Acceleration of Bounded Model Checking based on Satisfiability-Modulo-Theories for Embedded Software Designs

劉, 楽源

https://doi.org/10.15017/1470601

Acceleration of Bounded Model Checking based on Satisfiability-Modulo-Theories for Embedded Software Designs

Leyuan Liu

August 2014

Department of Advanced Information Technology

Graduate School of Information Science and Electrical Engineering

Kyushu University

Abstract

Software has become a critical part of our lives nowadays. Many software affects public security and our health care, such as those used in nuclear reactors, modern avionics system controllers, and artificial cardiac pacemaker etc. Consequentially, human life has become more and more dependent on the services provided by these systems. Embedded software is a very important proportion of such applications. Software failures in embedded systems are usually life-threatening and expensive in most cases. However, the complexity of embedded software increases substantially, which makes it challenging to develop techniques for ensuring highly reliable embedded software while considering the complexity, especially in the development of concurrent embedded software.

Proposing formal verification techniques to improve the reliability of embedded software is the topic of this thesis. In particular, the emphasis is put on verifying the correctness of designs rather than source code of embedded software. Verifying designs have the advantage that it helps to reveal bugs in the early phase of a software development process, and thus, avoid the expensive costs that are generally required for revising a bug found in source code.

Among the existing various formal verification techniques, satisfiability-modulo-theory (SMT) based bounded model checking (BMC) is adopted in this thesis. SMT-based BMC has the potential to avoid the notorious state-space explosion problem often suffered by other automated verification techniques such as explicit model checking and BDD-based symbolic model checking. Regarding embedded software designs, the thesis considers those developed

using the Hierarchical State Transition Matrix (HSTM) modeling language that is popular in Japans embedded software industry.

The main contributions of the thesis are as follows: (1) the thesis first proposes a basic SMT-based BMC algorithm (encoding approach) for HSTM designs. The designs consist of concurrent processes that communicate through shared-variables or asynchronous message passing. (2) since the verification speed of the algorithm is relatively slow, especially for large-scale designs, the thesis further proposes techniques to accelerate the BMC process. The key idea is to reduce the formula size of the design by exploring and memorizing legal execution paths of the designs with explicit state-space exploration techniques. During such state space exploration, state-space abstraction techniques, such as bounded context switch (BCS), are applied to filter out those execution paths that are unnecessary for revealing concurrent bugs. (3) the thesis proposes heuristics-based techniques to classify memorized execution paths into path clusters, and additionally, proposes a multicore computation structure for verifying those path clusters separately and concurrently. (4) the proposed techniques are implemented into an existing formal verification tool called Garakabu2. The experimental results demonstrate that verification speed can be accelerated significantly.

The structure of the thesis is as follows: Chapter 2 introduces fundamental knowledge related to model checking; Chapter 3 introduces an encoding approach to translating an embedded software design developed with HSTM into quantifier-free formulas, representing a BMC problem, which could be solved by an SMT solver; Chapter 4 proposes acceleration techniques which takes the advantage of explicit model checking as a pre-procedure; With the purpose of decreasing the state space of the design, Chapter 5 introduces the algorithms that integrate BCS into the explicit state space exploration procedure, and then the heuristic predicates for classifying possible system execution paths into path clusters; Chapter 6 describes a distributed SMT solving framework. Chapter 7 describes the implementation of the distributed framework and the acceleration techniques proposed in Chapters 4 and 5. As shown by the

experimental results, the scalability and efficiency of SMT-based BMC can be improved significantly; Chapter 8 concludes the contributions of this thesis and presents the limitation and future work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Software has become an critical part of our lives nowadays. Many software affects public security and our health care. This kind of software ranges from such as nuclear reactors and modern avionics system controllers, to artificial cardiac pacemaker that helps maintain adequate heart rate for patients who suffer from arrhythmia. Consequentially, human life has also become more and more dependent on the services provided by these types of system. Embedded system is a very important proportion of such applications.

Embedded system could be defined in general: an embedded system is a computer system with a dedicated funtion within a larger mechanical or electrical system, often with real-time computing constraints [1]. Embedded software is computer software, written to control an embedded system or a part of it [2]. Embedded software has its own characteristics which are different from conventional desktop applications. For instance, embedded software has the requirement to meet the timing constraints, access the memory region, handling concurrency and control the hardware. Therefore, the high reliability is required by embedded software. Software failures in embedded system are usually lifethreatening and expensive at most cases [3]. The Therac-25 accident [4, 5] is a tragic example of embedded software failure. The Therac-

25 was a radiation therapy machine produced by Atomic Energy of Canada Limited (AECL). It was involved in at least six accidents between 1985 and 1987, in which patients were given massive overdoses of radiation. As a result, several people died and others were seriously injured. The reason of failure is that a one-byte counter frequently overflowed and causes the interlock to prevent the overdoses of radiation fail to work. These accidents highlighted the dangers of software control of safety-critical systems. Another example of of embedded software defects is the recall of Toyota Prius in 2010. It was reported that some drivers in USA and Japan claimed the brake problems had let the car crash. Toyota said that it was caused by a software glitch.

On the other hand, the complexity of embedded software increasing substantially. For instance, the code of control software of spacecrafts lunched by NASA is more complex and uses more control software than its predecessor [6]. The control software of lander *Viking* [7], which was lunched in 1976, has 5 KLOC (Lines of Code in Thousands) onboard. Meanwhile, the code of lander *Phoenix*, which was lunched in 2007, has 300 KLOC. The *MSLRover* upped the ante to 3000 KLOC [6].

From a market point of view, the software development methodologies must be applied in order to manage the develop team size, the product requirements and the project's constraints (time-to-market and costs control,etc.) [8]. It is difficult to regulate the development to manage all the requirements and constraints which are often in opposition to each other. For example, we could not develop an embedded software which has more functionalities in a short time considering the economic factor and software quality.

Due to the reason mentioned above it is hard to verify correctness of embedded software although the verification is quite necessary. In the mean time, the exiting methods, *peer reviewing*, *testing*, etc., which are used to ensure the reliability and accountability, have a lot of shortcomings. Therefore, new method is needed to meet this demand.

Model checking is an automatic technique for verifying finite state concurrent systems [9].

It has a lot of advantages over original approaches to this problem such as peer viewing and testing. The process of model checking consists of system modeling, requirement specification and verification. Boolean Satisfiability (SAT) based bounded model checking (BMC) [10] is one important model checking techniques. This technique has been successfully applied to verify sequential software in industrial embedded software verification [11]. However, the concurrency of embedded software makes the verification which uses SAT-based BMC more difficult. The major advantage of BMC is considering the system state transition under a constraint of bound. Then the reduced state space is exploited internally by the state-of-the-art SAT solver with the DPLL algorithm [12, 13]. The basic idea of BMC is reasoning counterexamples within the execution paths restricted by the bound k of a system M that violate properties specified f in Linear Temporal Logic (LTL). The model checker unrolls the system k times and translates the unrolled M with f into a propositional formula to verify whether f is satisfy or not. The formula is satisfiable if and only if f has a counterexample of depth less than or equal to k. Recent years, the Satisfiability Modulo Theories (SMT) solver, which is an extension of SAT is used instead of SAT solvers [14]. The M and f are encoded into a quantifier-free formula and and use a state-of-the-art SMT solver to perform the satisfiability checking. With this method, a more compact and expressive formula can be obtained than using SAT solving.

Although the SMT-based BMC has a lot of advantages than the previous methods, it still has its inherent vice that makes it inpractical. For instance, the encoding approach of SMT-based BMC doesn't distinguish the reachability of the states, which leads to a large size of the encoded formula. The time consumption of BMC increases exponentially even if the checking bound is not very deep.

1.2 Objectives

In this thesis, the research focuses on providing acceleration methods for SMT-based bounded model checking of concurrent system designs to make it more practical. The basic ideas are 1) introduces explicit model checking algorithms to reduce the unnecessary states; 2) further more, takes the advantage of distributed computing to accelerate checking efficiency. It should be noted that the system designs is concerned neither the source code nor the system with time constrains. In particular, the research has been done on the following aspects:

1) Proposing a formalization of a system designed by Hierarchical State Transition Matrix (HSTM) [15] giving an encoding approach to translate the formalized system to logic formulas;

2) Exploring efficient model checking algorithm to reduce the state-space of the system to be verified. Algorithms are designed to reduce the state space of target system designs and explore the remaining parts efficiently. It appears as making the formulas more compact;

3) Developing solving methods to accelerating model checking process further more. It is means that we try to conduct a new solving architecture to improve the model checking efficiency;

4) Implementing a tool using proposed algorithms and methods to support effective software verification. All algorithms are implemented on a bounded model checker named Garakabu2.

1.3 Contributions

The main contributions of this thesis are the development, implementation, and evaluation of a serial methods to accelerate SMT-based bounded model checking from multiple perspectives. In this view, this thesis makes three major novel contributions.

First, this thesis provides formal verification support to HSTM designs that employ message-passing as the means of communication (hereinafter called message-passing HSTM designs or just HSTM Designs for simplicity). For this purpose, the structures and behaviors of HSTM designs are formalized. Consequentially, a symbolic encoding method is proposed, through which an HSTM design could be Bounded Model Checked using SMT solver. Furthermore, we have implemented the formal verification of software designs in HSTMs on a tool named Garakabu2. This work reveals the low efficiency shortcoming of previous solving methods which uses the classic BMC algorithm.

Second, the approaches to accelerating SMT-based bounded model checking are proposed. The approaches center around an unrolled bounded reachability tree (BRT) of a HSTM design which is built with stateless explicit state exploration (that is, states are not saved during exploration). Specifically, reachability of invalid cells (representing undesired states) of an HSTM design, which occurs within the bound concerned, could be discovered during construction of the BRT, and furthermore, if no such occurrence, the constructed BRT could be utilized to rule out unnecessary subformulas of a BMC instance and thus make the instance easier to solve. By such combination, the benefits of both explicit exploration and BMC with respect to speed as well as memory could be enjoyed. In addition, The observation shows that much BMC verification time is consumed by iterative search (i.e., gradually increase the search depth till the concerned bound), which is necessary for finding the shortest counterexamples. A binary search algorithm is presented to avoid iteration but still guarantee to find the shortest counterexamples, if any. These approaches are implemented in Garakabu2. The preliminary experiments show that verification could be accelerated substantially.

Third, bounded context switch (BCS) [16, 17], an under-approximation technique, is integrated into stateless explicit-state exploration (SESE). It has been found that only a few context switches (i.e., execution-order changes) are suffice to reveal concurrency bugs [16, 18]. Such integration thus allows SESE to explore limited number of context switches of multiple parallel processes in the system so as to reduce the state space. Further, rather than encoding all legal execution paths, which are memorized during SESE, into a single (usually large) formula and inquiring its satisfiability of SMT solvers, we introduce heuristic predicates and use them to classify the paths into path clusters. Each path cluster can be considered as an independent BMC instance, which is usually smaller and easier to solve. Furthermore, multiple such BMC instances can be solved concurrently with multiple SMT solvers running on multicores. Since no information sharing is needed among these independent BMC instances, once a counterexample is found, the computation on all other cores can be safely terminated. In addition, rather than directly applying SESE and BMC to a user-specified bound, we gradually deepen the checking depth from 0 with a fixed incremental number. Such iteration finishes until a counterexample is found or the bound is reached. In this way, counterexamples that are shorter than the user-specified bound can be revealed while avoiding expensive computation between the depths where the counterexample is found and the specified bound.

Fourth, a distributed SMT solving architecture is presented. The second and third contributions affect on the first stage of SMT-based BMC by reducing the reachable states of target system. The distributed SMT solving architecture affects on the second stage of SMT-based BMC. The whole BMC speed is enhanced further more by using this distributed solving architecture. The basic idea of this contribution is utilizing the computational capacity of multiple PCs. If the state space of the target system could be decomposed into smaller sub-state space which are encoded into formulas respectively, we may use SMT solvers to solve these formulas distributively. The experimental results show that the solving efficiency can be increased substantially at the most cases.

1.4 Outline

This section briefly presents the outline of this thesis and overviews of each chapter.

Chapter 2 is devoted to an introduction of fundamentals which are used in model checking. The related work of accelerating SMT-based BMC are classified and introduced also.

Chapter 3 introduces the formalization of HSTM and the methods to encode HSTM to quantifier-free formulas that could be solved by an SMT solver.

Chapter 4 proposed the first acceleration technique which takes the advantage of explicit model checking.

Chapter 5 introduces our algorithms which integrate bounded context switch (BCS) into SESE. Then a method to classify the possible execution paths of the system into path cluster is proposed. In addition, an incremental method is presented to ensure that the shortest counterexamples could be revealed.

Chapter 6 describes the implementation of distributed SMT solving, which contains the original method and its improvements.

Chapter 7 presents a series of experiments to evaluated the acceleration techniques proposed in the above chapters.

Chapter 8 concludes the contributions of this thesis. Further more, this chapter presents the limitation of our work and points some directions for future work.

Chapter 2

Background and Related Work

In this chapter, some preliminary knowledge which is necessary to understand the following chapter is given. The related work of acceleration techniques for symbolic BMC is discussed.

2.1 Logic Fundamentals

Logic is the use and study of valid reasoning [19]. The study of logic features most prominently in the subjects of philosophy, mathematics, and computer science. In [20], logic is defined as a system of rules to manipulate symbols. Due to the discussion of logic is not the key topic of this thesis, the logic fundamentals, which are the basis to understand the proposed theories in the following chapters, are briefly introduced in this section. Further information can be found in textbook [19, 20, 11].

2.1.1 Propositional Logic

The aim of logic in computer science is to develop languages to model the situations we encounter as computer science professionals [21]. Propositional logic is widely used in database queries, artificial intelligence (planning problems), automated reasoning and circuit design [20]. Following the definition given above, the syntax of a propositional logic formula is de-

Table. 2.1: The Basic Rules in Propositional Logic

Syntax	Meaning
$\neg$	The Negation of a variable p is denoted by $\neg p$.
$\vee$	Given p_1 and p_2 which we may wish to state at least one of them is true. $p_1 \vee p_2$ is called the disjunction of p_1 and p_2
$\wedge$	$p_1 \wedge p_2$ is true if and only if the two variables are true. It is called the conjunction of p_1 and p_2
$\rightarrow$	$p_1 \rightarrow p_2$ expresses an implication relation between p_1 and p_2. It suggests that p_2 is a result of p_1.

fined as the following [21].

$$formula : formula \ \wedge formula \mid \neg formula \mid (formula) \mid atom$$

$$atom : Boolean - identifier \mid TRUE \mid FALSE$$

The $formula$ above consists of basic symbols and rules. Informally, the formula is referred to "*sentences*". The propositional logic is a two-valued logic. Every "*sentence*" is assumed to be either true or false. The minimum elements of propositional logic are the constants *TRUE* and *FALSE* and some propositional variables: $p_1, p_2, \ldots, p_n$, which can be used to construct more complex *sentences* in a compositional way. The basic operators, namely the rules with which we can construct complex sentences are shown in Table 2.1. They are "negative" ($\neg$), "and" ($\wedge$), "or" ($\vee$) and "implies" ($\rightarrow$). Actually, the other operators can be obtained by using $\neg$ and $\wedge$. We assume that p_1 and p_2 are propositional variables. These rules are shown as follows:

$$p_1 \vee p_2 \equiv \neg(\neg p_1 \wedge \neg p_1)$$

$$p_1 \rightarrow p_2 \equiv \neg p_1 \vee p_2$$

The logic operators shown in Table 2.1 have different relative precedence which are $\neg$, $\wedge$,

Table. 2.2: The Truth Table of Propositional Logic

p_1	p_2	$\neg p_1$	$p_1 \vee p_2$	$p_1 \wedge p_2$	$p_1 \rightarrow p_2$
F	F	T	F	F	T
F	T	T	T	F	T
T	F	F	T	F	F
T	T	F	T	T	T

$\vee$ and $\rightarrow$ from the highest to lowest. With the purpose of determine the formula is true or false, there is a mechanism for evaluating the propositional variables, namely interpretations. That means every propositional variable exactly is assigned with one truth value. For a given formula, the truth value can be computed by a truth table or induction. The truth table of the propositional variables and its operation are shown in Table 2.2. In this table, the truth value of p_1 and p_2 are determined so that we can use the truth table to obtain the truth value of their conjunction, disjunction and imply. Induction is another way to determine the truth value of a propositional formula. The details of induction will not be presented here. An introduction of induction can be found in Chapter 1 of textbook [21].

2.1.2 Satisfiability Modulo Theories

SMT is a research topic that concerns with the satisfiability of formulas with respect to some background theories [11]. The development of SMT can be traced back to early work in the late 1970s and early 1980s. In the past two decades, SMT solvers have been well researched in both academic and industry, and achieved significant improvements on performance and capability, and thus it becomes possible to use SMT solver in BMC problem solving.

SMT is an extension of propositional satisfiability (SAT), which is the most well-known constraint-satisfaction problem [22]. SMT generalizes boolean SAT by adding equality reasoning, arithmetic, fixed-size bit-vectors, arrays, quantifiers, and other useful first-order theories. An SMT solver is a tool for deciding the satisfiability (or dually the validity) of formulas in

these theories [23]. In analogy with SAT, SMT procedures (whether they are decision procedures or not) are usually referred to as SMT solvers [11].

Actually, the SMT problem emerges in many fields, such as intelligence, formal verification for software and hardware, static program analysis, scheduling and planning, etc.. We focus on its application in formal verification especially BMC.

2.1.3 Linear Temporal Logic

Linear temporal logic [9], namely LTL for short, is a temporal logic, with connectives that allow us to refer to the future. Time is modeled as a sequence of states, extending infinitely into the future [21]. In software verification, we more concern about the sequences of states transitions of a system, and time is not mentioned in an explicit way, such as "eventually" and "never" are used for specifying the formula. Generally speaking, LTL extends propositional logic by considering temporal operators. The syntax of LTL is defined over a set of atomic propositions, logical operators and temporal operators as follows:

$$formula : TRUE \mid FALSE \mid p \mid \neg f \mid f \ \wedge \ g \mid f \ \vee \ g \mid f \ \rightarrow \ g \mid$$
$$Xf \mid Ff \mid Gf \mid f \ U \ g \mid f \ R \ g \mid$$

p, f, g are all LTL atomic formulas.

In LTL, besides the logic operators $\neg, \wedge, \vee, \rightarrow$ which are the same as in propositional logic, there are five temporal operators which are represented as "next state" **X**, "some state in the future (eventually)" **F**, "all future state (globally)" **G**, "until" **U** and "release" **R** [9]. We assume that f and g are LTL path formulas over a set of states, the intuitive meanings of the five temporal operators are shown in Table 2.3. In Figure 2.1, the six linear transition shows the temporal operators' meanings intuitively. Assuming a and b are all atomic propositions, the first linear transition is an atomic property a which currently holds. The second transition shows that the atomic a holds next time. $F\ a$ meanings that a will hold in some time point in

Table. 2.3: The Function of Temporal Operators

Operator	Meanings
X f	f has to hold at the ne**X**t time point.
F f	f has to hold at some time point in the **F**uture.
G f	f has to hold **G**lobally.
f **U** g	f has to hold continuously. **U**ntil g holds.
f **R** g	g has to hold when f holds, f **R**eleases g.

the future. $G\ a$ represents that a holds always. a U b denotes a must hold until b holds. The last transition means that a remains false till b becomes holds, and a is released.

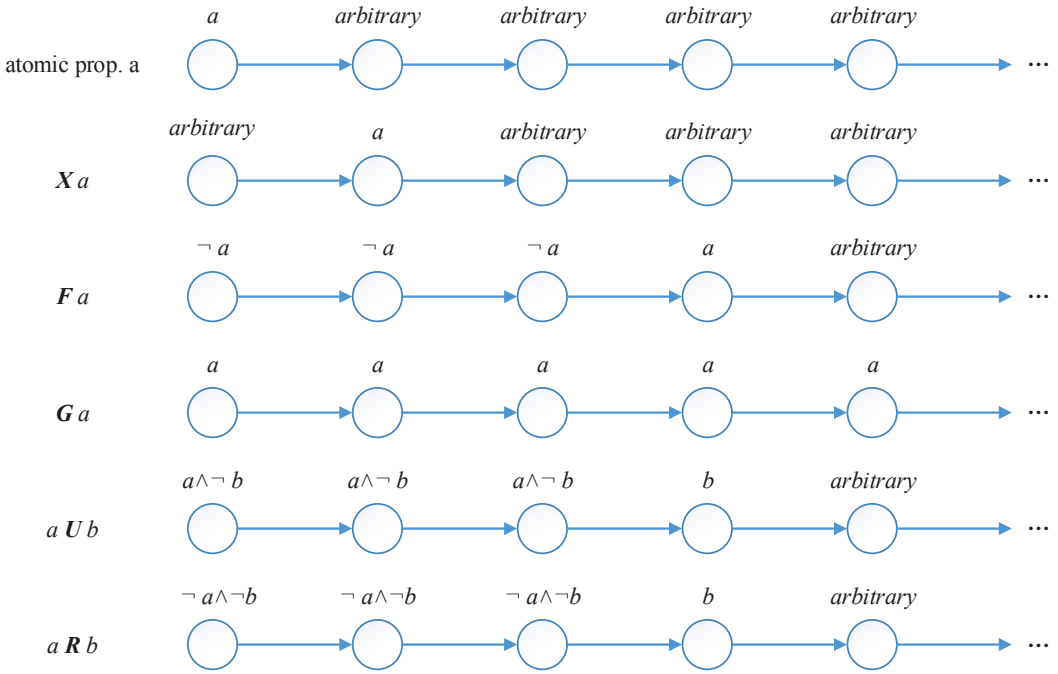


Figure. 2.1: Intuitive Semantics of LTL.

2.2 SMT-Based Bounded Model Checking

2.2.1 Model Checking

Model checking [24] is wildly acknowledged as an effective formal technique for verifying whether a finite state system satisfies desired properties expressed in temporal logic. The state space of the system is traversed automatically either in an explicit or symbolic manner in order to verify the properties. If the property fails to hold, a counterexample is generated in the form of a sequence of state transitions. This traverse usually is an exhaustive search of the state space. Generally, the properties can be classified into safety and liveness properties, where generally the former means that something bad will never happen and the latter means that something good will eventually happen. Model checking is often used for fasifying a system (namely to find designing errors) rather than verifying (namely to prove the correctness of) the system.

There are primarily two types of model checking approaches: explicit model checking and symbolic model checking. Initially, the model checking algorithms proposed are explicit ones, which enumerate all reachable states explicitly and checks whether a given property holds on each of the states. However, explicit model checking suffers from the notorious state exploration problem, namely that the state space of a system grows exponentially such that it could not be automatically analyzed with limited computing resource and time. Symbolic model checking was introduced firstly in [25]. This algorithm, which is different from explicit model checking, represents the states of the system symbolically using Boolean functions. In order to improve the efficiency, Boolean formulas are represented using ordered binary decision diagrams (OBDD or BDD for short). Although BDD-based symbolic model checking could increase significantly the state space that could be analyzed, one of the disadvantages is that the order of state variables, which usually has to be adjusted by human verifiers manually, can heavily influence the size of BDD and thus the performance of this method.

2.2.2 Bounded Model Checking and SMT-based BMC

To mitigate this disadvantage of BDD-based symbolic model checking technique, bounded model checking (BMC) was proposed in [10] [26]. The basic idea of BMC is to search for counterexamples (i.e., design bugs) in transitions (state space) whose length is restricted by an integer bound k. If no counterexample is found, then k increases and the analysis procedure repeats until either a counterexample is found or the pre-defined upper bound is reached. It is commonly acknowledged as an complementary technique to BDD-based symbolic model checking [26]. In BMC, a model checking problem is boiled down to a propositional satisfiability problem that can be solved with SAT solvers [11]. This model checking technique is named as SAT-based BMC. SAT-based BMC can benefit from the development of SAT technique in the aspects of both formula size and solving speed.

In SAT procedure the variables involved must be of the type Boolean, which, however, makes SAT to be inexpressive for industrial problems, e.g. computer program. That is, variables of types other than Boolean must be encoded into Boolean/bit variables, which can result in a large formula size.

Recent years, with the development of modern efficient SMT solvers like Z3 [23] and CVC4 [27] etc., there is a trend to use SMT solvers instead of SAT solvers in BMC for better expresiveness.

A bit more formally, BMC can be defined as follows: Given a finite state transition system M and a temporal property P, for the state space of all possible transitions of M whose length is bounded by an integer k, verify the property P to find a counterexample. A BMC problem in M with P is formulated in the seminal paper [10] like:

$$BMC(M, P, k) = I_0 \wedge \bigwedge_{i=0}^{k-1} T_i \wedge (\neg P) \tag{2.1}$$

where I_0 represents the initial states of system M, T_i denotes the transition relation of M with

step i, and $I_0 \wedge \bigwedge_{i=0}^{k-1} T_i$ represents all possible paths from an initial state of the system M. The negative form of P is a formula used to represent the situation that the property is violated. If there exits an assignment to all the variables used in formula (2.1) which makes (2.1) evaluate to true, then a counterexample is founded. Otherwise, the property is satisfied by the behaviors of M bounded by k. Although BMC can also be used for proving correctness (see [10] for more details), BMC is more often used for finding counterexamples rather than correctness proof. The negative form of property is used because the SAT solvers and SMT solvers tend to find the assignments that make the formula being evaluated true.

Actually in addition to SAT and SMT, diverse methods are exploited by academics for conducting BMC. In a nutshell, BMC can be divided into four types, which are BDD-based BMC, explicit-state BMC, SAT-based BMC and SMT-based BMC. It is difficult to judge which is the best methods, for every method has its advantage and shortcoming comparing with the others. In [28], the authors compared different methods for BMC by utilizing comprehensive industrial benchmarks. The results indicate that BDD-based BMC is better at targeting deep counterexamples. For the benchmarks with shallow counterexamples, SAT-based and explicit BMC are more efficient. In this thesis, the SMT-based BMC is considered.

Over the last several years, BMC has attracted scientists in developing bounded model checker or improving existing model checkers so as to make the checkers support BMC. Here we list and describe some of them.

SPIN [29, 30, 31] is a popular open-source software verification tool, which can be used for the formal verification of multi-threaded software applications. The tool is developed at Bell Lab, starting in 1980. SPIN uses PROMELA, a high level language, as its specification language and it supports embedded C code as part of model specification. SPIN works on-the-fly, supports multicore computers, and can cope with BMC problem. It is an explicit model checker that allows checking properties expressed in Linear Temporal Logic (LTL) [9].

NuSMV [32, 33] is a symbolic model checker based on BDDs and SAT. NuSMV has been

designed to be an open architecture for model checking, which can be reliably used for the verification of industrial designs. NuSMV allows for the representation of synchronous and asynchronous finite state systems and the properties expressed in Computation Tree Logic (CTL) [9] and LTL. The NuSMV supports SAT-based BMC. Minisat [34, 35] SAT Solver and zChaff [36] SAT Solver are used as back-end solvers. The model checking problem is generated to be a propositional propositional problem, then a SAT solver is called to solve it.

SAL [37] stands for Symbolic Analysis Laboratory. It is a framework for combining different tools for abstraction, program analysis, theorem proving, and model checking toward the calculation of properties (symbolic analysis) of transition systems. The current generation of SAL tools comprises a collection of state of the art LTL model checkers and auxiliary tools based on them [37]. There are two tools that can perform bounded model checking procedure, SAT-based bounded model checking (sal-bmc tool) for finite state systems and SMT-based bounded model checking (sal-inf-bmc tool) for infinite state systems.

2.3 Classification of Acceleration Techniques for SMT-based BMC

Although SMT-based BMC can obtain more compact and expressive first-order-logic formulas compared to SAT-based BMC, it still suffers from the state space explosion problem because the BMC instance becomes bigger in size and more complex to solve when unrolling the target system's executing steps [38]. In order to make SMT-based BMC more efficient and scalable, many research studies have been carried out on accelerating this technique. Recall again that, in a SMT-based BMC procedure, the system M and the property P are encoded into formulas written in first-order-logic, then the formulas are given to the backend SMT solver. So we will discuss the acceleration techniques in two aspects in this section.

2.3.1 Methods on Accelerating SMT-Based BMC

The first aspect is the acceleration methods used "befor" SMT solving, namely to reduce the state space or make the encoded formulas simpler before SMT solving.

1) **Parallelization**

The industrial size verification problems are hard to solve by model checker, though research on model checking have obtained significant performance enhancement. Meanwhile, hardware has been developing rapidly. When facing such a BMC problem, parallelization is a natural and intuitive way. Multicore CPU and multi-CPUs become a dominant trend. A PC has a multicore CPU and shared memory. It is possible to perform model checking problem on desktop. Distributed model checking algorithms have been studied for many years, but most of them are restricted to safety properties, but for liveness properties there is no good parallel solution.

