國立臺灣大學管理學院資訊管理學研究所

碩士論文

Department of Information Management

College of Management

National Taiwan University

Master Thesis

多執行緒 C 語言程式的顯式和符號模型驗證方法比較研究

Explicit and Symbolic Model-Checking Approaches for Multithreaded C Programs: A Comparative Study

劉韋成

Wei-Cheng Liu

指導教授：蔡益坤 博士

Advisor: Yih-Kuen Tsay, Ph.D.

中華民國 112 年 02 月

February 2023

doi:10.6342/NTU202300384

Explicit and Symbolic Model-Checking Approaches for Multithreaded C Programs: A Comparative Study

by Wei-Cheng Liu
Advisor: Yih-Kuen Tsay, Ph.D.

Department of Information Management
College of Management
National Taiwan University

February 2023

doi:10.6342/NTU202300384

摘要

國立臺灣大學資訊管理學系

學生：劉韋成　　　　　　　　　　　　　　　　　　　　　　民國 112 年 2 月

指導教授：蔡益坤

多執行緒C語言程式的顯式和符號模型驗證方法比較研究

有兩種主要的方法來實現模型檢查過程，即顯式模型驗證和符號模型驗證。顯式模型驗證首先使用某種深度搜索算法為核心，同時結合偏序歸約技術限制探索狀態的數量。符號模型檢查基於布林函數的操作，代表狀態和路徑的集合，而不是顯式狀態路徑圖的遍歷。二元決策圖通常被使用於符號模型驗證器的實現。這兩種方法都有自己的優點和缺點。

本論文的目的是提供顯式模型驗證和符號模型驗證之間的比較並分析其比較結果。對於顯式模型驗證，我們採用在程序分析框架 Ultimate 中，基於巢狀深度優先搜索的一個模型驗證工具。對於符號模型驗證，我們實現了一個固定點模型驗證工具，它也是在 Ultimate 中被實作，並且使用 JavaBDD 套件作為二元決策圖輔助。我們使用可擴展的源代碼分析器 ScanTU 來進行比較實驗。它允許通過其用戶界面選擇不同的模型驗證工具，因此我們可以在其中使用兩種方法來驗證各種 C 程式語言並進行比較。我們通過使用正確和錯誤的 C 程序和線性時序邏輯性質來設計不同的實驗，以產生更多樣化的比較結果。從實驗中，我們得出結論：在固定點計算期間，可達狀態和遍歷路徑的數量對於我們的模型驗證工具花費的時間有巨大影響。在這篇論文中，我們呈現比較結果並分析其原因。

關鍵詞：Büchi 自動機、固定點計算、線性時序邏輯、模型驗證、多執行緒程式、軟體驗證、二元決策圖

doi:10.6342/NTU202300384

THESIS ABSTRACT

Graduate Institute of Information Management
National Taiwan University

Student: Wei-Cheng Liu Month/Year: February, 2023
Advisor: Yih-Kuen Tsay

Explicit and Symbolic Model-Checking Approaches for Multithreaded C Programs: A Comparative Study

There are two primary approaches to implementing the process of model checking, namely explicit-state model checking and symbolic model checking. Explicit-state model checking uses some depth first search algorithm as the core, while incorporating partial-order reduction techniques to limit the number of states explored. Symbolic model checking is based on the manipulation of Boolean functions, which represent sets of states and transitions, rather than the traversal of explicit-state transition graphs. Binary decision diagrams are often used to support the implementation of a symbolic model checker. Both approaches have their own strengths and weaknesses.

The aim of this thesis is to provide a comparison between explicit model checking and symbolic model checking and to analyze the results of the comparison. For the explicit approach, we adopt an existing implementation of nested depth first search based on the Ultimate program analysis framework. For the symbolic approach, we implement a fixpoint model checker, also based on Ultimate and using JavaBDD as our BDD package. We use Scantu, an extensible source code analyzer, to facilitate the comparison. It allows the choice of different model checkers through its user interface, so we can alternate between the two approaches to verify various target C programs and carry out the comparison easily. We design different experimental settings by using different correctness combinations of temporal properties and C programs to produce more diverse comparison results. From the experiments, we conclude that the number of reachable states has a huge impact on the time spent, and so does the number of traversed transitions during the fixpoint calculation. We present the comparison results and analyze their causes in this thesis.

Keywords: Büchi Automata, Fixpoints, Linear Temporal Logic, Model Checking, Multithreaded programs, Software Verification, Binary Decision Diagram

doi:10.6342/NTU202300384

Contents

doi:10.6342/NTU202300384

doi:10.6342/NTU202300384

List of Figures

doi:10.6342/NTU202300384

List of Algorithms

doi:10.6342/NTU202300384

Chapter 1

Introduction

Ensuring that a system meets its specification is very important for every system designer. In many systems such as an elevator control system or a space launch system, an error in the system might incur severe cost, even the lost of human lives. To avoid such errors, software testing is developed and widely adopted by many companies and systems. Software testing uses test cases as input and checks whether the output of them is correct. However, this kind of testing cannot guarantee correctness. In other words, we cannot comfirm that the target system has no errors when the outputs of those test cases are all correct.

Thus, software verification is applied to detect errors and counterexamples comprehensively in a whole project or system and developers can exclude the errors successfully. In software verification, model checking plays a big role. It can help reaching an automatic and convenient approach for verification. Despite having those advantages, model checking is more complex and expensive compared with software testing. Therefore, methods making model checking systematical have been searched and developed for a long time, and automata-based model checking becomes more common.

1.1 Background

In 1963, Saul Kripke proposed the Kripke structure [12], which may be transformed into a corresponding Büchi automaton. This form of automata provides the convenience for analyzing a system and deriving useful propositions in a formal way. In 1977, Amir Pnueli proposed to use temporal logic [15], for program specification and verification. Temporal logic formulae can also be translated into Büchi automata.

In 1981, the conception of model checking was proposed by E. M. Clarke and E. A. Emerson [8], including a method which has the ability to determine whether a program meets its desired properties. Modeling, specification, and verification are the three phases in the model checking. For the modeling phase, Kripke structures provide a solid foundation, and for the specifica-

doi:10.6342/NTU202300384

tion phase, temporal logic helps. In these two phases, Büchi automata are obtained from Kripke structures and temporal logic formulae, providing convenience and integrity for the whole process.

According to the different representations for automata states, model checking can be distinguished into two kinds, explicit model checking and symbolic model checking. In symbolic model checking, fixpoint calculus and *mu*-calculus were assimilated into model checking in 1992, by D. L. Dill and L. J. Hwang [4], and provided convenience for model checking. In 2001, A. Biere and E. M. Clarke proposed bounded model checking [7], which unrolls the finite state machine for a fixed number of steps, and checks whether a property violation can occur in fewer steps.

1.2 Motivation and Objectives

Both symbolic and explicit algorithms have advantages and shortcomings, and choosing a most appropriate algorithm for different settings is not a trivial task. However, little research has prescribed which method should be used under which case. Thus, a tool with the ability for users to choose the matching method for each case can be useful.

In explicit model checking, [13] developed an Ultimate-based tool support for automata-based model checking of multithreaded programs, completing a library for supporting explicit-state model checkers. [5] compared different depth-first search algorithms for automata-based model checking, giving a comprehensive view about different algorithms and their performance in different scenario. In this thesis, we address the symbolic part, namely fixpoint calculus, and the comparison between symbolic model checking and explicit model checking to find the most appropriate approach for different settings. Our implementation is integrated in Ultimate, a plugin-based program analysis framework, and the comparison can be built with our implementation and others' in their research. Finally, we discuss the results, future works, and limitations of our research.

1.3 Thesis Outline

The rest of this thesis is organized as follows:

- In Chapter 2, preliminary research and studies will be unified. Related tools and techniques are also listed in this chapter, such as NuSMV and Ultimate.

doi:10.6342/NTU202300384

- In Chapter 3, the definitions of symbolic model checking and fix point calculus will be listed to help readers understand the methods this paper will discuss in the following chapters.
- In Chapter 4, we implement the fixpoint methods in Ultimate, including software architecture, algorithms and API provided.
- In Chapter 5, examples and the usage of our model checkers are demonstrated to elaborate our methods.
- In Chapter 6, we will arrange the caparison between symbolic model checking and explicit model checking. Finally, future work will be discussed.

doi:10.6342/NTU202300384

Chapter 2

Related Works

2.1 Existing Model Checkers

In this section, we will introduce several model checkers help our research.

2.1.1 NuSMV

SMV (https://www.cs.cmu.edu/ modelcheck/smv.html) system is a symbolic model checker which was developed by E. M. Clarke and his team in Carnegie Mellon. It concentrates on checking finite state systems In the Computation tree logic (CTL). Credit to the CTL logic, temporal properties, such as safety, liveness, fairness and deadlock freedom, can be specified in a concise syntax. Besides, OBDD-based symbolic model checking algorithm is applied in SMV to efficiently generate result, namely determine whether given specifications are satisfied.

NuSMV is a powerful symbolic model checker which was reimplemented from SMV model checking and it is the first model checking tool based on binary decision diagram (BDD). With the use of BDD, NuSMV provides an efficient memory management and becomes a widely used software verification tools and support technique in other research areas. Besides, NuSMV provides four more additional features with respect to SMV:

- NuSMV provides a textual interaction shell.
- Specialized routines allow for checking invariants.
- The model can be partitioned conjunctively and disjunctively.
- LTL Model Checking is performed via reduction to CTL model checking.

NuSMV2 is the new version of NuSMV, which reserves the advantages of the first version. Furthermore, new extension were updated, for instance, the collection of Minisat SAT solver (http://minisat.se/) and ZChaff SAT solver (http://www.princeton.edu/ chaff/zchaff.html). These extension help NuSMV2 evolving to a SAT-based model checker.

doi:10.6342/NTU202300384

2.1.2 Spin

Simple PROMELA Interpreter (SPIN), a widely used open-source software verification tool, was developed by Gerard J. Holzmann and his team in Bell lab in 1980s. SPIN is adopted to verify the correctness of concurrent software models, and brings convenience in model checking.

As the most famous explicit-state model checker, SPIN adopts improved nested depth first search as its main model checking algorithm. providing counterexamples for the corresponding specification. By this improved algorithm, a number of verification procedures can be implemented efficiently. On the other hands, an improvement called bit-state hashing is included to make the algorithm more space-efficient.

