B.Comp. Dissertation

Multi-Core Model Checking

By

Nguyen Chi Dat

Department of Computer Science

School of Computing

National University of Singapore

2011/12

B.Comp. Dissertation

Multi-Core Model Checking

By

Nguyen Chi Dat

Department of Computer Science

School of Computing

National University of Singapore

2011/12

Project No: H011830
Advisor: Assoc. Prof. Dong Jin Song and Dr. Liu Yang
Deliverables:
    Report: 1 Volume

Abstract

We address the Linear Temporal Logic (LTL) model checking problem for finite-state systems, which is often reduced to finding accepting cycles in a graph. Even though recent hardware developments bring a lot of potential to speed up the performance of existing algorithms by applying parallelism, SCC-based model checking with fairness constraints algorithm has not been much investigated. In this work, we propose a new version of this algorithm, adapted to shared memory, multi-core architectures. Experimental results show that our algorithm exhibits good speed up, especially when a system space contains many strongly connected components.

Subject Descriptors:
D.1.3 Concurrent Programming
D.2 Software Engineering
D.2.4 Software/Program Verification
D.4.1 Process Management
G.2.2 Graph Algorithms

Keywords:
Formal Verification, LTL Model Checking, PAT, Fairness, Shared Memory Architecture, Parallel Algorithm

Acknowledgement

At first, I would like to express my sincere gratitude to my advisors, Assoc. Prof. Dong Jin Song and Dr. Liu Yang whose encouragement, guidance and support from the initial to the final level enabled me to develop a deep understanding of the subject and also helped me stay on the track of doing research. I am grateful to my senior in the Software Engineering Lab, Mr. Khanh Truong, who introduced me to the area of model checking and helped me a lot during the project. Lastly, I offer my regards to my family and friends who supported me in any respect during the completion of the project.

List of Figures

Table of Contents

Chapter 1

Introduction

Model checking has become a very practical technique for automated formal verification. The purpose is to verify whether a given hardware or software system meets its specification. For the analysis of properties expressed in LTL, this problem is often reduced to checking the emptiness of a Büchi automaton defined as the product of the system and an automaton negating the formula to check. However, its applicability has suffered by the state explosion problem (i.e. the enormous increase in the size of the state space).

As the availability of multicore chips has been brought up by the rapid development in hardware industry, the use of parallel algorithms to combat the state explosion problem gained interest in recent years (Barnat, Brim, & Chaloupka, 2003) (Barnat, Brim, & Ročkai, 2007). Two main classes of algorithms for LTL model checking are Nested Depth-First Search (NDFS) (Courcoubetis, Vardi, Wolper, & Yannakakis, 1992) and SCC-based algorithms based on the Tarjain strongly connected components (SCC) detection (Tarjan, 1972). Both NDFS and SCC-based algorithms cannot trivially be adapted to a multi-core setting, since they strongly rely on depth-first search, which is inherently sequential (Reif, 1985).

Another problem we want to investigate in this project is fairness constraints, which are used to restrict the behavior of the system. Without fairness, verification often produces unrealistic loops where one process or event is infinitely ignored by the scheduler. Those counterexamples should be ruled out and resources should be utilized to find real bugs. However, combining model checking with fairness is expensive and non-trivial.

Even though parallel algorithms for LTL model checking have been researched extensively recently, not much work has been done for the Tarjan based algorithm, especially with fairness constraints. In this work, we propose an on-the-fly parallel model checking algorithm with fairness, which is based on Swarm Verification and Liu Yang et al. previous work (Liu, Sun, & Dong, 2009). At the same time, we develop the Process Analysis Toolkit (PAT) with various state-of-art model checking algorithms in the multicore architecture with shared memory for further research purpose.

The rest of the report is structured as follows. Chapter 2 introduces the basis of LTL model checking together with a family of different fairness notions and reviews some related algorithms for LTL model checking. In the next two chapters, we describe in details several algorithms that I implemented in PAT during this project. The algorithms are divided into two classes: NDFS and Tarjan SCC, which are presented in chapter 3 and chapter 4 respectively. Especially, the last section of chapter 4 details our proposed parallel algorithm and gives its analysis. Chapter 5 shows some experimental results to compare all the implemented algorithms and demonstrate the effectiveness of the parallel algorithm. Chapter 6 concludes the work and gives some perspectives for future work.

Chapter 2

Background

In order to facilitate the understanding of our work, we begin with a brief background on LTL model checking and different fairness constraints based on it.

2.1 The LTL Model Checking Problem

In this work, we will model the actions of processes in terms of states and transitions, which are captured in the definition of a Labeled Transition Systems (LTS).

Definition 1 (LTS). *A Labeled Transition System $\mathcal{L}$ is a tuple (S, s_0, E, T) where S is a set of system configurations/states, $s_0 \in S$ is the initial system state, E is the set of all events in the model, and $T \subseteq S \times E \times S$ is the set of transitions.*

Given two states $s, s' \in S$ and event $e \in E$, we write $s \xrightarrow{e} s'$ to denote a transition from s to s' with event e, and we call e the engaged event of the transition. An infinite execution of $\mathcal{L}$ is a infinite sequence $\langle s_0, e_0, s_1, e_1, ..., e_i, s_i, ...\rangle$ where $s_i \xrightarrow{e_i} s_{i+1}$ for all $i \geq 0$. The set of enabled event at s is $enabledEvt(s) = \{e \in E \mid \exists s' \in S, s \xrightarrow{e} s'\}$. If the system has multiple processes running in parallel, define $enabledProc(s)$ to be the set of enabled processes which can make a move from the system state s. Given a transition $s \xrightarrow{e} s'$, we denote $engagedProc(s, e, s')$ to be the set of the processes which have made some progress during the transition.

To be able to explain the model checking procedure, we formally define Büchi automaton as follows.

Definition 2 (Büchi automaton). *A Büchi automaton is a tuple* $\mathcal{B} = (B,\ b_0,\ \sum,\ L,\ F)$*, where* B *is a set of Büchi states,* $b_0 \in B$ *is the initial state,* $\sum$ *is an alphabet,* $L \subseteq B \times \sum$ *is a nondeterministic transition function, and* $F \subseteq B$ *is a set of accepting states.*

A run of $\mathcal{B}$ is a infinite sequence $\langle b_0, b_1, ...\rangle$ where $b_i \in B$ and there exists $a_i \in \sum$ such that $b_{i+1} \in L(b_i, a_i)$ for all $i \geq 0$. A run $\sigma = \langle b_0, b_1, ...\rangle$ is accepted if and only if at least one state from set F appears infinitely often in σ.

Assume we are given a LTS $\mathcal{L} = (S, s_0, E, T)$ and a property f of $\mathcal{L}$ expressed in LTL. Model checking is to search for an execution of $\mathcal{L}$ which fails f. In automata based model checking approach, the negation of f is converted into a Büchi automaton $\mathcal{B} = (B,\ b_0,\ \sum,\ L,\ F)$. Then, the intersection of LTS $\mathcal{L}$ and Büchi automaton $\mathcal{B}$ is computed by taking their automata product with certain restricted transition relations. Any infinite run accepted by this intersection product of $\mathcal{L}$ and $\mathcal{B}$ now corresponds to a run of $\mathcal{L}$ where $\neg f$ is satisfied. By simple argument, the automata has infinite run if and only if it contains a loop that has at least one accepting state. Verification is now reduced to the problem of finding accepting cycles in a graph. For the detailed algorithms to translate LTL formula to Büchi automaton and construct the product, interested readers may refer to (Holzmann, 1999) and (Vardi & Wolper, 1986).

2.2 Fairness Definitions

Given the basic concept of LTL model checking, we further look at some definitions relating to fairness.

Fairness is a concept that is used in multithreaded/multiprocess programming environment. It often refers to a fair scheduling of CPU time to threads/processes or the relative speed of the processors in distributed systems. Many recent self stabilizing distributed algorithms are designed to function only under fairness (Angluin, Aspnes, Fischer, & Jiang, 2008), (Angluin, Fischer, & Jiang, 2006). In order to verify those algorithms, model checking techniques must take the respective fairness into account. In the following, we review a variety of fairness notions from (Sun, Liu, Dong, & Pang, 2009).

