An Algorithm for Compositional Nonblocking Verification Using Special Events

Colin Pilbrow, Robi Malik

Department of Computer Science, University of Waikato, Hamilton, New Zealand

Abstract

This paper proposes to improve compositional nonblocking verification of discrete event systems through the use of special 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 remainder of the system and check how events are being used. *Always enabled* events, *selfloop-only* events, *failing* events, and *blocked* events are easy to detect and often help with simplification 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.

Keywords: Discrete event systems; finite-state machines; model checking; compositional verification; nonblocking; conflict equivalence.

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 and deadlocks [1, 2]. 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 [3].

Compositional verification [4–6] is an effective alternative that can be used independently of or in combination with symbolic methods. Compositional verification exploits the fact that large systems are typically modelled by several components interacting in synchronous composition. Then *compositional minimisation* or *abstraction* [4] can be used to simplify individual components before computing their synchronous composition, gradually reducing the state space of the system and allowing much larger systems to be verified in the end. The ways how components can be simplified to ensure correct verification results depends on the property being verified [7].

The nonblocking property considered in this paper is logically different from most properties commonly studied for compositional verification, and requires very specific abstraction methods [8]. A suitable theory is laid out in previous work [9], 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 used for compositional nonblocking verification. First, *observer projection* [10] and *weak observation equivalence* [11] have been used to simplify automata. The journal paper [8] introduces conflict equivalence to compositional nonblocking verification and proposes a set of conflict-preserving abstraction rules. The same technique has also been applied to compositional verification of the *generalised nonblocking* property [12], giving rise to an

Email addresses: `cgp5@students.waikato.ac.nz` (Colin Pilbrow), `robi@waikato.ac.nz` (Robi Malik) improved set of abstraction rules. It has also been proposed to replace abstraction rules by more general simplification processes using *annotated automata* [13] or *canonical automata* [14].

All the above methods are based on conflict equivalence, and make no assumptions about the automata *not* being simplified. 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 [9]. While this is easy to understand and implement, more simplification is possible by considering the other system components. To improve the degree of simplification, it has been proposed to take more information about the remainder of the system into account and to consider that certain events are *always enabled* or *selfloop-only* in the remainder of the system [15].

This paper is an extended version of the workshop paper [15]. It contains more detailed results about always enabled and selfloop-only events including formal proofs of correctness, plus the additional special event types of *failing* and *blocked* events. It also includes a description of the algorithm for compositional nonblocking verification with special events, at a level of detail not published before, and more elaborate experimental results.

In the following, Section 2 introduces the background of nondeterministic automata, the nonblocking property, conflict equivalence, and compositional nonblocking verification. Section 3 introduces four types of special events and their key properties for use in compositional nonblocking verification, and Section 4 presents simplification rules that exploit these special events. Then Section 5 describes the compositional nonblocking verification algorithm, and Section 6 presents the experimental results. Finally, Section 7 adds some concluding remarks.

2. Preliminaries

2.1. Events and Languages

Event sequences and languages are a simple means to describe discrete system behaviours [1, 2]. 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, while $\mathbf{A}^+ = \mathbf{A}^* \setminus \{\varepsilon\}$ does not include the empty trace. 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 $X \xrightarrow{s} y$, and $G \xrightarrow{s} x$ means $Q^\circ \xrightarrow{s} x$

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 *accepting* 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 $\to$.

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

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

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 [16]. Shared events (including ω) must be executed by all automata synchronously, while other events (including τ) are executed independently.

2.3. The Nonblocking Property and Conflict Equivalence

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. [9] 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.

The theory of *conflict equivalence* [9] allows compositional reasoning about the nonblocking property. According to process-algebraic testing theory, two automata are considered as equivalent if they both respond in the same way to all tests [17]. 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. [9] Two automata G and H are *conflict equivalent*, written $G \simeq_{\mathrm{conf}} H$, if, for any automaton T, it holds that $G \parallel T$ is nonblocking if and only if $H \parallel T$ is nonblocking.

Two automata are conflict equivalent if they always are both nonblocking (or blocking) when composed with the same test automaton T. This means that conflict equivalent components of a system can be substituted by each other without affecting the nonblocking property of the system. It has been shown [9] that conflict equivalence is the coarsest automata equivalence with this property.

The standard method [1] to verify whether a composed system of automata

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

is nonblocking is to construct the synchronous composition and check whether a terminal state can be reached from every reachable state. This can be done with the help of a strongly connected components algorithm [18] in a worst-case time complexity of $O(|{\rightarrow}|)$, where $|{\rightarrow}|$ is the number of transitions in the synchronous composition (2). However, the size of the synchronous composition grows exponentially in the number n of composed automata, and when measured in the number of components, the problem to check the nonblocking property is NP-complete [19].

To mitigate the complexity, compositional methods [8, 12] avoid building the full synchronous composition immediately. Instead, individual automata G_i are simplified and replaced by smaller conflict equivalent automata $H_i \simeq_{\mathrm{conf}} G_i$. If no simplification is possible, a subsystem of automata $(G_j)_{j \in J}$ is selected and replaced by its synchronous composition, which then is simplified. This approach is justified by the congruence properties [9] of conflict equivalence. For example, if G_1 in (2) is replaced by $H_1 \simeq_{\mathrm{conf}} G_1$, then by considering $T = G_2 \parallel \cdots \parallel G_n$ in Definition 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.

Most approaches for compositional nonblocking verification 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*.

Definition 5. Let $G = \langle \mathbf{A}, Q, \rightarrow, Q^\circ \rangle$ and $\mathbf{Y} \subseteq \mathbf{A}$. The result of *hiding* $\mathbf{Y}$ from G, written $G \setminus \mathbf{Y}$, is the automaton obtained from G by replacing each transition $x \xrightarrow{\upsilon} y$ with $\upsilon \in \mathbf{Y}$ by $x \xrightarrow{\tau} y$, and removing the events in $\mathbf{Y}$ from $\mathbf{A}$.

Hiding is the act of transforming certain events into silent τ events. This is a simple way of abstraction that in general introduces nondeterminism.

Proposition 1. Let $G = \langle \mathbf{A}_G, Q_G, \rightarrow_G, Q_G^\circ \rangle$ and $T = \langle \mathbf{A}_T, Q_T, \rightarrow_T, Q_T^\circ \rangle$ be two automata, and let $\mathbf{Y} \subseteq \mathbf{A}_G$ be a set of events local to G, i. e., $\mathbf{Y} \cap \mathbf{A}_T = \emptyset$. Then $G \parallel T$ is nonblocking if and only if $(G \setminus \mathbf{Y}) \parallel T$ is nonblocking.

Proposition 1 is a foundation of compositional nonblocking verification algorithms [4, 8, 12]. It follows immediately from the definitions of synchronous composition (Definition 2) and the nonblocking property (Definition 3). Based on Proposition 1, if an event appears in only one component G_1 of the system (2), then it is treated as a local event and hidden by replacing all its transitions by silent τ-transitions, resulting in $G_1 \setminus \mathbf{Y}$. The new silent transitions typically make it possible to simplify $G_1 \setminus \mathbf{Y}$ and replace it by a conflict equivalent abstraction. Conflict equivalence uses τ as a placeholder for events not used elsewhere, and in this setting is the coarsest conflict-preserving abstraction [9]. Yet, in practice the remainder $T = G_2 \parallel \cdots \parallel G_n$ is known. The following Section 3 introduces special events that provide additional information about T to inform the simplification of G_1, and afterwards Section 4 shows how this information is used to define conflict-preserving abstraction rules.

3. Conflict Equivalence with Special Events

In addition to hiding local events, it can be examined how the other events are used by the rest of the system. This section presents four different types of events that can help to simplify automata beyond standard conflict equivalence while being easy to detect. Subsection 3.1 introduces *always enabled* events, subsection 3.2 introduces *selfloop-only* events, subsection 3.3 introduces *failing* events, and subsection 3.4 introduces *blocked* events.

3.1. Always Enabled Events

The first type of special events are always enabled events. When simplifying a component G_1 of the system (2), it may be known that some event is enabled in every state of the remainder $T = G_2 \parallel \cdots \parallel G_n$ of the system. Such events are called *always enabled*, and they can be used in similar ways as local events when simplifying G_1.

Definition 6. Let $T = \langle \mathbf{A}, Q, \rightarrow, Q^\circ \rangle$ be an automaton. An event $\eta \in \mathbf{A}$ is *always enabled* in T, if for every state $x \in Q$ it holds that $x \overset{\eta}{\Rightarrow}$. In addition, all events $\eta \notin \mathbf{A}$ are always enabled in T.

An event is always enabled in an automaton if it can be executed from every state—possibly after some silent τ events. Events not in the alphabet are also considered as always enabled, because by Definition 2 the automaton does not synchronise on them and therefore never disables them. Several abstraction methods that exploit silent events to simplify automata can be generalised to exploit always enabled events also. Therefore the following definition modifies conflict equivalence by considering a set $\mathbf{E}$ of events that are always enabled in the rest of the system, i. e., in the test T.

Definition 7. Let G and H be two automata. G and H are *conflict equivalent with respect to always enabled events* $\mathbf{E} \subseteq \mathbf{A}$, written $G \overset{\text{econf}}{\simeq}_{\mathbf{E}} H$, if for every automaton T such that $\mathbf{E}$ is a set of always enabled events in T, it holds that $G \parallel T$ is nonblocking if and only if $H \parallel T$ is nonblocking.

Two automata are conflict equivalent with respect to always enabled events $\mathbf{E}$, if they always are both nonblocking (or blocking) when composed with the same test automaton T that has the events in $\mathbf{E}$ always enabled. This is weaker than standard conflict equivalence (Definition 4), because here the nonblocking property needs to be preserved by fewer tests.

Example 1. Automata G and H in Figure 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.

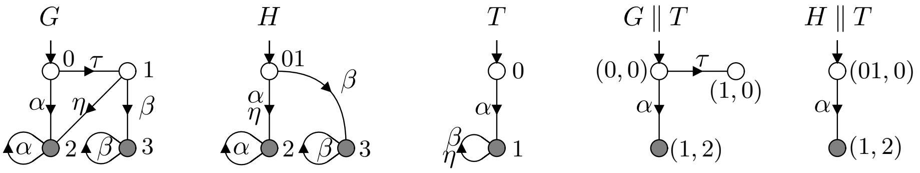

Figure 1: Two automata G and H such that $G \overset{\text{econf}}{\simeq}_{\{\eta\}} H$ but not $G \simeq_{\text{conf}} H$.

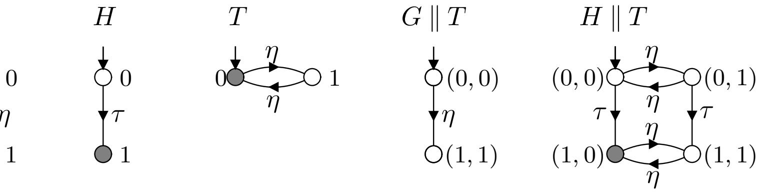

Figure 2: Two automata G and $H = G \setminus \{\eta\}$ such that $G \overset{\text{econf}}{\simeq}_{\{\eta\}} H$ does not hold.

Note that η is not always enabled in T since $0 \overset{\eta}{\Rightarrow}_T$ 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 Proposition 13 below that $G \overset{\text{econf}}{\simeq}_{\{\eta\}} H$.

Example 2. As a counterexample to show that always enabled events cannot be hidden like local events, consider automata G and $H = G \setminus \{\eta\}$ in Figure 2. These automata are not conflict equivalent with respect to the always enabled event η, i. e., $G \overset{\text{econf}}{\simeq}_{\{\eta\}} H$ does *not* hold. To see this, consider the test T in Figure 2. The event η is always enabled in T, but the composition $G \parallel T$ has no reachable accepting state and therefore is blocking, while $H \parallel T$ is nonblocking. This shows that hiding of the always enabled event η in G changes the nonblocking property if the remainder of the system is like T.

Example 2 shows that always enabled events need to be treated differently from local events for compositional nonblocking verification. On the other hand, standard conflict equivalence implies conflict equivalence with respect to always enabled events, as the latter considers fewer tests. Both equivalences have the same useful properties for compositional nonblocking verification. The following results are immediate from the definitions of synchronous composition (Definition 2) and conflict equivalence (Definition 7).

Proposition 2. Let G and H be two automata. Then the following properties hold.

1. $G \simeq_{\text{conf}} H$ if and only if $G \overset{\text{econf}}{\simeq}_{\emptyset} H$.
2. If $\mathbf{E} \subseteq \mathbf{E}'$ then $G \overset{\text{econf}}{\simeq}_{\mathbf{E}} H$ implies $G \overset{\text{econf}}{\simeq}_{\mathbf{E}'} H$.

It follows from Proposition 2 that conflict equivalence with respect to always enabled events 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, Definition 7 ensures that conflict equivalence with respect to always enabled events can be used instead of standard conflict equivalence when simplifying a component G_1 of a system (2), provided that a suitable set $\mathbf{E}$ of always enabled events can be determined.

Fortunately, it is enough to check the components G_i of a system (2) individually to determine what events are always enabled. This follows immediately from the definitions of synchronous composition (Definition 2) and always enabled events (Definition 6).

Proposition 3. Let G_1 and G_2 be two automata. If an event η is always enabled in G_1 and in G_2, then η is always enabled in $G_1 \parallel G_2$.

Proposition 3 confirms that an event is always enabled in a synchronous composition if it is always enabled in every component. Then an event can be considered as always enabled for the purpose of simplifying a component G_1 of a system (2), if it is always enabled in every other system component $G_2, \ldots, G_n$. This sufficient condition can be checked efficiently, as the components can be examined individually, and the complexity is determined by the size of the modular system (2) and not by its synchronous composition. The issue of finding always enabled events is investigated further in Section 4.6 below, where it is shown how always enabled events can be found without the need to explore sequences of τ-transitions as the use of $\Rightarrow$ in Definition 6 suggests.

3.2. Selfloop-Only Events

The second type of special events are *selfloop-only* events. *Selfloops* are transitions that have the same start and end states. If an event is known to appear only on selfloops in the rest of the system, then this fact can also be used to simplify an automaton.

Definition 8. Let $G = \langle \mathbf{A}, Q, \rightarrow, Q^\circ \rangle$ be an automaton. An event $\lambda \in \mathbf{A}$ is *selfloop-only* in G, if for every transition $x \xrightarrow{\lambda} y$ it holds that $x = y$. In addition, all events $\lambda \notin \mathbf{A}$ are selfloop-only in T.

An event is selfloop-only if it only appears on selfloop transitions. The presence of selfloops does not affect the state space of the synchronous composition and thus the nonblocking property, and this observation can help to simplify the system beyond standard conflict equivalence. The following definition modifies conflict equivalence for selfloop-only events in the same way as Definition 7 above for always enabled events.

Definition 9. Let G and H be two automata. G and H are *conflict equivalent with respect to selfloop-only events* $\mathbf{S} \subseteq \mathbf{A}$, written $G \overset{\text{sconf}}{\simeq}_{\mathbf{S}} H$, if for every automaton T such that $\mathbf{S}$ 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.

Conflict equivalence with respect to selfloop-only events has the same useful properties as conflict equivalence with respect to always enabled events. It is a generalisation of standard conflict equivalence and can be used in the same way to simplify individual system components. Furthermore, an event is selfloop-only in a synchronous composition if it is selfloop-only in every component, so these events can be detected by inspecting individual system components only. The following two propositions are analogous to Propositions 2 and 3 for always enabled events, and follow immediately from Definitions 2, 8, and 9.

