''_n)$. Then $T^{\overline{P}_i}((s_i, \alpha)) = s^*$ where $s^* = s'_i$ and $s^* = s''_i$ and transitively $s'_i = s''_i$. □

Definition 30. *A Asynchronous Network transition system* $N = (\overline{P}, S_0, S, T_{Async}, L, AP)$ *is a six tuple where:*

1. *$S_0 = S_0^{\overline{P}_1} \times S_0^{\overline{P}_2} \times ... \times S_0^{\overline{P}_n}$*

2. *$S = S^{\overline{P}_1} \times S^{\overline{P}_2} \times ... \times S^{\overline{P}_n}$*

3. *$T_{Async} \subseteq ((S^{\overline{P}_1} \times S^{\overline{P}_2} \times ... \times S^{\overline{P}_n}), (S^{\overline{P}_1} \times S^{\overline{P}_2} \times ... \times S^{\overline{P}_n}))$ is the transition set of the network, where if $((s_1, s_2, ..., s_{n-1}, s_n), (s'_1, s'_2, ..., s'_{n-1}, s'_n) \in T_{Sync}$ then*

$$
\begin{gathered}
((s_1, s_2, ..., s_{n-1}, s_n), (s_1, s_2, ..., s_{n-1}, s'_n) \in T_{Async} \\
((s_1, s_2, ..., s_{n-1}, s_n), (s_1, s_2, ..., s'_{n-1}, s_n) \in T_{Async} \\
((s_1, s_2, ..., s_{n-1}, s_n), (s_1, s_2, ..., s'_{n-1}, s'_n) \in T_{Async} \\
... \\
((s_1, s_2, ..., s_{n-1}, s_n), (s_1, s'_2, ..., s_{n-1}, s_n) \in T_{Async} \\
((s_1, s_2, ..., s_{n-1}, s_n), (s_1, s'_2, ..., s_{n-1}, s'_n) \in T_{Async} \\
... \\
((s_1, s_2, ..., s_{n-1}, s_n), (s_1, s'_2, ..., s'_{n-1}, s'_n) \in T_{Async} \\
((s_1, s_2, ..., s_{n-1}, s_n), (s'_1, s_2, ..., s_{n-1}, s_n) \in T_{Async} \\
... \\
((s_1, s_2, ..., s_{n-1}, s_n), (s'_1, s'_2, ..., s'_{n-1}, s'_n) \in T_{Async}
\end{gathered}
$$

4. *$L(S = S^{\overline{P}_1} \times S^{\overline{P}_2} \times ... \times S^{\overline{P}_n}) = \bigcup_{i \in [n]} L^{\overline{P}_i}(S^{\overline{P}_i})$*

5. *$AP = \bigcup_i AP^{\overline{P}_i}$*

The asynchronous network differs from a synchronous network by its transition set. The transition set of the synchronous network is a subset of the transition set of the asynchronous set by further considering any nonempty subset of processes making a transition while the complement idles. Without the non-empty requirement, there will always be a loop around every state, and most liveness properties fail.

Theorem 12. *An Asynchronous Network a total and transition system is non-deterministic.*

Proof. As the Synchronous transition system set is total and since $T_{Sync} \subseteq T_{Async}$, then T_{Async} must be total. Furthermore, the mapping formed by T_{Async} is nondeterministic as any state $(s_1, ..., s_n)$ maps to $2^n - 1$ things. □

Definition 31. *An infinite family of systems $\mathcal{F} = \{N_i\}_{i=1}^{\infty}$ is a sequence of network models where N_i denotes a either a synchronous or asynchronous network of i processes.*

For this thesis, we will assume that an infinite family is composed of a single process template. However, this can be easily generalized by including another parameter $j \in \mathbb{N}$ for a second process template and so on.

Definition 32. *The parametrized model checking problem (PMCP) asks $\forall i \in \mathbb{N}$, does $N_i \models \psi$?*

Theorem 13. *PMCP, is in general, undecidable.*

The proof is omitted as it is long and technical and given in [].

4.2 Mutual Exclusion Modeling

In this section, we will model a mutual exclusion protocol using the process and network definitions defined in the previous section. Dijkstra proposed the problem mutual exclusion and proposed a solution for it in [28]. Dijkstra's mutual exclusion protocol is known to exhibit safety specifications but fails common liveness and fairness specifications.

Dijkstra's algorithm as originally stated is presented in Figure 4.1. When the procedure is invoked it has three stages. The first loop is the first phase, the second loop being the second phase, and critical section access. The procedure maintains two boolean arrays $b[], c[]$ and an integer value K which are readable and writable by all processes. If $b[i] = 0$, this denotes the process with $pid = i$ is in the first phase, second phase, or has access to the critical section. If $c[i] = 0$, this denotes the process with $pid = i$ is in the second phase or has access to the critical section. The K integer denotes the identifier of the process that is about to complete the first phase, in the second phase, or has critical section access.

We will first construct a process template describing Dijkstra's mutual exclusion protocol. For states, we will have one for each of the phases as described. One for the two loops(P_1, P_2), one denoting critical section access(CS), and an additional state where the procedure is not invoked(NT). When invoked, a process loops in phase 1 until there are no other detected processes in the second phase or critical section or if there was another process in phase one that wrote to K before it. Otherwise, it advances to the second phase. In the second phase, the process transitions to the critical section upon completing a loop through the $c[]$ array by checking if there is another process in the second phase or the critical section.

Figure 4.1: Dijkstra's Algorithm for Mutual Exclusion

```
procedure MUTUAL EXCLUSION($pid, b[], c[], K$)
    $b[pid] \leftarrow false$                                ▷ $L_i0$ (Phase 1)
    if $K \neq pid$ then                                     ▷ $L_i1$
        $c[pid] \leftarrow true$                             ▷ $L_i2$
        if $b[K]$ then                                       ▷ $L_i3$
            $K \leftarrow pid$
        end if
        $goto - L_i1$
    else
        $c[pid] \leftarrow false$                            ▷ $L_i4$ (Phase 2)
        for $j = 1$ to $N$ do
            if $j \neq pid$ and $\neg c[j]$ then
                $goto - L_i1$
            end if
        end for
        $CriticalSection()$
        $c[i] \leftarrow true$
        $b[i] \leftarrow true$
        Remainder of cycle 1
        $goto - L_i0$
    end if
end procedure
```

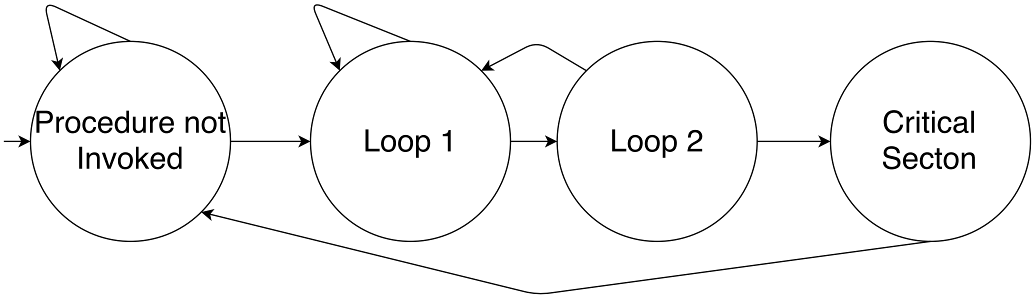

Figure 4.2: An example of a transition system of a process for a Dijkstra mutual exclusion algorithm

If one is found, it transitions back to the first phase, otherwise transitions to the critical section and then exits the procedure.

We will define a process template that represents the Dijkstra mutual exclusion protocol with the abstractions as described. The transition system is further illustrated pictorially in Figure 4.2.

1. $S_0 = \{NT\}$
2. $S = \{NT, P_1, P_2, CS\}$
3. $T = \{(NT, NT), (NT, P1), (P1, P1), (P1, P2), (P2, P1), (P2, CS), (CS, NT)\}$
4. $AP = \{p\}$, where p denotes critical section access.
5. $L = \{(NT, \{\})(P_1, \{\}), (P_2, \{\}), (CS, \{p\})\}$

Next, we will formulate a concurrent process of a Dijkstra Mutual Exclusion protocol with two processes communicating with each other. By definition, a process and process template only differ in its transition set. We will provide the transition set of process P_1. The process P_2 can be constructed analogously. It is pictorially represented in Figure 4.3.

