View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Teeside University's Research Repository

Concurrent On-the-fly SCC Detection for Automata-based Model Checking with Fairness Assumption

Zhimin Wu*, Yi Xu*, Akin Gunay* Yang Liu*, Shengchao Qin‡
*Nanyang Technological University, Singapore
‡Teesside University, United Kindom

***Abstract*—Model checking is an automated technique for verifying temporal logic properties of finite state systems. Tarjan's algorithm for detecting Strongly Connected Components (SCCs) is a widely used depth-first search procedure for Automata-based (LTL) model checking. It works on the SCC detection on-the-fly with the composition of transition systems and Büchi Automaton (state space generation), which has been deployed as sequential implementations in many tools. However, these implementations suffer from heavy time cost for systems which involve a large number of SCC explorations. To address this issue, in this paper, we develop a concurrent SCC detection approach for the on-the-fly generated state space in LTL model checking by expanding the existing concurrent Tarjan's algorithm. Besides, we involve fairness checking. Different that the previous work, which performs fairness checking after the generation of a complete SCC, in our approach we perform fairness checking during SCC generation to improve efficiency. We implement our approach in PAT model checker. Our experimental results show that our approach achieves up to 2X speedup for the complete SCC detection in large-scale system models compared to the sequential on-the-fly model checking in PAT. Besides, our parallel on-the-fly fairness checking approach speedups fairness checking around 2X∼45X.**

I. Introduction

Automata-based LTL model checking is emptiness checking of the composition of a transition system $\mathcal{M}$ and a Büchi Automaton $\mathcal{B}_{\neg\varphi}$, which represents the negation of an LTL property φ [2]. The composition process indicates that the state space is unknown in advance, which is on-the-fly generated. The idea of emptiness checking is to search the on-the-fly generated state space to find an execution path that is accepted by the Büchi automaton. *Strongly Connected Component* (SCC) based model checking that uses Tarjan's algorithm [18] for SCC detection, is a well-known approach for LTL model checking. In this approach, the problem of LTL model checking is converted to the detection of an infinite path that is accepted by the Büchi Automaton. The infinite path contains SCCs with accepting cycles. It uses depth-frist search to explore the state space to detect SCC.

Fairness and liveness are two essential notions for faithfully modeling the execution progress of a process in a collection of concurrent processes [14]. Fairness constraints can be expressed in LTL. Thus, fairness checking can be integrated into SCC-based LTL model checking, which expand the original process to the verification of the fairness assumption's satisfaction in all detected SCCs.

There are many sequential implementations of SCC detection for automata-based LTL model checking with fairness assumption. However, besides the state space explosion problem [4], sequential implementations of automata-based LTL model checking with fairness do not scale well for large scale systems with a large number of SCCs. In Automata-based LTL model checking with fairness, the verification process is overlapped with the state space generation, which is the product of $\mathcal{M}$ and $\mathcal{B}_{\neg\varphi}$. The verification process may immediately reports failure, if it detects a fair SCC. While sometimes the verification process has heavy costs, since the fair SCC does not exist or appear after a large number of SCCs being detected. In this paper, we extend the concurrent implementation [11] of Tarjan's algorithm based on known state space to concurrent SCC detection for LTL model checking with on-the-fly generated state space. We aim to improve the verification performance. Furthermore, we develop an efficient approach for parallel fairness checking.

The main challenges of utilizing parallel computing to accelerate Automata-based LTL model checking with fairness are: (1) Tarjan's algorithm is a DFS process, which has attracted many researches on its parallelization. However, all researches work on the complete state space. It is challenging and significant to convert a parallel Tarjan's algorithm to be available for on-the-fly generated state space. 2) An efficient data distribution approach is necessary since the state space is unknown in advance. It is also important to maintain data consistency. 3) The sequential algorithm mentioned in [21] verifies the fairness after the generation of a complete SCC, which can be regarded as an independent part from the verification process. A better way for fairness checking is promising.

To solve these challenges, we expand the concurrent Tarjan's algorithm from Lowe [11] to fulfil the Automata-based LTL model checking for SCC detection on on-the-fly generated state space. Our key contributions in this paper are as follows: 1) Lowe's Tarjan's algorithm depends on complete state space (unrooted mode). Although he mentioned rooted mode in both [12] and [11], there is no details or experiments for this. We expand the algorithm to fit the on-the-fly state space generation. Based on it, we build the parallel approach to cover the features of on-the-fly SCC-based LTL model checking. 2) We build our own data distribution rules for on-the-fly generated state space. 3) We design and develop an efficient on-the-fly parallel fairness checking approach, which performs the fairness checking during the generation of SCC

instead of performing it separately after the generation like [21] does. 4) We implement our approach in the Process Analysis Toolkit (PAT) [17] and make it work on a wide range of system models. Our evaluation shows that after integrating our approach, we achieve a 2X performance improvement in LTL model checking which involve the exploration of a large number of SCCs. And 2X∼45X speedup for fairness checking.

The structure of this paper is as follows: In Section II, we introduce the background and related work, including the concurrent Tarjan's algorithm from Lowe [11]. In Section III, we present our design of the concurrent on-the-fly SCC detection for automata based (LTL) model checking with fairness assumption. In Section IV we present our experiments and evaluation. Finally, we present the conclusion and future work in Section V.

II. Background and Related Work

A. Automata-based Model Checking for LTL

Given a system model M and an LTL property φ, model checking is to check $M \models \varphi$. The model of concurrent systems M_c can be represented as a composition of several interleaving processes, each of which is expressed as labelled transition systems.

Definition 1: Given a set of invisible and visible events Σ, a *labelled transition system* (LTS) is a 3-tuple $M = (S, s_0, \rightarrow)$ where S is a set of states, $s_0 \in S$ is the initial state, $\rightarrow \subseteq S \times \Sigma \times S$ is a transition relation.

In Automata-based model checking, the negation of φ is expressed in an equivalent Büchi automaton $\mathcal{B}_{\neg\varphi}$ (Def. 2), which is then composed with the LTS representing the system model. The composition process, formal defined in Def. 3, generates the state space of M_c. Each state in the state space of M_c is a meta-state (i.e., a vector of states), which we define as $sv = S \times S_{\mathcal{B}}$, where $n \in 1..n, S_{\mathcal{B}} \in \mathcal{B}_{\neg\varphi}$. Then, checking $M_c \models \varphi$ can be done as checking of the emptiness of the product between M_c and $\mathcal{B}_{\neg\varphi}$. SCC detection is a key subroutine in this process.

Definition 2: A Büchi Automaton is a tuple $\mathcal{A} = (\Sigma_{\mathcal{B}}, S_{\mathcal{B}}, \rho, b_0, F)$, where Σ is an alphabet, $S_{\mathcal{B}}$ is a set of Büchi states, $\rho : S_{\mathcal{B}} \times \Sigma$ is a nondeterministic transition function, $b_0 \in S_{\mathcal{B}}$ is an initial state, and $F \subseteq B$ is a set of accepting states.

Definition 3: Given a LTS representing the system model $M = (S, s_0, \rightarrow)$ and a set of alphabets (events) Σ, a Büchi Automaton $\mathcal{B}_{\neg\varphi}$, the *composition* of M and $\mathcal{B}_{\neg\varphi}$ is a transition system: $M \times \mathcal{B}_{\neg\varphi} = (S \times B, (s_0, b_0)), \rightarrow)$. The corresponding set of events is $\Sigma \cup \Sigma_{\mathcal{B}}$. The transition relation after the parallel composition is the smallest transition relation which satisfies the following:

1) $((s_1, b_1), \alpha, (s'_1, b'_1)) \in\rightarrow$ if $(s_1, \alpha, s'_1) \in\rightarrow \wedge ... \wedge (s_n, \alpha, b'_n \in\rightarrow)$.
2) $((s_1, b_1), \alpha, (s'_1, b_1)) \in\rightarrow$ if $(s_1, \alpha, s'_1) \in\rightarrow \wedge \alpha \notin \Sigma_{\mathcal{B}}$
3) $((s_1, b_1), \alpha, (s_1, b'_1)) \in\rightarrow$ if $(b_1, \alpha, b'_1) \in\rightarrow \wedge \alpha \notin \Sigma$.

