SOFTWARE MODEL CHECKING FOR DISTRIBUTED APPLICATIONS USING HYBRIDIZATION OF CENTRALIZATION AND CACHE APPROACHES

HING RATANA

UNIVERSITI SAINS MALAYSIA

2024

SOFTWARE MODEL CHECKING FOR DISTRIBUTED APPLICATIONS USING HYBRIDIZATION OF CENTRALIZATION AND CACHE APPROACHES

by

HING RATANA

Thesis submitted in fulfilment of the requirements
for the degree of
Doctor of Philosophy

April 2024

ACKNOWLEDGEMENT

I would like to express my deepest gratitude to all those who have played a part in the realization of this thesis, as their support and guidance have been invaluable.

First and foremost, I am deeply indebted to my supervisor, Associate Professor Dr. Chan Huah Yong, and Dr. Sharifah Mashita Sayed Mohamad, for their unwavering guidance, patience, and expertise. Their insightful feedback, encouragement, and dedication to my growth as a researcher have been instrumental in shaping this work.

I am immensely grateful to the members of my thesis committee, Associate Professor Dr. Wong Li Pei, Dr. Lim Chia Yean, and Dr. Sukumar Letchmunan, for their valuable insights, constructive criticism, and scholarly contributions. Their expertise in their respective fields has immensely enriched this research and broadened my understanding.

Special thanks go to my wife and my children for their unwavering love, understanding, and support. Their encouragement, patience, and belief in me have been a constant source of motivation, and I am forever grateful.

Thank you all for being an integral part of this transformative journey and for helping me reach this milestone in my academic career.

TABLE OF CONTENTS

LIST OF TABLES

Page

LIST OF FIGURES

Page

LIST OF SYMBOLS

M	Kripke structure
f	A formula of temporal logic
s	All states
S	A set of process states
L_I	A set of input labels
L_O	A set of output labels
T	A set of transitions
s_0	An initial state
τ	An unobservable action
$\emptyset$	Non-empty set
Act	A set of actions
I	Invisible actions
m	A message stream
SP_{req}	Request stream pointer
SP_{resp}	Response stream pointer
T	Thread
T	Communication trace
V	Visible actions
Φ	A single process
α	An action

LIST OF ABBREVIATIONS

API	Application Programming Interface
BFS	Breadth-First Search
BLAST	Berkeley Lazy Abstraction Software Verification Tool
CMC	Compositional Model Checking
DFS	Depth-First Search
GDB	Gnu Debugger
GNU	GNU's Not Unix
I/O	Input/Output
IP	Internet Protocol
IPC	Interprocess Communication
JNH	JPF Network Asynchronous Systems Using A Hybrid Approach
JPF	Java PathFinder
JPF-JVM	JPF Java Virtual Machine
JPF-Nas	JPF Network Asynchronous Systems
JVM	Java Virtual Machine
LTS	Labeled Transition System
MJI	Model Java Interface
NASA	National Aeronautics and Space Administration
NET-IOCACHE	Network Input/Output Cache
NOSA	NASA Open Source Agreement
OS	Operating System
RR Tree	Request and Response Tree
SLAM	Software Language Analysis and Modeling
SUT	System Under Test
TCP	Transmission Control Protocol
TS	Transition System
UDP	User Datagram Protocol
VM	Virtual Machine

LIST OF APPENDICES

MODEL PERISIAN MENYEMAK APLIKASI TEREDAR MENGGUNAKAN PENDEKATAN HIBRID PEMUSATAN DAN CACHE

ABSTRAK

Membangunkan sistem teragih yang boleh dipercayai menimbulkan cabaran yang ketara disebabkan oleh sifat bukan penentu bagi pelaksanaan *thread* dan proses, serta saluran komunikasi. Penyemakan model perisian menawarkan cara untuk mengesahkan ketepatan sistem dengan menganalisis secara menyeluruh semua laluan pelaksanaan program. Walau bagaimanapun, penyemak model kod bait sedia ada, yang mampu menyemak pelbagai proses, mengalami letupan ruang keadaan dan overhed pengiraan. Tesis ini memperkenalkan Java PathFinder (JPF)-Nas-Hybrid (JNH), penyemak model baru yang menangani masalah ini. JNH menggunakan model komunikasi antara proses (IPC) yang direka bentuk semula dan menyepadukan mekanisme caching berskala. Mekanisme ini menyimpan data komunikasi antara proses dengan cekap, mengurangkan overhed pengiraan dan menyatakan letupan ruang semasa pemeriksaan model. Dengan mengoptimumkan penggunaan sumber dan meminimumkan overhed, JNH meningkatkan prestasi pengesahan dengan ketara. Penambahbaikan utama termasuk pembangunan mekanisme caching berskala yang disepadukan ke dalam model IPC pemusatan, penempatan semula permintaan dan pokok tindak balas, dan pemprosesan data dalam ketulan berbilang bait. Penciptaan JNH melibatkan lanjutan daripada sistem teras JPF dan mengubah suai perpustakaan rangkaian Java. Selain itu, tesis ini meneroka strategi pengesanan pepijat, membezakan antara pepijat tempatan dan global, dan menilai pelbagai strategi carian untuk meneroka ruang keadaan program yang diedarkan. Melalui ujian komprehensif

dan analisis statistik, penyelidikan memberikan pandangan tentang pendekatan pengesanan pepijat yang berkesan, memajukan lagi kaedah penyemakan-model.