Proposition 4. Let G and H be two automata. Then the following properties hold.

1. $G \simeq_{\text{conf}} H$ if and only if $G \overset{\text{sconf}}{\simeq}_{\emptyset} H$.
2. If $\mathbf{S} \subseteq \mathbf{S}'$ then $G \overset{\text{sconf}}{\simeq}_{\mathbf{S}} H$ implies $G \overset{\text{sconf}}{\simeq}_{\mathbf{S}'} H$.

Proposition 5. Let G_1 and G_2 be two automata. If an event λ is selfloop-only in G_1 and in G_2, then λ is selfloop-only in $G_1 \parallel G_2$.

3.3. Failing Events

The next kind of special events considered in this paper are *failing* events. These are events known to make the system blocking if enabled.

Definition 10. Let $G = \langle \mathbf{A}, Q, \rightarrow, Q^\circ \rangle$ be an automaton. An event $\varphi \in \mathbf{A}$ is *failing* in G, if for every transition $x \xrightarrow{\varphi} y$ there does not exist $t \in \mathbf{A}^*$ such that $y \overset{t\omega}{\Rightarrow}$.

An event is failing in an automaton, if every transition by this event takes the automaton to a blocking state, i. e., a state from where the termination event ω can never be executed. If only one component G_i of a system (2) is in a blocking state, then it follows from the definition of synchronous composition (Definition 2) that the entire system is in a blocking state. That is, if a failing event ever gets executed, the system ends up in a blocking state, so it is blocking. This observation can be used to simplify an automaton subject to events being failing in another automaton, and justifies the following third variation of conflict equivalence.

Definition 11. Let G and H be two automata. G and H are *conflict equivalent with respect to failing events* $\mathbf{F} \subseteq \mathbf{A}$, written $G \overset{\text{fconf}}{\simeq_{\mathbf{F}}} H$, if for every automaton T such that $\mathbf{F}$ is a set of failing events in T, it holds that $G \| T$ is nonblocking if and only if $H \parallel T$ is nonblocking.

Conflict equivalence with respect to failing events also has the same useful properties as conflict equivalence with respect to always enabled events. It is a generalisation of standard conflict equivalence and can be used in the same way to simplify individual system components.

Proposition 6. Let G and H be two automata. Then the following properties hold.

1. $G \simeq_{\text{conf}} H$ if and only if $G \overset{\text{fconf}}{\simeq_{\emptyset}} H$.
2. If $\mathbf{F} \subseteq \mathbf{F}'$ then $G \overset{\text{fconf}}{\simeq_{\mathbf{F}}} H$ implies $G \overset{\text{fconf}}{\simeq_{\mathbf{F}'}} H$.

The detection of failing events in a composed system (2) is simple and can be achieved by inspecting individual components only. Unlike with always enabled or selfloop-only events, for an event to be failing in a synchronous composition it is enough if the event is failing in only one component. This is an immediate consequence of the definitions of failing events (Definition 10) and synchronous composition (Definition 2).

Proposition 7. Let G_1 and G_2 be two automata. If an event φ is failing in G_1, then φ is failing in $G_1 \parallel G_2$.

Proposition 7 means that an event is failing in a composed system if it is failing in a single component. That is, if an event is found to be failing in some component G_i of a system (2), then it is failing in the entire system and all components can be simplified based on this fact.

3.4. Blocked Events

The last kind of special events considered in this paper are blocked events. Occasionally, a system contains events that are always disabled and therefore can be removed entirely.

Definition 12. Let $G = \langle \mathbf{A}, Q, \rightarrow, Q^\circ \rangle$ be an automaton. An event $\beta \in \mathbf{A}$ is *blocked* in G, if there does not exist any transition $x \xrightarrow{\beta} y$ in G.

An event is blocked in an automaton if it is disabled in all states of that automaton. If an event is blocked in some component of a system, this means that the event is always disabled in the synchronous composition, and its transitions will never be taken. This is a general property of synchronous composition, which is not specific to nonblocking verification.

Definition 13. Let $G = \langle \mathbf{A}, Q, \rightarrow, Q^\circ \rangle$ be an automaton. The *restriction* of G to $\mathbf{A}' \subseteq \mathbf{A}$ is the automaton $G_{|\mathbf{A}'} = \langle \mathbf{A}', Q, \rightarrow_{|\mathbf{A}'}, Q^\circ \rangle$, where $\rightarrow_{|\mathbf{A}'} = \{\, (x, \sigma, y) \in \rightarrow \mid \sigma \in \mathbf{A}' \,\}$.

The restriction of an automaton to an event set $\mathbf{A}'$ is obtained by removing all transitions labelled with events not in $\mathbf{A}'$. The following Proposition 8 shows that the result of synchronous composition is preserved when restriction is used to remove events found to be blocked in an automaton, which implies that this operation preserves the nonblocking property.

Proposition 8. Let $G = \langle \mathbf{A}_G, Q_G, \rightarrow_G, Q_G^\circ \rangle$ and $T = \langle \mathbf{A}_T, Q_T, \rightarrow_T, Q_T^\circ \rangle$ be two automata, and let $\mathbf{B} \subseteq \mathbf{A}_G$ be a set of blocked events in T. Then $G \parallel T = G_{|\mathbf{A}_G \setminus \mathbf{B}} \parallel T$.

Similarly to failing events, the following Proposition 9 shows that it is enough for an event to be blocked in a single component for it to be blocked in a composed system.

Proposition 9. Let G_1 and G_2 be two automata. If an event β is blocked in G_1, then β is blocked in $G_1 \parallel G_2$.

In combination with Proposition 8 it follows that, if an event is found to be blocked in some component G_i of a system (2), then it can be removed from the system without changing its synchronous composition or its nonblocking property. The proofs of Propositions 8 and 9 follow directly from the definitions of blocked events (Definition 12) and synchronous composition (Definition 2).

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. This section presents rules from previous work [8, 12] and shows how they can be improved by taking into account always enabled and selfloop-only events. It also introduces the Selfloop Removal and Failing Events Removal Rules to eliminate transitions with selfloop-only and failing events.

Next, Subsection 4.1 introduces general terminology to describe all abstractions. In the following subsections, each abstraction rule is first described informally and then illustrated by an example. This is followed by a proposition, which provides the formal definition of the abstraction rule and asserts that its application is sound for compositional verification of the nonblocking property.

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 14. 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 $\rightarrow/{\sim} = \{\, ([x], \sigma, [y]) \mid x \xrightarrow{\sigma} y \,\}$ and $\tilde{Q}^\circ = \{\, [x^\circ] \mid x^\circ \in Q^\circ \,\}$.

When constructing a quotient automaton, classes of equivalent states in the original automaton are combined or *merged* into a single state. The quotient automaton contains a transition linking two classes of states if the original automaton contains a transition with the same event that links some states of these classes. A common equivalence relation to construct quotient automata is *observation equivalence* or *weak bisimulation* [20]. The following definition describes a slight variation of observation equivalence, which is known to work for compositional nonblocking verification.

Definition 15. [11] Let $G = \langle \mathbf{A}, Q, \rightarrow, Q^\circ \rangle$ be an automaton. A relation $\approx_{\mathrm{w}} \subseteq Q \times Q$ is a *weak observation equivalence* relation on G if, for all states $x_1, x_2 \in Q$ such that $x_1 \approx_{\mathrm{w}} x_2$ and all traces $s \in \mathbf{A}_\omega^+$ the following conditions hold:

1. if $x_1 \overset{s}{\Rightarrow} y_1$ for some $y_1 \in Q$, then there exists $y_2 \in Q$ such that $y_1 \approx_{\mathrm{w}} y_2$ and $x_2 \overset{s}{\Rightarrow} y_2$;
2. if $x_2 \overset{s}{\Rightarrow} y_2$ for some $y_2 \in Q$, then there exists $y_1 \in Q$ such that $y_1 \approx_{\mathrm{w}} y_2$ and $x_1 \overset{s}{\Rightarrow} y_1$.

Two states are observation equivalent if they have exactly the same sequences of enabled events, leading to equivalent successor states. Weak observation equivalence is very similar to observation equivalence [20], the only difference being that weak observation equivalence only requires equivalent states to be reached by non-empty traces $s \in \mathbf{A}_\omega^+$, while observation equivalence requires this for all traces $s \in \mathbf{A}_\omega^*$. This includes the empty trace ε, so observation equivalence also considers sequences of only silent τ-transitions. Weak observation equivalence is slightly coarser than observation equivalence, yet it implies conflict equivalence [11, 12]. That is, the automaton quotient obtained from a weak observation equivalence relation is conflict equivalent to the original automaton.

Proposition 10. [12] Let G be an automaton, and let $\approx_{\mathrm{w}}$ be a weak observation equivalence relation on G. Then $G \simeq_{\mathrm{conf}} G/{\approx_{\mathrm{w}}}$.

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 Definition 15 that these states are weakly observation equivalent, so by Proposition 10 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 16. 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 weak observation equivalence are easy to compute and produce good abstractions, there are conflict equivalent automata that are not weakly observation equivalent. Several other relations are considered for conflict equivalence [8, 12]. To confirm that an automaton quotient modulo a given equivalence relation is conflict equivalent to the original automaton, it is usually necessary to establish a relationship between the paths in an automaton and its quotient [8]. Firstly, it follows immediately from Definition 14 that every path between two states in an automaton also links the corresponding equivalence classes in its quotient automaton.

Lemma 11. [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. If $x_0 \xrightarrow{\sigma_1} x_1 \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_n} x_n$ is a path in G, then $[x_0] \xrightarrow{\sigma_1} [x_1] \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_n} [x_n]$ is a path in $G/{\sim}$.

Secondly, to establish that conflict equivalence is preserved by an automaton quotient, it is necessary to lift a path in the quotient back to a path in the original automaton. This is not possible with every equivalence relation. It is possible with a weak observation equivalence relation, and another possibility is *incoming equivalence* [8].

Definition 17. [8] 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,

1. $Q^\circ \overset{\varepsilon}{\Rightarrow} x$ if and only if $Q^\circ \overset{\varepsilon}{\Rightarrow} y$;
2. for all states $w \in Q$ and all events $\sigma \in \mathbf{A}_\omega$ it holds that $w \overset{\sigma}{\Rightarrow} x$ if and only if $w \overset{\sigma}{\Rightarrow} y$.

Two states are incoming equivalent if they have got the same incoming transitions from exactly the same source states. Incoming equivalent states typically result from nondeterministic branching. If a state in a nondeterministic automaton has two successor states reached by the same event, then these successor states are incoming equivalent provided that they do not have other incoming transitions. The additional requirement of incoming equivalence suffices to establish a converse of Lemma 11 and makes it possible to lift paths in the quotient automaton back to paths in the original automaton.

Lemma 12. [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 such that $\sim\, \subseteq\, \sim_{\text{inc}}$.

1. If $\tilde{x}_0 \xrightarrow{\sigma_1} \tilde{x}_1 \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_n} \tilde{x}_n$ with $\sigma_i \in \mathbf{A}_{\tau,\omega}$ for $i = 0, \ldots, n$ is a path in $G/{\sim}$, then there exist states $x_i \in \tilde{x}_i$ for $i = 0, \ldots, n$ such that $x_0 \overset{P(\sigma_1)}{\Longrightarrow} x_1 \overset{P(\sigma_2)}{\Longrightarrow} \cdots \overset{P(\sigma_n)}{\Longrightarrow} x_n$ is a path in G.
2. If $G/{\sim} \overset{s}{\Rightarrow} \tilde{x}$ for some $s \in \mathbf{A}_\omega^*$, then there exists $x \in \tilde{x}$ such that $G \overset{s}{\Rightarrow} x$.

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* [15] is a generalisation of the *Silent Continuation Rule* [8], which allows incoming equivalent states in a τ-loop free automaton to be merged provided that they 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” [8] 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). [15] 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 3. [15] Consider automaton G in Figure 1, and assume that η is an always enabled event. States 0 and 1 are both “initial” since they both can be reached silently from the initial state 0. This is enough to satisfy incoming equivalence in this case, since neither state is reachable by any event other than τ. Moreover, G has no τ-loops, state 0

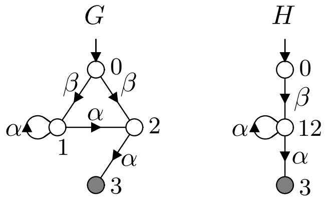

Figure 3: Active Events Rule.

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 weakly observation equivalent because $0 \overset{\alpha}{\to} 2$ while state 1 has no outgoing α-transition. The Silent Continuation Rule [8] 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.

The following Proposition 13 provides the formal definition of the Enabled Continuation Rule and asserts that its application preserves the nonblocking property of a composed system, subject to a set $\mathbf{E}$ of always enabled events in the remainder of the system.

Proposition 13. Let $G = \langle \mathbf{A}, Q, \to_G, Q^\circ \rangle$ 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 $x \overset{\eta_1}{\Rightarrow}$ and $y \overset{\eta_2}{\Rightarrow}$ for some events $\eta_1, \eta_2 \in \mathbf{E} \cup \{\tau\}$. Then $G \overset{\text{econf}}{\simeq}_{\mathbf{E}} G/\!\sim$.

Proof. Let T be an automaton such that $\mathbf{E}$ is always enabled in T.

First assume that $G \parallel T$ is nonblocking. To see that $(G/\!\sim) \parallel T$ is nonblocking, let $(G/\!\sim) \parallel T \overset{s}{\Rightarrow} (\tilde{x}, x_T)$. By Lemma 12, there exists $x \in \tilde{x}$ such that $G \overset{s}{\Rightarrow} x$. Therefore, $G \parallel T \overset{s}{\Rightarrow} (x, x_T)$. Since $G \parallel T$ is nonblocking, there exists $t \in \mathbf{A}^*$ such that $(x, x_T) \overset{t\omega}{\Rightarrow}$. By Lemma 11, this implies $(\tilde{x}, x_T) = ([x], x_T) \overset{t\omega}{\Rightarrow}$, i. e., $(G/\!\sim) \parallel T$ is nonblocking.

Conversely assume that $(G/\!\sim) \parallel T$ is nonblocking. Let $G \parallel T \overset{s}{\Rightarrow} (x, x_T)$. Then, by Lemma 11 it holds that $(G/\!\sim) \parallel T \overset{s}{\Rightarrow} ([x], x_T)$. Consider three cases.

1. $[x] = \{x\}$. Since $(G/\!\sim) \parallel T$ is nonblocking, there exists $t \in \mathbf{A}^*$ such that $([x], x_T) \overset{t\omega}{\Rightarrow}$. By Lemma 12 and since x is the only state in $[x]$, it follows that $(x, x_T) \overset{t\omega}{\Rightarrow}$.
2. $x \overset{\eta}{\to} y$ for some $\eta \in \mathbf{E}$ and $y \in Q$. Then $G \parallel T \overset{s}{\Rightarrow} (x, x_T) \overset{\eta}{\Rightarrow} (y, y_T)$ for some states y of G and y_T of T, because $\eta \in \mathbf{E}$ is always enabled in T. By Lemma 11, it follows that $(G/\!\sim) \parallel T \overset{s\eta}{\Rightarrow} ([y], y_T)$. Since $(G/\!\sim) \parallel T$ is nonblocking, there exists $t \in \mathbf{A}^*$ such that $([y], y_T) \overset{t\omega}{\Rightarrow}$. By Lemma 12, there exists $y' \in [y]$ such that $(y', y_T) \overset{t\omega}{\Rightarrow}$. Since $y' \sim_{\text{inc}} y$ and $x \overset{\eta}{\to} y$, it follows that $x \overset{\eta}{\to} y'$. Therefore, $(x, x_T) \overset{\eta}{\to} (y', y_T) \overset{t\omega}{\Rightarrow}$.
3. $x \overset{\tau}{\to} y$ for some $y \in Q$. Since G is τ-loop free and finite-state, there exists a state $y' \in Q$ such that $x \overset{\varepsilon}{\Rightarrow} y'$ and $y' \overset{\tau}{\to}$ does not hold. If $[y'] = \{y'\}$ then the proof continues as in case 1. Otherwise, since $y' \overset{\tau}{\to}$ does not hold, it follows from the assumptions about $\sim$ that $y' \overset{\eta}{\to}$ for some $\eta \in \mathbf{E}$, and the proof continues as in case 2.