Scalability of LTL model checking suffers from state space explosion [4]. On-the-fly model checking [5] checks the emptiness while constructing the product state space of the interleaving processes. In this context, Automata-based LTL model checking does SCC exploration on the composition of φ and $\mathcal{B}_{\neg\varphi}$ with Tarjan's algorithm. The sequential implementation of this process is shown in Algorithm , which is the version in PAT [21].

Algorithm 1: Sequential on-the-fly LTL Model Checking

```
   Input: M,B, s0, b0
1  Define OGTrans,StepStack,TaskStack,SCCSet;
2  Product : S0 = GenerateIniS(s0, b0, M, B) ;
3  Add(OGTrans, S0);
4  TaskStack.push(S0);
5  i = 0, preorder = {}, visited = {};
6  while TaskStack! = EMPTY do
7      S = TaskStack.peek();
8      if S ∉ preorder then
9          preorder[S] = i;
10         i + +;
11     done = true;
12     if S ∈ visited then
13         S'[] = visited[S];
14         forall the Si ∈ Sn do
15             if Si ∉ preorder then
16                 if done then
17                     TaskStack.push(Si);
18                     done = false;
19     else
20         s'n[] = (S.sn).MakeOneMove(M);
21         Product : S'[] = GlobalSuccessor(s'n[], S.bn);
22         forall the S'i ∈ S'[] do
23             if S'i ∉ preorder then
24                 OGTrans[S].add(S'i);
25                 if done then
26                     TaskStack.push(S'i);
27                     done = false;
28         visited.add(S);
29     if done then
30         lowlinkS = S.lowlink;
31         preorderS = lowlinkS;
32         forall the Si ∈ OGTrans[S] do
33             if Si ∉ SCCSet then
34                 if preorder[Si] > preorderS then
35                     S.lowlink = Min(S.lowlink, Si.lowlink);
36                 else
37                     S.lowlink = Min(S.lowlink, preorder[Si]);
38         if lowlink == preorder then
39             SCCSet.add(S);
40             backtrack Stepstack → SCCSet;
41             Optional: Fairness Checking; if ISFair then
42                 Record Result;
43                 Generate Counterexample;
44                 report Counterexample;
45             forall the S'i ∈ SCCSet do
46                 visited.remove(Si);
47                 OGTrams.remove(Si);
48         else
49             StepStack.push(S);
50 return Result;
```

B. Fairness Assumption and Model Checking with Fairness

Fairness is typically needed to prove liveness, which is concerned with a fair resolution of nondeterminism [13]. Without fairness, liveness verification may output unrealistic loops during which one process or event is infinitely ignored by the scheduler or one processor is infinitely faster than others [16].

Given a LTS $M = (S, s_0, \rightarrow)$ with event set Σ we first provide the following definitions on events and processes, where α is a typical element of Σ: (1) *enabledEvt(s)* is the set of enabled events at $s \in S$ such that $\{\alpha \mid (s, \alpha, s') \in\rightarrow\}$. (2) *enabledProc(s)* is the set of enabled processes at s. That is, if a process $p \in enablePro(s)$, it can progress in the system state s. (3) *engagedEvt(s, α, s')* indicates the engaged event α (4) *engagedProc(s, α, s')* indicates the set of processes which progress via the transition $(s, \alpha, s') \in\rightarrow$.

Given an execution R_i^j,

$$R_i^j = ((s_0, b_0), \alpha_0, ...(s_i, b_i), \alpha_i...(s_j, b_j), \alpha_j, (s_{j+1}, b_{j+1}))$$

where $s_j \in S$ and $b_j \in \mathcal{A}_{\neg\varphi}$. There is an SCC in this execution which indicates $s_i = s_{j+1}, b_i = b_{j+1}$. If $(b_0, b_1...b_k...)$ is accepting to $\mathcal{A}_{\neg\varphi}$, R_i^j is accepting. With specific fairness assumption, R_i^j is fair if and only if $(s_0, \alpha_0, s_1, \alpha_1...s_k, \alpha_k...)$ is fair. Several more definitions are shown: (1) $alwaysEvt((R_i)^j) = \{\alpha \mid \forall k : \{i..j\}, \alpha \in enabledEvt(s_k)\}$ (2) $alwaysPro((R_i)^j)$=$\{p|\forall k$: $\{i..j\}, p \in enabledPro(s_k)\}$. (3) $onceEvt((R_i)^j) = \{\alpha \mid \exists k : \{i..j\}, \alpha\in enabledEvt(s_k)\}$. (4)$oncePro((R_i)^j)$=$\{p|\exists k$: $\{i..j\}, p\in enabledPro(s_k)\}$.

We consider five categories of fairness assumptions: (1) *Strong Global Fairness (SGF)*: An execution satisfies SGF if and only if for all $(s, \alpha, s') \in\rightarrow$, if $s = s_i$ for infinitely many i , $s_i = s$ and $\alpha_i = \alpha$ and $s_{i+1} = s'$ for infinitely many i. (2) *Event-level Strong Fairness (ESF)*: An execution R satisfies ESF if and only if for all events α, if α is infinitely often enabled, then $\alpha_i = \alpha$ for infinitely many i. (3) *Event-level Weak Fairness (EWF)* [9]: An execution R satisfies EWF if and only if for all α, if α finally becomes enabled forever in R, then $\alpha_i = \alpha$ for infinitely many i. (4) *Process-level Strong Fairness (PSF)*: An execution satisfies PSF if and only if for all processes p, if p is infinitely often enabled, then $p \in engagedProc(s_i, \alpha_i, s_{i+1})$ for infinitely many i. (5) *Process-level Weak Fairness (PWF)*: An execution E satisfies PWF if and only if for every processes p, if p eventually becomes enabled forever in E, then $p \in engagedProc(s_i, \alpha_i, s_{i+1})$ for infinitely many i.

In particular, for automaton-based model checking with fairness assumption, we always build the LTL formula with fairness constraints. But the size of the Büchi automaton is exponential to the size of the LTL formula, which makes it infeasible to handle large formulas. e.g., formulas with many fairness constraints. Thus, in this paper, our sequential SCC-based LTL model checking with fairness algorithm adopts the solution of PAT [17] and handles the fairness checking on the generated SCC as follows. Given an SCC S in the product of M and $\mathcal{A}_{\neg\varphi}$ and fairness assumption $\mathcal{F}$, if there is no such S that S is accepting and:

1) $onceEvt(S) \subseteq engagedEvt(S) \iff M$ satisfies F_{ESF}.
2) $alwaysEvt(S) \subseteq engagedEvt(S) \iff M$ satisfies F_{ESF}.
3) $oncePro(S) \subseteq engagedPro(S) \iff M$ satisfies F_{PSF}.
4) $alwaysPro(S) \subseteq engagedPro(S) \iff M$ satisfies F_{PWF}.

C. Related work

Parallel computing has been widely used to deal with model checking problems. [1] presents the GPU accelerated state space generation. Our previous work [19] presents the GPU-based counterexample generation for LTL model checking. [20] presents the GPU-based on-the-fly reachability checking. [7] and [8] present a multicore NDFS algorithm for LTL model checking. [10] proposes a parallel LTL model checking algorithm which starts multiple threads to generate the SCC when it is detected, and the fairness checking occurs in the thread for SCC generation. [6] presents a state compression and reconstruction approach. It builds a state space exploration algorithm based on the shared memory multi-core architecture. [3] presents a novel emptiness checking approach for LTL model checking, which is based on SCC enumeration and support TGBA. Its key feature it the usage of a global union-find data structure. [15] presents a parallel SCC decomposition based on the set-based SCC algorithms instead of Tarjan's algorithm. [11] introduces some concurrent DFS-based algorithms, such as concurrent Tarjan's algorithm for SCC detection. It is the main related work for our approach. We describe it in detail below.