$T^{P_1} = \{$

- $(NT, (NT), NT), (NT, (P_1), NT), (NT, (P_2), NT), (NT, (CS), NT),$
- $(NT, (NT), P_1), (NT, (P_1), P_1), (NT, (P_2), P_1), (NT, (CS), P_1),$
- $(P_1, (NT), P_2), (P_1, (P_1), P_1), ((P_1, P_1), P_2), (P_1, (P_2), P_1), (P_1, (CS), P_1),$
- $(P_2, (NT), CS), (P_2, (P_1), CS), (P_2, (P_2), P_1), (P_2, (CS), P_1),$
- $(CS, (NT), NT), (CS, (P_1), NT), (CS, (P_2), NT), (CS, (CS), NT)$

$\}$

We will next provide the formulation for both a synchronous and asynchronous network model for the Dijkstra protocol with two processes. In the transition relation, we will use a super script $*$ to denote a transition that is only present in the asynchronous network model.

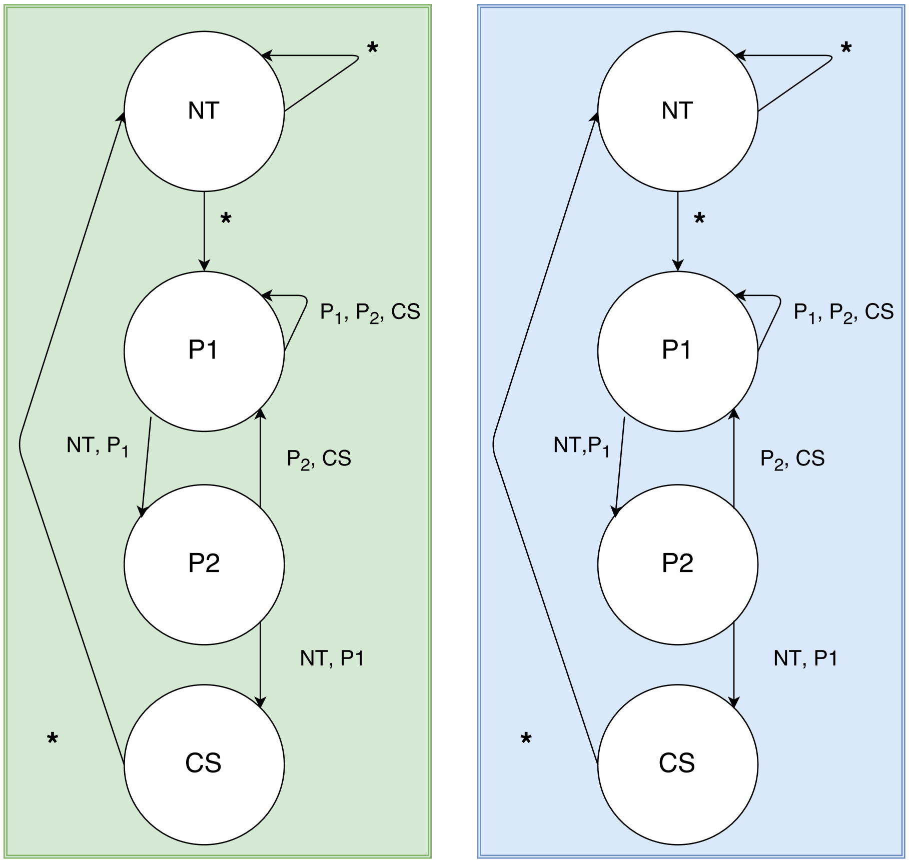

Figure 4.3: An illustration of two concurrent processes running Dijkstra's protocol. The label on the transition, denotes the required state of the process it is running concurrently with. The * token on the transition label denotes any state.

1. $S_0 = \{(NT, NT)\}$

2. $S = \{(NT, NT), (NT, P_1), (NT, P_2), (NT, CS), (P_1, NT), (P_1, P_1), (P_1, P_2),$ $(P_1, CS), (P_2, NT), (P_2, P_1), (P_2, P_2), (CS, NT), (CS, P_1), (CS, P_2), (CS, CS)\}$

3.
    - $((NT, NT), (NT, NT)), ((NT, NT), (NT, P_1)),$ $((NT, NT), (P_1, NT)), ((NT, NT), (P_1, P_1)),$
    - $((NT, P_1), (NT, P_1))^*, ((NT, P_1), (NT, P_2)),$ $((NT, P_1), (P_1, P_1))^*, ((NT, P_1), (P_1, P_2)),$
    - $((NT, P_2), (NT, P_2)), ((NT, P_2), (NT, CS)),$ $((NT, P_2), (P_1, P_2))^*, ((NT, P_2), (P_1, CS)),$
    - $((NT, CS), (NT, CS))^*, ((NT, CS), (NT, NT)),$ $((NT, CS), (P_1, CS))^*, ((NT, CS), (P_1, NT)),$
    - $((P_1, NT), (P_1, NT))^*, ((P_1, NT), (P_1, P_1))^*,$ $((P_1, NT), (P_2, NT)), ((P_1, NT), (P_2, P_1)),$
    - $((P_1, P_1), (P_1, P_1)), ((P_1, P_1), (P_1, P_2)),$ $((P_1, P_1), (P_2, P_1)), ((P_1, P_1), (P_2, P_2)),$
    - $((P_1, P_2), (P_1, P_2))^*, ((P_1, P_2), (P_1, CS)),$
    - $((P_1, CS), (P_1, CS)))^*, ((P_1, CS), (P_1, NT)),$
    - $((P_2, NT), (P_2, NT)), ((P_2, NT), (P_2, P_1))^*, ((P_2, NT), (CS, NT)),$ $((P_2, NT), (CS, P_1)),$
    - $((P_2, P_1), (P_2, P_1))^*, ((P_2, P_1), (CS, P_1)),$
    - $((P_2, P_2), (P_2, P_1))^*,$ $((P_2, P_2), (P_1, P_2))^*, ((P_2, P_2), (P_1, P_1)),$
    -
    - $((CS, NT), (CS, NT))^*, ((CS, NT), (CS, P_1))^*,$ $((CS, NT), (NT, NT)), ((CS, NT), (NT, P_1)),$
    - $((CS, P_1), (CS, P_1))^*, ((CS, P_1), (NT, P_1)),$
    -
    - $((CS, CS), (CS, NT))^*,$ $((CS, CS), (NT, CS))^*, ((CS, CS), (NT, NT))$

4. $L = \{((NT, NT), \{\}), ((NT, P_1), \{\}), ((NT, P_2), \{\}), ((NT, CS), \{p^{P_2}\}), ((P_1, NT), \{\}),$
$((P_1, P_1), \{\}), ((P_1, P_2), \{\}), ((P_1, CS), \{p^{P_2}\}), ((P_2, NT), \{\}), ((P_2, P_1), \{\}),$
$((P_2, P_2), \{\}), ((P_2, CS), \{p^{P_2}\}), ((CS, NT), \{p^{P_1}\}), ((CS, P_1), \{p^{P_1}\}), ((CS, P_2), \{p^{P_2}\}),$
$((CS, CS), \{p^{P_1}, p^{P_2}\})$
$\}$

Chapter 5

Structural Mutable Modeling

5.1 Network Structures and Topologies

Definition 33. *A topology $\mathcal{T}$ of order n is an $n-$tuple of sets of components of unique types.*

A well known topology is a graph. A graph has two types of components, vertices, and edges. In traditional graph theory, it is defined as a two tuple of a set of vertices and edges.

We will next model the topology of a network running a mutual exclusion protocol with multiple critical sections and users. We want to be able to model user to user communication as in the protocol users can read and write to shared memory. Furthermore, we would also like to a representation for a users capability to access a critical section. Therefore, we will model the mutual exclusion as $\mathcal{T} = (N, E, C, L)$ where:

1. N is the set of nodes, denoting users
2. E is the set of (undirected) edges of the form (u, v) for $u, v \in N$ to denoting the ability of process interaction
3. C is the set of critical sections
4. L is the set of links of the form u, c for $u \in N, c \in C$ denoting processes capability of accessing a critical section.