SOFTWARE MODEL CHECKING FOR DISTRIBUTED APPLICATIONS USING HYBRIDIZATION OF CENTRALIZATION AND CACHE APPROACHES

ABSTRACT

Developing reliable distributed systems poses significant challenges due to the non-deterministic nature of thread and process execution, as well as communication channels. Software model checking offers a means to verify system correctness by exhaustively analyzing all program execution paths. However, the existing bytecode model checker, capable of verifying multiple processes, suffers from state space explosion and computational overhead. This thesis introduces Java PathFinder (JPF)-Nas-Hybrid (JNH), a novel model checker addressing these limitations. JNH employs a redesigned inter-process communication (IPC) model and integrates a scalable caching mechanism. This mechanism efficiently stores communication data between processes, mitigating computational overhead and state space explosion during model checking. By optimizing resource utilization and minimizing overhead, JNH significantly improves verification performance. Key enhancements include the development of a scalable caching mechanism integrated into the centralization IPC model, relocating request and response trees, and processing data in multi-byte chunks. JNH's creation involves extending from the JPF-core system and modifying Java network libraries. Additionally, the thesis explores bug detection strategies, distinguishing between local and global bugs, and evaluates various search strategies to explore distributed program state spaces. Through comprehensive testing and statistical analysis, the research provides insights into effective bug detection approaches, further advancing model-checking methodologies.

CHAPTER 1

INTRODUCTION

1.1 Background

Modern society relies heavily on complex software systems, which can be found in various fields such as banking, automotive, shopping, entertainment, and so on. These systems are often large, distributed, and exhibit guaranteed qualities. They also consist of multiple processes spread across multiple computing devices, executing independently. While this provides the advantages of increased performance and scalability it also makes such systems much harder to test due to partial failure and asynchrony (McCaffrey, 2016). Partial failure refers to the components in distributed applications that can fail along the way, resulting in incomplete results or data. Asynchrony is the indeterminateness of ordering and timing within a distributed system that often leads to solutions with a high degree of complexity. Avoiding distributed system bugs also requires reasoning about the integration between nodes and must tolerate the failure of the underlying hardware. In addition, the probability of human error in either design, implementation, or operation also contributes to system bugs. Therefore, developing a reliable distributed system is a very challenging task.

In addition, complex software systems are composed of independent sub-systems that interact with each other, and these interactions are referred to as atomic events. These atomic events are subject to both measurable and non-measurable uncertainties. Measurable uncertainties are typically characterized by assigning probabilities to future events, while non-measurable uncertainties are modeled as non-determinism and resolved by an external scheduler.

Additionally, the most common aspects of complex software systems involve cost and time. Each event in the system incurs a cost and takes a certain amount of time to complete. While cost and time can be modeled as non-negative real values, their accumulation over the dynamics of the system is not directly comparable. The total cost of a set of events can be obtained by summing their costs, but the total time depends on the concurrency among the events. The concurrent interactions of multiple processes are non-deterministic, making these systems very hard to test. The interleaving between threads within a process also increases the challenges for software testers. Moreover, setting up an environment and instrumenting the software under test is also time-consuming and expensive.

Data race and deadlocks are known to be fundamental problems in any concurrent software system (Chen et al., 2022; Cai et al., 2021; Yuan et al., 2021; Bagherzadeh et al., 2020; Tu et al., 2019). A data race occurs when two threads are trying to access the same shared variable at the same time without any proper synchronization, e.g., not protected by a common lock. Deadlock occurs when two or more operations circularly wait for each other to release the acquired resources. Unlike concurrent software systems, distributed systems are involved in not only local concurrency bugs, which come from thread interleaving within a process but also concurrency bugs in globally distributed components, which come from inter-process communication interleaving.

As society increasingly relies on complex software systems, it is important not only to ensure their correctness but also to subject them to rigorous analysis. The most common way to analyze such systems is through testing (Hsaini et al., 2019; Moutai et al., 2019; Stuardo et al., 2019). Once the system is designed, it is tested against a

finite set of test cases to verify that it behaves as expected. While testing can be effective if the test cases are carefully chosen with relevant domain expertise, it is important to acknowledge that testing cannot guarantee success for all possible behaviors of the system.

At the opposite end of the analysis spectrum, formal methods employ mathematical techniques to determine whether a system satisfies a given property for all possible outcomes. One important sub-discipline of formal methods is model checking.

Model checking (Baier & Katoen, 2008; Clarke, Henzinger, & Veith, 2018) is a technique to detect property violations in a concurrent system by exploring every possible execution path. Accordingly, every possible state of the system is checked against given properties. This technique is very useful for the quality assurance of safety-critical systems and core algorithms/protocols of large systems. Model checking was originally developed for hardware verification, but the concept of state space exploration has been applied to a wide range of software systems as well.

The research on applying model checking for distributed systems has gained popularity in recent years due to two main reasons (Muscholl, 2018). First, distributed systems are error-prone, because programmers must consider all possible effects induced by different scheduling of events. Second, testing, which is widely used for certifying sequential programs, tends to have low coverage in distributed settings, because bugs are usually difficult to reproduce. They may happen under very specific thread schedules, and the likelihood of taking such corner-case schedules may be very low. Therefore, automated verification techniques represent crucial support in the development of reliable distributed systems.

Conventional model checkers for distributed systems necessitate the utilization of abstract modeling languages like TLA+ (Lamport, 1994), PlusCal (Lamport, 2009), Coq (Barras et al., 1999), and SPIN (Holzmann, 1997). This approach demands a substantial investment of developer effort and does not guarantee the identification of all bugs in the system implementation. These traditional model checkers can only detect bugs within the specified system model and have no way of finding bugs in the actual implementation (Anand, 2020).