In all three cases, it has been shown that $(x, x_T) \overset{u\omega}{\Rightarrow}$ for some $u \in \mathbf{A}^*$. Therefore $G \parallel T$ is nonblocking. □

The most difficult part of the Enabled Continuation Rule from an implementation perspective is the test for incoming equivalence. To reduce the effort in the implementation, the Enabled Continuation Rule is combined with the *Active Events Rule* [8], which also depends on incoming equivalence.

The idea of the Active Events Rule is to merge states with the same *active* events, where an event α is active in a state x if $x \overset{\alpha}{\Rightarrow}$, i. e., if the event α can be executed from that state—possibly after some sequence of silent τ-transitions. Assume that a nondeterministic choice leads to two different states with exactly the same active events. In order to preserve blocking, only the traces leading to terminal states are important. Therefore, the nondeterministic choice can be "postponed" by one step by merging the two states into a single state.

Rule 2 (Active Events Rule). Two incoming equivalent states with the same active events are conflict equivalent and can be merged.

Example 4. [8] In Figure 3, states 1 and 2 in G have incoming transitions from state 0 associated with β and from state 1 associated with α, which establishes incoming equivalence. Furthermore, they both have α as the only active event. Then the Active Events Rule can be applied to merge states 1 and 2 into a single state 12 as shown in H.

Proposition 14. [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 such that $\sim\, \subseteq\, \sim_{\mathrm{inc}}$, and for all events $\sigma \in \mathbf{A}$ and all states $x, y \in Q$ such that $x \sim y$ and it holds that $x \overset{\sigma}{\Rightarrow}$ if and only if $y \overset{\sigma}{\Rightarrow}$. Then $G \simeq_{\mathrm{conf}} G/{\sim}$.

Given Propositions 13 and 14, two incoming equivalent states can be merged if they either both have outgoing transitions with the silent event τ or with an always enabled event, or if both states have exactly the same active events. Because of this similarity, the Enabled Continuation and Active Events Rules are applied by a single algorithm after computing the incoming equivalence relation using a partition refinement algorithm similar to [21]. Once it is known what states are incoming equivalent, the abstraction algorithm examines each equivalence class. Within each class of incoming equivalent states, it first merges states with equal active event sets, then it merges all states that have an outgoing always enabled or τ-transition, and afterwards it merges states with equal active event sets a second time.

This is done because the merging of states with outgoing always enabled or τ-transitions can change the active event sets and make the Active Events Rule applicable or inapplicable, while merging states with equal active event sets preserves the presence of always enabled or τ-transitions and does not affect the Enabled Continuation Rule. By applying the Active Events Rule before and after the Enabled Continuation Rule, the number of states merged by the Active Events Rule can be increased without affecting the Enabled Continuation Rule.

The merging of states may affect incoming equivalence and make more states incoming equivalent. Therefore, if the above process has successfully merged states, a new incoming equivalence partition is computed, but partitioning is restricted to the successors of states that have been merged. Among the resulting equivalence classes, it is then attempted again to merge states, and the process continues until no further states can be merged. The number of these iterations is bounded by the number of states of the automaton, as each iteration except the last removes at least one state.

Given an automaton with $|\mathbf{A}|$ events, $|Q|$ states, and $|{\rightarrow}|$ transitions, this algorithm can perform up to $|Q|$ iterations. Each iteration requires the evaluation of incoming equivalence and active events for up to $|Q|$ states, each time exploring up to $|{\rightarrow}|$ transitions. This gives a worst-case time complexity of $O(|Q|^2|{\rightarrow}|) = O(|Q|^4|\mathbf{A}|)$ for the combined application of the Enabled Continuation and Active Events Rules to an automaton.

4.3. Only Silent Incoming Rule

The *Only Silent Incoming Rule* [8] is a combination of weak 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 be generalised as well. The original Only Silent Incoming Rule [8] 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 3 (Only Silent Incoming Rule). [15] 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 $q \xrightarrow{\sigma} y$ outgoing from state q can be replaced with new transitions $x \xrightarrow{\sigma} y$ outgoing from all the τ-predecessors x of q, which have the same event σ and target state y as the original transition. Afterwards, the τ-transitions to q can be removed.

Example 5. [15] In Figure 4 it holds that $G \overset{\mathrm{econf}}{\simeq}_{\{\eta\}} H$. 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 weakly observation equivalent states $3a$ and $3b$ in I, 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 weakly observation equivalent because of the β- and γ-transitions from states 1 and 2.

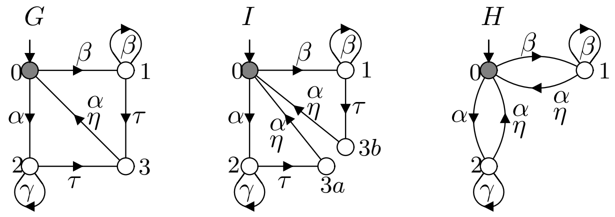

Figure 4: Only Silent Incoming Rule.

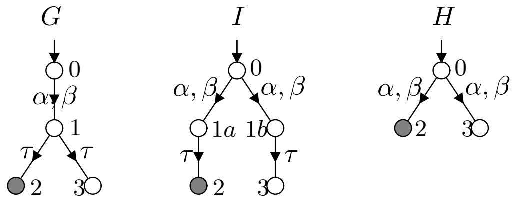

Figure 5: Only Silent Outgoing Rule.

Proposition 15. [15] 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 $G \overset{\text{econf}}{\simeq}_{\mathbf{E}} H$.

The Only Silent Incoming Rule deletes the τ-transitions to the state q and adds the transitions to the successor state of q to all the predecessor states of q. If the state q is not an initial state, it becomes unreachable after deleting the transitions, so it can be removed afterwards. It has been shown [8] that the Only Silent Incoming Rule can be expressed as a combination of weak observation equivalence and the Silent Continuation Rule as suggested in Example 5. The same argument can be used to prove Proposition 15.

The Only Silent Incoming Rule can be implemented efficiently after inspecting all the states and checking the preconditions of Proposition 15. The worst-case time complexity to check and apply this rule to all states of an automaton is $O(|Q|^3|\mathbf{A}|)$.

4.4. Only Silent Outgoing Rule

Another combination of the Silent Continuation Rule with weak observation equivalence gives rise to the *Only Silent Outgoing Rule* [8]. This abstraction rule is quoted here for the sake of completeness. There is no obvious way to improve it using special events.

Rule 4 (Only Silent Outgoing Rule). If a τ-loop free automaton has a state q for which all outgoing transitions are silent τ-transitions, then the state q can be removed after redirecting all its incoming transitions to all its τ-successor states.

This rule is best understood by splitting it into two steps. First the Silent Continuation Rule is applied backwards on state q, splitting q over its outgoing transitions, such that each copy retains exactly one of the outgoing τ-transitions, and all of the incoming transitions. Then the resulting states and their respective successors are weakly observation equivalent.

Example 6. [8] Automaton G in Figure 5 is conflict equivalent to H. State 1 in G has only τ-transitions outgoing, so it is removed after redirecting its incoming transitions $0 \xrightarrow{\alpha} 1$ and $0 \xrightarrow{\beta} 1$ to both its successor states 2 and 3. This can be seen more clearly if the automaton I is considered as an intermediate step. States $1a$ and $1b$ in I are incoming equivalent and both have outgoing τ-transitions, so they can be merged by the Silent Continuation Rule to

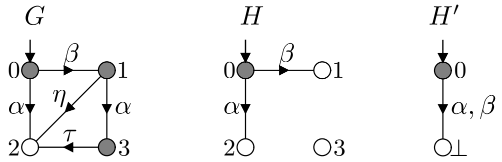

Figure 6: Limited Certain Conflicts Rule.

obtain $G \simeq_{\mathrm{conf}} I$. Furthermore states $1a$ and 2, and states $1b$ and 3 in I are weakly observation equivalent, so they can be merged to obtain $H \approx_{\mathrm{w}} I$. It follows that $G \simeq_{\mathrm{conf}} I \approx_{\mathrm{w}} H$ and thus $G \simeq_{\mathrm{conf}} H$ by Propositions 13 and 10.

Proposition 16. [8] Let $G = \langle \mathbf{A}, Q, \rightarrow_G, Q^\circ_G \rangle$ be a τ-loop free automaton, and let $q \in Q$ such that q has only the outgoing transitions $q \xrightarrow{\tau}_G r_j$ for $j = 1, \ldots, n$. Furthermore, let $H = \langle \mathbf{A}, Q \setminus \{q\}, \rightarrow_H, Q^\circ_H \rangle$ with

$$Q^\circ_H = \begin{cases} Q^\circ_G, & \text{if } q \notin Q^\circ_G; \\ (Q^\circ_G \setminus \{q\}) \cup \{r_1, \ldots, r_n\}, & \text{if } q \in Q^\circ_G; \end{cases} \tag{4}$$

$$\rightarrow_H = \{\, (x, \sigma, y) \mid x \xrightarrow{\sigma}_G y \text{ and } x \neq q \text{ and } y \neq q \,\} \cup \{\, (x, \sigma, r_j) \mid 1 \leq j \leq n \text{ and } x \xrightarrow{\sigma}_G q \,\} \,. \tag{5}$$

Then $G \simeq_{\mathrm{conf}} H$.

The Only Silent Outgoing Rule deletes all transitions attached to state q and replaces them by direct links from the predecessors of q to its τ-successors (5). In addition, if q is an initial state, then its τ-successors become initial states (4). Like in the case of the Only Silent Incoming Rule, checking and applying the Only Silent Outgoing Rule to a state in a τ-loop free automaton involves analysing all incoming and outgoing transitions of that state. The worst-case time complexity is $O(|Q|^3|\mathbf{A}|)$.

4.5. Limited Certain Conflicts Rule

The *Limited Certain Conflicts Rule* is concerned about *blocking* states, i. e., states from where no terminal state can be reached. The presence of such states typically allows for a lot of simplification. 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 [22].

In fact, this does not only apply to blocking states. For example, consider state 3 in automaton G in Figure 6. 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 [22]. To reduce the complexity, the set of certain conflicts can be approximated [8]. 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 5 (Limited Certain Conflicts Rule). [15] 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 7. [15] Consider automaton G in Figure 6, and assume η is an always enabled event. 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 weak observation equivalence to create H'. It holds that $G \overset{\mathrm{econf}}{\simeq}_{\{\eta\}} H \simeq_{\mathrm{conf}} H'$.

Proposition 17. 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 $G \overset{\text{econf}}{\simeq_{\mathbf{E}}} H$.

Proof. Let T be an automaton such that $\mathbf{E}$ is always enabled in T.

First assume that $G \parallel T$ is nonblocking. To see that $H \parallel T$ is nonblocking, let $H \parallel T \xrightarrow{s} (x, x_T)$. Clearly $\rightarrow_H \subseteq \rightarrow_G$, so $G \parallel T \xrightarrow{s} (x, x_T)$. Since $G \parallel T$ is nonblocking, there exists $t \in \mathbf{A}_\tau^*$ such that $(x, x_T) \xrightarrow{t\omega}$ in $G \parallel T$. Then let $t = \sigma_1 \cdots \sigma_n$ and write

$$(x, x_T) = (x^0, x_T^0) \xrightarrow{\sigma_1}_{G \parallel T} (x^1, x_T^1) \xrightarrow{\sigma_2}_{G \parallel T} \cdots \xrightarrow{\sigma_n}_{G \parallel T} (x^n, x_T^n) \xrightarrow{\omega}_{G \parallel T} \ . \tag{6}$$

Let $(x^i, x_T^i) \xrightarrow{\sigma_{i+1}}_{G \parallel T} (x^{i+1}, x_T^{i+1})$ be an arbitrary transition on the path (6). If $x^i \xrightarrow{\sigma_{i+1}}_G x^{i+1}$ does not hold, then $x_T^i \xrightarrow{\sigma_{i+1}}_T x_T^{i+1}$ is a transition in T and $x^i = x^{i+1}$. It follows that $(x^i, x_T^i) \xrightarrow{\sigma_{i+1}}_{H \parallel T} (x^i, x_T^{i+1}) = (x^{i+1}, x_T^{i+1})$.

Otherwise, if $x^i \xrightarrow{\sigma_{i+1}}_G x^{i+1}$, then assume for the sake of proof by contradiction, that this transition does not exist in H. This means that $x^i = p$. Consider the two cases for $\eta \in \mathbf{E} \cup \{\tau\}$. If $\eta \in \mathbf{E}$, then $x_T^i \xrightarrow{\eta}_T y_T$ for some state y_T of T as $\mathbf{E}$ is always enabled in T, and thus $(x^i, x_T^i) = (p, x_T^i) \xrightarrow{\eta}_{G \parallel T} (q, y_T)$. If $\eta = \tau$, then $(x^i, x_T^i) = (p, x_T^i) \xrightarrow{\tau}_{G \parallel T} (q, x_T^i)$. In both cases, it follows that $(x, x_T) \xrightarrow{\sigma_1 \cdots \sigma_i}_{G \parallel T} (x^i, x_T^i) = (p, x_T^i) \xrightarrow{\eta}_{G \parallel T} (q, y_T)$ for some state y_T of T. However, (q, y_T) is a blocking state in $G \parallel T$ because q is a blocking state in G. Then $G \parallel T$ is blocking in contradiction to the assumption. It follows that the transition $x^i \xrightarrow{\sigma_{i+1}}_G x^{i+1}$ was not removed and is still present in H. Again it holds that $(x^i, x_T^i) \xrightarrow{\sigma_{i+1}}_{H \parallel T} (x^{i+1}, x_T^{i+1})$.

Thus, the path (6) exists in $H \parallel T$, i. e., $H \parallel T$ is nonblocking.

