Compositional Nonblocking Verification with Always Enabled Events and Selfloop-only Events

Colin Pilbrow Robi Malik

Department of Computer Science, University of Waikato, Hamilton, New Zealand
colinpilbrow@gmail.com robi@waikato.ac.nz

Abstract. This paper proposes to improve compositional nonblocking verification through the use of always enabled and selfloop-only events. Compositional verification involves abstraction to simplify parts of a system during verification. Normally, this abstraction is based on the set of events not used in the remainder of the system, i.e., in the part of the system not being simplified. Here, it is proposed to exploit more knowledge about the system and abstract events even though they are used in the remainder of the system. Abstraction rules from previous work are generalised, and experimental results demonstrate the applicability of the resulting algorithm to verify several industrial-scale discrete event system models, while achieving better state-space reduction than before.

1 Introduction

The *nonblocking property* is a weak liveness property commonly used in *supervisory control theory* of discrete event systems to express the absence of livelocks or deadlocks [6, 22]. This is a crucial property of safety-critical control systems, and with the increasing size and complexity of these systems, there is an increasing need to verify the nonblocking property automatically. The standard method to check whether a system is nonblocking involves the explicit composition of all the automata involved, and is limited by the well-known *state-space explosion* problem. *Symbolic model checking* has been used successfully to reduce the amount of memory required by representing the state space symbolically rather than enumerating it explicitly [2].

Compositional verification [10, 27] is an effective alternative that can be used independently of or in combination with symbolic methods. Compositional verification works by simplifying individual automata of a large synchronous composition, gradually reducing the state space of the system and allowing much larger systems to be verified in the end. When applied to the nonblocking property, compositional verification requires very specific abstraction methods [9, 17]. A suitable theory is laid out in [18], where it is argued that abstractions used in nonblocking verification should preserve a process-algebraic equivalence called *conflict equivalence*. Various abstraction rules preserving conflict equivalence have been proposed and implemented [9, 17, 20, 25].

Conflict equivalence is the most general process equivalence for use in compositional nonblocking verification [18]. If a part of a system is replaced by a conflict equivalent abstraction, the nonblocking property is guaranteed to be preserved independently of the other system components. While this is easy to understand and implement, more

simplification is possible by considering the other system components. This paper proposes simplification rules that take into account that certain events are always enabled or only selfloops in the rest of the system, and shows how this additional information can achieve further state-space reduction.

In the following, Section 2 introduces the background of nondeterministic automata, the nonblocking property, and conflict equivalence. Next, Section 3 describes compositional verification and always enabled and selfloop-only events. Section 4 presents simplification rules that exploit such events, and Section 5 shows how these events are found algorithmically. Afterwards, Section 6 presents the experimental results, and Section 7 adds concluding remarks. Further details and formal proofs of technical results can be found in [21].

2 Preliminaries

2.1 Events and Languages

Event sequences and languages are a simple means to describe discrete system behaviours [6, 22]. Their basic building blocks are *events*, which are taken from a finite *alphabet* $\mathbf{A}$. In addition, two special events are used, the *silent event* τ and the *termination event* ω. These are never included in an alphabet $\mathbf{A}$ unless mentioned explicitly using notation such as $\mathbf{A}_\tau = \mathbf{A} \cup \{\tau\}$, $\mathbf{A}_\omega = \mathbf{A} \cup \{\omega\}$, and $\mathbf{A}_{\tau,\omega} = \mathbf{A} \cup \{\tau, \omega\}$.

$\mathbf{A}^*$ denotes the set of all finite *traces* of the form $\sigma_1\sigma_2\cdots\sigma_n$ of events from $\mathbf{A}$, including the *empty trace* ε. The *concatenation* of two traces $s,t \in \mathbf{A}^*$ is written as st. A subset $L \subseteq \mathbf{A}^*$ is called a *language*. The *natural projection* $P\colon\ \mathbf{A}^*_{\tau,\omega} \to \mathbf{A}^*_\omega$ is the operation that deletes all silent (τ) events from traces.

2.2 Nondeterministic Automata

System behaviours are modelled using finite automata. Typically, system models are deterministic, but abstraction may result in nondeterminism.

Definition 1. A (nondeterministic) *finite automaton* is a tuple $G = \langle \mathbf{A}, Q, \to, Q^\circ \rangle$ where $\mathbf{A}$ is a finite set of *events*, Q is a finite set of *states*, $\to \subseteq Q \times \mathbf{A}_{\tau,\omega} \times Q$ is the *state transition relation*, and $Q^\circ \subseteq Q$ is the set of *initial states*.

The transition relation is written in infix notation $x \xrightarrow{\sigma} y$, and is extended to traces $s \in \mathbf{A}^*_{\tau,\omega}$ in the standard way. For state sets $X, Y \subseteq Q$, the notation $X \xrightarrow{s} Y$ means $x \xrightarrow{s} y$ for some $x \in X$ and $y \in Y$, and $X \xrightarrow{s} y$ means $x \xrightarrow{s} y$ for some $x \in X$. Also, $X \xrightarrow{s}$ for a state or state set X denotes the existence of a state $y \in Q$ such that $X \xrightarrow{s} y$.

The termination event $\omega \notin \mathbf{A}$ denotes completion of a task and does not appear anywhere else but to mark such completions. It is required that states reached by ω do not have any outgoing transitions, i.e., if $x \xrightarrow{\omega} y$ then there does not exist $\sigma \in \mathbf{A}_{\tau,\omega}$ such that $y \xrightarrow{\sigma}$. This ensures that the termination event, if it occurs, is always the final event of any trace. The traditional set of *terminal* states is $Q^\omega = \{\, x \in Q \mid x \xrightarrow{\omega} \,\}$ in this notation. For graphical simplicity, states in Q^ω are shown shaded in the figures of this paper instead of explicitly showing ω-transitions.

To support silent events, another transition relation $\Rightarrow \subseteq Q \times \mathbf{A}^*_\omega \times Q$ is introduced, where $x \overset{s}{\Rightarrow} y$ denotes the existence of a trace $t \in \mathbf{A}^*_{\tau,\omega}$ such that $P(t) = s$ and $x \xrightarrow{t} y$. That is, $x \xrightarrow{s} y$ denotes a path with *exactly* the events in s, while $x \overset{s}{\Rightarrow} y$ denotes a path with an arbitrary number of τ events shuffled with the events of s. Notations such as $X \overset{s}{\Rightarrow} Y$ and $x \overset{s}{\Rightarrow}$ are defined analogously to $\rightarrow$.

