Local State Space Construction for Compositional Verification of Concurrent Systems

Hao Zheng
University of South Florida
4202 E Fowler Ave., Tampa, FL 33620
zheng@cse.usf.edu

ABSTRACT

Local state space construction is crucial for efficient compositional verification of local and global properties of concurrent systems. This paper presents such an approach where local state transition models are built by iteratively searching the joint state space of communicating processes. The resulting local models contain less unreachable states, which would reduce false counter-examples for verifying local safety properties. Alternatively, more precise transition dependence relations can be extracted from these local models for more effective partial order reduction when the global state space is searched. The prototype of this approach has been implemented in an explicit model checker, and experimented on several concurrent examples. The initial results are encouraging.

Keywords

model checking, compositional reasoning, minimization, abstraction

1. INTRODUCTION

Compositional methods are essential to address the state-explosion problem in verifying asynchronous concurrent systems. Figure 1 shows a framework of compositional verification where the approach presented in this paper contributes. Given a system as a composition of communicating processes, this framework, instead of considering the whole system, first builds local state transition models for the individual processes where available local safety properties can be verified. If any local properties are not verified or there are global properties, then these local models are analyzed to extract transition dependence information for effective partial order reduction when the global state space for the whole system is searched for verifying global properties and the local properties that cannot be verified at the local level. The effectiveness of this framework is demonstrated in [22].

Local state space construction is critical in this framework. It is based on the *thread-modular* model checking approach as presented in [8] to address the state explosion problem for verifying local safety properties of multi-thread programs. Safety property verification can be reduced to the problem of checking if an error state is reachable from the program's initial state. Instead of exploring the whole state space of the entire program, it verifies individual threads by automatically inferring their environment assumptions. This approach iteratively computes the guarantee for each thread, which is a binary relation over global variables indicating how they are modified by that thread. The environment assumption of a thread is the disjunction of the guarantees of all the other threads. Each thread is then verified using the standard algorithm. During the course of verification, if a thread modifies global variables, its guarantee is updated. Subsequently, all threads whose environment assumptions changed as a result of the guarantee update are verified again. The iteration continues until the reachable state space and the guarantee of each thread converge. The thread-modular approach is inherently incomplete as the guarantees do not keep track of the sequencing information on global variable modifications. The derived environment assumptions are weaker than necessary, and therefore can cause a large number of unreachable states to be searched. The unreachable states can easily lead to false counter-examples of local properties, and cause the extracted transition dependence relation to be less precise leading to less effective partial order reduction for global property verification.

This paper intends to address this problem by proposing a method that can take such sequencing information into account when traversing and building local state space models. The proposed method does not construct guarantees or environment assumptions explicitly. Instead, the interactions of a process with other processes are captured by searching their joint state space. More specifically, if two processes, M_1 and M_2, in a concurrent system share some state variables, the proposed method searches their joint state space. While their joint state space is searched, the local state space models for M_1 and M_2 are constructed such that each of them captures not only the states and state transitions as defined in the corresponding process but also those due to the modification on shared variables by the other process. This method continues until the state space models for all processes reach a fixpoint. Similar to the method in [8], the proposed method is sound but incomplete. On the other hand, this method is able to capture more sequencing in-

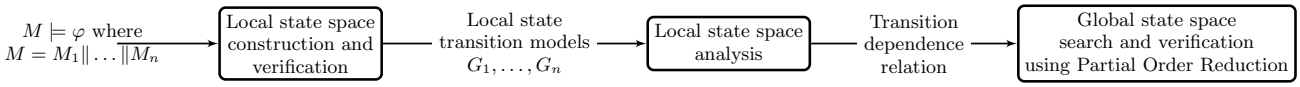

Figure 1: A Compositional Verification Framework.

formation about the interactions of a process with its environment into its state space model as its traverses the joint state space of processes. This often helps to avoid searching a large number of unreachable states, which lead to less false counter-examples and higher overall efficiency of the entire framework. The downside is higher complexity in some cases due to more information that needs to be considered.

Related Work The thread-modular approach in [8] is not implemented. [11] presents an implementation in the concurrent software model checker BLAST augmented with abstraction refinement to handle the infinite data space. A similar approach to asynchronous circuit verification is presented in [19]. Lately, [16] presents an abstraction refinement extension to make the thread modular approach complete. On the other hand, the completeness of our framework is achieved by searching the global state space with partial order reduction helped with the information extracted from the local state transitions models.

The approaches presented in this paper and [8, 11, 19] fall in a category, *assume-guarantee* reasoning, a general compositional verification method. Early work [12, 14, 17] requires users to provide an environment assumption to model how global states are updated by all other threads when a thread is verified in separation. If threads have complex interactions among themselves, it can be difficult to find a simple and accurate environment assumption for a thread. Some previous compositional approaches [4, 9, 10, 15, 20, 21] use a context abstraction or assumption for each process to find its state space for sound verification. Over-approximations are required in order to avoid any false positive results. On the other hand, the context should be sufficiently accurate to avoid excessive number of false counter-examples for higher efficiency. These methods assume user-provided contexts, which can be very difficult to obtain for systems with complex interactions among different processes. In [5], an automated approach is described to generate the assumptions for compositional verification based on a counter-example guided learning algorithm. This approach starts with a set of the weakest assumptions for a process, and iteratively refines these assumptions. That is, the initial contexts abstraction is made very coarse at the beginning, and it is gradually refined by refuting the encountered counter-examples until a real counter-example is found or the system is shown to satisfy its specification. A major potential problem of this approach is the large number of iterations of counter-example analysis and context refinement, which could have a large negative impact on the overall efficiency. Moreover, the initial coarse context can cause the unreachable state space to be very large, which may cause memory overflow. Similar work is also described in [3, 1, 2, 7].

On the other hand, the approaches presented in this paper and in [8, 11, 19] verify a process starting with an under-approximate environment assumption, and gradually extend the local state space model for each process by including global state updates allowed by other processes.

