Working Paper Series
ISSN 1177-777X

The Explicit Conflict Check Algorithm Implemented in the Waters Library

Robi Malik

Working Paper: 01/2018
September 3, 2018

Department of Computer Science
The University of Waikato
Private Bag 3105
Hamilton, 3240
New Zealand

The Explicit Conflict Check Algorithm Implemented in the Waters Library

Robi Malik
Department of Computer Science
The University of Waikato
Hamilton, New Zealand
robi@waikato.ac.nz

September 3, 2018

Abstract

This working paper describes the implementation of *explicit* model checking algorithms to verify the *nonblocking* or *nonconflicting* property of discrete event systems. Explicit algorithms enumerate and store all reachable states of a synchronous composition. Three alternatives optimised for memory consumption or runtime are described and compared. The algorithms have been implemented in C++ in the discrete event systems library Waters, and experimental results show that they can explore more than 100 million states on standard computers.

1 Introduction

The nonconflicting property is a weak liveness property commonly used in *supervisory control theory* of discrete event systems to capture the absence of livelocks and deadlocks [22]. A system is nonconflicting if, from any reachable system state, some terminal state is reachable. A lot of effort has been put into the development of algorithms for the automatic verification of this crucial property. Systems of considerable size can be verified using *symbolic model checking* [3] or *compositional verification* [8, 20, 21, 26].

This working paper is concerned with *explicit* verification of the nonconflicting property. Explicit algorithms are the simplest model checking algorithms that construct all reachable states and explore every single transition. As all states are stored in memory, they are only

suitable for small to medium-size model checking problems with a few million reachable states, which they may solve faster than more complicated algorithms. Compositional verification [8, 20, 21, 26], which works by simplifying larger models, relies on an explicit or symbolic algorithm in its final step.

This working paper describes the explicit verification algorithms in the Waters library, which is part of Supremica [1], and which can explore state spaces with more than 100 million states. Existing discrete event systems software such as TCT [7] and libFaudes [17] performs conflict check by synchronous composition and nonblocking verification, which is inferior to the direct algorithms shown here. The SPIN model checker [10] has powerful explicit algorithms, but its modelling language neither supports synchronisation with events shared by more than two components nor the nonconflicting property.

This working paper is an extended version with full appendix of the conference paper [14], where the explicit nonblocking verification algorithm in the Waters library was first described. In the following, Section 2 provides the background of finite-state machines and the nonconflicting property. Then Section 3 describes three design alternatives of explicit conflict check algorithms, and Section 4 evaluates them using experiments. Section 5 adds concluding remarks. As an addition to [14], the appendix includes further details about the iterative version of the strongly connected components algorithm used in Section 3 and about the models used for the experiments in Section 4.

2 Preliminaries

A *finite-state machine (FSM)* is a tuple $G = \langle \Sigma, Q, Q^\circ, Q^\omega, \to \rangle$, where Σ is a finite set of *events*, Q is a finite set of *states*, $Q^\circ \subseteq Q$ is the set of *initial states*, $Q^\omega \subseteq Q$ is the set of *accepting states*, and $\to \subseteq Q \times \Sigma \times Q$ is the *transition relation*.

The transition relation is written in infix notation, where $x \xrightarrow{\sigma} y$ means the existence of a transition from state $x \in Q$ to $y \in Q$ with event $\sigma \in \Sigma$. This notation is extended to *traces* $s \in \Sigma^*$ in the standard way. Given state sets $X, Y \subseteq Q$, the notation $X \xrightarrow{s} Y$ means $x \xrightarrow{s} y$ for some states $x \in X$ and $y \in Y$, and $X \xrightarrow{s}$ means $X \xrightarrow{s} Q$, and $X \to Y$ means $X \xrightarrow{s} Y$ for some $s \in \Sigma^*$. Events not in the event set of an FSM are assumed to be always enabled without state change, so the transition relation is further extended by letting $x \xrightarrow{\sigma} x$ for all $x \in Q$ and $\sigma \notin \Sigma$.

The FSM G is *deterministic* if $|Q^\circ| \leq 1$ and if $x \xrightarrow{\sigma} y_1$ and $x \xrightarrow{\sigma} y_2$ always implies $y_1 = y_2$. The *language* of G is $\mathcal{L}(G) = \{\, s \in \Sigma^* \mid Q^\circ \xrightarrow{s} \,\}$, and its *accepting language* is $\mathcal{L}^\omega(G) = \{\, s \in \Sigma^* \mid Q^\circ \xrightarrow{s} Q^\omega \,\}$. The *prefix-closure* of a language $L \subseteq \Sigma^*$ is $\mathrm{pre}(L) = \{\, s \in \Sigma^* \mid st \in L \text{ for some } t \in \Sigma^* \,\}$.

FSMs are synchronised in lock-step [9]. The *synchronous composition* of two FSMs

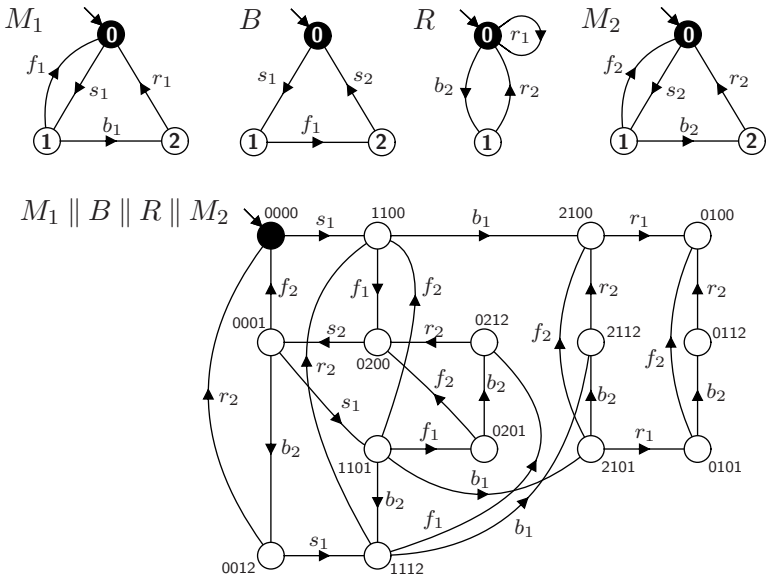

Figure 1: Blocking small factory.