For instance, Amazon employs TLA+ for system verification (Newcombe et al., 2015). However, a drawback of TLA+, along with other comparable specification languages, is their application to a system model rather than the actual implementation. There is still a significant gap between the system specification and its implementation, making the system prone to bugs at the implementation level. A noteworthy illustration of this occurred in 2017 when a bug in Amazon S3 resulted in $150 million in damages for companies relying on Amazon Web Services (Condliffe, 2017). Instances of such incidents are growing more prevalent.

A novel approach within the research community involves applying model checking directly to the actual implementations of distributed systems. The direct verification of real-world implementations enhances confidence in meeting software safety requirements. It's essential to recognize that adherence to system design specifications does not ensure compliance in the implementation phase. Numerous bugs related to concurrency, including race conditions, deadlocks, and assert violations, often stem from programming errors during implementation. Verification during the design phase cannot conclusively guarantee the absence of bugs in the final deliverables. Concrete model checkers (Anand, 2018, 2020; Artho et al., 2017;

Deligiannis et al., 2016; Guerraoui & Yabandeh, 2011; Guo et al., 2011a; Killian et al., 2007; Laroussinie & Larsen, 1998; Leesatapornwongsa et al., 2014; Lukman et al., 2019a; Musuvathi et al., 2008; Yabandeh et al., 2009; Yang et al., 2009a) concentrate on testing and debugging unmodified distributed systems to identify failures, crashes, and violations of user-defined properties. They are model checker tools for distributed systems, which are designed to address specific programming languages, such as Go (Anand, 2020), or to examine the systems operating at the operating system level to identify bugs. Furthermore, concrete model checkers suffer from massive state space explosion and programming language coverage.

Unlike any other concrete model checkers, the Java Pathfinder (Visser & Mehlitz, 2005) or JPF model checks bytecode rather than native code or operating system code. This approach reduces a large number of unrelated state spaces. Additionally, JPF offers fundamental support for verifying distributed systems. It serves as a framework for the analysis of Java bytecode, with its core functionality centered around an explicit state model checker. Developed by the Robust Software Engineering Group at NASA Ames Research Center, JPF has been open source since 2005 under the NOSA 1.3 license (Artho & Visser, 2019).

JPF extends its support to the model checking of the actual implementations (Shafiei & Mehlitz, 2014), and is the only bytecode model checker that supports model checking of distributed systems (Artho & Visser, 2019). Notably, it operates directly on the code without necessitating the remodeling of the system through tools like TLA+, PlusCal, Coq, or SPIN. As it directly handles bytecode, JPF is versatile and applicable to the verification of code compiled into bytecode, including languages like Java, Kotlin, Scala, Groovy, Clojure, Go, and more.

Functioning as an open-source framework, JPF is built around a Virtual Machine (VM) engine capable of interpreting Java bytecode and customizable to meet user specifications. The framework boasts various features, such as state storage, state matching, configurable execution semantics for bytecode instruction sets, and scheduling policies for state space exploration. The standard JPF core distribution includes default checks for generic properties, including the detection of unhandled exceptions, deadlocks, and data races.

The core of JPF consists of a customized Java Virtual Machine (JVM) dedicated to executing bytecode for model checking. As a result, it does not cover native codes and low-level operating system codes, thereby overcoming massive amounts of unrelated state space. JPF core offers fundamental components for model-checking distributed systems and distinguishes itself with its exceptional extensibility, achieved through a collection of stable interfaces and runtime-configurable components.

Various extensions are available, including symbolic execution, directed automated random testing, random choice generation, configurable state abstractions, heuristics for bug detection, alternative state space search strategies, temporal property verification, and more. All these extensions can be integrated into JPF using its well-designed extension mechanisms without modifying the core code of JPF. Moreover, JPF provides a comprehensive infrastructure for building, testing, and configuring, making it easier to create new extensions and applications.

JPF has been around for 15 years and has been open-sourced for nine years, which has led to a large and active user community worldwide. The Java PathFinder workshop (Sherman et al., 2023) is to bring together members of this community,

including researchers, developers, and users, to showcase their current work and discuss future directions for the framework. The workshop series provides a platform for presenting ongoing JPF-related research and promotes collaboration among participants to plan the development of the JPF framework and its community.

JPF comes with a default search, depth-first search, or *backtracking* feature that is utilized to capture different scheduling of threads within a process. This feature enables JPF to explore different orderings of concurrent transitions within the system under test (SUT), where executing them in different orders may result in different behaviors of the process. However, JPF's search is limited to individual processes and does not account for inter-process communications, which are essential in distributed systems to find global bugs. Communication channels between processes can lead to different behaviors based on the ordering in which they are accessed. Therefore, a mechanism is necessary to capture the various orderings of concurrent transitions involving inter-process communications.

This thesis focuses on the model checking of the actual implementation of distributed systems using bytecode executions. Employing bytecodes, as opposed to other concrete model checkers that check on native or operating system-specific codes, significantly reduces the unrelated state space. Furthermore, this approach allows customization of the model checker for different JVM languages. Additionally, the model checking of bytecodes has a more substantial impact on verifying distributed systems operating on the cloud, such as server-side Java or Scala.

The below sections explain the details of the research problem, research questions, research motivation, research objectives, research scope, research contributions, and finally thesis organization.