2.1.3 Frama-C

Frama-C is a C program analysis platform developed by the French CEA-List and Inria. It can inspect programs without executing them, making it easy for developers to assure the logic of programs are flawless, for instance, understanding C code written by others, proving formal properties, and dealing with security flaws.

2.1.4 Ultimate

ULTIMATE was developed Matthias Heizmann and Daniel Dietsch and their team in Freiburg. It is a modular, plugin-based program analysis framework, providing an integrated platform which combines the existing plugins. By this advantage, researchers can easily choose plugins they expect to utilize and develop new tools with less obstacles.

Different kinds of automizers and model checkers are included in ULTIMATE, and they are adopted in different circumstances. For instances, ULTIMATE Automizer is dedicated to verification of safety properties based on an automata-theoretic approach to software verification, and ULTIMATE Büchi Automizer concentrates on Termination analysis based on Büchi automata. Languages most of these tools use is Boogie, an intermediate verification Language and C language.

2.2 Comparative Tests

There are numbers of research which are studied the comparirson of explicit-state model checking and symbolic model checking. To conveniently

doi:10.6342/NTU202300384

observe the differences between our research and others' before, we will introduce those research in this part.

Cindy Eisner and Doron Peled compared symbolic and explicit model checking of a software system. [10] Explicit-state model checking is usually accepted that it performs better for verifing hardware systems, and symbolic model checking for verifing software systems. In their research, both explicit and symbolic approach were adopted in examining for verifing software systems, and the result showed that symbolic approach gave a better assurance about the verified system for a low number of processes, while explicit approach allowed to simulate executions with more processes.

Roberto Sebastiani, Stefano Tonetta, and Moshe Y. Vardi studied hybrid approaches to LTL symbolic model checking, namely approaches that use explicit representations of the property automaton, and symbolic representations of the system. This research applied the comparison to three model checking algorithm: doubly-nested fixpoint algorithm, the reduction of emptiness to reachability and the singly-nested fixpoint algorithm. [17]

doi:10.6342/NTU202300384

Chapter 3

Preliminaries

3.1 Automata on Infinite Words

In symbolic model checking, dealing with words with infinite length is a common work. As the reason mentioned above, the ω-automata is needed frequently. In this chapter, we will concentrate on the state-based ω-automata, whose acceptance conditions are defined over states. The definition of nondeterministic ω-automata is as follows:

Definition 1 (Nondeterministic ω-automata)**.** *A nondeterministic ω-automaton is a 5-tuple $\langle Q, \Sigma, \delta, Q_0, Acc\rangle$, where*

- *Q is a finite set of* ***states****,*
- *Σ is a finite* ***alphabet****,*
- *$\delta : Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$ is the* ***transition function****,*
- *$Q_0 \subseteq Q$ is the* ***set of possible initial sets of states****, and*
- *Acc is the* ***acceptance condition****.*

A run of a nondeterministic ω-automata $\mathcal{A}$ on an infinite word $w = \sigma_0\sigma_1 \cdots \in \Sigma^\omega$ for each $\sigma_i \in \Sigma$, $i \geq 0$, is a sequence of states $\rho_w = q_0, q_1, \cdots$ where:

- $q_0 \in Q_0$, and
- $q_{i+1} \in \delta(q_i, \sigma_i)$ for $i \geq 0$

In this thesis, only *Büchi automata* [3], which is a specific type of ω-automata, will be discussed. Let $Inf(\rho_w)$ denote the set of states occurring infinitely many times in the run ρ_w, formally

$$Inf(\rho_w) = \{q_i \in Q \mid \forall i \exists j > i, q_i = q_j\}$$

Then the acceptance condition of a Büchi automaton can be defined. We say ρ_w is an accepting run for a Büchi automaton $\mathcal{B}$ iff

doi:10.6342/NTU202300384

- $\alpha \subseteq$ Q, and
- $Inf(\rho_w) \cap \alpha \neq \varnothing$.

An infinite word $w \in \Sigma^\omega$ is accepted by $\mathcal{B}$ if there exists an accepting run of $\mathcal{B}$ over w.

Next, the following two propositions explain the closedness under intersection of Büchi automata. The first proposition talks about the intersection between two ordinary automata, and the second is the intersection between a safety one and an ordinary one:

Proposition 3.1. *Let $\mathcal{B}_1$ and $\mathcal{B}_2$ be two Büchi automata. There is a Büchi automaton $\mathcal{B}$ which accepts the intersected language $L(\mathcal{B}) = L(\mathcal{B}_1) \cap L(\mathcal{B}_2)$. In other words, the class of languages recognizable by Büchi automata is closed under intersection.* [6]

Proof. Given two Büchi automata $\mathcal{B}_1 = (\Sigma_1, Q_1, \Delta_1, Q_1^0, F_1)$ and $\mathcal{B}_2 = (\Sigma_2, Q_2, \Delta_2, Q_2^0, F_2)$. We can build an automaton for $L(\mathcal{B}_1) \cap L(\mathcal{B}_2)$ as follows:

- $\mathcal{B}_1 \cap \mathcal{B}_2 = (\Sigma, Q_1 \times Q_2 \times \{0, 1, 2\}, \Delta, Q_1^0 \times Q_2^0 \times \{0\}, Q_1 \times Q_2 \times \{2\})$.
- We have $(\langle r, q, x\rangle, a, \langle r', q', y\rangle) \in \Delta$ iff the following conditions hold:
  - $(r, a, r') \in \Delta_1$ and $(q, a, q') \in \Delta_2$.
  - The third component is affected by the accepting conditions of $\mathcal{B}_1$ and $\mathcal{B}_2$.
    - (i) if $x = 0$ and $r' \in F_1$, then $y = 1$.
    - (ii) if $x = 1$ and $q' \in F_2$, then $y = 2$.
    - (iii) if $x = 2$, then $y = 0$.
    - (iv) otherwise, $y = x$.
- The third component is responsible for guaranteeing that accepting states from both $\mathcal{B}_1$ and $\mathcal{B}_2$ appear infinitely often.

□

A simpler intersection may be obtained when all of the states of one of the automata are accepting:

Proposition 3.2. *Assuming all states of $\mathcal{B}_1$ are accepting and that the acceptance set of $\mathcal{B}_2$ is F_2, their intersection can be defined as follows:*

$$\mathcal{B}_1 \cap \mathcal{B}_2 = (\Sigma, Q_1 \times Q_2, \Delta', Q_1^0 \times Q_2^0, Q_1 \times F_2)$$

where $(\langle r, q\rangle, a, \langle r', q'\rangle) \in \Delta'$ iff $(r, a, r') \in \Delta_1$ and $(q, a, q') \in \Delta_2$.

doi:10.6342/NTU202300384

3.2 Kripke Structure and Temporal Logic

In this section, Kripke structure and temporal logic will be discussed, as they are basic knowledges of symbolic model checking.

3.2.1 Kripke Structure

Kripke Structure [11] is a kind of state-transition diagram which models a system formally. The usage of Kripke structure allows us to conveniently capture the necessary properties of a system.

Definition 2. *Given a finite set AP of atomic propositions, a Kripke Structure over AP is a tuple $\langle S, S_0, \rightarrow, \mathcal{L}\rangle$ where:*

- *S is a finite set of* ***states****,*
- *$S_0 \subseteq S$ is the* ***set of initial states****,*
- *$\rightarrow \subseteq S \times S$ is the* ***transition relation****, and*
- *$\mathcal{L} : S \rightarrow 2^{AP}$ is the* ***labeling function****.*

A *run* (or computation) of a Kripke Structure $\mathcal{K}$ is an infinite sequence $\rho = s_0 s_1 \cdots$ such that

- $s_0 \in S_0$, and
- $(s_i, s_{i+1} \in \rightarrow)$ for all $i \geq 0$.

And we define

- $\rho(i) = s_i$, and
- $\rho_i = s_i s_{i+1} \cdots$ for all $i \geq 0$.

Let $\mathcal{L}(\rho) = \mathcal{L}(s_0)\mathcal{L}(s_1)\cdots$ and the language of $\mathcal{K}$ is defined as $L(\mathcal{K}) = \{\mathcal{L}(\rho) \mid \rho \text{ is a run of } \mathcal{K}\}$.

3.2.2 Temporal Logic

Temporal logic is a formalism for describing temporal ordering between occurrences of "events" which are often represented by propositions. Look back at the history of temporal logic development, it was first introduced by A. Prior in the 1960's and further developed by A. Pnueli for computer usage [15]. Temporal logics are especially well suited to describe temporal

doi:10.6342/NTU202300384

constraints of concurrent, reactive, and non-terminating systems [16]. Therefore, temporal logic take a importante position in the field of formal software verification.

Propositional (Linear) Temporal Logic, namely PTL, is a restricted type of temporal logic which describe both future and past in a linear structure. And *Linear Temporal Logic* (LTL) is a subset of Propositional Temporal Logic keeping only future operators, which include **X** (next), **F** (finally), **G** (globally), **U** (until) and **R** (release). These operators can also be written symbolically, i.e., $\bigcirc$ (next), $\Diamond$ (eventually), $\Box$ (always), $\mathcal{U}$ (until) and $\mathcal{R}$ (release).

Definition 3 (Syntax of LTL Formulae)**.** *The LTL formulae over a set of atomic propositions AP are defined as the following grammar:*

$$\phi ::= p \mid \neg\phi \mid \phi \vee \phi \mid \bigcirc\phi \mid \phi\,\mathcal{U}\,\phi \mid \phi\,\mathcal{R}\,\phi$$

where $p \in AP$.

Definition 4 (Semantics of LTL)**.** *Given an infinite word $w = \sigma_0\sigma_1 \cdots \in \Sigma^\omega$ where $\Sigma = 2^{AP}$, and an LTL formula ϕ over AP, we say that w satisfies ϕ (written $w \models \phi$) if and only if (recursively):*

- $\phi \equiv p$ *and* $p \in \sigma_0$, *or*
- $\phi \equiv \neg\phi_1$ *and* $w \not\models \phi_1$, *or*
- $\phi \equiv \phi_1 \vee \phi_2$ *and* $w \models \phi_1$ *or* $w \models \phi_2$, *or*
- $\phi \equiv \phi_1 \wedge \phi_2$ *and* $w \models \phi_1$ *and* $w \models \phi_2$, *or*
- $\phi \equiv \bigcirc\phi_1$ *and* $\sigma_1\sigma_2 \cdots \models \phi_1$, *or*
- $\phi \equiv \phi_1\mathcal{U}\phi_2$ *and for some* $k \in \mathbb{N}$, $\sigma_k\sigma_{k+1} \cdots \models \phi_2$ *and for all* i, $0 \leq i < k$, $\sigma_i\sigma_{i+1} \cdots \models \phi_1$, *or*
- $\phi \equiv \phi_1\,\mathcal{R}\,\phi_2$ *and for all* $k \in \mathbb{N}$, *if for every* i, $0 \leq< k$, $\sigma_i\sigma_{i+1} \cdots \not\models \phi_1$, *then* $\sigma_k\sigma_{k+1} \cdots \models \phi_2$.