$G_1 = \langle \Sigma_1, Q_1, Q_1^\circ, Q_1^\omega, \rightarrow_1 \rangle$ and $G_2 = \langle \Sigma_2, Q_2, Q_2^\circ, Q_2^\omega, \rightarrow_2 \rangle$ is

$$G_1 \parallel G_2 = \langle \Sigma_1 \cup \Sigma_2, Q_1 \times Q_2, Q_1^\circ \times Q_2^\circ, Q_1^\omega \times Q_2^\omega, \rightarrow \rangle$$

where $(x_1, x_2) \xrightarrow{\sigma} (y_1, y_2)$ if $x_1 \xrightarrow{\sigma}_1 y_1$ and $x_2 \xrightarrow{\sigma}_2 y_2$.

Example 1 Figure 1 shows a faulty version of the "small factory" system [22], modelled with four FSMs M_1, B, R, and M_2, and its synchronous composition. The synchronised states represent combinations of the states each FSM is in. For example, $\mathsf{1100}$ is the state tuple $(\mathsf{1}, \mathsf{1}, \mathsf{0}, \mathsf{0})$, where M_1 and B are in their states $\mathsf{1}$ and R and M_2 are in their states $\mathsf{0}$.

This working paper is concerned with the nonblocking property of the synchronous composition of several FSMs, which is also referred to as nonconflicting. An FSM is nonblocking, if accepting states are reachable from all reachable states. More precisely, given $G = \langle \Sigma, Q, Q^\circ, Q^\omega, \rightarrow \rangle$, a state $x \in Q$ is *reachable* in G if $Q^\circ \rightarrow x$, and *coreachable* in G of $x \rightarrow Q^\omega$.

Definition 1 An FSM G is nonblocking if and only if every reachable state in G is also coreachable in G. FSMs $G_1, \ldots, G_n$ are *nonconflicting* if their synchronous composition $G_1 \parallel \cdots \parallel G_n$ is nonblocking.

Example 2 State 1100 in Figure 1 is coreachable because $1100 \xrightarrow{f_1 s_2 f_2} 0000 \in Q^\omega$, but states 0100 and 2100 are not coreachable. By Definition 1, this FSM is *blocking*, and thus M_1, B, R, and M_2 are *conflicting*.

If G is a deterministic FSM, then its nonblocking property can be stated equivalently in language-based form as $\text{pre}(\mathcal{L}^\omega(G)) = \mathcal{L}(G)$ [22]. Then a system G is nonblocking, or nonconflicting, if every trace in its language $\mathcal{L}(G)$ can be continued to some trace in its accepting language $\mathcal{L}^\omega(G)$.

This working paper distinguishes two kinds of blocking or conflict. A state $x \in Q$ is a *deadlock* state if $x \notin Q^\omega$ and for all $y \in Q$ such that $x \to y$ it holds that $x = y$, i.e., x is a non-accepting state with only selfloop transitions outgoing. For example, 0100 in Figure 1 is a deadlock state. This is different from the usual definition [9], because here a deadlock state can have selfloops. A state $x \in Q$ is a *livelock* state if x is not coreachable and not a deadlock state.

3 Nonblocking Verification Algorithms

Based on Definition 1, a conflict check can be done by exploring the reachable and coreachable states of a synchronous composition. Section 3.1 below shows how to compute the set of reachable states, and Section 3.2 shows how to find the coreachable states. Afterwards, Section 3.3 describes an alternative that avoids computing coreachable states, and Section 3.4 discusses counterexample computation.

3.1 Exploring Reachable States

A basic algorithm to explore the synchronous composition of n deterministic FSMs with exactly one initial state is shown in Algorithm 1. It performs a standard search, using two state sets Q and *Open*. The state set Q collects all state tuples of the synchronous composition, while *Open* contains states that are yet to be expanded.

First, lines 1–2 add the initial state tuple to both Q and *Open*. Then the loop in line 3 expands each state tuple x in *Open*. For each event σ, line 6 determines whether it is enabled in x, and if so, line 7 computes the successor state y. If y is not already contained in the state space Q, then it is a new state and line 9 adds it to Q and to *Open* for subsequent expansion. What information about transitions, if any, is recorded in line 11 depends on the second pass of the algorithm, as discussed in Section 3.2 below. When Algorithm 1 terminates, Q contains all reachable state tuples.

Algorithm 1 works for deterministic FSMs. If the FSMs to be composed are nondeterministic, then all initial state combinations of $Q_1^\circ \times \cdots \times Q_n^\circ$ must be generated and enqueued

Algorithm 1: Reachability

Input: Deterministic FSMs $G_i = \langle \Sigma_i, Q_i, \{x_i^\circ\}, Q_i^\omega, \rightarrow_i \rangle$ for $1 \leq i \leq n$
Output: Reachable states Q

```
1  x° ← (x_0°, ..., x_n°);
2  Q ← {x°}; Open ← {x°};
3  while Open ≠ ∅ do
4  |  remove x = (x_0, ..., x_n) from Open;
5  |  foreach σ ∈ Σ_1 ∪ ··· ∪ Σ_n do
6  |  |  if x_i -σ->_i y_i for each 1 ≤ i ≤ n then
7  |  |  |  y ← (y_1, ..., y_n);
8  |  |  |  if y ∉ Q then
9  |  |  |  |  Q ← Q ∪ {y}; Open ← Open ∪ {y};
10 |  |  |  end
11 |  |  |  record transition x -σ-> y;
12 |  |  end
13 |  end
14 end
```

in lines 1–2, and likewise several successor state combinations y must be considered in lines 7–11.

Data structures The memory requirements of Algorithm 1 are determined by the state set Q and the queue *Open*. They contain state tuples, which can be bit-packed to save memory. The states in Figure 1 can be encoded using one or two bits for each FSM, e.g., the tuple 2100 can be stored as the binary string 10 01 0 00. This working paper's implementation packs each tuple into the smallest possible number of 32-bit words.