Concurrent Tarjan's Algorithm: Lowe [11] design a concurrent Tarjan's algorithm for SCC detection. Given a system model M with complete state space, the SCC exploration starts from multiple different states. They define an object *Search* as the unit for exploration, and an object *Scheduler* for the arrangement of searches to threads. Compared to traditional Tarjan's algorithm, Lowe's approach differs in three parts: 1) If a state is visited, it should record the ID of the corresponding Search. A state can only be visited by one Search. 2) If a Search i explores a state that records other Search's ID j, i is suspend to j. 3) The suspended relation can be broken when the status of the state becomes *completed*, which means the state has been detected to be in one SCC. One iteration of the overall process is shown in Fig. 1: Each Search has its own *Taskstack* and *Stepstack*. It follows the sequential Tarjan's algorithm to explore the successors (child) of its initial state. Then based on the description above, they introduce a *SuspendingRealtion* set. Searches need to check if there is a cycle in the *SuspendingRelation*, shown in *part A* in Fig. 2. If so, the states in the Searches that related to this cycle are transferred to one single Search to block this cycle, shown in *part B*, Fig. 2. This cycle is also an SCC. More details can be found in [11]. When any search finds an SCC, all states in SCC should be marked as completed to activate the Searches which are suspended.

The difference between our work and Lowe's is that we expand the concurrent SCC detection for automata-based LTL model checking with fairness assumption based on the concurrent Tarjan's algorithm. We support the concurrent on-the-fly SCC detection, in which the state space is unknown in advance. Although Lowe mentioned this mode (rooted mode) in both [11] and [12], he doesn't supply details or experiments. We also support the parallel on-the-fly fairness checking.

III. Concurrent on-the-fly SCC detection for Model Checking Under Fairness Assumption

The core in the emptiness checking of automata-based model checking is SCC detection. Thus, our approach is based on the concurrent Tarjan's algorithm (Section II-C). Our target is to expand it to construct a concurrent version for the on-the-fly generated state space. To this end, we first present

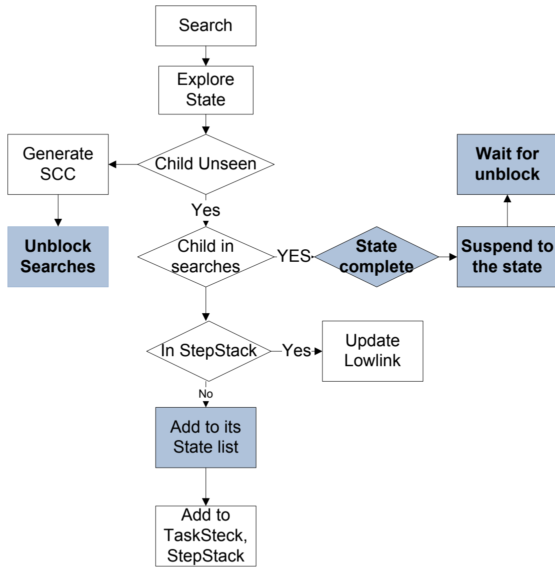

Figure 1. Execution Process in One Iteration

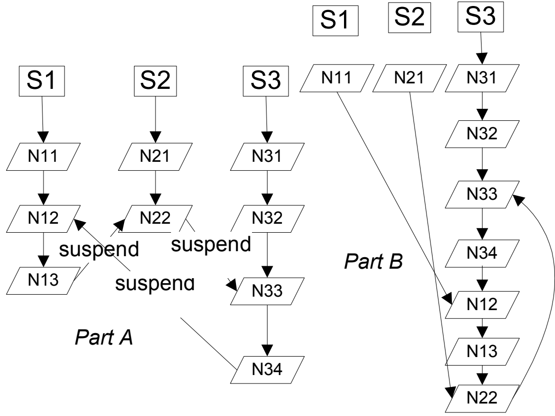

Figure 2. Blocking Cycle

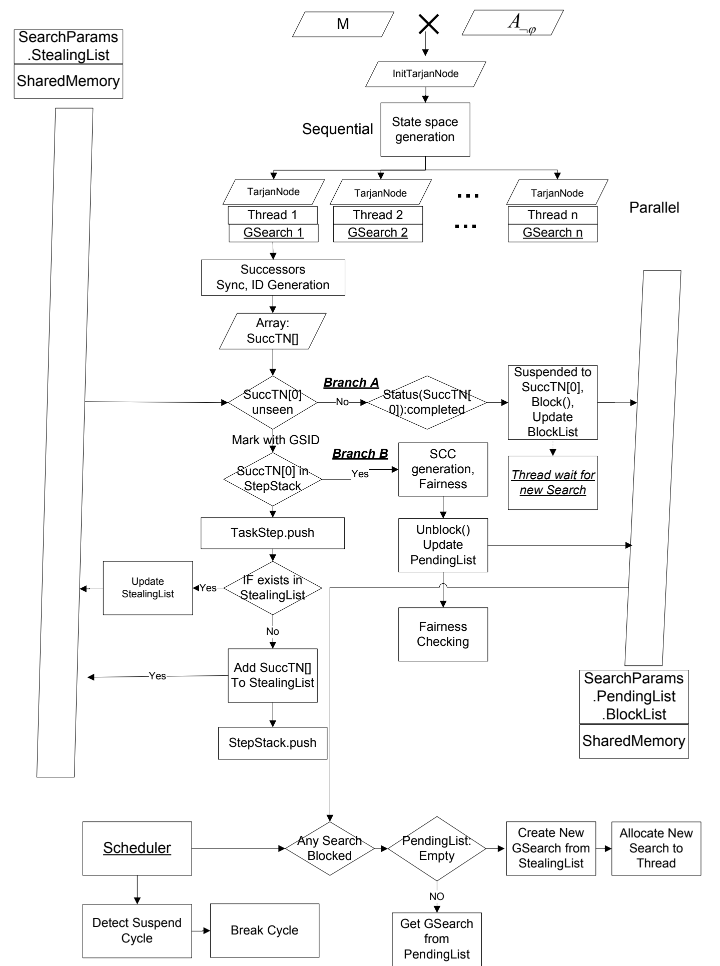

Figure 3. Overall Process

several key challenges, which indicate the efforts on making the concurrent Tarjan's algorithm feasible for automata-based model checking with on-the-fly generated state space: (1) In this context, the state space is generated by the composition of transition system and büchi automaton. Thus, the state space is unknown in advance, and the simple predefined ID for each state in the concurrent Tarjan's algorithm is not available. (2) In concurrent Tarjan's algorithm, if a state is explored in a search, there is no duplicate state being explored in any other searches since the state space is known in advance. However, duplicate elimination is an important problem for concurrent on-the-fly generated state space. If there are duplicate states, the judgment on the suspension of a search cannot be made correctly. (3) Preservation of data consistency in the synchronization process is another challenge for concurrent SCC detection on on-the-fly generated state space, which require upgrading of the execution process and data structures.

In this section, we present how our approach solves the above challenges in detail.

A. Overview of our approach

We introduce the overall execution process of our approach in Fig. 3. For readability, the figure shows only the main procedure of our approach. We present the complete algorithm later in this section.