In [39], the authors present a tool D-TSR for checking safety properties of low-level embedded software. D-TSR is a parallel SMT-based bounded model checking tool. *Tunneling* and *slicing* reduce method is used in this tool. *Tunneling* is a partition technique to decompose the system's control state graph into independent subgraphs. *Slicing* is used to simplify the results obtained from *tunneling*. The distribution framework consists of a central controller and several workers. The tunnels are created by each worker at every BMC unrolled depth statically and deterministically to reduce the communication overhead. The authors consider the uneven load balance problem in their framework by using relaxed synchronization criteria. The master can dynamically adjust an available client pool during synchronization, in which slower clients are removed from the pool. Consequently the idle time for clients is reduced. The master and clients are grouped as a star topology [39], while only clients can communicate with the master. The master creates the partitions for a BMC instance and assigns the partitions to available clients. Next, the master sends initial order to all available clients and waits for the results from different clients. Because the tunnels which the clients work on

are disjunctive subproblems of a BMC instance, if one of the clients reports satisfiable, then a counterexample is returned. The master sends an *abort* command to other clients who is working on some tunnels. For a client, if it receives the initial command, it will do the tunneling and control state reachability with the bound. When it receives the *solve* command, it unrolls the sub BMC problem, simplifies the formulated subproblem, and performs BMC on the tunnel. If the result is satisfiable, a counterexample will be reported to the master. After receiving the *Abort* command, the client will abort and report to the master. Comparing with the work in [40, 41, 42], the authors' framework uses tunneling technique to decompose the BMC instance into independent subproblems so that the communication overhead is reduced significantly. The D-TSR performs BMC on different clients with the reduced and simplified tunnel (partition), and usually the decomposed BMC instances are easier to solve on parallel clients separately than the original one. The message communicating with the clients and master is just the id of tunnels or some command, not as same as the work in [41] which transfers the whole partition between master and clients. Although the authors' work scales almost linearly with the number of CPUs, the tool only applies to safety properties.

2) **Abstraction**

Abstraction is one of the most important techniques for reducing the state space of systems in the model checking field. There are two important but basic abstraction techniques, the *cone of influence reduction* (COI) and *data abstraction*. Both of them are used in high level description of systems [9].

The *cone of influence reduction* attempts to eliminate the state transitions of the system by considering that the variables, which are in the system specification but not in the property to be verified, can be eliminated if they don't influence the variables in the properties. In such a way, the property expressed with temple logic is preserved but the system's state space becomes smaller.

Data abstraction can map the actual data values in the specification to a smaller data set.

Then through extending the mapping to the whole system specification, an abstracted system model can be obtained, which is smaller than original. Usually, the abstracted model is easier to be checked.

In [43], the authors propose a high-level SMT-based BMC framework in order to reduce the gap between theoretical and practical solution for high-levels of abstraction. Their framework consists of five steps. First, they use an extended finite state machine (EFSM) to model the system. Second, the system model M and property P are performed with a series of property preserving transformations. Third, they do a control state reachability on the results of transformations. Fourth, using this reachability, a further reduced state space is obtained. Finally, the reduced model is unrolling to do a BMC with P. The COI is used in step two where the EFSM is abstracted to obtain the abstracted model M'. Considering the property, non-contributed state and the outgoing transitions are removed from M. All the non-contributed transitions are replaced by an transition to a state marked as $SINK$. In this framework, the COI is not the main technique for accelerating BMC directly. The result of COI is used for obtaining a smaller model to perform the control reachability. If there is a large size BMC instance, it is expensive and difficult to determine a state or transition is contributed or not.

In [38], the authors present a *slicing* technique to simplify the execution path of system. The basic idea of slicing is that for a formulated BMC problem, the irrelevant path can be sliced. In this paper, firstly, the BMC problem is decomposed to smaller subproblem on independent partitions (called tunnel) T_1, T_2, ... T_i. Then the irrelevant paths not in T_i can be removed from BMC subproblem $BMC^k_{T_i}$. The relevance used to eliminate the irrelevant paths are obtained by a high level control state reachability analysis (a Breadth-First Search (BFS) traversal of the Control Flow Graph of the system). Because the BMC instance is decomposed into small tunnel, it is easy to determine the correlation among different tunnels.

In [44], Jeremy Morse et al. propose an SMT-based BMC algorithm with context-bound for ANSI-C software. Their approach converts the liveness properties expressed with LTL

into *Büchi* automata and finally into C monitor threads. The method used in their work to mitigate state space explosion is called state hashing. This technique can reduce the number of redundant interleavings in context-bounded model checking. During the exploration to determine the reachability tree of multi-threads software, many interleavings pass the identical states in the reachability tree. For instance, v_1 and v_5 are two nodes in the reachability tree of the multi-threads program. The transition from them leads to the same state further. When exploring the state v_5, the transitions originating from it can be eliminated simply. Then a set of hashes represent the states of nodes that have been explored on reachability tree. State hashing technique is used in explicit model checking and cannot be implemented on symbolic model checking directly. Thus they propose a two-level hashing scheme: a node-level hash represents a particular reachability tree node and a variable level hash represents the constraints that affect a particular assignment to a variable [44].

In [45], the authors use Boogie programming language to represent C program, then generate a verification condition solved by SMT solver from the BoogiePL program that considers context-bounded switches. In order to make their verification more scalable, they propose a *field slicing* technique. The basic idea of the technique is based on that the verification of a given property typically depends only on a small number of field in the data structures of the program [45]. Their algorithm partitions the set of fields into *tracked* and *untracked* fields. Only the tracked fields are accessed while the untracked fields are abstracted away. It means that the program context-switches from the untracked fields are dropped completely. The complexity of of the verification is reduced.

3) **Partition-Based Methods**

The partition based methods denote a technique that is used in symbolic state space traversal [38]. This technique can be used in BMC to decompose a big BMC problem into smaller subproblems. Then the memory utilization can be reduced, or even doing the BMC with the subproblems in parallel. It should be notice that the partition-based methods is used to decom-

pose a big state space into several smaller sub-space, make it possible to model check a system in parallel. Actually, the decomposed several sub-space can be solved in a serial manner.

Tunneling is such a technique that decomposes a BMC instance disjunctively into smaller and independent subproblems [38], e.g., sets of control paths. For instance, Suppose that BMC^k presents the bounded model checking problem with bound k, after the tunneling method is performed, the control path of the system is decomposed to T_1, T_2,..., T_i, then the BMC subproblem is obtained as $BMC^k_{T_1}$, $BMC^k_{T_2}$,..., $BMC^k_{T_i}$. If there exits at least one satisfiable$BMC^k_{T_i}$, then BMC^k is satisfiable. The authors also propose a partitioning heuristic, combined with subproblem ordering scheme targeted at exploiting incremental solving [38]. Their method not only decomposes the paths of the system, but also obtains a good balance between the number of partitions and their size.

4) **Hybrid**

Hybrid is such a technique that combines the symbolic model checking algorithms with the explicit model checking algorithms in order to take the advantage of explicit algorithms. At the early years, BMC is based purely on symbolic techniques, such as BDDs, SAT and SMT. Subsequently, explicit model checking algorithms support BMC also.

In [46], the authors propose an acceleration method that combines BMC with explicit model checking technique. They use Hierarchical State Transition Matrix (HSTM) [15], which is a table based modeling language for developing designs of software systems, to specify the target system. Then the HSTM is translated into logic formulas. Finally, the formulas are solved by the backend SMT solver CVC3 [47]. The basic idea of their method is removing unnecessary transitions from BMC procedure at $step[k]$. They use explicit state exploration technique (BFS is adopted) to execute HSTM design and construct a Bounded Reachability Tree (BRT). It is different from the exploration with normal (bounded) BFS model checking algorithms [9, 48]. In their method, states that have been explored are not saved in memory (except those temporally saved in state queue). Saving explored states is necessary for normal

BFS to avoid exploring same states, otherwise the normal BFS may simply fall into a loop. After the BFS exploration, the reduced system is encoded to logic formulas for conducting BMC. The use of explicit BFS in their method prevents the exploration from state explosion problem that normal BFS search suffers.

In [18], the authors describe and evaluate three approaches to model check multi-threaded software using bounded model checking based on SMT. They combine explicit state space exploration with symbolic model checking. An explicit exploration algorithm is used for exploring all the possible interleaving while the interleaving is treated symbolically to obtain a reachability tree of the multi-threads program. The multi-threads ANSI-C program is translated into goto-language, which is the internal language of the CMBC model checker. They symbolically execute each instruction of the goto-program written in goto-language, and expand the reachability trees by four rules they predefined. At this stage, the properties are not checked. The reachability tree is used to reduce the size of states to be explored by the DFS algorithm at the next stage.

2.3.2 Improvement of SMT Solver and Model Checker

SMT-based Bounded model checking (BMC) has benefited also from the advances of research on SMT solvers. As efficiency and capacity of SMT solver improve, BMC becomes more efficient in solving industrial-sized instances. We introduce two state-of-the-art SMT solvers about their advances to accelerating BMC.

1) **Z3**

Z3 [49] is a theorem prover being developed by Microsoft. The first external release of Z3 was in September 2007 [23]. It supports linear real, integer arithmetic, fixed-size bit-vectors, arrays, uninterpreted functions and quantifiers. It is integrated in some model checker for bounded model checking, e.g., CBMC [50]. In [51], the authors propose a portfolio approach to deciding the satisfiability of SMT formulas, and they introduce how they implement the

parallel version of Z3. They parallelize the sequential solver by running multiple solvers and each of them is configured to use different heuristics. Lemmas are shared between different solvers periodically. The input is copied to every core so that there is no need for locking the formulas during the solving, and the shared lemmas are the same. They claim that the parallel Z3 outperforms the previous sequential Z3, on many benchmarks, parallel version beats sequential version by orders of magnitude.

2) **CVC4**

CVC4 [27] is an efficient open-source automatic theorem prover for SMT problems. It can be used to prove the validity (or, dually, the satisfiability) of first-order formulas in a large number of built-in logical theories and their combination. [27]. Currently, CVC4 has supports for equality over free function and predicate symbols, real and integer linear arithmetic, bit vectors, arrays, tuples, records and user defined inductive data types.

CVC4 supports CVC4's native language, SMT-LIB 2.0 [52], SMT-LIB 1.0 [53], and supplies API for C++, JAVA. The most recent Nightly Build version of CVC4 supporting for parallel solving to increase the solving efficiency [54].

2.4 Related Work

This thesis primarily focus on related works on improving BMC performance through hybrid (combining explicit-state and symbolic) or parallel algorithms. The recent work in [18, 50] presents a hybrid BMC approach for the verification of multi-threaded software (called lazy approach, which performs best among the three proposed approaches). The approach first traverses a reachability tree (RT) for a given software in a depth-first manner while limiting the number of context switches, and then encodes each RT path into a formula and verifies it with BMC. The procedure stops when a counterexample is found or all paths are explored. However, in the case that no counterexample exists, this procedure has to explore all RT paths,

the number of which can grows exponentially with the number of parallel processes as well as the checking bound. In our approach, we explore reachability under the context of BCS (while checking deadlock and safety properties), and classify RT paths into clusters, which are then separately encoded and checked in parallel for liveness properties. In this way, path explosion could be avoided and multicore computation could further improve efficiency.

The work in [55] (an improvement of a similar work on JPF in [56]) presents a hybrid approach by firstly using explicit-state techniques to traverse the control flow graph, in contrast to the reachability tree/graph as above and in our approach, of a program and encodes the variable values of representative control flow paths (to avoid non-legal paths) into BMC instances to be solved. Only invariant (safety) properties are considered in this approach. Partial order reduction (POR) [9] can be trivially, as stated, integrated into this approach, while this is highly non-trivial for our approach since states explored are not saved, and we thus instead utilize BCS to reduce the state space.

The work in [57] also proposes a hybrid approach, in which explicit-state exploration is used to compute a set of frontier states at a certain depth (the states before reaching the frontier set are not stored), and the state space starting from that frontier set to some deeper depth is encoded and solved with BMC. Such interleaving between explicit exploration and symbolic encoding happens once memory limit is reached. The states in frontier set can be decomposed and thus enables a divide-and-conquer approach with multicore computation similar to ours. However, losing state information before frontier set makes this approach only possible for checking safety properties (our technique is stateless but path information is remained).

The work in [38] proposes to classify control flow paths, constructed explicitly but statically, into clusters based on tunnels, and each cluster of paths is then encoded and solved with BMC possibly in parallel with multicore computation. Our work is based on dynamic computation of reachability tree, rather than control flow. Also, program assertions are necessary for computing tunnels, which again makes the approach only applicable for verifying safety

properties.

In [58] the authors study to develop a verification strategy that make it possible to finish large verification problems with a high quality results. They focus on explore applying the parallelism and search diversification on the *SPIN* model checker [29, 30, 31]. They suppose that there already exits several different model checking problems. By using their strategy, the number of states explored in unit time (a default time duration) could be increased significantly. The verification time could be reduced from days to hours. How to decomposed one hard verification problem into several easy problems is not mentioned in this paper. In Chapter 5 of this thesis, an algorithm is proposed that can decompose the state space of one model checking problem into smaller ones.

Last, regarding parallel BMC solving, it is necessary to mention that modern SMT solvers like Z3 [23] and CVC4 [59] are designed to support multicore computation through lemma sharing in their own infrastructure, by which, a large BMC instance is divided and computed in multicores. However, in our divide-and-conquer strategy, a verification problem is firstly divided into smaller (and simpler) subproblems and encoded into independent BMC instances, which are then solved in different solvers. Our strategy can avoid the workload for communication between solvers, and furthermore, counterexamples that reside only in certain subproblems may possibly be revealed earlier and faster.

Chapter 3

Encoding Approaches for HSTM Design

In this chapter, the first minor contribution of this thesis is described. For BMC, the first thing is using one kind of modeling language to model the target system, then translating the system model into logic formulas. The first contribution is the encoding approaches by which the system model is translated to formulas.

3.1 Introduction

State Transition Matrix (STM) [15] is a table-based modeling language for developing software system designs. Each STM abstracts a function module of the design in the form of a table, in which the behavior of the module is specified according to the dispatch of certain events on certain states. A Hierarchical STM (HSTM) consists of several STMs that are structured hierarchically. An HSTM Design is a system developed with HSTM. The STMs in an HSTM design execute asynchronously and communicate with each other through shared variables or message passing. It lacks formal methods for automatic checking the correctness to improve reliability, though HSTM design is widely used and adopted by software industry especially in embedded software development in Japan (e.g., the commercial model-based CASE tools ZIPC [60]).

It is difficult to apply model checking [9] to concurrent software systems like HSTM designs, due to the tremendous number of possible interleavings of events and combinations of variable values [61], which usually cause the *state explosion problem*. Symbolic algorithms like Binary Decision Diagrams (BDD) [9] and Satisfiability Modulo Theories (SMT) [62, 11] based model checking, relieve this problem by representing and enumerating system state symbolically (comparing with explicit algorithms, e.g., SPIN [48]). Compared to BDD based techniques [10], the SMT based approach needs no human intervention and supports more *background theories*, .

The formalization of HSTM designs (that utilize message passing) presented in this chapter is based on our previous work in [63] and [64]. In [63], we proposed an encoding approach to formal verification of HSTM designs (that utilize shared variables) based on SMT solving. That is, a formalization of HSTM designs as state transition systems is presented firstly. Consequentially, based on this formalization, a symbolic encoding approach is proposed, through which correctness of an HSTM design with respect to LTL properties could be represented as Bounded Model Checking (BMC) problems that could be determined by SMT solving. In [64], we proposed an encoding approach to STM designs whose communication uses the method of message-passing (non-hierarchical). The static and dynamic behaviors of STMs are encoded into formulas that can be solved by SMT solvers.

The subject of this chapter is providing formal verification support to HSTM designs that employ message-passing and shared variables as the means of communication (hereinafter called message-passing HSTM designs or just HSTM Designs for simplicity). For this purpose, we first formalize structures and behaviors of HSTM designs. Consequentially, we propose a symbolic encoding method, through which an HSTM design could be Bounded Model Checked using SMT solver. The work in [63] by considering communication among STMs by means of message-passing and sharing variables is extended here. Furthermore, the formal verification of software designs in HSTMs are implemented on a tool named Garakabu2. The

designs are encoded for conducting BMC problem with respect to LTL properties checked by Garakabu2, which equipped with user-friendly features towards improving its usabilities for software engineers who are generally not specialists in formal techniques.

The formalization of HSTM designs has been influenced by the work on formalization and verification of hierarchical UML (HUML) state machines [65, 66]. HSTM and HUML are different in several aspects. For instance, cells in an HSTM is executed atomically and thus special care for this manner is necessary in our formalization and encoding, especially when call to and return from child STMs; the action language used to formalize HSTM involves program-specific elements such as the *return* statement, etc. Regarding verification, the primary focus of our work, the work in [65] proposed to translate UML models into SPIN, and the work in [66] proposed a symbolic encoding approach of UML models into NuSMV [67], through which BDD based [9] and (boolean) SAT based [11] model checking could be conducted. Our work is based on SMT, which is an extension of SAT with underlying theories such as linear arithmetic and arrays etc., and is more expressive and results in more compact formulas to be solved [14].

A simplified Money-Exchange Machine (MEM) shown in Figure 3.1 is used to demonstrate our method. MEM is modeled as two HSTMs and each consists of two STMs. The hierarchical structure is shown in the bottom-right of Figure 3.1, where MainInterface and ReturnController are root STMs, and Exchanger and Returner are their child STMs respectively. The function and working process of MEM are described as follow: With MainInterface, bills of small denominations could be deposited into MEM, and customers could request to exchange large bills into small ones; after the request is dispatched, the Exchanger will be called by MainInterface and exchange bill, and small bills are transferred to Returner; Returner takes charge of returning those bills to the customer under the control of ReturnController. Irrelevant details of MEM is omitted. For instance, `xUser10KRequest` is used to denote a user request to exchange 10K large-denomination bill, but whether the 10K bill is inserted or not, and where it

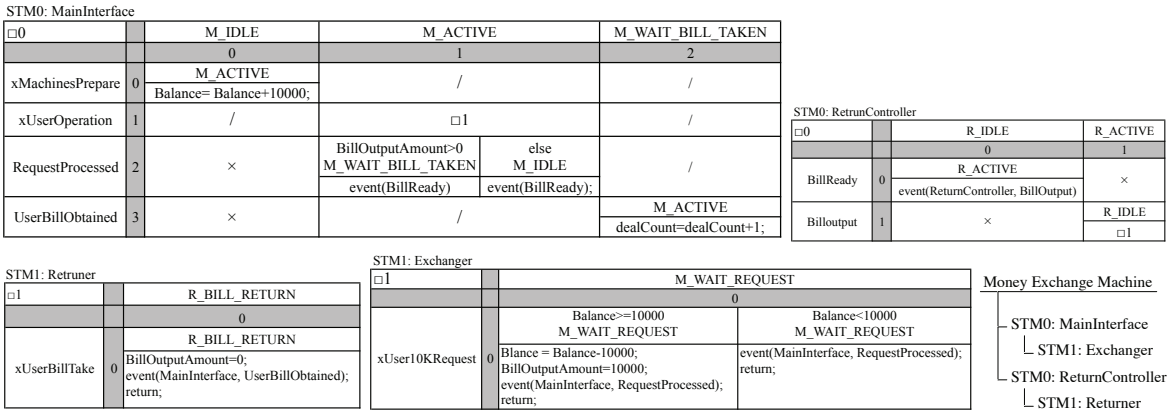


Figure. 3.1: Running example: A simplified Money-Exchange Machine modeled as two HSTMs with roots MainInterface and ReturnController.

goes if inserted, are not modeled. In addition, the number 10000 inside the STM denote bills of small-denominations (e.g., 1K denomination). Last, we intentionally introduced design errors into the HSTM model of MEM for demonstration purpose, for instance MEM's behaviors are *unreasonably* defined when there is no enough balance in Exchanger, which is to be discussed in detail in Section 3.2 and Section 3.3.

The chapter is organized as follows. Section 3.2 presents a formalization of message-passing HSTM designs and their dynamic behaviors , then proposes an encoding approach from HSTM designs (and LTL properties) into logical formulas that represent BMC problems. Section 3.3 proposed the formalization and encoding approach of HSTM designs communicating by sharing variables. Section 3.4 summarize this chapter and mentions future work.

3.2 Formalization and Encoding for HSTM Design Communicating by Message-passing

3.2.1 Formalization

In the first place we introduce an action language L to define HSTM designs. L is chosen to be a simple subset of C language with C's conventional syntax and semantics. Type sys-

tem of L consists of boolean, integers, and reals. Supported *expressions* of L are (1) Boolean literals `true` and `false`, integer and real literals, (2) Variable identifiers, and (3) Infix expressions $leftexpr\ op\ rightexpr$, where op can be one of +, -, *, /, &&, ||, >, <, >=, <=, ==, or !=, with the semantics of C. Supported *statements* of L are (1) Assignments of the form $lhs = rhs$, (2) Child STM calling statements of the form $\square child\_STM\_id$, (3) Parent STM returning statements written as *return*, and (4) Message sending statements of the form $event(HSTM.q\_name, event\_name)$, with the meaning of sending a message (usually an event) $event\_name$ to the message queue of $HSTM.q\_name$. We use L_{bool} and L_{stmt} to denote the set of boolean expressions, and respectively the set of statements, of L.

Definition 1 (STM). *Assuming the action language L, an STM is a tuple STM = (S, E, C) where S is a finite set of status, E is a finite set of events, and C is a finite set of cells.*

Each $s \in S$ has a (unique) index denoted by $index(s) \in Nat$. The $index(s)$ maps a status s to a unique integer number. At any time only one status denoted $active(STM)$ is active and initially s with $index(s) = 0$. E is the events sent to or retrieved from a message queue of the HSTM, which the STM is belonging to. There are two types of events, one is E_{int}, which is dispatched by a execution of a cell, the other is E_{ext} that dispatched by the environment. Each $e \in E_{int}$ is represented by a boolean expression and has a (unique) index denoted $index(e) \in Nat$. In the implementation on the Garakabu2, we use unique integers to represent the status and events in all STMs for simplicity. C consists of three types of cell, which are normal cells C_{nor}, ignore cells C_{ign}, and invalid cells C_{inv}. Each $c_N \in C_{nor}$ is a tuple $\langle s, e, u, a, s' \rangle \in S \times E \times L_{bool} \times L_{stmt} \times S$. We define $source(c_N) = s$, $event(c_N) = e$, $guards(c_N) = u$, $actions(c_N) = a$, and $target(c_N) = s'$. Each ignore cell is a tuple $\langle s, e, / \rangle$, and each invalid cell is a tuple $\langle s, e, \times \rangle$. Functions *source* and *event* are also defined for $c_I \in C_{ign} \cup C_{inv}$ as for c_N, but *guards*, *actions* and *target* are not. Intuitively, a cell c of an STM, pinpointed by indexes

(*index*(*source*(*c*)), *index*(*event*(*c*))), specifies behaviors of the STM when *event*(*c*) is retrieved from the message queue while *source*(*c*) is active. If $c \in C_{nor}$, *actions*(*c*) is executed atomically and then *target*(*c*) is set to be active at the following step. If $c \in C_{ign}$, denoted in an STM by symbol "/", nothing changes. If $c \in C_{inv}$, denoted in an STM by symbol "×", an error occurs. Informally, an ignore cell means the dispatch of an event in a status is ignored, and an invalid cell means the dispatch of an event in a status should never be possible.

We give more explanation to the above notations by using MEM in Figure 3.1. Cell (0,0) of STM1: Exchanger with guard `Balance>=10000` of STM Exchanger, denoted *c* for simplicity, is a normal cell, where *source*(*c*) = `M_WAIT_REQUEST`, *event*(*c*) = `xUser10KRequest`, *target*(*c*) = `M_WAIT_REQUEST`, and *actions*(*c*) consists of three assignment, one message-sending statement and one returning statement. Intuitive meaning of *c* is that: (After a customer starts operating MEM, namely calling to Exchanger) if an event (from the customer) for requesting an exchange of 10K bills occurs and a message is dispatched and sent into message queue of an HSTM which Exchanger is belonging to. When Exchanger is waiting for a request, then a message is retrieved and dequeue from the message queue and has enough small denominations, Exchanger reduces the balance of MEM with 10000, sets `BillOutputAmount` as 10000 to express outputting 10000 small bills to Returner, enqueue the message `RequestProcessed` to the message queue (actually the unique index of the event is send to message-queue), express that the exchange request has been processed, and last returns to cell (1,1), the calling cell, of MainInterface and Exchanger switches to status `M_WATI_REQUEST` to wait for another request. Cell (0,1) of MainInterface is an ignore cell, meaning intuitively that nothing changes if a customer operates MEM (denoted by event `xUserOperation`) when it is not on work (denoted by status `M_IDLE`). Cell (0,3) of MainInterface is an invalid cell, meaning that the event – a customer has obtained small bills (denoted by `UserBillObtained`) – should never be dispatched when MainInterface is not on work.

Definition 2 (STM Configuration). *Let* $STM = (S, E, C)$ *by assuming L. Let* $VS(STM) =$

$Var(STM) \cup \{actStatus\}$, where Var(STM) denotes all L variables used in STM, and actStatus is an additional integer variable denoting (the index of) STM's status that is currently active. An STM configuration C_{stm} is the valuation of variables VS(STM).

An STM configuration essentially captures a *state* of an STM. However, we do not describe directly the behaviors (namely state evolution) of an STM, but postpone it to the later descriptions for an HSTM design's behaviors.

Definition 3 (HSTM). *Let $\mathbb{M} = \{0, 1, \ldots, m\}$. An HSTM is a tuple HSTM = (STM_0, STM_1, ..., STM_m, q, R), where STM_0 is called root STM, q is a First-In-First-Out (FIFO) message queue, shared by STM_0, STM_1, ..., STM_m and $R : \mathbb{M}, \mathbb{M} \rightarrow$ boolean defines a parent-child relation among STMs of the HSTM such that R(i,j)=*`true` *iff STM_i is the direct parent of STM_j.*

For the MEM example, MainInterface and ReturnController are root STMs of their corresponding HSTMs. And $R(0, 1)$ is `true` for both of the two HSTMs respectively.

Definition 4 (HSTM Configuration). *Let HSTM = (STM_0, STM_1,..., STM_m, q , R). Let $VS(STM_i) = VS(STM_i) \cup \{c\text{-}flag_i\}$, where $0 \leq i \leq m$ and $c\text{-}flag_i$ is an additional boolean variable introduced to each STM_i. An HSTM configuration is defined as $C_{hstm} = \{C_{stm_0}, \ldots, C_{stm_m}$, q$\}$.*

The variable $c\text{-}flag_i$ is named and used as a *control* variable. Purpose of introducing such a variable to each STM_i is to control which STMs could be executed at one time step. That is, an STM is executable iff when its *c-flag* is `true`. The value of *c-flag* is changed when call to and return from a child STM, and initially *c-flag* is `true` for all the root STMs and `false` for all the other STMs. q is a finite FIFO message queue with the operation $size(q)$, *top(q)*, *push(q,m)*, *pop(q)* and predicates *empty(q)*, *full(q)*. Each message m in q is integer type. *event($HSTM_j$.q,*

(1) If $actions(c)$ is a list of statements in the form of Case1:
$C'_{hstm_j} = \{C_{stm_0},\ldots,C_{stm_i}[v(rhs)/v(lhs),\ index(target(c))/v(actStatus_i^j)],\ldots,C_{stm_m}, q\}$
For C'_{hstm_w}, where $0 \leq w \leq m \wedge w \neq j$, $C'_{hstm_w} = C_{hstm_w}$

(2) If $actions(c)$ is a list of statements in the form of Case2:
$C'_{hstm_j} = \{C_{stm_0}, \ldots, C_{stm_i}[v(rhs)/v(lhs), \texttt{false}/v(c-flag_i^j)],\ldots, C_{stm_z}[\texttt{true}/\text{v(c-flag}_z^j)],$
$\ldots,C_{stm_m}, q\}$
For C'_{hstm_w}, where $0 \leq w \leq m \wedge w \neq j$,
$C'_{hstm_w} = \{C^w_{stm_0}[\texttt{false}/v(c - flag_0^w)], C^w_{stm_1}, \ldots, C^w_{stm_m}\}$

(3) If $actions(c)$ is a list of statements in the form of Case3:
$C'_{hstm_j} = \{C_{stm_0},\ldots, C_{stm_i}[\texttt{false}/v(c - flag_i^j),\ index(target(c))/v(actStatus_i^j)],$
$\ldots, C_{stm_z}[\texttt{true}/v(c - flag_z^j),\ v(rhs')/v(\text{lhs}'),\ index(target(c^z))/v(actStatus_z^j)],$
$\ldots, C_{stm_m}, q\}$
For C'_{hstm_w}, where $0 \leq w \leq m \wedge w \neq j$, $C'_{hstm_w} = \{C^w_{stm_0}[\texttt{true}/v(c - flag_0^w)],$
$C^w_{stm_1}, \ldots, C^w_{stm_m}\}$

(4) If $actions(c)$ is a list of statements in the form of Case4:
$C'_{hstm_j} = \{C_{stm_0},\ldots, C_{stm_m}, q[(tail + 1)/tail, index(e)/qc(tail)]\}$ or
$C'_{hstm_j} = \{C_{stm_0},\ldots, C_{stm_m}, q[(head + 1)/head]\}$
For C'_{hstm_w}, where $0 \leq w \leq m \wedge w \neq j$, $C'_{hstm_w} = C_{hstm_w}$

Figure. 3.2: Definition of C'_{hstm_j} with respect to transition rule $r1$.

c) means sending message c to the message q of $HSTM_j$ and *retrieve($HSTM_j$.q)* implies retrieving a message from message queue of $HSTM_j$. More details will be introduced later when explaining encoding approach for a message queue.