1.2 Research Problem

Two primary approaches have been employed for model-checking distributed systems at the bytecode level executions: centralization (Artho & Garoche, 2006; Barlas & Bultan, 2007; Ma et al., 2013; Nakagawa et al., 2005; Stoller & Liu, 2001) and caching (Artho et al., 2008, 2009; Leungwattanakit et al., 2011, 2014). The centralization (Shafiei & Mehlitz, 2014) at the model checker level involves capturing multiple processes and checking for both local and global bugs. On the other hand, the caching (Artho et al., 2008, 2009; Leungwattanakit et al., 2011) approach entails model checking one process at a time, allowing other processes to run in their native environments. This technique aims to minimize the state space by examining only one process at a time. However, the caching technique faces challenges related to state synchronization among processes during the backtracking procedure, and the technique does not cover the states of communication between processes. This thesis focuses on the model checker level centralization, and the following sections elaborate on the research problem associated with centralization at the model checker level.

The first major problem in model checker level centralization is the state space explosion, as this technique covers all possible reachable states of multiple processes in distributed systems. The second problem with centralization (Shafiei & Mehlitz, 2014) is that the technique requires careful development of the Inter-Process Communication (IPC) model, which facilitates communication between multiple processes. The choice of IPC design can significantly impact the approach's effectiveness. The current IPC uses two "ArrayByteQueue" buffers—one for the server to send data to the client and another for the client to send data to the server. These buffers store communication and process data byte by byte. For example, the server may send "hello" to the client, and the client responds with "world!". During

state exploration, the centralization process writes one character to the queue, moves to the next state, removes the same character from the queue, and updates the state accordingly. Unfortunately, the write and read operations (which write and remove data from the queue) of the data streams "hello" and "world!" result in computational limitations.

Finally, the limitation imposed by the state space explosion (Clarke et al., 2011a) makes centralization impractical for exploring the state space of distributed systems. This challenge, known as state space explosion, arises when the exponential growth of system states becomes so extensive that the centralization technique becomes computationally impractical. With increasing system complexity, the technique must be able to manage a growing number of visited states, handle thread interleaving, and address other complexities. Therefore, the model checker may take an infinite of time.

1.3 Research Questions

The overall goal of the thesis is to address computational challenges encountered by distributed systems during bytecode-level executions, particularly in the centralized approach. The research to be conducted is guided by the following research questions:

1. What are the key considerations in designing a caching mechanism to store and manage multiple communication data between processes?
2. How does integrating the caching mechanism into the model checker level centralization effectively address computation overhead?

3. How can the hybridization of centralization and cache approaches be validated for mitigating the state space explosion problem through bug injection and different search strategies?

1.4 Research Motivation

Software model checking is a formal verification technique used to ensure the correctness of a system by exhaustively exploring its state space. In distributed systems, where multiple processes work together to achieve a common goal, model checking can help detect errors and ensure the system's correctness.

One key motivation for using software model checking for distributed systems is the need to guarantee that the system satisfies its requirements under all possible system configurations and execution scenarios. This is particularly important in safety-critical systems, where even a small error could have big impact consequences. By using model checking, developers can explore all possible system states and execution paths to ensure that the system behaves as expected and meets its safety requirements.

Another motivation for using software model checking for distributed systems is the need to handle complex interactions between multiple processes. In distributed systems, processes communicate and coordinate their actions to achieve a common goal. This coordination can be challenging to get right, especially when there are multiple possible execution paths that the system can take. Model checking can help developers ensure that all possible interaction scenarios have been considered and that the system behaves as expected in all of them.

Finally, software model checking can also help detect subtle errors that may be difficult to find through testing or other methods. Distributed systems are often complex and may exhibit unexpected behavior in certain scenarios. Model checking

can help detect these errors by exploring all possible system states and execution paths, even those that may be difficult to reproduce in a real-world setting.

1.5 Research Objectives

The main goal of the research is to propose an efficient centralized interprocess communication (IPC) framework with integrated caching mechanisms to enhance model-checking capabilities in detecting bugs at both local and global scales. The specific objectives of this thesis can, therefore, be broken down into the following:

1. To investigate existing caching mechanisms and their suitability for storing and managing multiple communication data between processes operating within the model checker level centralization.
2. To integrate the proposed caching mechanism into the centralization IPC model that can effectively address computation overhead.
3. To validate the hybridization approaches that can improve both local and global bug detections.

1.6 Research Scope

First, this research focuses on bytecode execution rather than native code execution. The choice to operate at the bytecode level is motivated by the ability to navigate the relevant state space more effectively compared to exploring the unrelated state space inherent in low-level native codes. The concrete model checkers for distributed systems discussed in section 1.1 address programming language coverage, testing, and debugging of the unmodified distributed systems. The goal is to identify failures, crashes, and violations of user-defined properties. These tools serve as model checkers specifically tailored for distributed systems, each designed to address the coverage of specific programming languages or platform-specific considerations.

Second, this thesis focuses on the hybridization of the current centralization and cache, expanding upon the core system of JPF. However, it does not dive into the process through which JPF constructs the model from the bytecode, generates state space, captures thread scheduling, and various other features inherent to the JPF core.

Finally, the thesis considers distributed multi-threaded applications that involve communicating processes, regardless of their physical location or means of communication. The proposed approach utilizes a combination of centralization and caching to minimize computation overhead and state space, thereby enhancing the model checker-level centralization performance. The research aims to enhance state space reduction for networked applications, with cache-based techniques being the primary solution proposed. The proposed approach is evaluated using specific metrics and compared with existing techniques. Additionally, the thesis includes experiments involving bug injection and various search strategies to detect bugs while exploring the state space of distributed applications and maintaining deadlocks. However, it is important to note that this research is limited to communication over the TCP/IP protocol with blocking I/O and unbounded buffers.