First, we describe some essential concepts in our approach: (1) We define *GSearch* as our scheduling unit. Different than *Search* in Fig. 1, GSearch is a combination of state space generation and exploration. Each *GSearch* has a *GSID* to identify itself. Each *GSearch* has its own *TaskStack* and *StepStack*. A *GSearch* also needs to record the status of itself, e.g., whether the *GSearch* is suspended to any state. (2) A *Thread* is the physical execution unit. A thread can handle multiple *GSearches* based on scheduling. (3) We define the *TarjanNode* to indicate the state generated on the fly by GSearch, which is a meta-state. In *TarjanNode*, we define the *SID* and *GSID*. *SID* is the identification of *TarjanNode* in global state space, which is constructed by the string concatenation of $S_1S_2...S_nS_A$. *GSID* represents the *GSearch* that explores this *TarjanNode*. We also define the *status*, *suspendlist*, which refer to the concurrent Tarjan's algorithm. *enableEvt* and *ParticipatingProcesses* are defined for the fairness checking. (4) We define *SearchParams* as the shared space among all *GSearches*. It consists of *StealingList*, *PendingList*, *BlockList*, *SCCList*, *SuspendMap*

and the generated state space *OGTrans*. *StealingList* is to store all the generated *TarjanNodes*. It is a unique hash structure for duplicate elimination since *TarjanNodes* are generated concurrently in all GSearches. *BlockList* stores all *GSearches* that have been suspended. *PendingList* stores the resumed *GSearch*, which is waiting for *Scheduler* to allocate the free thread for it. *SuspendMap* stores the suspended relationship. e.g., $GSearch_1$ is suspended to $TarjanNode_2$. *visited* set is also contained in *SearchParams*, which indicates the states that have been expanded. (5) We integrate the *Scheduler* from Lowe. It takes the charge of creating new *GSearch* and assigning it to a free thread. A thread is regarded as free if the *GSearch* it handles is suspended to any state. If there is no GSearch in PendingList, new *GSearches* are created by assigning new initial states. The *GSID* is maintained by *Scheduler*.

Within the multi-core environment, we start multiple *GSearches* for each thread at the beginning. Thus, we can see in Fig. 3, the startup sequential state space generation process is to generate an initial set of states for each *GSearch*. This is a difference from the concurrent Tarjan's algorithm with known state space. Fig. 3 shows the execution process of one *GSearch*. With initial *TarjanNode*, *GSearch* initiates the *preorder* for it and generates the successors by the parallel composition. The synchronization operation is needed if synchronized events exist. The generated successors should be transferred to the array of *TarjanNode SuccTN[]*. We initilize the *SID* and other variants in *TarjanNode*. *SuccTN[0]*[1] is the next *TarjanNode* to be expanded based on the rule of DFS. Thus, $GSearch_1$ should judge whether *SuccTN[0]* has been seen in other *GSearches* by accessing the *StealingList*.If not, *SuccTN[0]* is marked with $GSearch_1$'s *GSID*. If *SuccTN[0]* is not in *StepStack*, it is pushed to the *TaskStack*. Only *SuccTN[0]* belongs to $GSearch_1$. Other *TarjanNodes* in *SuccTN[]* should be directly transferred to *StealingList*. Duplicate elimination is handled with the unique *SID*. It should be noted that when a *GSearch* writes its *GSID* to *SuccTN[0]*, it should also update the *StealingList* if *SuccTN[0]* exists in *StealingList* without the mark of any *GSID*. Then the initial *TarjanNode* should also be pushed to *StepStack*.

Branch A in Figure 3 presents that if $GSearch_1$ detects that *SuccTN[0]* is marked with other *GSearch*'s *GSID*, it checks the status of *SuccTN[0]*. If *SuccTN[0]* is not completed, $GSearch_1$ is suspended to *SuccTN[0]*. It updates the *SuspendMap* and the *BlockList*, and detects if a blocking cycle exists. If this is the case, it blocks the cycle by transferring all related *TarjanNode* to another one *GSearch* (refer to Fig. 2 in Section II-C). $GSearch_1$ becomes blocked and waits to resume. When a blocked *GSearch* is resumed, it is transferred to *PendingList*. *Scheduler* works on two tasks: (1) When a *GSearch* is suspended and waiting for resuming, the corresponding thread becomes free. If the *PendingList* is empty, it creates a new *GSearch* for the free thread. 2) If the *PendingList* is not empty, it gets *GSearch* from *PendingList* and pushes it to the free thread.

Branch B in Figure 3 is the common *lowlink* updating process and the condition that a cycle being detected. In our approach, the fairness checking starts concurrently with the generation of the complete SCC. After the generation of the complete SCC, the suspended relationship should be updated. All *GSearches* which are suspended to *TarjanNode* in the SCC are resumed and continue their exploration.

[1]here we give the example in the first iteration, hence we regard *SuccTN[0]* as unexpanded

B. Data distribution

The data distribution in our approach for each search consists of two parts: startup distribution and runtime distribution.

Firstly, in the on-the-fly LTL model checking, the state space is not known in advance. It is generated during the product between M and $\mathcal{B}_{\neg\varphi}$. M is also the parallel composition of component LTSs. So at the beginning, we always just have one initial state. Based on Section II-C, in that algorithm, fixed number of threads are started for concurrent searches at the beginning, which start on different states to follow different traces to increase the possibility to find SCC. For our approach, we should also start multiple *GSearches* for high efficiency. The difference is that for on-the-fly generated state space, we first start the state space generation for several iterations to generate partial state space. Then we base on the number of cores in the working machine to start the corresponding number of threads and *GSearches*. So at the beginning, the parallelism can be fully utilized.

Secondly, during the concurrent search process in our approach, the data distribution is based on the state space generation. When *GSearch* is started, each *GSearch* marks its startup state with its *GSID*. Then each *GSearch* stores all generated successor states in its *Stepstack* and *DFSstack*. It should be noticed as the state space generation is a concurrent process in many threads, it is impossible to config the global ID to identify each successor state as [11] does. In our approach, we mention in Section II-A that each state is indeed composed of many states from component LTSs and the automaton, so we can use a *string* to identify the generated states, which is the montage of each states' ID in the original component LTS or the Büchi Automaton. As *GSearch* works on a DFS process, each *GSearch* only marks the front state in the *DFSstack* with its *GSID*, then all successors are copied to the *StealingList* and the outgoing relations are copied back to the table *OutgoingTrans*. The key point is that before marking any successor state, the *GSearch* should visit the *StealingList* to check if this successor state is visited or marked with other *GSID*, and then decide if the search should continue, backtrack or suspend.

C. Concurrent State Space Generation

In our approach, the state space is generated concurrently by all active *GSearches*. Each *GSsearch* handles the successor generation independently. During the successor generation, event synchronization operations are involved in both the parallel interleaving of M_i and the composition between M and $\mathcal{B}_{\neg\varphi}$. We allocate a private array for each *GSearch* to temporarily store the generated successors. Event synchronization also works on this array. When we convert each successor state (s_i, b_i) to *TarjanNode*, we need to explore all s_i to fill up the information for fairness checking. These are all independently handled by each *GSearch*. The key point is, the state space generation in the concurrent environment refers to a lot of operations on shared space. e.g., the *SearchParam*. Thus, it is important to handle the concurrent access to prevent any data inconsistency.

In order to ensure the consistency of shared data, *lock* is used throughout the program. In general, each shared variable should have a separate lock to ensure that at one time, only one *GSearch* can access it. *SearchParams*, as we have mentioned in Section III-A, is the structure to store shared variables among *GSearches*. In order to ensure synchronization[2], most collection variables in *SearchParams* are created as the concurrent collection data type, which is provided by C#.NET. For other shared variables, the *SearchParams* has static locks to ensure their synchronization. In order to ensure synchronization and correctness of the program, we present some details of locks below, which indicate how errors occur without these locks.