Conversely, assume that $H \parallel T$ is nonblocking. To see that $G \parallel T$ is nonblocking, let $G \parallel T \xrightarrow{s} (x, x_T)$. It is to be shown that $(x, x_T) \xrightarrow{t\omega}$ for some $t \in \mathbf{A}^*$. Let $s = \sigma_1 \cdots \sigma_n$ and write

$$(x^0, x_T^0) \xrightarrow{\sigma_1}_{G \parallel T} (x^1, x_T^1) \xrightarrow{\sigma_2}_{G \parallel T} \cdots \xrightarrow{\sigma_n}_{G \parallel T} (x^n, x_T^n) = (x, x_T) \tag{7}$$

where x^0 and x_T^0 are initial states of G and T, respectively. It is shown by induction that $(x^0, x_T^0) \xrightarrow{\sigma_1 \cdots \sigma_k}_{H \parallel T} (x^k, x_T^k)$ for $k = 0, \ldots, n$. This is trivial for $k = 0$. Now assume $(x^0, x_T^0) \xrightarrow{\sigma_1 \cdots \sigma_k}_{H \parallel T} (x^k, x_T^k)$ for some $k < n$. If $x^k \xrightarrow{\sigma_{k+1}}_G x^{k+1}$ does not hold, then clearly $x^k = x^{k+1}$ and thus $(x^0, x_T^0) \xrightarrow{\sigma_1 \cdots \sigma_k}_{H \parallel T} (x^k, x_T^k) \xrightarrow{\sigma_{k+1}}_{H \parallel T} (x^k, x_T^{k+1}) = (x^{k+1}, x_T^{k+1})$. Otherwise, if $x^k \xrightarrow{\sigma_{k+1}}_G x^{k+1}$, assume that this transition does not exist in H. This means that $x^k = p$, and thus by inductive assumption $(x^0, x_T^0) \xrightarrow{\sigma_1 \cdots \sigma_k}_{H \parallel T} (p, x_T^k)$, where p is a deadlock state in H (with no outgoing transitions by construction, and thus ω never possible). Then $H \parallel T$ is blocking in contradiction to the assumption. It follows that the transition $x^k \xrightarrow{\sigma_{k+1}}_G x^{k+1}$ was not removed and is still present in H. By inductive assumption, $(x^0, x_T^0) \xrightarrow{\sigma_1 \cdots \sigma_k}_{H \parallel T} (x^k, x_T^k) \xrightarrow{\sigma_{k+1}}_{H \parallel T} (x^{k+1}, x_T^{k+1})$. Since furthermore G and H have the same initial states, it follows from the induction that $H \parallel T \xrightarrow{s} (x, x_T)$.

Since $H \parallel T$ is nonblocking, it follows that $(x, x_T) \xrightarrow{t\omega}_{H \parallel T}$ for some $t \in \mathbf{A}_\tau^*$. Since $\rightarrow_H \subseteq \rightarrow_G$, this implies $(x, x_T) \xrightarrow{t\omega}_{G \parallel T}$. It follows that $G \parallel T$ is nonblocking. □

Proposition 17 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 is implemented by checking transitions, but it requires a search to determine the set of blocking states first, which runs in $O(|{\rightarrow}|)$ time. This check should to be repeated when transitions are removed, as this may introduce new blocking states. As there could be up to $|Q|$ iterations, the worst-case time complexity to apply the Limited Certain Conflicts Rule in all possible ways to an automaton is $O(|Q||{\rightarrow}|) = O(|Q|^3|\mathbf{A}|)$.

4.6. Finding Always Enabled Events

While selfloop-only, failing, and blocked events in an automaton can be detected easily by inspecting the transitions, always enabled events are more difficult to find and deserve special attention. According to Definition 6, an

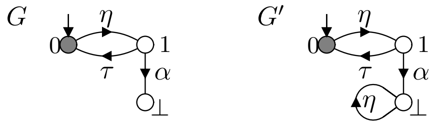

Figure 7: Finding an always enabled event.

event η is always enabled in an automaton if $x \overset{\eta}{\Rightarrow}$ holds for every state x, i. e., if every state can execute a sequence of silent τ-transitions followed by η. This condition is best checked after computing the transitive closure of τ-transitions, but the algorithm to do this has cubic time complexity in the number of states of the automaton [23]. This complexity can be avoided by the noting that the automata resulting from abstraction in compositional nonblocking verification are typically τ-loop free, and in this case the transitive closure computation can be avoided thanks to the following result.

Proposition 18. Let $G = \langle \mathbf{A}, Q, \rightarrow, Q^\circ \rangle$ be a τ-loop free automaton. An event $\eta \in \mathbf{A}$ is always enabled in G if and only if it holds for every state $x \in Q$ that $x \xrightarrow{\eta}$ or $x \xrightarrow{\tau}$.

Proof. If η is always enabled in G, then it holds by Definition 6 that $x \overset{\eta}{\Rightarrow}$ for every state $x \in Q$, which implies $x \xrightarrow{\eta}$ or $x \xrightarrow{\tau}$. For the converse, assume that $x \xrightarrow{\eta}$ or $x \xrightarrow{\tau}$ for every state $x \in Q$, and let $x \in Q$ be an arbitrary state. As G is finite-state and τ-loop free, there exists a state $y \in Q$ such that $x \xrightarrow{\tau^*} y$ and $y \xrightarrow{\tau}$ does not hold. Then it follows from the assumption that $y \xrightarrow{\eta}$, and therefore $x \xrightarrow{\tau^*} y \xrightarrow{\eta}$, i. e., $x \overset{\eta}{\Rightarrow}$. □

According to Proposition 18, in a τ-loop free automaton, it is enough to check the states without outgoing τ-transitions for always enabled events. This is because, if a state in a τ-loop free automaton has an outgoing τ-transition, then it must be possible to reach a state without outgoing τ-transitions, where every always enabled event must be enabled. This result makes it possible to determine the set of always enabled events by checking the outgoing transitions of each state once, which can be achieved in linear time in the number of transitions.

In addition to the above result, a further consideration is important to increase the amount of always enabled events. Automata encountered in compositional nonblocking verification may contain blocking states without any outgoing transitions, and such automata by definition do not have any always enabled events. This view is unnecessarily restrictive.

Example 8. [15] Consider automaton G in Figure 7. It neither holds that $\bot \xrightarrow{\eta}$ nor $\bot \xrightarrow{\tau}$, so η is not always enabled in G. However, $\bot$ is a blocking state and the set of enabled events for blocking states is irrelevant—it is known [22] that G is conflict equivalent to G'. Noting that $0 \xrightarrow{\eta}$, $1 \xrightarrow{\tau}$, and $\bot \xrightarrow{\eta}$ in G' it follows by Proposition 18 that η is always enabled in G'. Since $G \simeq_{\mathrm{conf}} G'$, the event η can also be considered as always enabled in G.

Based on Proposition 18 and Example 8, an event η can be considered as always enabled in a τ-loop free automaton G, if η is enabled in every state of G that has no outgoing τ-transition and at least one outgoing transition.

The requirement for always enabled events can be further relaxed by considering *conditionally* always enabled events [15], where the remaining automata of the system are examined more closely using an incremental controllability check [24]. Then an event η is considered as always enabled when simplifying an automaton G, if the environment T enables η in all states (x_G, x_T) of $G \,\|\, T$ where η is enabled in the state x_G of G. This approach allows for more events to be considered as always enabled, but repeatedly checking such conditions of all the automata not being abstracted is time-consuming [15].

4.7. Selfloop Removal Rule

The next abstraction rule concerns selfloop-only events. To verify nonblocking, it is enough to check if every state in the synchronous composition of all automata in the system can reach a terminal state. Selfloops in the 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 synchronous composition is nonblocking if and only if it is nonblocking with all selfloops removed.

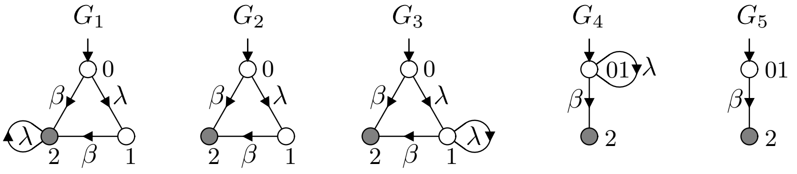

Figure 8: Removal and addition of selfloops.

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 freely be added to or removed from the automaton being simplified.

Rule 6 (Selfloop Removal Rule). [15] 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. The following example shows that the addition of selfloops to certain states may also be beneficial.

Example 9. [15] Figure 8 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 almost weakly 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, as λ is a selfloop-only event, a λ-selfloop can be added to state 1 to create G_3. Now states 0 and 1 are weakly observation equivalent and can be merged to create G_4. Finally, the λ-selfloop in G_4 is removed to create G_5. Overall it holds that

$$G_1 \overset{\text{sconf}}{\simeq}_{\{\lambda\}} G_2 \overset{\text{sconf}}{\simeq}_{\{\lambda\}} G_3 \approx_{\text{w}} G_4 \overset{\text{sconf}}{\simeq}_{\{\lambda\}} G_5 \,. \tag{8}$$

Proposition 20 below confirms that the Selfloop Removal Rule preserves conflict equivalence when selfloop-only events are considered. The proof uses a lemma to show that selfloop removal does not affect the existence of paths.

Lemma 19. 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 $q \in Q$ and $\lambda \in \mathbf{A}$. Furthermore, let T be an automaton such that λ is selfloop-only for T. For all paths $(x, x_T) \rightarrow_{H\|T} (y, y_T)$ it also holds that $(x, x_T) \rightarrow_{G\|T} (y, y_T)$.

Proof. Assume $(x, x_T) = (x^0, x_T^0) \xrightarrow{\sigma_1}_{H\|T} (x^1, x_T^1) \xrightarrow{\sigma_2}_{H\|T} \cdots \xrightarrow{\sigma_n}_{H\|T} (x^n, x_T^n) = (y, y_T)$. The claim is shown by induction on n. For $n = 0$, this is clear as $(x, x_T) = (y, y_T)$. Now consider a path $(x, x_T) = (x^0, x_T^0) \xrightarrow{\sigma_1}_{H\|T} \cdots \xrightarrow{\sigma_n}_{H\|T} (x^n, x_T^n) \xrightarrow{\sigma_{n+1}}_{H\|T} (x^{n+1}, x_T^{n+1}) = (y, y_T)$, where $(x^0, x_T^0) \rightarrow_{G\|T} (x^n, x_T^n)$ by inductive assumption. For the path's final transition $(x^n, x_T^n) \xrightarrow{\sigma_{n+1}}_{H\|T} (x^{n+1}, x_T^{n+1})$, consider three cases.

If $\sigma_{n+1} \notin \mathbf{A}$, then $x_T^n \xrightarrow{\sigma_{n+1}}_H x_T^{n+1}$ is a transition in T and $x^n = x^{n+1}$. By inductive assumption, $(x, x_T) = (x^0, x_T^0) \rightarrow_{G\|T} (x^n, x_T^n) \xrightarrow{\sigma_{n+1}}_{G\|T} (x^n, x_T^{n+1}) = (x^{n+1}, x_T^{n+1})$.

If $x^n \xrightarrow{\sigma_{n+1}}_H x^{n+1}$ is the selfloop $q \xrightarrow{\lambda}_H q$, then $x^{n+1} = q = x^n$ and $x_T^{n+1} = x_T^n$ because $\sigma_{n+1} = \lambda$ is selfloop-only for T. By inductive assumption, it follows that $(x, x_T) = (x^0, x_T^0) \rightarrow_{G\|T} (x^n, x_T^n) = (x^{n+1}, x_T^{n+1})$.

Otherwise, if $x^n \xrightarrow{\sigma_{n+1}}_H x^{n+1}$ is not the selfloop $q \xrightarrow{\lambda}_H q$, then $x^n \xrightarrow{\sigma_{n+1}}_G x^{n+1}$ is a transition in G. Again by inductive assumption, it follows that $(x, x_T) = (x^0, x_T^0) \rightarrow_{G\|T} (x^n, x_T^n) \xrightarrow{\sigma_{n+1}}_{G\|T} (x^{n+1}, x_T^{n+1})$. □

Proposition 20. 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 $q \in Q$ and $\lambda \in \mathbf{A}$. Then $G \overset{\text{sconf}}{\simeq}_{\{\lambda\}} H$.

Proof. Let T be an automaton such that λ is selfloop-only in T.

First assume that $G \parallel T$ is nonblocking. To see that $H \parallel T$ is nonblocking, let $H \parallel T \rightarrow (x, x_T)$. By Lemma 19, it holds that $G \parallel T \rightarrow (x, x_T)$. Since $G \parallel T$ is nonblocking, there exists $t \in \mathbf{A}_\tau^*$ such that $(x, x_T) \xrightarrow{t\omega}_{G\|T}$. Since $\rightarrow_G \subseteq \rightarrow_H$, it follows that $(x, x_T) \xrightarrow{t\omega}_{H\|T}$, i. e., $H \parallel T$ is nonblocking.

Conversely, assume that $H\|T$ is nonblocking. To see that $G\|T$ is nonblocking, let $G\|T \to (x, x_T)$. Since $\to_G \subseteq \to_H$, it holds that $H \parallel T \xrightarrow{s} (x, x_T)$. Because $H \parallel T$ is nonblocking, there exists $t \in \mathbf{A}_\tau^*$ such that $(x, x_T) \xrightarrow{t}_{H\|T} (y, y_T) \xrightarrow{\omega}_{H\|T}$. Using Lemma 19, it follows that $(x, x_T) \to_{G\|T} (y, y_T)$. Furthermore, it follows from $y \xrightarrow{\omega}_H$ that $y \xrightarrow{\omega}_G$ because $\lambda \neq \omega$ and $\to_G$ and $\to_H$ only differ in a λ-transition. Thus, $(x, x_T) \to_{G\|T} (y, y_T) \xrightarrow{\omega}_{G\|T}$, i. e., $G \parallel T$ is nonblocking. □

Proposition 20 shows that the addition of a single selfloop preserves conflict equivalence with respect to selfloop-only events. 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 removal of selfloops is straightforward to implement and always simplifies an automaton, and it can make other rules such as the Only Silent Incoming Rule and the Only Silent Outgoing Rule applicable. On the other hand, Example 9 suggests that the addition of selfloops may also be beneficial, if it is done in a controlled way in combination with specific rules. In the following, three abstraction rules are presented that can benefit from the addition of selfloops in a way that is easy to implement. These rules are related to *observation equivalence* [20], which has several applications in compositional nonblocking verification.

Firstly, observation equivalence is used to remove redundant transitions [25]. A transition $x \xrightarrow{\sigma} y$ is redundant by observation equivalence if the automaton contains another path connecting x and y by event σ, for example $x \xrightarrow{\tau} x' \xrightarrow{\sigma} y' \xrightarrow{\tau} y$. Then the removal of the transition $x \xrightarrow{\sigma} y$ results in a weakly observation equivalent automaton. The following result is immediate from the definition.

Proposition 21. Let $G = \langle \mathbf{A}, Q, \to, Q^\circ \rangle$ be an automaton such that $x \overset{\sigma}{\Rightarrow} y$ for some $\sigma \in \mathbf{A}$, and let $H = \langle \mathbf{A}, Q, \to \cup \{(x, \sigma, y)\}, Q^\circ \rangle$. Then $G \approx_\mathrm{w} H$.

Rule 7 (Observation Equivalent Transition Removal Rule). A transition $x \xrightarrow{\sigma} y$ can be removed if the relation $x \overset{\sigma}{\Rightarrow} y$ still holds after removal of the transition.

If an event λ is known to be selfloop-only, this allows more transitions to be removed. For example, a transition $x \xrightarrow{\lambda} y$ can be removed if $x \xrightarrow{\tau} y$, because then after the addition of the selfloop $y \xrightarrow{\lambda} y$, it holds that $x \xrightarrow{\tau} y \xrightarrow{\lambda} y$ and thus $x \overset{\lambda}{\Rightarrow} y$. Based on this observation, all transitions by a selfloop-only event that are parallel to a τ-transition can be removed.

A maximal set of observation equivalent redundant transitions can be removed by a cubic-complexity algorithm [25]. This algorithm can be modified to take selfloop-only events into account. As the addition of selfloops only increases the amount of redundant transitions, it is safe to add selfloops with the selfloop-only events to all states before starting the algorithm, and remove these selfloops afterwards. Alternatively, the algorithm can be modified to assume the presence of selfloops as it executes. Neither of these changes affects the worst-case time complexity of the algorithm, which is dominated by transitive closure computations and runs in $O(|Q|^3|\mathbf{A}|)$ time [25].