With this definition, additional formulae and $\Diamond$, $\Box$ operators can be derived as follows:

- $true = \phi \vee \neg\phi$,
- $false = \phi \wedge \neg\phi$,
- $\Diamond\phi = true\,\mathcal{U}\,\phi$, and

doi:10.6342/NTU202300384

- $\Box\phi = \neg\Diamond\neg\phi$.

Altough $\phi_1 \wedge \phi_2$ and $\phi_1 \mathcal{R} \phi_2$ can also be derived by $\phi_1 \wedge \phi_2 = \neg(\neg\phi_1 \vee \neg\phi_2)$ and $\phi_1 \mathcal{R} \phi_2 = \neg(\neg\phi_1 \mathcal{U} \neg\phi_2)$ respectively, we put them into the definition field under the consideration of the structure of the negative normal form (NNF).

Definition 5 (Language Defined by an LTL formula)**.** *Given an LTL formula ϕ, language of ϕ is defined by $L(\phi) = \{w \in \Sigma^\omega \mid w \models \phi\}$.*

Given an LTL formula ϕ and a Kripke structure $\mathcal{K}$ over AP, we say $\mathcal{K}$ satisfies ϕ (written $\mathcal{K} \models \phi$) if and only if $L(\mathcal{K}) \subseteq L(\phi)$, that is for all runs ρ of $\mathcal{K}$, $\mathcal{L}(\rho) \models \phi$.

3.3 Symbolic Model Checking

Symbolic model checking has been considered a more efficient approach in model checking, as it considering large numbers of states at a single step. In this section, two approaches in symbolic model checking we just mentioned in aforementioned sections will be discussed, the first one is fixpoint calculus, and next is bounded model checking. These two approaches will be implemented with Ultimate in our research.

3.3.1 Fixpoint Calculus

Recall that a **complete lattice** is a partially ordered set in which every subset of elements has a **least upper bound**(supremum) and a **greatest lower bound** (infimum).

Definition 6 (complete lattice)**.** *For a given set S, $\langle P(S), \subseteq\rangle$ forms a complete lattice. Let $S' \subseteq P(S)$, then*

- *the supremum of S', usually denoted $sup(S')$, equals $\cup S'$ and*
- *the infimum of S', denoted $inf(S')$, equals $\cap S'$.*

*The least element in $P(S)$ is the empty set ϕ, which we refer to as **False**. The greatest element in $P(S)$ is the set S, which we refer to as **True**.*

Use the definition of the complete lattice, Bronislaw Knaster and Alfred Tarski stated the Knaster-Tarski theorem [9]:

Definition 7 (Knaster-Tarski Fixpoint Theorem)**.** *Let L be a complete lattice and $F : L \to L$ an order-preserving map. Then,*

$$\mu(F) = \bigwedge\{x \in L \mid F(x) \leq x.\}$$

Dually, $\nu(F) = \bigvee\{x \in L \mid x \leq F(x)\}$

doi:10.6342/NTU202300384

Knaster-Tarski theorem has important applications in formal semantics of programming languages and abstract interpretation, bringing out the concepts of least fixpoint and greatest fixpoint. To introduce the procedures of calculaing least fixpoint and greatest fixpoint, we first need to introduce the concept of the predicate transformer [14]:

Definition 8 (predicate transformer)**.** *A **predicate transformer** on $P(S)$ is a function $\tau : P(S) \to P(S)$. $\tau^i(Z)$ is used to denote i applications of τ to Z:*

- $\tau^0(Z) = Z$
- $\tau^{i+1}(Z) = \tau(\tau^i(Z))$

Next, we will discuss **monotonic** and **continuous** of the predicate transformer. Let τ be a predicate transformer.

- τ is **monotonic** (order-preserving) provided that

$$P \subseteq Q \implies \tau(P) \subseteq \tau(Q).$$

- τ is **$\cup$-continuous** provided that

$$P_1 \subseteq P_2 \subseteq ... \implies \tau(\cup_i P_i) = \cup_i \tau(P_i).$$

- τ is **$\cap$-continuous** provided that

$$P_1 \supseteq P_2 \supseteq ... \implies \tau(\cap_i P_i) = \cap_i \tau(P_i).$$

In the next part, we will discuss procedures which calculate the least fixpoint and the greatest fixpoint [1]:

doi:10.6342/NTU202300384

Algorithm 1: Least Fixpoint Procedure

```
prog Lfp(τ : Predicate Transformer) : Predicate
    $Q$ := False;
    $Q'$ := $\tau(Q)$;
    while ($Q \neq Q'$) do
        $Q$ := $Q'$;
        $Q'$ := $\tau(Q)$;
    end
    return ($Q$)
end
```

Algorithm 2: Greatest Fixpoint Procedure

```
prog Gfp(τ : Predicate Transformer) : Predicate
    $Q$ := True;
    $Q'$ := $\tau(Q)$;
    while ($Q \neq Q'$) do
        $Q$ := $Q'$;
        $Q'$ := $\tau(Q)$;
    end
    return ($Q$)
end
```

3.3.2 Theorem of the Model Checker Using Fixpoints

In symbolic model checking, binary decision tree (BDD) has been frequently used as a symbolic representation of the system. [2] A BDD represents a Boolean function as a rooted, directed acyclic graph (function graph):

Definition 9 (Binary Decision Diagram (BDD))**.** *We use $r(G)$ to denote the root of a function graph G. The vertex set V of a function graph G contains two types of vertices:*

- *A **nonterminal** vertex v has*

  an argument index : $index(v) \in \{1, ..., n\}$ and

  two children : $low(v), high(v) \in V$.

- *A **terminal** vertex v has a value $value(v) \in \{0, 1\}$.*

Transition systems can be encoded in Boolean functions and thus representable in BDDs. Based on this reason, symbolic model checking becomes possible with BDDs.

doi:10.6342/NTU202300384

Next, we want to check the emptiness of $\mathcal{A}_{\mathcal{M}} \times \mathcal{A}_{\neg p}$ where $\mathcal{A}_{\mathcal{M}} = \langle Q_{\mathcal{M}}, I_{\mathcal{M}}, \Sigma, \delta_{\mathcal{M}}, Q_{\mathcal{M}} \rangle$ is the safety Büchi automaton for $\mathcal{M}$ and $\mathcal{A}_{\neg p} = \langle Q_{\neg p}, I_{\neg p}, \Sigma, \delta_{\neg p}, \alpha_{\neg p} \rangle$ is the Büchi automaton for $\neg p$, we may use the following fixpoint:

Definition 10 (Fixpoint Formula). *The fixpoint we used in our model checker:*

$$\mathcal{R}_\alpha \equiv \alpha \cap \mu x \cdot (Post(x) \cup I)$$

$$\mathcal{F}_p \equiv \nu y \cdot \mu x \cdot (Post(x) \cup (Post(y) \cap \mathcal{R}_\alpha))$$

where $I = I_{\mathcal{M}} \times I_{\neg p}$,
$\alpha = Q_{\mathcal{M}} \times \alpha_{\neg p}$,
$Post(S) = \{\langle r', q' \rangle \mid \exists \langle r, q \rangle \in S : r' \in \delta_{\mathcal{M}}(r) \wedge q' \in \delta_{\neg p}(q)$ *where* $r, r' \in Q_{\mathcal{M}}$ *and* $q, q' \in Q_{\neg p}\}$
R_α *is the set of reachable accepting states.*

In this definition, R_α is the set of reachable accepting states, and $\mathcal{F}_p$ is the set of reachable accepting states which are traversed infinitely often. In other words, the goal of this fixpoint formula is to check whether there are any reachable accepting states which are traversing infinitely often. Finally, we have the following theorem:

Theorem 1. $\mathcal{F}_p = \varnothing$ *iff* $L(\mathcal{A}_{\mathcal{M}} \times \mathcal{A}_{\neg p}) = \varnothing$.

By this theorem, we successfully finish calculating the fixpoint.

The last thing we concern about is the fairness issue. In our model checker, we use fairness constraints to ensure the fairness. Fairness constraints are imposed to the system (or the specification) so that processes in this system can be executed fairly. [18] lists three typical fairness constraints formulated in LTL:

Definition 11 (Absolute Fairness, Impartiality). *Every process should be executed infinitely often:*

$$\forall i : GFex_i$$

However, Absolute Fairness ignores that some processes might not be ready to execute. Strong Fairness and Weak Fairness consider that:

Definition 12 (Strong Fairness). *Every process that is infinitely often enabled should be executed infinitely often in a state where it is enabled:*

$$\forall i : (GFen_i) \Rightarrow (GF(en_i \wedge ex_i))$$

doi:10.6342/NTU202300384

Definition 13 (Weak Fairness)**.** *Every process that is almost always enabled should be executed infinitely often:*

$$\forall i : (FGen_i) \Rightarrow (GFex_i)$$

In our model checker, we use Weak Fairness.

3.3.3 Bounded Model Checking

In the last part, we have introduced the way we utilize BDD in the fixpoint model checker. However, there are drawbacks of BDDs:

- For large systems (with over a few hundred boolean variables), they can be prohibitively large.
- Selecting the right variable ordering is often time-consuming or need manual intervention.

To deal with these drawbacks, basic ideas of **bounded model checking (BMC)** was proposed. Bounded model checking considers only a finite prefix of a path that may be a witness of $\mathbf{E}f$. The length of the prefix is restricted to a certain bound k, and the propositional formula can be solved by a SAT solver. If there is no witness within bound k, we increase the bound and look for longer and longer possible witnesses.