Definition 2. Let $G = \langle \mathbf{A}_G, Q_G, \rightarrow_G, Q^\circ_G \rangle$ and $H = \langle \mathbf{A}_H, Q_H, \rightarrow_H, Q^\circ_H \rangle$ be two automata. The *synchronous composition* of G and H is

$$G \parallel H \;=\; \langle \mathbf{A}_G \cup \mathbf{A}_H, Q_G \times Q_H, \rightarrow, Q^\circ_H \times Q^\circ_H \rangle \;, \tag{1}$$

where

- $(x_G, x_H) \xrightarrow{\sigma} (y_G, y_H)$ if $\sigma \in (\mathbf{A}_G \cap \mathbf{A}_H) \cup \{\omega\}$, $x_G \xrightarrow{\sigma}_G y_G$, and $x_H \xrightarrow{\sigma}_H y_H$;
- $(x_G, x_H) \xrightarrow{\sigma} (y_G, x_H)$ if $\sigma \in (\mathbf{A}_G \setminus \mathbf{A}_H) \cup \{\tau\}$ and $x_G \xrightarrow{\sigma}_G y_G$;
- $(x_G, x_H) \xrightarrow{\sigma} (x_G, y_H)$ if $\sigma \in (\mathbf{A}_H \setminus \mathbf{A}_G) \cup \{\tau\}$ and $x_H \xrightarrow{\sigma}_H y_H$.

Automata are synchronised using lock-step synchronisation [12]. Shared events (including ω) must be executed by all automata synchronously, while other events (including τ) are executed independently.

2.3 The Nonblocking Property

The key liveness property in supervisory control theory is the *nonblocking* property. An automaton is nonblocking if, from every reachable state, a terminal state can be reached; otherwise it is *blocking*. When more than one automaton is involved, it also is common to use the terms *nonconflicting* and *conflicting*.

Definition 3. [18] An automaton $G = \langle \mathbf{A}, Q, \rightarrow, Q^\circ \rangle$ is *nonblocking* if, for every state $x \in Q$ and every trace $s \in \mathbf{A}^*$ such that $Q^\circ \overset{s}{\Rightarrow} x$, there exists a trace $t \in \mathbf{A}^*$ such that $x \overset{t\omega}{\Rightarrow}$. Two automata G and H are *nonconflicting* if $G \parallel H$ is nonblocking.

To reason about conflicts in a compositional way, the notion of *conflict equivalence* is developed in [18]. According to process-algebraic testing theory, two automata are considered as equivalent if they both respond in the same way to tests [7]. For *conflict equivalence*, a *test* is an arbitrary automaton, and the *response* is the observation whether the test composed with the automaton in question is nonblocking or not.

Definition 4. [18] Two automata G and H are *conflict equivalent*, written $G \simeq_{\text{conf}} H$, if, for any automaton T, $G \parallel T$ is nonblocking if and only if $H \parallel T$ is nonblocking.

3 Compositional Verification

When verifying whether a composed system of automata

$$G_1 \parallel G_2 \parallel \cdots \parallel G_n \;, \tag{2}$$

is nonblocking, compositional methods [9,17] avoid building the full synchronous composition immediately. Instead, individual automata G_i are simplified and replaced by smaller conflict equivalent automata $H_i \simeq_{\text{conf}} G_i$. If no simplification is possible, a subsystem of automata $(G_j)_{j\in J}$ is selected and replaced by its synchronous composition, which then may be simplified.

The soundness of this approach is justified by the *congruence* properties [18] of conflict equivalence. For example, if G_1 in (2) is replaced by $H_1 \simeq_{\text{conf}} G_1$, then by considering $T = G_2 \parallel \cdots \parallel G_n$ in Def. 4, it follows that the abstracted system $H_1 \parallel T = H_1 \parallel G_2 \parallel \cdots \parallel G_n$ is nonblocking if and only if the original system (2) is.

Previous approaches for compositional nonblocking verification [9, 17] are based on *local* events. A component G_1 in a system such as (2) typically contains some events that appear only in G_1 and not in the remainder $T = G_2 \parallel \cdots \parallel G_n$ of the system. These events are called local and are abstracted using hiding, i.e., they are replaced by the silent event τ. Conflict equivalence uses τ as a placeholder for events not used elsewhere, and in this setting is the coarsest conflict-preserving abstraction [18].

Yet, in practice, the remainder $T = G_2 \parallel \cdots \parallel G_n$ is known. This paper proposes ways to use additional information about T to inform the simplification of G_1 and produce better abstractions. In addition to using the τ events, it can be examined how other events are used by T. There are two kinds of events that are easy to detect: *always enabled* events and *selfloop-only* events.

Definition 5. Let $G = \langle \mathbf{A}, Q, \rightarrow, Q^\circ \rangle$ be an automaton. An event $\sigma \in \mathbf{A}$ is *always enabled* in G, if for every state $x \in Q$ it holds that $x \overset{\sigma}{\Rightarrow}$.

An event is always enabled in an automaton if it can be executed from every state—possibly after some silent events. If during compositional verification, an event is found to be always enabled in every automaton except the one being simplified, this event has similar properties to a silent event. Several abstraction methods that exploit silent events to simplify automata can be generalised to exploit always enabled events also.

Definition 6. Let $G = \langle \mathbf{A}, Q, \rightarrow, Q^\circ \rangle$ be an automaton. An event $\sigma \in \mathbf{A}$ is *selfloop-only* in G, if for every transition $x \xrightarrow{\sigma} y$ it holds that $x = y$.

Selfloops are transitions that have the same start and end states. An event is selfloop-only if it only appears on selfloop transitions. As the presence of selfloops does not affect the nonblocking property, the knowledge that an event is selfloop-only can help to simplify the system beyond standard conflict equivalence. In the following definition, conflict equivalence is generalised by considering sets $\mathbf{E}$ and $\mathbf{S}$ of events that are always enabled or selfloop-only in the rest of the system, i.e., in the test T.

Definition 7. Let G and H be two automata, and let $\mathbf{E}$ and $\mathbf{S}$ be two sets of events. G and H are *conflict equivalent* with respect to $\mathbf{E}$ and $\mathbf{S}$, written $G \simeq_{\mathbf{E},\mathbf{S}} H$, if for every automaton T such that $\mathbf{E}$ is a set of always enabled events in T and $\mathbf{S}$ is a set of selfloop-only events in T, it holds that $G \parallel T$ is nonblocking if and only if $H \parallel T$ is nonblocking.

Clearly, standard conflict equivalence implies conflict equivalence with respect to $\mathbf{E}$ and $\mathbf{S}$, as the latter considers fewer tests T. Yet, both equivalences have the same useful properties for compositional nonblocking verification. The following results are immediate from the definition.

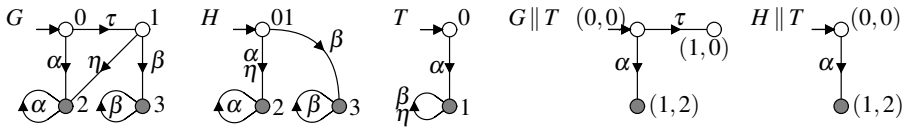

Fig. 1. Two automata G and H such that $G \simeq_{\{\eta\},\emptyset} H$ but not $G \simeq_{\text{conf}} H$.

Proposition 1. Let G and H be two automata.