All state tuples in Q are stored consecutively in a growing array list, so they can be uniquely identified by their *index*. The first initial state tuple has index 0, the next tuple encountered has index 1, etc. A hash table [4] facilitates the membership test in line 8. The hash table contains only state indices, with the hash function computed using the actual state tuples from the list. The set *Open* is better implemented as a queue. For example, a first-in-first-out queue ensures that states are processed in the order in which they are discovered, resulting in breadth-first search [4]. In fact, *Open* is represented as a single integer, namely the index of the first state tuple that has not yet been expanded.

Thus, the amount of memory grows linearly with the number $|Q|$ of reachable states in the synchronous composition and with the number n of FSMs. The space complexity

```
 1 case M_1 goto 2,5,9
 2 if-disabled B, s_1 goto 11
 3 execute s_1
 4 goto 11
 5 execute b_1
 6 if-disabled B, f_1 goto 11
 7 execute f_1
 8 goto 11
 9 if-disabled R, r_1 goto 11
10 execute r_1
11 case M_2 goto 12,15,19
12 if-disabled B, s_2 goto 21
13 execute s_2
14 goto 21
15 execute f_2
16 if-disabled R, b_2 goto 21
17 execute b_2
18 goto 21
19 if-disabled R, r_2 goto 21
20 execute r_2
21 end
```

Figure 2: Branching program for Figure 1.

of Algorithm 1 is $O(n|Q|)$. In practice, each state typically requires 1–4 words for the bit-packed tuple plus two words in the hash table with 50% load, i.e., 12–24 bytes per state.

State expansion The main factor influencing the runtime of Algorithm 1 is the loop in line 5 and the if-statement in line 6, which is executed for each state and event in the system. It is important that the test for $x_i \xrightarrow{\sigma}_i y_i$ is evaluated in constant time, which is achieved with an array structure set up in advance. The array contains, for each source state and event, either a null-value to indicate that the event is disabled, or the successor state, or a reference to a list of successor states in the case of nondeterminism.

Even more important is the observation that usually less than 10% of events tested for eligibility produce a transition. Performance is improved substantially by making the failing tests fail fast. This is achieved with pre-calculated lists of the FSMs that can disable a given event σ. For example, event r_1 in Figure 1 only appears in M_1 and R, so only the states of these FSMs are tested to determine whether r_1 is enabled. Further, r_1 is enabled in only 1 out of 3 states of M_1 as opposed to 1 out of 2 states of R. The average number of state tests is reduced by first testing M_1 and then R, i.e., the FSMs should be sorted by event probability. The successor state computation in line 7 only starts after the event has been found to be enabled. In case of nondeterminism, a lot of repetition is avoided by first computing the state components of FSMs that do not synchronise or have only one successor state with the current event.

While the above approach with per-event FSM lists sorted by enablement probability usually works well, its performance degrades for models with hundreds or thousands of events. Further improvement is possible because, in many models, several events can be

ruled out as disabled after testing the state of a single FSM. For example, all enablement decisions for Figure 1 can be made using the *branching program* in Figure 2. Here, line 1 branches to line 2, 5, or 9 depending on whether M_1 is in state 0, 1, or 2. If M_1 is in state 0, e.g., then M_1 disables f_1, b_1, and r_1, and the only event from M_1's event set that could be enabled is s_1. Therefore line 2 checks whether s_1 is disabled by B, using array lookup, and if it is, the program continues in line 11 to check the remaining events s_2, f_2, b_2, and r_2. Otherwise s_1 is enabled, and line 3 performs the successor state computation as per lines 7–11 of Algorithm 1, before continuing in line 11. Overall, this program inspects the states of only four FSMs to make enablement decisions for eight events. This working paper's implementation generates a branching program similar to Figure 2 for its input FSMs and encodes it as primitive bytecode. Then the execution of the bytecode replaces the loop in line 5 of Algorithm 1.

Complexity In the worst case, line 6 checks enablement of all events in all component FSMs, for each state in the synchronous composition, which gives up to $n|\Sigma||Q|$ operations. Lines 7–11 are executed once per transition of the synchronous composition, which may exceed $n|\Sigma||Q|$ in the nondeterministic case. Therefore, the worst-case time complexity of Algorithm 1 is $O(n|\Sigma||Q| + |\rightarrow|)$. The branching program often significantly reduces the $n|\Sigma||Q|$ part while maintaining the same worst-case complexity.

Early termination Algorithm 1 can terminate early in the case of deadlock. After the loop in line 5, if no successors of the expanded state x have been found, or all successors are equal to x, then it can be checked whether x is accepting. If it is not, then x is a reachable deadlock state and the system is conflicting. Only if Algorithm 1 reaches the end without terminating early, a second pass is needed to determine whether the system has a livelock or is nonconflicting.

3.2 Finding Coreachable States

Algorithm 2 continues after Algorithm 1 and performs a backwards search to find all coreachable states. This is a standard model checking algorithm to verify the CTL property **EF** *accepting* [3]. The loop in line 2 marks as coreachable all reachable states that are accepting, and the loop in line 8 adds to this all predecessors of states already marked as coreachable.

This approach uses one bit per state to record whether or not it is coreachable, which is stored conveniently as part of the bit-packed state tuples. Unlike Algorithm 1, the set *Open* of unvisited states must be stored explicitly as a queue or stack of state numbers,

Algorithm 2: Coreachability

Input: FSMs $G_i = \langle \Sigma_i, Q_i, Q_i^\circ, Q_i^\omega, \to_i \rangle$,
reachable states Q

Output: States marked as coreachable or not

```
1  Open ← ∅;
2  foreach x ∈ Q do
3  |  if x ∈ Q_1^ω × ··· × Q_n^ω then
4  |  |  Open ← Open ∪ {x};
5  |  |  mark x as coreachable;
6  |  end
7  end
8  while Open ≠ ∅ do
9  |  remove y from Open;
10 |  foreach transition x → y do
11 |  |  if x is not marked as coreachable then
12 |  |  |  Open ← Open ∪ {x};
13 |  |  |  mark x as coreachable;
14 |  |  end
15 |  end
16 end
```

and its size is bounded by the number of states. Thus, the memory requirements from Algorithm 1 increase by 33 bits or 4 bytes and 1 bit per state. This increase remains linear in the number of states, $O(|Q|)$.

Upon termination of Algorithm 2, exactly the coreachable states are marked, so a final loop can determine whether all reachable states in Q are coreachable. If so, the system is nonconflicting, otherwise it is conflicting.