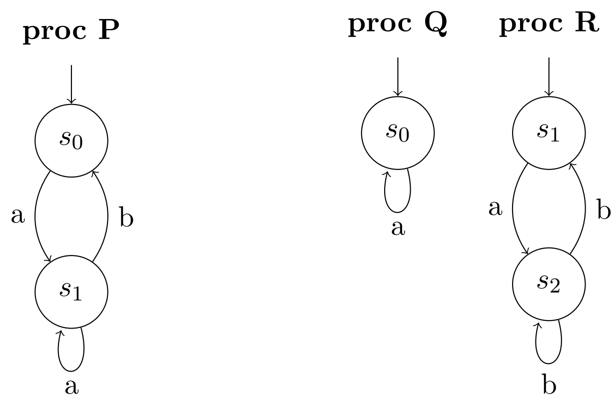
Definition 3 (Event-level weak fairness)**.** *Let $E = \langle s_0, e_0, s_1, e_1, ...\rangle$ be an execution. E satisfies event-level weak fairness, if and only if for every event e, if e eventually becomes enabled forever in E, then $e_i = e$ for infinitely many i, i.e., $\Box\Diamond$ e is enabled $\Rightarrow \Diamond\Box$ e is engaged .*

Event-level weak fairness (EWF) basically restricts that if there is a point in the run from where event e is always enabled, it must not be infinitely ignored. Now we look at a similar version of fairness, which is applied for process.

Definition 4 (Process-level weak fairness)**.** *Let $E = \langle s_0, e_0, s_1, e_1, ...\rangle$ be an execution. E satisfies process-level weak fairness, if and only if for every process p, if p eventually becomes enabled forever in E, then $p \in engagedPro(s_i, e_i, s_{i+1})$ for infinitely many i, i.e., $\Box\Diamond$ p is enabled $\Rightarrow \Diamond\Box$ p is engaged .*

Process-level weak fairness (PWF) states that if there is a point in the run from where process p can always make progress, it must be engaged infinitely often. PWF may be seen as a restriction that does not allow one process to be infinitely faster than other processes. Peterson's algorithm for mutual exclusion is one of the well known algorithms that requires at least PWF to function correctly (Sun et al., 2009).

Now we look at some examples to demonstrate more about these two fairness constraints.



(a) (b)

Figure 2.1: Event-level weak fairness and Process-level weak fairness

Consider the property $\Box\Diamond a$. From Figure 2.1a, we can see that event a is enabled forever. Thus, under EWF, the property is true. However, under PWF, the property is not necessarily satisfied for every run. Process P is enabled forever, so it will be engaged infinitely often. But, it can choose event b forever, so event a is never engaged, which means this is a counter example for the property under PWF. The LTS in 2.1b is different. Under PWF the property is true, because process P is enabled forever, hence it must be engaged infinitely often, and it can only choose event a. Generally, PWF is a weaker fairness constraint compared to EWF. Under EWF, PWF can be achieved by labeling all events in a process with the same name.

Definition 5 (Event-level strong fairness)**.** *Let $E = \langle s_0, e_0, s_1, e_1, ...\rangle$ be an execution. E satisfies event-level strong fairness, if and only if for every event e, if e is infinitely often enabled, then $e = e_i$ for infinitely many i, i.e. $\Box\Diamond e$ is enabled $\Rightarrow$ $\Box\Diamond e$ is engaged.*

Event-level strong fairness (ESF) is a stronger fairness constraint compared to EWF and PWF. It has several other names in different papers: *strong fairness* (Lamport, 2000), *strong local fairness* (Fischer & Jiang, 2006), *compassion* (Pnueli & Sa'ar, 2007). Under this fairness assumption, if event e is enabled infinitely often in a run, it must be engaged infinitely often.

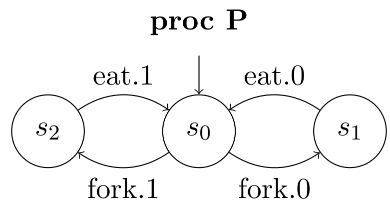


Figure 2.2: Event-level strong fairness

Given the LTS in Figure 2.2, and consider the property $\Box\Diamond eat.1$. Because $eat.1$ and $fork.1$ is not always enabled (when the system is in state s_1), under EWF, the system is allowed to take the branch $fork.0$, then $eat.0$, and traverse the loop forever. However, under PWF, cause $fork.1$ is infinitely often enabled (when the system is in state s_0), $fork.1$ will be engaged infinitely often. Because of that, $eat.1$ will also be engaged infinitely often.

Definition 6 (Process-level strong fairness)**.** *Let $E = \langle s_0, e_0, s_1, e_1, ...\rangle$ be an execution. E satisfies process-level strong fairness, if and only if for every process p, if p is infinitely often enabled, then $p \in engagedPro(s_i, e_i, s_{i+1})$ for infinitely many i, i.e. $\Box\Diamond p$ is enabled $\Rightarrow$ $\Box\Diamond p$ is engaged..*

Process-level strong fairness (PSF) is quite similar to ESF, but applies for process-level. It states that if in a run, a process p is enabled infinitely often, it will eventually be engaged.

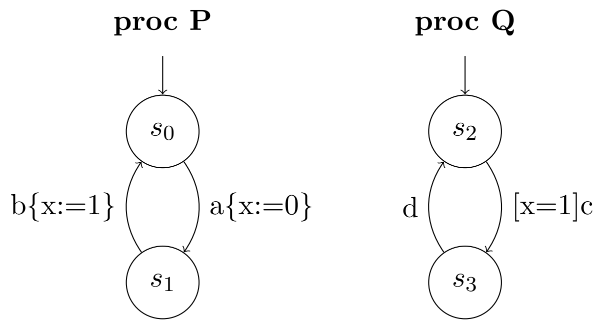


Figure 2.3: Process-level strong fairness

In the example in Figure 2.3, consider the property $\Box\Diamond c$. Under PWF assumption, as event c is not always enabled (it has a guarded condition $x = 1$), the property is not true. However, it is true under PSF. The reason is that event b is infinitely often enabled, which makes event c infinitely often enabled. By the definition of PSF, c is infinitely often engaged, i.e. $\Box\Diamond c$ is true.

Definition 7 (Strong global fairness)**.** *Let $E = \langle s_0, e_0, s_1, e_1, ...\rangle$ be an execution. E satisfies strong global fairness, if and only if for every s, e, s' such that $s \xrightarrow{e} s'$, if $s = s_i$ for infinitely many i, then $s_i = s$ and $e_i = e$ and $s_{i+1} = s'$ for infinitely many i, i.e. $\Box\Diamond(s, e, s')$ is enabled $\Rightarrow \Box\Diamond(s, e, s')$ is engaged.*

Different from previous notions of fairness, *Strong global fairness (SGF)* deals with the fairness of both events and states. It can be proven that SGF is stronger than both ESF and EPF.

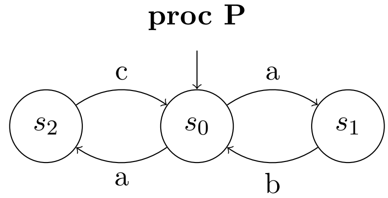


Figure 2.4: Process-level strong fairness

Consider the LTS from Figure 2.4 and the property $\Box\Diamond b$. Under SGF, both of the transitions with event a: $s_0 \xrightarrow{a} s_1$ and $s_0 \xrightarrow{a} s_2$ must be taken infinitely often. Thus, the transition $s_1 \xrightarrow{a} s_0$ must also be taken infinitely often, which means the property is true. However, under ESF or ESP, the system may run forever in the loop $s_0 \xrightarrow{a} s_2 \xrightarrow{c} s_0$ and event b will never be engaged. Therefore, the property is only true under SGF.

2.3 Related work

This section reviews different existing works on parallel model checking as well as model checking with fairness.

In the past, the difficulty of model checking lies on on enormous size of the system. In order to verify whether a system specification adheres to a given temporal property, the system needs to store the entire state space in memory, which consists of about $10^10 - 10^11$ states in a real systems. Recent hardware developments, such as the 64-bit technologies along with improvement in memory reduction techniques, has contributed to harnessing formal verification memory limitations. Recent observations support that the problem we face now is "time explosion" rather than a lack of memory (Barnat et al., 2007). Parallel algorithms seem to be a feasible approach to solve the "time explosion" problem and have been rigorously explored in the past five years.

One of most used parallel algorithms is the Maximal Accepting Predecessor algorithm (MAP) (Brim, Černá, Moravec, & Šimša, 2004). It relies on Bread First Search Techniques and is designed for distributed memory architecture. It computes the map function mapping each state s to the greatest accepting state that is backward reachable from s. Another algorithm, One-Way-Catch-Them-Young, uses the idea to repeatedly remove states from the graph that cannot be in the accepting cycle (Cerná & Pelánek, 2003). Combining MAP and One-Way-Catch-Them-Young, Barnat et al. proposed on-the-fly One-Way-Catch-Them-Young which has the advantages of both algorithms.

Two works (Barnat, Brim, Ceka, & Lamr, 2009) and (Brim et al., 2004) have drawn our interest particularly at the beginning of the project. Those works are some of the first that make use of NVIDIA GPU cards with CUDA technology to deal with LTL model checking problem, and we wanted to utilize the GPU platform to make a multicore Tarjan SCC algorithm. However, we found out the the nature of their approach is very different with ours. We concluded that it is not feasible to implement the Tarjan algorithm on CUDA and we decided to focus on Depth First Search based algorithms on shared memory architecture.