The most important use of weak observation equivalence is to reduce the state space of an automaton by computing a weak observation equivalence relation and replacing the automaton by the associated automaton quotient.

Rule 8 (Weak Observation Equivalence Rule). [12] If $\sim$ is a weak observation equivalence relation on an automaton G, then this automaton can be replaced by its quotient $G/{\sim}$.

This rule is justified by Proposition 10. It is implemented by a partitioning algorithm [21], which can be modified to take selfloop-only events into account in the same way as observation equivalent redundant transition removal: either by adding all possible selfloops or by assuming their presence as the algorithm executes. The worst-case time complexity is unchanged by this modification and remains at $O(|Q|^3)$ [26].

Another partitioning rule [12] is based on *reverse* observation equivalence.

Definition 18. [27] Let $G = \langle \mathbf{A}, Q, \to, Q^\circ \rangle$ be an automaton. An equivalence relation $\sim \subseteq Q \times Q$ is a *reverse observation equivalence* relation on G, if the following conditions hold for all $x_1, x_2 \in Q$ with $x_1 \sim x_2$.

- If $x_1 \in Q^\circ$, then $Q^\circ \overset{\varepsilon}{\Rightarrow} x_2$.
- For all states $w_1 \in Q$ and all traces $s \in \mathbf{A}^*$ such that $w_1 \overset{s}{\Rightarrow} x_1$ there exists a state $w_2 \in Q$ such that $w_2 \overset{s}{\Rightarrow} x_2$ and $w_1 \sim w_2$.

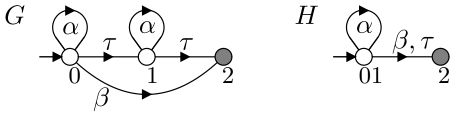

Figure 9: Reverse Observation Equivalence Rule.

Two states are reverse observation equivalent if they can be reached via the same traces from the initial states. Reverse observation equivalent states are essentially the result of nondeterministic branching. While not all nondeterminism can be removed if conflict equivalence is to be preserved, the *Non-α Determinisation Rule* [12] identifies reverse observation equivalent states that can be merged while preserving generalised nonblocking equivalence. This rule can be adapted to standard conflict equivalence by only merging states with outgoing τ-transitions.

Rule 9 (Reverse Observation Equivalence Rule). If $\sim$ is a reverse observation equivalence on an automaton G such that states without outgoing τ-transitions are only equated to themselves by $\sim$, then G can be replaced by $G/{\sim}$.

Example 10. States 0 and 1 in automaton G in Figure 9 are reverse observation equivalent, and both have τ-transitions outgoing. They can be merged into a single state 01 to obtain the abstraction $H \simeq_{\text{conf}} G$.

Note that the Enabled Continuation Rule is not applicable as states 0 and 1 are not incoming equivalent because of the transition $1 \xrightarrow{\alpha} 1$, and the Only Silent Incoming and Only Silent Outgoing Rules are not applicable either as state 1 does not have only τ-transitions incoming or outgoing.

Proposition 22. [12] Let $G = \langle \mathbf{A}, Q, \rightarrow, Q^\circ \rangle$ be an automaton, and let $\sim \subseteq Q \times Q$ be a reverse observation equivalence on G such that $x \sim y$ implies that $x = y$ or both $x \xrightarrow{\tau}$ and $y \xrightarrow{\tau}$ hold. Then $G \simeq_{\text{conf}} G/{\sim}$.

The reason why states with outgoing τ-transitions can be merged more easily than other states is that their continuations are "optional" [8]. Another automaton that is to ensure nonblocking with a τ-loop free automaton G only needs to ensure the reachability of terminal states from those of states of G that do not have outgoing τ-transitions.

Despite the apparent similarity to the Enabled Continuation Rule, the Reverse Observation Equivalence Rule cannot be generalised to take always enabled events into account. However, selfloop-only events can be used to compute a better reverse observation equivalence relation in the same way as explained above with weak observation equivalence. As in the case of weak observation equivalence, the worst-case time complexity is unchanged by the consideration of selfloop-only events and remains at $O(|Q|^3)$ [12].

4.8. Failing Events Rule

The final abstraction rule concerns failing events. A failing event is known to take the system to a blocking state in some automaton, i. e., in a state from where it is no longer possible to execute the termination event ω. Such events are often discovered when the model to be verified is blocking, and their presence can help to prove this fact more quickly. Whenever a failing event occurs, the composed system will be in a blocking state. Based on this observation, the *Failing Events Rule* redirects all transitions with failing events to a new blocking state, producing an automaton that blocks immediately when a failing event occurs.

Rule 10 (Failing Events Rule). If an event φ is failing in another automaton, then any transition $p \xrightarrow{\varphi} q$ can be replaced by $p \xrightarrow{\varphi} \bot$ where $\bot$ is a deadlock state without any outgoing transitions.

Example 11. Consider automaton G in Figure 10. Here, φ is a failing event, which means that all φ-transitions take the system to some blocking state in another automaton. The system is known to be blocking as soon as event φ gets executed, and therefore its transitions in G are replaced by transitions to the new deadlock state $\bot$, resulting in H. It holds that $G \overset{\text{fconf}}{\simeq}_{\{\varphi\}} H$.

Although the abstracted automaton H in Example 11 has more states than the original automaton G, it is simpler and more likely to help proving that the system is blocking. For example, event α becomes a failing event in H, because both its transitions take H to state 1, from where it is no longer possible to reach any state with ω enabled, and the fact that α is now failing can be used to simplify other automata.

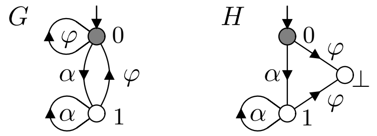

Figure 10: Failing Events Rule.

Proposition 23. Let $G = \langle \mathbf{A}, Q, \rightarrow_G, Q^\circ \rangle$ be an automaton with $p \xrightarrow{\varphi} q$, let $\perp \notin Q$ be a new state, and let $H = \langle \mathbf{A}, Q \cup \{\perp\}, \rightarrow_H, Q^\circ \rangle$ such that $\rightarrow_H = (\rightarrow_G \setminus \{(p, \varphi, q)\}) \cup \{(p, \varphi, \perp)\}$. Then $G \overset{\text{fconf}}{\simeq}_{\{\varphi\}} H$.

Proof. Let T be an automaton such that φ is a failing event in T.

First assume that $G \parallel T$ is nonblocking. To see that $H \parallel T$ is nonblocking, let $H \parallel T \xrightarrow{s} (x, x_T)$. This path does not include the introduced transition $p \xrightarrow{\varphi}_H \perp$, because otherwise the path has a prefix $H \parallel T \rightarrow (p, p_T) \xrightarrow{\varphi} (\perp, \perp_T)$, where this is the first occurrence of the removed transition, and where $\perp_T$ is a blocking state in T as φ is failing in T, so that by construction $G \parallel T \rightarrow (p, p_T) \xrightarrow{\varphi} (q, \perp_T)$ is blocking in contradiction to the assumption. As the path $H \parallel T \xrightarrow{s} (x, x_T)$ does not include the introduced transition $p \xrightarrow{\varphi}_H \perp$, it follows by construction that $G \parallel T \xrightarrow{s} (x, x_T)$. Since $G \parallel T$ is nonblocking, there exists $t \in \mathbf{A}_\tau^*$ such that $G \parallel T \xrightarrow{s} (x, x_T) \xrightarrow{t\omega}$. This path does not include the removed transition $p \xrightarrow{\varphi}_G q$, because otherwise the path has a prefix $G \parallel T \rightarrow (p, p_T) \xrightarrow{\varphi} (q, q_T)$ where q_T is a blocking state in T as φ is failing in T, so $G \parallel T$ is blocking in contradiction to the assumption. As the path $G \parallel T \xrightarrow{s} (x, x_T) \xrightarrow{t\omega}$ does not include the removed transition, it follows by construction that $H \parallel T \xrightarrow{s} (x, x_T) \xrightarrow{t\omega}$, i. e., $H \parallel T$ is nonblocking.

Conversely, assume that $H \parallel T$ is nonblocking. To see that $G \parallel T$ is nonblocking, let $G \parallel T \xrightarrow{s} (x, x_T)$. This path does not include the removed transition $p \xrightarrow{\varphi}_G q$, because otherwise the path has a prefix $G \parallel T \rightarrow (p, p_T) \xrightarrow{\varphi} (q, q_T)$ such that by construction $H \parallel T \rightarrow (p, p_T) \xrightarrow{\varphi} (\perp, q_T)$ is blocking in contradiction to the assumption. As the path $G \parallel T \xrightarrow{s} (x, x_T)$ does not include the removed transition, it follows by construction that $H \parallel T \xrightarrow{s} (x, x_T)$. Since $H \parallel T$ is nonblocking, there exists $t \in \mathbf{A}_\tau^*$ such that $H \parallel T \xrightarrow{s} (x, x_T) \xrightarrow{t\omega}$. This path does not include the introduced transition $p \xrightarrow{\varphi}_H \perp$, because otherwise the path has a prefix $H \parallel T \rightarrow (p, p_T) \xrightarrow{\varphi} (\perp, \perp_T)$, which means that $H \parallel T$ is blocking in contradiction to the assumption. Then it follows by construction that $G \parallel T \xrightarrow{s} (x, x_T) \xrightarrow{t\omega}$, i. e., $G \parallel T$ is nonblocking. □

Proposition 23 shows that a single failing transition can be redirected to a deadlock state while preserving conflict equivalence with respect to failing events. This result can be applied repeatedly to redirect all failing transitions in an automaton or in the entire system. Separate blocking states $\perp$ resulting from multiple applications of the rule can be merged into a single state based on weak observation equivalence. This abstraction is likely to result in unreachable states, which can be removed. The Failing Events Rule is easy to implement by visiting all transitions of an automaton and runs in $O(|\rightarrow|)$.

5. Compositional Nonblocking Verification Algorithm

This section describes the compositional nonblocking verification algorithm as it is implemented in the discrete event systems tool Waters/Supremica [28], which is freely available for download [29]. The presentation starts with a basic algorithm in Section 5.1, which is gradually improved in the following sections by adding special events and other techniques.

5.1. Basic Compositional Nonblocking Verification Algorithm

Algorithm 1 is the basic compositional nonblocking verification algorithm, which takes as input a system

$$\mathcal{G} = G_1 \parallel \cdots \parallel G_n \tag{9}$$

of automata to be checked for the nonblocking property. As a first step, the loop on lines 2–4 attempts to simplify every automaton G_i in the input as much as possible using the abstraction rules in Section 4.

Algorithm 1 Basic Compositional Nonblocking Verification

```
1: procedure conflictCheck($\mathcal{G} = \{G_1, \ldots, G_n\}$)
2: for all $G_i \in \mathcal{G}$ do
3:     $G_i \leftarrow \mathrm{simplify}(G_i)$
4: end for
5: while $|\mathcal{G}| > 2$ do
6:     choose candidate $\mathcal{C} \subseteq \mathcal{G}$
7:     $H \leftarrow \parallel \mathcal{C}$
8:     $H' \leftarrow \mathrm{simplify}(H)$
9:     $\mathcal{G} \leftarrow (\mathcal{G} \setminus \mathcal{C}) \cup \{H'\}$
10: end while
11: return monolithicConflictCheck($\mathcal{G}$)
```

When every automaton has been simplified individually, the algorithm enters the main loop on line 5, where it repeatedly composes and simplifies automata. It starts on line 6 by choosing a so-called *candidate*, i. e., a subset of the automata from the system $\mathcal{G}$, which are composed and turned into a single automaton on line 7. The new automaton is then simplified on line 8. Afterwards, the automata that were in the candidate are removed from the system and replaced with the simplified automaton on line 9. Then the loop starts over by choosing another candidate.

This loop continues until only two automata remain, which are verified using a straightforward monolithic conflict check [1] on line 11. Termination of the loop is guaranteed, because only candidates consisting of at least two automata are considered, so each iteration reduces the number of automata. The monolithic conflict check constructs the synchronous composition of the two remaining automata and checks whether it is nonblocking. Simplification of the final synchronous composition is avoided by stopping the loop as soon as only two automata remain, because it is faster to perform a nonblocking check than to simplify an automaton.

The simplify() procedure on lines 3 and 8 simplifies an automaton by applying the abstraction rules introduced in Section 4 in the following order.

1. **τ-Loop Removal.** The first step is to find all cycles of τ-transitions in the automaton and merge each of them into a single state. This is a special case of the Weak Observation Equivalence Rule (Rule 8), which is implemented efficiently in $O(|{\rightarrow}|)$ time using Tarjan's algorithm [18].
2. **Observation Equivalent Transition Removal Rule** (Rule 7).
3. **Only Silent Incoming Rule** (Rule 3).
4. **Only Silent Outgoing Rule** (Rule 4).
5. **Limited Certain Conflicts Rule** (Rule 5).
6. **Weak Observation Equivalence Rule** (Rule 8).
7. **Enabled Continuation Rule** (Rule 1) together with **Active Events Rule** (Rule 2).
8. **Reverse Observation Equivalence Rule** (Rule 9).

The order of rule applications can be important as the rules are not commuting or confluent, and the application of one rule can make another applicable or inapplicable. Yet, the precise dependencies are hard to predict, and experiments [30] suggest that the order has little impact in practice. The above sequence first removes τ-loops and redundant transitions to establish the preconditions of the other rules or increase the likelihood of them becoming applicable. Then the quicker transition-processing rules are applied in steps 3–5 before the more time-consuming partitioning rules in steps 6–8.

Another important consideration for the performance of compositional verification is the choice of the candidate on line 6. This issue is discussed further in Section 5.4 below.

5.2. Including Special Events

This section explains how the basic compositional nonblocking verification algorithm is extended to take special events into account. Four different types of special events have been introduced in Section 3, which need to be

Algorithm 2 Compositional Nonblocking Verification with Special Events

```
1:  procedure conflictCheck($\mathcal{G} = \{G_1, \ldots, G_n\}$)
2:  eventInfo $\leftarrow$ initEventInfo($\mathcal{G}$)
3:  needsSimplification $\leftarrow \mathcal{G}$
4:  while $|\mathcal{G}| > 2$ do
5:      while needsSimplification $\neq \emptyset$ do
6:          choose and remove $G_i$ from needsSimplification
7:          $G'_i \leftarrow$ specialEventsRemoval($G_i$, eventInfo)
8:          $G''_i \leftarrow$ simplify($G'_i$, eventInfo)
9:          updateInfo($G_i$, $G''_i$, eventInfo, needsSimplification)
10:         $\mathcal{G} \leftarrow (\mathcal{G} \setminus \{G_i\}) \cup \{G''_i\}$
11:     end while
12:     choose candidate $\mathcal{C} \subseteq \mathcal{G}$
13:     $H \leftarrow \parallel \mathcal{C}$
14:     $H' \leftarrow$ simplify($H$, eventInfo)
15:     updateInfo($\mathcal{C}$, $H$, eventInfo, needsSimplification)
16:     $\mathcal{G} \leftarrow (\mathcal{G} \setminus \mathcal{C}) \cup \{H'\}$
17: end while
18: return monolithicConflictCheck($\mathcal{G}$)
```

identified and used to inform the simplification process. In order to keep track of the different special event types, it is helpful to associate with each event σ an *event information* record, which contains the following information.