Definition 5 (An HSTM Design). *Let $\mathbb{N} = \{0, 1,\ldots, n\}$. An HSTM design D is defined as a tuple $D = (HSTM_0,\ldots, HSTM_n)$. Dynamic behaviors of D is captured by a transition system of the form $(\mathbb{C}_D, I_D, \rightarrow)$, where $\mathbb{C}_D$ denotes all reachable configuration of D, and each $C_D \in \mathbb{C}_D$ is of the form $\{C_{hstm_0},\ldots, C_{hstm_n}\}$; $I_D = \{I_{hstm_0},\ldots, I_{hstm_n}\}$ is the initial configuration; and $\rightarrow$ is the transition relation characterizing how D evolves from on configuration to another, i.e., $C_D \rightarrow C'_D$.*

Transition relation $\rightarrow$ is defined with two rules, $r1$ is internal rules corresponding to the execution of a normal cell of an STM and $r2$ is external rules for external events dispatched by the environment. Each rule consists of (enable/effective) *conditions* that captures the condition under which the rule could be executed, and $effects$ that captures its execution effects. We introduce and use symbol $T[v'_p/v_p, \ldots, v'_q/v_q]$ to denote updating v_p with v'_p, ..., and v_q with v'_q in T, while all the other elements in T remain unchanged.

The rule $r1$, which corresponds to the execution of a normal cell c in $STM_i \in HSTM_j$, $0 \leq i \leq m$ and $0 \leq j \leq n$, is written as: $C_D \xrightarrow{r} C'_D$, where

$$
\begin{aligned}
r1.condition &\triangleq v(c - flag_i^j) \ \wedge \ v(event(c)) \\
&= v(retrieve(HSTM_j.q)) \ \wedge \ v(guards(c)) \\
&\wedge \ v(actStatus_i^j) = index(source(c)) \\
r1.effects &\triangleq \ C'_D = C_D[C'_{hstm_0}/C_{hstm_0}, \ldots, C'_{hstm_n}/C_{hstm_n}]
\end{aligned}
$$

We define function $v(x)$ to map a variable or an expression to its value. $C'_{hstm_0}, \ldots, C'_{hstm_n}$ are defined by case analysis on composition of statements that $actions(c)$ may contain. We use $retrieve(HSTM_j.q)$ to denote retrieving a message from message queue q. In this chapter, we demonstrate our approach with four most common operations while it could be easily extended for other cases. (1) Case1: $actions(c)$ is of the form: a list of $lhs = rhs$ assignment statements; (2) Case2: $actions(c)$ is of the form: a list of $lhs = rhs$, a calling $\Box z$ to a child STM z, and a list of $lhs' = rhs'$ statements, where the pre-/post-positioned (to $\Box z$) lists of assignment statements can be empty; and (3) Case3: $actions(c)$ is of the form: a list of $lhs = rhs$, and a returning statement *return*, where STM_i (that c belongs to) is called by a cell c^z of its parent STM_z and $actions(c^z)$ is of the same form as Case2, and similarly, all the assignment statements there can be empty; (4) Case4: $action(c)$ is of the form: a list of $lhs = rhs$, an operation on message

queue is taken place (message sending or retrieving). Detailed definition of $C'_{hstm_0}, \ldots, C'_{hstm_n}$ are shown in Figure 3.2.

Informally, sub-rule (1) in Figure 3.2 states that the configuration of STM_i is changed by executing the assignments and setting status $target(c)$ as active. Configurations of all the other STMs and HSTMs remain unchanged; sub-rule (2) states that the list of $lhs = rhs$ assignments $before$ the calling are executed, control flag $c\text{-}flag_i$ (of the calling STM) is set to `false` and correspondingly flag $c\text{-}flag_z$ (of the called STM) is set to `true`, and control flags of the *root* STMs of all the other HSTMs are set to `false`. This change of *c-flag* variables is necessary since each cell is supposed to execute atomically and no other cells should interrupt before the calling finishes (returns); sub-rule (3) is similar to sub-rule (2) with respect to setting control flags reversely. A point that should be noticed is that the remaining list of $lhs' = rhs'$ assignments $after$ the calling statement in c^z (of the parent STM) are executed here, and additionally, the target status of c^z is set as active. These are also due to the atomic execution style of STM cells; sub-rule (4) states that action of a normal cell contains an operation of enqueue or dequeue. When enqueue, the variable *tail* is updated by *tail* + 1 and an event e is assigned to $qc(tail)$ in q. *head* is updated by assignment *head* + 1 when dequeue. Others remain unchanged.

The second rule $r2$ corresponding to the dispatch of the external event e_{Ext} to a $STM_i \in HSTM_j$, is defined as:
$C_D \xrightarrow{r2} C'_D$, where

$$
\begin{aligned}
&r2.condition \triangleq v(x_{Ext}) = \texttt{false}\\
&r2.effect \triangleq C'_D = C_D[C'_{hstm_j}/C_{hstm_j}], where\\
&\quad C'_{hstm_j} = \{C_{stm_0}, \ldots, C_{stm_i}[\texttt{true}/v(x_{Ext})],\\
&\quad \ldots, C_{stm_m}, e_{Ext}/q[(tail + 1)/tail, index(e_{Ext})/qc(tail)]\}
\end{aligned}
$$

The rule $r2$ states that if the variable x_{Ext} equivalent to `false` means there were no external

events dispatched, so that a new external event can be dispatched. After the external event x_{Ext} was sent to message-queue of $HSTM_j$, value of x_{Ext} is updated with `true` denotes an external event can be dispatched at any time.

3.2.2 A Symbolic Encoding Approach

The encoding is demonstrated by considering an HSTM design $D = (HSTM_0, \ldots, HSTM_n)$, where for $0 \leq j \leq n$, $HSTM_j = (STM_0^j, \ldots, STM_m^j, q)$, and the given bound is bd (namely, we encode all execution sequence of D whose length is bd). For demonstration simplicity, we use $VS(HSTM_j) = \{x \mid x \in VS(STM_i^j),\ 0 \leq i \leq m\}$ to denote all variables contained in $HSTM_j$. Similarly, we use $VS(D)$ to denote all variables contained in D. For each (time) step k, $0 \leq k \leq bd$, we use $x[k]$, $expr[k]$, and $stmt[k]$ to denote, respectively, a new variable, a new expression, and a new statement (all of language L) used in step k. The ways of generating $expr[k]$ and $stmt[k]$ from respectively $expr$ and $stmt$ are introduced later.

Encoding Message Queues

As far as we know, the state-of-the-art SMT solvers, don't support the queues theory, therefore we have to encode the message queue and its operations to formulas that could be solved by SMT solver. In this Chapter, the widely supported theories of *uninterpretedfunctions* (UIF) and *linearrealarithmetics* (LRA).

We adopted and used the *linear approach* proposed in [68]. Basic idea of this approach is to represent each queue $HSTM_j.q$ as an uninterpreted function `qc` : Index→Element, which is common to (shared by) all execution steps. Int for both Index and Element, the function could be re-declared as `qc` : Int → Int. Contents and attributes of q at each step k are defined using a set of variables $q[k]$ = {$head[k]$:Index, $tail[k]$:Index, $empty[k]$:Boolean, $full[k]$:Boolean, $1st[k]$:Element}, where variables whose names are suffixed with $[k]$ are fresh variables generated and used in step k. Changes of the contents and attributes are described as follows:

$$head[k+1] \triangleq \texttt{if } pop(q)[k] \texttt{ then } head[k]+1 \texttt{ else } head[k];$$
$$tail[k+1] \triangleq \texttt{if } push(q,m)[k] \texttt{ then } tail[k]+1 \texttt{ else } tail[k];$$
$$empty[k] \triangleq (tail[k] = head[k]);$$
$$full[k] \triangleq (tail[k] = head[k]+Z);$$
$$1st[k] \triangleq \texttt{qc}(head[k]);$$
$$push(q,m)[k] \Rightarrow (\texttt{qc}(tail[k]) = m).$$

In the formulas above, variable $head[k]$ holds the the first element of q at step k; variable $tail[k]$ holds the last position that is unused of the q; function $pop(q)[k]$ and $push(q,m)[k]$ is an enqueue and dequeue operation respectively at step k; $empty(q)$ and $full(q)$ are Boolean variables. All the messages in q are integers. We use $eventHSTM_i[k]$ to denote the current event dispatched to $HSTM_i$ as follows:

$$eventHSTM_i[k] \triangleq \ top(HSTM_i.q) \ \wedge \ HSTM_i.q = pop(HSTM_i.q)$$

Considering an event dispatched by the environment where the HSTM design resides in, we need to define the encoding rule for the external events at step k. $ext(e)[k]$ is used to denote an external events dispatched at step k.

$$enable(ext(e)[k]) \triangleq \neg full_j[k-1]$$
$$effects(ext(e)[k]) \triangleq push(HSTM_i.q, index(e))[k]$$
$$\wedge \bigwedge_{j=1}^{n,j\neq i}(q[k] = q[k-1]) \wedge \bigwedge_{j=1}^{n}(event_j[k] = event_j[k-1]) \wedge \bigwedge_{x\in VS(D)}(x[k] = x[k-1])$$

Enable condition of enqueuing a massage to $HSTM_i$, says that the message queue $HSTM_i.q$ at the previous step *k-1* is not full. Effects say that the variables of a queue of target HSTM are updated, the variables of other queues hold the values at previous step.

Encoding for HSTM Design Communicating by Message-passing

Encoded formula for step 0, denoted *step[0]*, that represents the initial configuration of HSTM Design D is defined as:

$$step[0] \triangleq \bigwedge_{x \in VS(D)} x[0] = v(x) \wedge \bigwedge_{y \in v(q)} y[0] = v(y) \wedge \bigwedge_{j=0}^{n} (empty(HSTM_j.q) = \texttt{true})$$

The formula simply expresses that all variables involved in D, at the initial step have their default values. For $0 \leq i \leq m$ and $0 \leq j \leq n$: initial values for $actStatus_i^j$ are 0; initial values for $c\text{-}flag_0^j$ are `true` and other $c - flag$ variables are `false`; the message queue (denoted by $HSTM_j.q$) are empty for all HSTMs, the symbol dot means that q is the element of $HSTM_j$; other unmentioned variables' initial values are given by user. The initial values for *c-flag* variables of root STMs are `true` imply that only root STMs are possible to execute at initial step. To describe encoding method for other steps, in the first place, we define encoding rules for a normal cell of *STM*. The encoding rules for external events are defined later. To define encoded formulas for steps other than 0, we first encode rule $r1$ that corresponds to the execution a normal cell, and then encode rule $r2$ that corresponds to the dispatch of an external event.

For rule $r1$, we consider a normal cell c in $STM_i \in HSTM_j$. The formula for enabling condition of $r1$ is defined as follows:

$$\begin{aligned} r1.condition[k] \triangleq\ & c - flag_i^j[k-1] \wedge\ event(c)[k-1] \\ & = retrieve(HSTM_j.q) \wedge guards(c)[k-1] \\ & \wedge actStatus_i^j[k-1] = index(source(c)) \end{aligned}$$

Expressions $event(c)[k-1]$ and $guards(c)[k-1]$ are generated by simply giving step number

$k-1$ to all variables involved in them. Variable $actStatus_i^j[k-1]$ is generated by renaming. $eventHSTM_j.q$ denotes that the message is retrieved from the message queue of $HSTM_j$.

If a normal cell c contains a message-sending operation to multiple queues, $\bigwedge_{j=1}^{t} event(HSTM_j.q, m)$, where $1 \leq t \leq n$, the condition for it is defined as:

$$\begin{aligned} r1.condition[k] \triangleq\ & c - flag_i^j[k-1] \wedge\ event(c)[k-1] \\ & = retrieve(HSTM_j.q) \wedge guards(c)[k-1] \\ & \wedge actStatus_i^j[k-1] = index(source(c)) \\ & \wedge (\bigwedge_{j=0}^{t} \neg full(HSTM_j.q)[k-1] \end{aligned}$$

The encoded formula for effects of $r1$ is defined as:

$$r1.effects[k] \triangleq actions(c)[k] \wedge \left(\bigwedge_{x \in VS(D)/X} x[k] = x[k-1] \right)$$

where X denotes the set of variables that have been changed (i.e., informally, appear in left-hand side of equations) in $actions(c)$. It implies that the other variables keep the values at step $k-1$. We then directly follow the case analysis shown in Figure 3.2 to define $actions(c)[k]$.

If $actions(c)$ is of the form **Case1**. We assume that the list of $lhs = rhs$ assignments is $st_1, \ldots, st_t$. Then each $st_l[k]$, $1 \leq l \leq t$, is defined as $lhs_l[k] = rhs_l[k-1|k]$, where $rhs_l[k-1|k]$ means (1) if variables $lhs_1, \ldots, lhs_{l-1}$ occurs in rhs_l, the occurrence of these variables in rhs_l is given step number k, and (2) for all other variables, step number $k-1$ is given. Note that, this encoding approach could not handle multiple assignments to a same variable in $actions(c)$. Please refer to [69] for the solution that we have proposed for this. The formula for $actions(c)$ is defined as follows where change of active status is added.

$$actions(c)[k] \triangleq \left(\bigwedge_{l=1}^{t} st_l[k] \right) \wedge actStatus_i^j[k] = index(target(c))$$

If *actions(c)* is of the form **Case2**. We assume that the calling statement is $\square z$ and that the list of $lhs = rhs$ assignments before it is $st_1, \ldots, st_t$. The formula for $actions(c)$ is defined as follows, where each assignment st_l is encoded in the same way as in **Case1**.

$$actions(c)[k] \triangleq \neg c - flag_i^j[k] \wedge c - flag_z^j[k] \wedge \left(\bigwedge_{l=1}^{t} st_l[k] \right) \wedge \left(\bigwedge_{0 \leq w \leq n}^{w \neq j} \neg c - flag_0^w[k] \right)$$

If $actions(c)$ is of the form **Case3**. We assume that the remaining list of $lhs' = rhs'$ assignments after the calling statement in the cell c^z is $st'_1, \ldots, st'_t$. The formula for $actions(c)$ is defined as follows similar to **Case2**.

$$\begin{aligned} actions(c)[k] \triangleq \neg c - flag_i^j[k] \wedge c - flag_z^j[k] \wedge \left(\bigwedge_{l=1}^{t} st'_l[k] \right) \wedge \left(\bigwedge_{0 \leq w \leq n}^{w \neq j} c - flag_0^w[k] \right) \\ \wedge actStatus_i^j[k] = index(target(c)) \wedge actStatus_z^j[k] = index(target(c^z)) \end{aligned}$$

If $actions(c)$ is of the form **Case4**. We can encode *enqueue* and *dequeue* operation as follows:

Firstly, if the st_l is an enqueue (message-sending) operation $enq(q, e)[k]$ that sends a message e to queue $HSTM_j.q$ can be encoded as:

$$\begin{aligned} &enable(enq(HSTM_j.q, e)) \triangleq \neg full_j[k-1] \\ &\quad \wedge eventHSTM_j[k-1] = -1 \\ &effects(enq(HSTM_j.q, e)[k]) \triangleq head_j[k] = head_j[k-1] \\ &\quad \wedge tail(HSTM_j.q)[k] = tail(HSTM_j.q)[k-1] + 1 \wedge push(HSTM_j.q, e) \end{aligned}$$

Enable condition for enqueuing a message e to $HSTM_j.q$ says that q is not full and at step $k-1$, no new event was dispatched.

Secondly, if the st_l is a dequeue (message retrieving) operation which dequeues a message

$HSTM_i.q$ at step k, we define the encoding rules for it as:

$$enable(deq(HSTM_i.q))[k] \triangleq (eventHSTM_j[k-1] = -1 \ \wedge \neg empty_i[k-1])$$

$$effects(deq(HSTM_i.q))[k] \triangleq eventHSTM_i[k] = 1st_i[k-1] \wedge pop(HSTM_i.q)[k]$$
$$\wedge \bigwedge_{j=1}^{n, j\neq i} (eventHSTM_j[k] = eventHSTM_j[k-1] \wedge q[k] = q[k-1]) \wedge \bigwedge_{x\in VS(D)} (x[k] = x[k-1])$$

The enable condition of dequeue a message says that the queue is not empty and at the previous step, $k-1$, no events are dispatched. The effects of dequeue operation are (1) the first message (head) is dispatched to $HSTM_j.q$; (2) corresponding value changes to message queue are done; (3) all the other variables of message queues are not change.

Encoded formula for $r1$ that corresponds to the execution of a normal cell is defined as follows by combining $r1$'s encoded condition and effects formulas.

$$r1[k] \triangleq \ r1.condition[k] \wedge r1.effects[k]$$

Encoded formula for $r2$ which dispatch a message $ext(e)[k]$ at step k by environment is defined as:

$$r2.condition \triangleq \neg full_j[k-1]$$

$$r2.effects \triangleq push(HSTM_i.q, index(e))[k] \wedge \bigwedge_{j=1}^{n, j\neq i} (q[k] = q[k-1])$$
$$\wedge \bigwedge_{j=1}^{n} event_j[k] = event_j[k-1] \wedge \bigwedge_{x\in VS(D)} (x[k] = x[k-1])$$

We define the whole encoded formula executed at one step k. We use $R1$ and $R2$ to denote the set of internal and external events in D respectively and $|R1| + |R2| = u$. We introduce a set of boolean variables $\{fl_1, \ldots, fl_u\}$, each corresponding either to a $r1 \in R1$ (denoted $fl(r1)$) or

to a $r2 \in R2$ (denoted $fl(r2)$). The formula at one $step[k]$ is defined as:

$$step[k] \triangleq \bigvee_{r1 \in R1}(r1[k] \wedge fl(r1)[k]) \vee \left(\bigvee_{r2 \in R2}(r2[k] \wedge fl(r2)[k]) \right)$$

Step formula $step[k]$ essentially represents all possible configurations in HSTM design D from the initial configuration (denoted by $step[0]$). We define a rule *interl* with bound *bd* as a restrict to the execution of HSTM design D, due to the variables in counter-example does not filter out the possibility of concurrent execution of cells.

$$interl \triangleq \bigwedge_{k=1}^{bd} \left(\bigwedge_{p=1}^{u} \left(fl_p[k] \Rightarrow \neg \left(\bigvee_{q=1, q \neq p}^{u} fl_q[k] \right) \right) \right)$$

At last, we use *negation* form, $\neg\rho$, to denote the LTL property to be checked. The BMC problem on an HSTM design D with the properties ρ is defined as follows:

$$BMC(D, \rho, bd) \triangleq \left(\bigwedge_{k=0}^{bd} step[k] \right) \wedge interl \wedge negation\_\rho$$

3.3 Formalization and Encoding for HSTM Designs Communicating by Sharing Variables

In Section 3.2 the formalization and encoding methods for communicating HSTM by message passing is proposed. In this section, the formalization and encoding approach for HSTMs communicating by sharing variables are presented.

3.3.1 Formalization

The formalized **Definition** 1 to **Definition** 3 are the similar to the definition in Section 3.2. Only **Definition** 4 and **Definition** 5 are redefined in this section.

Definition 4 (HSTM Configuration). *Let* $HSTM = (STM_0, STM_1, \ldots, STM_m, R)$. *Let*

$VS(STM_i) = VS(STM_i) \cup \{c\text{-}flag_i\}$, where $0 \leq i \leq m$ and c-flag$_i$ is an additional boolean variable introduced to each STM_i. An HSTM configuration is defined as $C_{hstm} = \{C_{stm_0}, \ldots, C_{stm_m}\}$.

Definition 5 (An HSTM Design). *Let $\mathbb{N} = \{0, 1, \ldots, n\}$. An HSTM design D is defined as a tuple $D = (HSTM_0, \ldots, HSTM_n)$. Dynamic behaviors of D is captured by a transition system of the form $(\mathbb{C}_D, I_D, \rightarrow)$, where $\mathbb{C}_D$ denotes all reachable configuration of D, and each $C_D \in \mathbb{C}_D$ is of the form $\{C_{hstm_0}, \ldots, C_{hstm_n}\}$; $I_D = \{I_{hstm_0}, \ldots, I_{hstm_n}\}$ is the initial configuration; and $\rightarrow$ is the transition relation characterizing how D evolves from on configuration to another, i.e., $C_D \rightarrow C'_D$.*

Transition $\rightarrow$ is defined with two rules (external and internal) also. Same as before, $r1$ denotes internal rules and $r2$ denotes the external rules. The rule is redefined as follows.

The rule $r1$, which corresponds to the execution of a normal cell c in $STM_i \in HSTM_j$, $0 \leq i \leq m$ and $0 \leq j \leq n$, is written as: $C_D \xrightarrow{r} C'_D$, where

$$
\begin{aligned}
r1.condition \triangleq\ & v(c - flag_i^j) \wedge\ v(event(c)) \wedge\ v(guards(c)) \\
& \wedge\ v(actStatus_i^j) = index(source(c)) \\
r1.effects \triangleq\ & C'_D = C_D[C'_{hstm_0}/C_{hstm_0}, \ldots, C'_{hstm_n}/C_{hstm_n}]
\end{aligned}
$$

The symbols in above formulas denote same function with the formulas in Chapter 3.2. Detailed definition are shown in Figure 3.3.

The second rule $r2$ corresponding to the dispatch of the external event e_{Ext} to a $STM_i \in HSTM_j$, is defined as:$C_D \xrightarrow{r2} C'_D$, where

$$
\begin{aligned}
& r2.condition \triangleq v(xE) = \texttt{false} \\
& r2.effect \triangleq C'_D = C_D[C'_{hstm_j}/C_{hstm_j}], where \\
& \quad C'_{hstm_j} = \{C_{stm_0}, \ldots, C_{stm_i}[\texttt{true}/v(xE)], \ldots, C_{stm_m}\}
\end{aligned}
$$

(1) If $actions(c)$ is a list of statements in the form of Case1:
$C'_{hstm_j} = \{C_{stm_0},\ldots,C_{stm_i}[v(rhs)/v(lhs),\ index(target(c))/v(actStatus_i^j)],\ldots,C_{stm_m}, q\}$
For C'_{hstm_w}, where $0 \leq w \leq m \wedge w \neq j$, $C'_{hstm_w} = C_{hstm_w}$

(2) If $actions(c)$ is a list of statements in the form of Case2:
$C'_{hstm_j} = \{C_{stm_0}, \ldots, C_{stm_i}[v(rhs)/v(lhs), \texttt{false}/v(c-flag_i^j)],\ldots, C_{stm_z}[\texttt{true}/\text{v(c-flag}_z^j)],$
$\ldots, C_{stm_m}\}$
For C'_{hstm_w}, where $0 \leq w \leq m \wedge w \neq j$,
$C'_{hstm_w} = \{C^w_{stm_0}[\texttt{false}/v(c - flag_0^w)], C^w_{stm_1}, \ldots, C^w_{stm_m}\}$
(3) If $actions(c)$ is a list of statements in the form of Case3:
$C'_{hstm_j} = \{C_{stm_0},\ldots, C_{stm_i}[\texttt{false}/v(c - flag_i^j),\ index(target(c))/v(actStatus_i^j)],$
$\ldots, C_{stm_z}[\texttt{true}/v(c - flag_z^j),\ v(rhs')/v(lhs'),\ index(target(c^z))/v(actStatus_z^j)],$
$\ldots, C_{stm_m}\}$
For C'_{hstm_w}, where $0 \leq w \leq m \wedge w \neq j$, $C'_{hstm_w} = \{C^w_{stm_0}[\texttt{true}/v(c - flag_0^w)],$
$C^w_{stm_1}, \ldots, C^w_{stm_m}\}$

Figure. 3.3: Definition of C'_{hstm_j} with respect to transition rule $r1$.

3.3.2 Encoding Approach for HSTM

The idea for converting a transition system into a logical formula and using SAT/SMT solving [62, 11], to conduct bounded model check has been proposed in [10], [70] etc. Our encoding partially follows the same techniques used in these works but tuned to HSTM designs.

Encoded formula for step 0, denoted $step[0]$, that represents the initial configuration of D is defined as:

$$step[0] \triangleq \bigwedge_{x \in VS(D)} x[0] = v(x)$$

The formula simply expresses that all variables involved in D, given step number 0 (i.e., rename/substitute each variable x with a new variable name $x[0]$), have their initial (default) values. For $0 \leq i \leq$ m and $0 \leq j \leq$ n: initial values for $actStatus_j^i$ are 0; initial values for $c\text{-}flag_j^0$ are true and other *c-flag* variables are false; other unmentioned variables have user-specified initial values. The initial values for *c-flag* variables characterize that only root $STMs$

are possible to execute initially.

To define encoded formulas for steps other than 0, we first encode rules $r1$ and $r2$. For $r1$, we consider a normal cell c of a $STM_i \in HSTM_j$. The formula for condition of *r1* is:

$$r1.condition[k] \triangleq c - flag_i^j[k-1] \wedge\ event(c)[k-1] \wedge guards(c)[k-1]$$
$$\wedge\ actStatus_i^j[k-1] = index(source(c))$$

Expressions $event(c)[k-1]$ and $guards(c)[k-1]$ are generated by simply giving step number $k-1$ to all variables involved in them. Variable $actStatus_i^j[k-1]$ is generated by renaming. The encoded formula for effects of $r1$ is defined as:

$$r1.effects[k] \triangleq\ actions(c)[k] \wedge \left(\bigwedge_{x \in VS(D)/X} x[k] = x[k-1] \right)$$

where X denotes the set of variables that have been changed in $actions(c)$. We then directly follow the case analysis shown in Figure 3.3 to define $actions(c)[k]$.

If actions(c) is of the form **Case1**. We assume that the list of $lhs = rhs$ assignments is st_1,...,st_t. Then each $st_l[k]$, $1 \leq l \leq t$, is defined as $lhs_l[k] = rhs_l[k-1|k]$, where $rhs_l[k-1|k]$ means (1) if variables lhs_1,...,lhs_{l-1} occurs in rhs_l, the occurrence of these variables in rhs_l is given step number k, and (2) for other variables, step number $k-1$ is given. The formula for $actions(c)$ is defined as follows where change of active status is added.

$$actions(c)[k] \triangleq \left(\bigwedge_{l=1}^{t} st_l[k] \right) \wedge actStatus_i^j[k] = index(target(c))$$

If *actions(c)* is of the form **Case2**. We assume that the calling statement is $\Box z$ and that the list of $lhs = rhs$ assignments before it is st_1,...,st_t. The formula for $actions(c)$ is defined as

follows, where each assignment st_l is encoded in the same way as in **Case1**.

$$actions(c)[k] \triangleq \neg c-flag_i^j[k] \wedge c-flag_z^j[k] \wedge \left(\bigwedge_{l=1}^{t} st_l[k]\right) \wedge \left(\bigwedge_{0\le w\le n}^{w\neq j} \neg c-flag_0^w[k]\right)$$

If $actions(c)$ is of the form **Case3**. We assume that the remaining list of $lhs' = rhs'$ assignments after the calling statement in the cell c^z is $st'_1, \ldots, st'_t$. The formula for $actions(c)$ is defined as follows similar to **Case2**.

$$actions(c)[k] \triangleq \neg c-flag_i^j[k] \wedge c-flag_z^j[k] \wedge \left(\bigwedge_{l=1}^{t} st'_l[k]\right) \wedge \left(\bigwedge_{0\le w\le n}^{w\neq j} c-flag_0^w[k]\right)$$
$$\wedge actStatus_i^j[k] = index(target(c)) \wedge actStatus_z^j[k] = index(target(c^z))$$

Encoded formula for $r1$ that corresponds to the execution of a normal cell is defined as follows by combining $r1's$ encoded condition and effects formulas.

$$r1[k] \triangleq r1.condition[k] \wedge r1.effects[k]$$

Encoded formula for *r2* that corresponds to the dispatch of an external event is defined similarly as follows.

$$r2[k] \triangleq \neg xE[k-1] \wedge xE[k] \wedge \left(\bigwedge_{x\in VS(D)/\{xE\}} x[k] = x[k-1]\right)$$

We define the whole encoded formula executed at one step k. We use $R1 = \{c|c \in STM_i^j.C_{nor}\}$ and $R2 = \{e|e \in STM_i^j.E_{ext}\}$ to denote the set of normal cells and external events in D, respectively, where $|R1|+|R2| = u$. We introduce a set of boolean variables $\{fl_1, \ldots, fl_u\}$, each corresponding either to a $r1 \in R1$ (denoted $fl(r1)$) or to a $r2 \in R2$ (denoted $fl(r2)$). The

formula at one $step[k]$ is defined as:

$$step[k] \triangleq \left(\bigvee_{r1 \in R1} (r1[k] \wedge fl(r1)[k]) \right) \vee \left(\bigvee_{r2 \in R2} (r2[k] \wedge fl(r2)[k]) \right) \vee \left(\left(\bigwedge_{r \in R1 \cup R2} \neg r.condition[k-1] \right) \wedge \left(\bigwedge_{x \in VS(D)} x[k] = x[k-1] \right) \right)$$

Step formula $step[k]$ essentially represents all possible Ds configurations (states) that could be reached from the initial configuration (represented by $step[0]$) after applying (in any possible orders) $k1$ transition rules. The third or-disjunction predicate captures the case when no rules could be enabled in depth $k1$. In this case, we let all variables have their values unchanged. However, we could also set a variable to true which represents that a deadlock has occurred. Note further that, formula $step[k]$ does not filter out the possibility of concurrent execution of multiple rules in step k (since multiple rules may be true if the whole formula is true). We thus define the following formula to restrict the interleaving execution manner of D.

$$interl \triangleq \bigwedge_{k=1}^{bd} \left(\bigwedge_{p=1}^{u} \left(fl_p[k] \Rightarrow \neg \left(\bigvee_{q=1, q \neq p}^{u} fl_q[k] \right) \right) \right)$$

Finally, the BMC problem for an HSTM design D with respect to a LTL property ρ is defined, in which $negation\_\rho$ denotes encoded formula for negation of ρ. The LTL encoding approach we used in this section is proposed in [71].

$$BMC(D, \rho, bd) \triangleq \left(\bigwedge_{k=0}^{bd} step[k] \right) \wedge interl \wedge negation\_\rho$$

3.4 Summary

In this chapter, the formalization approaches for HSTM designs which utilize message-passing and sharing variables as the means of communication are proposed, and following

with the encoding approaches for both of them respectively. Preliminary experiments which are mentioned in Chapter 7 show that counterexamples (design errors) could be discovered effectively with our approach.