The first parallel algorithm designed for shared memory architecture was the dual core NDFS based on the observation that the blue and the red DFS can be performed independently and is implemented in SPIN (Holzmann & Bosnacki, 2007). The linear complexity of NDFS is kept, though it is only applicable to two cores.

Two recent works of Laarman et al. (Laarman, Langerak, van de, Weber, & Wijs, 2011) and Evangelista et al. (Evangelista, Petrucci, & Youcef, 2011) have greatly gained our interest. They proposed for the first time multi-core NDFS algorithms that can scale beyond two threads, while keeping the same worst case time complexity. Their algorithms make use of the idea from swarm verification, which is primarily aimed at settings with distributed memory. In swarm verification, each processor performs a DFS with a unique ordering of successor states. Using this, each worker explores different parts of the graph, and bugs may be found in a much shorter time compared to sequential verification. However, in the absence of bugs, the graph will be explored N times, where N is the number of workers. Using the advantage of swarm verification with extra synchronization among workers, two versions of parallel Nested DFS has been presented. In their algorithm, Laarman et al. make use of the shared red states, whereas in the other algorithm, Evangelista et al. the shared blue states are used to synchronize and fasten the graph search. More details of Laarman et al. algorithm will be discussed in the subsequent chapters.

The practical applications of model checking with fairness have been discussed extensively. Despite the fact that fairness constraints are crucial for designing distributed algorithms, existing model checking algorithms with fairness are inefficient. One possible approach for model checking under fairness is to reformulate the property so that the fairness becomes its premise. However, as the size of the Büchi automaton is exponential to the size of the property, this approach does not scale well with large formulas, whereas a typical system may have multiple fairness constraints. SPIN, the most well known model checker, can only support weak fairness for population protocols (Pang, Luo, & Deng, 2008). Protocols relying on stronger fairness are beyond the capacity of SPIN even for small networks.

In the next two chapters, we look into details of several algorithms which are implemented into PAT during this project. All of the algorithms are based on Depth First Search and work in shared memory architecture.

Chapter 3

Nested Depth First Search

In this chapter, we introduce one of the two best known enumerative sequential algorithms based on fair-cycle detection: Nested DFS.

3.1 Sequential Nested DFS

Nested DFS was the first linear time algorithm to detect accepting cycles, and was initially introduced by Courcoubetis et al. in (Courcoubetis et al., 1992). Although there are many versions of the Nested DFS algorithms with various modifications and improvements, we decide to choose the algorithm from (Schwoon, 2005), as it exibits very good run-time performance experimentally.

The basic idea of Nest DFS is as follows. The algorithm consists of two basic depth-first search procedures. Initially, all the states are colored white. At first, NDFS(s_0) initiates a DFS from state s_0, which is called the *Blue* DFS, since all the explored states are colored blue. This DFS is to detect all accepting states that are reachable from the initial state. At line 25, if the *Blue* DFS backtracks over an accepting state s, it performs another DFS, called the *Red* DFS, to identify whether this state is reachable from itself. We call s the *seed* of the corresponding *Red* DFS. If s is reachable from itself, then an accepting cycle is found, and the NDFS procedure exits. Otherwise, for each blue successor, the *Red* DFS is called on line 12. The algorithm continues until it find an accepting cycle (which we call a counterexample) or the whole graph is traversed.

```
procedure NDFS(s0)
    BLUE_DFS(s0);
    report no cycle;

procedure RED_DFS(s)
    for s' in successor(s) do
        if s'.color = cyan then
            report cycle and exit;
        else if s'.color = blue then
            s'.color := red;
            RED_DFS(s');

procedure BLUE_DFS(s)
    s.color = cyan;
    for s' in successor(s) do
        if s'.color = cyan ∧ (s ∈ A ∨ t ∈ A) then
            report cycle and exit;
        else if s'.color = white then
            BLUE_DFS(s');
    if s ∈ A then
        RED_DFS(s);
        s.color := red;
    else
        s.color := blue;
```

Figure 3.1: The Nested NDFS algorithm

On top of the basis of the classic Nested DFS, we implemented several improvements which has been suggested in the past.

To detect a counterexample in the original algorithm, the *Red* DFS needs to traverse until it find the accepting state where the *Red* DFS is initiated. A modification from (Holzmann, Peled, & Yannakakis, 1996) suggests that as soon as the the *Red* DFS initiated at s finds a state t that is inside the call stack of of the *Blue* DFS, we can report the accepting run, because s is obviously reachable from t . To identify which states are inside the call stack of the *Blue* DFS in constant time, one additional bit per state is used. Each state now is encoded with two bits, and is assigned with one of four colours:

- *white* : All states are coloured *white* at the beginning to mark unvisited state.

- *cyan* : A state whose blue search has not terminated, i.e. a state in the call stack of the *Blue* DFS.

- *blue* : A state whose blue search has terminated, but has not been reached by a red search.

- *red* : A state that is already visited by the red search.

With the two-bit colour encoding, the *seed* remains cyan during the red search and is colored red if no accepting run is found. Thus the need for a *seed* variable is eliminated. The counterexample can be obtained using the call stacks of the *Blue* DFS and *Red* DFS at the time when the cycle is reported.

Another improvement from (Gastin, Moro, & Zeitoun, 2004) suggests that the *Blue* DFS can detect an accepting cycle if it finds an edge from an accepting state to a state in the call stack or from a state to an accepting state in the call stack (line 19) . With this improvement, a cycle may be detected without entering the red search.

An extension called *allred* (Gaiser & Schwoon, 2009) is also considered. The basic idea is that red state cannot be part of any counterexample; therefore a state that has only red successors cannot be either. The idea can be applied by incorporating an additional check in the *Blue* DFS, if all successors of a state s are red, then s can be coloured red as well. This may avoid certain invocations of the red search. However, the computational effort required to check for *allred* is comparable with the effort expended in the red search: one additional check for each successor state. Therefore, we do not use this extension in our sequential algorithm. However, in the next section, this extension proves to be very useful for the parallel algorithm.

The time complexity of Nested DFS is linear to the size of the automaton, since each reachable state is visited at most twice, one by the *Blue* DFS, and one by the *Red* DFS. The algorithm is correct due to the fact that the *Red* DFS is initiated according to the post-order of the accepting states imposed by the *Blue* DFS. Thus, a red state will not be re-visited by another *Red* DFS later. More detailed proof of the soundness and complexity of the algorithm can be referred in (Courcoubetis et al., 1992).

3.2 Multicore Nested DFS with Shared Red States

Because of the inherently sequential property of Depth First Search, the tasks of parallelizing or scaling up to multi-core for Nested DFS and Tarjan algorithms are non trivial. Laarman proposed the first multi-core on-the-fly LTL model checking algorithm which is linear-time in the size of the input graph, and has a potential speedup greater than two (Laarman et al., 2011).

The algorithm makes use of the idea from swarm verification, which is primarily aimed at settings with distributed memory. Swarm verification uses embarrassingly parallel techniques, where each individual worker operates fully independently, i.e. without communicating with the other workers. In swarm verification, each worker performs a DFS with unique ordering of successor states. By this setting, each worker explores different parts of the graph, and bugs may be found in a much shorter time compared to sequential verification. However, in the absence of bugs, the graph will be explored N times, where N is the number of workers, since the workers are unaware of each other's results. In (Laarman, van de Pol, & Weber, 2010), a shared lockless hash table is proposed to store all the states and proved to scale well with this purpose.