- *TarjanNode.visitLock*: A *TarjanNode* is a shared variable. During the exploration of the state space, each *GSearch* checks the *TarjanNode* via the following steps: (a) Checks the status. (b) Checks whether it is the first unexpanded successor in *GSearch*. If yes, the *GSearch* visits the node. These steps are supposed to be atomic. Otherwise, two *GSearches* may take the same *TarjanNode* and cause errors, as shown in Fig. 4. Two *GSearches* are handled by two threads. The first *GSearch* is taking the *TarjanNode* and hasn't finished, while the other *GSearch* is also checking the ownership of the node. Two GSearches take the same node and none of them being suspended. Thus, *visitedlock* is necessary to lock these three steps.

- *Suspendmap.Lockitself*: The updating and accessing of the *SuspendedMap* should be synchronized, otherwise blocking cycle may be missed. For example, in Fig. 5, one *GSearch* is checking whether blocking cycle existed or not. When it gets the conclusion that no cycle exists, and it has not added new suspension into the map, other *GSearch* starts to check the *SuspendedMap* to get the path. Both of the *GSearches* think that there is no blocking cycle and add the suspension into the map. At this condition, blocking cycle occurs but no one handles it. Then deadlock occurs.

- *TarjanNode.block() & TarjanNode.unblock()*: Two or more *GSearches* may become blocked on the same *TarjanNode* concurrently. Moreover, the *TarjanNode* may be detected in an SCC at the same time. Shown in Fig. 6, when the first *GSearch* is updating the status of the *TarjanNode* but not yet finished, the second *GSearch* is checking *TarjanNode*'s *GSID*. The second *GSearch* thinks that the *TarjanNode* is incomplete and starts to block itself on this node. However, the first *GSearch* sets the node as complete and unblocks all blocked searches on this node. In this situation, the status of this node has already been completed and cannot go through the unblocking process again. The first *GSearch* cannot get a chance to be active again.

D. On-the-fly Parallel Fairness Verification

We introduce the definition of fairness assumption and our major efforts on on-the-fly parallel fairness checking in previous sections. Our fairness checking is based on the exploration of SCCs. When the concurrent LTL model checking algorithm detects the existence of an SCC, it needs to generate

[2]Different from event synchronization, this represents the synchronization of data.

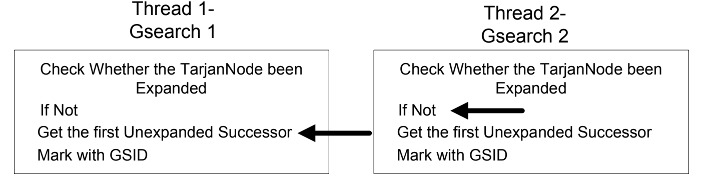

Figure 4. TarjanNode Synchronization

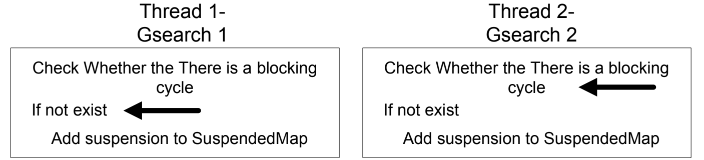

Figure 5. SuspendedMap Synchronization

the complete SCC. Shown in Fig 7, our key idea is to overlap part of the fairness checking process with the SCC generation process and do on-the-fly parallel checking so as to increase the performance. For instance, we overlap the generation of the sets *enabledEvt*, *enabledPro*, *engagedEvt* and *engagedPro* which base on the complete exploration of the SCC. Then we do parallel exploration to deal with these sets to generate the verification result. If any thread finds the condition that makes the SCC fair/unfair, the checking process terminates. In specific, Given an SCC Scc, we present the detail design for *ESF/PSF* as a sample. We also specify the *SGF*.

Sequential Algorithm: The algorithm explores every component s_i in Scc. For each state, the algorithm gets its engaged event list and add to the *engagedEvt/Pro(Scc)*. Then it explores each component again to find which component has any event $\alpha \in enabledEvt/Pro(s_i)$ but $\notin engagedEvt/Pro(Scc)$. These states are regarded as bad states. Then the algorithm removes these bad states from Scc to get Scc', and calls the modified Tarjan's algorithm again to check whether there is an SCC in the Scc'. If so, Scc is fair. Or it is not fair.

Parallel On-the-fly Approach: engagedEvt/Pro(Scc) are generated during the generation of Scc. Bad states are generated and removed from the SCC concurrently with the *lock*. Each thread takes charge of several components of Scc. The new SCC is used to call the modified Tarjan's Model Checking algorithm. The modified Tarjan based model checking algorithm is not the concurrent version mentioned in Section II-B because the overhead generated by concurrency may be more expensive than the advantages provided as the size of a single SCC may be limited.

For SGF, the sequential algorithm explores every component s_i in Scc. When s_i is being explored, it generates the $enabledEvt(s_i)$, in which it contains a set of event ID. Then the algorithm explores all successors of s_i. For successors that also $\in Scc$, in the corresponding events that leads to the transition to these successors, if the ID of any event $eid \in enabledEvt(s_i)$, Scc is fair. For SGF, there is no need to generate a set that based on the complete exploration of Scc, so this algorithm doesn't contain the on-the-fly part. In the parallel algorithm, shared memory is used to store Scc, and a fixed number of threads start to check all components. Checked

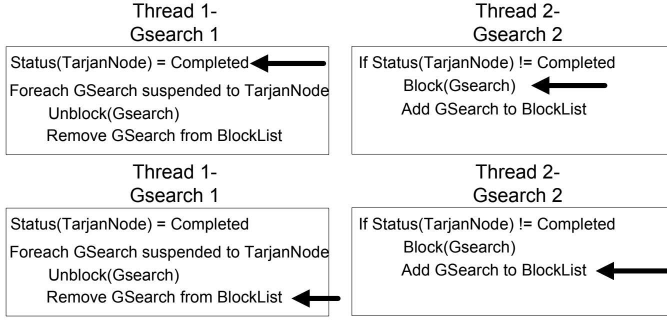

Figure 6. Block and Unblock GSearch Synchronization

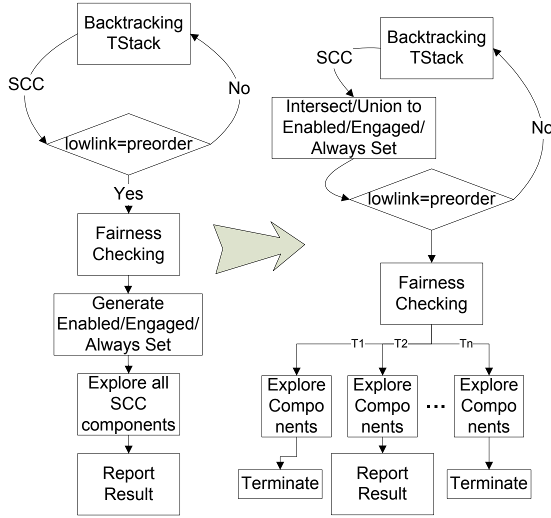

Figure 7. Parallel On-the-fly Fairness Checking

components are marked. If any thread finds the component that makes Scc fair, it broadcasts to other threads and all thread terminate.

E. Algorithm

In this section, we present the algorithm of the concurrent on-the-fly SCC detection for LTL model checking with fairness assumption synthesizing our descriptions in the preceding sections. They are shown in Algorithm 2, Algorithm 3 and List. 1.

We describe the startup of our approach in Algorithm 2. *SearchParams* is defined in line 1. Line 2 is a *Sequential* state space generation processThis process generates some states which are stored in the *StealingList*. It should be guaranteed that no duplicates exists. *Scheduler* starts in Line 3 to create *GSearch*. The number of threads is same as the initial number of *GSearches*. All *GSearches* work concurrently.