Chapter 4

Symbolic Bounded Model Checking Combined with Explicit Techniques

In this chapter, the first acceleration method for BMC is presented. The explicit model checking technique is used in the symbolic BMC to reduce some unreachable states in the state space of a system. The method proposed in this chapter is a stateless technique that is different from previews stateful combined BMC techniques.

4.1 Introduction

HSTM [72] is a table based modeling language for developing designs of software systems. An HSTM design, namely a design developed with HSTM, consists of multiple STMs organized in a hierarchical structure. In Chapter 3 and [69, 73], we have presented approaches to encode the HSTM design together with the negation of an invariant property to be checked against the design to logical formulas. With the proposed encoding approaches, the HSTM design could be conducted by BMC. After the encoding phase, the SMT solver is used to solve the problem by determining its satisfiability. If satisfied, a model of the formula (namely an interpretation to all the variables involved in the formula) is a witness of some bad behaviors

of the design that violate the property (namely a counterexample).

However, one problem of the proposed approach that we have observed is that, although the state-explosion problem could be avoided, its verification speed is slow, especially for large bounds, and such inefficiency has also been reported in [74]. In [63], the authors proposed an efficient heuristics-based idea to improve verification efficiency by introducing additional knowledge formulas into a BMC instance, but this idea relies heavily on human verifiers experience and expertise to come up with suitable/useful knowledge and thus could not be made automatic and is not implemented.

In this chapter, the approaches to accelerating SMT-based BMC are proposed. The approaches center around an unrolled bounded reachability tree (BRT) of anf HSTM design which is built with stateless explicit state exploration (that is, states are not saved during exploration). Specifically, reachability of invalid cells (representing undesired states) of an HSTM design, which occurs within the bound concerned, could be discovered during construction of the BRT, and furthermore, if no such occurrence, the constructed BRT could be utilized to rule out unnecessary subformulas of a BMC instance and thus make the instance easier to solve. By such combination,we could enjoy the benefits of both explicit exploration and BMC with respect to speed as well as memory. In addition, we observe that much BMC verification time is consumed by iterative search (i.e., gradually increase the search depth till the concerned bound), which is necessary for finding the shortest counterexamples. We propose a binary search algorithm to avoid iteration but still guarantee to find the shortest counterexamples, if any. We have implemented these approaches in Garakabu2. Our preliminary experiments show that verification could be accelerated substantially. The simplified Money-Exchange Machine (MEM) which is mostly the same as the example mentioned in chapter 2 is used as running example in this chapter except mineral revisions: (1) the amount deposited to balance is changed to 30000 to deepen the depth of counterexamples, and (2) a variable dealCount is added to count conducted exchanges.

This chapter is organized as follows. In contrast, the classic BMC algorithm is mentioned in Section 4.2. Section 4.3 proposed the static explicit algorithm and Section 4.4 presents the dynamic explicit algorithm. An iteration avoiding algorithm is proposed in Section 4.5 which is used to find the shortest counterexample. In Section 4.6 series experiments are conducted to evaluate the algorithms proposed above. The conclusion of the whole chapter is presented in Section 4.7.

4.2 Review of Classic BMC

BMC [10] is a technique for reasoning counterexamples within the bounded execution paths of a system that violate properties specified in LTL. The system under concern can be generally represented by a Kripke structure of the form $M = (S, I, T, L)$, where S is a set of states, $I \subseteq S$ in the set of initial states, $T \subseteq S \times S$ is the transition relation (which must be total), and L is the labeling function.

Given a Kripke structure M, an LTL formula f, and a bound k, the basic idea of BMC is to construct a propositional formula $|[M, f, k]|$, which is satisfiable iff there exists an execution of M within k steps that violates f. Such satisfiability can be solved by modern efficient SAT or SMT solvers such as Yices [75], CVC4 [59], and Z3 [23]. Formally, the high-level encoded propositional formula can be presented as $[|M, f, k]| \triangleq |[M]|_k \wedge |[\neg f]|_k$. For the encoding of $|[\neg f]|_k$, a simple translation that is linear both in the size of the formula and the length of the bound has been proposed in [71]. In this chapter, we focus on the encoding and optimization of $|[M]|_k$. The basic encoding described in the seminal paper [10] of BMC is as follows:

$$|[M]|_k \triangleq I(S_0) \wedge \bigwedge_{i=1}^{k} T_i(S_{i-1}, S_i) \tag{4.1}$$

where $I(S_0)$ is a predicate over state variables defining the initial states S_0, and $T_i(S_{i-1}, S_i)$ is the transition relation of M as a propositional formula.

4.3 Static Explicit Algorithm Aided Symbolic BMC

We could observe that step formula $step[k]$, $1 \leq k \leq bd$, described in Section 3.3.2 contains transitions that correspond to all cells as well as all external events in an HSTM design, even if some of them are impossible to execute in depth k. This seems to be inevitable since, during encoding, we do not know which ones are executable and which ones are not in k, and thus have to consider all possible transitions and let SMT solver compute its real execution.

A natural idea of accelerating BMC is to remove those unexecutable transitions (subformulas) from $step[k]$ for each k within the bound concerned. The resulted BMC formula could become smaller and thus easier to solve. We use a static explicit algorithm to denote this kind of technique in this chapter.

The initial attempt to be described in this subsection is based on Bounded (control) Status Tree (BST) of an HSTM design. Following the rules $r1$ and $r2$, which are used in Section **??** for defining transition relation, a BST could be constructed through expanding an HSTM design D statically by assuming the condition of these rules are always enabled except the predicates on restricting active status and active STM. That is, at each depth k, only those transitions, whose corresponding cells status and affiliated STM are both active at k, could be executed. We omit the detailed procedure of constructing BST.

For example, the BST of the MEM example is illustrated in Figure 4.1. Each node in the BST contains the status-combination of all STMs in an HSTM design. For simplicity, we use A and B to denote the status of STMs MainInterface and ReturnController, respectively, while the status of STMs Exchanger and Returner are omitted since their values are always 0. Each edge is labeled with transitions (correspond to cells or external events) whose execution on the source node results in status revolution specified in the target node. Again, for simplicity, we use S11-S22 to denote the four STMs, and S11.2.1.L, for example, to denote the transition corresponding to the left-hand side part (with guard `BillOutputAmount` > 0) of cell (2,1)

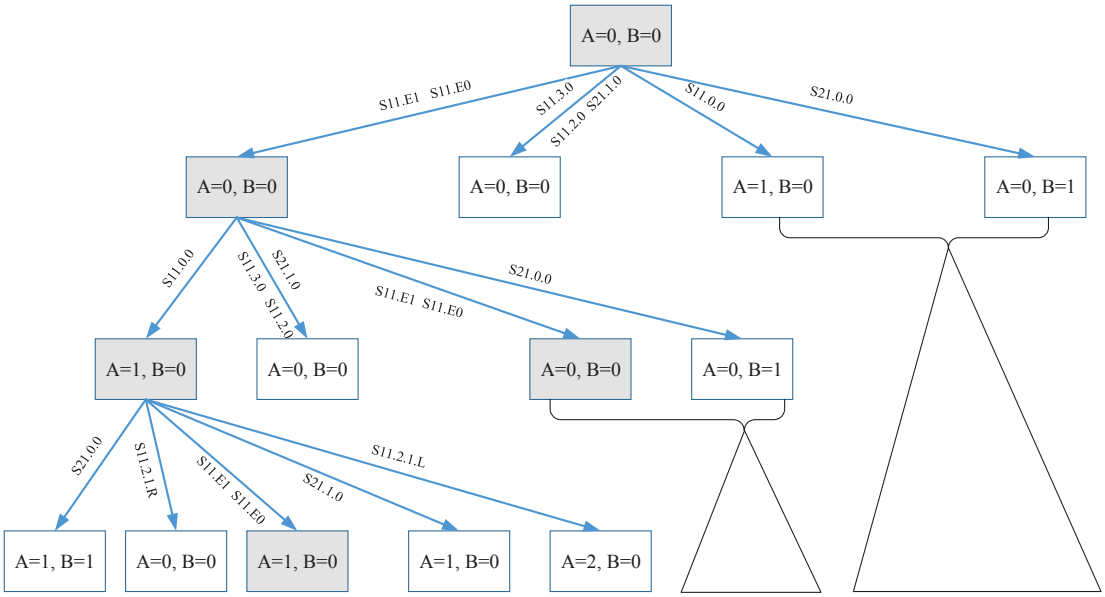


Figure. 4.1: Bounded Status Tree (BST) of the MEM till bound 3.

of STM MainInterface, and S11.E1, for example, to denote dispatch of the external event `xUserOperation` of S11. Note that a node with circled status-combination represents a status-combination where an invalid cell is reached. The triangles denote parts of the BST that are omitted.

The column titled BST in Table 4.1 shows all possible status-combination $Status_k$ and all possible transitions $Transition_k$ at each depth k till bound 3. Since $Transition_k$ contains all executable transitions (and more) at depth k, we could therefore remove from $step[k]$ any transitions that are not elements of $Transition_k$.

We implemented (temporarily) this attempt in Garakabu2 and the results are shown in the column NH of BST-based in Tables 4.2 and 4.3. The example and properties are similar with what we used in Chapter 3. The explanation for those properties is omitted here. Unfortunately, the results are not exciting due to neglectable improvement. After a close observation, we found that a BST easily got saturation at shallow depths, and it became quickly that no transitions could be removed. As an experiment, we further introduced heuristical status con-

Table. 4.1: States & Transitions of BRT and BST at Bound 3

	BST	BRT
$Status_0$	$\{< 0,0 >\}$	$\{< 0,0 >\}$
$Transitions_1$	{S11.0.0, S11.E0, S11.E1, S21.0.0, S11.2.0, S11.3.0, S21.1.0}	{S11.E0, S11.E1}
$Status_1$	$\{< 1,0 >, < 0,0 >, < 0,1 >\}$	$\{< 0,0 >\}$
$Transitions_2$	{S11.1.1, S11.E0, S11.E1, S11.2.1.L, S11.2.1.R, S21.0.0, S11.0.0, S21.1.1}	{S11.0.0, S11.E0, S11.E1}
$Status_2$	$\{< 1,0 >, < 2,0 >, < 0,0 >, < 1,1 >, < 0,1 >\}$	$\{< 1,0 >, < 0,0 >\}$
$Transitions_3$	{S12.0.0.L, S12.0.R, S11.3.2, S21.0.0, S11.0.0, S11.1.1, S21.2.1.L, S21.2.1.R, S22.0.0, S11.E0, S11.E1, S12.E0, S22.E0}	{S11.0.0, S11.E0, S11.E}
$Status_3$	$\{< 1,0 >, < 1,1 >, < 0,0 >, < 2,1 >, < 0,1 >\}$	$\{< 1,0 >, < 0,0 >\}$

Table. 4.2: Verification results of MEM (Accumulative Time in Sec.)

PROP	BD	VERDICT	ORIG	BST-NH	BST-H	BRT-CON	BRT-NI	BRT-BCS
UIC1	40	NO CX	86	84	-	1	1	**1**
UIC2	40	WHEN BD=30	19	19	-	1	1	**1**
SSC1	40	WHEN BD=26	12	12	-	3	13	**1**
SSC2	40	WHEN BD=28	17	16	-	4	13	**1**
FCF1	40	WHEN BD=28	22	22	-	18	19	**17**
FCF2	40	NO CX	16	15	-	15	15	**7**
SimG1	40	WHEN BD=32	26	26	-	21	15	**1**
SimG2	40	NO CX	126	121	-	106	47	**1**

Table. 4.3: Verification Results of the Revised MEM
(Accumulative time in Sec.)

PROP	BD	VERDICT	ORIG	BST-NH	BST-H	BRT-CON	BRT-NI	BRT-BCS
UIC1	40	NO CX	74	71	65	1	1	**1**
UIC2	40	NO CX	72	71	72	1	1	**1**
SSC1	40	NO CX	74	73	-	6	27	**1**
SSC2	40	NO CX	83	78	-	5	27	**1**
FCF1	40	WHEN BD=28	23	22	22	19	19	**18**
FCF2	40	NO CX	16	16	15	14	14	**6**
SimG1	40	WHEN BD=32	26	26	26	20	14	**1**
SimG2	40	NO CX	101	95	95	76	32	**1**

straints, e.g., SSC1 and SS2, for verifying the revised MEM. The idea here is to further filter out those transitions whose execution results in a status-combination that is out of the introduced constraints. However, the results shown in the column H of Table 4.3 are still not encouraging.

4.4 Dynamic Explicit Algorithm Aided Symbolic BMC

We keep on the direction of removing unnecessary transitions from $step[k]$. This time, instead of statically expanding an HSTM design D, we use explicit state exploration technique (Breadth First Search (BFS) is adopted in this chapter) to execute D and construct a Bounded Reachability Tree (BRT). In Figure 3, the rectangles with grey background, together with their associated edges, comprise the BRT of the MEM example. The column titled BRT in Table 4.1 shows the sets $Status_k$ and $Transition_k$, $0 \leq k \leq 3$, which could be computed during BRT exploration. The algorithm of exploring BRT is shown in Algorithm 1. Note that we further classify $Transition_k$ into In_Trans_k, denoting transitions leading to invalid cells, and Nm_Trans_k, denoting other normal transitions.

It should be noted that the key difference of this exploration with normal (bounded) BFS

model checking algorithms [9, 48], is that states that have been explored are not saved in memory (except those temporally saved in state queue). Saving explored states is necessary for normal BFS to avoid exploring same states, e.g., without state saving, a normal BFS might simply fall into a loop just continuously exploring two states in the loop. However, state saving is not necessary for our purpose since we are only interested in knowing which transitions are executable in the reachable states at depth k, and do not care whether those states have been explored in earlier depth. It should be noted as well that such exploration will not cause the state-explosion problem.For a normal BFS algorithm, however, even if only reachable states within an execution bound are saved, this may still cause state-explosion problem if the target system is large and the bound concerned is deep.

Algorithm 1. Exploring_BRT

```
1. Input: An HSTM design D, a bound bd
2. Output: In_Trans_k, Nm_Trans_k, 1 ≤ k ≤ bd
3.
4. Add_nodeQueue(Q, n_0), where n_0 is the initial node;
5. while (Empty_nodeQueue(Q) == FALSE) do
6.   n = Del_nodeQueue(Q);
7.   if (n.bound >= bd) then
8.     break;
9.   end if
10.  for all t ∈ Enabled_Transition(D, n) do
11.    m = Compute_ChildNode(n,t) where m.bound=n.bound+1;
12.    if (Is_InvalidCell(m)) then
13.      Add_Transition(t, In_Trans_m.bound);
14.    else
15.      Add_Transition(t, Nm_Trans_m.bound);
16.    end if
17.    if (In_Queue(Q, m) == FALSE) then
18.      Add_nodeQueue(Q, m);
19.    end if
20.  end for
21. end while
22. return In_Trans_k, Nm_Trans_k;
```

In Algorithm 2, it shows a BMC algorithm that utilizes the $Transition_k$ information obtained through exploring BRT. Note that in line 16, only concrete executable transitions at depth k are encoded (rather than all possible transitions in the original approach, and all statically executable transitions in the BST-based approach), and in line 19, we add constraints on active STM to reduce the search burden of a SMT solver. The constraints are in the form of, e.g., $c - flag_{Exchanger} \Rightarrow \neg c - flag_{MainInterface}$, since execution of a child STM will prevent the execution of all other STMs.

Algorithm 2. BMC_with_BRT

1. **Input:** An HSTM design D, a LTL property p, a bound bd
2. **Output:** TRUE/FALSE(CX)
3.
4. Exploring_BRT, // generating In_Trans$_k$, Nm_Trans$_k$;
5. **if** (Is_CheckInvalidCell(p)) == TRUE) **do**
6. **if** (Empty_Set($\cup_1^{bd}$ In_Trans$_k$) == FALSE) **do**
7. **Return** FALSE(CX) // CX could be computed (discuss later);
8. **end if**
9. **end if**
10. step[0] = Encode_Init(D, 0) // following the method in Section 3;
11. prop[0] = Encode_Prop($\neg p$, 0) // following the approach in [71];
12. **if** (Check_with_Solver(step[0], prop[0]) == SAT)
13. **return** FALSE(CX);
14. **end if**
15. **for** $1 \leq k \leq bd$ **do**
16. **for all** t $\in$ Nm_Trans$_k$ **do**
17. allTran = allTran$\vee$Encode_Tran(t, k) // allTran is empty initially;
18. **end for**
19. step[k] = allTran$\wedge$ Active_STM_Constraints$_k$;
20. prop[k] = Encode_Prop($\neg p$, k);
21. **if** (Check_with_Solver(step[0]$\wedge \ldots \wedge$step[k], prop[k]) == SAT)
22. **return** FALSE(CX);
23. **end if**
24. **end for**
25. **return** TRUE;

This BRT-based approach has been implemented in Garakabu2. The verification results are shown in the subcolumn CON of Tables 4.2 and 4.3. We could observe that the verification speed is generally accelerated, and particularly, verification of reachability as well as unreachability of invalid cells is accelerated dramatically thanks to the fast speed of explicit state exploration. It should be noted that, a counterexample reported for reachability of an invalid cell is the shortest one since breadth first exploration is used. However, no detailed counterexample information (e.g., variable values after each transition in the counterexample) is directly available since no states are saved during exploration. We could circumvent this by recording all paths (sequences of transition names) leading to the invalid cell during exploring BRT, and then simulate one of the paths to generate detailed counterexample information. Note also that, in Algorithm 2, during the explicit exploration of the BRT, only reachability of invalid cells were analyzed. However, we later realized that all reachability properties, including SSC1, SSC2, SimG1, and SimG2, could be analyzed/checked during explicit exploration.

4.5 Iteration Avoiding in BMC

Our next acceleration attempt is to avoid iterative verification [10] in BMC. Iterative verification is generally used in BMC, which means a verification in which the searching depth is gradually increased (usually by 1) until a counterexample is found or the bound concerned is reached. This approach is also adopted in Algorithm 2. Iterative verification is necessary for BMC to find the shortest counterexamples, which provide more accurate and useful violation information to a property concerned than non-shortest counterexamples. However, a disadvantage of this approach is that much more verification time is needed. For example, the time listed in columns other than the last column BRT-NI in Tables 4.2 and 4.3 is accumulative time used to iteratively verify all depths till the bound (or till the depth where a counterexample is found).

Algorithm 3. BMC_with_BRT_without_Iteration_for_ReachabilityProperty

1. **Input:** An HSTM design D, a reachability property p, a bound bd
2. **Output:** TRUE/FALSE(CX)
3.
4-14. Same as those in Algorithm 2.
15. low = 0; high = bd; mid = 0; cxPosition=-1;
16. Set list: c_result[0] = "nocx"; c_result[1... bd] = "unknown";
17. Generate: step[1]...step[bd], prop[1]...prop[bd] with the functions; Encode_Tran and Encode_Prop, respectively, in Algorithm 2;
18. **while** (low < high) **do**
19. mid = (low + high)/2;
20. **if** (Check_with_Solver((step[0]$\wedge \ldots \wedge$ step[mid])$\wedge$ (prop[1]$\vee \ldots \vee$prop[mid])) == SAT) **then**
21. c_result[mid]="cx"; high = mid;
22. **else**
23. c_result[mid] = "nocx"; low = mid + 1;
24. **end if**
25. **end while**
26. **if** (c_result[mid]=="nocx" $\wedge$ c_result[mid+1]=="cx") **then**
27. cxPosition = mid+1;
28. **else if** (c_result[mid]=="cx" $\wedge$ c_result[mid-1]=="nocx") **then**
29. cxPosition = mid;
30. **end if**
31. **return** (cxPosition>=0) ? FALSE(CX at cxPosition) : TRUE;

We show a BMC algorithm of HSTM designs for reachability properties (namely **G** properties) – Algorithm 3. The algorithm employs a binary search approach, and could avoid iterative verification while guaranteeing to report only the shortest counterexamples, if any. The key observation behind this algorithm is that: If BMC_k is satisfiable (has counterexample), then BMC_{k+1} is also satisfiable.

$$BMC_k = (step[0] \wedge \ldots \wedge step[k]) \wedge (\neg p[0] \vee \ldots \vee \neg p[k])$$

$$BMC_{k+1} = (step[0] \wedge \ldots \wedge step[k] \wedge step[k+1]) \wedge (\neg p[0] \vee \ldots \vee \neg p[k] \vee \neg p[k+1])$$

This observation simply follows the definition of $step[k+1]$ in Section 3.3 of Chapter 3, in which an or-disjunction subformula is defined to capture the case when no transitions are executable (called the "deadlock" case). In other words, subformula $step[k+1]$ in BMC_{k+1} is always satisfiable since either a transition in it or the "deadlock" case is satisfiable. Thus, the algorithm tries to find two consecutive positions in which the latter is checked to have a counterexample and the former has not. We have also implemented this approach in `Garakabu2`. The verification results in the column NI of Tables 4.2 and 4.3 show that, for BMC of reachability properties, this approach could further improve greatly the verification efficiency no matter whether counterexamples exist or not.

4.6 Summary

We presented in this work several attempts for accelerating SMT-based BMC of HSTM designs. Among proposed approaches, the BRT-based approaches, through making advantages of stateless explicit state exploration, have shown their effectiveness in acceleration. Regarding the BST-based approaches, although we failed to obtain exciting results from them, we hope this experience could be shared and inspire further investigation. Furthermore, the BRT-based approaches have been implemented in Garakabu2, a BMC tool for HSTM designs. Garakabu2 has been developed with special consideration for its usabilities for non-expert verifiers, e.g., graph-based counterexample simulation and graphic LTL editor [63]. We hope our this work, together with its usability, could make Garakabu2 practically more usable for on-site software development with HSTM.

Chapter 5

An Incremental SESE Technique with BCS and a Divide & Conquer Method

In this chapter, the second acceleration technique is proposed. Utilizing this technique, the state space of a target system is decomposed into different parts which can be verified in parallel. The efficiency and scalability of symbolic BMC are further increased.

5.1 Introduction

Bounded model checking (BMC) [10] is a restricted form of model checking technique, which reasons bounded execution paths of a system design against a desired property. In satisfiability modulo theory (SMT) [11] based BMC (simply called BMC below), a model checking problem is converted into a formula-satisfiability problem and analyzed with SMT solvers. SMT solving involves time-intensive computations and is often sensitive to the formula size, and therefore, the performance of BMC is heavily influenced by the system size and the checking bound.

The target of this chapter is to improve the performance of BMC for both safety and liveness properties expressed in linear temporal logic (LTL). An appealing idea for this is that

simpler is better, namely a smaller-sized formula is usually easier to solve [71]. Obeying this principle, we proposed in our prior work [46] the idea of integrating stateless explicit-state exploration (simply called SESE hereafter) into the BMC procedure as a preprocess. The key is to utilize SESE to prune transitions that are unexecutable at each depth till the user-specified bound so as to decrease the encoded formula size. However, we observed that the saturation problem occurs easily for deep depth, that is, all transitions become executable and none of them could be removed.

In this chapter, the work in chapter 4 is extended and novel contributions are made in several aspects as listed below. First, bounded context switch (BCS) [16, 17], an under-approximation technique, is integrated into SESE. It has been found that only a few context switches (i.e., execution-order changes) are suffice to reveal concurrency bugs [16, 18]. Such integration thus allows SESE to explore a limited number of context switches of multiple parallel processes in the system so as to reduce the state space. Second, rather than encoding all legal execution paths, which are memorized during SESE, into a single (usually large) formula and inquiring its satisfiability of SMT solvers, we introduce heuristic predicates and use them to classify the paths into path clusters. Each path cluster can be considered as an independent BMC instance, which is usually smaller and easier to solve. Furthermore, multiple such BMC instances can be solved concurrently with multiple SMT solvers running on multicores. Since no information sharing is needed among these independent BMC instances, once a counterexample is found, the computation on all other cores can be safely terminated. Third, rather than directly applying SESE and BMC to a user-specified bound, we gradually deepen the checking depth from 0 with a fixed incremental number. Such iteration finishes until a counterexample is found or the bound is reached. In this way, counterexamples that are shorter than the user-specified bound can be revealed while avoiding expensive computation between the depths where the counterexample is found and the specified bound.

5.2 Motivation of SESE with BCS

In this chapter, the formalization of target STMs is redefined. A new formalization is presented in this section. BMC has become a complementary technique to symbolic model checking methods based on binary decision diagrams (BDDS) [9]. Based on performance comparison on industry benchmarks [28], BMC has been reported to be efficient, especially for hardware systems verification. However, our experiences have shown that BMC does not work well for asynchronous software designs.

Consider the sample design D (in Table 5.1) written in State Transition Matrix (STM) [72], a popular modeling notations often used in embedded software industry. $EVT_0 \ldots EVT_n$ are predicates denoting events that may be dispatched to the system D, and $ST_0 \ldots ST_m$ are status that D may be in (Initially D is in status ST_0 by default). The informal meaning of the design is, for example, if event EVT_n happens when D is in status ST_m, then action act_{nm} will be executed and after that D switches its status to ST_* (some status of $\{ST_0 \ldots ST_m\}$). Let $C_{ij} = EVT_i \wedge ST_j$ denote the enable condition, and $E_{ij} = act_{ij} \wedge ST_*$ denote the effects of executing the cell indexed with $i \in \{0 \ldots n\}$ and $j \in \{0 \ldots m\}$. Note that one cell is nondeterministically executed even if multiple cells are enabled, and that D is only in one status at a time. Following the encoding method in [10], $T_1(S_0, S_1)$ for time step 1 is given in a high-level form as follows (more detailed descriptions of encoding STM designs can be found in [63, 69]):

$$T_1(S_0, S_1) \triangleq \underbrace{\bigvee_{i=0}^{n} \bigvee_{j=0}^{m} (C_{ij}^1 \wedge E_{ij}^1)}_{(a)} \vee \underbrace{(\bigwedge_{i=0}^{n} \bigwedge_{j=0}^{m} (\neg C_{ij}^1) \wedge (S_0 = S_1))}_{(b)} \tag{5.1}$$

where sub-formula (a) states that one of the enabled cells executes, sub-formula (b) states that all state variables remain unchanged if no cells are enabled, and the superscript 1 denotes that all state variables in those formulas are indexed with corresponding step numbers (either 0 or 1). Although the size of formula (5.1) would be very large in practice (representing $m \times n$ cells

Table. 5.1: A Sample Design D Written in State Transition Matrix (STM)

	ST_0	ST_1	$\cdots$	ST_m
EVT_0	ST_*	ST_*	$\cdots$	ST_*
	act_{00}	act_{01}		act_{0m}
EVT_1	ST_*	ST_*	$\cdots$	ST_*
	act_{10}	act_{11}		act_{1m}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
EVT_n	ST_*	ST_*	$\cdots$	ST_*
	act_{n0}	act_{n1}		act_{nm}

in D's table), this encoding is reasonable in the sense that all cells might possibly be executable at a given time step and we have to consider all the possibilities. Furthermore, a design D may contain multiple such STMs organized in a hierarchical structure, which will make formula (5.1) even larger.

However, recall that D is initially in status ST_0, which means that $C_{ij} = \bot$ for all $j \neq 0$. Intuitively, this means that at time step 1 only cells in the column ST_0 are executable. Thus, formula (2) can be reduced by keeping only the case of $j = 0$:

$$T_1(S_0, S_1) \triangleq \underbrace{\bigvee_{i=0}^{n} (C_{i0}^1 \wedge E_{i0}^1)}_{(a)} \vee \underbrace{(\bigwedge_{i=0}^{n} (\neg C_{i0}^1) \wedge (S_0 = S_1))}_{(b)} \tag{5.2}$$

Clearly the size of formula (5.2) (representing n cells) is much smaller than that of formula (5.1) (representing $m \times n$ cells), and is generally easier to solve by SAT/SMT solvers.