(i) $G \simeq_{\text{conf}} H$ if and only if $G \simeq_{\emptyset,\emptyset} H$.
(ii) If $\mathbf{E} \subseteq \mathbf{E}'$ and $\mathbf{S} \subseteq \mathbf{S}'$ then $G \simeq_{\mathbf{E},\mathbf{S}} H$ implies $G \simeq_{\mathbf{E}',\mathbf{S}'} H$.

Proposition 2. Let $G_1, \ldots, G_n$ and H_1 be automata such that $G_1 \simeq_{\mathbf{E},\mathbf{S}} H_1$, where $\mathbf{E}$ and $\mathbf{S}$ are sets of events that respectively are always enabled and selfloop-only for $G_2 \parallel \cdots \parallel G_n$. Then $G_1 \parallel \cdots \parallel G_n$ is nonblocking if and only if $H_1 \parallel G_2 \parallel \cdots \parallel G_n$ is nonblocking.

Prop. 1 confirms that conflict equivalence with respect to $\mathbf{E}$ and $\mathbf{S}$ is coarser than standard conflict equivalence and considers more automata as equivalent. Thus, the modified equivalence has the potential to achieve better abstraction. At the same time, Prop. 2 shows that the modified equivalence can be used in the same way as standard conflict equivalence to replace automata in compositional verification, provided that suitable event sets $\mathbf{E}$ and $\mathbf{S}$ can be determined.

Example 1. Automata G and H in Fig. 1 are *not* conflict equivalent as demonstrated by the test automaton T. On the one hand, $G \parallel T$ is blocking because the state $(1,0)$ is reachable by τ from the initial state $(0,0)$, and $(1,0)$ is a deadlock state, because G disables event α in state 1 and T disables events β and η in state 0. On the other hand, $H \parallel T$ is nonblocking.

Note that η is not always enabled in T since $0 \stackrel{\eta}{\Rightarrow}_T$ does not hold. In composition with a test T that has η always enabled, G will be able to continue from state 1, and H will be able to continue from state 01. It follows from Prop. 4 below that $G \simeq_{\{\eta\},\emptyset} H$.

4 Simplification Rules

To exploit conflict equivalence in compositional verification, it is necessary to algorithmically compute a conflict equivalent abstraction of a given automaton. Several abstraction rules are known for standard conflict equivalence [9, 17]. This section generalises some of these and proposes four computationally feasible rules to simplify automata under the assumption of always enabled and selfloop-only events. Before that, Subsection 4.1 introduces general terminology to describe all abstractions.

4.1 Automaton Abstraction

A common method to simplify an automaton is to construct its *quotient* modulo an equivalence relation. The following definitions are standard.

An *equivalence relation* is a binary relation that is reflexive, symmetric and transitive. Given an equivalence relation $\sim$ on a set Q, the *equivalence class* of $x \in Q$ with respect to $\sim$, denoted $[x]$, is defined as $[x] = \{ x' \in Q \mid x' \sim x \}$. An equivalence relation on a set Q partitions Q into the set $Q/\!\sim = \{ [x] \mid x \in Q \}$ of its equivalence classes.

Definition 8. Let $G = \langle \mathbf{A}, Q, \rightarrow, Q^\circ \rangle$ be an automaton, and let $\sim \subseteq Q \times Q$ be an equivalence relation. The *quotient automaton* $G/\!\sim$ of G with respect to $\sim$ is $G/\!\sim = \langle \mathbf{A}, Q/\!\sim, \rightarrow/\!\sim, \tilde{Q}^\circ \rangle$, where $\tilde{Q}^\circ = \{ [x^\circ] \mid x^\circ \in Q^\circ \}$ and $\rightarrow/\!\sim = \{ ([x], \sigma, [y]) \mid x \xrightarrow{\sigma} y \}$.

When constructing a quotient automaton, classes of equivalent states in the original automaton are combined or *merged* into a single state. A common equivalence relation to construct quotient automata is *observation equivalence* or *weak bisimulation* [19].

Definition 9. [19] Let $G = \langle \mathbf{A}, Q, \rightarrow, Q^\circ \rangle$ be an automaton. A relation $\approx \subseteq Q \times Q$ is an *observation equivalence* relation on G if, for all states $x_1, x_2 \in Q$ such that $x_1 \approx x_2$ and all traces $s \in \mathbf{A}^*_\omega$ the following conditions hold:

(i) if $x_1 \stackrel{s}{\Rightarrow} y_1$ for some $y_1 \in Q$, then there exists $y_2 \in Q$ such that $y_1 \approx y_2$ and $x_2 \stackrel{s}{\Rightarrow} y_2$;
(ii) if $x_2 \stackrel{s}{\Rightarrow} y_2$ for some $y_2 \in Q$, then there exists $y_1 \in Q$ such that $y_1 \approx y_2$ and $x_1 \stackrel{s}{\Rightarrow} y_1$.

Two states are observation equivalent if they have got exactly the same sequences of enabled events, leading to equivalent successor states. Observation equivalence is a well-known equivalence with efficient algorithms that preserves all temporal logic properties [5]. In particular, an observation equivalent abstraction is conflict equivalent to the original automaton.

Proposition 3. [17] Let G be an automaton, and let $\approx$ be an observation equivalence relation on G. Then $G \simeq_{\text{conf}} G/\!\approx$.

A special case of observation equivalence-based abstraction is τ*-loop removal*. If two states are mutually connected by sequences of τ-transitions, it follows from Def. 9 that these states are observation equivalent, so by Prop. 3 they can be merged preserving conflict equivalence. This simple abstraction results in a τ*-loop free* automaton, i.e., an automaton that does not contain any proper cycles of τ-transitions.

Definition 10. Let $G = \langle \mathbf{A}, Q, \rightarrow, Q^\circ \rangle$ be an automaton. G is τ*-loop free*, if for every path $x \xrightarrow{t} x$ with $t \in \{\tau\}^*$ it holds that $t = \varepsilon$.

While τ-loop removal and observation equivalence are easy to compute and produce good abstractions, there are conflict equivalent automata that are not observation equivalent. Several other relations are considered for conflict equivalence [9, 17].

Definition 11. [9] Let $G = \langle \mathbf{A}, Q, \rightarrow, Q^\circ \rangle$ be an automaton. The *incoming equivalence* relation $\sim_{\text{inc}} \subseteq Q \times Q$ is defined such that $x \sim_{\text{inc}} y$ if,

(i) $Q^\circ \stackrel{\varepsilon}{\Rightarrow} x$ if and only if $Q^\circ \stackrel{\varepsilon}{\Rightarrow} y$;
(ii) for all states $w \in Q$ and all events $\sigma \in \mathbf{A}$ it holds that $w \stackrel{\sigma}{\Rightarrow} x$ if and only if $w \stackrel{\sigma}{\Rightarrow} y$.

Two states are incoming equivalent if they have got the same incoming transitions from the exactly same source states. (This is different from reverse observation equivalence, which accepts *equivalent* rather than identical states.) Incoming equivalence alone is not enough for conflict-preserving abstraction. It is combined with other conditions in the following.