Line 10 requires the ability to iterate over the predecessor states of a given state y in the synchronous composition. This requires additional data structures or computational effort. The following two alternatives have been implemented.

Stored backwards transitions As Algorithm 1 explores the transitions, it can set up a data structure for the backwards search. The loop in line 10 of Algorithm 2 requires transitions indexed by target states. As the number of transitions per target state is not known a-priori, they are stored in linked lists [4]. Each target state is associated with the first node of its list of incoming transitions, and each node contains the source state of a transition and a reference to the next node. Events are not stored because Algorithm 2 does not need them. Thus, each transition occupies two words or eight bytes. Each time Algorithm 1 executes line 11, the current state x is prepended to the predecessors list of the target state y. Some memory is saved by suppressing selfloops and duplicate transitions, which do not affect the result of the conflict check. Duplicates are recognised in constant time, because all transitions $x \to y$ are encountered while expanding x, when x, if it is already listed as predecessor of y, must be the first entry of the list.

The time to construct the predecessor lists is bounded by the number $|\to|$ of transitions in the synchronous composition, and so is the runtime of Algorithm 2 using these lists. The complexity of this conflict check is dominated by Algorithm 1, $O(n|\Sigma||Q| + |\to|)$. However, the predecessor lists require additional memory proportional to the number of transitions, so the space complexity increases to $O(n|Q| + |\to|)$. This is a serious problem for large systems, as the number of transitions typically is at least ten times the number of states.

Computed backwards transitions It is also possible to find predecessor states by expanding the transition relation backwards. After reversing the direction of all transitions of the component FSMs, a branching program like Figure 2 produces the predecessors of any given state tuple. Care needs be taken as the reverse transition relation may be nondeterministic even if all FSMs are deterministic. A more serious problem arises because backwards exploration may lead to unreachable states. Every predecessor state must be checked for reachability using the hash table from Algorithm 1, so only reachable states are enqueued and marked in lines 12–13 of Algorithm 2.

No transitions are stored with this approach, so the space complexity of the conflict check remains $O(n|Q|)$. The time complexity of Algorithm 2 with computed backwards transitions is $O(n|\Sigma||Q| + |\rightarrow|)$ for the same reasons as explained above for Algorithm 1, which also remains the worst-case time complexity of the conflict check. However, now $|\rightarrow|$ includes some unreachable transitions. This rarely is a problem in practice, except for a few models with interrupt or reset events where the extra transitions outweigh the reachable transitions by several orders of magnitude.

3.3 Strongly Connected Components

This section proposes an alternative conflict check algorithm based on strongly connected components. Given an FSM $G = \langle \Sigma, Q, Q^\circ, Q^\omega, \rightarrow \rangle$, a *strongly connected component* in G is a maximal set $C \subseteq Q$ of states such that, for all states $x, y \in C$ it holds that $x \rightarrow y$. A strongly connected component $C \subseteq Q$ is a *leaf component*, if for all $y \in Q$ such that $C \rightarrow y$ it holds that $y \in C$.

A strongly connected component is a set of states all reachable from each other, and a leaf component is a strongly connected component from which only states in that component can be reached. The FSM $M_1 \parallel B \parallel R \parallel M_2$ in Figure 1 has seven strongly connected components, e.g., $\{0000, 0001, 0012, 0200, 0201, 0212, 1100, 1101, 1112\}$ and $\{2100\}$. Its only leaf component is $\{0100\}$.

It is known from graph theory that, for any state there exists a leaf component reachable from that state. Therefore, an FSM is blocking if and only if it has a blocking leaf component, i.e., a leaf component without any accepting states. This observation leads to the following result.

Proposition 1 An FSM $G = \langle \Sigma, Q, Q^\circ, Q^\omega, \rightarrow \rangle$ is nonblocking if and only if, for all leaf components $C \subseteq Q$ it holds that $C \cap Q^\omega \neq \emptyset$.

Tarjan's algorithm [27] is a popular method to find strongly connected components. It performs a single search over all transitions and outputs strongly connected components as it runs. Using it, Algorithm 3 performs the complete conflict check by testing the conditions of Proposition 1 as new components are detected.

Algorithm 3 is defined by the recursive procedure explore(), which must be called exactly once for each initial state. This replaces both Algorithms 1 and 2. All state tuples are stored in a growing list Q, in the order in which they are detected, as explained in Section 3.1 above. The argument i to explore() is the index of a state in this list, and $tuple[i]$ is the state tuple retrieved from the list. In addition to this, Tarjan's algorithm associates with each state a so-called *lowlink*, which records (roughly) the index of the

Algorithm 3: Conflict Check with Tarjan's Algorithm

```
1  procedure explore(state index i)
2  |  lowlink[i] ← i; stack.push(i); x ← tuple[i];
3  |  foreach transition x → y do
4  |  |  j ← index of y in Q;
5  |  |  if j undefined then
6  |  |  |  Q ← Q ∪ {y}; j ← index of y in Q;
7  |  |  |  explore(j);
8  |  |  |  lowlink[i] ← min(lowlink[i], lowlink[j]);
9  |  |  else if stack contains j then
10 |  |  |  lowlink[i] ← min(lowlink[i], j);
11 |  |  end
12 |  end
13 |  if lowlink[i] = i then
14 |  |  comp ← ∅;
15 |  |  repeat
16 |  |  |  j ← stack.pop(); comp ← comp ∪ {j};
17 |  |  until i = j;
18 |  |  foreach j ∈ comp do
19 |  |  |  if tuple[j] ∈ Q_1^ω × ··· × Q_n^ω then
20 |  |  |  |  return
21 |  |  |  else if ∃k : tuple[j] → tuple[k] ∧ k ∉ comp then
22 |  |  |  |  return
23 |  |  |  end
24 |  |  end
25 |  |  stop "The system is conflicting";
26 |  end
27 end
```

smallest state known to be in the same component. It also uses a *stack* containing the indices of all states not yet assigned to any component.

