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-
formation about the interactions of a process with its environment into its state space model as it 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 invariants 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 $Z$ 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 $Z_i \subset Z$. Let $Z = Z_1 \times \ldots \times Z_n$ 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 $M = (V, \text{init}, A)$. 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 $g : Z \rightarrow \{\text{true, false}\}$ is a total function that maps 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 $\alpha$ only modifies a subset of variables in $V$. Let $\alpha V(\alpha) \subseteq V$ be the set of variables modified by action $\alpha$.

An action $\alpha = (g, va)$ is enabled in a state $s$ if $g(s)$ evaluates to true in $s$. Let $\text{enb}(s)$ 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 $\text{val}(s, C)$ return the set of assignments in $s$ to variables in $C$. Two states, $s_1$ and $s_2$, are consistent on $C$, denoted as $s_1 \approx_C s_2$, iff $\text{val}(s_1, C) = \text{val}(s_2, C)$.

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 $M_i = (V_i, \text{init}_i, A_i)$, $1 \leq i \leq n$, 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.

$$M \models \varphi \text{ where } M = M_1 \| \ldots \| M_n$$

Figure 1: A Compositional Verification Framework.
A concurrent system as a composition of n processes is defined as
\[ M = \bigparallel_{1 \leq i \leq n} M_i = (V_i, init_i, A_i) \] where \[ V = \bigcup_{1 \leq i \leq n} V_i, \]
\[ init = (init_1, \ldots, init_n) \] such that \( V_1 \subseteq i, j \leq n, init_i = c_{ij} \)
\[ init_j, \text{ and } A = \bigcup_{1 \leq i \leq n} A_i. \]
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
\[ \forall i, j \leq n, s_i = c_{ij} s_j. \]

Consider a process \( M_i = (V_i, init_i, A_i) \) in a concurrent
system \( M = \bigparallel_{1 \leq i \leq n} M_i \). The actions in \( A_i \) are referred to as
local actions to \( M_i \). Let \( ext(M_j, \alpha) \) be a predicate that holds true if \( \alpha \notin A_j \) but
\[ \forall r(\alpha) \cap V_j \neq \emptyset. \]

Given a global state \( s = (s_1, \ldots, s_n), \) \( enb(s) = \bigcup_{1 \leq i \leq n} enb(s_i). \)
When an action \( \alpha \) in \( enb(s) \) is executed, one or multiple local
states may be changed resulting in a new global state
\[ s' = (s_1', \ldots, s_n'), \] denoted as \( \alpha(s) \), such that \( s_i' = \alpha(s_i) \), and
\[ \forall i, j \leq n, i \neq j \]
\[ \begin{cases} \text{ext}(M_j, \alpha) & \Rightarrow (s_i' = c_{ij} s_j' \land s_j = v_j) \quad \text{or} \quad s_j' = v_j' \end{cases} \]
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 \( \alpha_1 \) and \( \alpha_3 \) of \( M_1 \) are invisible as they only modify
the internal variables of \( M_1 \), while \( \alpha_2 \) and \( \alpha_4 \) 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 \( G = (S, i, R) \) where

1. \( S \) is a finite non-empty set of states,
2. \( i \in S \) is the initial state,
3. \( R \subseteq S \times (A \cup Ext) \times S \) is the set of state transitions
6. \[ Ext \] 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
\( (S, i, R) \) is for a system \( \bigparallel_{1 \leq i \leq n} M_i \), then \( Ext = \emptyset \), it is referred to as the global state graph where \( S \) and \( R \) are the set of
local states and the set of global state transitions of \( M \), respectively, and \( i = (init_1, \ldots, init_n) \). If a state graph is
defined for a process \( M_i \) in a system \( \bigparallel_{1 \leq i \leq n} M_i \), it is referred to as the local state graph of
\( M_i \), and its set of external actions is
\[ Ext_i = \{ \alpha | \exists 1 \leq j \leq n s.t. \alpha \in A_j \land ext(M_i, \alpha) \}. \]

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 \( G_i = (S_i, i, R_i) \) 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 \( i = init_i \). 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 \( G = (S, i, R) \), a path of \( G \) is a sequence \( s_0 \overset{\alpha_0}{\rightarrow} s_1 \overset{\alpha_1}{\rightarrow} \ldots \) such that for every \( i \geq 0 \), \( (s_i, \alpha_i, s_{i+1}) \in R \) holds for some \( \alpha_i \). State \( q \) is reachable from \( r \) if there exists a path \( s_0 \overset{\alpha_0}{\rightarrow} s_1 \overset{\alpha_1}{\rightarrow} \ldots \overset{\alpha_{n-1}}{\rightarrow} s_n \) 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 \( M = \bigparallel_{1 \leq i \leq n} M_i \), 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 \( \langle s, \alpha, s' \rangle \) 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,
\( \rho = s_0 \overset{\alpha_0}{\rightarrow} s_1 \overset{\alpha_1}{\rightarrow} \ldots \) and \( \rho' = s_0' \overset{\alpha_0'}{\rightarrow} s_1' \overset{\alpha_1'}{\rightarrow} \ldots \), 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