The corresponding topology to the two process example used previously in Chapter 4.2 with one critical section and two users would be formulated as:

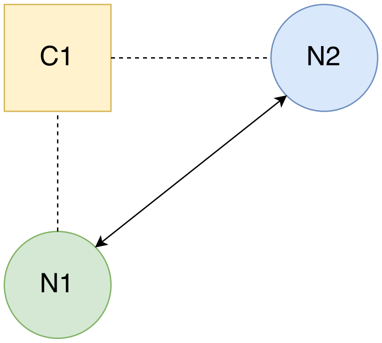

Figure 5.1: Topology of the two process mutual exclusion protocol

1. $N = \{n_1, n_2\}$
2. $E = \{(n_1, n_2)\}$
3. $C = \{c_1\}$
4. $L = \{(n_1, c_1), (n_2, c_2)\}$

The topology is illustrated in Figure 5.1. The circles denote nodes illistrating processes as in Figure 4.3. The solid line denotes the edge between the two nodes. The dotted line denotes the edge between the nodes and the critical section.

Now that we have established a notion of concurrent transition systems and topology we will combine these notions of defining a structured network model.

Definition 34. *A Structured Network Model is a 3-tuple such that $\mathcal{M} = (N, \mathcal{T}, \rho)$, where:*

1. *N is either a synchronous or asynchronous network model.*
2. *$\mathcal{T}$ is the topology of the model*
3. *$\rho : P \to \mathcal{T}$ is a mapping of the processes of N to a topological component.*

We have seen both an example of N and $\mathcal{T}$ for the mutual exclusion problem. ρ is a mapping from the node set in the topology to the process set in the network, and in the example $\rho = \{(\overline{P}_1, n_1), (\overline{P}_2, n_2)\}$ where $\overline{P}_1, \overline{P}_2$ denote the two processes.

There has been no discussion of the specification of the mutual exclusion problem to this point. Previously we stated that Dijkstra's mutual exclusion protocol has neither fairness nor liveness properties in both synchronous and asynchronous network models. One bug can be visualized by reflecting on Figure 4.3 and observing how if two processes are both in state P_2 to which they can both revert to P_1 and continue in this fashion.

However, the safety specification, which no two processes can both be in a critical section simultaneously, is true in this protocol. We can express this in the CTL logic with $AG(\neg p^{\overline{P}_1} \lor p^{\overline{P}_2})$. However, this specification is programmatic, what must be true about the topology? In the programmatic abstraction, we asserted that all the variables of the algorithm $b[], c[], K$ are accurately maintained and readable and writable from the processes in the network. With a structured model, we can say this a bit more precisely. Since each process running on a node is attempting to obtain access to a critical section, we can verify that the there is a link between the node and the critical section in the topology. Furthermore, to assure that the variables are maintained as asserted by the process, we can verify that every node two nodes that have a link to a common critical section share an edge. For the two process mutual exclusion example we can state this as:

$$p = ((n_1, c_1) \overset{?}{\in} C) \land q = ((n_2, c_1) \overset{?}{\in} C) \land r = ((n_1, n_2) \overset{?}{\in} E)$$

To accommodate this, we will later assume every specification ψ of a structured network model is of the form

$$\psi = \psi_T \land \psi_P$$

where ψ_P denotes the program specification and ψ_T denotes the topological specification.

Definition 35. *The Structured Model Checking Problem asks if*

$$\mathcal{M} \models \psi \iff (N \models \psi_P) \land (\mathcal{T} \models \psi_T)$$

5.2 Mutation

(Deterministic) Mutation

Definition 36. *A mutation ϕ is an algorithm that takes as input a structured model and either returns a structured model in a fixed number of steps or rejects the input. On each algorithmic step, it:*

1. *Adds/Removes a fixed number of components to $\mathcal{T}$.*
2. *(and/or) Adds/Removes a fixed number of process to N.*

Definition 37. *A nondeterministic mutation ϕ is a terminating algorithm that takes as input a structured model and either returns a structured model or rejects the input. On termination,*

1. *A fixed number of additions/removals of topological components are made to $\mathcal{T}$.*
2. *And/Or a fixed number of process additions/removals to N.*

Notationally, for a structured model $\mathcal{M}$, and mutation ϕ of n steps, $\phi(\mathcal{M})$ denotes the model that is returned. Furthermore, for $i \leq n$, $\phi_i(\mathcal{M})$ denotes the mutated model at the ith step of the mutation and $\phi_0(\mathcal{M}) = \mathcal{M}$. If ϕ is nondeterministic a super script will be used to denote a uniqueness of a structured model after so many steps.

Definition 38. *A Dynamic Model is a three tuple $\mathcal{D} = (\mathcal{M}_0, \mathcal{P}, \Phi)$ where:*

1. *$\mathcal{M}_0$ is the structured network model.*
2. *$\mathcal{P}$ is the set of process templates used.*
3. *Φ is the set mutations that add/remove elements of $\mathcal{T}$ and processes $\overline{P}$ of the network of the structured model.*

Definition 39. *A directly reachable model of a Dynamic Model is defined inductively as:*

1. *$\mathcal{M}_0$ is directly reachable*
2. *If $\mathcal{M}$ is directly reachable and $\phi \in \Phi$, $\phi(\mathcal{M})$ is directly reachable.*

Definition 40. *A reachable sub-model that are the models achieved through the mutation process. Formally, for dynamic model $\mathcal{D}$, mutation $\phi \in \Phi$ and directly reachable model $\mathcal{M}$, $\phi_i(\mathcal{M})$ is a sub-model for $0 < i \leq n$.*

To further reason about mutations and dynamic models, we will consider mutation sets of the following form:

1. *Instantaneously Mutating Dynamic Models.* Instantaneous mutation is used to denote when the model undergoes a *primitive* structural change.

Definition 41. *A primitive mutation is a mutation that does one of the following:*

(a) Adds a single component into the topology.

(b) Removes a single component from the topology.

(c) Adds a single process into the network.

(d) Removes a single process from the network.

Definition 42. *An $\mathcal{D}$ is an instantaneous dynamic model if every ϕ is primitive.*

$$\mathcal{M} \overset{\phi}{\to} \mathcal{M}'$$

Here $\mathcal{M}$ is a directly reachable model, and as a consequence from ϕ, $\mathcal{M}'$ is a directly reachable model. There are no sub-models in Instantaneous Mutation as ϕ is executed in a single step as it is a single primitive change.

2. *Path Like Mutating Dynamic Models.* In contrast,

Definition 43. *$\mathcal{D}$ is a Path Like Mutating Dynamic Model if every ϕ is deterministic and there exists at least one mutation that is non-primitive.*

$$\mathcal{M} \overset{\phi}{\to} \phi_1(\mathcal{M}) \overset{\phi}{\to} \phi_2(\mathcal{M}) \overset{\phi}{\to} ... \overset{\phi}{\to} \phi_m(\mathcal{M}) \overset{\phi}{\to} \mathcal{M}'$$

Here $\mathcal{M}$ and $\mathcal{M}'$ are directly reachable models. For every ith step of the mutation procedure, denoted as $\phi_i(\mathcal{M})$, is a sub-model for $i \in [1, n]$.

Claim 1. *The total number of sub-models generated by path mutation is linear in the number of structurally primitive changes.*

While this claim may seem fairly obvious, it is used to distinguish the following much more complex paradigm of mutation.

3. *Branch Like Mutating Dynamic Model*

Definition 44. *$\mathcal{D}$ is a Branch Like Mutating Dynamic Model if there exists at least one ϕ that is non-deterministic.*

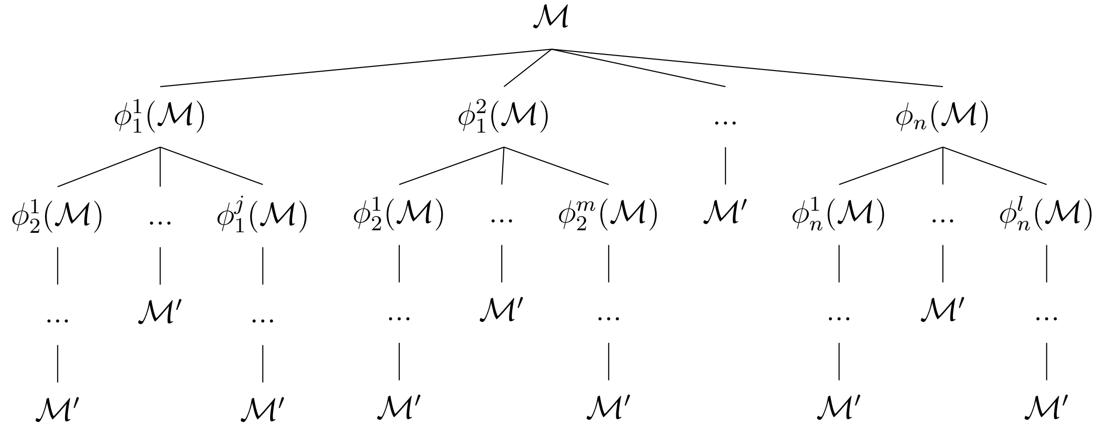

Figure 5.2: An illustration of branching mutation.