Lines 2–12 in Algorithm 3 perform depth-first search as prescribed by Tarjan's algorithm. Each state is pushed on the *stack*, before the loop in line 3 expands it as described in Section 3.1. New states are added to the list Q and explored recursively, updating the *lowlink*. Tarjan's algorithm ensures that, if line 13 is reached and the *lowlink* of state number i is still i, then this is the *root state* of a strongly connected component, which includes all states after it on the *stack* [27]. The loop in line 15 adds these states to a temporary set *comp*, and the loop in line 18 checks whether it is a blocking leaf component. If the component is not blocking because it contains an accepting state (line 19), or not a leaf because it has a transition to another component (line 21), the call to explore() returns to the previous level of recursion. Otherwise, a blocking leaf component has been found and line 25 terminates the algorithm early.

The appeal of this algorithm is that it explores transitions only in the forward direction and allows early termination even in case of livelock. Each state is expanded once in line 3 and possibly a second time in line 21. The second expansion is often skipped in practice by stopping the loop in line 18 early. Furthermore, Tarjan's algorithm guarantees that the first component detected is a leaf, so the second expansion can be avoided entirely in the common case of systems with only one component. The worst-case time complexity remains $O(n|\Sigma||Q| + |{\to}|)$.

The recursive control structure of Tarjan's algorithm is problematic for large systems. The depth of recursion is only bounded by the number of states, $|Q|$, and the location of the recursive call in the middle of state expansion requires a lot of context information to be stored for recursive calls, considerably increasing memory consumption or causing stack overflow. Standard iterative algorithms [4] to compute strongly connected components require both forward and backward transitions. This working paper is based on an iterative implementation of Tarjan's algorithm described in Appendix A, which uses a second stack for pending recursive calls. Stack size is bounded by the number of states, and the worst-case memory overhead is 20 bytes per state. The algorithm retains both the linear space complexity in the number of states and the linear time complexity in the number of transitions.

3.4 Counterexamples

Counterexamples are of great value to users of model checkers, because they explain a detected problem and facilitate the finding of a fix. A *counterexample* to the nonblocking property of an FSM $G = \langle \Sigma, Q, Q^\circ, Q^\omega, {\to} \rangle$ is a trace $s \in \Sigma^*$, together with state information in case of nondeterminism, which takes the system to a non-coreachable state, i.e.,

$Q^\circ \xrightarrow{s} x \not\to Q^\omega$. As the counterexample is presented to users, it should be as short as possible, and counterexamples that end in a deadlock state or a blocking leaf component, are more specific and thus more helpful.

Counterexample computation is straightforward if breadth-first search is used in Algorithm 1. The end state of the counterexample is either the deadlock state that triggered early termination, or after Algorithm 2, the non-coreachable state with the smallest index in breadth-first order. From the end state, the trace is constructed backwards until an initial state is reached, either using stored transitions or by backwards expansion. To get a shortest trace, the first predecessor in breadth-first order must be used at each step, which is either the first state added to the list, or the predecessor with the smallest index. When using transition lists without stored events, the event is obtained by expanding the predecessor state a second time forwards. This method expands one state for each step of the trace, whose length can be the total number of states in the worst case. It usually is shorter in practice, so that the overhead for counterexample computation is insignificant. The counterexample is guaranteed to be a shortest trace to a deadlock state, if one exists, and otherwise a shortest trace to a livelock state.

The same method can be used to compute a counterexample after Algorithm 3, but state numbers in depth-first order do not guarantee a shortest trace and the counterexamples are often unusably long. To improve counterexample quality, this working paper's implementation performs a breadth-first search to find a shortest trace to the first blocking leaf component encountered, visiting only states discovered in the depth-first search, and then constructs the counterexample as explained above. This results in overhead of expanding all discovered states up to two more times. While the counterexample is not necessarily the shortest possible, it usually is a fair compromise, and is guaranteed to end in a blocking leaf component.

4 Experimental Results

The algorithms described above are implemented as part of the *Waikato Analysis Toolkit for Events in Reactive Systems (Waters)*. The Waters library is programmed in C++ and released under the GNU Public License. It is available as part of Supremica [1]. This implementation has been used to check the nonconflicting property of 15 discrete event systems models from industrial applications and case studies. Appendix B gives brief descriptions of these models.

Table 1 shows the results of the experiments. It shows for each model, the number of events ($|\Sigma|$), the number of FSMs (n), the number of bits to encode the state tuples (Enc), the number of reachable states in the synchronous composition (State space), and the

Table 1: Experimental results

Model						Stored			Computed			Tarjan			BDD		
Name	$\lvert\Sigma\rvert$	n	Enc	State space	Verification result	Time [s]	Mem [MiB]	CE	Time [s]	Mem [MiB]	CE	Time [s]	Mem [MiB]	CE	Time [s]	Mem [MiB]	CE
agv	53	16	40	$2.57 \cdot 10^7$	nonconflicting	33.2	1493.5		55.8	590.9		38.2	842.4		0.1	32.0	
agvb	53	17	42	$2.29 \cdot 10^7$	deadlock	4.7	394.1	56	3.9	208.4	56	0.0	80.9	92	0.1	31.6	56
aip0tough_abs1	44	6	64	$1.02 \cdot 10^8$	livelock	84.4	3763.4	143	69.2	1364.2	143	2.6	142.4	153	37.1	63.8	161
aip1efa $\langle 2 \rangle$	94	50	112	$3.55 \cdot 10^7$	nonconflicting	89.7	2919.8		259.2	1372.4		75.6	1708.5		23.6	17.3	
aip1efa16_abs1	32	5	53	$3.13 \cdot 10^{10}$	deadlock	39.4	1748.7	91	32.2	665.1	91	2.0	166.4	178	34.2	93.0	101
big_bmw	66	31	57	$3.14 \cdot 10^7$	nonconflicting	105.4	3012.3		692.8	608.0		92.7	854.1		0.1	8.2	
dynamic_prime_sieve $\langle 5 \rangle$	4535	25	168	$6.79 \cdot 10^7$	nonconflicting	451.0	5722.0		1000.8	3653.1		457.9	3873.0		53.4	327.2	
fencaiwon09	73	32	79	$1.03 \cdot 10^8$	nonconflicting	212.3	6861.9		485.2	2395.3		168.0	3146.8		1.2	10.0	
fencaiwon09b	73	31	75	$8.93 \cdot 10^7$	deadlock	134.5	6134.7	267	108.1	2388.2	267	0.0	82.5	272	0.4	9.4	267
fencaiwon09s	73	29	67	$3.00 \cdot 10^8$	deadlock	0.5	119.0	41	0.4	99.9	41	0.0	84.5	41	0.1	32.0	41
ftechnik	117	36	120	$1.21 \cdot 10^8$	livelock	Out of memory			448.9	2985.8	0	0.0	115.9	20	0.1	24.4	0
profisafe_i4_abs1	50	4	45	$4.92 \cdot 10^7$	nonconflicting	125.4	3796.7		186.5	1373.6		116.4	1862.0		324.9	197.6	
profisafe_ihost_efa_2 $\langle 24 \rangle$	498	21	37	$6.69 \cdot 10^7$	nonconflicting	Out of memory			2818.0	1288.0		1820.8	1821.7		4.8	78.2	
tbed_reset1	194	98	254	664128	nonconflicting	0.9	154.6		3.4	141.6		0.9	157.2		613.2	17.8	
verriegel2	88	41	70	$2.18 \cdot 10^7$	nonconflicting	71.9	2146.6		Timeout			48.9	797.0		1.0	12.0	