Definition 14 (LTL Semantic for Bounded Model Checking)**.** *Let M be a Kriple structure, π be a path in M, and f be an LTL formula (in negation normal form). $\pi \models f$ (f is valid along π) is defined as follows:*

- $\pi \models p$ *iff* $P \in L(\pi(0))$
- $\pi \models \neg p$ *iff* $P \notin L(\pi(0))$
- $\pi \models f \wedge g$ *iff* $\pi \models f$ *and* $\pi \models g$
- $\pi \models f \vee g$ *iff* $\pi \models f$ *or* $\pi \models g$
- $\pi \models \Box f$ *iff* $\forall j \in [0, \infty).\pi^j \models f$
- $\pi \models \Diamond f$ *iff* $\exists j \in [0, \infty).\pi^j \models f$
- $\pi \models \bigcirc f$ *iff* $\pi^1 \models f$
- $\pi \models f \ \boldsymbol{U}\ g$ *iff* $\exists j \in [0, \infty).(\pi^j \models g$ *and* $\forall k \in [0, j).\pi^k \models f)$
- $\pi \models f \ \boldsymbol{R}\ g$ *iff* $\forall j \in [0, \infty).(\pi^j \models g$ *or* $\exists k \in [0, j).\pi^k \models f)$

doi:10.6342/NTU202300384

Definition 15 (Bounded Semantics for a Loop)**.** *Let $k \in \mathbb{N}$ and π be a k-loop. Then an LTL formula f is valid along the path π with bound k (in symbols $\pi \models_k f$) iff $\pi \models f$*

Definition 16 (Bounded Semantics without a Loop)**.** *Let $k \in \mathbb{N}$ and π be a path that is not a k-loop. $\pi \models_k f$ iff $(\pi, 0) \models_k f$ where*

- $(\pi, i) \models_k p$ *iff* $P \in L(\pi(i))$
- $(\pi, i) \models_k \neg p$ *iff* $P \notin L(\pi(i))$
- $(\pi, i) \models_k f \wedge g$ *iff* $(\pi, i) \models_k f$ *and* $(\pi, i) \models_k g$
- $(\pi, i) \models_k f \vee g$ *iff* $(\pi, i) \models_k f$ *or* $(\pi, i) \models_k g$
- $(\pi, i) \models_k \Box f$ *iff false*
- $(\pi, i) \models_k \Diamond f$ *iff* $\exists j \in [i, k].(\pi, j) \models_k f$
- $(\pi, i) \models_k \bigcirc f$ *iff* $i < k$ *and* $(\pi, i+1) \models_k f$
- $(\pi, i) \models_k f \ \boldsymbol{U} \ g$ *iff* $\exists j \in [i, k].((\pi, j) \models_k g$ *and* $\forall n \in [i, j).(\pi, n) \models_k f)$
- $(\pi, i) \models_k f \ \boldsymbol{R} \ g$ *iff* $\forall j \in [i, k].((\pi, j) \models_k f$ *or* $\exists n \in [i, j].(\pi, n) \models_k g)$

doi:10.6342/NTU202300384

Chapter 4

Design and Implementation

The goal of this thesis is building an Ultimate-based symbolic fixpoint model checker, which can deal with multithread C programs. Our fixpoint model checker is called Scantu-Fixpoint-Model-Checker and our library is called Library-FixpointModelChecker. Scantu-Fixpoint-Model-Checker is a symbolic model checker using BDD and RCFG provided by Ultimate. Scantu-Fixpoint-Model-Checker can check a multithreaded C program against an LTL formula that describes a desired behavior.

In this section, we will first introduce some technique supports provided by Ultimate. Second, the basic usage of JavaBDD will be presented. Last but not least, we will introduce the way we design Scantu-Fixpoint-Model-Checker, including the usage of supports provided by Ultimate and JavaBDD and how to check specifications with fixpoints calculated by Scantu-Fixpoint-Model-Checker. Of course, the way we implement the fixpoint formula introduced in the previous section will be illustrated in this section. Because of these well-designed supports, we succesfully implemented our fixpoint model checker.

4.1 Software Architecture

RCFG provided by Ultimate and JavaBDD package play big roles in the processes we designed Scantu-Fixpoint-Model-Checker. Because Ultimate does not provide any symbolic tools, we need to adopt JavaBDD to create symbolic environment.

In this following section, we will give RCFG and JavaBDD detailed introduction:

4.1.1 Recursive Control Flow Graph (RCFG)

A recursive control flow graph is a directed graph which is composed of two elements, the first one are nodes, which are called locations in the graph, and the second are edges, which are called transition labeled with

doi:10.6342/NTU202300384

statements. No matter in explicit or symbolic model checking, the RCFG helps us clearly analizing the concurrent C code. The definition of recursive control flow graph is as follows:

Definition 17 (Recursive Control Flow Graph (RCFG))**.** *A recursive control flow graph (RCFG) is a tuple* $R = (Loc, Loc_{init}, Loc_{error}, E, st)$ *where*

- Loc *is a set of program locations,*
- $Loc_{init} \subseteq Loc$ *is a set of initial program locations,*
- $Loc_{error} \subseteq Loc$ *is a set of program locations that violate a specification,*
- $E \subseteq P(Loc) \times Loc$ *is a transition relation and*
- $st : E \to Stmt$ *is a labeling function that labels each transition with a statement* s*, where* s *is either an assignment, an assume, a havoc, a call, or a return.*

An element e *of* E *is an ordered pair* (S, t)*, where* S *is the non-empty set of source locations of* e *and* t *is the target location. The size of* S *is restricted to two locations, i.e.,* $|S| \leq 2$.

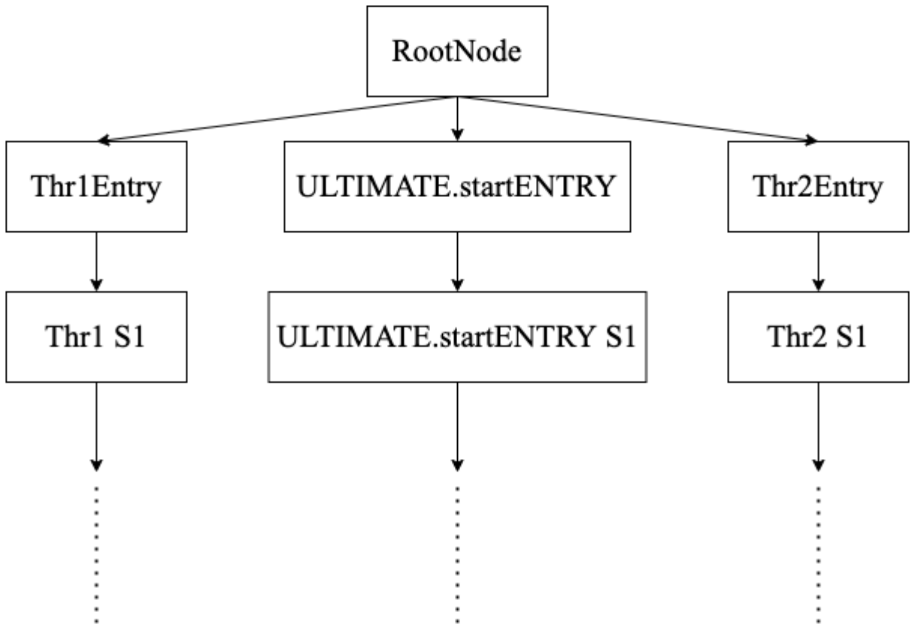


Figure 4.1: An example RCFG of one concurrent code which has two threads.

Figure 4.1 shows an RCFG of one concurrent code which has two threads. the left-side chain and the right-side chain represent each thread in the target code, and the middle chain represent other control flows which are not in both threads.

doi:10.6342/NTU202300384

When creating BDD transitions for the RCFG, we need to observe all changes in value of all variables of the RCFG. As mentioned above, transitions of the RCFG are labeled with statements, and those are what we need. As a result, dealing with the extraction of assignment statements and the integration of changes in value of all variables are important tasks.

4.1.2 Different Kinds of Transitions in RCFG

There are different kinds of transitions in the RCFG provided by Ultimate, and each kind of transition handles different statements. In this section, we will introduce all statements we have dealt with in our model checker and different expressions in these statements.

First of all, all expressions we have worked on are listed below, and we will present how we use these expressions in Ultimate:

- **Integer Literal**
  An Integer Literal is a form of an integer element, for example:

  IntegerLiteral[0]

  means an integer 0. In Ultimate, the function $getValue()$ is designed to get the value of this expression, which type is String.

- **Identifier Expression**
  An Identifier Expression is a form of primary expression, for example:

  IdentifierExpression[x]

  means an identifier x. In Ultimate, the function $getIdentifier()$ is designed to get the identifier of this expression, which type is String.

- **Unary Expression**
  A **Unary Expression** contains one unary operator and one expression, for example:

  UnaryExpression[LOGICNEG, IdentifierExpression[x]]

  means $!x$. In Ultimate, the function $getOperator()$ is designed to get the operator in Unary Expression, and the function $getExpr()$ is designed to get the expression in Unary Expression. Basically, this kind of Expression is used to deal with case which operator is **LOGICNEG**.

doi:10.6342/NTU202300384

- **Binary Expression**
  A **Binary Expression** contains one binary operator and two expression, for example:

  $$\textbf{BinaryExpression[ARITHPLUS, expr1, expr2]}$$

  where

  $$\textbf{expr1} = \textbf{IdentiferExpression}[\textbf{x}], \textbf{expr2} = \textbf{IdentiferExpression}[\textbf{y}]$$

  means $x + y$. In Ultimate, the function *getOperator*() is designed to get the operator in Binary Expression, and the function $getLeft()$ and *getRight* are designed to get the first and second expression in Binary Expression. All kinds of Operator are listed below:

  – **LOGICIFF** means $\Leftrightarrow$
  – **LOGICIMPLIES** means $\rightarrow$
  – **LOGICAND** means logical AND
  – **LOGICOR** means logical OR
  – **COMPLT** means $<$
  – **COMPGT** means $>$
  – **COMPLEQ** means $\leq$
  – **COMPGEQ** means $\geq$
  – **COMPNEQ** means $\neq$
  – **ARITHPLUS** means $+$
  – **ARITHMINUS** means -
  – **ARITHMUL** means $\times$
  – **ARITHDIC** means $\div$
  – **ARITHMOD** means %

- **Boolean Literal**
  A **Boolean Literal** is a form of boolean value **TRUE** or **FALSE**, for example:

  $$\textbf{BooleanLiteral[TRUE]}$$

  means a boolean value **TRUE**. In Ultimate, the function $getValue()$ is designed to get the boolean value in Boolean Literal.