An approach is described in [6] for verifying global safety properties using local state space search. It gradually exposes more local information which is used to strengthen the local state space invariances until global safety properties are verified or real counter-examples are found. While both the approach in this paper and [6] rely on local state space search, the proposed approach alone does not intend to be complete. Instead, the completeness is achieved by searching the global state space as indicated at the beginning of this section. Also, the work in [6] is symbolic while this approach is explicit.

2. BACKGROUND

Let $\mathbb{Z}$ be the set of integers, and $V = (v_1, \ldots, v_n)$ be an ordered finite set of variables. Each $v_i \in V$ takes its values from its finite domain $Z_i \subset \mathbb{Z}$. Let $Z = Z_i \times \ldots \times Z_n$ be the set of all possible assignments to all variables in V.

A concurrent system is typically described in some high-level language. Since the exact formalism for describing a concurrent system is not important, this paper simply assumes that a finite state concurrent system is described with a tuple $M = (V, init, A)$. A state of M, $s \in Z$, is an assignment to V. *init* is the initial state which defines the values for the variables when the system starts. A is a set of actions defining how the system changes its states when actions in A are executed. Each action $\alpha \in A$ is specified with (g, va) where

- A guard $g : Z \rightarrow \{\mathsf{true}, \mathsf{false}\}$ is a total function that map each state to truth value.
- An assignment operation $va : Z \rightarrow Z$ is a total function that maps one state to another.

In general, an action α only modifies a subset of variables in V. Let $wr(\alpha) \subseteq V$ be the set of variables modified by action α.

An action $\alpha = (g, va)$ is enabled in a state s if $g(s)$ evaluates to true in s. Let $enb(s)$ denote all actions enabled in s. An action can be executed once it becomes enabled, and possibly changes to a new state. The successor state s' after executing an enabled action *act* is denoted by $\alpha(s)$.

In this paper, we often need to check the consistency of the assignments in two states on a subset of variables. Given a set of variables C, let $val(s, C)$ return the set of assignments in s to variables in C. Two states, s_1 and s_2, are consistent on C, denoted as $s_1 =_C s_2$, iff $val(s_1, C) = val(s_2, C)$.

A large and complex system usually consists of processes connected in a network where communications among the processes can be done through *shared variables*. Let $M_i = (V_i, init_i, A_i)$, $1 \leq i \leq n$, be n processes. The shared variables C_{ij} between M_i and M_j is $V_i \cap V_j$. Assume that the sets of actions of different processes are pairwise disjoint.

A concurrent system as a composition of n processes is defined as $M = \|_{1\leq i\leq n} M_i = (V, init, A)$ where $V = \cup_{1\leq i\leq n} V_i$, $init = (init_1, \ldots, init_n)$ such that $\forall 1 \leq i, j \leq n, init_i =_{C_{ij}} init_j$, and $A = \cup_{1\leq i\leq n} A_i$. A state s_i of M_i is referred to as a local state of M_i, and a state of M is referred to as a global state, which is a $n-$tuple of local states, i.e. $(s_1, \ldots, s_n)$ such that $\forall 1 \leq i, j \leq n, s_i =_{C_{ij}} s_j$.

Consider a process $M_i = (V_i, init_i, A_i)$ in a concurrent system $M = \|_{1\leq i\leq n} M_i$. The actions in A_i are referred to as local actions to M_i. If a M_i-action α modifies some variables of M_j, this action is referred to as external to M_j. Let $ext(M_j, \alpha)$ be a predicate that holds true if $\alpha \notin A_j$ but $wr(\alpha) \cap V_j \neq \emptyset$.

Given a global state $\vec{s} = (s_1, \ldots, s_n)$, $enb(\vec{s}) = \cup_{1\leq i\leq n} enb(s_i)$. When an action $\alpha \in enb(\vec{s})$ is executed, one or multiple local states may be changed resulting in a new global state $\vec{s}\,' = (s'_1, \ldots, s'_n)$, denoted as $\alpha(\vec{s})$, such that $s'_i = \alpha(s_i)$, and

$$\forall 1 \leq j \leq n, i \neq j$$

and

$$\begin{cases} ext(M_j, \alpha) & \Rightarrow \quad (s'_i =_{C_{ij}} s'_j \wedge s_j =_{V_j - V_i} s'_j) \\ \neg ext(M_j, \alpha) & \Rightarrow \quad s_j = s'_j \end{cases}$$

where $C_{ij} = V_i \cap V_j$. Naturally, $\forall 1 \leq i, j \leq n, s'_i =_{C_{ij}} s'_j$.

Figure 2 shows a simple example of a concurrent system with three processes. In this example, variable x is shared by all three processes, and y is shared between M_2 and M_3. All the other variables are internal to their respective processes. Actions α_1 and α_3 of M_1 are invisible as they only modify the internal variables of M_1, while α_2 and α_4 are external to both M_2 and M_3.

The state transition level semantics of concurrent systems are represented in state graphs, which are defined below.

Definition 2.1. *A state graph for a concurrent system M is a tuple $G = (S, \imath, R)$ where*

1. *S is a finite non-empty set of states,*
2. *$\imath \in S$ is the initial state.*
3. *$R \subseteq S \times (A \cup Ext) \times S$ is the set of state transitions where Ext is the set of eternal actions of the environment to M.* ∎

The above state graph definition is used to represent both concurrent systems and their processes. If a state graph $(S, \imath, R)$ is for a system $\|_{1\leq i\leq n} M_i$, then $Ext = \emptyset$, it is referred to as the global state graph where S and R are the set of global states and the set of global state transitions of M, respectively, and $\imath = (init_1, \ldots, init_n)$. If a state graph is defined for a process M_i in a system $\|_{1\leq i\leq n} M_i$, it is referred to as the local state graph of M_i, and its set of external actions is

$$Ext_i = \{\alpha | \exists 1 \leq j \leq n \; s.t. \; \alpha \in A_j \wedge ext(M_i, \alpha)\}.$$