- The list of automata G_i in the system that have σ in their alphabet.
- The list of automata G_i in the system in which σ is always enabled.
- The list of automata G_i in the system in which σ is selfloop-only.
- A flag indicating whether σ has been found to be *failing*.
- A flag indicating whether σ has been found to be *blocked*.

By Propositions 7 and 9, an event assumes the *failing* or *blocked* status as soon as it is found to be failing or blocked in a single automaton. Therefore it is enough to record this status in a single flag, whereas the *always enabled* or *selfloop-only* status is recorded for each automaton individually and only takes effect when the event has this status for all its automata except one. Based on the above information, it is easy to determine whether an event should be treated as special when simplifying a given automaton. In particular, the following additional properties can be determined quickly from an event information record.

- The event is *local* if the list of automata that have it in their alphabet consists of a single automaton G_i. In this case, the event can be hidden, i. e., replaced by τ when simplifying G_i.
- The event can be treated as always *always enabled* (or *selfloop-only*) when simplifying automaton G_i if all automata with this event in their alphabet except G_i appear in the list of *always enabled* (or *selfloop-only*) automata for this event.
- The event is *globally selfloop-only* if it is listed as selfloop-only in all automata that have it in their alphabet.

Algorithm 2 shows how the basic compositional nonblocking verification algorithm is extended to use event information and take special events into account. In this algorithm, the initial loop to simplify all automata (lines 2–4 of Algorithm 1) is replaced by a set needsSimplification to keep track of automata that may need to be simplified as the status of events changes, and the task of simplification is moved into the main loop.

First, the event information records are created on line 2. The initEventInfo() procedure loops over the transitions of all the automata and checks for selfloop-only, blocked, and failing events, and initialises the event information accordingly. Always enabled events are not detected at this stage, because the efficient method to detect always enabled events based on Proposition 18 requires τ-loop free automata, and the automata are not yet guaranteed to be τ-loop free at this stage. All events are assumed to be not always enabled initially. The set $\mathrm{needsSimplification}$ is initialised on line 3 to contain all automata of the system.

At the beginning of each iteration of the main loop, the inner loop on lines 5–11 attempts to simplify all automata contained in $\mathrm{needsSimplification}$. First, the $\mathrm{specialEventsRemoval}()$ procedure on line 7 removes blocked and globally selfloop-only events according to Propositions 8 and 20, and redirects failing events according to Proposition 23. If any transitions are removed or changed, a reachability check is performed to remove any additional states and transitions that have become unreachable. Then the $\mathrm{simplify}()$ procedure line 8 applies abstraction rules to simplify the automaton subject to special events. Both the $\mathrm{specialEventsRemoval}()$ and $\mathrm{simplify}()$ procedures read the event information to determine which events should be treated as local, selfloop-only, or always enabled.

After simplification, the $\mathrm{updateInfo}()$ procedure on line 9 checks the event information for all events in the simplified automaton to see whether the event receives any additional special status. This includes a check for always enabled events in the simplified automaton using Proposition 18, as the automaton now is known to be τ-loop free. If the event status of an event changes, all automata using it are added to the set $\mathrm{needsSimplification}$ so they can be simplified again using the new event status. Termination of the inner loop is guaranteed, because once an event receives a status such as *blocked* or *selfloop-only*, it will never lose that status again, and the number of events is finite.

Once all automata have been simplified, Algorithm 2 proceeds to compose a candidate on line 12. After composition and simplification, which takes event information into account, the event information for the events of the simplified automaton is updated on line 15. If this causes the status of any event to change, the affected automata are added to the set $\mathrm{needsSimplification}$, so they get simplified before the next iteration. Algorithm 2 is designed to perform as much simplification and event removal as possible before composing candidates.

5.3. Event-Disjoint Subsystems, Early Termination, and State Limits

This section shows some further steps to improve Algorithm 2. For practical use, it is important to check for early termination conditions, and to control the amount of memory used by the automata constructed.

Sometimes the automata to be checked for the nonblocking property can be separated into groups of automata that do not have any events in common. The system may have this property initially, or it may arise after simplification and event removal. If the system can be separated, then the following fact can be used to verify the event-disjoint subsystems separately.

Proposition 24. Let $G_1 = \langle \mathbf{A}_1, Q_1, \rightarrow_1, Q_1^\circ \rangle$ and $G_2 = \langle \mathbf{A}_2, Q_2, \rightarrow_2, Q_2^\circ \rangle$ be two automata such that $\mathbf{A}_1 \cap \mathbf{A}_2 = \emptyset$. Then $G_1 \parallel G_2$ is nonblocking if and only if both G_1 and G_2 are nonblocking.

If two automata do not share any events, then their synchronous composition is the same as a *shuffle product* [1], which is nonblocking if and only if both automata are nonblocking on their own. Proposition 24 is a simple consequence of the definition of synchronous composition.

Algorithm 3 takes advantage of Proposition 24 by splitting the system into event-disjoint subsystems and verifying them separately. If any subsystem is found to be blocking, then the entire system is blocking and the remaining subsystems do not need to be checked. This idea is implemented using a priority queue of subsystems, ordered by size so that smaller subsystems are checked first. At the start, the complete system $\mathcal{G}$ is added as the only entry to the queue on line 4. Then the loop on lines 5–10 processes each subsystem using the $\mathrm{conflictCheckSubsystem}()$ procedure in Algorithm 4. This is a modified version of Algorithm 2, which may assert that the subsystem is *"blocking"* or *"nonblocking"*, or it may return *"split"* to indicate that the subsystem has been split further and two or more event-disjoint subsystems have been added to the queue. If a subsystem is found to be *"blocking"*, then Algorithm 3 immediately stops and returns this result on line 8. Otherwise the loop continues until every single subsystem is found to be *"nonblocking"*, in which case line 11 asserts that the complete system is nonblocking.

Algorithm 4 implements two further improvements over Algorithm 2 to check whether a subsystem is nonblocking. The first of these improvements is *early termination*. In some cases, it is immediately clear whether a system is blocking or nonblocking. If every state in the system is terminal then the system must be nonblocking, and if some automaton has no terminal states then the system must be blocking. The following result follows immediately from the definition of synchronous composition.

Proposition 25. Let $\mathcal{G} = G_1 \parallel \cdots \parallel G_n$.

Algorithm 3 Compositional Nonblocking Verification with Event-Disjoint Subsystems Queue

```
procedure conflictCheck($\mathcal{G} = \{G_1, \ldots, G_n\}$)
eventInfo $\leftarrow$ initEventInfo($\mathcal{G}$)
needsSimplification $\leftarrow \mathcal{G}$
queue $\leftarrow \{\mathcal{G}\}$
while queue is not empty do
    remove the first subsystem $\mathcal{G}$ from queue
    if conflictCheckSubsystem($\mathcal{G}$, eventInfo, needsSimplification, queue) $=$ "blocking" then
        return "blocking"
    end if
end while
return "nonblocking"
```

Algorithm 4 Subsystem Verification

```
procedure conflictCheckSubsystem($\mathcal{G} = \{G_1, \ldots, G_n\}$, eventInfo, needsSimplification, queue)
result $\leftarrow$ earlyTerminationCheck($\mathcal{G}$)
if result $=$ "blocking" or result $=$ "nonblocking" then
    return result
end if
while $|\mathcal{G}| > 2$ do
    while needsSimplification $\neq \emptyset$ do
        choose and remove $G_i$ from needsSimplification
        $G'_i \leftarrow$ specialEventsRemoval($G_i$, eventInfo)
        $G''_i \leftarrow$ simplify($G'_i$, eventInfo)
        updateInfo($G_i$, $G''_i$, eventInfo, needsSimplification)
        $\mathcal{G} \leftarrow (\mathcal{G} \setminus \{G_i\}) \cup \{G''_i\}$
    end while
    result $\leftarrow$ earlyTerminationCheck($\mathcal{G}$)
    if result $=$ "blocking" or result $=$ "nonblocking" then
        return result
    else if disjointSubsystemsCheck($\mathcal{G}$, queue) then
        return "split"
    end if
    repeat
        if all possible candidates are marked as "failed" then
            return monolithicConflictCheck($\mathcal{G}$)
        end if
        choose candidate $\mathcal{C} \subseteq \mathcal{G}$
        $H \leftarrow \parallel \mathcal{C}$
        if $\parallel \mathcal{C}$ exceeds state limit during composition then
            mark $\mathcal{C}$ as "failed"
            success $\leftarrow$ false
        else
            $H' \leftarrow$ simplify($H$, eventInfo)
            updateInfo($\mathcal{C}$, $H$, eventInfo, needsSimplification)
            $\mathcal{G} \leftarrow (\mathcal{G} \setminus \mathcal{C}) \cup \{H'\}$
            success $\leftarrow$ true
        end if
    until success
end while
return monolithicConflictCheck($\mathcal{G}$)
```

- If some automaton $G_i = \langle \mathbf{A}_i, Q_i, \rightarrow_i, Q_i^\circ \rangle$ has no terminal states, i. e., $x \overset{\omega}{\rightarrow}_i$ does not hold for any $x \in Q_i$, then $\mathcal{G}$ is blocking.
- If every automaton $G_i = \langle \mathbf{A}_i, Q_i, \rightarrow_i, Q_i^\circ \rangle$ has only terminal states, i. e., $x \overset{\omega}{\rightarrow}_i$ for all $x \in Q_i$, then $\mathcal{G}$ is nonblocking.

The conditions in Proposition 25 can be checked frequently and quickly by inspecting the states of all automata. As soon as this simple check determines the system to be blocking or nonblocking, this answer can be returned immediately as the result of nonblocking verification without the need for further synchronous composition.

The last improvement implemented in Algorithm 4 is the use of *state limits*. Because of the state-space explosion problem, synchronous composition may try to create an automaton that is too large to fit into memory. This may occur due to a poorly chosen candidate, or because simplification is not sufficient to help verify a massive system. In an attempt to avoid the construction of overly large automata, Algorithm 4 enforces a state limit for synchronous composition. If the synchronous composition of a candidate becomes too large, it is aborted and another candidate is attempted instead.

Using these improvements, Algorithm 4 proceeds as follows to check a subsystem passed from Algorithm 3. First, line 2 checks for early termination. The earlyTerminationCheck() procedure checks the conditions of Proposition 25 and, if successful, it is asserted immediately that the subsystem is *"blocking"* or *"nonblocking"*. Otherwise, if the result is inconclusive, the algorithm enters the main loop. At the beginning of each iteration, special events are removed and individual automata are simplified in the same way as in Algorithm 2. For the case that this causes any automaton to be changed, early termination is checked again on line 14.

After event removal and simplification, line 17 checks for event-disjoint subsystems. If the current system $\mathcal{G}$ can be split into event-disjoint subsystems, then the disjointSubsystemsCheck() procedure performs the split, puts the resultant subsystems into the queue, and returns true. Then line 18 returns *"split"* to signal to Algorithm 3 that there are new subsystems in the queue that need to be checked. The check for event-disjoint subsystems involves a search of the event-dependency structure of $\mathcal{G}$ for strongly connected components, which can be time-consuming for systems with a large number of events. However, the check is rarely needed in practice. Once the system has been found to be connected, it can only be split if event removal or simplification causes an event to be removed from an automaton. Performance is improved by remembering when the event alphabet of an automaton changes after lines 9, 10, or 30, and only performing the event-disjointness check on line 17 after an event has been removed.

If there are no event-disjoint subsystems, the loop in lines 20–35 attempts to find and compose a candidate. If the synchronous composition of a candidate on line 25 exceeds the state limit, it is aborted and the candidate is marked as *"failed"* on line 27 to prevent it from being selected again. Then the loop starts over and attempts another candidate. When all possible candidates have been marked as *"failed"*, the algorithm stops on line 22 with a final attempt to verify the remaining automata by means of a monolithic conflict check.

5.4. Candidate Selection

A key aspect of the compositional verification algorithm is how automata are selected for composition on line 24 of Algorithm 4. The algorithm proposed here follows a two-step approach [8]. First, in the *preselection* step, some candidate sets of automata are formed. Second, in the *selection* step, the candidates from the preselection step are heuristically evaluated to choose a most promising candidate.

The preselection step is needed to narrow down the set of possible candidates. In principle, every proper subset of the automata of the system with at least two elements could be a candidate. These are far too many possible candidates, many of which are clearly inferior to other choices. For instance, if two automata do not share any events, then their synchronous composition increases the number of states to the product of the numbers of states of the two automata, and it is unlikely that any simplification will be possible. Candidates marked as *"failed"* on line 27 of Algorithm 4, because their composition is too large, are also suppressed at this stage.

The experiments below in Section 6 use the preselection strategy **MustL**, used in previous work [8, 12], and a variation **MustSp** that takes special events into account.

- The preselection strategy **MustL** [8] ensures that every candidate must have new local events. For every event in the system, a candidate is created that consists of the automata using that event. If two events appear in exactly the same automata, only one candidate consisting of these automata is created, and the fact that the candidate has two local events is recorded, so that it can be used afterwards in the selection step.

- The preselection strategy **MustSp** is a variation of **MustL** that takes special events into account. **MustSp** ensures that candidates must introduce new events that are at least always enabled or selfloop-only in the remainder of the system. For every event σ in the system, two candidates are created: the first candidate consists of the automata using the event σ where σ is not always enabled, and the second consists of the automata using the event σ where σ is not selfloop-only.

Both **MustL** and **MustSp** guarantee some degree of abstraction by ensuring that every candidate has at least one special event. **MustSp** may produce smaller candidates, while **MustL** may be more goal-driven towards the hiding and removal of events. Failing and blocked events are not considered for candidate selection, because they are handled by the $\mathrm{specialEventsRemoval}()$ procedure (line 9 of Algorithm 4) immediately when encountered.

Assuming the system consists of a set of automata,

$$\mathcal{G} = \{G_1, \ldots, G_n\} \quad \text{where} \quad G_i = \langle \mathbf{A}_i, Q_i, \rightarrow_i, Q_i^\circ \rangle \tag{10}$$

the preselection strategy **MustL** or **MustSp** produces a collection $\mathcal{C}_1, \ldots, \mathcal{C}_m$ of candidates $\mathcal{C}_j \subseteq \mathcal{G}$. In the second step of candidate selection, a best candidate is chosen from these preselected candidates. This is done by assigning a numerical heuristic value to each candidate $\mathcal{C}_j$, and choosing the candidate with the smallest heuristic value. The following four methods to compute this heuristic value are considered here.

- The selection strategy **MinS** [8] estimates the number of states of the abstracted synchronous composition of candidates and chooses the candidate with the smallest estimate. More precisely, the heuristic value of a candidate $\mathcal{C} \subseteq \mathcal{G}$ is:

$$\textbf{MinS} = \frac{|\mathbf{A}_{\mathrm{shared}}|}{|\mathbf{A}_{\mathcal{C}}|} \cdot \prod_{G_i \in \mathcal{C}} |Q_i| \ , \tag{11}$$

where

$$\mathbf{A}_{\mathcal{C}} = \bigcup_{G_i \in \mathcal{C}} \mathbf{A}_i \ ; \tag{12}$$

$$\mathbf{A}_{\mathrm{shared}} = \mathbf{A}_{\mathcal{C}} \cap \bigcup_{G_i \in \mathcal{G} \setminus \mathcal{C}} \mathbf{A}_i \ . \tag{13}$$

The number of states of the synchronous composition is roughly estimated as the product of the numbers of states of the automata composed, and the result is multiplied with the ratio of the number of events shared with the rest of the system (13) over the total number of events in the candidate (12). This formula assumes that local events can simplify the candidate and reduces the estimated number of states accordingly.