As the mutation is well defined, it will always have the same result, but the path to which it gets there is nondeterministic.

Theorem 14. *The total number of sub-models generated by branching mutation is $\mathcal{O}(n^n)$, with n being the number of structural primitive operations.*

Proof. In a nondeterministic setting, a known set C of n singular changes(addition/removal of a component/process) is applied to a structured network. The order these steps take place, groups of changes applied simultaneously, and the steps of the mutation process is not known, with each unknown being a valid reachable sub-model of the dynamic network. We will claim that the number of sub-models is equal to the size of the set of all permutations of every possible partition of the set C, denoted as $\mathbb{C}(C)$. We will compute the size using Sterlings number of the second kind, which gives us the number of ways to partition a set of n items into k nonempty subsets, denoted as $\mathcal{S}_k^n$. The number of permutations can then be computed by the factorial function. We can calculate the size of $\mathbb{C}(C)$ then as

$$\mathbb{C}(C) = \sum_{k=1}^{n} \mathcal{S}_n^k k! = \sum_{k=1}^{n} \sum_{j=0}^{k} (-1)^{k-j} \binom{k}{j} j^n \in \mathcal{O}(n^n) \tag{5.1}$$

□

Chapter 6

Dynamic Model Checking

6.1 Safety Proof Rules

Definition 45. *Denote* $\mathbb{M}(\mathcal{D})$ *as the set of all reachable models in a Dynamic Model* $\mathcal{D}$.

Definition 46. *A specification* Ψ *of a dynamic model* $\mathcal{D}$ *is a function that maps every* $\mathcal{M} \in \mathbb{M}(\mathcal{D})$ *to its structured network model specification* $\Psi(\mathcal{M}) = \psi_P \wedge \psi_T$.

In this section, we will describe the requirements of proof to verify specifications of dynamic models. We assert the specification Ψ of a dynamic model $\mathcal{D}$, for each reachable model, it maps to a conjunction formed by Ψ_P and Ψ_T, where Ψ_P denotes the programmatic specification of the model and Ψ_T denotes the topological specification of the model.

Definition 47 (Dynamic Model Checking)**.** *The Dynamic Model Checking Problem asks* $\forall \mathcal{M} \in \mathbb{M}(\mathcal{D})$*, does* $\mathcal{M} \models \Psi(\mathcal{M})$*, denoted as* $\mathcal{D} \models \Psi$.

In the case of the mutual exclusion model, Ψ_P denotes that no two processes will be in the same critical section simultaneously. Ψ_T in the mutual exclusion model denotes that any two nodes which share a mutex have an edge between them so they can properly communicate and that each node that has a process that attempts to access a critical section has a link to that critical section.

The program code running on each process may be correctly written. Yet, it may not properly communicate with all nodes due to a mis-configured topology and hardware. Consequently, correctness certificates could be generated for incorrect programmatic models.

Definition 48 (Graph Representation of Dynamic Models)**.** *Let $G = (V, E)$ be the graph representation of a dynamic model $\mathcal{D} = (\mathcal{M}_0, \mathcal{P}, \Phi)$. The set of vertices V is defined as the directly reachable models. There exists a directed edge $(u, v) \in E$ if and only if there exists a $\phi \in \Phi$ such that $\phi(u) = v$.*

Definition 49. *(Proof Rules of Safety Properties)*

1. *Verify $\mathcal{M}_0 \models \Psi_T(\mathcal{M}_0)$ (Initial Topology)*
2. *Verify $\Psi_T(\mathcal{M}_0), \mathcal{M}_0 \models \Psi_P(\mathcal{M}_0)$ (Initial Programmatic)*

   The first step is to verify the topology of the initial model indeed upholds the topological aspect of Ψ. If not, then the dynamic model does not satisfy the specification. Now, granted that it indeed does uphold the specification, verify the programmatic part of the specification on the initial model.

3. *Verify $\phi_i(\mathcal{M}), \Psi_T(\mathcal{M}) \models (\forall i, \Psi_T(\phi_i(\mathcal{M})))$, for every ϕ, directly reachable $\mathcal{M}$. (Reachable Topology)*
4. *Verify $\mathcal{M}, \Psi_T(\mathcal{M}) \models \Psi_P(\mathcal{M})$, for every reachable $\mathcal{M}$ (Reachable Programmatic)*

   For 3, we need to show that for every directly reachable model $\mathcal{M}$, asserting the topological specification is upheld on start of a mutation process, the topological specification is upheld throughout and on termination of the mutation process for every permitted mutation on every directly reachable model. For 4, for every reachable model $\mathcal{M}$, given the topology of $\mathcal{M}$ satisfies $\Psi(\mathcal{M})_T$, the programmatic specification is upheld.

Theorem 15. *$\mathcal{D} \models \Psi$ if and only if the above proof rules are satisfied for a safety property Ψ.*

Proof. We will prove soundness of the above proof rules by structural induction by showing that every reachable model upholds Ψ given (1)-(4) are true. The base case that $\mathcal{M}_0 \models \Psi(\mathcal{M})$ is a direct result from (1) and (2) since $\Psi(\mathcal{M}_0) = \Psi_T(\mathcal{M}_0) \wedge \Psi_P(\mathcal{M}_0)$ and (1) gives us $\Psi_T(\mathcal{M}_0)$ is true and (2) gives $\Psi_P(\mathcal{M}_0)$ is true. Inductively let $\mathcal{M}$ be a directly reachable model from $\mathcal{M}_0$ and mutation set Φ. To complete the proof we need to show that all resulting sub-models when applying any $\phi \in \Phi$ upholds its specification and every directly reachable model from any ϕ also upholds its specification. By inductive hypothesis $\Psi(\mathcal{M})$ is true, hence both $\Psi_T(\mathcal{M})$ and $\Psi_P(\mathcal{M})$ is true. Hence, (3) gives us all sub-models over every mutation upholds the topological specification. Furthermore, the case $i = n$ gives us that the resulting directly reachable model for each mutation also upholds the topological specification. Since the topological specification is meet on all resulting models from $\mathcal{M}$, property (4) gives us the $\Psi_P(\mathcal{M})$ for each sub-model and directly reachable model from $\mathcal{M}$. □

Automizability: These proof rules are automizeable. We will assert Ψ_T is written in first order propositional logic, but Ψ_P is not restricted to a particular logic. (1) can be verified through a SAT Solver to confirm the initial topology correctness. (2) can be verified through a standard model checker as it would require solving the static model checking problem. (3) can be verified through a SAT Solver as done in (1) by verifying the topology for every reachable model, or a theorem prover to show that given any arbitrary model that has correct topology, $\Psi_T(\phi_i(\mathcal{M}))$ is invariant throughout the mutation process. (4) Asymptotically dominates steps (1) - (3). In this paper and formulation, in the general case, will require static model checking for every reachable model. We will later demonstrate how the use of parametric assumptions and briefly discuss the use of symmetry detection to overcome this setback.