We present the algorithm of *GSearch* in Algorithm 3 and List 1, which is the key algorithm of our approach. All operations are based on *TarjanNode* instead of the original state. In Algorithm 3, all *GSearches* access the shared memory to visit M, $\mathcal{A}_{\neg\varphi}$ and *SearchParams*. All key differences compared to sequential algorithm are shown with *underline*. Each *GSearch* has its own *TaskStack* and *StepStack*. In line 1, *GSearch* gets its initial state and converts it to a initial *TarjanNode*. The iteration for SCC detection starts in line 3. Line 4 is to set the

Algorithm 2: Startup of Concurrent on-the-fly SCC detection for Automata-based Model Checking

Input: M, $\mathcal{A}_{\neg\varphi}$
1. Define Shared *SearchParams*, Local *StepStack*,*TaskStack*;
2. Duplicate Elimination:Sequential LTL Model Checking $\rightarrow$ $SearchParams.StealingList$;
3. Start a thread for *Scheduler*; *Scheduler*: create *GSearch(i)* $\leftarrow SearchParams.StealingList[i]$;
4. Start n threads;
5. *Scheduler*: allocate *GSearches* to all threads to work concurrently;

Algorithm 3: GSearch-Concurrent on-the-fly SCC detection for Automata-based Model Checking

Input: M, $\mathcal{A}_{\neg\varphi}$, *SearchParams*
1. $T = TarjanNode(SearchParams.StealingList[GSID])$;
2. $TaskStack.push(T \leftarrow Mark(T, GSID))$;
3. **while** $\neg\, TaskStack.Empty$ **do**
4. Get T, Update $T.preorder$; $done = true$;
5. **if** $T \in SearchParams.visited$ **then**
6. $T'[] = SearchParams.visited[T]$;
7. **forall the** $T_i \in T'[]$ **do**
8. **if** $T_i.preorder = false$ **then**
9. Module:Add New Task;
10. **else**
11. $succTN[] = T.MakeOneMove(M, \mathcal{A}_{\neg\varphi})$;
12. **forall the** $T'_i \in succTN[]$ **do**
13. lock and AvoidDup: $SearchParams.OGTrans[T].add(T'_i)$;
14. lock: Sync T'_i with $SearchParams.StealingList$;
15. **if** $T'_i.preorder = false$ **then**
16. Module:Add New Task;
17. $SearchParams.visited.add(T)$;
18. **if** $done$ **then**
19. update $lowlink$;
20. **if** $lowlink == preorder$ **then**
21. $SCCSet.add(T)$;
22. backtrack $Stepstack \rightarrow SCCSet\&Fairness$;
23. $SearchParams.SCCList.Add(SCCSet)$;
24. $T'_i.complete = true$;
25. Resume $GSearches$ suspended to $T'_i \in SCCSet$;
26. lock: Update $SearchParams.PendingList$;
27. Option: Concurrent On-the-fly Fairness Checking;
28. **if** $ISFair$ **then**
29. Record Result; Generate Counterexample;
30. **forall the** $T'_i \in SCCSet$ **do**
31. lock:$SearchParams.visited.remove(T_i)$;
32. lock:$SearchParams.OGTrans.remove(T_i)$;
33. **else**
34. $StepStack.push(T)$;

local $preorder$ of *TarjanNode* in a certain *GSearch*. Line 5 and 10 lead to two conditions shown below:

(1) *From line 5 to 9*. If the *TarjanNode* T is in *SearchParams.visited*, it has been expanded. Then in lines 6 to 9, *GSearch* gets and chooses the available successor to push into *TaskStack*, which is handled in the module *Add New Task*. We present that in List. 1. We judge whether the successor *TarjanNode* belongs to other *GSearches* in line 1. If not, we just mark the first available successor with *GSID* and push it into *TaskStack* in line 3 to 6. If the successor already exists in *SearchParams.StealingList* but not been occupied by any *GSearch*, we should update its *GSID*. If the successor belongs to another *GSearch* and hasn't being completed, *GSearch* is suspended to this successor in lines 11 to 14. It updates the *SuspendedMap* and detects whether this operation generates a

blocking cycle. If so, it breaks this cycle, refers to Fig. 2 in Section II.

(2) *From line 10 to 17*. If the visiting *TarjanNode* T is not expanded, *GSearch* generates successors based on T by the interleaving between M and $\mathcal{A}_{\neg\varphi}$ (line 11). It updates the *OGTrans[T]* without duplication (line 13). This operation should use $lock$ to avoid conflicts. For each *TarjanNode* in *SuccTN[]*, *GSearch* synchronizes the value of it with the data in *StealingList* (line 14). It checks whether the *TarjanNode* has been expanded before or been occupied by other *GSearch*. If it is a new *TarjanNode* without being marked with any *GSID*, the module *Add New Task* works as mentioned. Line 17 marks T as expanded. During the process from line 5 to 17, if all successors of T have been expanded, it comes to line 19. It is to update the *lowlink* of T and is same to the sequential algorithm.

GSearch generates an SCC in lines 20 to 26. The status of all *TarjanNodes* in SCC is marked as $completed$. All *GSearches* that being suspended to these *TarjanNodes* are resumed to be ready for scheduling. If fairness checking is required, the concurrent on-the-fly fairness checking works in line 27. It should be noted that in line 22, the fairness checking has started and overlaps with the generation of SCC. This is part of our concurrent on-the-fly fairness checking, which can improve the performance of fairness checking. If the detected SCC satisfies the fairness assumption, on-the-fly LTL model checking enters the counterexample generation process. Line 31 and 32 work as we always need to generate all SCCs. In line 34, if the SCC is not detected, *GSearch* pushes T to the local $StepStack$.

Listing 1. Add New Task
```
1  if(T_i.GSID = -1){
2      if(done){
3          T_i.mark(GSID);
4          lock: Update SearchParams.StealingList;
5          TaskStack.push(T_i);
6          done = false; break;
7      }
8  }
9  else{
10     if{T_i.status ≠ completed}{
11         lock: Update SearchParams.SuspendedMap;
12         Check Block Cycle, lock:break cycle;
13         lock: Update SearchParams.BlockList;
14         Wait for Resume;
15     }
16 }
```

F. Complexity

In order to compare with sequential Tarjan's algorithm and Lowe's concurrent version with unrooted mode, We discuss the complexity of concurrent on-the-fly SCC detection without the complexity of state space generation. Given a transition system with N nodes and E edges, the sequential Tarjan's algorithm shows the complexity of $O(N+E)$ and Lowe shows the complexity of concurrent Tarjan's algorithm is $O(N^2+E)$. Our concurrent on-the-fly SCC detection has extra node transfer for the construction of global outgoing transition relationships and the StealingList mentioned before. The complexity of these part is $O(N)$ since we use lock and hash table to avoid duplicates. Thus, the complexity of our concurrent on-the-fly SCC detection without the complexity of state space generation is also $O(N^2+E)$.

Table I. EVALUATION OF LTL MODEL CHECKING (TIME IN SEC)

Model	Proc	$\lvert S\rvert$	$\lvert T\rvert$	$\lvert SCCs\rvert$	PAT-O	PAT-C	SP
MsS	1000	$9*10^3$	$1.8*10^4$	0	222.6	207.3	1.1
MsS	1200	$2.6*10^4$	$5.3*10^4$	0	611.9	523.4	1.2
MsS	1500	$3.3*10^4$	$6.7*10^4$	0	1104	626	1.76
CC	4	$1.5*10^4$	$7*10^4$	$5.2*10^3$	1.05	1.57	0.67
CC	5	$5*10^5$	$3.3*10^6$	$1.6*10^5$	49.7	31.9	1.6
ABP	100	$2.1*10^5$	$7*10^5$	1	10.8	23.2	0.5
ABP	200	$8*10^5$	$2.7*10^6$	1	53	92	0.58
ABP	300	$1.8*10^6$	$6*10^6$	1	150.2	247	0.61
Lift.1	3,2	$3.2*10^5$	$7.2*10^5$	1349	39.6	24.7	1.6
Lift.2	3,2	$3.8*10^5$	$8.7*10^5$	1569	45.8	33.4	1.4
Lift.3	3,2	$4.6*10^5$	$1.5*10^6$	983	55.6	32.6	1.7

IV. IMPLEMENTATION AND EVALUATION

We implement our approach using $C\#$ in Process Analysis Toolkit (PAT) [17]. We call it *PAT-C*. We evaluate the performance of *PAT-C* by comparing it to the original PAT with sequential on-the-fly LTL model checking and fairness checking, which we call *PAT-O* and *PAT-OF*. We conduct our experiments on a laptop with Intel(R) Xeon(R) CPU E5-1650, 3.2GHZ, 12 cores, 16GB RAM.