1.7 Research Contributions

The key contributions of this research are outlined as follows:

1. A scalable caching mechanism has been designed to automatically adjust the size of the request and response trees based on the number of connections created by processes. These trees are designed to store multiple communication data from multiple processes, identified by their respective endpoints, such as client and server endpoints. They can scale efficiently without causing a state space explosion. Furthermore, the inter-process

communication (IPC) models that utilize these trees are also scalable and can handle large volumes of input and output data streams without incurring excessive overhead computation. This helps to minimize the state space.

2. The proposed model checker employs a scalable caching mechanism along with enhanced read-and-write algorithms. This results in reduced overhead computation and minimizes state space exploration during the backtracking process.
3. The proposed bug injection techniques ensure the preservation of both local and global bug-detection properties, while the suggested search strategies guarantee the framework's adaptability for various state space exploration scenarios.

1.8 Thesis Organization

This thesis is organized into six chapters.

Chapter 1 offers a thorough explanation and discussion of the background, research problem, research questions, research motivation, research objectives, research scope, and research contributions, focusing particularly on model-checking distributed systems employing bytecode-level execution.

Chapter 2 offers a comprehensive overview of model checking, including its application in software model checking and the model checker for Java. It delves into system modeling for distributed systems, examines existing literature on model checking distributed systems, conducts literature reviews on bytecode model checking for distributed systems, engages in discussion, and concludes with a summary.

Chapter 3 discusses the overall research methodology, benchmark networked applications, experimentation of cache and centralization techniques, the customization of caching techniques, the proposed design, the proposed reduction techniques, tools, and experimental settings, the evaluation of the proposed method, and finally the summary.

Chapter 4 provides an in-depth explanation of the development and implementation of the software model checking for distributed applications using a hybridization approach.

Chapter 5 provides an in-depth analysis and discussion of the research's experimental results. The proposed model checker has experimented with four benchmarked distributed applications including Echo, Daytime, Chat, and Alphabet.

Chapter 6 summarizes the achievements of the research objectives, research contributions, and future work.

CHAPTER 2

BACKGROUND AND LITERATURE REVIEW

2.1 Introduction

This chapter covers the background and literature surrounding the research studies. The background materials serve as the theoretical foundation of the thesis, offering a concise overview of crucial concepts related to model checking and its applications in distributed systems. Furthermore, the chapter examines existing studies, identifying research gaps in bytecode-level model checkers for distributed systems. In conclusion, the chapter wraps up with a thorough discussion and summary of its key points.

2.2 Model Checking

The invention of model checking represents a fundamental change in the application of logic for detecting bugs in both hardware and software industries (Clarke, Henzinger, & Veith, 2018). This technique is a sub-discipline of formal methods that exhaustively checks for property violations in a concurrent system. It explores all possible system states in a brute-force and systematic manner. Despite its initial success in hardware verification, the principles of model checking have found extensive application and adoption in the realm of software as well (Beyer & Podelski, 2022).

There are two major advantages of model checking over the other formal verification techniques (Baier & Katoen, 2008). First, it is fully automatic. This means that model checking does not require any user supervision to control the input during the design simulation. Second, it provides a *counterexample* when the given model does not satisfy the given properties. The *counterexample* is like a bug trace, which provides important clues to fix the software bug.

The model-checking problem can be stated as:

$$M, s \mid = f \qquad 2.1$$

Where *M* is a *Kripke* structure (i.e., labeled transition system) and *f* is a formula of temporal logic (i.e., the specification). The problem is to find all states *s* of *M* such that *M, s |= f*. The system model is formally described as a *Kripke* structure or Labelled Transition System (LTS), and the system properties are generally expressed in temporal logic. When the state LTS satisfies the property, the model checking continues to the next state until the error is found, or the end state is reached. If the error is found, it produces the *counterexample* that gives an important clue to fix the error.

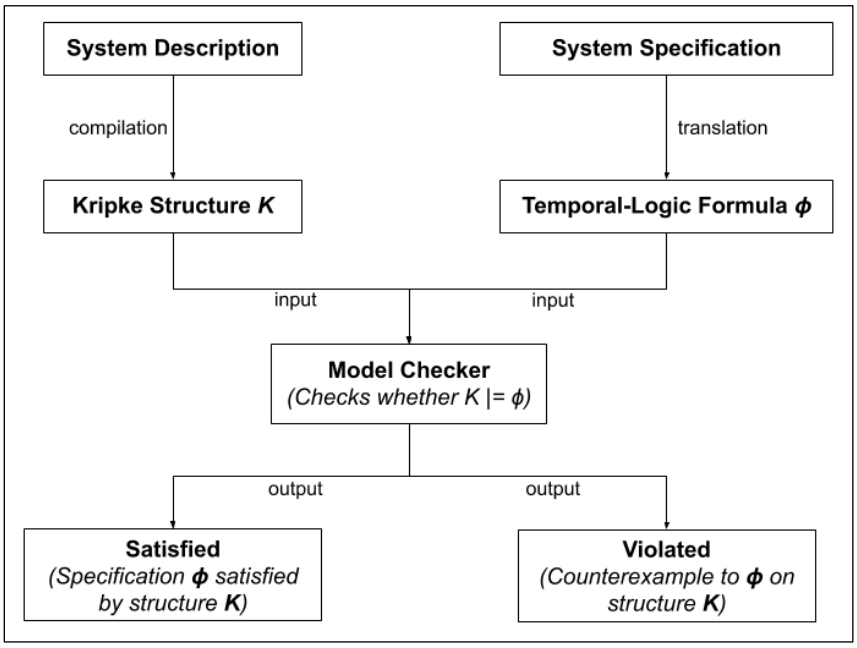

Figure 2.1 Basic model-checking methodology (Clarke, Henzinger, Veith, et al., 2018).