Complexity: For the sake of complexity analysis we will assume $|\mathcal{D}|$ is finite and that the number of directly reachable models is bounded by a number n. The number of primitive operations over any given mutation is therefore of order $\mathcal{O}(n)$. Under primitive, non-primitive, and nondeterministic the number of sub-models generated under a single mutation is then $\mathcal{O}(1), \mathcal{O}(n), \mathcal{O}(n^n)$ for instantaneous dynamic model checking, path dynamic model checking, and branching dynamic model checking respectively. We will determine the total number of reachable models for each type by considering the graph representation (see definition 48). In the graph representation, $|V| \in \mathcal{O}(n)$ (directly reachable), and then $|E| \in \mathcal{O}(n^2)$ (occurrences of submodels spaces through a mutation process). Therefore, the total number of reachable models then $\mathcal{O}(n) + \mathcal{O}(1) = \mathcal{O}(n)$ for instantaneous dynamic model checking, $\mathcal{O}(n) + \mathcal{O}(n^3) = \mathcal{O}(n^3)$ for path dynamic model checking, and $\mathcal{O}(n) + \mathcal{O}(n^{n+2}) = \mathcal{O}(n^{n+2})$ for branching dynamic model checking. Verifying (1) takes $\mathcal{O}(2^n)$ by a DPLL/CDCL SAT Solver. (2) requires a single call to a static model checker which takes $\mathcal{O}(2^{\text{poly}(n)})$ for most common model checking logics. Under our assertions, (3) takes $|\Phi|$ calls to a theorem prover, which is solvable in $\mathcal{O}(2^n)$ through resolution based theorem provers for example. (4) requires a model checking call for each reachable model. This results in $\mathcal{O}(2^{\text{poly}(n)}n)$ for instantaneous dynamic model checking, $\mathcal{O}(2^{\text{poly}(n)}n^3)$ for path dynamic model checking, and $\mathcal{O}(2^{\text{poly}(n)}n^{n+2})$ branching dynamic model checking.

6.2 Dynamic Model Checking Analysis

Theorem 16 (Finding the Minimalist $\mathcal{M}_0$)**.** *Let $\mathcal{D} = (\mathcal{M}_0, \mathcal{P}, \Phi)$ and $G = (V, E)$ be its graph representation. Suppose there exists a cycle C in G such that $\mathcal{M}_0 \in C$. Let $\mathcal{D}' = (\mathcal{M}_0', \mathcal{P}, \Phi)$ with $\mathcal{M}_0' \in C$ and $\mathcal{M}_0'$ is directly reachable. Then*

$$\mathcal{D} \models \Psi \textit{ if and only if } \mathcal{D}' \models \Psi$$

Proof. We will prove this by showing the set of reachable models of both $\mathcal{D}, \mathcal{D}'$ are equivalent. Let S denote the set of reachable models of $\mathcal{D}$, and S' denote the set of reachable models of $\mathcal{D}'$. $S' \subseteq S$ as $\mathcal{M}_0'$ is directly reachable from $\mathcal{M}_0$ as the graph contains a path from $\mathcal{M}_0$ to $\mathcal{M}_0'$ due to the cycle and all of its reachable models must also be reachable then by $\mathcal{M}_0$ by applying the same sequence of mutations to construct the path. Analogously $S \subseteq S'$ as $\mathcal{M}_0$ is reachable from $\mathcal{M}_0'$ due to the cycle. Thus $S = S'$. Therefore if $\mathcal{D} \models \Psi$ then, Ψ is upheld for all reachable models in S and therefore S' so hence $\mathcal{D}' \models \Psi$. The converse argument is analogous. □

For a challenging to verify model $\mathcal{D} = (\mathcal{M}_0, \mathcal{P}, \Phi)$, we can use theorem 16 to reduce the complexity of proof rules (1)-(2). We can select the smallest and easiest to verify model as our $\mathcal{M}_0'$ that is cyclic to $\mathcal{M}_0$ and verify $\mathcal{D}'$ rather than $\mathcal{M}$. We suspect most dynamic models to have mutations that both enlarge and decrease the size of the model, so several cycles should exist. Computing the smallest $\mathcal{M}_0'$ can be done through a depth first search.

Theorem 17. *Let $\mathcal{D} = (\mathcal{M}_0, \Psi, \Phi)$. Let Φ' be any other set of mutators and $\mathcal{D} = (\mathcal{M}_0, \mathcal{P}, \Phi')$. If there exists a sequence of mutations $(\phi_1, \phi_2, ...)$ over mutations in Φ' such that $\circ_{j=1}^{n} \phi_j' \equiv \phi$ for each $\phi' \in \Phi'$ then*

$$\mathcal{D}' \models \Psi \text{ implies } \mathcal{D} \models \Psi$$

where $\circ_j^n \phi_j' \equiv \phi$ denotes a nested composition of mutations that result in an equal structural mutation.

Proof. We will first establish what exactly it means for two mutations ϕ, ϕ' to be equivalent. $\phi \equiv \phi'$ if and only if for any model $\mathcal{M}$, $\phi(\mathcal{M}) = \mathcal{M}' = \phi'(\mathcal{M})$, and for sub-model set S of ϕ, and sub-model set S' for ϕ', $S = S'$.

For $\phi \in \Phi, \phi' \in \Phi'$ we have that there exists a mutation sequence over Φ' for each mutation in Φ such that $\circ_j^n \phi_j' \equiv \phi$ that entails then the corresponding $S' = S_1' \cup ... \cup S_n'$ is equal to S, the sub-models generated from ϕ', ϕ respectively. From the above, for the set of reachable models $\mathcal{S}$ from $\mathcal{D}$, and $\mathcal{S}'$ from $\mathcal{D}'$ it follows that $\mathcal{S} \subseteq \mathcal{S}'$. Conditions (3),(4) are satisfied as $\mathcal{D}' \models \Psi$ and $\mathcal{D}'$ reachable space is contained within $\mathcal{D}'$. (1),(2) hold as they have the same initial model. The Theorem is proven by Theorem 15.

□

Theorem 17 breaks down complex mutation protocols compositionally to construct a smaller mutation set. By redefining the model with a compositional mutation set, we are over approximating the reachability of $\mathcal{D}$, as the resulting graph representation of $\mathcal{D}'$ has higher degree.

Definition 50 (The React Assumption)**.** *A network can "react" to dynamic change. For η_P being a programmatic invariant for $\mathcal{M}$. For a mutation ϕ, throughout the process of ϕ, η_p is invariant.*

Theorem 18. *Under the "react" assumption, a non-dynamic programmatic invariant η_P of a model $\mathcal{M}$ is invariant in a dynamic model $\mathcal{D}$. Furthermore, $\mathcal{M}_0 \models \Psi_P(\mathcal{M}_0)$ implies $\forall \mathcal{M}, \models \Psi_P(\mathcal{M})$.*

Proof. The react assumption and this theorem were stated and proved in [63]. The assumption has been adapted for this work. The original theorem proved the strongest invariant, to which we weakened to be any invariant.

□

Theorem 19. *Let $\mathcal{D} = (\mathcal{M}_0, \mathcal{P}, \Phi)$, and $\mathcal{D} = (\mathcal{M}'_0, \mathcal{P}, \Phi')$ is is the corresponding model referencing Theorems 16, 17. If $|\mathcal{M}'_0| \in \mathcal{O}(\log|\mathcal{D}|), |\Phi'| \in \mathcal{O}(\log|\Phi|)$. Assuming Definition 50 and by Theorem 18 hold, we can verify $\mathcal{D}$ in polynomial time, for all mutation models*

Proof. This theorem highlights the cutoff analysis by constructing a small enough dynamic model from the targeted model that is logarithmic in size cancels out the exponential run time of proof rules (1) - (3). (4) is given to us by Theorem 18.

□

Theorem 20. *Let N be a network, and $\mathcal{F} = \{N_i\}_{i \to \infty}$ be its infinite family . Let $\mathcal{D}$ be a dynamic model such that $\mathcal{M}_0 = (N_1)$ and $\phi \in \Phi$ is a primitive process addition mutation. If $\mathcal{D} \models \Psi$ then PMCP($\mathcal{F}$, Ψ_P) is decidable.*

Proof. For $PMCP(\mathcal{F})$ to be decidable it needs to be shown that for each i, $N_i \models \Psi_P$. To prove this, we need to show that for each i, $N_i \models \Psi_P$. Assume then that $\mathcal{D} \models \Psi$. Therefore, by our construction $\mathcal{M}_0 \models \Psi$ and further $\mathcal{M}_0 \models \Psi_P$ as $\Psi = \Psi_T \wedge \Psi_P$. On the converse of Theorem 15, we have that Ψ_P is true for each reachable model, and therefore each directly reachable model. By our construction of ϕ in $\mathcal{D}$ each N_i has a representative directly reachable model to which Ψ_P is satisfied. □

Chapter 7

Dynamic Model Checking Examples

7.1 Verifying a Dynamic Mutual Exclusion Protocol

In chapter 4.2 we introduced a Dijkstra protocol, and formulated a process template, a concurrent process, a synchronous/asynchronous network of two processes. In Chapter 5.1 we outlined the structured network model of the Dijkstra protocol.