The details of the algorithm are shown at Figure 3.2. We denote $successor_i^b(s)$ ($successor_i^r(s)$) to be the permutation of successors used in the *Blue (Red)* DFS by worker i. The key idea of this algorithm is to share the information about red states in the backtrack of the *Red* DFS. A *pink* colour is introduced to replace the local *red* colour in the sequential algorithm, representing the nodes which are current processed by the *Red* DFS. The *red* colour now becomes global and every worker can access this information. Because of this global *red* colour, one additional bit is needed for each state. A state is globally coloured *red* after the *Red* DFS makes sure that the state is not in any accepting cycle. The shared red states will be ignored by both the *Blue* and *Red* DFS of all workers, thus pruning the total search spaces. In line 18 and 19, the total number of workers that initiate the *Red* DFS, *s.count* is used as a synchronization mechanism. This enforces that multiple workers calling *Red* DFS from the same accepting states will have to finish simultaneously. The purpose for this synchronization is illustrated in Figure 3.3.

procedure MC_NDFS(s_0, N)
 for $i = 1$ to N **do**
  BLUE_DFS(s_0, i);
 report no cycle;

procedure RED_DFS(s, i)
 $s.color[i] := pink$;
 for s' in $successor_i^r(s)$ **do**
  if $s'.color[i] = cyan$ **then**
   report cycle and exit;
  else if $s'.color[i] \neq pink \wedge \neg s'.red$ **then**
   RED_DFS(s', i);
 if $s \in A$ **then**
  $s.count := s.count - 1$;
  await $s.count = 0$;
 $s.red := true$;
procedure BLUE_DFS(s, i)
 $allred := true$;
 $s.color[i] = cyan$;
 for s' in $successor_i^b(s)$ **do**
  if $s'.color[i] = cyan \wedge (s \in A \vee t \in A)$ **then**
   report cycle and exit;
  else if $s'.color[i] = white \wedge \neg s'.red$ **then**
   BLUE_DFS(s', i);
  if $\neg s'.red$ **then**
   $allred := false$;
 if $allred$ **then**
  $s.red = true$;
 else if $s \in A$ **then**
  $s.count := s.count + 1$;
  RED_DFS(s, i);
 $s.color[i] := blue$;

Figure 3.2: The Multi-core NDFS Algorithm with Shared Red States

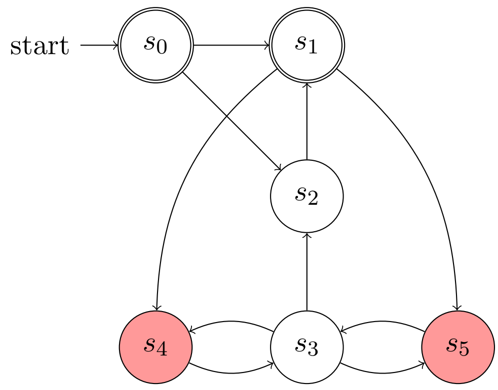


Figure 3.3: Illustration for **await** statement

We will show that without the await statement, there are some cases that the algorithm will miss out accepting cycles. A worker 1 can have a *Blue* DFS that explores state s_0, s_1, s_4, s_3, s_5, backtracks from s_5, explored s_2 and then backtracks to the accepting state s_1. Worker 1 then performs a *Red* DFS from s_1 which visits s_4, s_3 and stops at s_5. A second worker, worker 2, now starts and performs a similar*Blue* DFS but has different exploring order: $s_0, s_1, s_5, s_3, s_4, s_2$ and then backtracks to s_1. Its *Red* DFS starts from s_1 and visits s_5, s_3, s_4, and colours s_4 *red* because it can not traverse further. Worker 1 resumes, and, colours s_5 red as well. Note that, in this scenario, the accepting cycle can still be found by any of the two workers visiting s_3, s_2 and then s_1. However, the problem arises when there is a third worker starts a *Blue* DFS traversing s_0, s_2, s_1. Then it performs a *Red* DFS at s_1 and colours s_1 red because it cannot proceed further as s_5 is globally *red*. No accepting cycle is found in this situation. The **await** can prevent this problem, by stopping worker 3 at b_1, does not allow it to colour s_1 as red. Thus, either worker 1 or 2 can detect the accepting cycle.

For each state, the local colour $white, cyan, blue, pink$ can be encoded using two bits similar to the sequential algorithm. With the additional global red colour, the status of each state can be stored in three bits. The early cycle detection by *Blue* DFS extension can still be applied in this algorithm (line 27 and 28). Since the parallel workload of the algorithm depends entirely on the proportion of the state graph that can be marked red, *allred* extension can be used to improve the performance (line 32, 33, 36, 37).

The correctness of the algorithm is based on the fact that it will never miss all reachable accepting cycles. Its time complexity is linear in the size of the input graph, and it acts on-the-fly. However, in the worst case, each worker may have to traverse the whole graph. For detailed proof of correctness, readers may refer to (Laarman et al., 2011). Experiments show that this algorithm has very good speed up with models with accepting cycles, and performs reasonably well with models without accepting cycles.

Evangelista's algorithm also uses the same idea from swarm verification. Two algorithms seem to be complementary, since one shares the red states and the other shares the blue states. Also, instead of using synchronization, Evangelista's approach speculatively continues parallel execution and calls a sequential repair procedure in the case of dangerous situations. We choose to implement Laarman et. al algorithm because it has been reported to have better performance experimentally.

Chapter 4

Tarjan SCC Algorithm

In the first two sections, we presents two existing SCC-based model checking algorithms under fairness constraints which are implemented in PAT, one sequential (Sun et al., 2009), and one parallel algorithm (Liu et al., 2009). And in the last section, we propose a new parallel SCC-based algorithm based on the idea of swarm verificaion and shared memory.

4.1 Sequential SCC-based algorithm under Fairness

Fairness and model checking with fairness have attracted much theoretical interests for decades, and their practical implications in system/software design and verification have been discussed extensively (Giannakopoulou, Magee, & Kramer, 1999 lamport). However, existing model checkers are shown to be ineffective with respect to fairness (Liu, 2010). In the two main families of model checking algorithms, Nested DFS is shown to work efficiently under no fairness. However, it is not suitable for verification under fairness because whether an execution is fair depends on the whole path instead of one state (Holzmann, 2003). Recently, Jun Sun et al. propose a unified on-the-fly SCC-based model checking algorithm which handles a variety of fairness including process-level weak/strong fairness, event-level weak/strong fairness, strong global fairness, etc. Before going to the details, we look at the key idea of all SCC- based algorith: the original Tarjan's algorithm from (Tarjan, 1972).

4.1.1 Original Tarjan's Algorithm

Tarjan's Algorithm is a graph theory algorithm for finding strongly connected components in an input graph. The algorithm performs a DFS from the root state and visits all states which have not been explored. The states are placed on a stack in the order of the DFS traversal. When the search backtracks to any state, the state is taken out from the stack and checked whether it is the head of a strongly connected component (the first state in the strongly connected component which is visited during the DFS.

To determine whether a state is the head of a strongly connected component, each state contains two integers: *index* and *lowlink*. The *index* value of state s is the number of states visited before s in the DFS. The *lowlink* value of state s is equal to the smallest index of some node reachable from s, and always less than the *index* of s, or equal to the *index* of s if no other state is reachable from s. The details of the algorithm are presented in Figure 4.1. Because the state spaces may contain millions of states, and using the recursive version of the Tarjan's algorithm will most likely to cause StackOverFlow problem, we implemented the iterative version of Tarjan's algorithm in PAT.

The variable *done* in line 8, 12, 13 is to keep track whether the DFS finishes traversing all of the neighbours of the current state. After traversing all of the neighbours w of state v, the lowlink value of v can be calculated by applying the following formula for each of the neighbour.

$$v.lowlink = \begin{cases} min\{v.lowlink, w.lowlink\} & \text{if } w.index > v.index \\ min\{v.lowlink, w.index\} & \text{otherwise} \end{cases}$$

Variable *scc_queue* is to store the states in the processing SCC. After finding the head of the SCC, all of those states will be pushed into *scc*, and the SCC is reported. Aside from *index* and *lowlink*, each state contains one more additional bit, scc_found, which is used to check whether a state in is a found SCC.

```
procedure TARJAN(s0)
    Stack S := ∅; Stack scc_queue := ∅; index := 0;
    S.push(s0);
    while S ≠ ∅ do
        v := S.peek();
        if v is un-visited then
            index := index + 1; v.index := index;
        done := true;
        for w in successor(v) do
            if w is un-visited then
                S.push(w);done := false; break;
        if done then
            S.pop(); v.lowlink = v.index;
            for w in successor(v) do
                if ¬w.scc_found  then
                    if w.index > v.index then
                        v.lowlink := min{v.lowlink, w.lowlink};
                    else
                        v.lowlink := min{v.lowlink, w.index};
            if v.lowlink = v.index then
                scc := ∅; scc.push(v); v.scc_found = true;
                repeat
                    k = scc_queue.pop();
                    scc.push(k); k.scc_found = true;
                until scc_queue.Peek().index <= v.index
                report scc;
            else
                scc_queue.push(v) ;
```

Figure 4.1: The Tarjan's Algorithm

4.1.2 Model checking with Fairness

First, we introduce important definitions and lemmas that the algorithm from (Sun et al., 2009) is based on.

Given a LTS $\mathcal{L} = (S,\ s_0,\ \sum,\ L)$ and a property f expressed in LTL. Model checking under fairness is to search for an infinite execution which is accepting to the Büchi automaton and at the same time satisfies the fairness constraints.