verification result. Then it shows the runtime (Time), the memory consumption (Mem), and if applicable the length of the counterexample (CE), for four conflict check algorithms.

Stored is the combination of Algorithms1 and 2 with stored backwards transitions, while **Computed** expands states backwards. **Tarjan** is Algorithm 3. To put the data in perspective, the **BDD** column shows the results of an algorithm based on binary decision diagrams (BDD) [3]. This is a symbolic version of Algorithms1 and 2, tuned for better performance on discrete event systems models. It uses a variable ordering based on the FORCE heuristics [2] and a disjunctive partitioning and search strategy [25]. It terminates early in case of deadlock but not livelock, and does not ensure shortest counterexamples. BDD-based breadth-first search gives shortest counterexamples but runs up to ten times slower.

All experiments were run on a standard PC with a 3.3 GHz microprocessor and 8 GiB of RAM. The table shows that this is enough to explore state spaces with 100 million states completely within minutes. **Computed** is the most memory-efficient of the explicit algorithms, but it can be slow and was aborted after failing to solve the **verriegel2** model in an hour. This model has prohibitively many backwards transitions from unreachable states. **Stored** is faster except in case of deadlock, but requires the most memory. The **ftechnik** and **profisafe_ihost_efa_2** $\langle 24 \rangle$ models have too many transitions for this algorithm. **Tarjan** usually runs faster than **Stored** and **Computed**, particularly when it terminates early due to livelock, but uses more memory than **Computed** and sometimes produces longer counterexamples.

The focus of this working paper is to compare the explicit algorithms and not necessarily to beat symbolic methods. The **BDD** algorithm usually uses far less memory and runs faster than the explicit algorithms, but there are exceptions. The **tbed_reset1** model has many FSMs and requires many bits to encode, and **aip1efa16_abs1** and **profisafe_i4_abs1** have only few FSMs with thousands of states each. Such models can be difficult to encode as BDDs, causing the **BDD** algorithm to struggle, while their relatively small state numbers in combination with early termination make them amenable for explicit methods. The models **aip1efa16_abs1** and **profisafe_i4_abs1** are results of compositional minimisation of much larger models [21], which is a potential application of explicit algorithms.

5 Conclusions

Three explicit algorithms to verify the nonconflicting property of discrete event systems have been described and compared. These algorithms maintain linear time and space complexity in the size of the synchronous composition to be explored. They are optimised

differently for runtime or memory usage and implemented in C++. Experimental results show that the implementation exhaustively explores state spaces of more than 100 million states on standard computers within minutes.

References

[1] Åkesson, K., Fabian, M., Flordal, H., Malik, R.: Supremica—an integrated environment for verification, synthesis and simulation of discrete event systems. In: 8th Int. Workshop on Discrete Event Systems, WODES '06, pp. 384–385. IEEE (2006). DOI 10.1109/WODES.2006.382401

[2] Aloul, F.A., Markov, I.L., Sakallah, K.A.: FORCE: A fast & easy-to-implement variable-ordering heuristic. In: 13th ACM Great Lakes Symp. VLSI, pp. 116–119 (2003). DOI 10.1145/764808.764839

[3] Bérard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., Schnoebelen, P.: Systems and Software Verification. Springer (2001)

[4] Cormen, T.H., Leiserson, C.E., Rivest, R.E., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press (2009)

[5] Dierikx, O.L.E.: Fault-tolerance of a DES supervisor for a manufacturing testbed. Internship Report CST 2015 053, Eindhoven University of Technology, Eindhoven, The Netherlands (2015). URL `https://static.tue.nl/fileadmin/content/faculteiten/wtb/Onderzoek/Onderzoeksgroepen/Manufacturing_Networks/MN_Pdfs/Internships/2015_CST2015_053_Oriane_Dierikx.pdf`

[6] Feng, L., Cai, K., Wonham, W.M.: A structural approach to the non-blocking supervisory control of discrete-event systems. Int. J. Adv. Manuf. Technol. **41**, 1152–1168 (2009). DOI 10.1007/s00170-008-1555-9

[7] Feng, L., Wonham, W.M.: TCT: A computation tool for supervisory control synthesis. In: 8th Int. Workshop on Discrete Event Systems, WODES '06, pp. 388–389. IEEE (2006). DOI 10.1109/WODES.2006.382399

[8] Flordal, H., Malik, R.: Compositional verification in supervisory control. SIAM J. Control Optim. **48**(3), 1914–1938 (2009). DOI 10.1137/070695526

[9] Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)

[10] Holzmann, G.J.: The SPIN model checker. IEEE Trans. Softw. Eng. **23**, 279–295 (1997)

[11] Leduc, R.J.: Hierarchical interface-based supervisory control. Ph.D. thesis, Dept. of Electrical Engineering, University of Toronto, ON, Canada (2002). URL `http://www.cas.mcmaster.ca/~leduc`

[12] Lötzbeyer, A., Mühlfeld, R.: Task description of a flexible production cell with real time properties. Tech. rep., FZI, Karlsruhe, Germany (1996). URL `http://www.fzi.de/divisions/prost/projects/korsys/korsys.html`

[13] Malik, P.: From supervisory control to nonblocking controllers for discrete event systems. Ph.D. thesis, University of Kaiserslautern, Kaiserslautern, Germany (2003)