Algorithm 1: $DFS(\|_{1\leq i\leq n} M_i)$

Input: A system description of n processes.
Output: A global state graph $(S, \imath, R)$.

```
1  ι := Σ(init_1, ..., init_n);
2  S := S ∪ ι;
3  stack.push((ι, enb(ι)));
4  while stack is not empty do
5  |  (s, E) := stack.pop();
6  |  if E = ∅ then
7  |  |  continue;
8  |  Select α ∈ E to execute, and remove it from E;
9  |  stack.push((s, E));
10 |  s' := α(s);
11 |  /* Check safety properties here. Terminate if a
   |  violation is found. */ R := R ∪ {(s, α, s')};
12 |  if s' ∉ S then
13 |  |  stack.push((s', enb(s')));
14 |  |  S := S ∪ s';
```

As to be shown later, the local state graph of a process contains not only the state transitions on local actions but also those on Ext_i to take into account updates on shared variables caused by actions executed in other processes that are external to M_i. Then, in $G_i = (S_i, \imath_i, R_i)$ for M_i, S_i is the set of local states, R_i is the set of state transitions on local actions in A_i of M_i or on actions in Ext_i external to M_i, and $\imath_i = init_i$. The local state graph of M_i in a system captures the behavior as defined in M_i as well as updates on its shared variables by actions of other processes in the system.

Executions of a concurrent system are represented by paths in its state graph. Given a state graph $G = (S, \imath, R)$, a path of G is a sequence $s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots$ such that for every $i \geq 0$, $(s_i, \alpha_i, s_{i+1}) \in R$ holds for some α_i. State q is reachable from r if there exists a path $s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{n-1}} s_n$ such that $r = s_0$ and $q = s_n$. State q is reachable in G if q is reachable from the initial state through a path. The set of paths in a state graph G from the initial state is denoted as $Path(G)$.

Given a concurrent system $M = \|_{1\leq i\leq n} M_i$, its state graph can be constructed using reachability analysis by executing exhaustively every enabled action in every state starting from the initial state. A general depth-first search algorithm for reachability analysis is shown in Algorithm 1. Checking safety properties can be easily combined with the given algorithm.

Given a set of variables C, a state transition (s, α, s') is invisible relative to C if the assignments in s and s' are different only on the variables not in C. Two paths are called *stutter equivalent* if they only differ in their corresponding invisible state transitions.

Definition 2.2. *Let C be a set of variables. Two paths, $\rho = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots$ and $\rho' = s'_0 \xrightarrow{\beta_0} s'_1 \xrightarrow{\beta_1} \ldots$, are stutter equivalent relative to C, denoted $\rho \approx_C \rho'$, if there exist two infinite sequences of positive integers $0 \leq i_1 \leq i_2 \leq \ldots$ and $0 \leq j_1 \leq j_2 \leq \ldots$ such that for every $k \geq 0$, the following*

$$
\begin{array}{rcl|rcl|rcl}
M_1 & = & (V_1, q_0, A_1); & M_2 & = & (V_2, p_0, A_2); & M_3 & = & (V_3, s_0, A_3); \\ \hline
V_1 & = & \{l_1, x, z\}; & V_2 & = & \{l_2, x, y\}; & V_3 & = & \{l_3, x, y\}; \\
q_0 & = & (l_1 = 0, x = 0, z = 0); & p_0 & = & (l_2 = 0, x = 0, y = 0); & s_0 & = & (l_3 = 0, x = 0, y = 0); \\
A_1 & = & \{\alpha_1, \alpha_2, \alpha_3, \alpha_4\}; & A_2 & = & \{\beta_1, \beta_2\}; & A_3 & = & \{\gamma_1, \gamma_2\}; \\
\text{where} & & & \text{where} & & & \text{where} & & \\
\alpha_1 & = & (l_1 = 0 \wedge x > 0, & \beta_1 & = & (l2 = 0 \wedge y = 0, & \gamma_1 & = & (l_3 = 0 \wedge y = 1, \\
 & & \quad z := x + 1; l_1 = 1); & & & \quad x := 2; l_2 := 1); & & & \quad x := 3; l_3 := 1); \\
\alpha_2 & = & (l_1 = 1, & \beta_2 & = & (l_2 = 1 \wedge x = 0, & \gamma_2 & = & (l_3 = 1 \wedge x = 0, \\
 & & \quad x := 0; l_1 = 2); & & & \quad y := 1; l_2 := 0) & & & \quad y := 0; l_3 := 0) \\
\alpha_3 & = & (l_1 = 2 \wedge x > 0, & & & & & & \\
 & & \quad z := z * x; l_1 = 3); & & & & & & \\
\alpha_4 & = & (l_1 = 3, & & & & & & \\
 & & \quad x := 0; z := 0; l_1 = 0); & & & & & &
\end{array}
$$

Figure 2: An example of a simple concurrent system with three processes communicating over shared variables x and y.

condition holds.

$$s_{i_k} =_C s_{i_k+1} =_C \ldots =_C s_{i_{k+1}-2} =_C s_{i_{k+1}-1} =_C s'_{j_k} =_C s'_{j_k+1} =_C \ldots =_C s'_{j_{k+1}-2} =_C s'_{j_{k+1}-1}$$

The stutter equivalence for two finite paths is defined similarly. ∎

Two state graphs can be related by the stutter simulation relation, which is defined below.

DEFINITION 2.3. *Let G_i, $i \in \{1, 2\}$, be two state graphs, and C a set of variables. G_2 is said to be a stutter simulation of G_1 relative to C, denoted as $G_1 \preceq_C G_2$, if for every $\rho_1 \in Path(G_1)$, there is $\rho_2 \in Path(G_2)$ such that $\rho_1 \approx_C \rho_2$.* ∎

3. LOCAL STATE GRAPH CONSTRUCTION

Given a concurrent system under verification, its state graph can be constructed directly by performing reachability analysis. The main drawback is state explosion if the system consists of a large number of concurrent processes. On the other hand, it is typical that properties to verify are often defined locally, therefore it is not necessary to construct the global state graph for verification if those properties can be verified locally. This section describes an approach to the construction of local state graphs for the consisting processes of a given system. The constructed local state graphs are shown to be stutter simulations of the global state graph. This indicates that the safety properties specified locally, if they hold in the local state graphs, also hold in the global one.