We will define an example dynamic model $\mathcal{D} = (\mathcal{M}_0, \mathcal{P}, \Phi)$. For structured network model $\mathcal{M}_0$ we will use the one as described in Chapter 5.1. The process template set $\mathcal{P}$ is formed of a single template of the the Dijkstra protocol that is described in Chapter 4.2. The Mutation set is described in Figure 7.2.

In chapter 6.2 we considered the construction of a Dynamic model $\mathcal{D}'$ from a target model $\mathcal{D}$ whose verification implies the verification of $\mathcal{D}$. Further, we argued that the of $\mathcal{D}'$ also gives a decidability result for the PMCP problem. Our formulated model of the Dijkstra's protocol is representative of the construction of the small and tractable $\mathcal{D}'$, which then proves a large family of network families.

The use of NDET denotes a nondeterministic action that takes place in a nondeterministic order. The remaining actions are done iteratively and orderly.

Theorem 21. *Under Definition 50, the dynamic model $\mathcal{D}$ of Dijkstra's mutual exclusion protocol, $\mathcal{D} \models \Psi$.*

Proof. We will verify the proof rules of Definition 49 and then by Theorem 15, we will have shwon $\mathcal{D} \models \Psi$. For proof rule (1), $\mathcal{M}_0 \models \Psi_T$ from inspection as the appropriate edges and

Node Addition

1. **Rejection Criteria:** None.
2. Instantiate the new node

Critical Section Addition

1. **Rejection Criteria:** None.
2. Instantiate new critical section

Edge Addition

1. **Rejection Criteria:** None.
2. Instantiate new edge

Link Addition

1. **Rejection Criteria:** None
2. Instantiate new edge

Process Addition

1. **Rejection Criteria:** For process p, where $\rho(p) = (n, c)$, reject if $(n, c) \notin L$. Further for any p' with $\rho(p') = (n', c)$, reject if $(n, n') \notin E$
2. Instantiate process.

Figure 7.1: Removal Mutations for a Mutual Exclusion Protocol.

Node Removal

1. **Rejection Criteria:** For node n, Reject if there exists process p where $n \in \rho(p)$
2. Deinstantiate node

Critical Section Removal

1. **Rejection Criteria:** For critical section c, Reject if there exists process p where $c \in \rho(p)$
2. Deinstantiate Critical Section

Edge Removal

1. **Rejection Criteria:** For edge $e = (u, v)$, reject if there exists processes p_1, p_2, and $u \in p_1$ and $v \in p_2$ and there is critical section c with $c \in p_1$ and $c \in p_2$
2. Deinstantiate edge

Link Removal

1. **Rejection Criteria:** For link $\ell = (u, c)$ reject if there exists a process p with $\ell \in \rho(p)$.
2. Deinstantiate link.

Process Removal

1. **Rejection Criteria:** None.
2. Deinstantiate process.

Figure 7.2: Removal Mutations for a Mutual Exclusion Protocol.

links are already in place. For proof rule (2), we will claim that the structured model $\mathcal{M}_0$ upholds its programmatic specification from inspection. Further, for both $p^{\overline{P}_1}, p^{\overline{P}_1}$ to be true simultaneously they must both be in the the state denoted as CS. For them both to transition to CS they must both com from the $P2$ state. However, if both processes are in the $P2$ state they would both make the transition to $P1$ in the synchronous network model. In the asynchronous network model the only other alternative would be for one of the processes to idle in $P2$ while the other reverts back to $P1$. For proof rule (3), we will claim this to be true as the mutation procedures as described in figure 7.2 have appropriate rejection criteria. For proof rule (4), from (2) we have that ψ_P is an invariant in $\mathcal{M}_0$. Under definition 50 and Theorem 18 ψ_P is preserved through dynamic change and is true for each model. □

7.2 A Multicast Protocol

In this section, we will apply our built up analysis to a multicast security protocol. We will provide a dynamic model and argue its specification is meet as done in the previous section.

The most well known type of cryptographic protocol is a *one-to-one* communication scheme, where a network operator can communicate with one other source. The network operator may run several simultaneous instances of this kind of protocol, but the protocol itself only involves two sources. An example of a *one-to-one* communication scheme is the process of identification of a client to a network operator. In identification, a trusted network user proves to the network operator that he is who he claims to be by using private information only a trusted user could know. He does so without transmitting private information or information that could be used to deduce private information.

A multicast cryptography scheme is a *one-to-many* communication scheme in contrast to a *one-to-one* communication scheme. In the multicast setting, the protocol is defined such that the network operator communicates to several users at a time and users can communicate with other users in a secured manner. For example, consider a scenario when several users are trying to get access to a super computing cluster. Users would like to communicate amongst other users to observe scheduling and computing allocation. Further, the network operator would like to assure scheduling is done fairly and only trusted users have access. Multicast schemes have numerous applications across the Internet. They are employed in a variety of Internet protocols, notably in streaming and copy right protection services and often incorporate dynamic behavior. [75].

In this multicast model, a *network operator* controls a secret object $\mathcal{K}$. Each user in the network is running a $k-$threshold scheme. A $k-$threshold scheme is where a user must obtain exactly k keys to obtain access to $\mathcal{K}$. A *trusted user* is someone the network operator allows access to $\mathcal{K}$. An *untrusted user* is someone who the network operator does not allow access to $\mathcal{K}$. The trusted user status is not permanent as a trusted user can become an untrusted user over time. The untrusted user can use his knowledge adversarily and try to obtain access to $\mathcal{K}$.

We will model the user with a process template that represents the k-threshold scheme. There are three states, $\{NT, T, K\}$, where NT denotes the process not trying to obtain access to secret $\mathcal{K}$, C denotes a state of computation where he is using his keys to compute $\mathcal{K}$, and K denotes access to $\mathcal{K}$. The process when invoked transitions from NT to T, otherwise loops at NT. In state T, the process can loop as it may not have enough information to compute $\mathcal{K}$ or it transitions to K and back to a state of NT. Each process runs independently and does not make decisions with respect to other processes. The set of atomic propositions will be $AP = \{trusted, untrusted\}$, to denote whether the user is trusted or not.

In this model, we will assume the security of identification and all trusted users can identify themselves to the network operator. Whenever a user obtains a key, knowledge of the key is permanently retained (he does not forget it). However, the network operator can choose to invalidate any key at will. To get access to $\mathcal{K}$ a user must have k valid keys. If a user's key becomes invalid, a new key must be obtained from the network operator to maintain access to $\mathcal{K}$. In this model $\mathcal{K}$ is considered to be a resource rather than a piece of knowledge. A trusted user whom once had access to $\mathcal{K}$ turned into adversary can not exploit his previous access to attack the network.

A Logical Key Hierarchy (LKH) is a structural representation of knowledge of the users in the form of a complete binary tree of depth d. Each leaf node denotes both a user and his private key. Each non-leaf nodes denote a shared key. A user knows all keys in the path from the user's representative node to the root node. All keys in the logical key hierarchy are valid keys. When a network operator chooses to invalidate a key, users may lose access to $\mathcal{K}$ and new valid keys must be rebroadcasted appropriately.

We will use a topology to model the network operator and the symbolic knowledge of each user within the logical key hierarchy. The topology of the network $\mathcal{T} = (U, K, E, L)$ is a four tuple composed of a set of users, a set of keys, a set of directed edges from key to key, and a set of links from users to a key. In our example a user has exactly one link to a key which represents his private key.

Figure 7.3 demonstrates a logical key hierarchy for a network with 16 users denoted as squares. Each user has a link to one key (the users private key) denoted with a dotted line. The path formed from the root key to the user's leaf key denotes the knowedge of the user.