In our experiments, $|S|$, $|T|$ and $|SCCs|$ separately represent the number of states, the number of transitions and the number of SCCs. $LSCC$ and $SSCC$ separately represent the size of the largest SCC and smallest SCC. *SP* means the speedup. For the testing models, *MsS* represents the *Miners Scheduler*. *CC* represents *Consensus with Crashes*. *DP* represents *Dining Philosophers*. *ABP* represents *Alternation Bit Protocol*. *KvR.1/2* represents *K-valued Register.12*. *MLS.12* represents *Multiple Lift Sysmtem 12*. *TBM* represents *DBM Testing*.

A. Performance Evaluation on SCC Detection for Model Checking

We conduct the performance evaluation on multiple LTS models on a range of state space. We initialize 12 *GSearches*. We start 12 threads in parallel to handle these *GSearches* concurrently since our machine owns 12 cores. It should be noted that in order to completely evaluate the performance of our approach, in our experiments we find all SCCs instead of just one. We involve four models in this part: *MsS*, *CC*, *ABP* and *Lift*. Our experimental results are shown in Table I: (1) The product between *MsS* model and the Büchi Automaton generates no SCCs. Hence, our approach works as a concurrent complete state space generation process. We use 1000, 1200 and 1500 processes for our experiments. From Table I, we can observe: (1) Our approach gains up to 2X speedup compared to the sequential LTL model checking, which is significant since the original execution cost is fairly large. 2) The product between *CC*&*Lift.1/2/3* model and the Büchi Automaton can generate a large number of small SCCs. Results in Table I show that with a large number of small SCCs, our approach can also gain around 2X speedup compared to the sequential LTL model checking in PAT. 3) The product between *ABP* model and the Büchi Automaton generates just one large size SCC. Results in Table I show that the performance with these kind of models is expected to be even slower than the sequential LTL model checking in PAT.

In conclusion, our approach can generally improve the performance of automata-based LTL model checking. The performance is better for larger state spaces. The performance

Table II. TESTING MODEL FOR FAIRNESS CHECKING

Model	Proc	$\|S\|$	$\|T\|$	$\|SCCs\|$	LSCC	SSCC
DP	8	$1.4 * 10^5$	$1.08 * 10^6$	51	$3 * 10^4$	1
DP	9	$5.3 * 10^5$	$4.5 * 10^6$	66	$1.1 * 10^5$	1
DP	10	$1.9 * 10^6$	$1.8 * 10^7$	83	$4.2 * 10^5$	1
Peterson	4	$5.1 * 10^4$	$2.1 * 10^5$	5	$4.5 * 10^4$	1
Peterson	5	$1.4 * 10^6$	$7.2 * 10^6$	17	$1.3 * 10^6$	1
CC	4	$1.2 * 10^4$	$5.4 * 10^4$	$3.8 * 10^3$	2	1
CC	5	$3.8 * 10^5$	$2.5 * 10^6$	$1.2 * 10^5$	2	1
KvR.2	4,3	$6.2 * 10^5$	$2.5 * 10^6$	1042	$4.1 * 10^5$	104
KvR.2	5,3	$3.7 * 10^6$	$1.5 * 10^7$	2748	$2.3 * 10^6$	256
KvR.2	4,4	$3.8 * 10^6$	$1.8 * 10^7$	6342	$2.4 * 10^6$	104
MLS.1	2,2,3	$3.8 * 10^5$	$1.9 * 10^6$	940	$2.7 * 10^5$	4
MLS.1	3,2,2	$1.2 * 10^6$	$5.1 * 10^6$	1416	$9.8 * 10^5$	4
MLS.1	2,2,4	$1.5 * 10^6$	$9.3 * 10^6$	3243	$1.1 * 10^6$	4
MLS.2	2,2,3	$6.5 * 10^5$	$3.5 * 10^6$	3029	$3.5 * 10^5$	1
MLS.2	3,2,2	$2.2 * 10^6$	$9.5 * 10^6$	5457	$1.5 * 10^6$	1
MLS.2	2,2,4	$2.7 * 10^6$	$1.6 * 10^7$	8345	$1.4 * 10^6$	1
DBM	2,4	$4.2 * 10^4$	$1.7 * 10^6$	1	$1.4 * 10^4$	-
DBM	2,5	$9.5 * 10^4$	$4.5 * 10^6$	1	$3.2 * 10^4$	-
DBM	2,6	$1.9 * 10^5$	$1.1 * 10^7$	1	$6.3 * 10^4$	-
PMC	50,1000	$1 * 10^6$	$2 * 10^6$	51	$2 * 10^4$	$2 * 10^4$
PMC	60,1000	$1.2 * 10^6$	$2.4 * 10^6$	61	$2 * 10^4$	$2 * 10^4$
PMC	80,1000	$1.6 * 10^6$	$3.2 * 10^6$	81	$2 * 10^4$	$2 * 10^4$

Table III. EVALUATION OF ESF&EWF CHECKING (TIME IN MS)

Model	Proc	ESF Checking			EWF Checking		
		PAT-OF	PAF-C	SP	PAT-OF	PAF-C	SP
DP	8	$1.1/1.5 * 10^3$	48/304	25/5	-	-	-
DP	9	$1.5/1.6 * 10^4$	81/1218	188/13	-	-	-
Peterson	4	29	30	0.97	-	-	-
Peterson	5	$1.1 * 10^3$	267	4.3	-	-	-
CC	4	3	3	1	3	95	—
CC	5	137	157	0.87	157	3415	—
KvR.2	4,3	-	-	-	119	4	29.7
KvR.2	5,3	-	-	-	716	16	44.8
KvR.2	4,4	-	-	-	750	38	19.7

Table IV. EVALUATION OF PSF&PWF CHECKING (TIME IN MS)

Model	Proc	PWF Checking			PSF Checking		
		PAT-OF	PAF-C	SP	PAT-OF	PAF-C	SP
DP	8	50	1	50	1327/1705	14/364	95/5
DP	9	224	1	224	$1.7/1.8 * 10^4$	48/1497	364/11
MLS.1	2,2,3	153/1728	4/1270	38/1.4	187	71	2.6
MLS.1	3,2,2	892/6100	3/5100	298/1.2	617	161	3.8
MLS.1	2,2,4	733/7256	9/5453	81/1.35	819	288	2.9
MLS.2	2,2,3	257/2797	4/2019	64/1.3	340	219	1.6
MLS.2	3,2,2	1432/12601	19/10613	77/1.2	$1.1 * 10^3$	475	2.4
MLS.2	2,2,4	1290/13382	16/11800	80/1.2	$1.4 * 10^3$	833	1.7

decreases at *ABP* model since it has only one large SCC in the state space. Thus, based on our approach, all *GSearches* easily encounter *TarjanNode* in other *GSearches*. According to the cycle breaking rules, our approach works like a sequential algorithm as all data is transferred to one *GSearch*. The parallelism cannot be well utilized and a lot of time is consumed at transferring nodes. Finally the concurrent on-the-fly model checking is more suitable for the models with a large amount SCCs and trivial average SCC size so as to reduce the cost of suspending, which is also indicated in [11].