First, the general idea and limitation of the thread-modular approach in [8] are shown using the example shown in Figure 2. Next, a new method is proposed to overcome such limitation. Finally, the soundness of the proposed method is shown in the last section.

3.1 Thread-Modular Approach: Illustration and Limitation

Consider the system $\|_{1 \le i \le 3} M_i$ shown in Figure 2. Initially, the environment assumptions for all processes are empty. Only action β_1 in M_2 is enabled. After β_1 is executed, state graph G_2 for M_2 is extended with a state transition on β_1 as shown in Figure 3(a). Since this state transition updates the shared variable x, such update is a guarantee of M_2, and it is added to the environment assumptions for M_1 and M_3, respectively. Since the environment assumptions for M_1 and M_3 have changed, the state space of M_1 and M_3 under the new environment assumptions is traversed, and their state graphs are extended, as shown in Figure 3(b), with external state transitions on β_1 drawn with dotted arrows. G_1 is also extended with new states and state transitions as a result of the added external state transition.

Now, state graph G_1 of M_1 contains a state transition on action α_2 that updates the shared variable x. The guarantee derived from this state transition is added to the environment assumptions of M_2 and M_3. Then, G_2 and G_3 of M_2 and M_3, respectively, are extended with new states and state transitions under their corresponding extended environment assumptions. The resulting state graphs are shown in Figure 3(c).

In a few more steps, the state graphs shown in Figure 3(d) are obtained. In G_3, state transition on action γ_1 of M_3, (s_3, γ_1, s_4), updates the shared variable x by changing it from 0 to 3. This update, which is a guarantee of M_3, is added into the environment assumption of M_1, which results in two external transitions (q_0, γ_1, q_{11}) and (q_3, γ_1, q_4) to be added into G_1. These two external state transitions can be added into G_1 as $x = 0$ in both state q_1 and q_3. However, according to the system model shown in Figure 2, action γ_1 can happen only after action β_1 has fired. This indicates that the external transition (q_0, γ_1, q_{11}) and all reachable states from it in gray are in fact unreachable. This undesirable result is due to the fact that the current representation of process guarantees is not able to capture the sequencing relations among actions.

3.2 Local State Space Search

This section describes a local state graph construction method that addresses the problem with the thread-modular approach as shown in the previous section. Consider a system $\|_{1 \le i \le n} M_i$. The basic idea is as follows. For every pair of M_i and M_j such that $i \neq j$ and $V_i \cap V_j \neq \emptyset$, this apporach traverses their joint state space starting from their joint initial states $(init_i, init_j)$. In each joint state (s_i, s_j), all the enabled actions are checked. When a M_i-action α, if external to M_j, is executed, a state transition on α is added to the state graph G_i, and an external state transition also on α is added to the state graph G_j as well to represent such change on the shared variables due to α. Apply the same procedure for M_j. If any new local states are found, the state space of

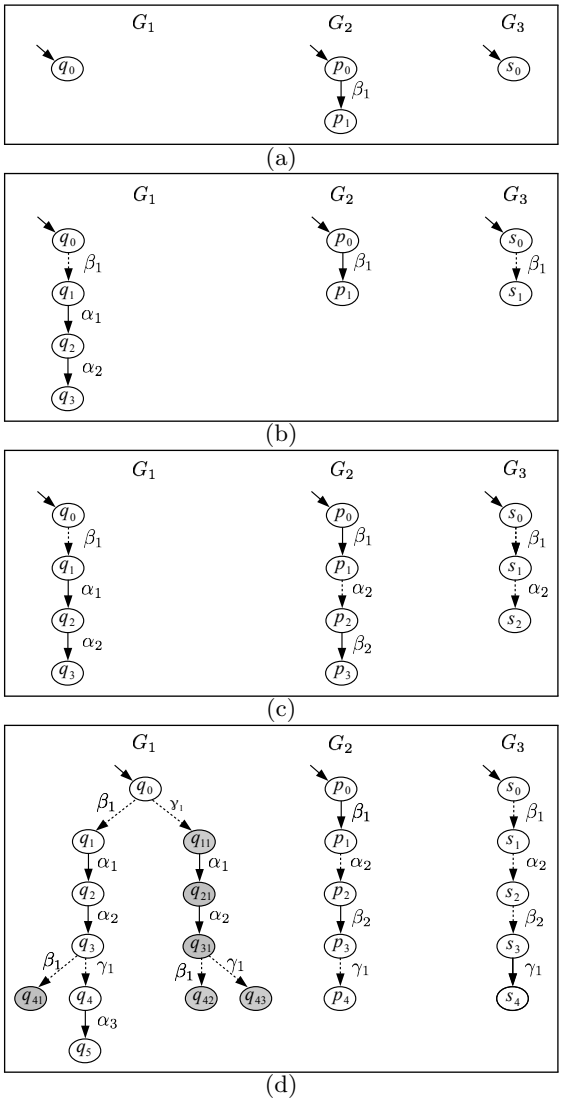

Figure 3: (a)-(d) Snapshots of partial SGs generated for the example shown in Figure 2 using the thread-modular approach.

Algorithm 2: *localSearch*(M_i, M_j, G_i, G_j)

Input: Two Processes M_i and M_j and their SGs G_i and G_j.