Figure 2.1 shows the schematic view of the model checking. The model checking accepts two inputs, the system description, and system specification. The system description is formally described as *TS*, and the property of the system is

generally expressed in temporal logic. When the state of *TS* satisfies the property, the model checking will continue to check the next state until the error is found. Otherwise, it produces the *counterexample* that gives an important clue to finding the error.

Model checking can be categorized into two fundamental approaches: explicit-state (Holzmann, 2018) and symbolic-state (Chaki & Gurfinkel, 2018) model checking. The primary distinctions between these approaches lie in how they manage and manipulate explored states during computation. Explicit-state algorithms explore the program state sequentially, storing explored states in full program states. Consequently, algorithm optimization concentrates on frequently visited states. In contrast, symbolic algorithms assign either the set of initial states or the set of valid states, examining the state space until an error is detected. This thesis considers only explicit-state model checking since it is the most effective for software systems.

2.3 Software Model Checking of Single Process

The process of verifying software involves using theorem proving (Goguen, 2021) and static analysis (Rival & Yi, 2020). These methods analyze the software system by examining its model, without actually executing it. In contrast, dynamic analysis (Afianian et al., 2019) involves running the system to identify any errors and provides information about these errors through trace data. Software model checking can be performed at different stages during the software development process and can be seen as a combination of both static and dynamic analysis. It is used to verify abstract versions of a program during the design phase or its actual implementation. Model checkers such as VeriSoft (Godefroid, 2005) and CMC (Musuvathi et al., 2002) dynamically analyze non-determinism in a program by running it in a specialized environment. However, dynamic analysis requires a runtime environment for the target

system to be executed, which may not have all the capabilities of a full operating system, such as a file system.

In a concurrent system, where multiple processes are running at the same time, the operating system selects a process from a pool of candidates to run, based on its scheduling policy. The order in which the processes are selected is known as the execution schedule, and this order may differ between each run of the system. This can result in issues such as race conditions. To address this, programmers use mutual exclusion, which ensures that only one process has control over a shared resource at a time. However, this can lead to further problems like deadlocks and starvation. Software testing can only verify one specific execution schedule, making it difficult to reproduce failures that occur in a specific order. Model checking, on the other hand, systematically examines all possible schedules to check if any specified properties are being violated.

Previously, model checkers were used to verify a system during the analysis and design phase. But now, modern model checkers work directly on the implementation of the application rather than its abstract model. Some modern model checkers, like SLAM (Ball et al., 2011) and Blast (Beyer et al., 2007; Henzinger et al., 2003) take program code as input and verify the abstracted code instead of the actual code. However, other model checkers process the program code directly. These model checkers control the execution of the target program and guide it through the execution tree using methods such as depth-first search, breadth-first search, and heuristic search, which traverse the tree based on a heuristic function.

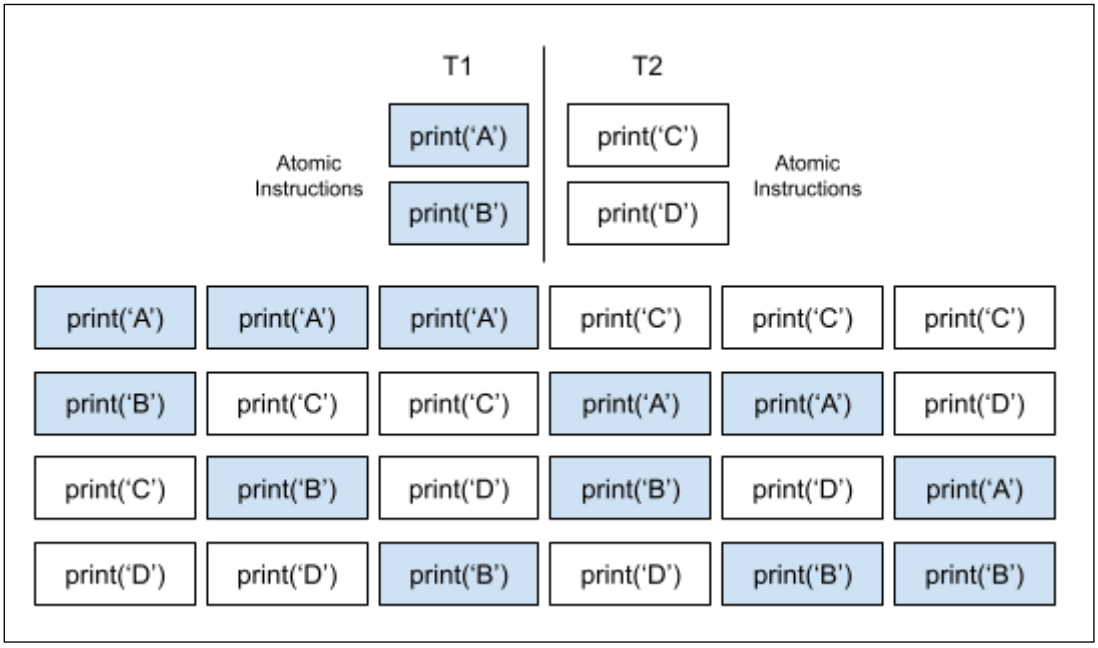

Figure 2.2 Example of model checker executing six different ways of a system which has two threads and each thread has two atomic instructions.