We write $\mathcal{L} \models_{EWF} f$, $\mathcal{L} \models_{PWF} f$, $\mathcal{L} \models_{ESF} f$ and $\mathcal{L} \models_{PSF} f$ to mean that $\mathcal{L}$ satisfies f under event-level weak fairness, process-level weak fairness, event-level strong fairness, and process-level strong fairness respectively. We take the product of $\mathcal{L}$ and the negation Büchi automaton $\mathcal{B}^{\neg f}$. Let $R^i_j = \langle (s_0, b_0), e_0, ..., (s_i, b_i), e_i, ..., (s_j, b_j), e_j, (s_{j+1}, b_{j+1})\rangle$, where s_i is a state of $\mathcal{L}$, b_i is a state of $\mathcal{B}^{\neg f}$, $s_i = s_{j+1}$ and $b_i = b_{j+1}$. We define the following sets:

$$
\begin{aligned}
alwaysEvt(R^i_j) &= \{e \mid \forall k : \{i, ..., j\}, e \in enabledEvt(s_k)\} \\
alwaysProc(R^i_j) &= \{p \mid \forall k : \{i, ..., j\}, e \in enabledProc(s_k)\} \\
onceEvt(R^i_j) &= \{e \mid \exists k : \{i, ..., j\}, e \in enabledEvt(s_k)\} \\
onceProc(R^i_j) &= \{p \mid \exists k : \{i, ..., j\}, e \in enabledProc(s_k)\} \\
onceStep(R^i_j) &= \{(s, e, s') \mid \exists k : \{i, ..., j\}, s = s_k \wedge s \xrightarrow{e} s'\} \\
engagedStep(R^i_j) &= \{(s, e, s') \mid \exists k : \{i, ..., j-1\}, s = s_k \wedge e = e_k \wedge s' = s_{k+1}\} \\
engagedEvt(R^i_j) &= \{e \mid \exists k : \{i, ..., j\}, e = e_k)\} \\
engagedProc(R^i_j) &= \{p \mid \exists k : \{i, ..., j\}, p \in engagedProc(s_k, e_k, s_{k+1})\}
\end{aligned}
$$

Two lemmas that form the basis for the algorithm are presented in the following:

Lemma 1. *Let $\mathcal{L}$ be a LTS, $\mathcal{B}$ be a Büchi automaton equivalent to the negation of a LTL formula f, S be a strongly connected component in the product of $\mathcal{L}$ and $\mathcal{B}$.*

- $\mathcal{L} \models_{EWF} f$ *if and only if there does not exist* R^i_j *such that* R^i_j *is accepting and* $alwaysEvt(R^i_j) \subseteq engagedEvt(R^i_j)$
- $\mathcal{L} \models_{PWF} f$ *if and only if there does not exist* R^i_j *such that* R^i_j *is accepting and* $alwaysProc(R^i_j) \subseteq engagedProc(R^i_j)$

- $\mathcal{L} \models_{ESF} f$ *if and only if there does not exist* R^i_j *such that* R^i_j *is accepting and* $onceEvt(R^i_j) \subseteq engagedEvt(R^i_j)$

- $\mathcal{L} \models_{PSF} f$ *if and only if there does not exist* R^i_j *such that* R^i_j *is accepting and* $onceProc(R^i_j) \subseteq engagedProc(R^i_j)$

- $\mathcal{L} \models_{SGF} f$ *if and only if there does not exist* R^i_j *such that* R^i_j *is accepting and* $onceStep(R^i_j) \subseteq engagedStep(R^i_j)$

The lemma can be proved straightforwardly using contradiction. From this lemma, another lemma which is helpful for the algorithm is presented.

Lemma 2. *Let* $\mathcal{L}$ *be a LTS,* $\mathcal{B}$ *be a Büchi automaton equivalent to the negation of a LTL formula* f*,* S *be a strongly connected component in the product of* $\mathcal{L}$ *and* $\mathcal{B}$*.*

- $\mathcal{L} \models_{EWF} f$ *if and only if there does not exist* S *such that* S *is accepting and* $alwaysEvt(S) \subseteq engagedEvt(S)$

- $\mathcal{L} \models_{PWF} f$ *if and only if there does not exist* S *such that* S *is accepting and* $alwaysProc(S) \subseteq engagedProc(S)$

- $\mathcal{L} \models_{ESF} f$ *if and only if there does not exist* S *such that* S *is accepting and* $onceEvt(S) \subseteq engagedEvt(S)$

- $\mathcal{L} \models_{PSF} f$ *if and only if there does not exist* S *such that* S *is accepting and* $onceProc(S) \subseteq engagedProc(S)$

- $\mathcal{L} \models_{SGF} f$ *if and only if there does not exist* S *such that* S *is accepting and* $onceStep(S) \subseteq engagedStep(S)$

Interested readers can find the proof for these two lemmas at (Liu, 2010). We now present the algorithm in Figure 4.2.

```
1: index := 0
2: procedure MC(states)
3:     while there are un-visited states s do
4:         scc := TARJAN(s)
5:         mark states in scc as visited
6:         if ISFAIR(scc)=true then
7:             generate a counterexample
8:             return false
9:         else
10:            scc :=PRUNE(scc)
11:            if MC(scc)= false then
12:                return false
13:    return true
```

Figure 4.2: Algorithm for sequential model checking under fairness assumption

The basic idea is to identify one SCC at a time and then check whether it is fair or not. If it is, the search is over. Otherwise, the SCC is partitioned into multiple smaller strongly connected subgraphs, which are then checked recursively one by one.

At line 4, SCC is identified using Tarjan procedure. If the found SCC is fair, a counterexample is generated and the procedure returns false. If SCC is not fair, a procedure *prune* is used to prune bad states. Bad states are the states that cause the SCC not fair. The intuition behind the pruning procedure is that there may be a fair strongly connected component in the subgraph after removing the bad states. Different fairness can be handled by modifying the *prune* differently. After pruning, at line 11, a recursive call to MC is made to check whether there is a fair strongly connected subgraph within the remaining states.

The definition of $ISFAIR$ function is based on Lemma 2, which deals with all notions of fairness that we are considering. Now, we show different modifications of the $PRUNE$ for different notions of fairness. For EWF, PWF and SGF, if the SCC does not satisfy the fairness assumption, none of its subgraphs will do. The following defines the $PRUNE$ function for those 3 fairness notions.

$$PRUNE(S, EWF) = \begin{cases} S & \text{if } alwaysEvt(S) \subseteq engagedEvt(S) \\ \emptyset & \text{otherwise} \end{cases}$$

$$PRUNE(S, PWF) = \begin{cases} S & \text{if } alwaysProc(S) \subseteq engagedProc(S) \\ \emptyset & \text{otherwise} \end{cases}$$

$$PRUNE(S, SGF) = \begin{cases} S & \text{if } onceStep(S) \subseteq engagedStep(S) \\ \emptyset & \text{otherwise} \end{cases}$$

For ESF and PSF, a state is pruned if and only if there is an event (process) enabled at this state but never enaged in the subgraph.

$$PRUNE(S, ESF) = \{s : S \mid enabledEvt(S) \subseteq engagedEvt(S)\}$$

$$PRUNE(S, PSF) = \{s : S \mid enabledProc(S) \subseteq engagedProc(S)\}$$

The $PRUNE$ function has a linear worst case time complexity to the size of the input SCC. Under no fairness assumption, there is no need for pruning, so the time complexity of the algorithm is linear in the number of transitions. Under EWF, PWF or SGF, each state is only visited at most twice, once by the $TARJAN$ function, and once by the $PRUNE$ function, so the complexity is linear to the number of transitions of the graph as well. For ESF and PSF, in the worst case (i.e., the whole system is strongly connected and only one state is pruned every time), the complexity is equal to the product of the number of states and the number of transitions in the system. The proof for soundness and different fairness can be found in (Sun et al., 2009).

4.2 Multicore SCC algorithm with Spawning Fair Thread

Based on the previous work in the previous subsection, Liu Yang et al. proposed an on-the-fly algorithm (Liu et al., 2009). As observed from the sequential algorithm, when a SCC is detected, it will be processed by four actions: fairness testing, bad states pruning, counterexample generation and recursive sub-SCC detection. We can see that the processing of each SCC can be done independently with each other and also independently with the main Tarjan algorithm. Inspired by these observations, a SCC-based parallel algorithm has been presented with four parts: Tarjan thread, SCC worker thread, SCC worker thread pool and parallel model checker.