Output: G_i and G_j expanded with new states and state transitions.

```
1  C_ij = V_i ∩ V_j;
2  init = (init_i, init_j);
3  S = S ∪ init;
4  stack.push(init);
5  while stack is not empty do
6      (s_i, s_j) = stack.top();
7      if s_i is not in G_i then
8          DFS(M_i, s_i);
9      if s_j is not in G_j then
10         DFS(M_j, s_j);
11     foreach (s_i, α, s_i') ∈ outgoing(G_i, s_i) do
12         if α ∈ A_i ∧ ext(M_j, α) = true then
13             Create a state transition (s_j, α, s_j') s.t.
               s_i' =_{C_ij} s_j';
14             Add (s_j, α, s_j') to G_j;
15             s⃗' = (s_i', s_j');
16         if α ∈ A_i ∧ ext(M_j, α) = false then
17             s⃗' = (s_i', s_j);
18         if ext(M_i, α) = true ∧ ext(M_j, α) = true then
19             if (s_j, α, s_j') ∈ outgoing(G_j, s_j) ∧ s_i' =_{C_ij} s_j'
               then
20                 s⃗' = (s_i', s_j');
21     foreach (s_j, α, s_j') ∈ outgoing(G_j, s_j) do
22         if α ∈ A_j ∧ ext(M_i, α) = true then
23             Create a state transition (s_i, α, s_i') s.t.
               s_i' =_{C_ij} s_j';
24             Add (s_i, α, s_i') to G_i;
25             s⃗' = (s_i', s_j');
26         if α ∈ A_j ∧ ext(M_i, α) = false then
27             s⃗' = (s_i, s_j');
28         if ext(M_j, α) = true ∧ ext(M_i, α) = true then
29             if (s_i, α, s_i') ∈ outgoing(G_i, s_i) ∧ s_i' =_{C_ij} s_j' then
30                 s⃗' = (s_i', s_j');
31     if s⃗' ∉ S then
32         stack.push(s⃗');
33         S = S ∪ s⃗';
```

M_i or M_j is then searched individually starting from these new local states. This subsequently may result in more local states and local state transitions added into G_i or G_j. Once the individual state space search is done, the local search is resumed on their joint state space. The above local state space search algorithm alternates between two steps: joint and individual state space search. The purpose of the joint state space search of two processes is to expand the state graph of a process with states as a result of interactions between these two processes, while the individual state space search is to expand the state graph with additional reachable states from the states resulting from the interactions of these two processes.

Algorithm 2 shows the local search algorithm as described above. First, $outgoing(G, s)$ is defined to be the set of state transitions of G originating from state s. It takes a pair of processes, M_i and M_j, and their corresponding partially constructed state graphs, G_i and G_j, and expands the state graphs by adding states and state transitions to either state graph as allowed by the other process. The algorithm considers every pair of local states starting from the pair of initial states. Line $7-10$ in Algorithm 2 performs the individual state space search on M_i and M_j separately if either local state does not exist in the corresponding state graph. Next, for each pair of local states currently on the stack, (s_i, s_j), its successor pair of local states is determined. Line $11-20$ considers M_i as the environment to M_j. The outgoing state transitions of state s_i in G_i are divided into three groups. State transitions in the first group are due to the actions local to M_i but external to M_j. For each transition (s_i, α, s_i') in this group, an external state transition (s_j, α, s_j') is created and added to G_j as shown in line $12-15$. And the successor pair of local states is (s_i', s_j'). The second group contains the state transitions in G_i on actions that are not external to G_j. These state transitions do not update the shared variables, therefore, the succes-

sor pair of local states is determined as in line 17 where the local state of G_j remains the same. The third group considers state transitions due to actions external to both processes. These actions are local to a third process that update some common shared variables of M_i and M_j. If both local states s_i and s_j have outgoing state transitions, (s_i, α, s_i') and (s_j, α, s_j'), on such external action α such that the corresponding successor states s_i' and s_j' are consistent relative to the shared variables between M_i and M_j, then the successor pair of local states is (s_i', s_j'). If such an external state transition exists in M_i but not in M_j, or both external state transitions exist in M_i and M_j on α but their local successor states are not consistent, this indicates that these two processes cannot be synchronized in (s_i, s_j) on the external action α, therefore their joint successor is not possible in the global state space, and ignored consequently. This group of state transitions are handled in line $18-20$. Line $21-30$ shows the symmetric case where M_j is treated as the environment to M_i. Finally, if the joint successor states are not considered yet, they are pushed onto stack to be considered later. When the algorithm terminates, all the joint local state pairs that might exist in the global state space are considered, and legal external state transitions representing the interactions between these two processes are added to their corresponding local state graphs. These two processes and their respective state graphs are said to be *synchronized* when the algorithm terminates. This algorithm can be naturally extended to consider multiple processes at once.

Given a concurrent system consisting of only two processes, this algorithm does not have any advantage as its performance is close to that of the traditional state space search on the whole system. From now on, it is assumed that concurrent systems considered in this paper consist of at least three or more processes. Let $M = \|_{1 \le i \le n} M_i$ with $n \ge 3$. After applying Algorithm 2 to M_i and M_j, it is possible that G_i or G_j now are expanded with new state transitions that may represent new interactions on the interface with other processes. Therefore, if G_i or G_j is expanded with new state transitions, Algorithm 2 is applied again to these processes and other ones if they share common variables. It is iteratively applied until all local state graphs reach a fixpoint. The top level algorithm is shown in Algorithm 3.

In Algorithm 3, each process is associated with a variable new_i to indicate if there are new state transitions added to state graph G_i after the local search. This variable is used to decide whether Algorithm 2 is applied. If the state graph of neither process is extended with new state transitions from the previous local search, it is not necessary to consider these two processes, as shown in line 8. The algorithm terminates when all new_i are false. Additionally, the local search only applies to processes that share common variables.

As an example, consider constructing the local state graphs for processes as shown in Figure 2. Initially, all three state graphs are empty. First, apply the local state space search to M_1 and M_2, and the resulting local state graphs are shown in Figure 4(a). Notice that the local state transitions on α_2 and β_1 in M_1 and M_2, respectively, are added to G_2 and G_1 as external state transitions. Since M_2 is expanded with new state transitions, and it shares common variables x and y with M_3, the local state space search is applied to M_2 and M_3 next. The resulting partial local state graphs are shown in Figure 4(b). M_1 and M_3 are considered next as G_3 is expanded with a new state transition, and the resulting state graphs are shown in Figure 4(c). After a few more step, the final local state graphs are shown in Figure 4(d). Note that the unreachable states in the state graphs shown in Figure 3(d) are not generated during the local state space search.