4.2 Enabled Continuation Rule

The Enabled Continuation Rule is a generalisation of the Silent Continuation Rule [9], which allows to merge incoming equivalent states in a τ-loop free automaton provided they have both have an outgoing τ-transition. The reason for this is that, if a state has an outgoing τ-transition, then the other outgoing transitions are "optional" [9] for a test that is to be nonblocking with this automaton. Only continuations from states without further τ-transitions must be present in the test. Using always enabled events, the condition on τ-transitions can be relaxed: it also becomes possible to merge incoming equivalent states if they have outgoing always enabled transitions instead of τ.

Rule 1 (Enabled Continuation Rule). In a τ-loop free automaton, two states that are incoming equivalent and both have an outgoing *always enabled* or τ-transition are conflict equivalent and can be merged.

Example 2. Consider automaton G in Fig. 1 with $\mathbf{E} = \{\eta\}$. States 0 and 1 are both "initial" since they both can be reached silently from the initial state 0. This is enough to satisfy $\sim_{\text{inc}}$ in this case, since neither state is reachable by any event other than τ. Moreover, G has no τ-loops, state 0 has an outgoing τ-transition, and state 1 has an outgoing always enabled event η. Thus, by the Enabled Continuation Rule, states 0 and 1 in G are conflict equivalent and can be merged into state 01 as shown in H.

Note that states 0 and 1 are not observation equivalent because $0 \xrightarrow{\alpha} 2$ while state 1 has no outgoing α-transition. The Silent Continuation Rule [9] also is not applicable because state 1 has no outgoing τ-transition. Only with the additional information that η is always enabled, it becomes possible to merge states 0 and 1.

Proposition 4. Let $G = \langle \mathbf{A}, Q, \rightarrow_G, Q^\circ \rangle$ be a τ-loop free automaton, let $\mathbf{E} \subseteq \mathbf{A}$, and let $\sim \subseteq Q \times Q$ be an equivalence relation such that $\sim \subseteq \sim_{\text{inc}}$, and for all $x, y \in Q$ such that $x \sim y$ it holds that either $x = y$, or $x \stackrel{\eta_1}{\Rightarrow}$ and $y \stackrel{\eta_2}{\Rightarrow}$ for some events $\eta_1, \eta_2 \in \mathbf{E} \cup \{\tau\}$. Then $G \simeq_{\mathbf{E},\emptyset} G/\sim$.

Prop. 4 confirms that the nonblocking property of the system is preserved under the Enabled Continuation Rule, provided that $\mathbf{E}$ is a set of always enabled events for the remainder of the system.

4.3 Only Silent Incoming Rule

The Only Silent Incoming Rule [9] is a combination of observation equivalence and the Silent Continuation Rule. Since the Silent Continuation Rule has been generalised to use always enabled events, the Only Silent Incoming Rule can as well.

The original Only Silent Incoming Rule [9] makes it possible to remove a state with only τ-transitions incoming and merge it into its predecessors, provided that the removed state has got at least one outgoing τ-transition. Again, the requirement for an outgoing τ-transition can be relaxed to allow an always enabled transition also.

Rule 2 (Only Silent Incoming Rule). If a τ-loop free automaton has a state q with only τ-transitions entering it, and an always enabled or τ-transition outgoing from state q, then all transitions outgoing from q can can be copied to originate from the states with τ-transitions to q. Afterwards, the τ-transitions to q can be removed.

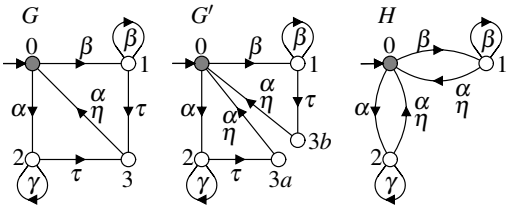

Fig. 2. Only Silent Incoming Rule.

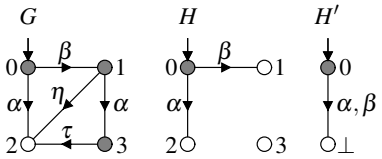

Fig. 3. Limited Certain Conflicts Rule.

Example 3. In Fig. 2 it holds that $G \simeq_{\{\eta\},\emptyset} H$. State 3 in G has only τ-transitions incoming and the always enabled event η outgoing. This state can be removed in two steps. First, state 3 is split into two observation equivalent states $3a$ and $3b$ in G', and afterwards the Silent Continuation Rule is applied to merge these states into their incoming equivalent predecessors, resulting in H. Note that states 1, 2, and 3 are not observation equivalent because of the β- and γ-transitions from states 1 and 2.

Proposition 5. Let $G = \langle \mathbf{A}, Q, \rightarrow_G, Q^\circ \rangle$ be a τ-loop free automaton, and let $\mathbf{E} \subseteq \mathbf{A}$. Let $q \in Q$ such that $q \xrightarrow{\eta}_G$ for some $\eta \in \mathbf{E} \cup \{\tau\}$, and for each transition $x \xrightarrow{\sigma}_G q$ it holds that $\sigma = \tau$. Further, let $H = \langle \mathbf{A}, Q, \rightarrow_H, Q^\circ \rangle$ with

$$\rightarrow_H = \{\, (x,\sigma,y) \mid x \xrightarrow{\sigma}_G y \text{ and } y \neq q \,\} \cup \{\, (x,\sigma,y) \mid x \xrightarrow{\tau}_G q \xrightarrow{\sigma}_G y \,\} \,. \tag{3}$$

Then $G \simeq_{\mathbf{E},\emptyset} H$.

It is shown in [9] that the Only Silent Incoming Rule can be expressed as a combination of observation equivalence and the Silent Continuation Rule as suggested in Example 3. The same argument can be used to prove Prop. 5.

4.4 Limited Certain Conflicts Rule

If an automaton contains blocking states, i.e., states from where no state with an ω-transition can be reached, then a lot of simplification is possible. Once a blocking state is reached, all further transitions are irrelevant. Therefore, all blocking states can be merged into a single state, and all their outgoing transitions can be deleted [16].

In fact, this rule does not only apply to blocking states. For example, consider state 3 in automaton G in Fig. 3. Despite the fact that state 3 is a terminal state, if this state is ever reached, the composed system is necessarily blocking, as nothing can prevent it from executing the silent transition $3 \xrightarrow{\tau} 2$ to the blocking state 2. State 3 is a state of *certain conflicts*, and such states can be treated like blocking states for the purpose of abstraction.

It is possible to calculate all states of certain conflicts, but the algorithm to do this is exponential in the number of states of the automaton to be simplified [16]. To reduce the complexity, the Limited Certain Conflicts Rule [9] approximates the set of certain conflicts. If a state has a τ-transition to a blocking state, then the source state also is a state of certain conflicts. This can be extended to include always enabled events, because if an always enabled transition takes an automaton to a blocking state, then nothing can disable this transition and the composed system is necessarily blocking.