In this approach, the Tarjan thread is the main thread that performs the DFS searching of Tarjan's algorithm (line 6). A global variable $jobFinished$ is used to stop the Tarjan thread and all the worker threads as soon as a worker thread reports a counterexample. When a SCC is detected , if the forking conditions are satisfied, a new worker thread is forked and it will process the SCC. Otherwise, the SCC will be process locally. The function for local process is the same as the $WORKER$ function. The *workerthread* basically processes a detected SCC and determine whether the SCC contains an accepting cycle. The forking conditions can be that of size of SCC is big enough and the thread pool is not full. The conditions are there to avoid increasing the overhead cost of creating thread and passing the SCC to the worker thread. The worker threads work on a detected SCC and report whether that SCC contains a counterexample or not.

```
1:  jobFinished := false;
2:  procedure RUN(threadpool, states)
3:      while there are un-visited states s do
4:          if jobFinished then
5:              return ;
6:          scc := TARJAN(s);
7:          mark states in scc as visited;
8:          if forking conditions then
9:              threadpool.forkWorker(scc);
10:         else
11:             process scc locally;
12:     return true
13: procedure WORKER(scc)
14:     if jobFinished then
15:         return ;
16:     if ISFAIR(scc)=true then
17:         generate a counterexample;
18:         JobFinished := true;
19:         return false;
20:     else
21:         scc :=PRUNE(scc)
22:         if MC(scc)= false then
23:             return false;
```

Figure 4.3: Tarjan thread and Fair thread implementation

```
Queue threadQueue := ∅;
procedure FORKWORKER(states)
    lock(threadQueue);
    if ¬JobFinished then
        t = new Thread( WORKER);
        threadPool.Add(t);
        register t to THREAD_TERMINATION;
        threadQueue.enqueue(t);
    unlock(threadQueue);
procedure THREAD_TERMINATION(t)
    lock(threadQueue);
    if t finds counterexample ∧¬JobFinished then
        JobFinished := true;
    threadPool.Remove(t);
    threadQueue.e = dnqueue(t);
    unlock(threadQueue);
```

Figure 4.4: Thread pool implementation

We used thread pool for the task of spawning fair threads. Because the number of threads created is indeterminable at the beginning of the model checking procedure, thread pool is a much more efficient compared to spawning the thread in the normal way. As thread pool has a fixed number threads when created, the overhead of thread creation and thread destruction is avoided, and threads can be reused after finishing its task. The $THREAD\_TERMINATION$ is triggered whenever a fair thread finish its job. The function will check whether the thread found any counterexample. If there is, it will stop all fair thread and the tarjan thread by setting the flag $JobFinished$, which is visible to all threads, otherwise, it waits until all the threads stop.

The time and space complexity of the parallel algorithm are the same as its sequential version, since the parallel algorithm simply splits SCC analysis into worker threads, and the total number of states visited in this algorithm is equal. The algorithm is designed for shared memory architecture, the transitions of the graph and the SCCs are shared among the fair threads and the tarjan thread, so there is no communication overhead. In order to import the algorithm to distributed memory settings, only the states in SCC will be passed. The fair thread will explore the transitions locally to avoid communication overhead.

4.3 Multicore SCC algorithm with Shared SCC Found States

Even though possessing a very good theoretical complexity, the problem with the parallel algorithm in the previous section is that to be able to scale well, the model must contains many big SCCs. In the case of model with very few SCCs, or the SCCs are mostly trivial, the algorithm does not have much speed up compared to the sequential algorithm, no matter how much processors the algorithm is run on.

Inspired by Laarman's algorithm (Laarman et al., 2011) and Evangelista's algorithm (Evangelista et al., 2011) with the idea of using randomized verification with shared memory for synchronization, we propose a new on-the-fly parallel SCC-based algorithm with fairness constraints.

4.3.1 Difficulty of Parallelizing SCC based algorithm

First, we demonstrate the problem with naive parallelizing SCC based algorithm. Consider the LTS in Figure 4.5. Accepting states are marked with double circle. The graph contains a reachable accepting SCC ($s_1 \rightarrow s_2 \rightarrow s1$)

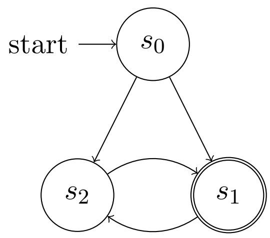


Figure 4.5: A graph with possible faulty execution of the naive parallel version of Tarjan's algorithm

As the order of traversal can be totally different (maybe even in reverse order), the information of index and lowlink data is not possible to be shared among thread. Consider the naive parallel version of the Tarjan's algorithm, where all the visited states are shared, and thread does not re-traverse visited states. Running this algorithm with two thread t_1 and t_2 may not detect the accepting run in the graph.

The traversal done by thread t_1 first visits s_0. At the same time, thread t_2 explores s_0 before

t_1 marks s_0 as visited. Then, t_1 visits s_1, marks s_1 as visited, and halts there temporarily. t_2 now proceeds to s_2 and mark s_2 as visited. At this moment, both threads cannot proceed anymore, backtrack to s_0 and terminate without report any counterexample. Using a lock to make each traversing step atomic is not feasible, cause it negates the initial purpose of using parallel algorithm, and furthermore, it still produces the same problem of cycle detection if the graph has multiple starting states. This example highlights the key idea that one state can only be marked as visited for other processes to avoid visiting only if when the whole SCC contains that state is detected or other thread does not interrupt with the SCC detection.

4.3.2 Details of the algorithm

Observing that if one state is found to be in a SCC which is already processed or stored locally by one thread, that state does not need to be revisited by other threads. Thus, having a global bit for each state to mark whether the state has been found along with its SCC or not, and making other threads to avoid those states will prune down the total search space. We can make use of the scc_found bit in the sequential algorithm, so that the memory space for each thread does not increase. At the same time, the global bit must be stored in a data structure that allows concurrent reads and writes without creating duplicate entries.

We choose to use the ConcurrentDictionary provided by Microsoft .NET 4.0 which make use of lightweight synchronization and smart locking mechanisms. Another possible choice is to use a shared lockless hash table from (Laarman et al., 2010) that scales well for this purpose.

```
procedure PARALLEL_TARJAN(s_0, i)
    Stack S := ∅; Stack scc_queue := ∅; index := 0; S.push(s_0);
    while S ≠ ∅ do
        v := S.peek(); done := true;
        if (v.scc_found) then
            repeat
                scc_queue.pop(); continue;
            until scc_queue.Peek().index[i] <= v.index[i]
        if v is un-visited then
            index := index + 1; v.index[i] := index;
        for w in successor_i(v) do
            if w is un-visited ∧¬w.scc_found then
                S.push(w); done := false; break;
        if done then
            S.pop(); v.lowlink[i] = v.index[i];
            for w in successor_i(v) do
                if ¬w.scc_found  then
                    if w.index[i] > v.index[i] then
                        v.lowlink[i] := min{v.lowlink[i], w.lowlink[i]};
                    else
                        v.lowlink[i] := min{v.lowlink[i], w.index[i]};
            if v.lowlink[i] = v.index[i] then
                scc := ∅; scc.push(v); v.scc_found = 1;
                repeat
                    k = scc_queue.pop();scc.push(k); k.scc_found = true;
                until scc_queue.Peek().index[i] <= v.index[i]
                Process scc;
            else
                scc_queue.push(v) ;
```


Figure 4.6: The Parallel Algorithm with shared SCC Found States

The details for the algorithm is shown in Figure 4.6. Similar to Laarman's algorithm, with different DFS traversal $successor_i$ for each thread, different threads may explore different parts of the reachable state graph. In line 12, we modify the conditions for choosing which neighbour to traverse. If one of the neighbour is found to be found in a SCC, we do not need to traverse that state further. Line 5-8 add another checking condition that accommodate with the global SCC found bit. If the current state is found to be in a SCC by another thread, we can stop exploring that state. For the algorithm to be correct, all the states inside *scc_queue* which has greater index than the current state must be popped out. Each thread keeps a local copy of the call stack S and *scc_queue*. Each thread must also store an local index value and and local lowlink value for each states. An additional bit for each state is required for the global SCC found bit. In line 27, we process the SCC the same as the sequential algorithm with fairness from (Sun et al., 2009). Thus, our algorithm also has the capability to verify models under different fairness constraint. When a counterexample is found, all threads are stopped and the counterexample is returned to the main thread.

In the worst case where there is only one SCC in the graph, or all threads find SCCs at the same time, the graph will be explored N times by N threads. Therefore the worst case time complexity of the algorithm is $O(N \times M)$ where N is the number of threads. M is equal the number of transitions in the model under no fairness, EWF, PWF or GSF, and is equal to the product of the number of states and the number of transitions in the model under PSF or ESF.

Now we prove the correctness of the algorithm

Theorem 1. *Algorithm* PARALLEL_TARJAN *reports an accepting fair SCC if and only if there is an accepting SCC in the model.*

Proof. First we argue that the algorithm correctly detect SCCs in the model. In case the SCC is found by only one thread, and no other thread has visited any states inside the SCC, we can see it is similar to SCC detection in the sequential algorithm, thus the correctness is confirmed. The only problem is when two or more threads are accessing one SCC, cause there may be the case that the SCC may be left out undetected. However, as stated in the algorithm, a state is marked as SCC found if and only if the whole SCC containing that state is either processed or is stored locally in one of the threads. This storing thread will later process the SCC, even if all other threads stop traversing in this SCC. Thus, if the graph has any SCC, it will be found by the algorithm.