Algorithm 3: $search(M_1, \ldots, M_n)$

Input: M_i $(1 \le i \le n)$: consisting processes of a concurrent system.
Output: G_i $(1 \le i \le n)$: local state graph for M_i.

```
1  foreach M_i do
2  |  Create an empty state graph G_i;
3  |  new_i := true;
4  |  new_i' := false;
5  while ∨_{1≤i≤n} new_i do
6  |  foreach i, j : 1 ≤ i, j ≤ n do
7  |  |  if i ≠ j ∧ V_i ∩ V_j ≠ ∅ ∧ (new_i ∨ new_j) then
8  |  |  |  localSearch(M_i, M_j, G_i, G_j);
9  |  |  |  if New transitions added into G_i then
10 |  |  |  |  new_i' := true;
11 |  |  |  if New transitions added into G_j then
12 |  |  |  |  new_j' := true;
13 |  foreach 1 ≤ i ≤ n do
14 |  |  new_i := new_i';
```

3.3 Soundness

The following theorem shows that G_i $(1 \le i \le n)$ constructed with Algorithm 3 for M_i in $M = \|_{1 \le i \le n} M_i$ is a stutter simulation of the state graph G for the whole system M. This result shows that any next-free LTL safety properties, if hold on G_i, also hold on G, therefore the soundness of this approach.

THEOREM 3.1. *Let $M = \|_{1 \le i \le n} M_i$ be a concurrent system model, and G its state graph. Also, let G_i be local the state graph constructed for M_i for $1 \le i \le n$ with Algorithm 3. The following statement is true.*

$$\forall 1 \le i \le n, G \preceq G_i.$$

Proof: We need to show that every path in G has a corresponding stutter-equivalent path in G_i. Suppose $\rho = (\vec{s}_0, t_0, \vec{s}_1, t_1, \ldots)$ be a path in G where $\vec{s}_0 = (init_1, \ldots, init_n)$ and $\vec{s}_k = (s_{k1}, \ldots, s_{kn})$. For any $k \ge 0$, $s_{ki} =_{C_{ij}} s_{kj}$, $1 \le i, j \le n$ and $i \ne j$.

First, apply Algorithm 2 to every pair of local states $(init_i, init_j)$ such that $1 \le i, j \le n$, $i \ne j \wedge V_i \cap V_j \ne \emptyset$. Consider every $\alpha_0 \in enb(init_i) \cup enb(init_j)$ in three cases.

- Case 1: α_0 is local to M_i. Obviously state transition $(init_i, \alpha_0, s_{1i})$ is included in G_i, and an external state transition $(init_j, \alpha_0, s_{1j})$ is added to G_j as shown in Algorithm 2.
- Case 2: α_0 is local to M_j but external to M_i. Then, a state transition $(init_j, \alpha_0, s_{1j})$ is added to G_j, and an

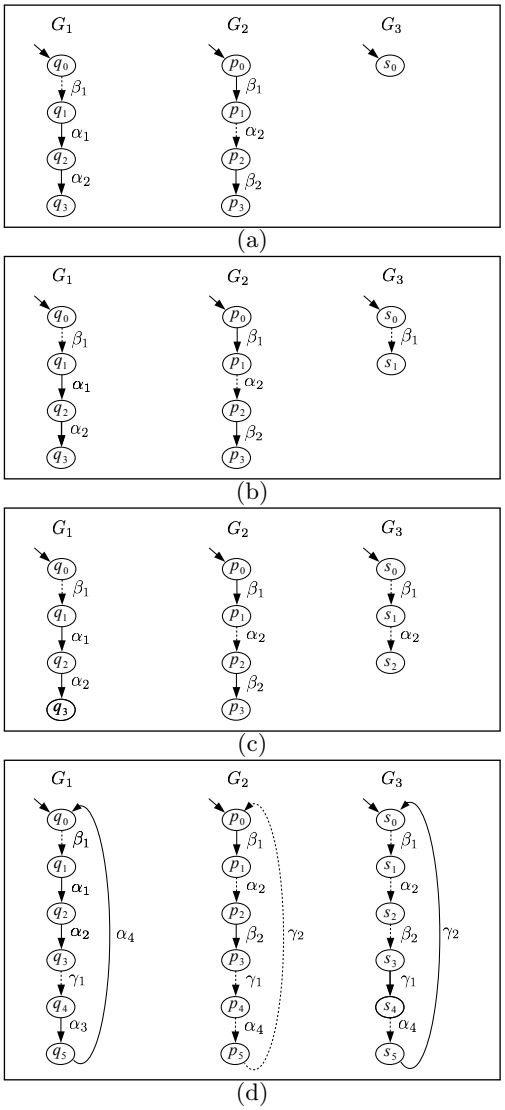

Figure 4: Partial state graphs for the example shown in Figure 2 during the course of apply Algorithm 3.

external state transition $(init_i, \alpha_0, s_{1i})$ is added into G_i.

- Case 3: α_0 is neither local nor external to M_i, i.e. invisible. Then $s_{1i} = init_i$.

Therefore, according to Definition 2.2, for ρ, there exists a path in G_i such that they are stutter-equivalent for one step. According to the construction by Algorithm 2, $\forall 1 \leq i, j \leq n, i \neq j \wedge s_{1i} =_{C_{ij}} s_{1j}$ holds.

Next, Algorithm 2 is applied again to every pair (s_{1i}, s_{1j}) such that $V_i \cap V_j \neq \emptyset$, and now consider every $\alpha_1 \in enb(s_{1i}) \cup enb(s_{1j})$. Similarly, we can show that there exists a path in G_i such that it is stutter-equivalent to ρ for two steps. By applying the above reasoning repeatedly, we can show that there exists a path in G_i that is stutter-equivalent to ρ for arbitrary length. ∎

4. EXPERIMENTAL RESULTS

The method and the algorithms described in this paper are implemented in a package, and integrated into a concurrent system verification tool *Platu*, an explicit model checker implemented in Java. A number of examples are selected for experiments. These examples include communication protocols and mutual exclusion algorithms for parallel systems from [18]. The thread-modular approach described in [8] is also implemented for comparison. All experiments are performed on a iMac desktop with a Intel quad-core processor, however, only a single thread is used for all experiments. An upper bound of 300 seconds on time and an upper bound of 2 GB on memory use are enforced.