[14] Malik, R.: Programming a fast explicit conflict checker. In: 13th Int. Workshop on Discrete Event Systems, WODES'16, pp. 464–469. IEEE (2016). DOI 10.1109/WODES.2016.7497885

[15] Malik, R., Mühlfeld, R.: A case study in verification of UML statecharts: the PROFIsafe protocol. J. Universal Computer Science **9**(2), 138–151 (2003). URL `http://www.jucs.org/jucs_9_2/a_case_study_in`

[16] Moody, J.O., Antsaklis, P.J.: Supervisory Control of Discrete Event Systems Using Petri Nets. Kluwer (1998)

[17] Moor, T., Schmidt, K., Perk, S.: libFaudes — an open source C++ library for discrete event systems. In: 9th Int. Workshop on Discrete Event Systems, WODES'08, pp. 125–130. IEEE (2008). DOI 10.1109/WODES.2008.4605933

[18] Nuutila, E., Soisalon-Soininen, E.: On finding the strongly connected components in a directed graph. Information Processing Letters **49**(1), 9–14 (1994). DOI 10.1016/0020-0190(94)90047-7

[19] Pearce, D.J.: An improved algorithm for finding the strongly connected components of a directed graph (2005). URL `http://homepages.mcs.vuw.ac.nz/~djp/files/P05.ps`

[20] Pena, P.N., Cury, J.E.R., Lafortune, S.: Verification of nonconflict of supervisors using abstractions. IEEE Trans. Autom. Control **54**(12), 2803–2815 (2009). DOI 10.1109/TAC.2009.2031730

[21] Pilbrow, C., Malik, R.: An algorithm for compositional nonblocking verification using special events. Sci. Comput. Programming **113**(2), 119–148 (2015). DOI 10.1016/j.scico.2015.05.010

[22] Ramadge, P.J.G., Wonham, W.M.: The control of discrete event systems. Proc. IEEE **77**(1), 81–98 (1989). DOI 10.1109/5.21072

[23] KorSys Project: URL `http://www4.in.tum.de/proj/korsys/`

[24] Shaw, A.M.: Partial order reduction with compositional verification. Master's thesis, Dept. of Computer Science, University of Waikato (2014). URL `http://hdl.handle.net/10289/9001`

[25] Song, R.: Symbolic synthesis and verification of hierarchical interface-based supervisory control. Master's thesis, Dept. of Computing and Software, McMaster University, Hamilton, ON, Canada (2006). URL `http://www.cas.mcmaster.ca/~leduc`

[26] Su, R., van Schuppen, J.H., Rooda, J.E., Hofkamp, A.T.: Nonconflict check by using sequential automaton abstractions based on weak observation equivalence. Automatica **46**(6), 968–978 (2010). DOI 10.1016/j.automatica.2010.02.025

[27] Tarjan, R.: Depth first search and linear graph algorithms. SIAM J. Computing **1**(2), 146–160 (1972). DOI 10.1137/0201010

Appendices

A Iterative Implementation of Tarjan's Algorithm

This appendix describes the iterative implementation of Algorithm 3. This is an improved version of an algorithm proposed in [24], which is adapted to the situation in this paper where the transition relation is only available through forward exploration. Other improvements to Tarjan's algorithm are known [18, 19] but not easily adaptable to this case.

The main conflict check procedure is shown in Algorithm 4 and invokes two subroutines for state expansion and state closing in Algorithms 5 and 6. The recursion in Algorithm 3 is simulated by storing pending calls on a second stack, called *controlStack*. The *stack* of Algorithm 3 is also used and for distinction called component stack, *compStack*. States are added to the state set Q and pushed on the control stack as soon as they are discovered, and

Algorithm 4: Iterative version of Algorithm 3

Input: FSMs $G_i = \langle \Sigma_i, Q_i, Q_i^\circ, Q_i^\omega, \rightarrow_i \rangle$ for $1 \leq i \leq n$

```
1  foreach x° ∈ Q_1° × ··· × Q_n° do
2  |  Q ← Q ∪ {x°}; i ← index of x° in Q;
3  |  mode[i] ← OPEN;
4  |  controlStack.push(⟨i, i⟩);
5  end
6  while controlStack not empty do
7  |  ⟨i, p⟩ ← top entry of controlStack;
8  |  if mode[i] = OPEN then
9  |  |  mode[i] ← EXPANDED;
10 |  |  expand(i);
11 |  else if mode[i] = EXPANDED then
12 |  |  remove top entry from controlStack;
13 |  |  close(i, p);
14 |  end
15 end
16 stop "The system is nonconflicting";
```

Algorithm 5: State expansion subroutine

```
1  procedure expand(state index i)
2  |  dfs[i] ← size of compStack; lowlink[i] ← dfs[i];
3  |  compStack.push(i);
4  |  foreach transition tuple[i] → y do
5  |  |  j ← index of y in Q;
6  |  |  if j undefined then
7  |  |  |  Q ← Q ∪ {y}; j ← index of y in Q;
8  |  |  |  mode[j] ← OPEN;
9  |  |  |  controlStack.push(⟨j, i⟩);
10 |  |  else if mode[j] = OPEN then
11 |  |  |  find entry ⟨j, _⟩ on controlStack;
12 |  |  |  change ⟨j, _⟩ to ⟨j, i⟩;
13 |  |  |  move ⟨j, i⟩ to top of controlStack;
14 |  |  else if mode[j] = EXPANDED then
15 |  |  |  lowlink[i] ← min(lowlink[i], lowlink[j]);
16 |  |  end
17 |  end
18 end
```

Algorithm 6: State closing subroutine

```
1  procedure close(state index i, parent index p)
2      if lowlink[i] = dfs[i] then
3          comp ← ∅;
4          repeat
5              j ← compStack.pop(); comp ← comp ∪ {j};
6              mode[j] ← CLOSED;
7          until i = j;
8          foreach j ∈ comp do
9              if tuple[j] ∈ Q_1^ω × ··· × Q_n^ω then
10                 return
11             else if ∃k : tuple[j] → tuple[k] ∧ k ∉ comp then
12                 return
13             end
14         end
15         stop "The system is conflicting";
16     else
17         lowlink[p] ← min(lowlink[p], lowlink[i]);
18     end
19 end
```

expanded later. The positions of the state tuple in Q no longer represent depth-first search order and cannot be used to compare *lowlink* values. Instead, the iterative algorithm uses the position on the component stack, which is recorded for state number i as $dfs[i]$. As state tuples are discovered, expanded, and assigned to components, they go through the following four *modes*, recorded as $mode[i]$.