![Algorithm 1: DFS(∥1≤i≤nMi∥)](image-url)

**Input:** A system description of \( n \) processes.

**Output:** A global state graph \( (S, i, R) \).

1. \( \rho := \Sigma(init_1, \ldots, init_n); \)
2. \( S := S \cup \{ i \}; \)
3. \( stack.push((i, \{ \})); \)
4. \( while \) \( stack \) is not empty \( \triangleright \)
5. \( (s, E) := stack.pop(); \)
6. \( if \ E = \emptyset \) \( \triangleright \)
7. \( continue; \)
8. \( Select \alpha \in E \) to execute, and remove it from \( E; \)
9. \( stack.push((s, \{ \alpha \})); \)
10. \( s' := \alpha(s); \)
11. \( /* Check safety properties here. Terminate if a \)
12. \( violation is found. */ \) \( R := R \cup \{ (s, \alpha, s') \}; \)
13. \( stack.push((s', \{ \alpha \})); \)
14. \( S := S \cup s'; \)
Consider the system $\parallel M_1 \parallel M_2 \parallel M_3$, shown in Figure 2. Initially, the environment assumptions for all processes are empty. Only action $\beta_1$ in $M_2$ is enabled. After $\beta_1$ is executed, state graph $G_2$ for $M_2$ is extended with a state transition on $\beta_1$ 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 $\beta_1$ drawn with dotted arrows. $G_3$ 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 $\alpha_2$ 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 $\gamma_1$ of $M_3$, $(s_{11}, \gamma_1, s_{11})$, 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 $(q_0, \gamma_1, q_1)$ and $(q_3, \gamma_1, q_4)$ 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 $\gamma_1$ can happen only after action $\beta_1$ has fired. This indicates that the external transition $(q_0, \gamma_1, q_1)$ 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. 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 $\parallel \parallel M_1 \parallel M_2 \parallel M_3$. 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 approach traverses their joint state space starting from their joint initial states (init, init, init). In each joint state $(s_i, s_j)$, all the enabled actions are checked. When a $M_i$-action $\alpha_i$ external to $M_i$ is executed, a state transition on $\alpha_i$ is added to the state graph $G_i$, and an external state transition also on $\alpha_i$ is added to the state graph $G_j$ as well to represent such change on the shared variables due to $\alpha_i$. Apply the same procedure for $M_j$. If any new local states are found, the state space of

---

**Table 1:**

<table>
<thead>
<tr>
<th>$M_1$</th>
<th>$M_2$</th>
<th>$M_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V_1 = {x, y}$</td>
<td>$V_2 = {y}$</td>
<td>$V_3 = {y}$</td>
</tr>
<tr>
<td>$q_0 = (t_1 = 0, x = 0, y = 0)$</td>
<td>$p_0 = (t_2 = 0, x = 0, y = 0)$</td>
<td>$s_0 = (t_3 = 0, x = 0, y = 0)$</td>
</tr>
<tr>
<td>$A_1 = {a_1, a_2, \alpha_1, \alpha_4}$</td>
<td>$A_2 = {\beta_1, \beta_2}$</td>
<td>$A_3 = {\gamma_1, \gamma_2}$</td>
</tr>
<tr>
<td>$\alpha_1 = (t_1 = 1)$</td>
<td>$\beta_1 = (t_2 = 0 \land y = 0, x := 0, t_2 := 1)$</td>
<td>$\gamma_1 = (t_3 = 0 \land y = 1, x := 3, t_3 := 1)$</td>
</tr>
<tr>
<td>$\alpha_2 = (t_2 = 0 \land x = 0, t_2 := 1)$</td>
<td>$\beta_2 = (t_2 = 0 \land x = 0, y := 1, t_2 := 0)$</td>
<td>$\gamma_2 = (t_3 = 0 \land x = 0, y := 0, t_3 := 0)$</td>
</tr>
</tbody>
</table>