In the next section, different kinds of statements are as follows. These kinds of statements are all composed of expression we have introduced. By the way, we will illustrate the way our fixpoint model checekr deal with these statements.

doi:10.6342/NTU202300384

- **Assignment Statement**
  An **Assignment Statement** deals with assignments of variables, and has two important parameters: The type of the first one is an Array of Identifier and the type of *rhs* is an Array of Expression. For examples:

  $$\textbf{AssignmentStatement[lhs, rhs]}$$

  where

  $$\textbf{lhs = VariableLHS[x, GLOBAL], rhs = IntegerLiteral[0]}$$

  means $x = 0$. In ULTIMATE, multiple variables may be assigned simultaneously. This is the reason that the type of *lhs* and *rhs* are all arrays. The function *getLhs*() is designed to get the first array, and the function *getRhs*() is designed to get the second one in ULTIMATE.

- **Assume Statement**
  An **Assume Statement** assumes that a boolean formula holds. The target program will terminate if the formula does not hold. This kind of statement has one parameter which type is Expression. For examples:

  $$\textbf{AssumeStatement[BooleanLiteral[true]]}$$

  means **assume true**. In ULTIMATE, The function *getFormula*() is designed to get the Expression in the statement.

- **Call Statement**
  A **Call Statement** represents a procedure call. This kind of statement is always combined with an assignment and has three elements need to be dealt with. The left hand side, the method name and arguments. The left hand side must be variables, which type is an Array of Identifiers, and the method name represents the name of the procedure, which type is String. Last of all, the type of arguments is an Array of Expressions. Next, we list all of the method names we met:

  – **#Ultimate.allocOnStack**
    When the method name is **#Ultimate.allocOnStack**, the system allocates memory to the left-hand-side variables.

  – **write~init~int**
    When the method name is **write~init~int**, the system initializes values to the variables which have been allocated memory.

  – **write~int**
    When the method name is **write~int**, the system change values stored in the left-hand-side variables.

doi:10.6342/NTU202300384

  - **read~int**
    When the method name is **read~int**, the system get values stored in the left-hand-side variables.
  - **ULTIMATE.dealloc**
    When the method name is **ULTIMATE.dealloc**, the system deallocates memory out of the left-hand-side variables.

  This kind of statements are always used to handle Arrays or List in C code.

- **Havoc Statement**
  A **Havoc Statement** destroy the contents of variables by over-writting them with non-deterministically chosen values. In symbolic model checking, this kind of statement does not change values of any variables, so in our model checker, when meeting Havoc Statement, we just change the program counter and do nothing.

- **Fork Statement**
  A **Fork Statement** is an asynchrone procedure call. This kind of statement is used to handle thread-creating actions in C code. In our model checker, we need to check how many times one thread is created. Thanks to Fork Statement, we can successfully record the exact number. For examples:

  ForkStatement[[**left, thr1, right**]]

  where **thr1** is the procedure name. The example Fork Statement means that the procedure **thr1** is created once.

4.1.3 Java Binary Decision Diagram (JavaBDD)

Because ULTIMATE is built on Java, we choose JAVABDD, designed by John Whaley, as BDD supporter in our model checker. JAVABDD is a Java library which provides tools for manipulating BDDs. Thanks to its convenience and systematic APIs, we can succesfully implement our fixpoint model checker.

The JAVABDD APIs is based on BUDDY package, which is a famous package written in C language and have been used a lot in model checking project. Different from BUDDY package, JAVABDD is designed to be object-oriented, so we do not have to struggle with the C function interface which is harder to be familiar with.

doi:10.6342/NTU202300384

Besides BuDDy interface, JavaBDD can also interface with JDD library, or with other BDD libraries written in C, such as CUDD and CAL. In our model checker, JavaBDD library, which is the default one, is used to support our model checker.

In the next part, we will illustrate some common classes provided by this library and how it helps us implementing our model checker:

- **BDDFactory**
  To use functions JavaBDD provides, we have to initialize **BDDFactory** first. Without BDDFactory, we can not do any operations. BDDFactory is an interface for all manipulations of BDDs, for example, the creation and the logic operations.

  To initialize the factory successfully, we have to provide three parameters. The first parameter is the BDD package we want to initialize, in our model checker, **JFactory**, which is the default package is used. The second parameter is an integer, representing the node number. This number decides the initial node table size of the whole system. The last parameter is the cache number, deciding the operation cache size. However, the inital number of nodes is not that important because the table will resized at the expense of efficiency when needed.

- **BDDDomain**
  BDDDomain is a block of BDD variables which can represent integer value as opposed to only boolean values. To build BDD transitions, we need to define all changes of states after doing each transition. **BDDDomain** helps us doing all of these works.

  In our model checker, besides changes of variables in the target C code, changes of program counters are also a critical event. Thanks to the help of BDDDomain, we did not have too much obstacles when implementing our model checker.

- **BDDBitVector**
  BDDBitVector is a data structure designed to store boolean vectors, and there are two fields in it. The first field is *bitnum*, which means the number of elements in the vector, and the second field is *bitvec*, which contains the actual BDDs in the vector.

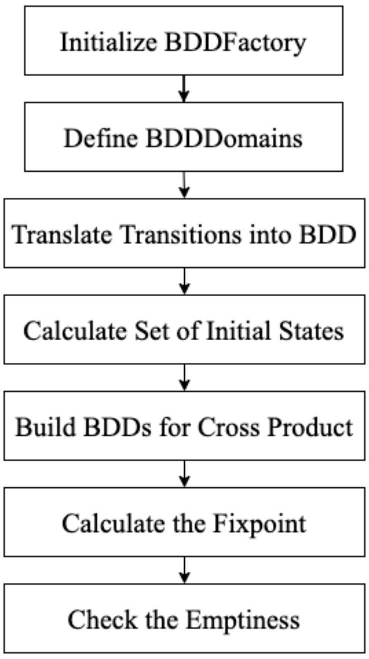
- **BDD**
  BDD is the most important class. It is created by BDDFactory we introduced in the previous section. This class provides lots of functions

doi:10.6342/NTU202300384

and operations that are commonly used, such as basic logic *AND*, *OR*. We do not have enough room to introduce all the functions it provides, so in the next section, we will illustrate how we build our model check step by step, and it is more appropriate to give the functions we used some discussion.

4.2 Design and Implementation

We now describe the processes we implement our model checker, starting from getting the RCFG of the target system and the automaton of the property we want to check by running toolchains RCFGBuilder and LTL2BA provided by ULTIMATE. By utilizing these two structure and the theorem we have introduced in Theorem of the Model Checker Using Fixpoints, we list all steps listed below:



Initialize BDDFactory
Define BDDDomains
Translate Transitions into BDD
Calculate Set of Initial States
Build BDDs for Cross Product
Calculate the Fixpoint
Check the Emptiness

Figure 4.2: Steps of Our Fixpoint Model Checker

Figure 4.2 shows the steps taken by our fixpoint model checker. We discuss them one by one:

- Initialize BDDFactory
  First of all, we initialize BDDFactory and we can use all functions provided by JAVABDD. In our system, the BDD package we use is the default package: **JFactory**. Next, after setting 1000000 as the node size and 1000000 as the cache size, we can start translating commands in the system RCFG into BDDs.

doi:10.6342/NTU202300384

- Define BDDDomains

  In the next step, we need to define the universe of all variables. In other words, we need to find out all variables in the RCFG of the system and the property automaton and create BDDDomains for each of those variables. The way we find out all varialbes is to analyze the type of system RCFG transitions starting from its initial state of thread **ULTIMATE.startENTRY**. This thread deals with works such as creating threads, forking threads and joining threads. By analyzing this thread first, we can take a detailed look on the input of different threads and how many times each thread is created. We need to prepare one BDDDomain for each created thread to record the change of program counter.

  Next, we browse all transitions in system RCFG, and then filter all transitions which type are AssignmentStatement or CallStatement. AssignmentStatement handles the assignment of normal variables, and CallStatement handles the assignment of arrays. For example, there is an AssignmentStatement:

  $$\textbf{AssignmentStatement}[\textbf{lhs, rhs}]$$

  where

  $$\textbf{lhs} = \textbf{VariableLHS}[\textbf{x, GLOBAL}], \textbf{rhs} = \textbf{IntegerLiteral}[\textbf{0}]$$

  we can find one normal variable: **x**, and it is stored in our variable list. For another example for CallStatement:

  $$\textbf{CallStatement}[\textbf{assignedValue, write\textasciitilde init\textasciitilde int, left, right}]$$

  where

  $$\textbf{assignedValue} = \textbf{IntegerLiteral}[\textbf{0}]$$

  $$\textbf{left}=\textbf{IdentifierExpression}[\textbf{flag.base}]$$

  $$\textbf{right}=\textbf{IdentifierExpression}[\textbf{flag.offset}]$$

  This CallStatement shows the initialization of **int flag**[**0**] = **0**. We can not store an array with index in our variable list, so we create one new String: **flag~0**, which represents **flag**[**0**], and stored it into our variable list.

  By repeating these actions, we can get one variable list which contains all variables used in RCFG. By the way, we do not concern variables used in property automaton, the reason is that in ULTIMATE, variables that have not appeared in RCFG are not allowed in the property.

doi:10.6342/NTU202300384

- Translate transitions into BDDs for the system RCFG.
  We designed different methods for each type of transitions to translate them into BDDs:
  - **AssignmentStatement**
    The way we translate transitions with AssignmentStatement type is to separate it into the left-hand side and the right-hand side, which the functions ULTIMATE provides can help us getting these elements. The left-hand side is the variable which value will be changed after doing this transition, and the right-hand side provides the value about to assigned to the left-hand-side variable. For example:

    $$\mathbf{x} = \mathbf{y}$$

    can be seen as:

    $$\mathbf{x'} = \mathbf{y}$$

    x, x, y and **y'** are represented by four different BDDDomains, and we only need two BDDDomains which represent **x'** and **y** to translate this transition into BDD. Because this kind of work will be called lots of times, so we design a AssignmentStatementEvaluator to help us translating transitions with AssignmentStatement type into BDDs. Back to this example, our AssignmentStatementEvaluator first transforms these two BDDDomains into two BDDBitVectors. We make sure that the bit number of these two BDDBitVectors are the same. Then we do conjunction of a biimplication of each element in these two BDDBitVector. Finally, we can get one result BDD which represents the work of manipulations of **assignment** .
    The structure of our AssignmentStatementEvaluator is recursive, meaning that we can deal with assignments with arithmetic operations.
  - **AssumeStatement**
    The way we translate transitions with AssumeStatement type is similar to the method we do to transitions with AssignmentStatement type. The difference between these two methods is that we have additional deal with binary operations, such as **AND** and **OR**. Next, instead of the change of values of variables, whether the current values of states satisfy the assumes are what we really concern about.
    If any of the four arithmetic operations appear in the transition, our AssignmentStatementEvaluator help us finishing those works.

doi:10.6342/NTU202300384

Same as AssignmentStatementEvaluator, the structure of AssumeStatementEvaluator is recursive.