Rule 3 (Limited Certain Conflicts Rule). If an automaton contains an always enabled or τ-transition to a blocking state, then the source state of this transition is a state of certain conflicts, and all its outgoing transitions can be deleted.

Example 4. Consider automaton G in Fig. 3 with $\mathbf{E} = \{\eta\}$. States 1, 2, and 3 are states of certain conflicts. State 2 is already blocking, and states 1 and 3 have a τ- or an always enabled η-transition to the blocking state 2. All outgoing transitions from these states are removed, including the ω-transitions from states 1 and 3. This results in automaton H. Now state 3 is unreachable and can be removed, and states 1 and 2 can be merged using observation equivalence to create H'. It holds that $G \simeq_{\{\eta\},\emptyset} H \simeq_{\text{conf}} H'$.

Proposition 6. Let $G = \langle \mathbf{A}, Q, \rightarrow_G, Q^\circ \rangle$ be an automaton and $\mathbf{E} \subseteq \mathbf{A}$, let $q \in Q$ be a blocking state, and let $p \xrightarrow{\eta} q$ for some $\eta \in \mathbf{E} \cup \{\tau\}$. Furthermore, let $H = \langle \mathbf{A}, Q, \rightarrow_H, Q^\circ \rangle$ where $\rightarrow_H = \{ (x, \sigma, y) \in \rightarrow \mid x \neq p \}$. Then $G \simeq_{\mathbf{E},\emptyset} H$.

Prop. 6 confirms that a state with a τ- or always enabled transitions to some other blocking state can also be made blocking, by deleting all outgoing transitions (including ω) from it. The Limited Certain Conflicts Rule should be applied repeatedly, as the deletion of transitions may introduce new blocking states and thus new certain conflicts.

4.5 Selfloop Removal Rule

The final abstraction rule concerns selfloop-only events. To verify nonblocking, it is enough to check if every state in the final synchronous composition of all automata can reach a terminal state. Selfloops in the final synchronous composition have no effect on the blocking nature of the system, since any path between two states still passes the same states when all selfloops are removed from the path. So the final synchronous composition is nonblocking if and only if it is nonblocking with all selfloops removed.

Based on this observation, if an event is known to be selfloop-only in all automata except the one being simplified, then selfloops with that event can be added or removed freely to the automaton being simplified.

Rule 4 (Selfloop Removal Rule). If an event λ is selfloop-only in all other automata, then selfloop transitions $q \xrightarrow{\lambda} q$ can be added to or removed from any state q.

This rule can be used to remove selfloops and save memory, sometimes reducing the amount of shared events or allowing other rules to be used. If an event only appears on selfloops in all automata, then it can be removed entirely. Furthermore, the addition of selfloops to certain states may also be beneficial.

Example 5. Fig. 4 shows a sequence of conflict-preserving changes to an automaton containing the selfloop-only event λ. First, the λ-selfloop in G_1 is removed to create G_2. In G_2, states 0 and 1 are close to observation equivalent, as they both have a β-transition to state 2; however 0 has a λ-transition to 1 and 1 does not. Yet, it is possible to add a λ-selfloop to state 1 and create G_3. Now states 0 and 1 are observation equivalent and can be merged to create G_4. Finally, the λ-selfloop in G_4 is removed to create G_5.

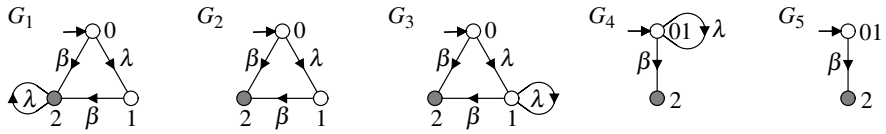

Fig. 4. Removal and addition of selfloops.

Proposition 7. Let $G = \langle \mathbf{A}, Q, \rightarrow_G, Q^\circ \rangle$ and $H = \langle \mathbf{A}, Q, \rightarrow_H, Q^\circ \rangle$ be automata with $\rightarrow_H = \rightarrow_G \cup \{(q, \lambda, q)\}$ for some $\lambda \in \mathbf{A}$. Then $G \simeq_{\emptyset,\{\lambda\}} H$.

Prop. 7 shows that the addition of a single selfloop preserves conflict equivalence. It can be applied in reverse to remove selfloops, and it can be applied repeatedly to add or remove several selfloops in an automaton or in the entire system.

The implementation in Section 6 uses selfloop removal whenever applicable to delete as many selfloops as possible. In addition, observation equivalence has been modified to assume the presence of selfloops for all selfloop-only events in all states, so as to achieve the best possible state-space reduction.

5 Finding Always Enabled and Selfloop-only Events

While the simplification rules in Section 4 are straightforward extensions of known rules for standard conflict equivalence [9], their application requires the knowledge about always enabled and selfloop-only events. Assume the system (2) encountered during compositional verification is

$$G_1 \parallel G_2 \parallel \cdots \parallel G_n \,, \tag{4}$$

and automaton G_1 is to be simplified. Then it is necessary to know always enabled and selfloop-only events in $T = G_2 \parallel \cdots \parallel G_n$. For each component automaton G_i, such events are easy to detect based on Def. 5 and 6. It also is a direct consequence of the definitions that these properties carry over to the synchronous product.

Proposition 8. Let G_1 and G_2 be two automata. If an event σ is always enabled (or selfloop-only) in G_1 and G_2, then σ is always enabled (or selfloop-only) in $G_1 \parallel G_2$.

Given Prop. 8, an event can be considered as always enabled or selfloop-only if it has this property for every automaton in (4) except the automaton being simplified. When checking the individual automata, selfloop-only events are easily found by checking whether an event in question only appears on selfloop transitions. For always enabled events, it is checked whether the event in question is enabled in every state, but additional considerations can help to find more always enabled events.

Example 6. Consider automaton G in Fig. 5. It clearly holds that $0 \stackrel{\eta}{\rightarrow}$, and $1 \stackrel{\tau}{\rightarrow} 0 \stackrel{\eta}{\rightarrow}$ and thus $1 \stackrel{\eta}{\Rightarrow}$. Although η is not enabled in state $\perp$, this state is a blocking state and the set of enabled events for blocking states is irrelevant—it is known [16] that G is conflict equivalent to G'. Then η can be considered as always enabled in G' and thus also in G.

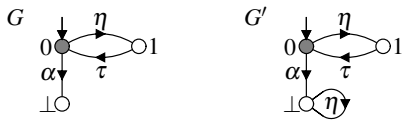

Fig. 5. Finding an always enabled event.

By definition, an always enabled event η must be possible in every state of the environment T, except for blocking states according to Example 6. However, this condition is stronger than necessary, as η typically is not always possible in the automaton G being simplified. This observation leads to *conditionally* always enabled events.