The first set of experiments consider small examples of low complexity. The state space of these examples can be searched by the traditional reachability analysis algorithm without difficulty, therefore the local state graphs for the processes of these examples can be constructed when the traditional reachability analysis is performed on the whole system. This approach is referred to as Mono in the results tables. Similarly, the thread modular approach in [8] is referred to as TMMC, and the method presented in this paper is referred to as LS3. The purpose of this experiment is to compare the results from using TMMC and LS3 against the results obtained with Mono which are used the baseline. From Table 1, it can be seen that method LS3 always produces local state graphs with less unreachable states compared to those obtained with method TMMC. In two cases shown in Table 1 and several other small examples whose results are not included, LS3 produces local state graphs with the same number of states as what can be obtained with method Mono for all local state graphs. These results show that including the sequencing information among the external actions when building local state graphs can really help to avoid a lot of unreachable states.

The second set of experiments consider some large examples of higher complexity, which cannot be finished by SPIN [13] even when the partial-order reduction is turned on. In addition to showing the comparison among the local state graphs generated with the method TMMC and LS3, their runtime and memory use are also compared. Additionally, local safety properties are defined for processes of these examples whenever appropriate. All these local properties hold true if the global state space is searched. In this experiment, we compare the efficiencies of TMMC and LS3 as well as their effectiveness in reducing unreachable states and thus false counter-examples for local safety properties. As shown in Figure 1, the local state graphs need to be fully constructed so that transition dependence information can be extracted to assist partial order reduction in global state space search for verifying global properties or local properties that cannot proved locally, TMMC or LS3 does not terminate even when a local property is shown violated until none of the local state graphs can be expanded.

From Table 2, it can be seen that method LS3 still produces local state graphs with less unreachable states for all examples. For almost all examples, LS3 can prove the local safety properties where TMMC fails. In the particular case for peterson.4, TMMC times out as it keeps generating more states for the local state graphs while LS3 proves all local

Table 1: Comparison of the results from using the traditional reachability analsysis (Mono), the method described in [8] (TMMC), and the method described in this paper (LS3). Time is in seconds, and memory is in MBs. For each example, the number of states in its local state graphs are shown.

	phil	peterson.3	syzmanski.4
Mono	$(9, 9, 9, 9)$	(2627, 2421, 2745)	(4311, 4415, 4383, 4352)
TMMC	$(16, 16, 16, 16)$	(2997, 2952, 2952)	(5875, 6125, 6250, 6375)
LS3	$(9, 9, 9, 9)$	(2627, 2421, 2745)	(5201, 5453, 5598, 5755)

Table 2: Comparison of the results from using the the method described in [8] (TMMC), and the method described in this paper (LS3) on some more complex examples. TO indicates timeout.

	Method	Time	Mem	State count in local state graphs
brp	TMMC	2.7	174	$(5600^*, 2226^*, 351694^*, 295^*, 84, 30)$
	LS3	9.8	214	$(1368, 1496, 25091, 77, 35, 14)$
iprotocol	TMMC	8.8	408	$(19, 1256^*, 53, 18104^*, 110627^*, 283444^*)$
	LS3	10.1	68	(19, 230, 29, 2647, 3656, 23747)
lamport	TMMC	15.9	106	$(9344^*, 9344^*, 9344^*, 9344^*, 9344^*)$
	LS3	24.8	143	(8800, 8800, 8800, 8800, 8800)
lann	TMMC	1.3	15	(250, 250, 250, 250, 566,560,561, 412)
	LS3	4.3	33	(250,248,248,248,566,554,555,408)
peterson.4	TMMC	TO	–	$(124535^*, 104922^*, 104088^*, 103319^*)$
	LS3	10.9	88.5	$(13573, 12993, 12869, 12801)$
syzmanski.5	TMMC	59.4	198	$(35000^*, 36250^*, 36875^*, 37500^*, 38125^*)$
	LS3	59.1	211	$(30684^*, 31934^*, 32659^*, 33444^*, 34265^*)$

1. In the table, numbers labeled with * indicate that the local propertied defined for the corresponding processes are violated caused by the unreachable states.
2. Numbers without being labeled with * indicate either the local properties hold in the local SGs for the corresponding processes or no local properties are defined.

properties. For syzmanski.5, even though LS3 cannot prove the local properties, it constructs the local state graphs with significantly reduced unreachable states. This reduction can lead to better partial order reduction in global state space search in the later stage.

In general, the runtime required by LS3 is higher, except for example peterson.4. This is due to the repeated search of joint local state space of processes when LS3 runs. Memory requirements by these two methods do not fall into any pattern. In some cases, TMMC uses more memory as a large number of unreachable states are generated into the local state graphs, while in some other cases LS3 uses more memory for searching the joint local state spaces when the local state graphs contain large numbers of local states.

Next, a better analysis of the possible inefficiency of LS3 is provided. Whenever a local state graph is expanded, it must be considered again with any other local state graphs if they share common variables where their joint state space is searched. In the current implementation, the joint state space of local state graphs are not kept in memory, and it is constructed on demand. Every time the joint state space of two local state graphs is searched, it is actually built from scratch. This implementation intends to keep memory use low, however, it may cause a lot of waste of work as large part of the same joint state space is built and searched over and over again. Another reason for high runtime overhead it that LS3 sometimes may expand local state graphs slowly. This problem becomes very serious when the algorithm approaches the end where all local state graphs are close to their fixpoints. For all examples, in the first number of main iterations as shown in Algorithm 3, the state counts of the local state graphs are already very close to the corresponding numbers as shown in Table 2. Afterwards, only a small number of states or state transitions are added to the local state graphs in each iteration. This slow expansion results in a large number of the main iterations. Coupled with the high computational cost associated with the joint local state space search used frequently in each iteration, LS3 often spends a very large portion of the total runtime to finish the last few iterations.