- **CallStatement**
  There are different kinds of transitions with type CallStatement, so we treat them individually:
  * **#Ultimate.allocOnStack**
    The work of CallStatements with this method name is not to do changes on variables, but to provide the length of each array emerged in our target C code. These length of arrays will be used when we build BDDs for CallStatements with other method name. So the only thing we meet this kind of transitions is to build a BDD representing the change of program counters.
  * **write~init~int**
    CallStatement with this method name can be regarded as another kind of AssignmentStatement, for examples:

    $$\textbf{int flag[2] = \{0, 0, 0\}}$$

    can be seen as:

    $$\textbf{flag\textasciitilde 0 = 0, flag\textasciitilde 1 = 0, flag\textasciitilde 2 = 0}$$

    Our AssignmentStatementEvaluator can handle all of these transitions.
  * **write~int**
    For this kind of method name, there are two cases we have to give a discuss: If the index of the array is an integer, then as same as the last part, CallStatement with this method name can also be regarded as another kind of AssignmentStatement, for examples:

    $$\textbf{flag[2] = 1}$$

    can be seen as:

    $$\textbf{flag\textasciitilde 2 = 1}$$

    Our AssignmentStatementEvaluator can handle all of these transitions.
    However, if the index of the array is a variable, then we could not treat this transition as one simple AssignmentStatement. We have to discuss the length of each arrays we recorded exhaustively before to translate this kind of transitions into BDD.

doi:10.6342/NTU202300384

For examples:

flag[k]= 1

Supposed that the length of the array **flag** is 2, then there are three cases:

if k == 0 then flag~0 = 1

if k == 1 then flag~1 = 1

if k == 2 then flag~2 = 1

We build BDDs for these three cases by utilizing our AssignmentStatementEvaluator and AssumeStatementEvaluator individually, and then do **AND** operation between these BDDs. The result BDD is what we need.

  ∗ **read~int**
  Same as the last part, CallStatements with this method name can also be regarded as another kind of AssignmentStatement, and ur AssignmentStatementEvaluator can handle all of these transitions.

  ∗ **ULTIMATE.dealloc**
  We do not concern with this type of transitions because despite of program counters, these transitions do not change any value of variables aftering doing them. So the only thing we meet this kind of transitions is to build a BDD representing the change of program counters.

- **HavocStatement**
  As the same reason for CallStatement with type ULTIMATE.dealloc, We do not concern with this type of transitions. The only thing we meet this kind of transitions is to build a BDD representing the change of program counters.

- **ForkStatement**
  The work of transitions with type ForkStatement is all done, so the only thing we meet this kind of transitions is to build a BDD representing the change of program counters.

After doing these works, we add program counters in BDDs we just created. These program counters are seen as BDD variables, which are parts of pre-condition and post-condition. The input set of states only do transitions on the corresponding program counter. There is one last thing we need to do, which is to handle unchanged variables. Transition

doi:10.6342/NTU202300384

BDDs we just created only present the variables which represent the pre-condition and the post-condition. Unchanged variables do not be dealt with in these transition BDDs. We have tried to encode these unchanged variables in our transition BDDs, but we face the out-of-memory problem, which means that we cost too much memory to handle these kind of variables. To mitigate this problem, we do not encode these unchanged variables in our transtion BDDs, and we do not do disjunction on these transition BDDs. In other words, we separate the set of transition BDDs into lots of small BDDs to handle unchanged varialbes BDDs more easily. The part we actually handle them is in the process of calculating fixpoints, which will be discussed later.

In addition to the constructions of normal transitions, we also construce self loops at the end of each threads. These self loops are built to avoid losing states when we calculate the fixpoint. Without these self loops, the result fixpoint we calculate is always be empty, and this is not what we expected.

- Translate transitions into BDDs for the property automaton.
  The way we build BDD transitions for the property automaton is as same as the way adopted in building for RCFG. However, Building BDD transitions for the property automaton is more easily because there is only one type of transition which is AsumeStatement in the property automaton. Aftering building BDD transitions for the property automaton, we can get one List of BDD which represents all BDD transitions of the property automaton.

- Calculate the set of initial states.
  To start caluculating the fixpoint formula, we have to obtain the Set of initial states first. The way we get this Set is to trace system RCFG from initial states of each thread. It is important that only transition with type AssignmentStatement or CallStatement can be seen as variable-initialized transition. Transitios with the type AssignmentStatement handle initializations of normal variables, and those with the type CallStatement, which method name is **write~init~int**, handle initializations of array variables.

  Starting from initial states of the thread **ULTIMATE.startEntry**, we will check whether the type of outgoing transitions of the initial state of **ULTIMATE.startEntry** is AssignmentStatement or CallStatement. If the type of the transition is AssignmentStatement, we check whether

doi:10.6342/NTU202300384

the assigned variable is initializeed before this transition. If it is not initialized before, then this transition is an **initialzation transition**, and if it is initialized before, then we end the trace of the thread **ULTIMATE.startEntry** and do the same method to other threads.

For another circumstance, the type of the transition is CallStatement, we directly check the method name of the CallStatement. If the method name equals to **write~init~int**, then this transition is an initialzation transition, and if its method name is not **write~init~int**, then we end the trace of the thread **ULTIMATE.startEntry** and do the same method to other threads.

By doing this method for each thread, we can get numbers representing the program counter of the state. We have to do initializtions before we meet this state for each thread, and all initialization transitions will be filtered out from Set of RCFG transitions. Next, we create one new BDD with all-zero program counters, and do all initial transitions(have been translated into BDD) to get one initial BDD state. The last step is doing **AND** operation between this BDD state and initial states of property automaton(also have been translated into BDD). Finally, we get one Set of initial states and are about to calculating fixpoint.

- Build BDD transitions for the cross product of the system automaton and property automaton.
  Now we have BDD transitions of system RCFG (do not include transitions filtered out when deciding Set of initial states) and property automaton. It is easy to calculate BDD transitions of the cross product of the system automaton and property automaton by doing **AND** operations between each BDD transition of system RCFG and each BDD transition of property automaton.

  Different from the theorem, we do not build a big BDD transition for each RCFG and property automaton. The reason is the way we implement the function which is designed for getting the post state is not as same as the theorem. We took a special approach to handle those **unchanged** variables. If we do not do it separately, the BDD structure for each transition will become very magnificant, and has a tremendous memory cost. This is why we separate all BDD transitions of system RCFG and property automaton. A more detailed instructions of the function used to get the post state will be introduced in the next section.

doi:10.6342/NTU202300384

- Calculate the fixpoint

  In the definition 10 we introduced in the preliminary section, we have automaton $\mathcal{A}_{\mathcal{M}}$, automaton $\mathcal{A}_{\neg p}$, and the following fixpoint formula:

  $$\mathcal{R}_\alpha \equiv \alpha \cap \mu x \cdot (Post(x) \cup I)$$

  $$\mathcal{F}_p \equiv \nu y \cdot \mu x \cdot (Post(x) \cup (Post(y) \cap \mathcal{R}_\alpha))$$

  In our model checker, $\mathcal{A}_{\mathcal{M}}$ represents the RCFG of the C code generaterd by ULTIMATE, which is a safe automaton, and $\mathcal{A}_{\neg p}$ represents the Büchi automaton of the given LTL property. We have one new symbolic automaton by doing cross product between these two automaton. This symbolic automaton, which represents the C program with the LTL property, is not fully expanded. It records the change of each variable but not the exact value of each variable. In other words, We record the relation between different states in this symbolic automaton, so we can not get the value of each variables in its states.

  To implement our model checker, we can separate this formula into two parts: the first part is the calculation of R_α, and the second part is the calculation of $\mathcal{F}_p$.

  To calculate R_α, we first need to design the function $Post()$ which is used to calculate $Post(x)$ in the above fixpoint and calculate the set of initial states and the set of accepting states. The set of initial states has been calculated in the previos step, and the way we deal with the set of accepting states is to check the program counter of the property automaton of all the states we have calculated. If the program counter of the property automaton of the state is at the accepting state of the property automaton, then this state will be left. As we get the fixpoint, we finish calculating and get R_α.

  Greatest fixpoint procedure and Least fixpoint procedure are introduced in Preliminaries, so the last thing we concern about is the function $Post()$, which is designed to get post states. As mentioned before, our $Post()$ is different from the theorem's because the BDDs we translated do not afford the unchanged values. This is the reason our $Post()$ need to handle unchanged values after doing BDD transitions.

  Our $Post()$ has two parameters. The first parameter is the set of input states, and the second parameter is the set of transitions after the process of the cross product. To compute the set of post states, we use the function **restrict()** provided by JAVABDD. The function *restrict*()

doi:10.6342/NTU202300384

restricts the variables in transition to constant true or false. How this is done depends on how the variables are included in the state. For example, there is a transition $t1$ represented by BDD:

$$\mathbf{t1 = (x, y, x', y')}$$

and one state in the input set of states $s1$ represented by BDD:

$$\mathbf{s1 = (x, y\ )}$$

Where **x, x', y, y'** are BDD variables. Then the result after doing *restrict*() is:

$$\mathbf{t1.restrict(s1) = (x', y')}$$

which means BDD varialbe **x** becomes **x'** and **y** becomes **y'** after doing this transition. The value of **x** in the post state will be set as **x'**, and the value of **y** in the post state will be set as **y'**. If the result of **restrict()** is a NULL BDD, meaning the input set of states can not do this transition, so we have nothing to do with it. If the result is not a NULL BDD, we have to handle the unchanged values. For example, there is a transition $t1$ represented by BDD:

$$\mathbf{t2 = (x, x')}$$

and one state in the input set of states $s1$ represented by BDD:

$$\mathbf{s2 = (x, y)}$$

Then the result after doing **restrict()** is:

$$\mathbf{t2.restrict(s2) = (x')}$$

The result of the function *restrict*() implies that the value of **x** becomes **x'** and the value of **y** remains the same. The value of **x** in the post state will be set to **x'**, and the value of **y** in the post state will be set to **y**. Our *Post*() creates a new BDD which represents the combination of the result after doing the function **restrict()**(changed variables) and the rest variables which remain the same(unchanged variables). Finally, we get one post state of the input state after doing the input transition. The value of x in that post state becomes x' and the value of y in the post state remains the same.