In this subsection, a simple example program is used to illustrate the basic workings of a software model checker. The program consists of two threads, each containing two atomic instructions. An atomic instruction is a single, uninterrupted operation. The operating system that the program is running on can execute these instructions in six different orders: (T1, T1, T2, T2), (T1, T2, T1, T2), (T1, T2, T2, T1), (T2, T1, T1, T2), (T2, T1, T2, T1), and (T2, T2, T1, T1), as shown in Figure 2.2. The software model checker examines all six possible schedules. After the execution of one atomic instruction, the execution tree branches. When configured to search using a depth-first approach, the model checker backtracks once the program terminates. The execution tree is used to display all possible states of the program, as depicted in Figure 2.3.

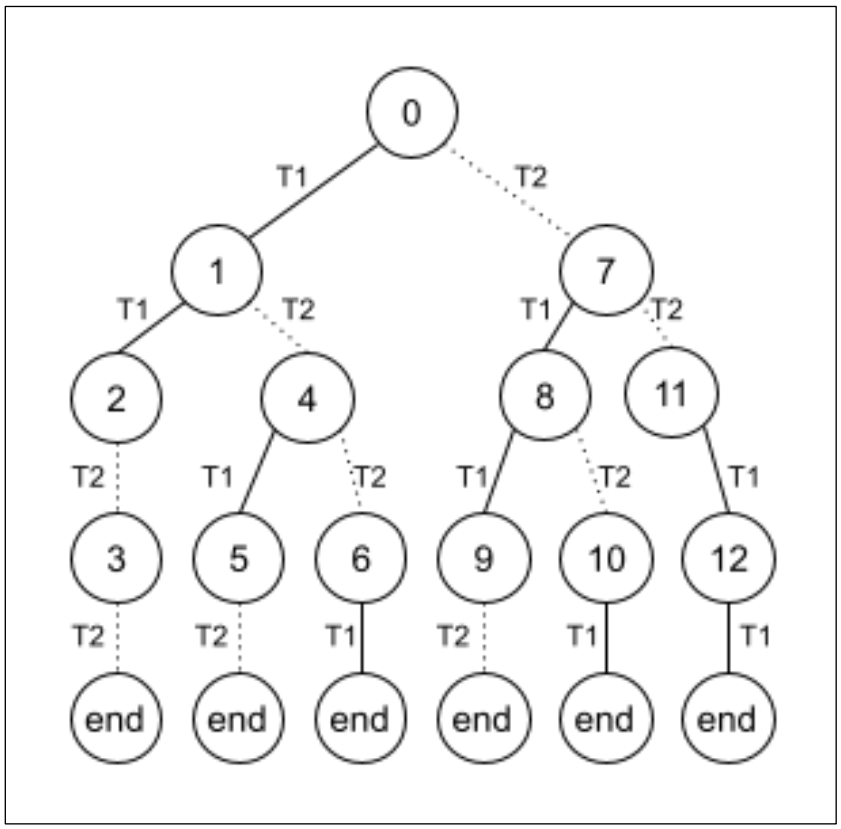

Figure 2.3 Example of execution tree by the model checker.

As mentioned earlier, modern model checkers have been applied directly to the actual implementation of software programs, written in extreme programming languages such as C or Java. These model checker tools help programmers to detect software bugs and errors during the implementation phase. An example of a model checker tool that model checks real programs is Java PathFinder (Visser & Mehlitz, 2005).

The model checker requires backtracking of the system under test (SUT). To illustrate the backtracking concept, let's look at the example of how the model checker executes the Java program as shown in Figure 2.4.

SIMPLE JAVA PROGRAM

```
1   import java.util.Random;
2   public class Rand {
3       public static void main (String[] args) {
4           Random random = new Random(42);
5           int a = random.nextInt(2);
6           System.out.println("a=" + a);
7           int b = random.nextInt(3);
8           System.out.println("b=" + b);
9           int c = a/(b+a -2);
10          System.out.println("c=" + c);
11      }
12  }
```

Figure 2.4 Simple Java program using Random class.

Figure 2.4 shows an example of a simple Java program that computes two random variables, *a* and *b*. The program starts with the initialization of the Random class with a value of 42. The integer variables *a* and *b* are initialized and given the *.nextInt()* method with the values of 2 and 3, respectively. Variable *c* does computation as shown in line 9. Finally, the program prints out the result of *c*.

Figure 2.5 (a) indicates the execution graph on normal execution, and Figure 2.5 (b) shows the execution graph of the program by the model checker. The octagon, single circle, and double circle represent the start state, and end state, respectively. Notice that in (a) the program executes on normal execution. It does not involve backtracking thus the program does not cause any errors. However, in (b), the model checker executes the program in all possible ways until it finds the error state.