*Figure 2:* An example of a simple concurrent system with three processes communicating over shared variables $x$ and $y$. 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 $G_1, i \in \{1, 2\}$, 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 $G_1 \approx_C G_2$, if for every $\rho_1 \in \text{Path}(G_1)$, there is $\rho_2 \in \text{Path}(G_2)$ such that $\rho_1 \approx_C \rho_2$.
$M_i$ 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, $\text{outgoing}(G, s)$ 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 $(s_i, \alpha, s'_i)$ in this group, an external state transition $(s_j, \alpha, s'_j)$ 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 succe-
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, \((s_i, \alpha, s'_i)\) and \((s_j, \alpha, s'_j)\), on such external action \( \alpha \) 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 a but their local successor states are not consistent, this indicates that these two processes cannot be synchronized in \( G \). 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 \( \alpha_2 \) and \( \beta_1 \) in \( M_1 \) and \( M_2 \), respectively, are added to \( G_2 \) and \( G_1 \) as external state transitions. Since \( M_3 \) 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 \( M_4 \) 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.

### 3.3 Soundness

The following theorem shows that \( G_i \) (1 \( \leq i \leq n \)) constructed with Algorithm 3 for \( M_i \) in \( M = \{1 \leq i \leq n \} M_i \) 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 \( M = \{1 \leq i \leq n \} M_i \) be a concurrent system model, and \( G \) its state graph. Also, let \( G_i \) be the state graph constructed for \( M_i \) for 1 \( \leq i \leq n \) with Algorithm 3. The following statement is true.

\[ \forall 1 \leq i \leq n, G \leq G_i. \]

**Proof:** We need to show that every path in \( G \) has a corresponding stutter-equivalent path in \( G_i \). Suppose \( p = (s_0, t_0, s_1, t_1, \ldots) \) be a path in \( G \) where \( s_0 = (init_1, \ldots, init_n) \) and \( s_k = (s_{k1}, \ldots, s_{kn}) \). For any \( k \geq 0 \), \( s_{ki} \equiv \text{enb}(init_i) \) \( s_{kj} \), 1 \( \leq i, j \leq n \) and \( i \neq j \).

First, apply Algorithm 2 to every pair of local states (\( init_i, init_j \)) such that 1 \( \leq i, j \leq n \), \( i \neq j \) and \( V_i \cap V_j = \emptyset \). Consider every \( \alpha_0 \in \text{enb}(init_i) \cup \text{enb}(init_j) \) in three cases.

- **Case 1:** \( \alpha_0 \) is local to \( M_i \). Obviously state transition \((init_i, \alpha_0, s_{i1})\) is included in \( G_i \), and an external state transition \((init_j, \alpha_0, s_{j1})\) is added to \( G_j \) as shown in Algorithm 2.

- **Case 2:** \( \alpha_0 \) is local to \( M_i \) but external to \( M_i \). Then, a state transition \((init_j, \alpha_0, s_{j1})\) is added to \( G_j \), and an
external state transition \((init_i, \alpha_0, s_{i1})\) is added into \(G_i\).

- Case 3: \(\alpha_0\) is neither local nor external to \(M_i\), i.e. invisible. Then \(s_{i1} = init_i\).

Therefore, according to Definition 2.2, for \(\rho\), there exists a path in \(G_i\) such that they are stutter-equivalent for one step. According to the construction by Algorithm 2, \(\forall 1 \leq i, j \leq n, i \neq j \land s_{i1} = C_{ij}, s_{i1} \) holds.

Next, Algorithm 2 is applied again to every pair \((s_{i1}, s_{i1})\) such that \(V_i \cap V_j \neq \emptyset\), and now consider every \(\alpha_i \in enb(s_{i1}) \cup enb(s_{i1})\). Similarly, we can show that there exists a path in \(G_i\) such that it is stutter-equivalent to \(\rho\) 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 \(\rho\) 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 an iMac desktop with an 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
properties. For syzmanski.5, even though LS3 cannot prove the local properties, it constructs the joint 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

Table 1: Comparison of the results from using the traditional reachability analysis (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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Time</th>
<th>Mem</th>
<th>State count in local state graphs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mono</td>
<td></td>
<td></td>
<td>(9344, 68, 104922, 103319, 9344)</td>
</tr>
<tr>
<td>TMMC</td>
<td></td>
<td></td>
<td>(2627, 2421, 2745)</td>
</tr>
<tr>
<td>LS3</td>
<td></td>
<td></td>
<td>(2627, 2421, 2745)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Time</th>
<th>Mem</th>
<th>State count in local state graphs</th>
</tr>
</thead>
<tbody>
<tr>
<td>brp</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>iprotocol</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lamport</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lann</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>peterson.4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>syzmanski.5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. In the table, numbers labeled with * indicate that the local property 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.
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