- The selection strategy **MinSSp** is a variation of **MinS** to take special events into account. The estimate (11) is reduced for candidates that produce always enabled or selfloop-only events as follows:

$$\textbf{MinSSp} = \frac{|\mathbf{A}_{\mathrm{shared}}| - 0.5 \cdot |\mathbf{A}_{\mathrm{ae}}| - 0.5 \cdot |\mathbf{A}_{\mathrm{sl}}|}{|\mathbf{A}_{\mathcal{C}}|} \cdot \prod_{G_i \in \mathcal{C}} |Q_i| \ , \tag{14}$$

where

$$\mathbf{A}_{\mathrm{ae}} = \{\, \sigma \in \mathbf{A}_{\mathrm{shared}} \mid \sigma \text{ is always enabled in every } G_i \in \mathcal{G} \setminus \mathcal{C} \,\} \ ; \tag{15}$$

$$\mathbf{A}_{\mathrm{sl}} = \{\, \sigma \in \mathbf{A}_{\mathrm{shared}} \mid \sigma \text{ is selfloop-only in every } G_i \in \mathcal{G} \setminus \mathcal{C} \,\} \ . \tag{16}$$

In the estimate (14), shared events that are always enabled or selfloop-only in all automata outside of the candidate have only half the weight as regular shared events. This slightly reduces the estimated state number for candidates with such events, increasing their chance to be selected.

- The selection strategy **MinSync** [12] computes the synchronous composition of the automata in a candidate and chooses the candidate with the fewest states in the synchronous composition. This is more accurate than estimating the state number by multiplying the state numbers of the automata, however local or special events are not taken into account.

While it seems time-consuming to compute the full synchronous composition for many candidates of large automata, the effort for **MinSync** can be limited. As the running minimum is computed for a list of candidates, which can first be ordered by a state estimate such as **MinS**, the synchronous composition of new candidates can be stopped as soon as the state number exceeds the smallest state number found for another candidate. This avoids the computation of large synchronous compositions and makes the **MinSync** strategy feasible.

- The selection strategy **MinF** [31] chooses the candidate with the smallest *frontier*, where the frontier of a candidate $\mathcal{C} \subseteq \mathcal{G}$ is the number of automata outside of $\mathcal{C}$ that share events with $\mathcal{C}$:

$$\mathbf{MinF} = |\{\, G_j \in \mathcal{G} \setminus \mathcal{C} \mid \mathbf{A}_\mathcal{C} \cap \mathbf{A}_j \neq \emptyset \,\}| \,. \tag{17}$$

  This strategy avoids the size of automata as a criterion, and uses the event-dependency structure of the system instead. The hope here is to compose groups of more tightly coupled automata that have little interaction with the rest of the system.

Other preselection and selection strategies [8] have also been considered. However, they were found to perform poorly for the large models considered in the experiments, and therefore are not investigated further here.

6. Experimental Results

The Compositional Nonblocking Verification Algorithm with Event-Disjoint Subsystems Queue (Algorithm 3) has been used to verify the nonblocking property of the same models used in previous work [8, 12]. The test suite includes complex industrial models and case studies from various application areas such as manufacturing systems, communication protocols, and automotive electronics. Included here are the previously studied models [12] with at least $5 \cdot 10^7$ reachable states in their synchronous composition. The following list gives some details about these models.

aip Model of the automated manufacturing system of the Atelier Inter-établissement de Productique [32]. The tests consider three early versions (**aip0**) [33], and a more detailed version (**aip1**) [34], which has been modified for a parametrisable number of pallets.

fencaiwon09 Model of a production cell in a metal-processing plant [35].

ftechnik Flexible production cell model [36].

profisafe PROFIsafe field bus protocol model [37]. 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 [38] in four different versions and designs.

tip3 Model of the interaction between a mobile client and event-based servers of a Tourist Information System [39].

verriegel Car central locking system, originally from the KORSYS project [40].

6link Models of a cluster tool for wafer processing [41].

Figures 11–13 and Tables 1–2 show the results of several experiments to verify the nonblocking property of the above models using compositional verification. The tables show for each model the number of automata in the model (Aut), the number of reachable states in the synchronous composition (State space) if known, and whether or not the model is nonblocking (Res). Then they show, for different compositional verification strategies, 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). A final states number of 0 indicates that compositional verification has terminated early without computing a final synchronous composition. The best result in each category is highlighted in bold.

All experiments are run on a standard desktop computer using a single core 3.3 GHz CPU and 8 GB of RAM. The state limit for candidates in these experiments is set to 100,000 states. If the synchronous composition of a candidate has more states, it is discarded and another candidate is chosen instead. The state limit for the final synchronous

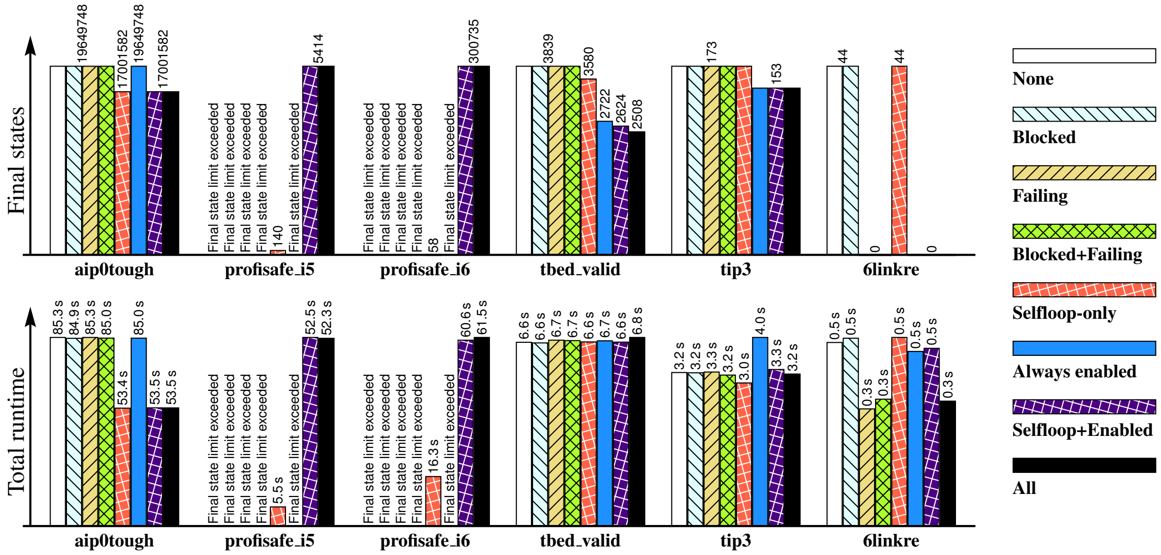

Figure 11: Final state numbers and runtimes using different types of special events.

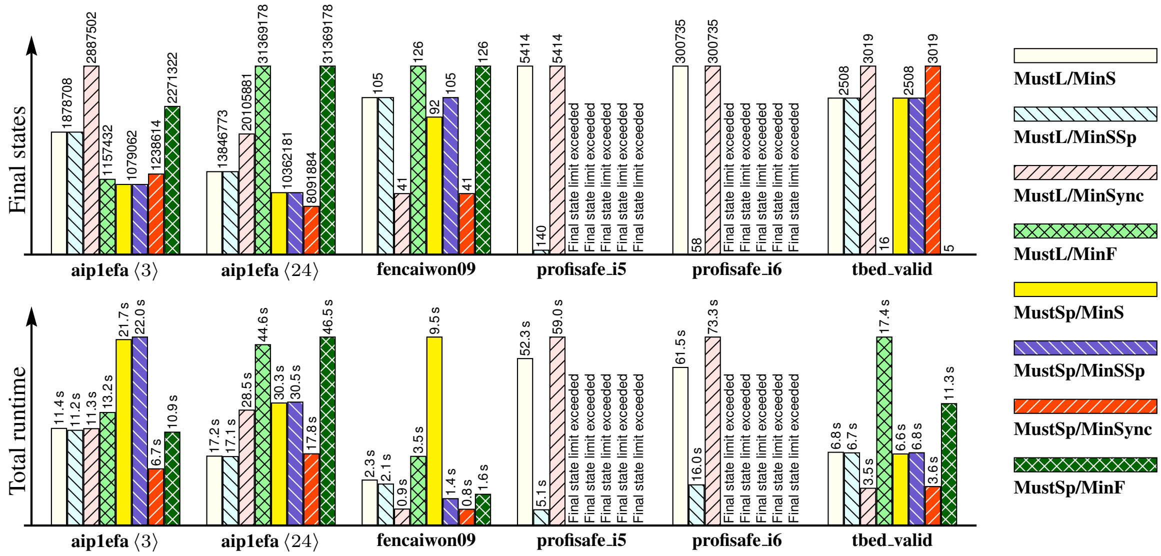

Figure 12: Final state numbers and runtimes with different candidate selection strategies.

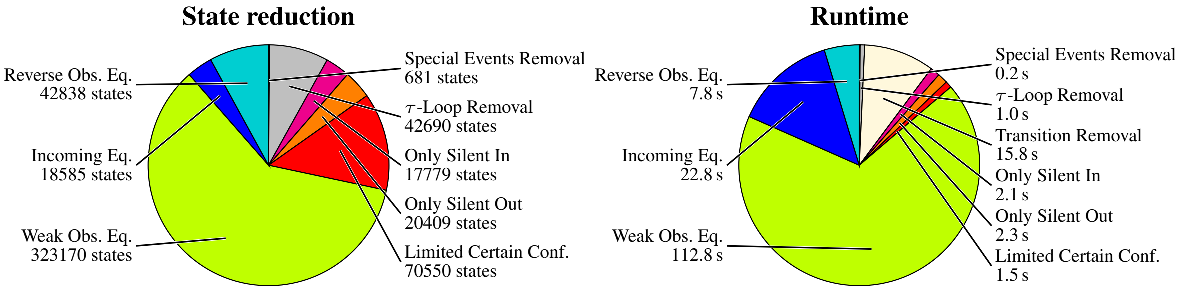

Figure 13: Performance statistics of abstraction rules.

Table 1: Experimental results using different types of special events.

Model				**None**			**Blocked**			**Failing**			**Blocked+Failing**		
Name	Aut	State space	Res	Peak States	Final States	Time [s]	Peak States	Final States	Time [s]	Peak States	Final States	Time [s]	Peak States	Final States	Time [s]
aip0aip	117	$1.02 \cdot 10^{9}$	yes	1090	5	1.1	1090	5	1.1	1090	5	1.1	1090	5	1.1
aip0alps	35	$3.00 \cdot 10^{8}$	no	18	16	**0.2**	18	16	0.2	18	16	0.2	18	16	0.2
aip0tough	60	$1.02 \cdot 10^{10}$	no	96049	19649748	85.3	96049	19649748	84.9	96049	19649748	85.3	96049	19649748	85.0
aip1efa $\langle 3 \rangle$	50	$6.88 \cdot 10^{8}$	yes	40290	1878708	11.3	40290	1878708	11.3	40290	1878708	11.3	40290	1878708	11.3
aip1efa $\langle 16 \rangle$	50	$9.50 \cdot 10^{12}$	no	65520	13799628	20.7	65520	13799628	20.7	65520	13799628	**20.6**	65520	13799628	20.8
aip1efa $\langle 24 \rangle$	50	$1.83 \cdot 10^{13}$	no	6384	13846773	**16.9**	6384	13846773	17.0	6384	13846773	17.0	6384	13846773	17.0
fencaiwon09	32	$1.03 \cdot 10^{8}$	yes	10421	105	**1.8**	10421	105	1.9	10421	105	1.8	10421	105	2.2
fencaiwon09b	31	$8.93 \cdot 10^{7}$	no	10421	81	**2.3**	10421	81	2.3	10421	81	2.4	10421	81	2.4
ftechnik	36	$1.21 \cdot 10^{8}$	no	172	0	0.3	172	0	0.4	172	0	0.3	172	0	0.4
profisafe_i4	82		yes	Final state limit exceeded			Final state limit exceeded			Final state limit exceeded			Final state limit exceeded		
profisafe_i5	88		yes	Final state limit exceeded			Final state limit exceeded			Final state limit exceeded			Final state limit exceeded		
profisafe_i6	94		yes	Final state limit exceeded			Final state limit exceeded			Final state limit exceeded			Final state limit exceeded		
tbed_ctct	84	$3.94 \cdot 10^{13}$	no	43825	0	**11.3**	43825	0	11.4	43825	0	11.4	43825	0	11.4
tbed_hisc0	196	$5.99 \cdot 10^{12}$	yes	1757	33	2.5	1757	33	2.5	1757	33	2.6	1757	33	2.5
tbed_hisc1	184	$2.87 \cdot 10^{17}$	no	193	**0**	0.9	41	2	0.6	41	**0**	0.7	41	2	0.6
tbed_valid	84	$3.01 \cdot 10^{12}$	yes	33529	3839	6.6	33529	3839	**6.6**	33529	3839	6.7	33529	3839	6.7
tip3	58	$2.27 \cdot 10^{11}$	yes	6399	173	3.2	6399	173	3.2	6399	173	3.3	6399	173	3.2
tip3_bad	54	$5.25 \cdot 10^{10}$	no	1176	14	1.0	1176	14	1.0	1176	14	1.1	1176	14	1.1
verriegel3	53	$9.68 \cdot 10^{8}$	yes	**3107**	2	1.8	**3107**	2	1.8	**3107**	2	1.8	**3107**	2	1.8
verriegel3b	52	$1.32 \cdot 10^{9}$	no	76	0	0.4	162	0	0.4	**27**	0	0.4	**27**	0	0.3
verriegel4	65	$4.59 \cdot 10^{10}$	yes	**2455**	2	1.6	**2455**	2	1.6	**2455**	2	1.6	**2455**	2	1.6
verriegel4b	64	$6.26 \cdot 10^{10}$	no	151	0	0.6	386	0	0.8	**27**	0	0.5	**27**	0	0.4
6linka	53	$2.45 \cdot 10^{14}$	no	64	0	0.4	64	0	0.4	64	0	0.4	64	0	0.4
6linki	53	$2.75 \cdot 10^{14}$	no	61	0	**0.3**	61	0	0.3	61	0	0.3	61	0	0.3
6linkp	48	$4.43 \cdot 10^{14}$	no	32	0	0.3	32	0	**0.3**	32	0	0.3	32	0	0.3
6linkre	59	$6.21 \cdot 10^{13}$	no	118	44	0.5	118	44	0.5	**29**	**0**	**0.3**	**29**	**0**	0.3