One problem of applying such simplification to other execution steps except the 1st step is that it is impossible to statically determine the status that D is in at an arbitrary time. However, by applying stateless explicit-state exploration (SESE) as a preprocess before constructing $|[M]|_k$ formula, we could circumvent this problem dynamically. Furthermore, rather than merely the status information, the exact cells that are executable at each time step could also be known, which could further help to reduce the formula size of $|[M]|_k$.

In the next sections, we will investigate the above idea for simplifying $|[M]|_k$ in more detail, and demonstrate the experimental results. Note that although designs written in state transition matrix is used in the discussions, our proposed techniques are applicable to BMC of general systems.

5.3 Incremental SESE Equipped with BCS

In prior work in [46], which are presented in Chapter 4, we have tentatively investigated the idea of integrating stateless explicit-state exploration (SESE) techniques into the basic BMC encoding method. Given a software design D and a specified depth k, the general idea is to utilize SESE to compute and memorize executable cells (transitions) at each depth till k, and only these cells are encoded into propositional formulas composing $|[M]|_k$ while others are omitted. However, such pure integration suffers from, among other efficiency problems, the saturation problem for large bounds. That is, all cells become executable at deep bounds (originated from possibly different execution paths), and thus, no cells could be removed.

In this chapter, the work in [46] is extended in several aspects. First, an interactive deepening stateless explicit-state exploration techniques is utilized, which could guarantee to find counterexamples shorter than k, and thus save unnecessary state exploration and BMC workload. Second, bounded context switching (BCS) techniques [16, 17] is incorporated into the exploration procedure to limit the execution-order changes between parallel processes of D, and thus to reduce the number of executable cells in each time step. Last, the execution paths remembered during state exploration could be grouped into path clusters, each of which is then encoded and solved in parallel with a multicore computing environment, and thus to accelerate solving speed. The novel approach is elaborated gradually in the following parts.

5.3.1 SESE technique with Bounded Context Switch

The basic stateful bounded breadth-first search (BFS) [9, 48] (slightly tuned for our demonstration purpose) is illustrated in Algorithm 1. BFS explores all successor states of a given state s within a pre-specified bound bd. Successor states that have not been visited before will be further explored. s_0 is the initial system state of D; stack Q saves the states to be explored; set $SSet$ saves all visited states; both Q and $SSet$ are initially empty. In addition to state variables, a state s also contains the fields *sbound* and *path*, which holds the information about the depth at which the state is visited, and respectively, the transition path (a list of transitions) through which the state is reached.

Algorithm 1. Basic Stateful Bounded Breadth-First Search (BFS)

```
1. Input: Design D, Search bound s_bd
2. Output: Set of reachable States SSet
3.
4. Add_stateStack(Q, s0); // where s0 is the initial system state
5. Add_stateSet(SSet, s0);
6. while (Empty_stateStack(Q) == FALSE) do
7.    s = Del_stateStack(Q); // get and remove a state from Q
8.    if (s.sbound >= s_bd) then
9.       break;
10.   end if
11.   for all t ∈ Enabled_Tran(D, s) do
12.      s' = Compute_SuccessorState(s, t);
13.      if (In_StateSet(SSet, s') == FALSE) then
14.         s'.sbound = s.sbound + 1;  s'.path = s.path.append(t)
15.         Add_stateStack(Q, s');   Add_stateSet(SSet, s');
16.      end if
17.   end for
18. end while
19. return SSet;
```

In the basic BFS algorithm, maintaining the state set $SSet$ is necessary to avoid exploring already visited states, e.g., without state saving, BFS might simply fall into a loop just con-

tinuously exploring two states in the loop. On the other hand, state saving may also cause the notorious state-explosion problem if the system is large and the bound given is deep. In the context of this work, however, state saving is not necessary since we are only interested in knowing which transitions are executable in states at each depth within k, and do not care whether the states have been explored in earlier depths or not.

Bounded context switching (BCS) techniques [16, 17] can be incorporated into the basic (either stateful or stateless) BFS algorithm to reduce the state space. The key idea of BCS, an under approximation approach, is to limit the context switches (execution-order changes, i.e., one active process stops and another is resumed) of multiple parallel processes of a system. Since concurrency bugs can often be revealed with a small number of context switches, BCS has been reported by the literature to be efficient for detecting bugs in concurrent systems [76, 16, 17].

The sketch of an algorithm that combines stateless BFS with BCS is illustrated in Algorithm 2. Each state is equipped with an additional field *cbound* for recording the number of context switches seen when reaching the state, and each transition is equipped with a field *pid* for recording its affiliated process.

Similar to the BCS method described in [17], we distinguish whether a context switch is a $forced$ one or not. If the set of enabled transitions on state s contains no transitions of the same process, by a transition of which s is reached (called currently active process blow), then this process is said to be blocked (i.e., not executable) and expanding s by executing transitions of other processes is said to be forced. Such evaluation is computed by the procedure Forced_ContextSwitching in the bottom of Algorithm 2. Lines 21-23 states that if an enabled transition is of a different process as the currently active one, and if the expanding is not forced, then this context switch is counted. The operator `[]` is used to access the transition at a given position, here the last (tail) transition, of a transition path. Lines 24-27 states that any successor state s' whose context-switch number exceeds the pre-specified context-switch number

c_bd will be ignored and not further explored.

Algorithm 2. Stateless BFS with Bounded Context Switch (BFS_BCS)

```
1. Input: Design D, Search bound s_bd, BCS bound c_bd, Prop f
2. Output: Set of reachable states SSet, or deadlock
3.         or TRUE/FALSE(CX), when prop f is a safety property
4.
5. Add_stateStack(Q, s0); // where s0 is the initial system state
6. Add_stateSet(SSet, s0);
7. while (Empty_stateStack(Q) == FALSE) do
8.   s = Del_stateStack(Q); // get and remove a state from Q
9.   if (Is_SafetyProp(f) ∧ Violation(s, f) == TRUE) then
10.    return FALSE(CX); // if found violation of safety property
11.  end if
12.  if (s.sbound >= s_bd) then
13.    break;
14.  end if
15.  if (Enabled_Tran(D, s) == EMPTY) then
16.    return deadlock;
17.  end if
18.  forced = Forced_ContextSwitching(D, s);
19.  for all t ∈ Enabled_Tran(D, s) do
20.    s' = Compute_SuccessorState(s, t);
21.    if (t.pid != s.path[tail].pid ∧ forced == FALSE) then
22.      s'.cbound = s.cbound + 1;
23.    end if
24.    if (s'.cbound <= c_bd) then
25.      s'.sbound = s.sbound + 1;   s'.path = s.path.append(t);
26.      Add_stateStack(Q, s');
27.    end if
28.  end for
29. end while
30. return SSet;

1. Procedure Forced_ContextSwitching(D, s)
2.   if (∃t ∈ Enabled_Tran(D, s): t.pid = s.path[tail].pid) then
3.     return FALSE; // if the currently active process is not blocked
4.   end if
```

Note that, if the property f to be checked is a safety property, Algorithm 2 checks the violation of f by a state s in lines 9-11, and a deadlock is reported when no transition is enabled on a state s in lines 15-17. Note also that in our actual implementation of Algorithm 2, before adding a successor state s' into stack Q (line 26), we make an additional check on whether a state s'', which has the same *sbound* and state-variable values as those of s', exists in Q. If it exists, the transition path of s' is merged into the path (cluster) of s'', and s' is not added into Q. This can help reduce the number of states to be explored by the algorithm. For demonstration simplicity, we omit the details and simply equip each state with a transition path rather than a path cluster.

We are now ready to describe the novel hybrid BMC method that integrates Algorithm 2 as a preprocess for decreasing the size of the encoded formula to be solved by SAT/SMT solvers. The algorithm is illustrated in Algorithm 3. We start from explaining the lines inside the outermost `while` loop. In line 9, the set $frontierSet$ of all reachable states at depth *curbd* are computed and the transition paths leading to them are recorded through executing the procedure BFS_BCS of Algorithm 2. The image of computing $frontierSet$ in each iteration is shown in (a) of Figure 5.1.

Deadlock is reported if any, and if f is a safety property, it is evaluated with BFS_BCS only (although we did not add a control flow for this in Algorithm 3). In lines 12-15, the transitions executed at each depth i in between 1 and *curbd* of all states in $frontierSet$ are merged and recorded into $tranSet[i]$, and only these transitions are encoded into a propositional formula representing $T_i(S_{i-1}, S_i)$ (lines 16-19). The combination of the initial step formula $I(S_0)$, and the step formulas $T_i(S_{i-1}, S_i)$ from 1 to *curbd*, together with the encoded formula for LTL property f is checked for satisfiability in line 21. Again, we follow the encoding approach for LTL properties proposed in [71] and omit the details. If the formula is satisfiable, then a counterexample of depth *curbd* has been found and Algorithm 3 returns; otherwise, the current bound is increased with *inc* and the above procedure repeats.

Algorithm 3. BMC Guided by Incrementally Deepening BFS_BCS

1. **Input:** Design D, Search bound s_bd, BCS bound c_bd,
2. Interaction deepening number inc, LTL prop f
3. **Output:** TRUE/FALSE(CX), or *deadlock*
4.
5. $I(S_0)$ = Encode_init(D, 0);
6. $curbd = inc$;
7. **while** ($curbd <= s\_bd$) **do**
8. // record reachable states at depth $curbd$; report deadlock if any
9. $frontierSet$ = BFS_BCS(D, $curbd$, c_bd, f)
10. $prop_{curbd}$ = Encode_Prop($\neg f$, $curbd$);//use the approach in [71]
11. **for** ($1 <= i <= curbd$) **do**
12. **for all** $s \in frontierSet$ **do**
13. // merge enabled trans at i; initially $tranSet[i]$ is empty;
14. $tranSet[i] = tranSet[i]$.insert($s.path[i]$);
15. **end for**
16. **for all** $t \in tranSet[i]$ **do**
17. // encode $T_i(S_{i-1}, S_i)$, which is initially empty;
18. $T_i(S_{i-1}, S_i) = T_i(S_{i-1}, S_i) \vee$ Encode_Tran(t, i);
19. **end for**
20. **end for**
21. **if** (is_SAT($I(S_0) \wedge \bigwedge_{i=1}^{curbd} T_i(S_{i-1}, S_i) \wedge prop_{curbd}$))
22. **return** FALSE(CX);
23. **end if**
24. $curbd = curbd + inc$;
25. **end while**
26. **return** TRUE;

The above method of incrementally deepening the search depth is inspired by a similar approach proposed in [77] for stateful explicit depth-first search (DFS) model checking of safety (invariant) properties. With such an incremental method, counterexamples shorter than depth s_bd could be revealed, and thus, unnecessary workload for explicit exploration (line 9), symbolic encoding (line 18), and SMT solving (line 28) can be avoided.

Recall, for example, formula (5.2) mentioned in Section 5.2. Algorithm 3 can guarantee that at each time step only cells (transitions) that are actually executable are encoded. Thus, sub-formula (a) of formula (5.2) could be further shortened by removing those unexecutable

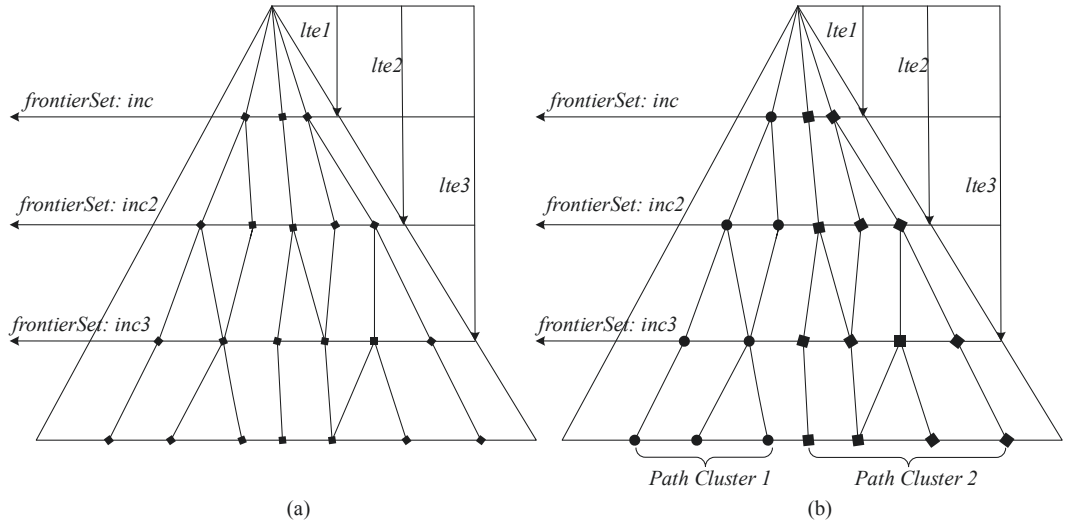


Figure. 5.1: (a): The image of computation of $frontierSet$ in each iteration; (b): The image of path clusters to be solved with multicores.

cells. Furthermore, sub-formula (b), which is potentially used to capture the deadlock case, can be totally removed since there must exist a cell that is executable (otherwise, a deadlock should have already been reported in line 9). To summarize, the size of the formulas $T_i(S_{i-1}, S_i), 0 \leq i \leq k$ can be decreased significantly by following Algorithm 3. Regarding the saturation problem that we met in our prior work [46], we found through experiments that, with an appropriate BCS number, saturation can be postponed to deeper bounds or resolved in many cases.

5.3.2 Divide and Conquer

Next, we present Algorithm 4, which extends Algorithm 3 by employing a divide-and-conquer approach to further improve efficiency. Given a set $predSet$ of n predicates, the key idea is to utilize these predicates to classify transition paths into clusters of paths, which are then encoded into independent BMC instances and solved in parallel with multicores or distributed environments. In lines 12-25 of Algorithm 4, for each predicate $pred_l$ $(1 \leq l \leq n)$,

Algorithm 4. A Divide-and-Conquer approach for BMC

1. **Input:** Design D, Search bound s_bd, BCS bound c_bd,
2. Interaction deepening number inc, LTL prop f,
3. Set $predSet$ of n predicates // for dividing transition paths
4. **Output:** TRUE/FALSE(CX), or *deadlock*
5.
6. $I(S_0)$ = Encode_init(D, 0);
7. $curbd = inc$;
8. **while** ($curbd$ <= s_bd) **do**
9. // record reachable states at depth $curbd$; report deadlock if any
10. $frontierSet$ = BFS_BCS(D, $curbd$, c_bd, f)
11. $prop_{curbd}$ = Encode_Prop($\neg f$, $curbd$);//use the approach in [71]
12. **for each** $pred_l \in predSet$, where $1 \leq l \leq n$ **do**
13. **for** (1 <= i <= $curbd$) **do**
14. **for all** $s \in frontierSet \wedge$ eval($pred_l$, s) == TRUE **do**
15. // merge enabled trans at i; initially $tranSet[i]$ is empty;
16. $tranSet[i] = tranSet[i]$.insert($s.path[i]$);
17. **end for**
18. **for all** $t \in tranSet[i]$ **do**
19. // encode $T_i(S_{i-1}, S_i)$, which is initially empty;
20. $T_i(S_{i-1}, S_i) = T_i(S_{i-1}, S_i) \vee$ Encode_Tran(t, i);
21. **end for**
22. **end for**
23. $bmcInstance_l = I(S_0) \wedge \bigwedge_{i=1}^{curbd} T_i(S_{i-1}, S_i) \wedge prop_{curbd}$;
24. $bmcSet = bmcSet$.insert($bmcInstance_l$);
25. **end for**
26. // compute the BMC instances in $bmcSet$ in parallel
27. is_SAT($bmcInstance_1$) || ... || is_SAT($bmcInstance_n$);
28. **if** $\exists l \in n$: is_SAT($bmcInstance_l$) == FALSE **then**
29. **return** FALSE(CX); // and stop all other executing threads
30. **else**
31. $curbd = curbd + inc$;
32. **end if**
33. **end while**
34. **return** TRUE;

the transition paths of state s, which is of the current frontier state set and satisfies $pred_l$, are extracted (lines 14-17). These transitions are then encoded to construct a propositional formula (lines 18-21). A BMC instance is composed of the propositional formula together with the

formula for the negation of the LTL property f (lines 23-24). Finally, multiple such BMC instances are solved in parallel: if a counterexample is found for any of the instances, then the counterexample is reported and all parallel processes are stopped; otherwise, the current bound is increased (lines 27-32) and the algorithm starts to explore deeper state space.

Predicates $pred_l$ are generated based on heuristics provided by human verifiers. In our actual implementation and experiments, we introduce a variable $statusVar$ to denote the current status that a STM is in, and correspondingly, $pred_l$ are of the form $statusVar = l$. Heuristics for predicates in other forms can also be used by Algorithm 4.

The image of such a divide-and-conquer approach is illustrated in (b) of Figure 5.1. Considering the frontier set at the bottom of the figure, the states in that frontier set are classified into those with circle marks and those with diamond marks. Precedent states on the paths leading to the corresponding states are equipped with the same marks. As an example, path clusters 1 and 2 can be encoded into two independent BMC instances and solved in parallel. In the case that two different path clusters contain common states (e.g., the states can be marked with both circle or diamond), the transitions leading to those states are simply considered to belong to both clusters and encoded. Through dividing the state space to a certain bound into multiple smaller ones, which are then encoded and solved concurrently, the BMC efficiency can be improved much. A feature of such a divide-and-conquer approach is that the BMC instances are totally independent ones and no communication among them, which can affect the overall efficiency much, is needed for solving them.

5.4 Summary

In this chapter, the work in [46] is extended and a novel hybrid approach that combines stateless explicit-state and BMC model checking techniques for system design verification is proposed. In particular, through explicit state space exploration, only paths leading to reach-

able states are encoded into BMC formulas, which avoids encoding non-executable transitions at each depth; bounded context switch techniques are integrated into the explicit exploration, which avoids process interleavings that are unnecessary for revealing counterexamples; deadlock and safety properties are checked during explicit exploration, which, as shown in the experiments in [28] and ours, is more faster than pure BMC techniques; execution paths are classified into clusters and encoded into independent BMC instances, which are solved in parallel with multicore computation; the hybrid checking progresses in an incrementally deepening manner, which guarantees to find shorter examples and avoids unnecessary explicit exploration and BMC workload. All of these techniques help accelerate the verification performance of pure BMC method for both safety and liveness properties.

Chapter 6

Distributed SMT Solving

In this chapter we describe the contribution on implementation of distributed SMT solving. With this method, the scalability and efficiency of SMT-based BMC can be improved effectively.

6.1 Introduction

SMT-based BMC consists of two primary tasks: (1) encoding a bounded model checking problem into a propositional formula that represents the problem, and (2) using a SMT solver to solve the formula, that is, finding a set of variable assignments that makes the formula true. Solving the formula (namely, SMT solving) involves computation-intensive processes and is thus time-consuming. Furthermore, as the model-checking bound increases, the encoded formulas become larger in size and harder to solve. The computational complexity of most SMT problems is very high [78], [22]. For all that, it is difficult to accelerate the SMT solving procedure for the engineers engaged in model checking. We have conducted [79] an implementation of using distributed computation and utilizing the power of multi-cores Central Processing Unit (CPU), multi-CPUs, and/or even cloud computing, to accelerate SMT solving. Although a series of experiments has shown the effectiveness of our implementation on increasing the solving efficiency, there still exist shortcomings which prevent the distributed solving to take

advantage of CPU cores as much as possible. For example, the communication protocols could be reformed in order to reduce the unnecessary usage of network communication. In this paper, we describe our work on improving the distributed SMT solving by changing the file dispatching schemes that consider work load balance, and by reforming the communication protocols. We change file dispatching from coarse-grained to fine-grained, which can help in increasing the usage of CPU cores. Some unnecessary steps in the communication protocols are removed or merged. We have discussed the effectiveness of our improvement theoretically. We repeat the experiments conducted in our previous work [79] to make comparisons between these two schemes. We also conduct an experiment by increasing the CPU cores used in parallel SMT solving to investigate the influence in a microscopic view. The experimental results demonstrate the feasibility and efficiency of our improved implementation. However, we have also found, for a given target benchmark, that increasing CPU cores involved in computing will not always increase the solving speed.

The rest of this chapter is structured as follows. Section II provides necessary preliminary knowledge and a brief introduction of the tools and techniques that are used in our work. Section III describes our previous work about distributed SMT solving. Section IV shows our methods used to improve the distributed SMT solving. Section V presents the experiments to evaluate the improvement and discusses the results. Finally, Section VI mentions possible extension (application scenarios) of our work and concludes the paper.

6.2 Background

In our implementation, there are some techniques and tools which will be introduced in this section. We use Message Passing Interface (MPI) and Open MPI to implement distributed and parallel, respectively. Z3 is a high performance SMT solver that is used as a backend solving core.

6.2.1 MPI and MPICH

Message Passing Interface (MPI) [80] is a standardized and portable message-passing system. The MPI standard defines the syntax and semantics of a core of library routines useful to wide range of user writing portable message passing programs in different program languages. It is a language-independent communications protocol used to program parallel computers. We choose MPI 2.0 standard in our implementation.

There are several well-tested implementations of MPI, such as LAM/MPI [81], MPICH and Open MPI [82] etc. They are designed for high performance on both massively parallel machines and on workstation clusters. MPICH [83] is a high-performance and widely portable implementation of the MPI standard. It provides an MPI implementation that supports different communication and computation platforms including commodity clusters, high-speed networks and proprietary high-end computing systems. It also provides enable cutting-edge research in MPI through an easy-to-extend modular framework for other derived implementations [83]. We use MPICH2 in our implementation to realize control signals and files transmission among different PCs. In the remaining part of this paper, the MPI and MPICH are all referred to MPICH2.

6.2.2 OpenMP

OpenMP (Open Multi-Processing) [84] is an API that supports multi-platform shared memory multiprocessing programming in C, C++, and FORTRAN, on most processor architectures and operating systems, including Solaris, AIX, Mac OS X, and Windows platform [85][86][87][88]. It is developed in 1997 and promoted and supported by a nonprofit organization named OpenMP Architecture Review Board (ARB). The members in ARB are AMD, Fujitsu, HP, IBM, Intel, NEC, and Oracle etc.

OpenMP is different from MPI. MPI is a multi-process mechanism but OpenMP is a multi-thread one. MPI supplies the communication ability to the processes even when the processes

are in different PCs. The MPI task manager allocates independent memory to every MPI process. OpenMP achieves parallel by creating multiple threads. The created threads exist in one PC and sharing same memory space.

6.3 A Prototype of Distributed SMT Solving

6.3.1 Overview

We have implemented distributed SMT solving in C language, using a Z3 SMT solver for satisfiability verification. The system has a Client/Server (C/S) architecture. The topology of the network is shown in Fig. 6.1. All clients are connected to a center server, data is transmitted between server and clients. The server responses the requirement of acquiring file from clients. If there exist enough `SMT` files, then the files will be sent to the target client. The SMT solving procedure happens on the clients after receiving `SMT` files from the server. OpenMP is used to create multiple threads, each thread invokes a Z3 SMT solver to solve specific `SMT` files. The solving process will be finished until the server has no file to send.

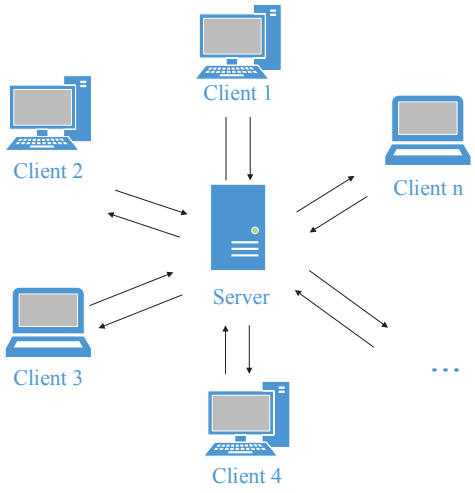


Figure. 6.1: Network Topology.

In our implementation, both MPICH and OpenMP are used to gain the purpose of taking

best advantage of clients' computing capacity. The architecture of the implementation is shown in Fig. 6.2. The MPI task managers which are running on different hosts, create server and clients processes independently. They have the unique names in `int` type, which can be used to distinguish these processes, numbered from `0` to `n`. The MPI task managers set up every process on different hosts assigned by user deterministically from the server. One PC has only one MPICH process running on it. The server and clients need to communicate with each other when the server dispatches workloads to the clients. MPI supplies the message passing mechanism which allows us to send control signals and files among hosts in a network. At this point, we are ready to apply distributed computing. OpenMP is used to perform further parallel SMT solving. In a MPICH process, OpenMP is utilized to create multiple threads which communicate with each other using shared variables. It should be noted that, on each client, the OpenMP threads are not created at the beginning of the process. They are created after the file transmission finished, only for parallel SMT solving. The file receiving and dispatching are completed by MPICH processes. One client creates 4 threads by default running in parallel. Each thread invokes a Z3 SMT solver to solve specific `SMT` files dispatched to it. Generally speaking, we use MPICH to achieve distributed computing and communication, and OpenMP to achieve parallel SMT solving as well.

6.3.2 Design of Server

The server takes charge of responding the requirement from clients and dispatching the `SMT` files to clients. If there exits enough files, they will be sent to the clients one by one. Every time four files are sent to one client by default. The default value can be changed by user as their wish. The server denoted by the `MPI process_0` in our implementation. The pseudocode of server is shown in Algorithm 6.1.

The server is running in a `do while` loop until there is no SMT file to be transferred to the clients. It starts with checking and receiving messages from any clients in the entire MPI

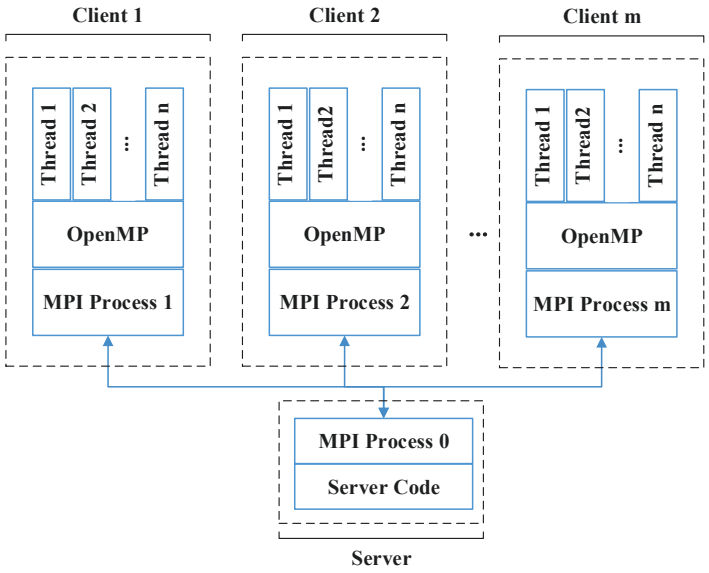


Figure. 6.2: Architecture of the Distributed Environment.