Definition 12. Let $G = \langle \mathbf{A}, Q_G, \rightarrow_G, Q_G^\circ \rangle$ and $T = \langle \mathbf{A}, Q_T, \rightarrow_T, Q_T^\circ \rangle$ be two automata. An event $\sigma \in \mathbf{A}$ is *conditionally always enabled* for G in T, if for all $s \in \mathbf{A}^*$ such that $Q_G^\circ \stackrel{s\sigma}{\Rightarrow}_G$ and all states $x_T \in Q_T$ such that $Q_T^\circ \stackrel{s}{\Rightarrow}_T x_T$, it holds that $x_T \stackrel{\sigma}{\Rightarrow}_T$.

An event is conditionally always enabled if the environment T enables it in all states where it is possible in the automaton G to be simplified. The following Prop. 9 shows that the result of compositional nonblocking verification is also preserved with events that are only conditionally always enabled.

Proposition 9. Let G, H, and T be automata, and let $\mathbf{E}$ and $\mathbf{S}$ be event sets such that $G \simeq_{\mathbf{E},\mathbf{S}} H$, and $\mathbf{E}$ is a set of conditionally always enabled events for G in T, and $\mathbf{S}$ is a set of selfloop-only events for T. Then $G \parallel T$ is nonblocking if and only if $H \parallel T$ is nonblocking.

Conditionally always enabled events can be used like general always enabled events, but they are more difficult to find. To check the condition of Def. 12, it is necessary to explore the state space of $G \parallel T$, which has the same complexity as a nonblocking check. Yet, the condition is similar to *controllability* [6], which can often be verified quickly by an *incremental controllability check* [4]. The incremental algorithm gradually composes some of the automata of the system (4) until it can be ascertained whether or not a given event is conditionally always enabled. In many cases, it gives a positive or negative answer after composing only a few automata.

By running the incremental controllability check for a short time, some conditionally always enabled events can be found, while for others the status remains inconclusive. Fortunately, it is not necessary to find all always enabled events. If the status of an event is not known, it can be assumed that this event is *not* always enabled. The result of nonblocking verification will still be correct, although it may not use the best possible abstractions. It is enough to only consider events as always enabled or selfloop-only, if this property can be established easily.

6 Experimental Results

The compositional nonblocking verification algorithm has been implemented in the discrete event systems tool Waters/Supremica [1], which is freely available for download [26]. The software is further developed from [17] to support always enabled and selfloop-only events.

The new implementation has been applied to all models used for evaluation in [17] with at least $5 \cdot 10^8$ reachable states. The test suite includes complex industrial models and case studies from various application areas such as manufacturing systems, communication protocols, and automotive electronics. The following list gives some details about these models.

aip Model of the automated manufacturing system of the Atelier Inter-établissement de Productique [3]. The tests consider two early versions (**aip0**) based on [14], and a more detailed version (**aip1**) according to [24], which has been modified for a parametrisable number of pallets.

profisafe PROFIsafe field bus protocol model [15]. The task considered here is to verify nonblocking of the communication partners and the network in input-slave configuration with sequence numbers ranging up to 4, 5, and 6.

tbed Model of a toy railroad system [13] in three different designs.

tip3 Model of the interaction between a mobile client and event-based servers of a Tourist Information System [11].

verriegel Car central locking system, originally from the KORSYS project [23].

6link Models of a cluster tool for wafer processing [28].

Compositional verification repeatedly chooses a small set of automata, composes them, applies abstraction rules to the synchronous composition, and replaces the composed automata with the result. This is repeated until the remaining automata are considered too large, or there are only two automata left. The last two automata are not simplified, because it is easier to check the nonblocking property directly by explicitly constructing and exploring the synchronous composition.

A key aspect for a compositional verification algorithm is the way how automata are selected to be composed. The implementation considered here follows a two-step approach [9]. In the first step, some *candidate* sets of automata are formed, and in the second a most promising candidate is selected. For each event σ in the model, a candidate is formed consisting of all automata with σ in their alphabet. Among these candidates, the candidate with the smallest estimated number of states after abstraction is selected. The estimate is obtained by multiplying the product of the state numbers of the automata forming the candidate with the ratio of the numbers of events in the synchronous composition of the candidate after and before removing any local events. This strategy is called **MustL/MinS** [9, 17].

After identification of a candidate, its automata are composed, and then a sequence of abstraction rules is applied to simplify it. First, τ-loops (Def. 10) and observation equivalent redundant transitions [8] are removed from the automaton. This is followed by the Only Silent Incoming Rule (Prop. 5), the Only Silent Outgoing Rule [9], the Limited Certain Conflicts Rule (Prop. 6), Observation Equivalence (Prop. 3), the Non-α Determinisation Rule [17], the Active Events Rule [9], and the Silent Continuation Rule (Prop. 4).

During simplification, all selfloops with selfloop-only events are deleted, and observation equivalence and the removal of observation equivalent redundant transitions exploit selfloop-only events for further simplification. Furthermore, the Only Silent Incoming Rule, the Limited Certain Conflicts Rule, and the Silent Continuation Rule

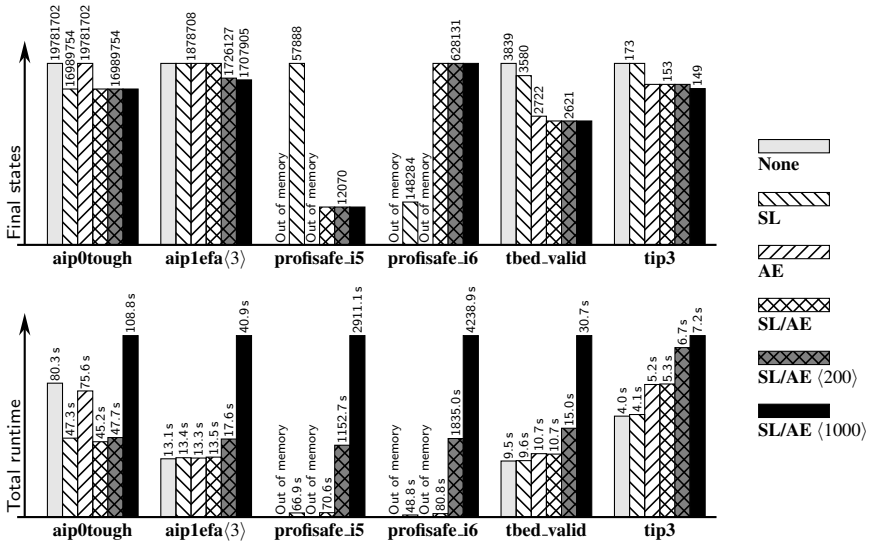

Fig. 6. Final state numbers and runtimes for representative experiments.

take always enabled events into account. For the experiments, the detection of always enabled events and selfloop-only events can be turned on and off separately, producing four strategies **None** (no special events), **SL** (selfloop-only events), **AE** (always enabled events), and **SL/AE** (selfloop-only and always enabled events).