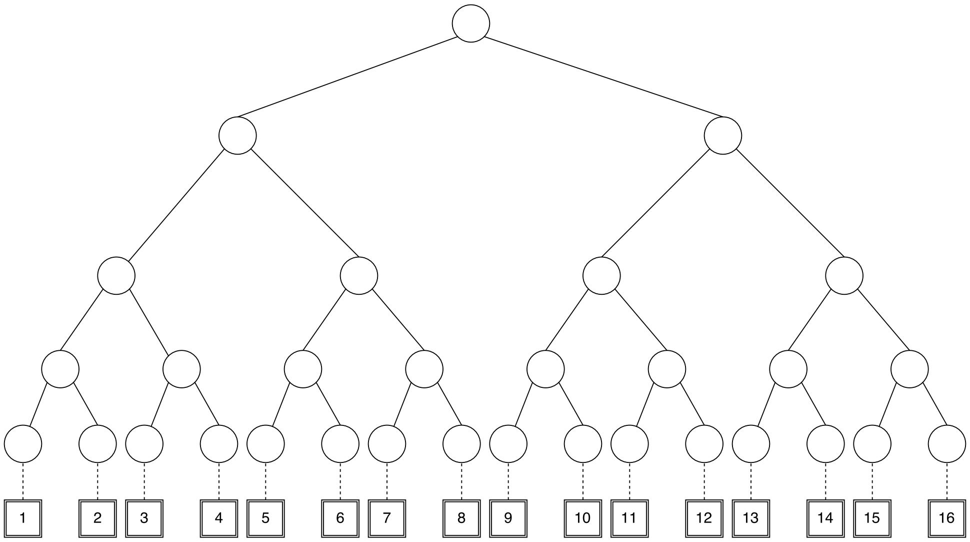

Figure 7.3: An example Logical Key Hierarchy of depth 5 and 16 users

In the figure, each user knows exactly the depth of the tree $d = 5$ keys. The logical key hierarchy was independently invented by [79] and [81], We use the formulation provided in [75].

We will use d to denote the depth of the key tree, N to denote the number of users. Each process is representative of a k threshold scheme with $k = d$. For mutations, we want to be able to support the addition and removal of users into the network. While doing so, we will need to regenerate keys as their values before are no longer valid. These regenerated keys will need to be rebroadcasted to the appropriate trusted users. The replaced keys are deemed *invalid* while the keys currently in the logical key hierarchy are *valid*.

The security specification Ψ of $\mathcal{D}$:

1. A trusted user knows exactly d valid keys. (Topological Specification 1)
2. An untrusted user knows zero valid keys. (Topological specification 2)
3. A user can compute secret $\mathcal{K}$ if k unique keys are known. (Programmatic Specification)

We will next define two mutations to add and remove a user, and then we will argue that they uphold the topological specification.

Add User

1. **Rejection Criteria:** None
2. Check if $2^{d-1} < N + 1 \leq 2^d$ is satisfied.
   (a) If not, the topology needs to be recreated to maintain a complete binary tree key structure. Create a tree with updated keys of depth $d + 1$ and connect the $N + 1$ users with new keys with the instantiated new user. Recompute and broadcast keys for each node.
   (b) Otherwise, instantiate user marked as neither trusted nor untrusted and add appropriately to the logical key hierarchy, so the tree remains complete.
3. Compute path π from the user u to the root node.
4. Broadcast all keys in π to u.
5. Mark new user as trusted

Revoke User

1. **Rejection Criteria:** Reject if $N = 0$
2. Check if $2^{d-1} < N - 1 \leq 2^d$ is satisfied.
   (a) If not, a new topology needs to be created by creating a complete binary tree of depth $d - 1$ with $N - 1$ leaf nodes with new keys with the instantiated new user. Compute new keys for each node and broadcast, and return.
   (b) Otherwise, compute path π from the user u who is going to be removed and root node.
   (c) Invalidate and broadcast new keys on path π to all users but u.
   (d) Mark as untrusted.

Theorem 22. *The multicast model is upholds the described security specification.*

Proof Sketch. We will argue that both the add user and remove user upholds the topological specifications when applied to a model where the topological specification is meet. For adding a single user, preexisting users in the logical key hierarchy are all trusted users as they were in the logical key hierarchy on the start. Furthermore, the user before addition was untrusted and knew no keys on the start. The logical key hierarchy will need to be rebuilt if $N = 2^d$ on start of mutation, to make room for the incoming user. On the rebuild, all old keys become invalid on the reconstruction, so there is no risk of a user having more than d valid keys. Otherwise, the only information change is that to user u who had no keys on start and he receives exactly d keys in the mutation process. Similarly, for node removal in the case of reconstruction, all preexisting keys become invalid and only trusted users receive valid keys. Otherwise, all d keys of u are made invalid, so he knows zero valid keys on termination. All remaining trusted who lose valid keys get updated valid keys in the rebroadcast process. We now have established proof rule (3) and hence (4) under our parametric assumption. This completes our model's verification.

Chapter 8

Conclusions

In this thesis, we discussed computer aided verification and model checking. We began by considering models that resembled a single computational structure and then furthered it to consider concurrent computational structures.

The focus of this thesis is dynamic model checking. We provided a formulation for a mutation procedure that manipulates the overall topology of the model. We further promoted the use of a topology as an abstraction technique to reduce the dominating complexity of programmatic verification.

When verifying a dynamic model, we analyze all resulting models. This approach is exhaustive and ideal for the verification of safety properties. This exhaustive analysis is very expensive from a computational complexity standpoint and we promote the use of parametric assumptions to make the overall complexity reasonable.

While full reachable model analysis is valid, in practice, mutations can exhibit a notion of time, and a complete state space evaluation of each part may not necessarily be required. Further, due to this exhaustive approach liveness properties will almost always fail to be verified.

Two common techniques that have had empirical success in the model checking of transition systems that resemble concurrent computational structures. The first is symmetry analysis which was outlined in this thesis, but no results were made available in this thesis. The second is partial order reduction, which is not discussed in this thesis.

Model Checking is a technique that is fully automizable, and our formulation of dynamic model is claimed to be automizable. Yet, in this thesis, all examples were analyzed manually. A full proof of concept of dynamic model checking would require an implementation.

References

[1] Sat contest, 2015.

[2] Martín Abadi. Security protocols: principles and calculi tutorial notes, foundations of security analysis and design iv, 2007.

[3] Martín Abadi and Roger Needham. Prudent engineering practice for cryptographic protocols. *IEEE transactions on Software Engineering*, 22(1):6, 1996.

[4] Parosh Aziz Abdulla, Frédéric Haziza, and Lukás Holík. All for the price of few. In *VMCAI*, volume 13, pages 476–495. Springer, 2013.

[5] Bowen Alpern and Fred B Schneider. Defining liveness. *Information processing letters*, 21(4):181–185, 1985.

[6] Krzysztof R Apt and Dexter C Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 22(6):307–309, 1986.

[7] André Arnold and Paul Crubille. A linear algorithm to solve fixed-point equations on transition systems. *Information Processing Letters*, 29(2):57–66, 1988.

[8] Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009.

[9] Michael Backes and Christian Jacobi. Cryptographically sound and machine-assisted verification of security protocols. In *STACS 2003*, pages 675–686. Springer, 2003.

[10] Massimo Benerecetti, Fausto Giunchiglia, Maurizio Panti, and Luca Spalazzi. A logic of belief and a model checking algorithm for security protocols. In *Formal Methods for Distributed System Development*, pages 393–408. Springer, 2000.

[11] Josh Berdine, Cristiano Calcagno, and Peter W O'hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *International Symposium on Formal Methods for Components and Objects*, pages 115–137. Springer, 2005.

[12] Sergey Berezin, Sérgio Campos, and Edmund M Clarke. Compositional reasoning in model checking. *Lecture Notes in Computer Science*, 1536:81–102, 1998.

[13] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

[14] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. Conflict-driven clause learning sat solvers.

[15] Roderick Bloem, Swen Jacobs, Ayrat Khalimov, Igor Konnov, Sasha Rubin, Helmut Veith, and Josef Widder. Decidability of parameterized verification. *Synthesis Lectures on Distributed Computing Theory*, 6(1):1–170, 2015.

[16] Nikola Bogunovic and Edgar Pek. Verification of mutual exclusion algorithms with smv system. In *EUROCON 2003. Computer As A Tool. The IEEE Region 8*, volume 2, pages 21–25. IEEE, 2003.

[17] Ahmed Bouajjani, Yan Jurski, and Mihaela Sighireanu. A generic framework for reasoning about dynamic networks of infinite-state processes. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 690–705, 2007.

[18] Aaron R Bradley. Sat-based model checking without unrolling. In *Vmcai*, volume 6538, pages 70–87. Springer, 2011.

[19] Randal E Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691, 1986.

[20] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. Nusmv: A new symbolic model verifier. In *International conference on computer aided verification*, pages 495–499. Springer, 1999.

[21] Edmund Clarke, Orna Grumberg, and Kiyoharu Hamaguchi. Another look at ltl model checking. In *Computer Aided Verification*, pages 415–427. Springer, 1994.