From the experiments, it is clear that both TMMC and LS3 perform much better for the loosely coupled systems. By loosely coupled we mean that such a system does not have many states and state changes on the shared variables among its consisting processes. In the same benchmark, there are a number of examples that are tightly-coupled. Since states on the shared variables are replicated in the local state graphs where these variables are shared, this can result in high degree of redundant duplications in the local state graphs, thus higher memory use. On the other hand, loosely coupled examples could pose another challenge to LS3. When LS3 is applied to two local state graphs, their joint state space is actually constructed. If these two local

state graphs have sparse interactions, but a lot of internal state transitions, then the interleavings of their internal state transitions can lead to a very large joint state space, hence higher runtime and memory use. This explains why TMMC sometimes is more efficient as no joint state state space is built and searched.

Based on the above discussion, the LS3 algorithm needs to be improved by reducing redundant work that are often carried out by the current implementation. Additionally, the joint state space of some abstract forms of two local state graphs should be searched, instead of considering the local state graphs themselves. These abstract forms of the local state graphs should keep the state space on shared variables intact but hide internal state transitions as much as possible. Finally, a parallel implementation of the LS3 algorithm can reduce the runtime, although more memory is probably needed to keep multiple joint state space in memory at the same time.

5. CONCLUSION

This paper presents an approach to the construction of local state graphs of concurrent systems for local safety verification. The method and algorithms behind this approach overcome a drawback in the previous approach and can produce local state graphs with less unreachable states, which leads to less false counter-examples for safety properties. Experimental results are promising, but also shows several issues with this new approach. In the future, we plan to improve this approach by addressing these issues and run more experiments.

6. REFERENCES

[1] R. Alur, P. Madhusudan, and W. Nam. Symbolic compositional verification by learning assumptions. In *Proc. Int. Conf. on Computer Aided Verification*, volume 3576 of *LNCS*, pages 548 – 562. Springer-Verlag, 2005.

[2] M. Bobaru, C.S.Pasareanu, and D. Giannakopoulou. Automated assume-guarantee reasoning by abstraction refinement. In *Proc. Int. Conf. on Computer Aided Verification*, LNCS. Springer-Verlag, 2008.

[3] S. Chaki, E. Clarke, N. Sinha, and P. Thati. Automated assume-guarantee reasoning for simulation conformance. In *Proc. Int. Conf. on Computer Aided Verification*, LNCS, pages 534 – 547. Springer-Verlag, 2005.

[4] S. Cheung and J. Kramer. Context constraints for compositional reachability analysis. *ACM Transations on Software Engineering and Methodology*, 5(4):334–377, 1996.

[5] J. Cobleigh, D. Giannakopoulou, and C. Pasareanu. Learning assumptions for compositional verification. In *Proc. Int. Conf. on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 2619 of *LNCS*, pages 331–346. Springer-Verlag, 2003.

[6] A. Cohen and K. Namjoshi. Local proofs for global safety properties. In *Proc. Int. Conf. on Computer Aided Verification*, volume 4590 of *LNCS*. Springer-Verlag, 2007.

[7] C.S.Pasareanu, D. Giannakopoulou, and M. Bobaru. Learning to divide and conquer: applying $l^\star$ algorithm to automate assume-guarantee reasoning. *Formal Methods in System Design*, (3), 2008.

[8] C. Flanagan and S. Qadeer. Thread-modular model checking. In *Proceedings of the 10th International Conference on Model Checking Software*, SPIN'03, pages 213–224. Springer-Verlag, 2003.

[9] D. Giannakopoulou, C. Pasareanu, and H. Barringer. Assumption generation for software component verification. In *Proceedings of ASE'02*, pages 3–12. IEEE Computer Society, 2002.

[10] S. Graf and B. Steffen. Compositional minimization of finite state systems. In *Proc. Int. Conf. on Computer Aided Verification*, pages 186–196, 1990.

[11] T. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *Proc. Int. Conf. on Computer Aided Verification*, volume 2725 of *LNCS*, pages 262–274. Springer-Verlag, 2003.

[12] T. Henzinger, S. Qadeer, and S. Rajamani. You assume, we guarantee: methodology and case studies. In *Proc. Int. Conf. on Computer Aided Verification*, pages 440–451. Springer, 1998.

[13] G. J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[14] C. Jones. Tentative steps toward a development for interfering programs. *ACM TOPLAS*, 5(4):596–619, 1983.

[15] J. Krimm and L. Mounier. Compositional state space generation from lotos programs. In *Proc. Int. Conf. on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 239–258, London, UK, 1997. Springer-Verlag.

[16] A. Malkis, A. Podelski, and A. Rybalchenko. Thread-modular counterexample-guided abstraction refinement. In *Proceedings of the 17th International Conference on Static Analysis*, SAS'10, pages 356–372. Springer-Verlag, 2010.

[17] K. L. Mcmillan. A methodology for hardware verification using compositional model checking. Technical report, Cadence Berkeley Labs, 1999.

[18] R. Pelánek. Beem: Benchmarks for explicit model checkers. In *Proc. of SPIN Workshop*, volume 4595 of *LNCS*, pages 263–267. Springer, 2007.

[19] H. Zheng. Compositional reachability analysis for efficient modular verification of asynchronous designs. *IEEE Transactions on COMPUTER-AIDED DESIGN of Integrated Circuits and Systems*, 29(3), March 2010.

[20] H. Zheng, E. Mercer, and C. Myers. Modular verification of timed circuits using automatic abstraction. *IEEE Transactions on Computer-Aided Design*, 22(9):1138–1153, 2003.

[21] H. Zheng, C. Myers, D. Walter, S. Little, and T. Yoneda. Verification of timed circuits with failure directed abstractions. *IEEE Transactions on Computer-Aided Design*, 25(3):403–412, 2006.

[22] H. Zheng and Y. Zhang. Local state space analysis leads to better partial order reduction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2014. accepted for publication.