The strategies **AE** and **SL/AE** consider events as always enabled if they are always enabled in every automaton except the one being simplified. Two further strategies **SL/AE** $\langle 200 \rangle$ and **SL/AE** $\langle 1000 \rangle$ also search for events that are conditionally always enabled (Def. 12). This is done using an incremental controllability check [4] that tries to compose an increasing part of the model until it is known whether or not an event is always enabled, or until a state limit of 200 or 1000 states is exceeded; in the latter case, the check is abandoned and the event is assumed to be not always enabled.

The results of the experiments are shown in Table 1 and Fig. 6. The table shows for each model the total number of reachable states in the synchronous composition (Size) if known, and whether or not the model is nonblocking (Res). Then it shows for each strategy, the number of states in the largest automaton encountered during abstraction (Peak States), the number of states in the synchronous composition explored after abstraction (Final States), and the total verification time (Time). The best result in each category is highlighted in bold in the table. Fig. 6 displays the final state numbers and runtimes for six representative experiments graphically.

In some cases, compositional nonblocking verification terminates early, either because all reachable states of all automata are known to be terminal, or because some automaton has no reachable terminal states left. In these cases, the final synchronous composition is not constructed and the final states number is shown as 0 in the table.

All experiments are run on a standard desktop computer using a single core 3.3 GHz CPU and 8 GB of RAM. The experiments are controlled by state limits. If during ab-

Table 1. Experimental results.

Model	Size	Res	None			SL			AE		
			Peak states	Final states	Time [s]	Peak states	Final states	Time [s]	Peak states	Final states	Time [s]
aip0aip	$1.02 \cdot 10^9$	yes	1090	5	1.4	1090	5	**1.4**	1090	5	1.4
aip0tough	$1.02 \cdot 10^{10}$	no	96049	19781702	80.3	96049	**16989754**	47.3	96049	19781702	75.6
aip1efa⟨3⟩	$6.88 \cdot 10^8$	yes	40290	1878708	**13.1**	40290	1878708	13.4	40290	1878708	13.3
aip1efa⟨16⟩	$9.50 \cdot 10^{12}$	no	65520	13799628	22.2	65520	13799628	**22.2**	65520	13799628	22.9
aip1efa⟨24⟩	$1.83 \cdot 10^{13}$	no	6384	13846773	**18.4**	6384	13846773	18.6	6384	13846773	18.7
profisafe_i4		yes				74088	**409**	82.3			
profisafe_i5		yes				98304	57888	**66.9**			
profisafe_i6		yes				55296	**148284**	**48.8**			
tbed_ctct	$3.94 \cdot 10^{13}$	no	43825	0	**14.5**	43825	0	14.8	43825	0	16.6
tbed_hisc	$5.99 \cdot 10^{12}$	yes	1757	**33**	2.7	1757	**33**	**2.7**	**1705**	**33**	2.8
tbed_valid	$3.01 \cdot 10^{12}$	yes	50105	3839	**9.5**	50105	3580	9.6	50105	2722	10.7
tip3	$2.27 \cdot 10^{11}$	yes	**6399**	173	**4.0**	**6399**	173	4.1	12303	153	5.2
tip3_bad	$5.25 \cdot 10^{10}$	no	1176	14	**1.0**	**1032**	14	1.0	1176	**0**	1.1
verriegel3	$9.68 \cdot 10^8$	yes	3303	2	2.0	3303	2	1.7	3349	2	1.8
verriegel3b	$1.32 \cdot 10^9$	no	**1764**	0	**1.2**	**1764**	0	1.3	1795	0	1.2
verriegel4	$4.59 \cdot 10^{10}$	yes	**2609**	2	**1.4**	**2609**	2	1.5	2644	2	1.7
verriegel4b	$6.26 \cdot 10^{10}$	no	**1764**	0	1.4	**1764**	0	1.4	1795	0	**1.4**
6linka	$2.45 \cdot 10^{14}$	no	64	0	**0.4**	64	0	0.4	64	0	0.5
6linki	$2.75 \cdot 10^{14}$	no	61	0	**0.3**	61	0	0.3	61	0	0.3
6linkp	$4.43 \cdot 10^{14}$	no	32	0	0.3	32	0	**0.3**	32	0	0.3
6linkre	$6.21 \cdot 10^{13}$	no	118	12	0.5	118	12	0.5	**106**	**0**	0.5

Model	Size	Res	SL/AE			SL/AE ⟨200⟩			SL/AE ⟨1000⟩		
			Peak states	Final states	Time [s]	Peak states	Final states	Time [s]	Peak states	Final states	Time [s]
aip0aip	$1.02 \cdot 10^9$	yes	1090	5	1.4	**892**	5	24.5	**892**	5	32.0
aip0tough	$1.02 \cdot 10^{10}$	no	96049	**16989754**	**45.2**	96049	**16989754**	47.7	96049	**16989754**	108.8
aip1efa⟨3⟩	$6.88 \cdot 10^8$	yes	40290	1878708	13.5	32980	1726127	17.6	**31960**	**1707905**	40.9
aip1efa⟨16⟩	$9.50 \cdot 10^{12}$	no	65520	13799628	22.9	65520	13799628	28.6	65520	13799628	47.5
aip1efa⟨24⟩	$1.83 \cdot 10^{13}$	no	6384	13846773	19.2	5313	13846773	24.0	**5292**	13846773	41.9
profisafe_i4		yes	**49152**	9864	**61.6**	**49152**	9864	638.4	**49152**	9864	2848.9
profisafe_i5		yes	98304	**12070**	70.6	98304	**12070**	1152.7	98304	**12070**	2911.1
profisafe_i6		yes	**52224**	628131	80.8	**52224**	628131	1835.0	**52224**	628131	4238.9
tbed_ctct	$3.94 \cdot 10^{13}$	no	43825	0	16.4	43825	0	20.9	43825	0	43.7
tbed_hisc	$5.99 \cdot 10^{12}$	yes	**1705**	**33**	3.0	**1705**	**33**	24.5	**1705**	138	81.5
tbed_valid	$3.01 \cdot 10^{12}$	yes	50105	**2621**	10.7	50105	**2621**	15.0	50105	**2621**	30.7
tip3	$2.27 \cdot 10^{11}$	yes	12303	153	5.3	12303	153	6.7	12303	**149**	7.2
tip3_bad	$5.25 \cdot 10^{10}$	no	1096	**0**	1.1	1096	**0**	2.9	1096	**0**	3.8
verriegel3	$9.68 \cdot 10^8$	yes	3349	2	**1.5**	**2644**	2	6.0	**2644**	2	9.6
verriegel3b	$1.32 \cdot 10^9$	no	1795	0	1.3	1795	0	5.8	1795	0	8.3
verriegel4	$4.59 \cdot 10^{10}$	yes	2644	2	1.6	2644	2	8.6	2644	2	17.3
verriegel4b	$6.26 \cdot 10^{10}$	no	1795	0	1.4	1795	0	8.1	1795	0	13.6
6linka	$2.45 \cdot 10^{14}$	no	64	0	0.5	64	0	2.2	64	0	2.7
6linki	$2.75 \cdot 10^{14}$	no	61	0	0.3	61	0	1.7	61	0	2.0
6linkp	$4.43 \cdot 10^{14}$	no	32	0	0.3	32	0	1.6	32	0	2.0
6linkre	$6.21 \cdot 10^{13}$	no	**106**	**0**	**0.5**	**106**	**0**	2.3	**106**	**0**	2.8

straction the synchronous composition of a candidate has more than 100,000 states, it is discarded and another candidate is chosen instead. The state limit for the final synchronous composition after abstraction is 10^8 states. If this limit is exceeded, the run is aborted and the corresponding table entries are left blank.