[22] Edmund M Clarke and I Anca Draghicescu. Expressibility results for linear-time and branching-time logics. In *Workshop/School/Symposium of the REX Project (Research and Education in Concurrent Systems)*, pages 428–437. Springer, 1988.

[23] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[24] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. 1999.

[25] Byron Cook, Heidy Khlaaf, and Nir Piterman. Fairness for infinite-state systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 384–398. Springer, 2015.

[26] Giorgio Delzanno, Arnaud Sangnier, Riccardo Traverso, and Gianluigi Zavattaro. On the complexity of parameterized reachability in reconfigurable broadcast networks. 2012.

[27] Giorgio Delzanno, Arnaud Sangnier, and Gianluigi Zavattaro. Parameterized verification of safety properties in ad hoc network protocols. *arXiv preprint arXiv:1108.1864*, 2011.

[28] Edsger W Dijkstra. Solution of a problem in concurrent programming control. In *Pioneers and Their Contributions to Software Engineering*, pages 289–294. Springer, 2001.

[29] Edsger W Dijkstra and Carel S Scholten. *Predicate calculus and program semantics*. Springer Science & Business Media, 2012.

[30] E Allen Emerson. The beginning of model checking: A personal perspective. In *25 Years of Model Checking*, pages 27–45. Springer, 2008.

[31] Marcelo Fiore and Martín Abadi. Computing symbolic models for verifying cryptographic protocols. In *csfw*, page 0160. IEEE, 2001.

[32] Rob Gerth, Doron Peled, Moshe Y Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification, Testing and Verification XV*, pages 3–18. Springer, 1996.

[33] Alex Goldschmidt. Replay attack vulnerabilities and mitigation strategies : A study of a unique class of attack and several methods attempting to prevent attacks of a kind. sep 2015.

[34] Ben Goodspeed. *Formal methods for secure software construction*. PhD thesis, Saint Mary's University, 2016.

[35] Alexey Gotsman, Josh Berdine, and Byron Cook. Interprocedural shape analysis with separated heap abstractions. In *International Static Analysis Symposium*, pages 240–260. Springer, 2006.

[36] Alex Groce and Rajeev Joshi. Extending model checking with dynamic analysis. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 142–156. Springer, 2008.

[37] Viet Tung Hoang, Jonathan Katz, and Alex J. Malozemoff. Automated analysis and synthesis of authenticated encryption schemes. Cryptology ePrint Archive, Report 2015/624, 2015. http://eprint.iacr.org/2015/624.

[38] Gerard J Holzmann. The model checker spin. *IEEE Transactions on software engineering*, 23(5):279, 1997.

[39] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. Automata theory, languages, and computation. *International Edition*, 24, 2006.

[40] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and reasoning about systems.* Cambridge university press, 2004.

[41] C Norris Ip and David L Dill. Better verification through symmetry. *Formal methods in system design*, 9(1-2):41–75, 1996.

[42] Alexander Kaiser, Daniel Kroening, and Thomas Wahl. Dynamic cutoff detection in parameterized concurrent programs. In *CAV*, volume 10, pages 645–659. Springer, 2010.

[43] Richard Kemmerer, Catherine Meadows, and Jonathan Millen. Three systems for cryptographic protocol analysis. *Journal of CRYPTOLOGY*, 7(2):79–130, 1994.

[44] Yonit Kesten, Zohar Manna, Hugh McGuire, and Amir Pnueli. A decision algorithm for full propositional temporal logic. In *Computer Aided Verification*, pages 97–109. Springer, 1993.

[45] DE Knuth. Additional comments on aproblem in concurrent program control, 1966.

[46] Jan Krajícek. Proof complexity. In *European Congress of Mathematics Stockholm, June 27–July 2, 2004*, pages 221–232, 2005.

[47] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.

[48] Zarrin Langari and Richard Trefler. Formal modeling of communication protocols by graph transformation. *FM 2006: Formal Methods*, pages 348–363, 2006.

[49] Zarrin Langari and Richard Trefler. Symmetry for the analysis of dynamic systems. *NASA Formal Methods*, pages 252–266, 2011.

[50] Zarrin Langari and Richard J Trefler. Application of graph transformation in verification of dynamic systems. In *IFM*, volume 9, pages 261–276. Springer, 2009.

[51] Arjen K Lenstra, Xiaoyun Wang, and BMM de Weger. Colliding x. 509 certificates. Technical report, 2005.

[52] Xavier Leroy. The compcert verified compiler, software and commented proof, 2008.

[53] Orna Lichtenstein and Amir Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 97–107. ACM, 1985.

[54] Gavin Lowe. An attack on the needham-schroeder public-key authentication protocol. *Information processing letters*, 56(3):131–133, 1995.

[55] Gavin Lowe. Breaking and fixing the needham-schroeder public-key protocol using fdr. In *International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, pages 147–166. Springer, 1996.

[56] João P Marques-Silva and Karem A Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[57] Jean Everson Martina and Lawrence Charles Paulson. Verifying multicast-based security protocols using the inductive method. *International Journal of Information Security*, 14(2):187–204, 2015.

[58] Catherine Meadows. Formal methods for cryptographic protocol analysis: Emerging issues and trends. *IEEE journal on selected areas in communications*, 21(1):44–54, 2003.

[59] Jorge J Moré. The levenberg-marquardt algorithm: implementation and theory. In *Numerical analysis*, pages 105–116. Springer, 1978.

[60] Sape J Mullender. Eleven principles and why cryptosystems fail.

[61] Kedar S Namjoshi and Richard J Trefler. Local symmetry and compositional verification. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 348–362. Springer, 2012.

[62] Kedar S Namjoshi and Richard J Trefler. Uncovering symmetries in irregular process networks. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 496–514. Springer, 2013.

[63] Kedar S Namjoshi and Richard J Trefler. Analysis of dynamic process networks. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 164–178. Springer, 2015.

[64] Kedar S Namjoshi and Richard J Trefler. Loop freedom in aodvv2. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, pages 98–112. Springer, 2015.

[65] Kedar S Namjoshi and Richard J Trefler. Parameterized compositional model checking. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 589–606. Springer, 2016.

[66] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[67] Ramamohan Paturi, Pavel Pudlák, Michael E Saks, and Francis Zane. An improved exponential-time algorithm for k-sat. *Journal of the ACM (JACM)*, 52(3):337–364, 2005.

[68] Lawrence C Paulson. The inductive approach to verifying cryptographic protocols. *Journal of computer security*, 6(1-2):85–128, 1998.

[69] Gary L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.

[70] Juan Carlos Lopez Pimentel, Raul Monroy, and Dieter Hutter. A method for patching interleaving-replay attacks in faulty security protocols. *Electronic Notes in Theoretical Computer Science*, 174(4):117–130, 2007.

[71] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 179–190. ACM, 1989.

[72] Ph Schnoebelen. The complexity of temporal logic model checking.

[73] A Prasad Sistla and Edmund M Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM (JACM)*, 32(3):733–749, 1985.

[74] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending sat solvers to cryptographic problems. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 244–257. Springer, 2009.

[75] Douglas R Stinson. *Cryptography: theory and practice*. CRC press, 2005.

[76] Boleslaw K Szymanski. A simple solution to lamport's concurrent programming problem with linear wait. In *Proceedings of the 2nd international conference on Supercomputing*, pages 621–626. ACM, 1988.

[77] Moshe Y Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 322–331. IEEE Computer Society, 1986.

[78] Moshe Y Vardi and Pierre Wolper. Reasoning about infinite computations. *Information and computation*, 115(1):1–37, 1994.

[79] Debby Wallner, Eric Harder, and Ryan Agee. Key management for multicast: Issues and architectures. Technical report, 1999.

[80] Chao Wang, Yu Yang, Aarti Gupta, and Ganesh Gopalakrishnan. Dynamic model checking with property driven pruning to detect race conditions. In *International Symposium on Automated Technology for Verification and Analysis*, pages 126–140. Springer, 2008.

[81] Chung Kei Wong, Mohamed Gouda, and Simon S Lam. Secure group communications using key graphs. *IEEE/ACM transactions on networking*, 8(1):16–30, 2000.

[82] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter O'Hearn. Scalable shape analysis for systems code. In *International Conference on Computer Aided Verification*, pages 385–398. Springer, 2008.