B. Performance of On-the-fly Parallel Fairness Verification

We conduct the experiments on fairness checking using different models with different SCC numbers and SCC sizes. For each type of fairness, we choose three models and we perform the experiments independently. The features of our test models are shown in Table II. These models differ in the number and size of SCCs, which can help reflect the features of our approach. In this part, the parallelism started is less then 11 threads, which is adjusted dynamically based on the scheduling of the Microsoft .Net platform.

In our experiments, we measure the time cost in milliseconds. Firstly, we compare our approach to the sequential fairness checking in PAT. The results of *ESF*, *EWF*, *PWF* and *PSF* checking is shown in Table III. We can see that our approach for fairness checking can gain significant speedup for most models[3]. The performance improvement is more visible when the state space consists of a lot of SCCs and the average SCC size is large. A performance decrease is observed at the *CC* model because all SCCs in the state space are very small, with the size of 1 or 2. For *Peterson*, where the state space consists of just 5 SCCs, the performance is also decreased. Under this condition, the overhead of parallelism is more dominant than the advantage of parallelism. Secondly, as we mentioned in Section III-E, our approach generates the sets for fairness checking together with the SCC generation, so it introduces costs to the SCC generation process. We count the cost of the entire SCC generation and Fairness checking for some models, which is shown in the right side of "/" in Table III. We can see that our approach can still gain significant speedup. The reason why the speedup is not as high as the results in the left side of "/" is that the SCC generation cost a lot compared with the fairness checking. It can also be concluded that the parallelization of fairness checking can bring more performance improvements for weak fairness type than strong fairness type. The reason is that the weak fairness checking does not have to update any shared variables(bad states), therefore *lock* operation is not necessary. In contrast, in strong fairness checking, a *lock* exists. During the execution, some threads need to wait for other threads for the right to access.

The results of SGF checking is shown in Table V. For SGF checking, as we mentioned in Section III-E, the complete checking process works in parallel after the generation of SCC. Hence, we just record the fairness checking time and compare it with the sequential version. We can see besides the *CC* model with very small SCC, our approach gains around 5X speedup for the *DBM* and around 4X speedup for the *PMC*.

Finally, we compare our experimental results with the previous results of Liu, Sun and Dong [10], which also works on the parallel on-the-fly LTL model checking in PAT. In [10], the major Tarjan process is taken in one thread and SCC generation is taken in parallel by forking new threads to handle it. The performance of both the approach in [10] and this paper depends on the ratio of SCCs and the average SCC size. The differences are that the on-the-fly LTL model checking in [10] works better than sequential algorithm only with large number of large size SCCs. Our approach can also gain performance improvement without any SCC, which is not possible for the approach in [10]. The large number of small size SCCs may influence the fairness checking process in both of these approaches. But for large sized SCCs, the approach shown in this paper may have better performance improvement than PAT as we start the exploration from different directions and the process is on-the-fly. In conclusion, the approach in this paper can be suitable for much more types of models and gain more performance improvement than the approach in [10].

V. CONCLUSION AND FUTURE WORK

In this paper, we expanded the concurrent Tarjan's algorithm and developed a concurrent on-the-fly SCC detection for

[3]Here all models represent the state space generated from the product between the model and the Büchi Automaton.

Table V. SGF CHECKING (TIME IN MS)

Model	Proc	PAT-OF	PAT-C	SP
DBM	2,4	$1.7 * 10^3$	353	4.8
DBM	2,5	$5.4 * 10^3$	$1.1 * 10^3$	4.95
DBM	2,6	$1.5 * 10^4$	$2.9 * 10^3$	5.1
PMC	50,1000	$1.1 * 10^3$	370	3.2
PMC	60,1000	$1.7 * 10^3$	453	3.7
PMC	80,1000	$1.9 * 10^3$	605	3.1
CC	4	1	83	-
CC	5	77	$2.7 * 10^3$	-

automata-based (LTL) model checking with fairness checking. To this end, we built a novel abstract data structure for concurrent LTL model checking taking data consistency into account. In addition, we developed a parallel on-the-fly fairness checking approach for different types of fairness assumptions. We also implemented our proposed approach in the practical model checker PAT. Our experiments show that our approach achieves significant speedup comparing to the sequential version of PAT. In the future, we plan to transfer our approach work to many-core (GPU) platform, which support massive parallelism and promises to accelerate the performance of model checking.

REFERENCES

[1] W. Anton and B. Dragan. GPUexplore: Many-Core On-the-Fly State Space Exploration Using GPUs. In *TACAS*, pages 233–247. 2014.

[2] C. Baier, J.-P. Katoen, et al. *Principles of model checking*, volume 26202649. MIT press Cambridge, 2008.

[3] V. Bloemen, A. Laarman, and J. van de Pol. Multi-core on-the-fly SCC decomposition. In *PPoPP*, page 8. ACM, 2016.

[4] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani. Model checking and the state explosion problem. In *Tools for Practical Software Verification*, pages 1–30. 2012.

[5] J.-M. Couvreur. On-the-fly verification of linear temporal logic. In *FM99łFormal Methods*, pages 253–271. Springer, 1999.

[6] S. Evangelista, L. M. Kristensen, and L. Petrucci. Multi-threaded explicit state space exploration with state reconstruction. In *ATVA*, pages 208–223. Springer, 2013.

[7] S. Evangelista, A. Laarman, L. Petrucci, and J. Van De Pol. Improved multi-core nested depth-first search. In *ATVA*, pages 269–283. Springer, 2012.

[8] S. Evangelista, L. Petrucci, and S. Youcef. Parallel nested depth-first searches for LTL model checking. In *ATVA*, pages 381–396. Springer, 2011.

[9] L. Lamport. Proving the correctness of multiprocess programs. *TSE*, (2):125–143, 1977.

[10] Y. Liu, J. Sun, and J. S. Dong. Scalable multi-core model checking fairness enhanced systems. In *Formal Methods and Software Engineering*, pages 426–445. Springer, 2009.

[11] G. Lowe. Concurrent Depth-First Search Algorithms. In *TACAS*, pages 202–216. 2014.

[12] G. Lowe. Concurrent depth-first search algorithms based on Tarjans Algorithm. *STTT*, 18(2):129–147, 2016.

[13] J. Pang, Z. Luo, and Y. Deng. On automatic verification of self-stabilizing population protocols. *Frontiers of Computer Science in China*, 2(4):357–367, 2008.

[14] A. Puhakka and A. Valmari. Liveness and fairness in process-algebraic verification. In *CONCUR*, pages 202–217. Springer, 2001.

[15] E. Renault, A. Duret-Lutz, F. Kordon, and D. Poitrenaud. Parallel explicit model checking for generalized Büchi automata. In *TACAS*, pages 613–627. Springer, 2015.

[16] J. Sun, Y. Liu, J. S. Dong, and J. Pang. Towards a Toolkit for Flexible and Efficient Verification under Fairness. Technical report, Technical Report TRB2/09, National Univ. of Singapore, 2008.

[17] J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards Flexible Verification under Fairness. In *CAV*, pages 709–714, 2009.

[18] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[19] Z. Wu, Y. Liu, Y. Liang, and J. Sun. GPU Accelerated Dynamic Counterexample Generation in LTL Model Checking. In *ICFEM*, pages 413–429, 2014.

[20] Z. Wu, Y. Liu, J. Sun, J. Shi, and S. Qin. Gpu accelerated on-the-fly reachability checking. In *ICECCS*, pages 100–109. IEEE, 2015.

[21] L. Yang. *Model checking concurrent and real-time systems: the PAT approach*. PhD thesis, 2009.