Given a detected SCC, the fairness checking and pruning procedures are done independently by the detecting thread without any communication, the correctness of these two procedures are not affected by parallel settings. □

Because the graph has finite number of states, in addition, there is no waiting statement in this algorithm, it is easy to see that the algorithm will eventually terminate.

Chapter 5

Experiments

In this section, we compare the performance of various model checking algorithms which have been implemented in PAT during this project. The experiments are divided into two sections: one for model checking under different notions of fairness, and the other for general model checking algorithms.

5.1 Experiments for Model Checking with Fairness

In this section, 4 SCC-based algorithms without fairness constraint are experimented: the sequential algorithm (*TJ*) from (Sun et al., 2009), the parallel algorithm (*PTJ1*) from (Liu et al., 2009), the swarm Tarjan algorithm (*SV-TJ*) and the purposed parallel algorithm (*PTJ2*)

Table 1 summaries the verification statistics on classic dining philosophers (DP) and recently developed population protocols. The population protocols include leader election for complete networks (LE_C) for network rings (LE_R) (Fischer & Jiang, 2006) and token circulation for network rings (Angluin et al., 2008). We modify the DP model so that it is deadlock-free (i.e., by letting one of the philosophers to pick up the forks in a different order). The property is that a philosopher never starves to death. The property for the leader election protocols is that eventually always there is one and only one leader in the network. Correctness of all these algorithms relies on different notions of fairness.

Model	Size	EWF					SGF				
		Res.	TJ	PTJ1	SV-TJ	PTJ2	Res.	TJ	PTJ	SV-TJ	PTJ2
DP	6	No	0.36	0.39	0.51	0.46	Yes	1.10	1.07	1.51	1.33
DP	7	No	1.68	1.63	1.42	1.49	Yes	4.27	4.17	5.06	3.42
DP	8	No	14.47	13.38	10.71	10.84	Yes	21.41	19.62	23.66	15.87
DP	9	No	164.15	162.96	128.55	130.37	Yes	95.38	89.51	101.4	77.08
LE_C	5	Yes	1.54	1.51	1.97	1.76	Yes	1.62	1.6	2.24	2.1
LE_C	6	Yes	8.63	8.31	10.39	8.03	Yes	8.97	8.61	10.96	7.74
LE_C	7	Yes	44.54	43.32	49.86	40.25	Yes	44.67	43.09	51.27	41.58
LE_C	8	Yes	206.49	202.69	215.32	195.17	Yes	209.22	196.98	223.16	176.61
LE_R	3	No	0.09	0.10	0.24	0.27	Yes	1.40	1.23	1.51	1.21
LE_R	4	No	0.27	0.3	0.41	0.43	Yes	17.43	15.04	20.71	12.6
LE_R	5	No	0.71	0.72	0.74	0.77	Yes	203.21	185.47	246.82	154.65

Table 5.1: Experiment results on a PC running Windows 7 with 2.13 GHz quad-core Intel 720QM CPU and 3 GB memory on DP and population protocols

As we see from table 5.1, when the model is small, either TJ or $PTJ1$ is faster because they are not penalized from the thread creation and destruction as in $SV-TJ$ and $PTJ2$. For bigger model, when the model does contain accepting fair SCCs, swarm verification $SV-TJ$ has the best performance. $PTJ2$ is a bit slower due to work sharing effects. TJ and $PTJ1$ nd accepting cycles roughly within the same time, which is expected. For big model without counterexample, $SV-TJ$ has the worst runtime because it has to travel to graph 4 times. The overhead of $SV-TJ$ is quite considerable, increases about from 10% to 50% compared to the sequential algorithm TJ. $PTJ2$ has the best performance, followed by $PTJ1$.

Even though, $PTJ1$ and $PTJ2$ have certain speed up compared to the sequential algorithm, they do not exhibit a good scalability in these models. For $PTJ1$, the reason is that these models contains a lot of trivial SCCs, and there are only few non-trivial SCCs. As a result, there is little work that can be separated out for the worker threads to speed up the model checking, and the communication overhead makes $PTJ1$ slower. For $PTJ2$, Table 5.2 shows some statistics of the graph traversal for the parallel algorithm.

Model	Size	NoStates	NoVisitedStates	Stddev	Ratio
DP	6	9841	10252	1891	4.17
DP	7	37761	39246	9718	3.93
DP	8	143501	147154	34752	2.54
DP	9	533681	543752	142516	1.74
LE_C	5	2587	2710	682	4.75
LE_C	6	7831	8141	2358	3.95
LE_C	7	22058	22557	6910	2.26
LE_C	8	58946	60179	17308	2.09
LE_R	3	6946	7239	1391	4.21
LE_R	4	65468	66893	17371	2.17

Table 5.2: Statistics of $PTJ2$ algorithm

In this table, NoStates denotes the total number of states in the graph, NoVisitedStates denotes the total number of states in the graphs which are visited by the algorithm, Ratio is equal to NoStates divided by NoVisitedStates, which is used to calculate the percentage of redundant states that the parallel algorithm visited, Stddev is the standard deviation of the numbers of visited states by each thread. Even though the Ratio is small, which means only a few states are re visited, the Stddev is really big compared to the total number of states. This only happens when most of the traversing work is done by about 2 threads. We illustrate this problem by one example in Figure 5.1.

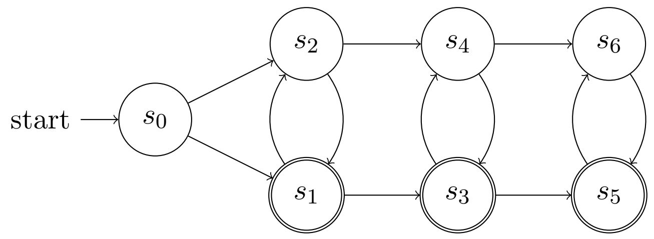


Figure 5.1: A graph with fixed order of SCC-exploring

This model has 3 accepting SCCs ($s1 \rightarrow s2 \rightarrow s1$), ($s3 \rightarrow s4 \rightarrow s3$), ($s5 \rightarrow s6 \rightarrow s5$). However, these 3 SCCs cannot be detected in parallel, instead there is a fixed order of detecting these SCCs. The (s_5, s_6) SCC must be detected in order to detect the SCC (s_3, s_4), and similarly, we have to detect SCC (s_3, s_4) before detecting (s_1, s_2). In this kind of model, even though there are multiple SCCs in the graph, the parallel algorithm have little speed up compare to the sequential algorithm, except the case when the SCCs are very big, and processing a SCC takes a lot of time. From this example, we can see that our algorithm can only achieve excellent speed up when the graph is sparse enough.

In order show the potential effectiveness of the parallel algorithm, we create two models (PAR1, PAR2) such that the their state space contains several SCCs, each of which has a big number of states. As a result, both $PTJ1$ and $PTJ2$ exhibit very impressive speed up. The statistics are summarized in table 5.3. From the table, we see that both $PTJ1$ and $PTJ2$ performs more than 60% speed up for model without accepting cycles. For big model, the speed up can be up to 90%. For big model with long counter example, swarm verification $SV - TJ$ has the best performance with more than 50% speed up in most of the cases.

Model	Size	EWF					SGF				
		Res.	TJ	PTJ1	SV-TJ	PTJ2	Res.	TJ	PTJ1	SV-TJ	PTJ2
PAR1	6	No	8.76	4.75	4.12	4.46	Yes	18.3	8.16	24.62	8.12
PAR1	7	No	14.43	6.72	6.51	6.79	Yes	67.23	26.85	75.03	26.91
PAR1	8	No	21.37	8.56	8.38	8.52	Yes	159.27	54.12	184.33	53.74
PAR1	9	No	35.1	12.71	11.46	12.15	Yes	462	127	499	126
PAR2	7	No	0.24	0.26	0.37	0.41	Yes	8.97	3.61	10.96	3.74
PAR2	8	No	0.29	0.28	0.43	0.44	Yes	24.12	11.09	28.46	9.91
PAR2	9	No	0.41	0.46	0.55	0.62	Yes	135.7	45.6	156.1	45.8

Table 5.3: Experiment results on a PC runnings Windows 7 with 2.13 GHz quad-core Intel 720QM CPU and 3 GB memory on sparse big SCC model

5.2 Experiments for Model Checking under no Fairness

In this section, we compare all the algorithm implemented in this project, and summarize the result in table 5.4. Sequential Nested DFS (NDFS), swarm Nested DFS (SV-DFS) and parallel Nested DFS (MC-DFS) fromalfons are added to compare with the SCC-based algorithms. With the early detection in Blue DFS improvement, Nested DFS can find simple counter example much faster compared to SCC-based algorithms. For example, PAR1 model contains a 2-states SCC right at the beginning of the graph, which is detected very fast by Nested DFS- based algorithms, but not so well by SCC-based algorithms. However, when there is no counterexample in the graph, Nested DFS is a bit slower than SCC-based because it has to traverse each states at most twice by the Blue DFS and Red DFS. The property for the DP below is that a philosopher never eat twice in a row, and it is valid under no fairness constraint for the deadlock-free DP. From the result with DP model, we see that MC-DFS also does not do well with non-sparse graph, similar to our proposed algorithm. We create a new model PAR3 with sparse graph and a valid property to compare the effectiveness of these two algorithms on sparse graph. The result shows that our algorithm is comparable with MC-DFS and scales very well with sparse graphs.