UNVISITED Unvisited states have not yet been encountered at all. They appear neither in the global state list Q nor on any stack. Such states are recognised by the state tuple's index in Q being undefined.

OPEN Open states have been discovered and queued for processing by the depth-first search. The state tuple appears in Q with index i, and the control stack has an entry $\langle i, p\rangle$ where p is the index of the parent from where expansion of the state was triggered. However, the state is not yet on the component stack. Open states are recognised by $mode[i] = \text{OPEN}$.

EXPANDED These states have been expanded or are in the process of being expanded by the depth-first search, but have not yet been assigned to a component. The state tuple appears in Q with index i, and the control stack may or may not have an entry $\langle i, p\rangle$ as above. The state does appear on the component stack, at the position recorded in $dfs[i]$, and $lowlink[i]$ contains the smallest component stack position of states known to be in the same component as per Tarjan's algorithm. Expanded states are recognised by $mode[i] = \text{EXPANDED}$.

CLOSED Closed states are fully expanded and assigned to a strongly connected component. The state tuple appears in Q with index i. Closed states appear on no stack and are recognised by $mode[i] = \text{CLOSED}$.

The loop in line 1 of Algorithm 4 marks all initial states as OPEN and puts them on the control stack. Control stack entries have the form $\langle i, p\rangle$ where i is the state index in Q and p is the index of the parent that triggers expansion. Initial states use their own index as fake parent. After the initialisation, the main loop in line 6 processes all control stack entries depending on their *mode*, which can be either OPEN or EXPANDED. OPEN states are changed to EXPANDED and expanded by Algorithm 5. EXPANDED states are removed from the control stack, and Algorithm 6 tries to close them.

Algorithm 5 expands states by performing (roughly) the operations in lines 2–12 of Algorithm 3. First, lines 2–3 put the state index on the component stack and record its position there in $dfs[i]$ and $lowlink[i]$. Then the loop in line 4 explores the successor states depending on their mode. Lines 6–9 add UNVISITED states to the state set Q, set them

to OPEN, and create control stack entries with the current state i as parent. Lines 10–13 handle OPEN states, i.e., states already on the control stack but not expanded. These states would be encountered here for the first time by the recursive Algorithm 3. They should be pushed on the control stack like UNVISITED states, but this results in duplicates. A simple solution [24] detects and removes such duplicates on second expansion, but then the control stack size can be quadratic in the number of states. To maintain linear space complexity, lines 11–13 move the existing entry to the top of the control stack, changing the parent to the current state i. Lines 14–15 process EXPANDED states by updating the *lowlink* as per line 10 in Algorithm 3. Note that the *lowlink* update of line 8 of Algorithm 3 cannot be performed in this subroutine, because states are only pushed on the control stack and not yet expanded. This is done while closing the successor state in line 17 of Algorithm 6.

Algorithm 6 is called after all successor states have been expanded and performs the operations in lines 8 and 13–25 of Algorithm 3. Line 2 recognises the root state of a strongly connected component when its position $dfs[i]$ on the component stack is equal to $lowlink[i]$. In this case, the states of the component are removed from the component stack and marked as CLOSED, and the check for a blocking leaf component is performed like in Algorithm 3.

The above description mentions arrays *mode* and *dfs* that do not appear in Algorithm 3. These do not need to be stored explicitly, as not all values are needed at all times. This working paper's implementation extends the state tuples by only one 32-bit word, called $link[i]$, which replaces the $lowlink[i]$ of Algorithm 3. To facilitate the rearrangement in lines 11–13 of Algorithm 5, the control stack is implemented as a linked list, where each stack entry $\langle i, p\rangle$ is extended with a 32-bit reference to the entry immediately below it on the stack. The component stack is an array list of 32-bit state numbers. The values of *mode*, *lowlink*, and *dfs* are encoded as follows.

- The modes OPEN and EXPANDED are distinguished by tagging the most significant bit of $link[i]$ and the first component of the control stack entries. CLOSED states are recorded by a special value -1 in $link[i]$.

- If state i is OPEN, then its control stack entry is $\langle i, p\rangle$ as described above, and $link[i]$ contains a reference to the entry immediately above this entry on the control stack. (The control stack always has one entry more than necessary, and the top is an empty dummy node.) This ensures that lines 11–13 of Algorithm 5 can locate and move the stack entry in constant time. The values $lowlink[i]$ and $dfs[i]$ are not needed for OPEN states.

- If state i is EXPANDED, then it is on the component stack at position $dfs[i]$, and its control stack entry is changed to $\langle dfs[i], p\rangle$. The state index i can be retrieved

from the component stack. The position on the control stack is no longer stored, and $link[i]$ contains the value of $lowlink[i]$.

This encoding requires one 32-bit word $link[i]$ per state, plus the two stacks. Control stack entries use three words for the state index, parent index, and reference to the next node. Component stack entries occupy one word. The upper limit for both stack sizes is the number of states, so the worst-case memory overhead is five words or 20 bytes per state. The bit tagging and special value for CLOSED reduce the maximum number of states that can be handled to $2^{31} - 1 = 2,147,483,647$.

There are no other significant overheads, so the iterative algorithm retains the linear space complexity in the number of states and the linear time complexity in the number of transitions of its recursive version Algorithm 3.

B Example Models

Following is a brief description of the models in Table 1.

agv Automated guided vehicle coordination models [16].

aip Models and abstractions of the automated manufacturing system of the Atelier Inter-établissement de Productique [11, 25].

big_bmw Automotive window lift controller model [13].

dynamic_prime_sieve Distributed prime number sieve software model.

fencaiwon09 Models of a production cell in a metal-processing plant [6].

ftechnik Flexible production cell model [12].

profisafe Parts or abstractions of PROFIsafe field bus protocol model [15].

tbed Fault-tolerant train testbed model [5].

verriegel Automotive central locking system, originally from the KORSYS project [23]. The model here is reduced to two doors.