The experiments show that compositional verification can check the nonblocking property of systems with up to 10^{14} states in a matter of seconds. The exploitation of always enabled and selfloop-only events reduces the peak or final state numbers in many cases. This is important as these numbers are the limiting factors in compositional verification.

The runtimes tend to increase slightly when always enabled or selfloop-only events are used, because the smaller state numbers are outweighed by the effort to find the special events. The search has to be repeated after each abstraction step, because each abstraction can produce new always enabled or selfloop-only events, and the cost increases with the number of steps and events. Conditionally always enabled events can produce better abstractions, but as shown in Fig. 6, it takes a lot of time to find them.

There are also cases where the state numbers increase with always enabled and selfloop-only events. A decrease in the final state number after simplification can come at the expense of increase in the peak state number during simplification. With more powerful simplification algorithms, originally larger automata may fall under the state limits. Also, different abstractions may trigger different candidate selections in following steps, which are not always optimal, and in some cases, the merging of states may prevent observation equivalence from becoming applicable in later steps.

Yet, the large PROFIsafe models [15] can only be verified compositionally with selfloop-only events. By adding always enabled and selfloop-only events to the available tools, it becomes possible to solve problems that are not solvable otherwise.

7 Conclusions

It has been shown how conflict-preserving abstraction can be enhanced by taking into account additional information about the context in which an automaton to be abstracted is used. Specifically, *always enabled* and *selfloop-only* events are easy to discover and help to produce simpler abstractions. Experimental results demonstrate that these special events can make it possible to verify the nonblocking property of more complex discrete event systems. In future work, it is of interest whether the algorithms to detect and use always enabled and selfloop-only events can be improved, and whether other conflict-preserving abstraction methods can also be generalised.

References

1. Åkesson, K., Fabian, M., Flordal, H., Malik, R.: Supremica—an integrated environment for verification, synthesis and simulation of discrete event systems. In: Proc. 8th Int. Workshop on Discrete Event Systems, WODES '06. pp. 384–385. Ann Arbor, MI, USA (Jul 2006)
2. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press (2008)
3. Brandin, B., Charbonnier, F.: The supervisory control of the automated manufacturing system of the AIP. In: Proc. Rensselaer's 4th Int. Conf. Computer Integrated Manufacturing and Automation Technology. pp. 319–324. Troy, NY, USA (1994)
4. Brandin, B.A., Malik, R., Malik, P.: Incremental verification and synthesis of discrete-event systems guided by counter-examples. IEEE Trans. Control Syst. Technol. 12(3), 387–401 (May 2004)
5. Brookes, S.D., Rounds, W.C.: Behavioural equivalence relations induced by programming logics. In: Proc. 16th Int. Colloquium on Automata, Languages, and Programming, ICALP '83. LNCS, vol. 154, pp. 97–108. Springer (1983)
6. Cassandras, C.G., Lafortune, S.: Introduction to Discrete Event Systems. Springer, 2 edn. (2008)

7. De Nicola, R., Hennessy, M.C.B.: Testing equivalences for processes. Theoretical Comput. Sci. 34(1–2), 83–133 (Nov 1984)
8. Eloranta, J.: Minimizing the number of transitions with respect to observation equivalence. BIT 31(4), 397–419 (1991)
9. Flordal, H., Malik, R.: Compositional verification in supervisory control. SIAM J. Control and Optimization 48(3), 1914–1938 (2009)
10. Graf, S., Steffen, B.: Compositional minimization of finite state systems. In: Proc. 1990 Workshop on Computer-Aided Verification. LNCS, vol. 531, pp. 186–196. Springer (Jun 1990)
11. Hinze, A., Malik, P., Malik, R.: Interaction design for a mobile context-aware system using discrete event modelling. In: Proc. 29th Australasian Computer Science Conf., ACSC '06. pp. 257–266. Hobart, Australia (2006)
12. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
13. Leduc, R.J.: PLC Implementation of a DES Supervisor for a Manufacturing Testbed: An Implementation Perspective. Master's thesis, Dept. of Electrical Engineering, University of Toronto, ON, Canada (1996), `http://www.cas.mcmaster.ca/~leduc`
14. Leduc, R.J.: Hierarchical Interface-based Supervisory Control. Ph.D. thesis, Dept. of Electrical Engineering, University of Toronto, ON, Canada (2002), `http://www.cas.mcmaster.ca/~leduc`
15. Malik, R., Mühlfeld, R.: A case study in verification of UML statecharts: the PROFIsafe protocol. J. Universal Computer Science 9(2), 138–151 (Feb 2003)
16. Malik, R.: The language of certain conflicts of a nondeterministic process. Working Paper 05/2010, Dept. of Computer Science, University of Waikato, Hamilton, New Zealand (2010)
17. Malik, R., Leduc, R.: Compositional nonblocking verification using generalised nonblocking abstractions. IEEE Trans. Autom. Control 58(8), 1–13 (Aug 2013)
18. Malik, R., Streader, D., Reeves, S.: Conflicts and fair testing. Int. J. Found. Comput. Sci. 17(4), 797–813 (2006)
19. Milner, R.: Communication and concurrency. Series in Computer Science, Prentice-Hall (1989)
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)
21. Pilbrow, C.: Compositional nonblocking verification with always enabled and selfloop-only events. Working Paper 07/2013, Dept. of Computer Science, University of Waikato, Hamilton, New Zealand (2013), `http://hdl.handle.net/10289/8187`
22. Ramadge, P.J.G., Wonham, W.M.: The control of discrete event systems. Proc. IEEE 77(1), 81–98 (Jan 1989)
23. KORSYS Project: `http://www4.in.tum.de/proj/korsys/`
24. 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), `http://www.cas.mcmaster.ca/~leduc`
25. 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 (Jun 2010)
26. Supremica: `http://www.supremica.org`, the official website for the Supremica project
27. Valmari, A.: Compositionality in state space verification methods. In: Proc. 18th Int. Conf. Application and Theory of Petri Nets. LNCS, vol. 1091, pp. 29–56. Springer, Osaka, Japan (Jun 1996)
28. Yi, J., Ding, S., Zhang, M.T., van der Meulen, P.: Throughput analysis of linear cluster tools. In: Proc. 3rd Int. Conf. Automation Science and Engineering, CASE 2007. pp. 1063–1068. Scottsdale, AZ, USA (Sep 2007)