Algorithm 6.1 Pseudocode of the Server

```
1. Process_0 (nTasks) {
2.   Initialization and definition of variables;
3.   Timer(start);
4.   Terminate == 0;
5.   do {
6.      MPI_Recv(request_type from any client);
7.      if(require_type is files request){
8.        search_result=seek_file(process_id);
9.        if (search_result == no_file_found)
10.          MPI_Send( no_file_funded to process_id);
11.     }
12.     else
13.       terminate++;
14.  }while (terminate < (nTask-1));
15.  Timer(stop);
16.  Compute time consuming;
17.  return 0;
18. }
```

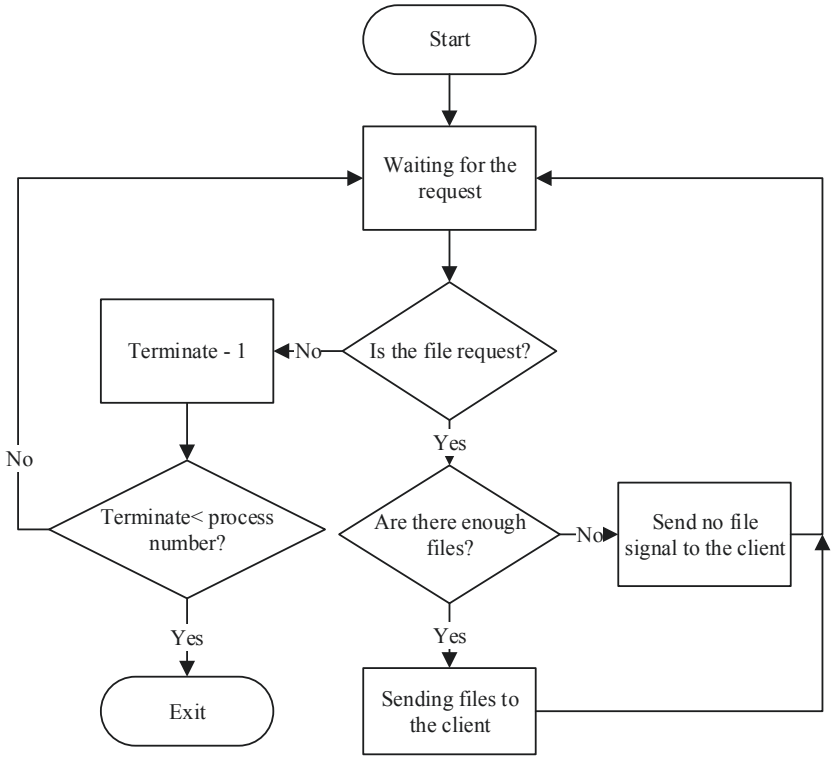


Figure. 6.3: Control Flow Graph of the Server.

communication domain using MPI function `MPI_Recv()`. It is paired with a blocked message sending function `MPI_Send()` invoked by clients at the beginning. The request receiving followed by determining the `request_type` which represents the type the message. If it is a message of requiring `SMT` files from a client, the function `seek_file(process_id)` is invoked. This function explores the local directory of the server, try to find whether there exits `SMT` files that can be sent to the client. If so, they will be sent to the client which require files. If not, the server will send a message to notify the client that the server does not have enough files. If the received message is not a file request message, the variable `terminate` will plus one. When the value of `terminate` is bigger than the number of the process, it means that all clients are terminated, the server `(process_0)` will stop loop and terminate. A timer, which is used to calculate the whole parallel solving procedure, is set at the beginning of the solving process and will stop after all processes finished their solving. The control flow graph is shown in Fig. 6.3. The file sending procedure takes place in the function `seek_file()`. The function

searches and sends files to the target process (denoted by `process_id)`. The pseudocode of `seek_file()` is shown in Algorithm 6.2. It starts from trying to find at least one file in the directory, if so, the function will inform the client that files are founded (Line 7). Then, it counts how many files can be transferred to the client, `send_counter` in Algorithm 6.2. Initially, we give the default value 4 to the `send_counter_threshold`. It is the upper bound of the number which

Algorithm 6.2 Pseudocode of the Function seek_file(process_id)

```
1. seek_file (process_id) {
2.   Initialization and definition of variables;
3.   Construct the path of files to be sent;;
4.   if(no file is founded)
5.     return 1;
6.   else{
7.     MPI_Send(file_founded, to Process_id);
8.     Acquire the files number: send_counter;
9.     MPI_Send(send_counter, to Process_id);
10.    do {
11.      If (not reach the send_counter_threshold)
12.        break;
13.      Construct the path_of_file;
14.      f_open(path_of_file);
15.      Acquire file_size;
16.      MPI_Send(file_name to Process_id);
17.      MPI_Send(file_size to Process_id);
18.      MPI_Send(file to Process_id);
19.      Delete the file has been sent;
20.    } while (exit another file to be sent);
21.  }
22.  return 0;
23. }
```

counts the number of files sent to the client every time. Since there are more than one clients

in the system and we cannot decided the quantity of SMT files, it is necessary to determine the number of files to be sent (sometimes it will be less than send_counter_threshold). If the number is less than the send_counter_threshold, the do while loop (Line 19) will ensure that the file sending procedure will finish within the threshold. Conversely, Line10-Line11 will guarantee that only threshold files will be sent to the client. Secondly, the quantity of files is sent to the client by MPI_Send() function from the server (Line 9). At every sending round (Line 10-Line 20), the file name, file size and the file itself are sent separately in sequence. We use blocked MPI communication mechanism to send files. It means that after the server invokes a MPI_Send(), it suspends until the client receives the message.

6.3.3 Design of Client

The client is implemented by MPI process numbered from 1 to n depending on user requirements. The main process of the client is requiring and receiving SMT files to local machines, then invoking an SMT solver to do the verification. After SMT solving, client will report to the server if a counterexample is found. At the beginning, the client require and receive four (or less than four) SMT files from server, if there exist such files on server. Then the client verifies these files. We call this process as a solving round. After a solving round is finished, the client will require new files until there is no file could be transferred. In a single solving run, we utilize OpenMP to accomplish parallel SMT solving on a PC to exploit the full computation resources. We suppose that the client in our implementation is equipped a CPU with four cores consequently we set 4 threads using OpenMP in each MPI process as default setting. The pseudocode of client n is shown in Algorithm 6.3.

The flow of a client is as follows: (1) the client sends file requirement to the server (Line 4), and then waits for the response of the server. If the server has no files to send, this client will receive a no_file signal and terminate, otherwise, the client will receive the number of files which will be sent later. (2) Line 10 to Line 17 are the file receiving procedure. The file

name, size and the file itself are received respectively. The procedure is a while loop, and will

Algorithm 6.3 Pseudocode of the Client

```
1. Process_n () {
2.   Initialization and definition of variables;
3.   do {
4.     MPI_Send (request files, to process_0);
5.     MPI_Recv (file_existence_condition from server);
6.     if (no file is founded)
7.       break;
8.     MPI_Recv(file_num from server);
9.     j = 0;
10.    while (j < file_num) {
11.      MPI_Recv(file_name from server);
12.      MPI_Recv(file_size from server);
13.      MPI_Recv(file from server);
14.      fwrite (file to local HDD);
15.      rename(file);
16.      Clear receiving buffers;
17.      j++;
18.    }
19.    invoke smt_solving();
20.    if (a counterexample is found)
21.      report solving result to the server;
22.    clean local files;
23.  } while (there still exists files to be received);
24.  MPI_Send(finish_signal to server);
25.  return 0;
26. }
```

be executed many loops which equal to the number of the files. The buffers to receive the data mentioned above will be cleared after a file is received successful. (3) After all the files are received successful, the client invokes the function `smt_solving()` to do the parallel verification SMT solving procedure. We will describe the parallel solving procedure in the following

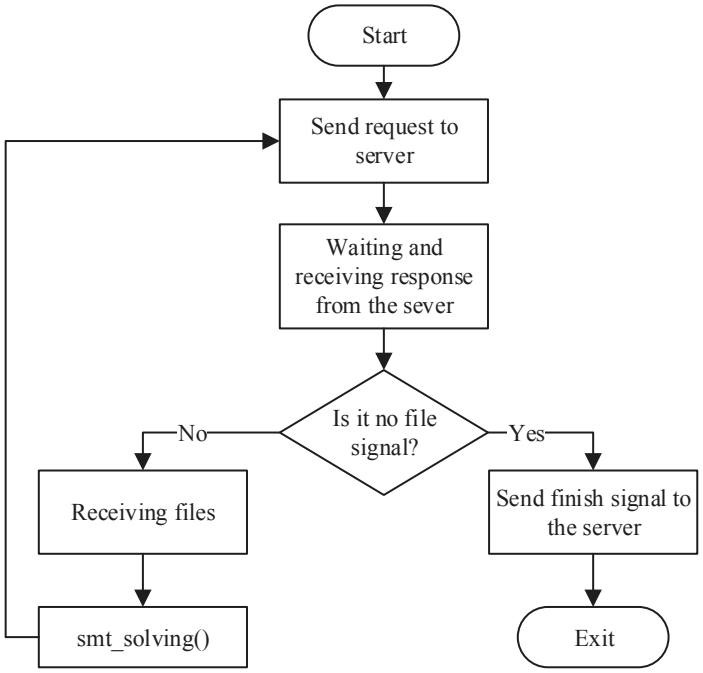


Figure. 6.4: Control Flow Graph of a Client.

paragraph. (4) After the SMT solving procedure is finished, the client will check the saved solving results. If a counterexample is found, it will be sent to the server. (5) The client will send a request to the server unless the server returns no file founded message to the client. (6) The client sends the `finish_signal` to the server in order to notice that it will be terminated soon. The whole control flow of the client is shown in Fig. 6.4.

The parallel SMT solving is accomplished by the function `smt_solving()` which utilize OpenMP to achieve parallel. The pseudocode is shown in Algorithm 6.4. OpenMP can create a specified number of threads which can communicate with each other by sharing variables (the variable is named as signal in Algorithm 6.4). Line 2 sets the number of thread to be created. In consideration of all our PCs have quad-core CPU, in our implementation, we set the value to four to exploit all the compute resources of a client. The `smt_solving()` function will be invoked after the file receiving procedure. This function will create 4 threads, every thread uses C function `system()` to execute an SMT solver to solve `SMT` files one by one. The

verification results are written to four files separately on a local hard disk driver.

6.3.4 Design of Communication Protocol

We describe the communication protocol of the system in this section. In our implementation, there are two types of communication. The first type is controlling signal transmission between the server and clients. It happens in the beginning and terminate stages. The other type is `SMT` file transmission from the server to clients. There is no communication among different clients.

1) *Control Signal Transmission* : In our program the control signal is an `int` array (denoted by `power[n]`) with two elements. The first element `power[0]` used to indicate the signal's meaning. The value and its corresponding meaning are shown in TABLE. 6.1. The second element is `power[1]` whose value is the source of the signal. The MPICH process ID is used to mark the source of the control signal. `power[0]` = 0 and `power[0]` = 1 can only be generated by a client. Furthermore, `power[0]` = 2 can only be dispatched by the server.

The distributed SMT solving system starts from a client sends the file request to the server, and the server searches its local directory to determine whether there exist enough files. Then it sends the result as a control signal to the requesting client. It can be `power[0]=0` (files exit) or `power[0]=2` (no file). It should be noted that the control signal has two `int` value so that the file request from the client is `power[]={0, process_id}` (value scope of `process_id` is 1 to n) and the files exit signal is `power[]={0,0}`. If the client receives the `power[]={0,0}`, it will turn to file receiving process. Otherwise it will send `power[]={1,process_id}` to server. The control signal protocol is shown in Fig. 6.5. The dashed line of step 2 and step 3 will occur when the server has no files to send. The filled line denoted step 2 and the dashed line denoted step 2 do not happen all at once. If the filled line step 2 happens, the client will turn to file receiving process immediately. After a client finishes the SMT solving process, it will request new files, if the server sends the signal in step 2 denoted by the dashed line, the client

will execute the step 3 to inform the server its termination.

Algorithm 6.4. Pseudocode of the Function smt_solving()

```
1. smt_solving () {
2.   omp_set_num_threads(m);
3.   omp_init_lock(&lock_signal);
4. #pragma omp parallel
5.   {
6.     switch (omp_get_threadnum()) {
7.     case 0:
8.     {
9.       Exploring all files belonging to thread_0 {
10.        Construct command to invoke Z3;
11.        system (command);
12.        goto next file;
13.      }
14.    }
15.      break;
16.    case 1:
17.    {
18.      Exploring all files belonging to thread_1 {
19.        ... // same as case 0
20.      }
21.    }
22.      break;
23.    ... // case 3 and case 4 are omitted
24.    default: break;
25.    }
26.  }
27.  return 0;
28. }
```

2) File Transmission The file transmission process is much easier. This process starts from the client receives the file exists signal `power[] = {0, 0}` sending by the server. The

Table. 6.1: Meanings of the Signal `power[0]`

`power[0]`	Meaning
0	Request files from a client or server has files to be sent
1	This client will be terminated
2	No file in the server

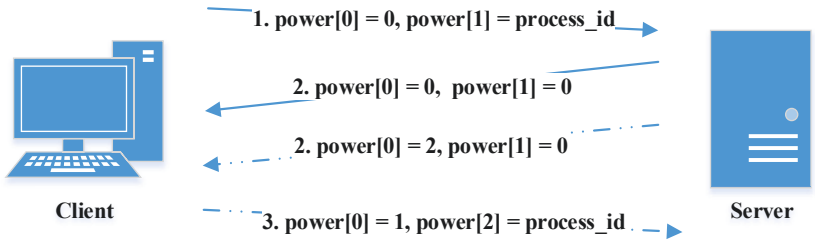


Figure. 6.5: Control Signal Transmission Protocol.

file transmission protocol is shown in Fig. 6.6. The MPI blocked communication mechanism is used, so we simplify the file receiving process. The confirmation after message receiving is eliminated. After step 2, the client knows that how many files will be sent. The sending and receiving processes both run in a loop. Step 3 to step 5, which is boxed by a dashed rectangle, will be repeated until all files have been sent. To make the 3 message sending and receiving in sequence, we use different message tags so that the client can distinguish them and receive them correctly. The file size is sent for establishing receiving buffer dynamically to save memory. The dashed line represented step 6 is an optional step. If a counterexample was found after the parallel SMT solving, the client will report the counterexample to the server. If not, the client will require new file from the server with entering control signal transmission process.

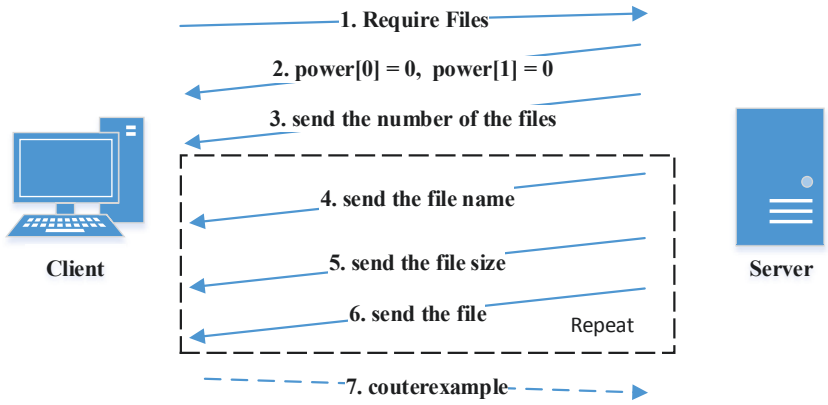


Figure. 6.6: File Transmission Protocol.

6.4 An Improved Implementation of Distributed SMT solving

The purpose of our distributed SMT solving is increasing the solving performance by leveraging computing resources of multiple computers as much as possible. In this section, we have discussed two main shortcomings in our previous work firstly. These shortcomings prevent our distributed implementation from further enhancing the performance of the distributed solving. We try to improve the utilization of computing resources in the following two aspects: *Fine-grained Dispatching Scheme* and *Communication Reduction*.

6.4.1 Shortcomings of the Prototype

Although our attempt gains significant improvement on solving performance, there are still some shortcomings of our previous work. The first is load balance, which is the core problems in the development of distributed model checker [89]. It means that our previous file distribution strategy is inefficient for scenarios where the hard problems or the combination of the easy and hard problems are considered. By *hardproblems*, we mean problems that consume, e.g., 600 seconds or more per file in our experiment. The easy problems often take less than 1 second per file. In our previous work, the files are sent to clients by group. That means 4 files as a group are sent to a client after one file request. The server chooses files randomly. In other words, a client may receive easy problems as well as hard problems. For

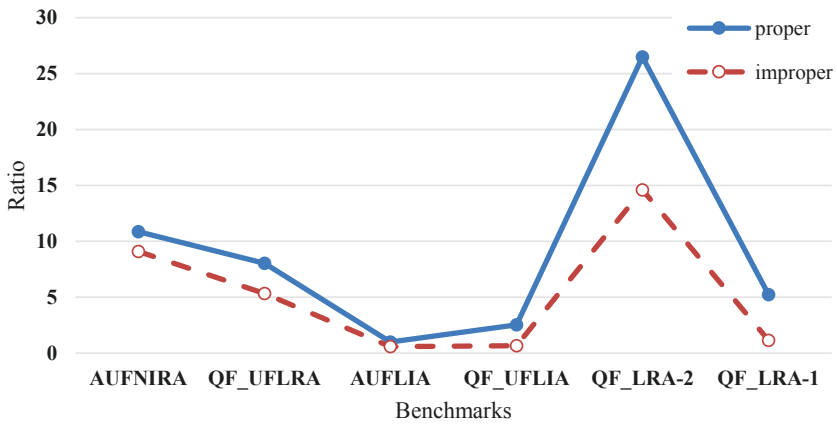


Figure. 6.7: Comparison of Proper/Improper Task Dispatch.

instance, there are four tasks named `Task1, Task2, Task3` and `Task4`. `Task4` is a hard problem and takes more time to solve. In a client, the 4 tasks are solved in parallel on different CPU cores. After `Task1` - `Task3` are finished, `Task4` is still being handled. In this case, 3 CPU cores are idle and no new tasks is assigned to them until `Task4` is finished. The best case is that all files have the same solving hardness. The more different the computing divergences are, the longer the total solving time is. A primitive experiment has been done to demonstrate this shortcoming. The result is shown in Figure 6.7. The two lines denote time-improvement ratio of distributed solving to serial solving. The blue solid line denotes the results where the workload is dispatched evenly to all clients (called *proper* case) while the red dash line denotes uneven dispatching (called *improper* case). It is clear that the proper dispatching gains higher improvement ratio than the improper case. It should be noted that this experiment is a trivial one just for demonstrating the importance of task dispatch. To summarize, an improper task dispatching can slow down the whole solving procedure.

The second shortcoming is that there are some unnecessary communication between the server and clients during file transfer. In our previously proposed file transferring protocol, which is shown in Figure 6.6, when we try to transfer one file to a client, a 3-time communication is needed (step 4 to step 6). Actually it is unnecessary to send two messages in step 3

Table. 6.2: Measurements of Easy Problems (in Sec.)

Benchmarks	Serial	1 Client	2 Clients	3 Clients
QF_UFLRA	191.10	52.36	23.81	17.41
AUFNIRA	31.30	30.64	10.55	6.65
AUFLIA	105.80	88.87	41.90	50.48
QF_UFLIA	114.52	72.58	31.65	25.84

and step 4. If we did so, we have to spend more time on establishing connections. In Table 6.2 (recall that all tasks in this table are easy ones that can be solved less than 1 second), when we increase the number of clients, the improvement are limited. One of the reasons is that the delays brought by establishing connections and the data transmission overwhelm the superiority gained by distributed computing. In addition, not only the file transferring stage but also other unnecessary communication between the server and clients could be reduced. In Figure 6.6, the array `power[]` is used to send controlling signals. The first element `power[0]` stores the signal's type. The second element `power[1]` is used to indicate the source of a message. The values and the meaning represented by the value are shown in Table 6.1.

6.4.2 A Fine-Grained Dispatching Scheme

The first considered aspect is changing file dispatching grain from coarse-grain to fine-grain. Our previous file dispatching scheme is coarse-grained. That means the minimum unit to assign workload is client which realized by one MPI process even 4 threads running in it. The client (process) sends request to the server and receives returned files. After it receives files from the server, the client dispatches files to different threads where the SMT solving is done in a parallel way. This procedure is shown in Figure 6.8. In this figure, the part boxed by a dashed rectangles denotes a client connected to the server. The whole procedure starts from sending file requirement from a client to the server for the first time. If there exits at

least one file on the server, they will be sent to the client. Then the control flow enters the parallel solving stage. After the parallel solving, all four threads need to synchronize with each other before a new require-receiving round. The synchronization shown in Figure 6.8 means that threads which have finished their tasks in the current round have to wait for other threads which are still in solving to finish their tasks. After the synchronization, the client will request new files again and the procedure will be repeated. The synchronization is needed because the file is dispatched to a client not a single thread. From the macroscopic view, the work load is dispatched to a client on process-level and the synchronization of threads will cause the shortcoming we discussed in the above section.

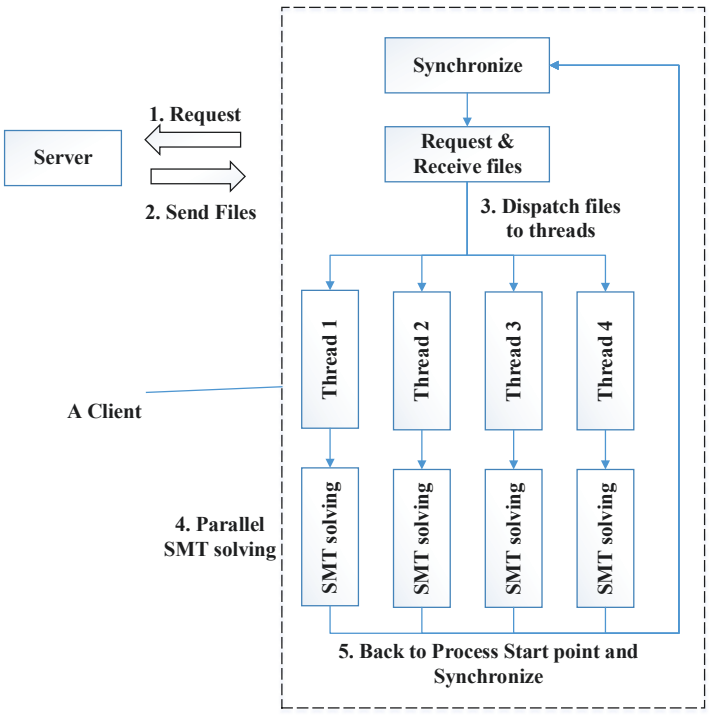


Figure. 6.8: Previous File Dispatching Scheme.

We change the dispatching scheme described above so that the synchronization between threads is removed after finishing one solving round. The improved scheme is shown in Figure 6.9. In our new scheme, the client starts from sending initial request to the server and receiving

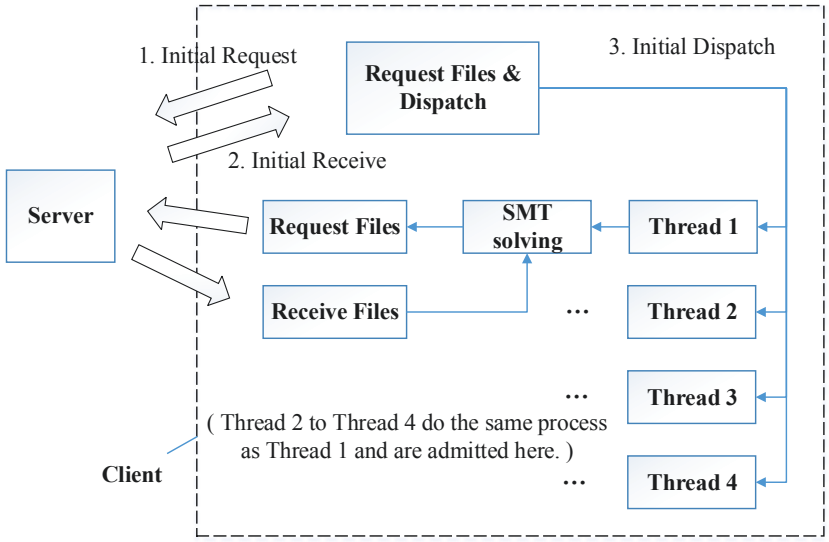


Figure. 6.9: New File Dispatching Scheme.

the first file set. It should be noted that this requesting-receiving round is executed only once. Then the received files are solved by 4 threads respectively in parallel. After that, 4 threads (in one client) will send file requests to the server separately when they need new files. The following receiving procedure is conducted by these threads also. If new files were received, threads will enter parallel SMT solving procedure again until a counterexample is founded or no files in the server. All four threads conduct the same procedures so that we admitted the detail of Thread 2 to Thread 3 in Figure 6.9. At this point, no synchronization is needed. Threads in a client are more independent than the previous scheme. The function of requiring and receiving files is implemented in thread level. Each thread can obtain new tasks from the server by itself. It is no longer necessary to wait for other threads to finish their solving tasks.

In our implementation with the new designed fine-grained dispatching scheme, the architecture is still C/S. The server runs in a loop to receive the messages sent by the clients and prepare files for them. So we only expand the length of the control message `power[]` without any other changes. A major change comes up on the client side so we present it here in Algorithm 6.5. The argument `process_id` is an unique `int` number to distinguish a process

Algorithm 6.5 Newly Designed Client Procedure

```
1. Process_n (int process_id) {
2.   Initialization and definition of variables;
3.   MPI_Send (request files, to process_0);
4.   MPI_Recv (file_existence_condition from server);
5.   if (no file is founded)
6.     return 0;
7.   j = 0;
8.   while (j < file_num) {
9.     MPI_Recv(file_information from server);
10.    MPI_Recv(file from server);
11.    fwrite (file to local HDD);
12.    rename(file);
13.    Clear receiving buffers;
14.    j++;
15.  }
16.  invoke parallel_solving(char *working_path);
17.  clean local files;
18.  MPI_Send(finish_signal to server);
19.  return 0;
20. }
```

which is running as a client. At the beginning, the client sends a request to the server (Line 3) and receives the file existence condition. The function `MPI_Send()` is a message sending function supplied by MPI [80]. It is a basic blocking message send operation. The routine returns only after the application buffer in the sending task is free for reuse. The function `MPI_Recv()`, which is also a MPI supplied function, receives a message and block until the requested data is available in the application buffer in the receiving task. The two functions must be used in as a pair. Otherwise, a dead block will happen. If file exists, an initial receiving and dispatching procedure will be done (Line 8 to Line 15). The initial dispatching is done by a client in our design because at the beginning, all CPU cores are idle, there is no need to consider the load balance problem at that time. The parallel SMT solving will take place by invoking the function `parallel_solving`() with the parameter `char *working_path` which

indicates the local path where the target files are saved in.

The parallel solving part is also changed. It is done inside each client. We use OpenMP [84] to create 4 threads to leverage computing capacity of clients as much as possible. As we mentioned above every thread in a client now has the ability to request files from the server if necessary. Meanwhile, the files sent by the server will be transferred to the thread to achieve our fine-grained dispatching. We show the procedure of function `parallel_solving`() in Algorithm 6.6. In each client, firstly, a thread resolves the files dispatched in the initial step of the client. Then it enters a `do {...} while`() loop to request and receive files until there is no file in the server. After one receiving round, which means that a thread has received a set of files from the server, the received files are solved by an SMT solver. The number of files in a set is a variable and its value is set to 2 by default. The argument `working_path_t0` is generated from the argument `working_path` to indicate its own working path. Thread 0 to Thread 3 are running in parallel and we omit the pseudo-code of thread 1 to thread 3 in Algorithm 6.6. If a counterexample is found in a solving round, the thread will report to the server. This activity of a counterexample finding and reporting is the same as our previous one so that we omit it in this algorithm.

In practical implementation, the new scheme might be less effective for the case which the solved problems are all easy problem or easy problem dominated (most of files are easy problems). In such a case, the solving time of a single file is short enough. In our previous scheme, the main time consuming is the synchronization of threads. Even if the previous scheme is used, the synchronization time is a short duration. It will not effect the total solving time using the synchronization or not. However, by using the new dispatching scheme, we expect to get better performance for solving hard problems or combined problems under the server's random file dispatching strategy.

Algorithm 6.6 Newly designed Function parallel_solving()

```
1. parallel_solving (char *working_path) {
2.   omp_set_num_threads(m);
3. #pragma omp parallel
4.   {
5.     switch (omp_get_threadnum()) {
6.     case 0:
7.     {
8.       Exploring all files in working_path {
9.         invoke Z3 to solve;
10.         goto next file;
11.       }
12.       do{
13.          Require and Receive files from the server;
14.          if (no file is founded)
15.            break;
16.          receive all files in this round;
17.          Exploring all files in working_path_t0 {
18.            invoke Z3 to solve;
19.            goto next file }
20.          delete solved files in working_path_t0;
21.       } while(server_has_files)
22.     }
23.       break;
24.    ... // Case 1 to case 3 are mostly like case 0 and omitted
here
25.     }
26.  }
27.   return 0;
28. }
```

6.4.3 Communication Reduction

The second aspect to improve solving efficiency is by reducing unnecessary communications between the server and clients. In Figure 6.6, we can emerge step 2 with step 3 firstly. We expand the control signal `power[]`, which is an array with two elements, to 3 elements. For instance, if the `power[0] = 0` (means there exists files on the server), `power[2]` will be the number of the files while `power[1]` denotes the source of the message. If not, `power[2]` will set to 0 and `power[0] = 2`. In the file sending stage, before sending file data, the server will inform the file names and sizes, which are used by the client to create a receiving buffer dynamically. We use a `struct`, which consists of the name and size of the file to be sent in the future, and the structure could be sent by using `MPI_send()` function once. The step 4 and step 5 in Figure 6.6 can be merged by using the new data structure.

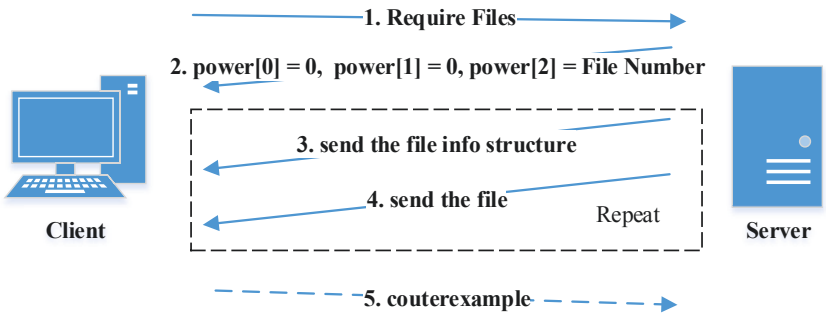


Figure. 6.10: New File Transferring Protocol.