This Approach to handle unchanged variables will lead to a decrease in time efficiency, and we will discuss the result in the next chapter.

doi:10.6342/NTU202300384

- Check the emptiness of the fixpoint.
  After finishing all of steps listed above, we can finally check the emptiness of the fixpoint we just calculated. If it is empty, the system accepts the specification, and if it is not empty, the specification does not hold. This is the final result presented by our fixpoint model checker.

4.3 Fairness Issues

When verifing a program, we need to check whether the program satisfies the fairness. Without checking the fairness, we could not guarantee the result of the verification is exactly what we expected. In this section, we discuss fairness issues at two different levels: one is at the execution level, and the other is at the application level.

The system of our model checker using fixpoint is centralized. In other words, our model checker traversed all possible transitions from the set of initial states. However, we still have the fairness issue because we could not ensure that any process that can execute a statement should eventually proceed with that statement. To ensure fairness of the underlying execution model, we use auxiliary variables to formulate Weak Fairness conditions. We encode them in the property automaton: Suppose we have a fairness condition F and a target property p. We write $p' = F \rightarrow p$ as the property that we want to verify under the fairness condition. Then, we generate the automaton $\mathcal{A}_{\mathcal{M}}$ by using the RCFG provided by ULTIMATE, and $\mathcal{A}_{\neg p'}$ is generated by the ULTIMATE's toolchain LTL2BA. The last two things are to construct the cross product of these two automata and to check its emptiness. We take Algorithm 3 as an example:

doi:10.6342/NTU202300384

Algorithm 3: two-process Peterson's algorithm with auxiliary variables

```
//@ ltl invariant positive: $\Box((auxv1 == 1) ==> \Diamond((auxv1 == 2) \vee (auxv1 == 3)))\wedge$
  $\Box((auxv1 == 2) ==> \Diamond(auxv1 == 3))\wedge$
  $\Box((auxv2 == 1) ==> \Diamond((auxv2 == 2) \vee (auxv2 == 3)))\wedge$
  $\Box((auxv2 == 2) ==> \Diamond(auxv2 == 3)) ==> (\Diamond AP(x < 1))$
int flag1 = 0, flag2 = 0, turn = 0
int $auxv1 = 0, auxv2 = 0$
int x = 0
proc thread0
    flag1 = 1, turn = 1
    int f21 = flag2
    int t1 = turn
    $auxv1 = 1$
    while f21==1 and t1==1 do
        $auxv1 = 2$
        f21 = flag2
        t1 = turn
    end
    $auxv1 = 3$
    int y1 = 0
    y1 = x
    y1++
    x = y1
    flag1 = 0
end
proc thread1
    flag2 = 1, turn = 0
    int f12 = flag1
    int t2 = turn
    $auxv2 = 1$
    while f12==1 and t2==1 do
        $auxv2 = 2$
        f12 = flag1
        t2 = turn
    end
    $auxv2 = 3$
    critical section
end
```

doi:10.6342/NTU202300384

In Algorithm 3, we show the design of the fairness condition for the Peterson's algorithm. The blue part in the LTL property is the fairness condition of the target program, the red part is the property we want to check, and auxiliary variables **auxv1** and **auxv2** are added in the process *thread*0 and the process *thread*1. The first line of the fairness condition:

$$\Box((auxv1 == 1) ==> \Diamond((auxv1 == 2) \vee (auxv1 == 3)))$$

ensures that the process *thread*0 is given a chance to do its statements, and the second line:

$$\Box((auxv1 == 2) ==> \Diamond(auxv1 == 3))$$

ensures that the process *thread*0 will eventually get a chance to check the loop condition and proceed to the corresponding following statement. The third line and the fourth line work similarly as same as the first and the second lines. This kind of auxiliary variables enforce Weak Fairness for the system model. The time cost of the verification without auxiliary variables reduces significantly, and we will show the data in the next section.

doi:10.6342/NTU202300384

Chapter 5

Experiments and Results

In this chapter, we will introduce the way we implement our fixpoint model checkers with some program as examples. Our model checkers will illustrste those programs and check whether the specification in the example program holds.

To completely implement our verification work, we need to degine an XML file which represents the whole process of our work, which starts from a C file and ends up with the result of our fixpoint model checker.

```
 1      <rundefinition>
 2      <name>Ultimate Toolchain</name>
 3      <toolchain>
 4
 5      <plugin id="de.uni_freiburg.informatik.ultimate.plugins.generator.cacsl2boogietranslator"/>
 6      <plugin id="de.uni_freiburg.informatik.ultimate.boogie.procedureinliner"/>
 7      <plugin id="de.uni_freiburg.informatik.ultimate.boogie.preprocessor" />
 8      <plugin id="de.uni_freiburg.informatik.ultimate.plugins.generator.rcfgbuilder"/>
 9      <plugin id="de.uni_freiburg.informatik.ultimate.ltl2aut" />
10      <plugin id="model-checking algorithms here" />
11
12      </toolchain>
13      </rundefinition>
```

Figure 5.1: The sample toolchain XML file.

As mentioned in the previous chapters, some tools designed by ULTIMATE are used to deal with RCFG-related and never claim automata processing. These tools are included in the toolchain in this thesis, and our model checker will be implemented in line 10 in Figure 5.1.

doi:10.6342/NTU202300384

5.1 Comparison between Different Algorithms

In this section, we will compare the result of the verification between our symbolic model checker (fixpoint approach) and the explicit model checker (double DFS approach) by using SCANTU system. We will concentrate the result of doing verifications on different kinds of C files and properties, such as the incorrect property with the correct code, the incorrect code with the correct property, and the code and the property are all correct. The target programs includes Algorithm 3 and programs shown at the end of this section.

In Algorithm 3, we verified the correct C program of Peterson's algorithm with one correct property ($\Diamond AP(x < 1)$). The verification result of Algorithm 3:

	Double DFS	Fixpoint
with auxiliary variables	37,16ms	203.17ms
no auxiliary variables	33.98ms	179.31ms

In Algorithm 4, we verified the correct C program of Peterson's algorithm with another correct property ($\Box(AP(x == 0) ==> \Diamond AP(x == 2))$). The result of the verification of Algorithm 4:

	Double DFS	Fixpoint
with auxiliary variables	4001.83ms	125019.97ms
no auxiliary variables	196.37ms	11707.37ms

There are some things worth noticing in results of these two cases. The first information We observed is that the verification result of Algorithm 3 spent less time than the result of Algorithm 4, through C files and the properties are all correct. The key point is that the fixpoint computed in Algorithm 3 becomes the empty set more earlier than the computation of Algorithm 4. We can find some clues from the number of reachable states of these two cases. The first case has only one reachable state which is the initial state when it is verified with no auxiliary variables, which means the initial state can not do any transition. The second case has 1968 reachable states under the same premise, so our model checker has to spend time more time computing the fixpoint.

The second thing we observed is that verifications with auxiliary variables spent more time than verifications with no auxiliary vairalbes. The result is affected by the number of accepting reachable states, just like the reason we presented in the previous case. We take Algorithm 4 as example:

doi:10.6342/NTU202300384

The number of accepting reachable states in the process of verification with auxiliary variables is 3540, and 1968 with no auxiliary variables. We can not guarantee that the more transitions there are, the more time the model checker will cost, but we could infer that the number of reachable states is positively related to time cost.

Next, we show the rest of results: In Algorithm 5, we verified the correct C program of Peterson's algorithm with one incorrect property ($\Box(AP(x == 2) ==> \Diamond AP(x == 0))$). The verification result of Algorithm 5:

	Double DFS	Fixpoint
with auxiliary variables	90.81ms	23353.62ms
no auxiliary variables	76.14ms	3141.73ms

In Algorithm 6, we verified the incorrect C program of Peterson's algorithm with one correct property ($\Box(AP(x == 0) ==> \Diamond AP(x == 2))$). We deleted the critical section in Thread1 in Algorithm 6, so the value of x will be 1 when all processes are all finished. The verification result of Algorithm 6:

	Double DFS	Fixpoint
with auxiliary variables	98.44ms	83213.20ms
no auxiliary variables	71.56ms	6376.86ms

We can observed that the model checker with double DFS approach has a high level of performance when there is something wrong in the property or the C file. The model checker with fixpoint approach is struggled in the exponential explosion of states when it computes the fixpoint. An incorrect property always leads to a larger number of reachable states, thus reduces the efficiency of the model checker.

In the verification result of Algorithm 6, the deletion of the critical section leads to a smaller number of reachable states, so the time cost of it is less then the verification result of Algorithm 5. It is intuitive that the deletion reduces the difficulty of computing fixpoint.