Model	Size	Res.	NDFS	SV-NDFS	MC-NDFS	TJ	SV-TJ	PTJ1	PTJ2
PAR1	6	No	0.016	0.031	0.04	8.54	4.12	4.36	4.47
PAR1	7	No	0.017	0.033	0.041	13.98	6.69	7.85	7.15
PAR1	8	No	0.017	0.032	0.045	20.12	9.79	10.33	10.2
DP	8	Yes	9.81	12.14	7.38	9.63	11.73	7.95	7.52
DP	9	Yes	43.2	49.87	32.59	42.93	50.11	31.83	32.24
DP	10	Yes	187.76	211.3	143.9	185.89	215.6	146.13	147.5
PAR3	7	Yes	23.61	26.57	9.07	22.11	25.9	10.02	9.24
PAR3	8	Yes	176.3	199.8	49.61	161.69	191.82	48.41	47.72

Table 5.4: Experiment results on a PC running Windows 7 with 2.13 GHz quad-core Intel 720QM CPU and 3 GB memory with all algorithms

Chapter 6

Conclusion

6.1 Contributions

We have proposed a new parallel algorithm for the model checking under fairness assumption problem. It is a variation of the well-known Tarjan Strongly Connected Component dedicated to multi-core and shared memory architectures. Although, it does not theoretically scale, our experiments revealed that it can provide good accelerations on a variety of different models. Moreover, the algorithm can detect accepting cycles on-the-fly under various fairness assumptions which few parallel algorithms designed so far are able to. At the same time, various state-of-the-art algorithms have been implemented in PAT during the project which may prove to be very helpful in further research.

6.2 Future Work

In the future, several extensions of the work presented here will be considered. First, in the current implementation, the post-order of $successor_i$ function is totally based on randomness. A heuristic function might be of great benefit here in order to reduce the number of threads visiting a given state. More experiments will be conducted in the future to see both the scalability and limitations with more CPU cores. At the same time, as discussed in the chapter 5 an analysis of graphs structures will help to determine which extent the proposed algorithm could be improved.

References

Angluin, D., Aspnes, J., Fischer, M. J., & Jiang, H. (2008). Self-stabilizing population protocols. *ACM Transactions on Autonomous and Adaptive Systems*, *3*(4), November, 2008, 13.

Angluin, D., Fischer, M. J., & Jiang, H. (2006). Stabilizing Consensus in Mobile Networks. *IN DCOSS* (pp. 37–50), 2006.

Barnat, J., Brim, L., Ceka, M., & Lamr, T. (2009). CUDA Accelerated LTL Model Checking. *2009 15th International Conference on Parallel and Distributed Systems*, (201), 2009, 34–41.

Barnat, J., Brim, L., & Chaloupka, J. (2003). Parallel Breadth-First Search LTL Model-Checking. *18th IEEE International Conference on Automated Software Engineering (ASE'03)* (pp. 106–115), Oct., 2003: IEEE Computer Society.

Barnat, J., Brim, L., & Ročkai, P. (2007). Scalable Multi-core LTL Model-Checking. *Model Checking Software*, Vol. 4595 of *LNCS* (pp. 187–203), 2007.

Brim, L., Černá, I., Moravec, P., & Šimša, J. (2004). Accepting Predecessors are Better than Back Edges in Distributed LTL Model-Checking. *5th International Conference on Formal Methods in Computer-Aided Design (FMCAD'04)*, Vol. 3312 of *LNCS* (pp. 352–366), 2004.

Cerná, I., & Pelánek, R. (2003). Distributed Explicit Fair Cycle Detection (Set Based Approach). *SPIN* (pp. 49–73), 2003.

Courcoubetis, C., Vardi, M., Wolper, P., & Yannakakis, M. (1992). Memory-Efficient Algorithms for the Verification of Temporal Properties. *Formal Methods In System Design* (pp. 275–288), 1992.

Evangelista, S., Petrucci, L., & Youcef, S. (2011). Parallel Nested Depth-First Searches for LTL Model Checking. *ATVA* (pp. 381–396), 2011.

Fischer, M., & Jiang, H. (2006). Self-stabilizing leader election in networks of finite-state anonymous agents. *In Proc. 10th International Conference on Principles of Distributed Systems, number 4305 in LNCS* (pp. 395–409), 2006: Springer.

Gaiser, A., & Schwoon, S. (2009). Comparison of Algorithms for Checking Emptiness on Büchi Automata. *MEMICS*, 2009.

Gastin, P., Moro, B., & Zeitoun, M. (2004). Minimization of counterexamples in SPIN. *Model Checking Software: 11th International SPIN Workshop (SPIN'04)*, Vol. 2989 of *Lect. Notes in Comp. Science* (pp. 92–108), 2004: Springer.

Giannakopoulou, D., Magee, J., & Kramer, J. (1999). Checking Progress with Action Priority: Is it Fair? *Proc. ESEC/FSE '99* (pp. 511–527), 1999.

Holzmann, G. (2003). *Spin model checker, the: primer and reference manual.* Addison-Wesley Professional, first edition.

Holzmann, G. J. (1999). Software Model Checking. *In Proceeding Forte*, *28*, 1999, 481–497.

Holzmann, G. J., & Bosnacki, D. (2007). The Design of a Multicore Extension of the SPIN Model Checker. *IEEE Transactions on Software Engineering*, *33*, 2007, 659–674.

Holzmann, G. J., Peled, D., & Yannakakis, M. (1996). On nested depth first search. *Proc. SPIN Workshop* (pp. 23–32), 1996: American Mathematical Society.

Laarman, A., Langerak, R., van de, J. P., Weber, M., & Wijs, A. (2011). Multi-Core Nested Depth-First Search. *9th International Symposium on Automated Technology for Verification and Analysis, ATVA 2011*, Lecture Notes in Computer Science, July, 2011.

Laarman, A., van de Pol, J., & Weber, M. (2010). Boosting multi-core reachability performance with shared hash tables. *FMCAD* (pp. 247–255), 2010.

Lamport, L. (2000). Fairness and hyperfairness. *Distributed Computing*, *13*(4), 2000, 239–245.

Liu, Y. (2010). *Model Checking Concurrent and Real-time Systems: the PAT Approach.* PhD thesis, National University of Singapore.

Liu, Y., Sun, J., & Dong, J. S. (2009). Scalable Multi-core Model Checking Fairness Enhanced Systems. *Proceedings of the 11th IEEEInternational Conference on Formal Engineering Methods*, Vol. 5885 (pp. 426–445), 2009.

Pang, J., Luo, Z., & Deng, Y. (2008). On Automatic Verification of Self-Stabilizing Population Protocols. *Theoretical Aspects of Software Engineering, Joint IEEE/IFIP Symposium on*, *0*, 2008, 185–192.

Pnueli, A., & Sa'ar, Y. (2007). *All You Need is Compassion (TR)* (Technical report). Department of Computer Science, New York University.

Reif, J. H. (1985). Depth-First Search is Inherently Sequential. *Information Processing Letters*, *20*(5), 1985, 229–234.

Schwoon, S. (2005). A note on on-the-fly verification algorithms. *In Proc. of TACAS05, LNCS* (pp. 174–190), 2005: Springer-Verlag.

Sun, J., Liu, Y., Dong, J. S., & Pang, J. (2009). PAT: Towards Flexible Verification under Fairness. *Computer Aided Verification*, Vol. 5643 (pp. 709–714), 2009.

Tarjan, R. E. (1972). Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, *1*(2), 1972, 146–160.

Vardi, Y. M., & Wolper, P. (1986). An Automata-Theoretic Approach to Automatic Program Verification. *Proceedings of the 1st Annual Symposium on Logic in Computer Science (LICS'86)* (pp. 332–344), June, 1986: IEEE Comp. Soc. Press.