The reformed protocol is shown in Figure 6.10. The new File transferring protocol needs one communication to send the control information and two communications before sending one single file, while the previous protocol needs two and three communications, respectively. The first merging brings an expansion of the array size from 2 to 3. In C language, one `int` element consumes 2 Byte memory. For a modern PC and Ethernet, 2 Byte is not an issue. In the second merging we use a `struct` to store the file name and size. Comparing with sending these information separately, it consumes more bandwidth for one time sending. Even so, the

increased bandwidth overhead is nothing to a 100M/1000M Ethernet.

$$T_{pre} = n * (\bar{s} + t_{name} + t_{size}) + (t_{control} + t_{number}) * n/number \tag{6.1}$$

$$T_{new} = n * (\bar{s} + t_{structure}) + t_{control} * n/number \tag{6.2}$$

However, not every case can gain positive effect on promoting solving performance. If the target problems are all hard problems the communication overhead is not obvious comparing with the solving time. But for easy problems, the situation is opposite. The solving time of a easy problem is much more shorter than establishing communication and transfer control messages. The constitution of previous distributed SMT solving time T_{pre} is shown in Formula (6.1). n denotes the total number of files to be solved, $\bar{s}$ is the average solving of a single file. t_{name} and t_{size} represent the time of establishing communication and sending file's name and size respectively. $t_{control} + t_{number}$ are consumption of sending control signal and the number of files. *number* denotes files which will be sent by the server over one requirement of a client. After the reduction, the time consumption constitution T_{new} is shown in Formula (6.2). In some cases, the average solving time $\bar{s}$ is no match for establishing the connection between the server and client. So reducing the time denoted by t_x can increase the performance significantly. t_x presents the time consummation of t_{name}, $t_{control}$ and t_{number}.

6.5 Discussion

The point of our work on parallel SMT solving is to make SMT solving distributed and ultimately to accelerate SMT-based BMC. In general, two procedures are involved for this purpose: (1) decomposing a whole BMC problem (formula) into sub-problems (sub-formulas), each of which is relatively smaller in size, easier to solve, and can be solved in parallel. In our previous work, we have done work in this aspect [46, 90]; (2) paralleling SMT-based BMC, as

described above in this paper. The classic SMT-based BMC encodes the system and properties into a single formula at specific bound. Restricted by the computation ability of a single PC, the efficiency of SMT-based BMC will decrease sharply. On the basis of decomposition, we can use the parallel SMT solving framework to solve the sub-problem parallel. Obviously, in this way, not only the capacity/scalability, but also the solving speed of model checking can be increased significantly. However, note that this is not to say our framework can increase BMC in all circumstances. Consider a situation where the `SMT` files in the server are all of small size. This means generally the files are easy to solve. Taking network latency into account, the time consumption here can be much higher than the SMT solving time. The final solving time of the parallel solving with multiple PCs may become longer than serial solving with a single PC.

Another issue we are going to discuss is about load balance. The server sends files to the client, and the client will arrange the work to different threads. For instance, there are four `SMT` files: $\{f_1, f_2, f_3, f_4\}$ and two threads T_1 and T_2. The solving time of the files f_1 to f_4 are 1s, 2s, 4s and 5s separately. If f_1 and f_2 are assigned to T_1 and the others are assigned to T_2, the solving time is 9s in total. But if f_1 and f_4 assigned to T_1 and the others are assigned to T_2, the solving time is 6s in total. The situation will be worse when the clients have different compute capacity. Considering the worst case that the most complex workload is dispatched to the lowest client, the total solving time depends on the lowest client's compute capacity. How to arrange the workload for different clients impacts directly on the final solving time. In our implementation, we dispatch the `SMT` files with a random strategy. We will investigate the load balance problem in our future work.

6.6 Summary

In this section, first the prototype of distributed SMT solving, which include the designs of server, client and the protocol, is described. Then after a discussion of its shortcomings,

we present our work on accelerating SMT solving. To tackle those shortcomings, we proposed the fine-grained dispatching scheme and communication reduction methods. A series of experiments, which is presented in Chapter 7, has been carried out to demonstrate the feasibility and efficiency of our improved techniques. Discussions and analysis are raised after the experiments.

Chapter 7

Implementation and Experiment

In this chapter, the implementation of distributed model checker is presented. A series of experiments has been done to evaluate our proposed methods.

7.1 Evaluation of Encoding Approach for HSTM Communicating by Message Passing

The encoding approach described in Chapter 3 has been implemented on a model checker named Grakabu2. The core components included in Garakabu2 are: a HSTM parser, a LTL formula parser, two encoders for both HSTM and LTL formulas, the SMT solver and a counterexample extractor and a simulator. The SMT solver is CVC3 [47] and CVC4 is used in the latest version of Garakabu2. The design with HSTM developed by ZIPC tool [60], can be read by Garakabu2 directly. The LTL properties to be checked are translated to logical formulas using CVC3's C++ API.

We demonstrate our experiments with the running example – Money-Exchange Machine (MEM), on a Windows PC with Intel Core i7 CPU and 24 GB memory. We present three kinds of properties (among others that we have checked), which are particularly interesting for HSTM designs. We use **G** and **F** to represent the LTL temporal operators *globally* and

eventually, respectively.

The first kind of properties is called *unreachability of invalid cells* (UIC). An invalid cell, in STM is represented as "×", implies that a specific event should never be dispatched at any given time. To justify such expectation, we express a UIC property with an invalid cell c of STM_i as a LTL property in the form $\mathbf{G}$ ($\neg$ ($actStatus_i$=$index(source(c))\wedge$ $event(c)$ = $event(HSTM_j.q)$)). In our implement, the unreachable cells are cell(0,2) and cell(0,3) of STM MainInterface. The properties UIC1 and UIC2 we checked correspond to the invalid cells(0,2) and cell(0,3) of STM MainInterface respectively.

The second kind of properties is called *static status constraints* (SSC), which demands certain correlation between status of different STMs. For example, the SSC between STM_i and STM_j, is in the form $\mathbf{G}$ ($actStatus_i = sa \Rightarrow actStatus_j = sb$). In this section, we check properties SSC1 and SSC2. SSC1 demands STM MainInterface is in actStatus $M\_Wait\_BILL\_TAKEN$ while STM ReturnController is in actStatus R_ACTIVE. SSC2 means that STM ReturnController is in actStatus R_IDLE and STM MainInterface is in M_ACTIVE at the same time.

The third kind is named as *functional correctness under weak fairness* (FCF) constrains. It states certain conditions are true infinitely often. We checked property FCF1, expressed as $\mathbf{GF}(xUserBillTake) \Rightarrow (\mathbf{GF}\ (BillOutputAmount > 0))$. $\mathbf{GF}(xUserBillTake)$ is the assumption of weak fairness constraint. This property implies that if a user takes the output bills infinitely often, and the user will obtain the bills eventually.

The last kind property is some simple safety properties (SimG). This is a property named invariant, namely the property must hold `true` globally. SimG1 has the form like $\mathbf{G}(dealCount \leqslant 2)$. SimG2 is $\mathbf{G}(Balance \geqslant 0)$.

The results of checking these seven properties are shown in Table 7.1, where time is total solving time (in seconds). The default bound is basically set to 25, except for UIC2, to find a counterexample of which, 30 is a needed search depth. Taking UIC2 as an example, we explain the reason of the occurrence of counterexamples. According to the HSTM design in

Table. 7.1: Experiments Results on the Original MEM

Property	Bound	Counterexample	Time(s)
UIC1	25	not found	103
UIC2	30	when step=30	340
SSC1	24	when step=24	74
SSC2	25	not found	73
FCF1	11	when step=11	3
SimG1	25	not found	57
SimG2	25	not found	118

Figure 3.1, when a user operates the MEM, actually the MainInterface and request to Exchange 10K bill. If the exchanger does not have enough balance, the MainInterface send the message *BillReady* to HSTM ReturnController by mistake (executes cell(1,2)). Due to this, Returner sends a message $UserBillObtained$ to MainInterface. Then the property UIC2 is violated (reaching the invalid cell(0,3)). Garakabu2 catches the counterexample (execution sequence) by observing the variables fl_k , where $1 \leqslant k \leqslant u$, since in each time step only one of fl_k has the value `true`.

Table. 7.2: Experiment Results on the Revised MEM

Property	Bound	Counterexample	Time(s)
UIC1	25	not found	100
UIC2	25	not found	109
SSC1	25	not found	75
SSC2	25	not found	92
FCF1	3	when step=3	1
SimG1	25	not found	57
SimG2	25	not found	129

We revise the HSTM design of MEM by removing the message sending operation when $BillOutputAmount \leqslant 0$. Then we checked the seven properties again. The results are shown in Table 7.2. This time no counterexample is found except FCF1, which is essentially an incorrect property for both original and revised MEMs.

Table. 7.3: Verification Results (in Sec.)

Problems				SAL	Garakabu2	
name	prop	bound	verdict		orig	hrid
cruise	p1 (S)	50	CX=10	4	7	1
	p2 (L)	50	NO CX	5	4	5
	p3 (L)	50	NO CX	5	4	5
vend1	p1 (S)	50	NO CX	10	30	1
	p2 (L)	50	CX=40	34	36	21
	p3 (L)	50	CX=40	17	30	20
vend2	p1 (S)	50	NO CX	8	153	1
	p2 (L)	50	NO CX	16	7	3
	p3 (L)	50	CX=30	9	57	3
exch1	p1 (S)	50	CX=50	32	74	1
	p2 (S)	50	NO CX	57	52	1
	p3 (L)	50	NO CX	TO	12	8
	p4 (L)	50	CX=50	TO	131	28
exch2	p1(S)	50	CX=50	41	89	1
	p2 (S)	50	NO CX	137	125	1
	p3 (L)	50	NO CX	TO	14	8
	p4 (L)	50	CX=50	494	158	45
exch3	p1 (S)	100	CX=90	TO	TO	1
	p2 (S)	100	NO CX	TO	TO	1
	p3 (L)	100	NO CX	TO	64	39
	p4 (L)	100	CX=90	TO	TO	258

`(S)`: Safety property; `(L)`: Liveness property;
`CX`: Counterexample; `CX=N`: A CX found at depth N;
`orig`: G2 with the basic BMC algorithm
`hrid`: G2 with Algorithm 4; `TO`: TimeOut (>= 1000 sec.)
Accumulative time from depth 10 (each iteration adds 10)

7.2 Evaluation of Incremental SESE with BCS and Divide & Conquer Method

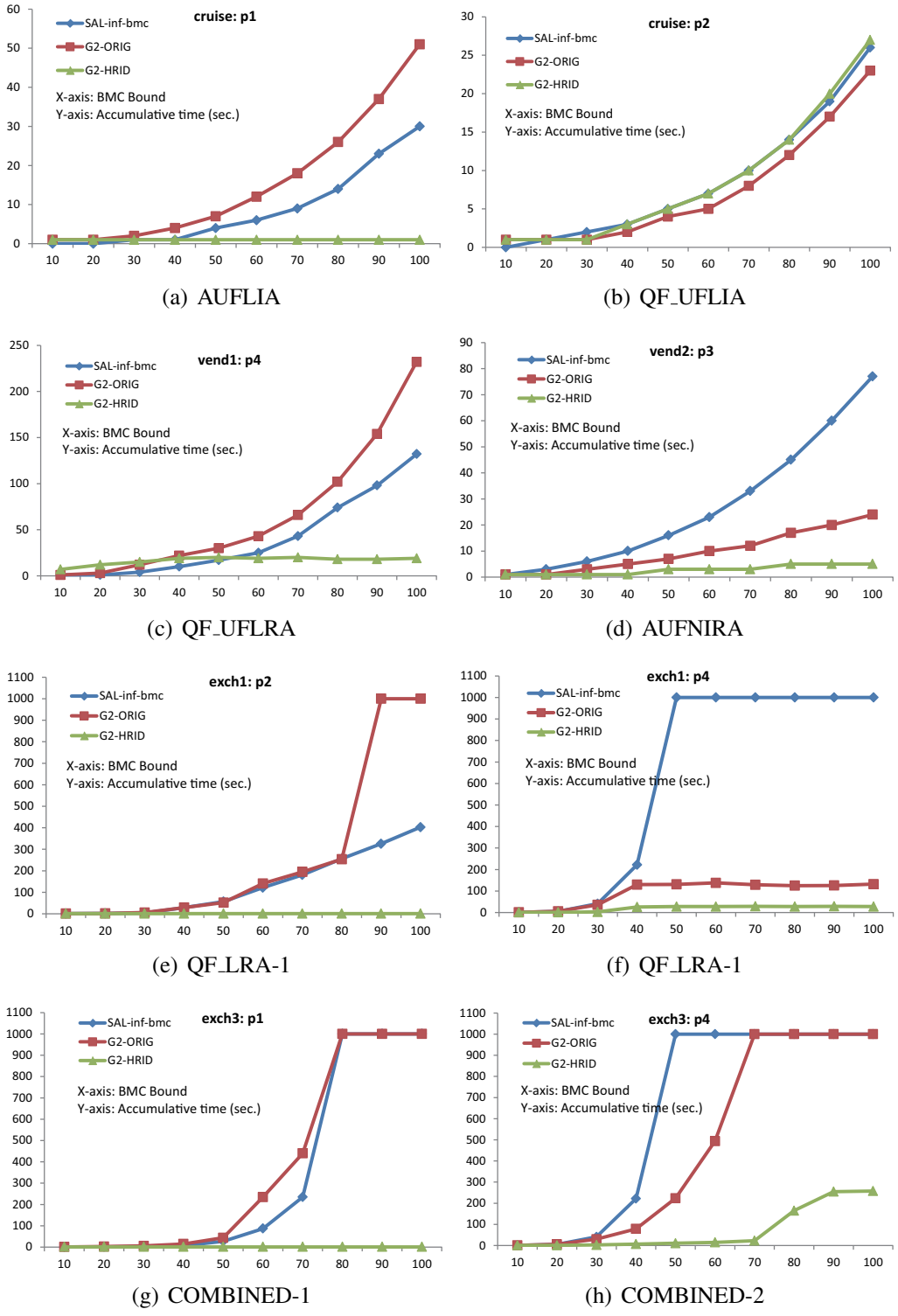
The hybrid BMC approach proposed in Algorithm 4 (in Chapter 5) has been implemented in a tool called Garakabu2 (G2 in short). G2 reads as input a software design developed using STM [15] (mentioned in Chapter 3, multiple STMs of a design can be formed in a hierarchical

structure) and an LTL property; encodes a model-checking problem into a propositional formula in SMT-LIB2 format [52] by following Algorithm 4; and asks its backend SMT solver CVC4 (version 1.2) [59] to solve the formula and extracts counterexamples if any. Message-Passing Interface (MPI) [80] is used to implement the multicore computation procedure in Algorithm 4.

We experimented with six HSTM designs, each with 3 to 4 safety or liveness properties, where the designs contain generally 20-25 variables and 17-25 parameterized transitions. Note that, during encoding, some additional variables and transitions (at each depth) are needed to represent the call/return functions in the hierarchical structure, and to generate counterexamples (refer to [63] for details). `cruise` is easy to solve with shallow counterexamples and `exch1-3` are relatively hard to solve with deeper counterexamples, where `vender1-2` belong to types in between. In all the six designs, STMs are affiliated with 2 parallel processes, and we set the BCS number as 1 (i.e., *c_bd* in Algorithm 4 is 1). The check starts from 10 and is incrementally increased by 10 until 50/100 or a counterexample is found. (i.e., *inc* in Algorithm 4 is 10). Totally, there are 210 BMC instances. The environment for the experiments is a Windows PC with Intel Core i7 CPU (virtually 8 cores) and 24 GB memory.

In the experiments, we compare the performance of our proposed approach (called `hybrid G2` below) with those of the state-of-the-art SAL infinite bounded model checker (SAL-inf-bmc) [91, 92], and the early version of G2 (called `original G2` below) [63] that implements the basic BMC method described in [10]. SAL-inf-bmc is chosen because, to our knowledge, it is the only state-of-the-art BMC model checker that could directly read as input and check models that use unbounded variables (or in other words, with infinite state space), and STM designs are typically such models. We manually translate the STM designs as general state transition machines into SAL's specification language.

In Table 7.3, we show the verification results for all the six designs and properties when bound is set to 50 or 100. The first column specifies the name of the problem, the second is



(a) AUFLIA (b) QF_UFLIA

(c) QF_UFLRA (d) AUFNIRA

(e) QF_LRA-1 (f) QF_LRA-1

(g) COMBINED-1 (h) COMBINED-2

Figure. 7.1: The Verification Results for Selected BMC Instances.

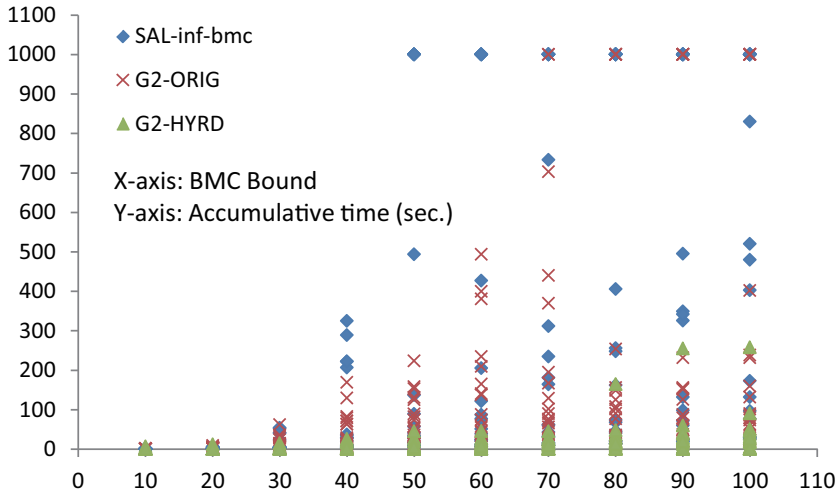
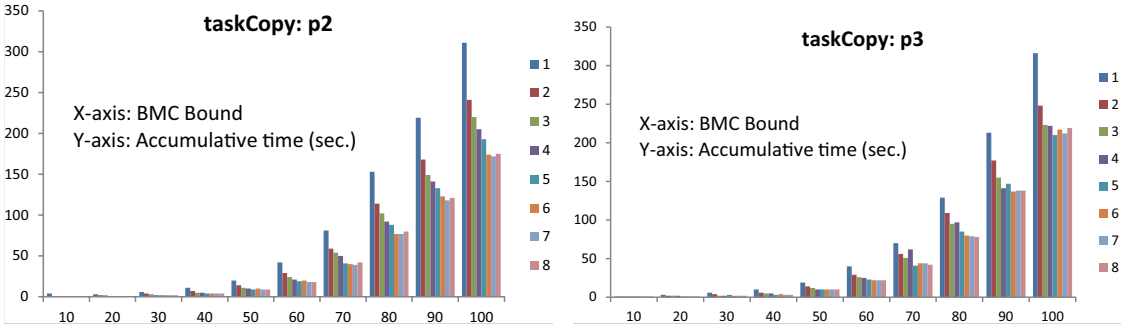


Figure. 7.2: Scatter Plots of the Verification Results for All BMC Instances.

the property name and type, the third is the bound set, the fourth is the verdict of the problem, and the next three columns show the checking time used. Note that since SAL-inf-bmc does not support gradually increasing the depth with 10 (the `-it` parameter increases 1 at each iteration), we manually record the accumulative time incrementally. From the results of `cruise` and `vend1-2`, we found that the performance difference for SAL-inf-bmc and hybrid G2 is not much for revealing shallow counterexample or easy-to-solve problems. However, from the results of `exch1-2`, which contain medium-sized (50 depth) counterexamples, hybrid G2 outperforms SAL-inf-bmc and original G2 by several orders of magnitude for safety properties. This is consistent with the observation made in [28] that bounded explicit-state checking, when it did finish, outperforms SAT-based BMC on safety properties. For liveness properties, hybrid G2 performs best. Such performance difference is exaggerated in `exch3`, which contains deep counterexamples and is hard-to-solve, for both safety and liveness properties. Furthermore, to show the tendency for performance difference, we illustrate in Figure 7.1 the verification results for some selected BMC instances. In Figure 7.2, the verification results for all the 210 BMC instances are illustrated.

Note that Yices [75] is used in SAL-inf-bmc as its backend SMT solver. Since Yices cannot solve formulas in SMT-LIB2 language, the format that G2 utilizes, we did not conduct

the experiments by changing G2's backend solver from CVC4 to Yices, which may give a more convincing and fair comparison between hybrid G2 and SAL-inf-bmc. However, Yices is a commonly-acknowledged efficient SMT solver, and we thus believe our results are to some extent reasonable. Or, at least, the comparison between the original and hybrid G2 can demonstrate the effectiveness of our proposed approach in this paper.



(a) (b)

Figure. 7.3: Comparison of Speedup Effects by Different Core Numbers.

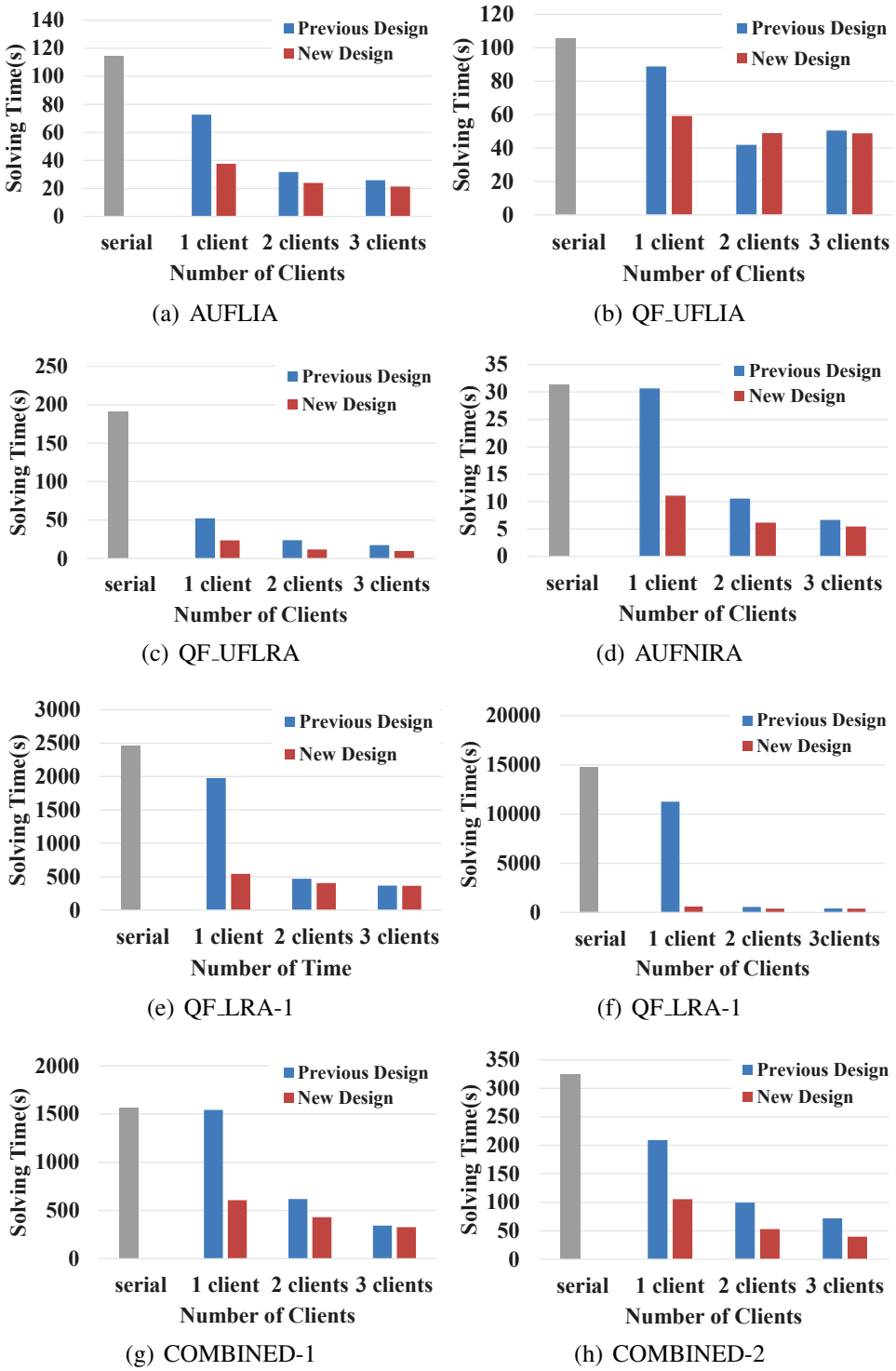
To further evaluate the speedup effects obtained from multicore computation, we show the experimental results of a large STM design taskCopy for two liveness properties. The example `taskCopy` used here is relatively hard to solve or with deeper (90 steps) counterexamples. The verification results of Hybrid G2 with core numbers ranging from 1 to 8 are illustrated in Figure 7.3.

7.3 Evaluation of Distributed SMT-Solving Architecture

To evaluate the efficiency of our improved distributed SMT solving implementation proposed in Chapter 6.6, we conduct a serial experiments on six groups of benchmarks downloaded from the Satisfiability Modulo Theories Library (SMT-LIB) [93] that conform to version 2.0 of the SMT-LIB format. The benchmarks are same as our previous work which are `AUFNIRA`, `QF_UFLRA`, `AUFLIA`, `QF_UFLIA`, `QF_LRA-1`, and `QF_LRA-2`. We will not go into

the details of those benchmarks, which are basically not relevant to the topic of this paper. Please refer to [93] for the meaning of those benchmarks. We deploy the program of the above described algorithms in four PCs running Windows 7 Enterprise Edition with MPICH 2.0 installed. One of the PCs is used as the server and the others are as clients. The hardware of the PCs are as follows: PC1 has a quad-core Intel Xeon CPU (2.66GHz) with 8GB RAM; PC2 and PC3 have a quad-core Intel i7 CPU (2.7GHz) with 8GB RAM; PC4 has an Intel Core2 Duo dual-core CPU (1.8GHz) with 2GB RAM. All the four PCs are connected to 100MB Ethernet. To evaluate the effective of our improvement, we use our prototype to solve those benchmarks and compare the results of our previous work. As we mentioned in the previous section, the four benchmarks, which are `AUFNIRA, QF_UFLRA, AUFLIA` and `QF_UFLIA`, are easy problems and the benchmarks `QF_LRA-1` and `QF_LRA-2` are hard problems. We design some new experiments beside the previous one. `Combined-1` is a combination of easy problems with hard problems and `Combined-2` is a set of 3000 easy problems. We use our previous algorithm and the new algorithms on solving these benchmarks respectively to prove the effectiveness of the latter.

Firstly, we conduct same experiments using the improved implementation for benchmarks which are used in our previous experiments [79]. We perform a serial solving experiment for each benchmarks using PC1. The SMT files are solved one after another in a serial way. Then the clients are connected to the server one by one and the same benchmarks are solved. PC4 is used as the server. Secondly we perform the same procedure mentioned above on new benchmarks `Combined-1` and `Combined-2`. The results are shown in Figure 7.4. The vertical rectangular marked as grey denotes the solving time of serial solving. The blue vertical rectangular presents results using previous algorithm. The red vertical rectangular denotes the solving time by using our new algorithm with two improvements. It is obvious that our improvements are useful on accelerating our previous distributed SMT solving implementation, especially for combined benchmarks. In Figure 7.4(e) and 7.4(f) when we add clients to 2 and



(a) AUFLIA (b) QF_UFLIA

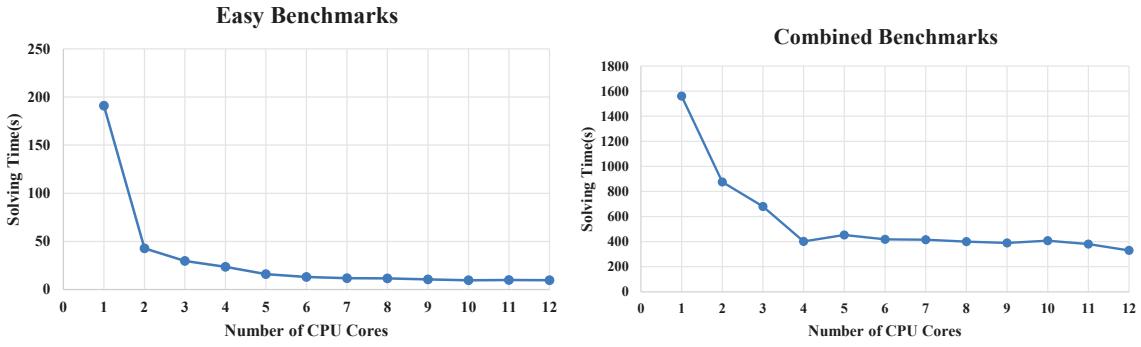
(c) QF_UFLRA (d) AUFNIRA

(e) QF_LRA-1 (f) QF_LRA-1

(g) COMBINED-1 (h) COMBINED-2

Figure. 7.4: The Distributed SMT solving Results.

3, the improvement seems elusive. The reason is that these two benchmarks contain one or more hard problems which take nearly 300 seconds to be solved. In other words, the limit of distributed solving time is about 300 seconds no matter how many clients are connected.



(a) (b)

Figure. 7.5: Comparison of Speedup Effects by Different Core Numbers in Distributed Environment.