doi:10.6342/NTU202300384

Algorithm 4: two-process Peterson's algorithm with another right property and right code

```
//@ ltl invariant positive: $fairness\ condition$ ==>
  $\Box(AP(x == 0) ==> (\Diamond AP(x == 2)))$
int flag1 = 0, flag2 = 0, turn = 0
int $auxv1 = 0, auxv2 = 0$
int x = 0
proc thread0
    flag1 = 1, turn = 1
    int f21 = flag2
    int t1 = turn
    $auxv1 = 1$
    while f21==1 and t1==1 do
        $auxv1 = 2$
        f21 = flag2
        t1 = turn
    end
    $auxv1 = 3$
    int y1 = 0
    y1 = x
    y1++
    x = y1
    flag1 = 0
end
proc thread1
    flag2 = 1, turn = 0
    int f12 = flag1
    int t2 = turn
    $auxv2 = 1$
    while f12==1 and t2==1 do
        $auxv2 = 2$
        f12 = flag1
        t2 = turn
    end
    $auxv2 = 3$
    critical section
end
```

doi:10.6342/NTU202300384

Algorithm 5: two-process Peterson's algorithm with right code but incorrect property

```
//@ ltl invariant positive: $fairness\ condition$ ==>
    $\Box(AP(x == 2) ==> (\Diamond AP(x == 0)))$
int flag1 = 0, flag2 = 0, turn = 0
int $auxv1 = 0, auxv2 = 0$
int x = 0
proc thread0
    flag1 = 1, turn = 1
    int f21 = flag2
    int t1 = turn
    $auxv1 = 1$
    while f21==1 and t1==1 do
        $auxv1 = 2$
        f21 = flag2
        t1 = turn
    end
    $auxv1 = 3$
    int y1 = 0
    y1 = x
    y1++
    x = y1
    flag2 = 0
end
proc thread1
    flag2 = 1, turn = 0
    int f12 = flag1
    int t2 = turn
    $auxv2 = 1$
    while f12==1 and t2==1 do
        $auxv2 = 2$
        f12 = flag1
        t2 = turn
    end
    $auxv2 = 3$
    critical section
end
```

doi:10.6342/NTU202300384

Algorithm 6: two-process Peterson's algorithm with the right property but incorrect code

```
//@ ltl invariant positive: $fairness\ condition$ ==>
  $\Box(AP(x == 0) ==> (\Diamond AP(x == 2)))$
int flag1 = 0, flag2 = 0, turn = 0
int $auxv1 = 0, auxv2 = 0$
int x = 0
proc thread0
    flag1 = 1, turn = 1
    int f21 = flag2
    int t1 = turn
    $auxv1 = 1$
    while f21==1 and t1==1 do
        $auxv1 = 2$
        f21 = flag2
        t1 = turn
    end
    $auxv1 = 3$
    int y1 = 0
    y1 = x
    y1++
    x = y1
    flag1 = 0
end
proc thread1
    flag2 = 1, turn = 0
    int f12 = flag1
    int t2 = turn
    $auxv2 = 1$
    while f12==1 and t2==1 do
        $auxv2 = 2$
        f12 = flag1
        t2 = turn
    end
    $auxv2 = 3$
    no critical section
end
```

doi:10.6342/NTU202300384

5.2 Comparison between Different Number of Threads

In this section, we use a simple multithread program to discuss the impact of memory and time cost when increasing the number of threads. The property we used in this example is safe, so we can ensure that it is fair when we compute the fixpoint and concentrate on the influence of the number of threads. We also design different experimental settings by using different correctness combinations of temporal properties and C programs to produce more diverse comparison results.

Algorithm 7 is the simple program we just mentioned. We first discuss the combinition with the correct program and property:

Algorithm 7: Simple N-process Multithread Program with the correct program and property

```
//@ ltl invariant positive: $\Diamond AP(x == N)$
**int** x = 0
**proc** *thread*
    x = x + 1
**end**
**proc** *main*
    create and join thread N times.
**end**
```

Then we record the time cost of this combination:

thread number (N)	N = 2	N = 4	N = 6	N = 8	N = 10
Double DFS	90.55ms	254.66ms	917.51ms	1570.30ms	3797.24ms
Fixpoint	204.77ms	617.52ms	7757.02ms	835836.08ms	-

We can observe that as the number of threads increases, the time spent grows on both model checkers, and the growth rate of the model checker with fixpoint approach is more dramatic than that of the model checker with double DFS approach.

Next we discuss the influence of the incorrect property, Algorithm 8 is the program with the incorrect property and the correct program:

doi:10.6342/NTU202300384

Algorithm 8: Simple N-process Multithread Program with the correct program but incorrect property

```
//@ ltl invariant positive: $\Diamond AP(x == N + 1)$
int x = 0
proc thread
    x = x + 1
end
proc main
    create and join thread N times.
end
```

As same as before, we record the time cost of this combination:

thread number (N)	N = 2	N = 4	N = 6	N = 8	N = 10
Double DFS	54.56ms	89.88ms	109.36ms	151.74ms	217.14ms
Fixpoint	169.76ms	584.61ms	8023.92ms	860803.38ms	-

We can find that the time cost of the model checker using double DFS is almost at the same level, and the time cost of the model checker using fixpoints is not much difference from the previous comparison. Once the first model checker finds the accepting cycle, the process of the verification is end and this is the reason it is not strongly affected by the number of threads. However, the second model checker still dramatically influenced by the the number of threads because it still takes a long time to compute the fixpoint.

Last we discuss the influence of the incorrect program, Algorithm 9 is the program with the correct property but the incorrect program:

doi:10.6342/NTU202300384

Algorithm 9: Simple N-process Multithread Program with the correct property but incorrect program

```
//@ ltl invariant positive: $\Diamond AP(x == N)$
**int** x = 0
**proc** *thread1*
    Do nothing
**end**
**proc** *thread2*
    x = x + 1
**end**
**proc** *main*
    create and join thread1 1 times. create and join thread2 N-1 times.
**end**
```

the record of the time cost:

thread number (N)	N = 2	N = 4	N = 6	N = 8	N = 10
Double DFS	55.99ms	80.87ms	109.81ms	131.04ms	153.38ms
Fixpoint	152.28ms	498.90ms	5223.06ms	543766.59ms	-

We can find that the time cost of the first model checker is roughly the same as the time cost we recorded in the previous comparison. It always quickly find the accepting cycle during the verification process. The model checker using double DFS spends more time only if there is nothing incorrect.

The time cost of the second model checker is less than the time cost we recorded in the previous comparison. The reason is just mentioned in the previous section: the number of the reachable accepting states is less than the previous comparison, and we have a conclusion that the growth of the number of threads is more influential for the model checker with fixpoint appraoch than the model checker with double DFS approach.

Finally, we discuss why our model checker is inefficient. As we just mentioned in the previous chapter, we handle unchanged variables when calculating fixpoints because of the out-of-memory problem. We confirm that this is the reason why our model checker is more inefficient than the explicit one. We spent too much time computing *Post*(). In other words, the separation of transitions mitigates the out-of-memory problem, but at the cost of spending too much time.

doi:10.6342/NTU202300384

Chapter 6

Conclusion

Model checking has been proposed for a long time, helping the progress of formal verification. More and more model checking tools have been developed. We have reviewed the previous studies related to the comparison between explicit-state model checking and symbolic model checking. In this thesis, we have implemented a symbolic model checker using fixpoints with Ultimate, and compared our model checker with an explicit model checker. We now summarize the contributions of this thesis and suggest directions for future research.

6.1 Contributions

Our research contributes to the field of formal software verification with the following perspectives:

- *Implementation of a symbolic model checker using fixpoints*
  With the support of ULTIMATE and JAVABDD, we have succesfully built a symbolic model checker using fixpoints, though its efficiency is not as good as expected. Our model checker can directly verify C programs against temporal properties, and, unlike NUSMV does not need other modeling languages.

- *Extension of SCANTU*
  We have extended SCANTU with one more model checking engine. Now there are two model checking engines in SCANTU: one using double DFS and the other using fixpoints. Users can choose different model checkers under different circumstances.

- *Implications of the comparison results*
  Thanks to SCANTU, we have a very convenient way to do comparisons between different model checkers using different approaches. We have done some comparisons with different experimental settings in this thesis and discussed experimental results. We confirm that:

doi:10.6342/NTU202300384

1. Under normal circumstances, when computing Post() the transition is the disjunction of thr transitions in the automaton which is the synchronized product of system automaton and property automaton. However, we faced the out-of-memory problem when we did the disjunction of those transitions. To get around the problem, we separate the transition into different small BDDs. We found that the computation of Post() takes a long time because of the separation of the transition. So, the separation of the transition does not seem to be a good idea.
2. The model checker using fixpoints is affected by the number of threads more dramatically than the model checker using double DFS.
3. The model checker using double DFS uses less time when there are errors in the input program or property.

6.2 Future Work

There are several remaining issues for reseachers to futher explore:

- *Use different BDD packages*
  First, we use JavaBDD to build transition BDDs. There are lots of BDD packages, such as CUDD, CacBDD, JDD, and so on. Doing comparisons between the implementations with different BDD packages and discussing the strengths and weaknesses of different BDD packages are good to try.

- *Improve the efficiency of our symbolic model checker*
  Second, we have tried to improve efficiency with different techniques. However, the experimental results of our symbolic model checker using fixpoints are not as good as expected. Due to the limited time, there are still other techniques which may help us improve the efficiency but are not adopted in our tools, for instance, antichains.

- *More model checkers with different algorithms in the comparative study*
  Last but not least, the last future work is to compare more algorithms, in other words, to expand the scope of our thesis. By this way, a more effcient model checking approach for different circumstances can be found more easily.

doi:10.6342/NTU202300384

References

[1] David Baelde. Least and greatest fixed points in linear logic. *ACM Transactions on Computational Logic (TOCL)*, 13(1):1–44, 2012.

[2] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207. Springer, 1999.

[3] J. Richard Büchi. On a decision method in restricted second order arithmetic. In *The Collected Works of J. Richard Büchi*, pages 425–435. Springer, 1990.

[4] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and Lain-Jinn Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[5] Jo-Chuan Chou. A comparative study of depth-first search algorithms for automata-based model checking. Master's thesis, National Taiwan University, Jan 2021.

[6] Yaacov Choueka. Theories of automata on ω-tapes: A simplified approach. *Journal of Computer and System Sciences*, 8(2):117–141, 1974.

[7] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[8] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, pages 52–71. Springer, 1981.

[9] Federico Echenique. A short and constructive proof of Tarski's fixed-point theorem. *International Journal of Game Theory*, 33(2):215–218, 2005.

[10] Cindy Eisner and Doron Peled. Comparing symbolic and explicit model checking of a software system. In *International SPIN Workshop on Model Checking of Software*, pages 230–239. Springer, 2002.

doi:10.6342/NTU202300384

[11] Saul Kripke. Semantical considerations of the modal logic. *Studia Philosophica*, 1, 2007.

[12] Saul A. Kripke. Semantical analysis of modal logic I normal modal propositional calculi. *Mathematical Logic Quarterly*, 9(5-6):67–96, 1963.

[13] Hong-Yang Lin. Tool support for automata-based model checking of multithreaded programs. Master's thesis, National Taiwan University, Jan 2021.

[14] Carroll Morgan, Annabelle McIver, and Karen Seidel. Probabilistic predicate transformers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(3):325–353, 1996.

[15] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (FOCS 1977)*, pages 46–57. IEEE, 1977.

[16] Amir Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[17] Roberto Sebastiani, Stefano Tonetta, and Moshe Y. Vardi. Symbolic systems, explicit properties: on hybrid approaches for LTL symbolic model checking. In *International Conference on Computer Aided Verification*, pages 350–363. Springer, 2005.

[18] Thomas Wahl. Fairness and liveness.

doi:10.6342/NTU202300384