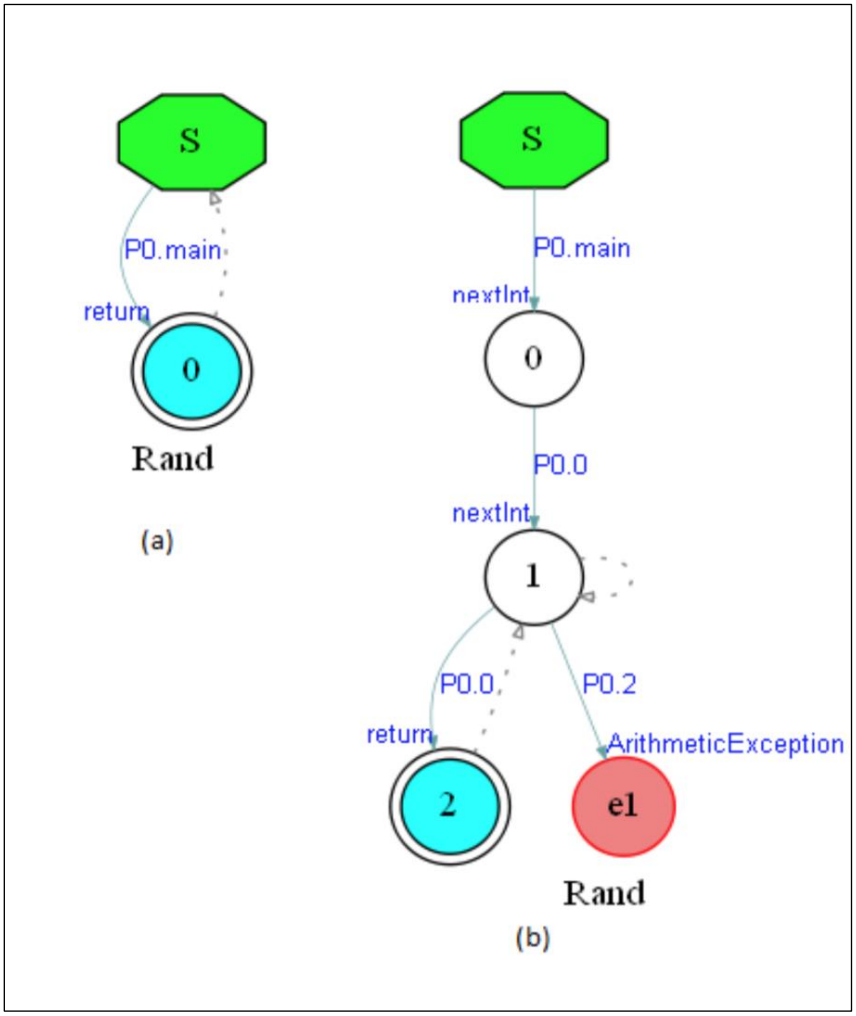

Figure 2.5 Execution graph by (a) executed normally and (b) executed by the model checker.

2.4 The State Space Explosion Problem

The cornerstone of applying model checking is the state space explosion (Clarke et al., 2011). The exhaustive state exploration expands the system description of *M* as a *TS* or *Kripke* structure. The mathematical definition of such a structure is equivalent to directed graphs with additional labels at the vertices. The computation of statements or functions in the digraphs often leads to new global states in the system under test (SUT), with different program counters and variables. For efficient model checking, the states

of the SUT need to be stored in the main memory. This memory should be fast enough to verify the system of interest. The state-space is known to be exponential to the size of the SUT, in addition, the number of parallel components, data variables and properties, and channel buffers normally lead to an event larger state-space. The exponential growth of states during the model checking is often called the *"state-space explosion problem"*.

2.5 The Model Checker for Java

State space explosion is the cornerstone of software model checking. Executing bytecodes instead of native codes or operating system low-level codes can avoid a tremendous amount of unrelated state space (*Model Checker for Java Programs - NASA Technical Reports Server (NTRS)*, n.d.). This section describes the Java thread model, the Java PathFinder, Java Networking, and applying model checking for distributed systems.

2.5.1 The Java Thread Model

In Java, a thread represents an executable task and has thread-local information that cannot be directly accessed by other threads. It shares the global heap with other threads. A Java program starts with only one thread, the main thread, which begins executing in the *main(String[] args)* method. Additional threads are created when Thread objects are initialized. A task can be assigned to a thread in two ways: by extending the Thread class or by implementing the Runnable interface. In either case, the programmer must provide the task to be executed in the *run()* method. The thread can commence its work after its *start()* method is invoked by another thread, known as the *parent thread*.

Multiple threads in a system can access shared computing resources. Figure 2.6 provides an example where two threads share the same resource, an instance of the Printer class. The Configuration and JobExecutor classes each have their tasks, but both access the same information in a Printer instance. To avoid conflicts, a mechanism must be in place to manage access to these shared resources.

AN OBJECT SHARED BY TWO THREADS

```
1   class Configuration extends Thread {…}
2   class JobExecutor extends Thread {…}
3   class Printer {
4       public static void main(String[] args) {
5           Printer p = Printer.getInstance();
6           Thread conf = new Configuration(p);
7           Thread exec = new JobExecutor(p);
8       }
9   }
```

Figure 2.6 Example of an object shared by two threads in Java.

Java has a synchronization mechanism, which is linked to a lock object, to manage access to shared resources. Every object in Java can serve as a lock. Programmers use the lock to create a block of code in which only one thread is allowed to enter at a time. The thread must acquire the lock before entering the mutually exclusive code. Figure 2.7 illustrates an example of the use of a mutual exclusion lock. To ensure that two threads do not access the printer simultaneously, the programmer uses the synchronized keyword to protect the printer from concurrent access. In this example, the Printer instance itself is used as the lock.