Our new improved architecture gives us the ability to control the usage of CPU cores more precisely. This means that we can add threads involved in parallel solving procedure one by one, in an easier way than before. We conduct experiments with `Easy Benchmarks` and `Combined Benchmarks` to investigate the influence by increasing of CPU cores. We use PC4 as the server and other PC as clients. At first one client is connected, but only one CPU core is used, the second time the number of the CPU cores is increased to 2. Four cores will be used on each PC, after one PC reached the max value of used CPU cores, new client will be connected. We increase the CPU cores by one each time and repeat this procedure with two benchmarks respectively. The results are shown in Figure 7.5(a) and Figure 7.5(b). The results show that the curve of solving speed decreases sharply until the fourth CPU cores are involved. After that, the curve becomes flat. We have mentioned the possible reason in the paragraph above. For solving the `Combined Benchmarks`, the solving time of the hardest single problem is a limit of distributed SMT solving. For `Easy Benchmarks`, due to the CPU cores are on different PCs which are distributed in networks, the more CPU cores involved,

the more communication will take place. Considering the time consumption resolving a single easy problem and the overhead taken by network communication, the whole communication time consumption will be the predominant factor. In other words, if the solving target is determined, the whole solving time consumption could not be decreased always by simply adding more clients.

7.4 Summary

In this Chapter, three groups of experiments are presented to evaluate the correctness of the encoding approach for HSTMs which communicate by message passing, the effectiveness of the acceleration techniques proposed in previous chapters, respectively. The experimental results show that the efficiency of SMT-based BMC which implemented on Garakabu2 are increased significantly. The encoding approach are working correctly.

Chapter 8

Conclusion and Future Work

This chapter concludes this thesis and describes future directions.

8.1 Summary of the Thesis

This thesis focuses on the approaches of accelerating SMT-based bounded model checking. We proposed one formalization method which is used to model an embedded software design into SMT formulas. Then we proposed three algorithms to accelerate the SMT-based BMC and find the shortest counterexample as soon as possible if the properties are not satisfied. Finally, we present a distributed SMT solving framework with the purpose of accelerating SMT-based BMC further more. In the following, we summarize the contributions of this thesis, which are all implemented in Garakabu2, an SMT-based bounded model checker.

First of all, we provide formal verification support to HSTM designs. For this purpose, we formalize structures and behaviors of HSTM designs. Consequentially, we propose a symbolic encoding method, through which an HSTM design could be Bounded Model Checked using SMT solver. Furthermore, we have implemented the formal verification of software designs in HSTMs on a tool named Garakabu2. The results demonstrate that the encoding approach is correct. This work reveals the low efficiency shortcoming of our previous solving methods

which uses the classic BMC algorithm.

Second, the approaches to accelerating SMT-based bounded model checking are proposed. The approaches center around an unrolled bounded reachability tree (BRT) of a HSTM design which is built with stateless explicit state exploration (that is, states are not saved during exploration). Specifically, reachability of invalid cells (representing undesired states) of a HSTM design, which occurs within the bound concerned, could be discovered during construction of the BRT, and furthermore, if no such occurrence, the constructed BRT could be utilized to rule out unnecessary subformulas of a BMC instance and thus make the instance easier to solve. By such combination,we could enjoy the benefits of both explicit exploration and BMC with respect to speed as well as memory. In addition, we observe that much BMC verification time is consumed by iterative search (i.e., gradually increase the search depth till the concerned bound), which is necessary for finding the shortest counterexamples. We propose a binary search algorithm to avoid iteration but still guarantee to find the shortest counterexamples, if any. We have implemented these approaches in Garakabu2. The preliminary experiments show that verification could be accelerated substantially.

Third, bounded context switch (BCS) [16, 17], an under-approximation technique, is integrated into stateless explicit-state exploration (SESE). Such integration thus allows SESE to explore limited number of context switches of multiple parallel processes in the system so as to reduce the state space. Further, rather than encoding all legal execution paths, which are memorized during SESE, into a single (usually large) formula and inquiring its satisfiability of SMT solvers, we introduce heuristic predicates and use them to classify the paths into path clusters. Each path cluster can be considered as an independent BMC instance, which is usually smaller and easier to solve. Furthermore, multiple such BMC instances can be solved concurrently with multiple SMT solvers running on multicores. Since no information sharing is needed among these independent BMC instances, once a counterexample is found, the computation on all other cores can be safely terminated. In addition, rather than directly applying

SESE and BMC to a user-specified bound, we gradually deepen the checking depth from 0 with a fixed incremental number. Such iteration finishes until a counterexample is found or the bound is reached. In this way, counterexamples that are shorter than the user-specified bound can be revealed while avoiding expensive computation between the depths where the counterexample is found and the specified bound.

Fourth, a distributed SMT solving architecture is presented. The second and third contributions affect on the first stage of SMT-based BMC by reducing the reachable states of target system. The distributed SMT solving architecture affects on the second stage of SMT-based BMC. The whole BMC speed is enhanced further more by using this distributed solving architecture. The basic idea of this contribution is utilizing the computational capacity of multiple PCs. If the state space of the target system could be decomposed into smaller sub-state spaces which are encoded into formulas respectively, we may use solving these formulas distributively. The experiments shows that the solving efficiency can be increased substantially at the most cases.

8.2 Future Work Directions

We are actively developing methods to accelerate SMT-based BMC. In this section, we discuss some on-going and works in the future.

8.2.1 Tool Development

The first possible direction is that refine the formalization and encoding of message-passing HSTM designs. As could be observed, verification with the approach proposed in Section 3.2 of Chapter 3 is still not fast enough. One of the main reasons might be that our encoding of message queues and corresponding operations generate too many additional variables (especially for large queue size), and thus make the BMC problem hard to solve by SMT solvers.

And a further improvement on the usabilities of Garakabu2 for on-site software engineers is under consideration. One direction is to further ease the specification of LTL properties. We have been developing an auxiliary tool call LTL Editor, through which LTL properties could be draw based on graphs.

The second possible work in tool development is integrating the Distributed SMT solving framework on our bounded model checker Garakabu2. Up to now, Garakabu2 has the ability to perform bounded model checking parallel in one PC utilizing multi-CPU cores. With the algorithm we proposed in Chapter 5, we can divide the state-space of the system to several subspaces. These subspaces can be model checked in parallel even distributed. By implementing distributed model checking, we expect that the solving efficiency and the scalability of Garakabu2 could be increased significantly.

In addition to the techniques and methods proposed in Chapter 6, there are other methods that can be used for improving the efficiency of distributed SMT solving. The methods proposed in this paper are only for the client side. However, we can actually further improve the efficiency from the server side as well. In our current implementation, the number of requests from clients is four times higher than before, which may make the server get stuck. The server responds to the clients' requests in a serial way while parallel I/O can be used to give the server an ability to respond to various requests at the same time. Currently, the server randomly chooses files to send to the clients without considering the computation ability of different clients. Another possible idea is that the sever could use other optimized file choosing strategy, e.g., by the size of files, so as to avoid dispatch hard problems to weak clients. We will investigate those possibilities in the future.

8.2.2 Other Accelerating Techniques

The BRT-based approach may also suffer from saturation for large bounds. Thus, reducing the number of paths of BRT becomes one direct solution for this. We plan to combine

techniques for reducing paths, e.g., representatively partial-order reduction [48]. In addition, actually, the BRT built before carrying out BMC provides a platform for applying different explicit model checking techniques. We plan to follow some techniques from the most advanced explicit model checkers like SPIN [48] and PAT3 [94, 95]. HSTM designs are essentially concurrent systems with possibly exponential number of interleaving. The bounded context-switch [96] has shown its effectiveness to discover counterexamples for large concurrent systems. We plan to investigate this technique as well. Last, as mentioned in Section V, we plan to study [57] to further investigate other tight combination of explicit exploration and BMC (symbolic) techniques.

We plan to follow the abstraction techniques implemented in the state-of-the-art explicit model checkers like SPIN [48] and PAT3 [94] and integrate those techniques into our explicit state exploration procedure to further reduce the state space to be solved with BMC. Representatively, as an example, partial order reduction (POR) [9] is an effective technique for decreasing the number of interleaving sequences, and its combination with BCS has been discussed in [17]. Our explicit exploration is stateless but path information is remained. How to integrate POR into such a stateless exploration for verifying LTL properties is to be investigated. For the engineering aspect, we have found that much time is used for encoding large LTL properties at deeper depths. We are now optimizing our implementation of the encoding approach proposed in [71] and investigating alternative encoding approaches.

Acknowledgements

First and foremost I would like to express my deep and sincere gratitude to my supervisor Professor Akira Fukuda. I could not have completed this thesis without his invaluable support, guidance, encouragement and friendship over the three years of my research. I appreciate Prof. Fukuda when he helps me in my most difficult time in Japan. I benefit a lot from the experience that study with him.

Besides Professor Fukuda, I would like to thank the rest of my thesis committee: Professor Ryuzo Hasegawa and Professor Naoyasu Ubayashi, for their insightful comments, encouragement, and hard questions. It is not a easy task, reviewing my thesis, and I am grateful for their thoughtful comments.

I would also like to thank Prof. Weiqiang Kong for his objective advice and assistance. His effort helped me to considerably to turn this work into PhD dissertation. I benefit greatly from his scientific advice and knowledge and many insightful discussions and suggestions. Thanks to Ms. Otsuru, her supports and assistance give me the power of concentrating on my research and finish my study in Kyushu University.

I am truly grateful for the supports and advice of Professor Zhiguang Qin and Professor Shijie Zhou from University of Electronics Science and Technology of China. I would also like to thank my sponsors, China Scholarship Council (CSC), for their financial support.

I am indebted to the students in Fukuda Lab and my friends. Especially, thanks to Xinni Zhao, let me reacquaint myself. Great thanks to Mr. Ming Li, Mr. Bin Tong and Mr. Yichao

Xu. Thanks a lot for their help and encourage.

Last but not least, I would like to thank my parents for their unconditional love, understanding and support. Without their continuous encouragement, I would not be where I am today.

References

[1] J. Catsoulis, *Designing Embedded Hardware*. O'Reilly Media, 2005.

[2] B. Stroustrup, "Abstraction and the c++ machine model," in *Embedded Software and Systems*. Springer, 2005, pp. 1–13.

[3] G. Tassey, "The economic impacts of inadequate infrastructure for software testing," *National Institute of Standards and Technology, RTI Project*, vol. 7007, no. 011, 2002.

[4] N. Leveson and C. Turner, "An investigation of the therac-25 accidents," *Computer*, vol. 26, no. 7, pp. 18–41, July 1993.

[5] N. Leveson *et al.*, "Medical devices: The therac-25," *Appendix of: Safeware: System Safety and Computers*, 1995.

[6] G. J. Holzmann, "Landing a spacecraft on mars," *IEEE Software*, vol. 30, no. 2, pp. 83–86, 2013.

[7] G. A. Soffen and C. W. Snyder, "The first viking mission to mars," *Science*, vol. 193, pp. 759–766, 1976.

[8] L. Cordeiro, R. Barreto, R. Barcelos, M. Oliveira, V. Lucena, and P. Maciel, "Txm: An agile hw/sw development methodology for building medical devices," *SIGSOFT Softw. Eng. Notes*, vol. 32, no. 6, Nov. 2007.

[9] E. M. Clarke, O. Grumberg, and D. Peled, Eds., *Model Checking*. The MIT Press, 1999.

[10] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs," in *Proc. of the Workshop on Tools and Algorithms for the Constrction and Analysis of Systems (TACAS'99)*. Springer Berlin Heidelberg, 1999, pp. 193–207.

[11] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli, *Handbook of Satisfiability*, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, 2009, vol. 185.

[12] M. Davis and H. Putnam, "A computing procedure for quantification theory," *Journal of the ACM (JACM)*, vol. 7, no. 3, pp. 201–215, 1960.

[13] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Communications of the ACM*, vol. 5, no. 7, pp. 394–397, 1962.

[14] A. Armando, J. Mantovani, and L. Platania, "Bounded model checking of software using smt solvers instead of sat solvers," *International Journal on Software Tools for Technology Transfer*, vol. 11, no. 1, pp. 69–83, 2009.

[15] M. Watanabe, "Extended hierarchy state transition matrix design method version 2.0," Tokyo, Tech. Rep., Jun. 2006.

[16] S. Qadeer and J. Rehof, "Context-bounded model checking of concurrent software," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2005, pp. 93–107.

[17] G. Holzmann and M. Florian, "Model checking with bounded context switching," *Formal Aspects of Computing*, vol. 23, no. 3, pp. 365–389, 2011.

[18] L. Cordeiro and B. Fischer, "Verifying Multi-threaded Software using SMT-based Context-Bounded Model Checking," in *Proc. The 33rd International Conference on Software Engineering*. ACM, 2013, pp. 331–340.

[19] R. Popkin and A. Stroll, *Philosophy Made Simple*, ser. A made simple book. Doubleday, 1993.

[20] D. Kroening and O. Strichman, *Decision Procedures: An Algorithmic Point of View*, ser. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2008.

[21] M. Huth and M. Ryan, *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2004.

[22] L. De Moura and N. Bjørner, "Satisfiability modulo theories: introduction and applications," *Communications of the ACM*, vol. 54, no. 9, pp. 69–77, 2011.

[23] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[24] E. M. Clarke and E. A. Emerson, *Design and synthesis of synchronization skeletons using branching time temporal logic*. Springer, 1982.

[25] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L.-J. Hwang, "Symbolic model checking: 10^{20} states and beyond," *Information and Computation*, vol. 98, no. 2, pp. 142–170, 1992.

[26] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded Model Checking," *Advances in computers*, vol. 58, pp. 117–148, May 2003.

[27] "CVC4: the SMT Solver," 2014, URL: http://cvc4.cs.nyu.edu/web/ [accessed: 2014-01-18].

[28] N. Amla, R. Kurshan, K. L. McMillan, and R. Medel, *Experimental analysis of different techniques for bounded model checking*. Springer, 2003.

[29] G. J. Holzmann, *The SPIN Model Checker*. Addison-Wesley, 2004.

[30] "Spin Formal Verfication," 2014, URL: http://spinroot.com/spin/whatispin.html [accessed: 2014-01-18].

[31] G. J. Holzmann, "The model checker SPIN," *IEEE Trans Software Eng*, vol. 23, no. 5, pp. 279–295, Jan. 2002.

[32] "NuSMV homepage," 2014, URL: http://nusmv.fbk.eu [accessed: 2014-01-18].

[33] R. Cavada, A. Cimatti, C. A. Jochim, G. Keighren, E. Olivetti, M. Pistore, M. Roveri, and A. Tchaltsev, "Nusmv 2.5 user manual.[Available Online] http://nusmv. fbk. eu," *NuSMV/userman/index-v2. html*, pp. 1–140, Jun. 2011.

[34] N. Eén and N. Sörensson, "An extensible sat-solver," in *Theory and applications of satisfiability testing*. Springer, 2004, pp. 502–518.

[35] N. Een and N. Sörensson, "Minisat: A sat solver with conflict-clause minimization," *Sat*, vol. 5, 2005.

[36] Z. Fu, Y. Mahajan, and S. Malik, "New features of the sat04 versions of zchaff," *SAT Competition*, 2004.

[37] "Symbolic Analysis Labaratory," 2014, URL: http://sal.csl.sri.com/index.shtml [accessed: 2014-01-18].

[38] M. Ganai and G. Aarti, "Tunneling and Slicing: Towards Scalable BMC," in *Proc. DAC '08: the 45th Annual Design Automation Conference*, Apr. 2008, pp. 137–142.

[39] M. K. Ganai and W. Li, "Proc. d-tsr: Parallelizing smt-based bmc using tunnels over a distributed framework," in *Hardware and Software: Verification and Testing*. Springer, 2009, pp. 194–199.

[40] M. K. Ganai, A. Gupta, Z. Yang, and P. Ashar, "Efficient distributed SAT and SAT-based distributed Bounded Model Checking," *International Journal on Software Tools for Technology Transfer*, vol. 8, no. 4-5, pp. 387–396, Aug. 2006.

[41] H. Barros, S. Campos, M. Song, and L. Zarate, "Exploring clause symmetry in a distributed bounded model checking algorithm," in *Proc. Engineering of Computer-Based Systems*. IEEE, 2007, pp. 531–538.

[42] E. Abrahám, T. Schubert, B. Becker, M. Fränzle, and C. Herde, "Parallel SAT solving in bounded model checking," in *Proc. FMICS 2006 and PDMC 2006*. Springer, 2007, pp. 301–315.

[43] M. K. Ganai and A. Gupta, "Accelerating high-level bounded model checking," in *Proc. IEEE/ACM International Conference on Computer-aided Design*. ACM, 2006, pp. 794–801.

[44] J. Morse, L. Cordeiro, D. Nicole, and B. Fischer, *Context-bounded model checking of LTL properties for ANSI-C software*. Springer, 2011.

[45] S. K. Lahiri, S. Qadeer, Z. Rakamari, and Z. R. c, "Static and precise detection of concurrency errors in systems code using smt solvers," in *Proc. Computer Aided Verification (CAV'09)*, 2009, pp. 509–524.

[46] W. Kong, L. Liu, Y. Yamagata, K. Taguchi, H. Ohsaki, and A. Fukuda, "On Accelerating SMT-based Bounded Model Checking of HSTM Designs." in *Proc. 19th Asia-Pacific Software Engineering Conference*. IEEE, Jun. 2012, pp. 614–623.

[47] C. Barrett and C. Tinelli, "Cvc3," in *Proc. Computer Aided Verification (CAV)*. Springer, 2007, pp. 298–302.

[48] G. J. Holzmann, Ed., *The SPIN Model Checker: Primer and Reference Manual*. ADDISON WESLEY Publishing Company Incorporated, 2003, ISBN: 978-0-321-77371-5.

[49] "Z3 Homepage," 2014, URL: http://z3.codeplex.com [accessed: 2014-01-18].

[50] L. Cordeiro, B. Fischer, and J. Marques-Silva, "SMT-Based Bounded Model Checking for Embedded ANSI-C Software," *IEEE Transactions on Software Engineering*, vol. 38, no. 4, pp. 957–974, 2012.

[51] C. M. Wintersteiger, Y. Hamadi, and L. de Moura, "A concurrent portfolio approach to SMT solving," in *Proc. Computer Aided Verification (CAV)*. Springer, 2009, pp. 715–720.

[52] C. Barrett, A. Stump, and C. Tinelli, "The smt-lib standard: Version 2.0," in *Proc. The 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, 2010.

[53] S. Ranise and C. Tinelli, "The smt-lib standard: Version 1.2," *Department of Computer Science, The University of Iowa, Tech. Rep*, 2006.

[54] (2013) Cvc4: Tutorials. [Online]. Available: http://cvc4.cs.nyu.edu/wiki/Tutorials

[55] S. Barner, C. Eisner, Z. Glazberg, D. Kroening, and I. Rabinovitz, *ExpliSat: Guiding SAT-based software verification with explicit states*. Springer, 2007.

[56] S. Khurshid, C. S. Păsăreanu, and W. Visser, *Generalized symbolic execution for model checking and testing*. Springer, 2003.

[57] M. Ganai, C. Wang, and W. Li, "Efficient state space exploration: Interleaving stateless and state-based model checking," in *Proc. Computer-Aided Design (ICCAD), 2010 IEEE/ACM International Conference on*, Nov 2010, pp. 786–793.

[58] G. Holzmann, R. Joshi, and A. Groce, "Swarm verification techniques," *Software Engineering, IEEE Transactions on*, vol. 37, no. 6, pp. 845–857, Nov 2011.

[59] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "Cvc4," in *Proc. Computer Aided Verification (CAV'11)*. Springer, 2011, pp. 171–177.

[60] J. CATS Co., Ltd., "An smt-based approach to bounded model checking of designs in communicating state transition matrix." www.zipc.com, 2010.

[61] J. Dubrovin, "Checking bounded reachability in asynchronous systems by symbolic event tracing," in *Technical Report R14*. Helsinki University of Technology, 2010.

[62] R. Sebastiani, "Lazy satisfiability modulo theories," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 3, pp. 141–224, 2007.

[63] W. Kong, N. Katahira, M. Watanabe, T. Katayama, K. Hisazumi, and A. Fukuda, "Formal verification of software designs in hierarchical state transition matrix with smt-based bounded model checking," in *Proc. The 18th Asia Pacific Software Engineering Conference (APSEC'11)*, Dec 2011, pp. 81–88.

[64] W. Kong, N. Katahira, W. Qian, M. Watanabe, T. Katayama, and A. Fukuda, "An smt-based approach to bounded model checking of designs in communicating state transition matrix," in *Proc. The 11th International Conference on Computational Science and Its Applications (ICCSA'11)*. Springer, 2011, pp. 159–167.

[65] T. Jussila, J. Dubrovin, T. Junttila, T. Latvala, and I. Porres, "Model checking dynamic and hierarchical uml state machines," *Proc. MoDeV2a: Model Development, Validation and Verification*, pp. 94–110, 2006.

[66] J. Dubrovin and T. Junttila, "Symbolic model checking of hierarchical uml state machines," in *Proc. The 8th International Conference on Application of Concurrency to System Design (ACSD'08)*. IEEE, 2008, pp. 108–117.

[67] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "Nusmv 2: An opensource tool for symbolic model checking," in *Proc. Computer Aided Verification (CAV'02)*. Springer, 2002, pp. 359–364.

[68] T. Junttila and J. Dubrovin, "Encoding queues in satisfiability modulo theories based bounded model checking," in *Logic for Programming, Artificial Intelligence, and Reasoning*, ser. Lecture Notes in Computer Science, I. Cervesato, H. Veith, and A. Voronkov, Eds. Springer Berlin Heidelberg, 2008, vol. 5330, pp. 290–304.

[69] W. Kong, T. Shiraishi, N. Katahira, M. Watanabe, T. Katayama, and A. Fukuda, "An smt-based approach to bounded model checking of designs in state transition matrix," *IEICE Transactions*, vol. 94-D, no. 5, pp. 946–957, 2011.

[70] M. Veanes, N. Bjørner, and A. Raschke, "An smt approach to bounded reachability analysis of model programs," in *Formal Techniques for Networked and Distributed Systems–FORTE 2008*. Springer, 2008, pp. 53–68.

[71] T. Latvala, A. Biere, K. Heljanko, and T. Junttila, "Simple bounded ltl model checking," in *Proc. Formal Methods in Computer-Aided Design*. Springer, 2004, pp. 186–200.

[72] M. Watanabe, "Extended hierarchy state transition matrix design method - version 2.0," in *CATS Technical Report*. A Japanese Version is published as a book by Higashiginza Publisher, Tokyo, 1998.

[73] L. Liu, W. Kong, S. Zhou, Z. Qin, and A. Fukuda, "Formal verification of communicating hstm designs," in *Proc. IEEE The 12th International Conference on Computer and Information Technology (CIT'12)*. IEEE, 2012, pp. 383–390.

[74] Y. Yamagata, W. Kong, A. Fukuda, N. V. Tang, H. Ohsaki, and K. Taguchi, "Formal semantics of extended hierarchical state transition matrix by csp," *ACM SIGSOFT Software Engineering Notes*, vol. 37, no. 4, pp. 1–8, 2012.

[75] L. d. M. Bruno Dutertre. (2014) IEEEtran homepagethe yices smt solver. [Online]. Available: http://yices.csl.sri.com/tool-paper.pdf

[76] A. Bouajjani, S. Fratani, and S. Qadeer, "Context-bounded analysis of multithreaded programs with dynamic linked structures," in *Computer Aided Verification*. Springer Berlin Heidelberg, 2007, pp. 207–220.

[77] A. Udupa, A. Desai, and S. K. Rajamani, "Depth bounded explicit-state model checking," in *Proc. SPIN'11*, 2011, pp. 57–74.

[78] L. de Moura and N. Bjørner, *Satisfiability modulo theories: An appetizer*. Springer, 2009.

[79] L. Liu, W. Kong, and A. Fukuda, "Implementation and Experiments of a Distributed SMT Solving Environment," *International Journal on Computer Science and Engineering*, vol. 6, pp. 80–90, 2014, ISSN: 0975-3397.

[80] "The Message Passing Interface (MPI) Standard," 2014, URL: http://www.mcs.anl.gov/research/projects/mpi/ [accessed: 2014-06-25].

[81] "LAM/MPI Parallel Computing," 2014, URL: http://www.lam-mpi.org/ [accessed: 2014-06-25].

[82] "Open MPI: Open Source High Performance Computing," 2014, URL: http://www.open-mpi.org/ [accessed: 2014-06-25].

[83] "MPICH User's Guide (Version 3.0.4)," 2014, URL: http://www.mpich.org/static/downloads/3.0.4/mpich-3.0.4-userguide.pdf [accessed: 2014-06-25].

[84] "Open MPI: Open Source High Performance Computing," 2014, URL: http://openmp.org/wp/2013/12/tutorial-introduction-to-openmp/ [accessed: 2014-06-25].

[85] "OpenMP 4.0 Specifications Released," 2014, URL: http://openmp.org/wp/2013/07/openmp-40/ [accessed: 2014-06-25].

[86] "OpenMP Compilers," 2014, URL: http://openmp.org/wp/openmp-compilers/ [accessed: 2014-06-20].

[87] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating system concepts*. J. Wiley & Sons, 2009.

[88] "OpenMP Tutorial at Supercomputing 2008," 2013, URL: http://openmp.org/wp/2008/10/openmp-tutorial-at-supercomputing-2008/ [accessed: 2014-6-25].

[89] G. J. Holzmann and D. Bosnacki, "The Design of A Multi-core Extension of the SPIN Model Checker," *IEEE Trans on Software Engineering*, vol. 33, no. 10, pp. 659–674, 2007.

[90] W. Kong, L. Liu, T. Ando, H. Yatsu, K. Hisazumi, and A. Fukuda, "Harnessing smt-based bounded model checking through stateless explicit-state exploration," in *Proc. The 20th Asia-Pacific Software Engineering Conference (APSEC'13)*. IEEE CS, 2013, pp. 355–362.

[91] B. Dutertre, M. Sorea *et al.*, "Timed systems in sal," in *Computer Science Laboratory*, 2004.

[92] L. De Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari, "Sal 2," in *Proc. Computer Aided Verification (CAV'04)*. Springer, 2004, pp. 496–500.

[93] "SMT-LIB: The Satisfiability Modulo Theories Library," 2013, URL: http://www.smtlib.org/ [accessed: 2014-6-25].

[94] Y. Liu, J. Sun, and J. S. Dong, "Pat 3: An extensible architecture for building multi-domain model checkers," in *Proc. IEEE The 22nd International Symposium on Software Reliability Engineering (ISSRE'11)*, Nov 2011, pp. 190–199.

[95] Y. Liu, "Model checking concurrent and real-time systems : the pat approach," Ph.D. dissertation, Univ. of Singapore, May 2009. [Online]. Available: http://www.ntu.edu.sg/home/yangliu/thesis/thesis.pdf

[96] A. Lal and T. Reps, "Reducing concurrent analysis under a context bound to sequential analysis," *Formal Methods in System Design*, vol. 35, no. 1, pp. 73–97, 2009.

Appendix A

List of Publications

Journal Paper

1. Leyuan Liu, Weiqiang Kong, and Akira Fukuda. Implementation and Experiments of a Distributed SMT Solving Environment. International Journal on Computer Science and Engineering, vol. 6, No. 3, pp.80-90, 2014.

International Conference

1. Leyuan Liu, Weiqiang Kong, Shijie Zhou, Zhiguang Qin, and Akira Fukuda: Formal Verification of Communicating HSTM Designs. In: the 12th IEEE International conference on Computer and Information Technology (CIT 2012), IEEE CS, pp.383-390, 2012.
2. Weiqiang Kong, Leyuan Liu, Hirokazu Yatsu, and Akira Fukuda: A Combined Formal Analysis Methodology and Towards Its Application to HSTM Designs. In: the 4th International Conference on Advances in System Testing and Validation Lifecycle (VALID 2012), pp.99-106, 2012.
3. Weiqiang Kong, Leyuan Liu, Yoriyuki Yamagata, Kenji Taguchi, Hitoshi Ohsaki and Akira Fukuda: On Accelerating SMT-based Bounded Model Checking of HSTM Designs. In: the 19th Asia-Pacific Software Engineering Conference (APSEC 2012), IEEE CS, pp.614-623,

2012.

4. Weiqiang Kong, Leyuan Liu, Takahiro Ando, Hirokazu Yatsu, Kenji Hisazumi, and Akira Fukuda; Harnessing SMT-Based Bounded Model Checking Through Stateless Explicit-State Exploration, In: the 20th Asia-Pacific Software Engineering Conference (APSEC 2013), IEEE CS, pp.355-362, 2013.
5. Leyuan Liu, Weiqiang Kong, Takahiro Ando, Hirokazu Yatsu, and Akira Fukuda: A Survey of Acceleration Techniques for SMT-Based Bounded Model Checking, In: the 2013 International Conference on Computer Science and Applications (CSA 2013), IEEE, pp.554-559, 2013.
6. Leyuan Liu, Weiqiang Kong, Takahiro Ando, Hirokazu Yatsu, and Akira Fukuda. An Improvement on Acceleration of Distributed SMT Solving. In: the 6th International Conference on Future Computational Technologies and Applications (FUTURE COMPUTING 2014), pp.69-75, 2014.