Model				**Selfloop-only**			**Always enabled**			**Selfloop+Enabled**			**All**		
Name	Aut	State space	Res	Peak States	Final States	Time [s]	Peak States	Final States	Time [s]	Peak States	Final States	Time [s]	Peak States	Final States	Time [s]
aip0aip	117	$1.02 \cdot 10^{9}$	yes	1090	5	**1.1**	1090	5	1.1	1090	5	1.1	1090	5	1.1
aip0alps	35	$3.00 \cdot 10^{8}$	no	18	16	0.2	18	16	0.2	18	16	0.2	18	16	0.2
aip0tough	60	$1.02 \cdot 10^{10}$	no	96049	**17001582**	**53.4**	96049	19649748	85.0	96049	**17001582**	53.5	96049	**17001582**	53.5
aip1efa $\langle 3 \rangle$	50	$6.88 \cdot 10^{8}$	yes	40290	1878708	**11.2**	40290	1878708	11.3	40290	1878708	11.3	40290	1878708	11.4
aip1efa $\langle 16 \rangle$	50	$9.50 \cdot 10^{12}$	no	65520	13799628	20.8	65520	13799628	20.7	65520	13799628	21.9	65520	13799628	20.7
aip1efa $\langle 24 \rangle$	50	$1.83 \cdot 10^{13}$	no	6384	13846773	17.0	6384	13846773	17.2	6384	13846773	17.3	6384	13846773	17.2
fencaiwon09	32	$1.03 \cdot 10^{8}$	yes	10421	105	1.9	10421	105	2.2	10421	105	2.1	10421	105	2.3
fencaiwon09b	31	$8.93 \cdot 10^{7}$	no	10421	81	2.4	10421	81	2.4	10421	81	2.4	10421	81	2.4
ftechnik	36	$1.21 \cdot 10^{8}$	no	172	0	0.3	172	0	**0.3**	172	0	0.3	172	0	0.3
profisafe_i4	82		yes	**7696**	**36**	**3.1**	Final state limit exceeded			37044	4440	13.7	37044	4440	13.7
profisafe_i5	88		yes	**16408**	**140**	**5.5**	Final state limit exceeded			98304	5414	52.5	98304	5414	52.3
profisafe_i6	94		yes	60200	**58**	**16.3**	Final state limit exceeded			**52224**	300735	60.6	**52224**	300735	61.5
tbed_ctct	84	$3.94 \cdot 10^{13}$	no	43825	0	11.6	43825	0	11.3	43825	0	11.3	43825	0	11.4
tbed_hisc0	196	$5.99 \cdot 10^{12}$	yes	1757	33	**2.3**	**1705**	33	2.4	**1705**	33	2.4	**1705**	33	2.4
tbed_hisc1	184	$2.87 \cdot 10^{17}$	no	193	**0**	0.8	98	**0**	0.8	98	**0**	0.8	**19**	**0**	**0.4**
tbed_valid	84	$3.01 \cdot 10^{12}$	yes	33529	3580	6.6	33529	2722	6.7	33529	2624	6.6	33529	**2508**	6.8
tip3	58	$2.27 \cdot 10^{11}$	yes	**6351**	173	**3.0**	12303	**153**	4.0	**6351**	**153**	3.3	**6351**	**153**	3.2
tip3_bad	54	$5.25 \cdot 10^{10}$	no	**992**	**0**	**0.8**	1176	**0**	1.3	**992**	**0**	0.9	**992**	**0**	0.9
verriegel3	53	$9.68 \cdot 10^{8}$	yes	**3107**	2	**1.3**	3171	2	1.9	3171	2	1.6	3171	2	1.5
verriegel3b	52	$1.32 \cdot 10^{9}$	no	76	0	0.4	76	0	0.4	76	0	0.4	**27**	0	**0.3**
verriegel4	65	$4.59 \cdot 10^{10}$	yes	**2455**	2	**1.4**	2500	2	1.6	2500	2	1.5	2500	2	1.5
verriegel4b	64	$6.26 \cdot 10^{10}$	no	151	0	0.6	151	0	0.6	151	0	0.6	**27**	0	**0.4**
6linka	53	$2.45 \cdot 10^{14}$	no	64	0	0.4	64	0	**0.4**	64	0	0.4	64	0	0.4
6linki	53	$2.75 \cdot 10^{14}$	no	61	0	0.3	61	0	0.3	61	0	0.3	61	0	0.3
6linkp	48	$4.43 \cdot 10^{14}$	no	32	0	0.3	32	0	0.3	32	0	0.3	32	0	0.3
6linkre	59	$6.21 \cdot 10^{13}$	no	118	44	0.5	106	**0**	0.5	106	**0**	0.5	**29**	**0**	0.3

Table 2: Experimental results with different candidate selection strategies.

Model				MustL/MinS			MustL/MinSSp			MustL/MinSync			MustL/MinF		
Name	Aut	State space	Res	Peak States	Final States	Time [s]	Peak States	Final States	Time [s]	Peak States	Final States	Time [s]	Peak States	Final States	Time [s]
aip0aip	117	$1.02 \cdot 10^{9}$	yes	1090	5	**1.1**	892	5	1.1	**226**	66	1.7	1293	**4**	1.5
aip0alps	35	$3.00 \cdot 10^{8}$	no	18	16	**0.2**	18	16	0.2	**13**	**9**	0.4	41	**9**	0.3
aip0tough	60	$1.02 \cdot 10^{10}$	no	96049	17001582	53.5	96049	17001582	53.4	**177**	**0**	1.4	6017	**0**	2.5
aip1efa $\langle 3\rangle$	50	$6.88 \cdot 10^{8}$	yes	40290	1878708	11.4	40290	1878708	11.2	10734	2887502	11.3	79661	1157432	13.2
aip1efa $\langle 16\rangle$	50	$9.50 \cdot 10^{12}$	no	65520	13799628	20.7	65520	13799628	20.8	74100	20072368	34.3	39752	31369178	40.9
aip1efa $\langle 24\rangle$	50	$1.83 \cdot 10^{13}$	no	**6384**	13846773	17.2	**6384**	13846773	**17.1**	10734	20105881	28.5	88808	31369178	44.6
fencaiwon09	32	$1.03 \cdot 10^{8}$	yes	10421	105	2.3	10421	105	2.1	**279**	**41**	0.9	57186	126	3.5
fencaiwon09b	31	$8.93 \cdot 10^{7}$	no	10421	81	2.4	10421	81	2.5	**279**	68	**1.1**	57186	81	3.0
ftechnik	36	$1.21 \cdot 10^{8}$	no	172	**0**	**0.3**	172	**0**	0.4	**152**	**0**	1.1	576	**0**	0.4
profisafe_i4	82		yes	37044	4440	13.7	**7696**	**36**	**3.2**	37044	4440	16.8	Final state limit exceeded		
profisafe_i5	88		yes	98304	5414	52.3	**15780**	**140**	**5.1**	98304	5414	59.0	Final state limit exceeded		
profisafe_i6	94		yes	**52224**	300735	61.5	60200	**58**	**16.0**	**52224**	300735	73.3	Final state limit exceeded		
tbed_ctct	84	$3.94 \cdot 10^{13}$	no	43825	**0**	11.4	43825	**0**	11.3	**15039**	**0**	**6.3**	96464	4726	141.0
tbed_hisc0	196	$5.99 \cdot 10^{12}$	yes	1705	**33**	2.4	1705	**33**	**2.3**	**766**	50	3.5	54016	**33**	4.1
tbed_hisc1	184	$2.87 \cdot 10^{17}$	no	**19**	**0**	0.4	**19**	**0**	0.4	**19**	**0**	0.4	**19**	**0**	0.4
tbed_valid	84	$3.01 \cdot 10^{12}$	yes	33529	2508	6.8	33529	2508	6.7	**4640**	3019	**3.5**	54894	16	17.4
tip3	58	$2.27 \cdot 10^{11}$	yes	6351	153	3.2	22908	153	8.6	**192**	**0**	**1.1**	23664	1915400	13.8
tip3_bad	54	$5.25 \cdot 10^{10}$	no	992	**0**	0.9	992	**0**	0.9	**192**	**0**	1.1	2856	6	1.1
verriegel3	53	$9.68 \cdot 10^{8}$	yes	3171	**2**	1.5	2594	**2**	1.7	**636**	**2**	1.6	2781	42	1.5
verriegel3b	52	$1.32 \cdot 10^{9}$	no	**27**	**0**	0.3	**27**	**0**	**0.3**	**27**	**0**	0.6	28	**0**	0.3
verriegel4	65	$4.59 \cdot 10^{10}$	yes	2500	**2**	1.5	2594	**2**	1.6	636	**2**	1.7	**628**	**2**	**1.2**
verriegel4b	64	$6.26 \cdot 10^{10}$	no	**27**	**0**	0.4	**27**	**0**	**0.4**	**27**	**0**	0.7	28	**0**	0.4
6linka	53	$2.45 \cdot 10^{14}$	no	64	**0**	0.4	64	**0**	0.4	**61**	**0**	0.8	**61**	**0**	**0.2**
6linki	53	$2.75 \cdot 10^{14}$	no	61	**0**	0.3	61	**0**	0.3	**32**	**0**	0.5	56	**0**	0.2
6linkp	48	$4.43 \cdot 10^{14}$	no	32	**0**	0.3	32	**0**	0.3	**16**	**0**	0.4	32	**0**	**0.2**
6linkre	59	$6.21 \cdot 10^{13}$	no	**29**	**0**	0.3	**29**	**0**	0.3	**29**	**0**	0.8	58	**0**	0.3

Model				MustSp/MinS			MustSp/MinSSp			MustSp/MinSync			MustSp/MinF		
Name	Aut	State space	Res	Peak States	Final States	Time [s]	Peak States	Final States	Time [s]	Peak States	Final States	Time [s]	Peak States	Final States	Time [s]
aip0aip	117	$1.02 \cdot 10^{9}$	yes	327	5	1.3	328	10	1.3	289	66	2.2	635	**4**	1.5
aip0alps	35	$3.00 \cdot 10^{8}$	no	32	28	0.3	32	28	0.3	**13**	**9**	0.4	41	**9**	0.3
aip0tough	60	$1.02 \cdot 10^{10}$	no	4260	**0**	2.1	4260	**0**	2.1	235	**0**	**1.3**	6017	**0**	2.6
aip1efa $\langle 3\rangle$	50	$6.88 \cdot 10^{8}$	yes	90752	**1079062**	21.7	90752	**1079062**	22.0	**8800**	1238614	**6.7**	11740	2271322	10.9
aip1efa $\langle 16\rangle$	50	$9.50 \cdot 10^{12}$	no	90752	10334085	34.0	90752	10334085	33.9	**31016**	**8091884**	**17.3**	40193	31369178	42.0
aip1efa $\langle 24\rangle$	50	$1.83 \cdot 10^{13}$	no	90752	10362181	30.3	90752	10362181	30.5	31016	**8091884**	17.8	89473	31369178	46.5
fencaiwon09	32	$1.03 \cdot 10^{8}$	yes	72407	92	9.5	3538	105	1.4	**279**	**41**	**0.8**	6246	126	1.6
fencaiwon09b	31	$8.93 \cdot 10^{7}$	no	50529	**48**	5.7	50529	**48**	5.6	**279**	68	1.2	6246	81	1.4
ftechnik	36	$1.21 \cdot 10^{8}$	no	164	**0**	0.4	164	**0**	0.4	165	**0**	1.3	576	**0**	0.6
profisafe_i4	82		yes	Final state limit exceeded			Final state limit exceeded			Final state limit exceeded			Final state limit exceeded		
profisafe_i5	88		yes	Final state limit exceeded			Final state limit exceeded			Final state limit exceeded			Final state limit exceeded		
profisafe_i6	94		yes	Final state limit exceeded			Final state limit exceeded			Final state limit exceeded			Final state limit exceeded		
tbed_ctct	84	$3.94 \cdot 10^{13}$	no	89489	**0**	21.7	89489	**0**	22.0	35356	**0**	20.1	89761	5863	504.5
tbed_hisc0	196	$5.99 \cdot 10^{12}$	yes	2719	**33**	2.5	2719	**33**	2.5	817	**33**	3.8	7285	66	3.2
tbed_hisc1	184	$2.87 \cdot 10^{17}$	no	**19**	**0**	**0.4**	**19**	**0**	0.4	**19**	**0**	0.4	**19**	**0**	0.4
tbed_valid	84	$3.01 \cdot 10^{12}$	yes	33529	2508	6.6	33529	2508	6.8	**4640**	3019	3.6	54648	**5**	11.3
tip3	58	$2.27 \cdot 10^{11}$	yes	1216	1964	1.5	1216	1964	1.5	561	1265	1.3	71904	4441616	33.1
tip3_bad	54	$5.25 \cdot 10^{10}$	no	936	145	1.1	936	145	1.2	561	145	1.3	729	340	**0.7**
verriegel3	53	$9.68 \cdot 10^{8}$	yes	1587	**2**	1.5	802	**2**	1.4	**636**	**2**	1.9	2118	**2**	**1.3**
verriegel3b	52	$1.32 \cdot 10^{9}$	no	**27**	**0**	0.4	**27**	**0**	0.3	**27**	**0**	0.7	28	**0**	0.5
verriegel4	65	$4.59 \cdot 10^{10}$	yes	2185	**2**	2.1	802	**2**	1.4	636	**2**	1.8	1046	5500	1.3
verriegel4b	64	$6.26 \cdot 10^{10}$	no	**27**	**0**	0.4	**27**	**0**	0.4	**27**	**0**	1.0	28	**0**	0.5
6linka	53	$2.45 \cdot 10^{14}$	no	64	**0**	0.4	64	**0**	0.4	**61**	**0**	0.8	**61**	**0**	0.2
6linki	53	$2.75 \cdot 10^{14}$	no	61	**0**	0.3	61	**0**	0.3	**32**	**0**	0.6	56	**0**	**0.2**
6linkp	48	$4.43 \cdot 10^{14}$	no	32	**0**	0.3	32	**0**	0.3	**16**	**0**	0.4	32	**0**	0.2
6linkre	59	$6.21 \cdot 10^{13}$	no	**29**	**0**	0.3	**29**	**0**	0.3	**29**	**0**	0.7	58	**0**	**0.3**

composition after abstraction is 10^8 states. If this limit is exceeded, the run is aborted. The state limits reflect the available computing resources and keep the runtime of the experiments manageable. Attempts to obtain additional results with larger state limits have been unsuccessful.

Table 1 and Figure 11 show the results of a series of tests with different types of special events being used. For this experiment, the algorithm is run without any special events, with only blocked events, failing events, selfloop-only events, or always enabled events, with the combination of blocked and failing events, the combination of selfloop-only and always enabled events, and with all four types of special events. The candidate selection strategy is **MustL** followed **MinS** throughout this experiment. Figure 11 shows the final state numbers and runtimes for a selection of the experiments in Table 1.

The data shows that the use of special events reduces the peak or final state numbers in several cases. They help to decrease runtimes notably in the **aip0tough**, **profisafe**, and **6linkre** examples, while in other cases the slightly smaller state numbers are outweighed by the effort to find the special events and the more complicated simplification rules. Blocked and failing events are rare and therefore seem to have only a small impact. Yet, they can help to speed up termination when blocking is detected, as evidenced by the **tbed_hisc1**, **verriegel3b**, and **verriegel4b** examples.

There are also cases such as **verriegel3** and **verriegel4** where the peak state numbers increase with special events. A decrease in the final state number after simplification can come at the expense of an 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 weak observation equivalence from becoming applicable in later steps.

Table 2 and Figure 12 show the experimental results for different combinations of the candidate preselection and selection strategies proposed in Section 5.4. In this experiment, all types of special events are enabled. It becomes clear that the candidate preselection and selection strategies have a huge impact on the performance of compositional nonblocking verification. There is no clearly best approach, as every strategy ends up with the smallest peak or final states numbers for at least one example. However, only the combinations of **MustL** with **MinS**, **MinSSp**, and **MinSync** can solve all the problems. The **MinSync** selection strategy often has small peak state numbers, but it seems to get less accurate as the number of local events increases in later steps of the algorithm, resulting in larger final state numbers for the **aip1efa** $\langle 3 \rangle$ and **profisafe** examples. The combination of **MustL** and **MinSSp** solves the large **profisafe** problems [37] faster than all other methods, while not being much slower for most of the other problems. Maybe the **MinSSp** selection strategy gives a reasonable weight to special events, while the **MustSp** preselection strategy values them too high in relation to local events.

Figure 13 shows a breakdown of the performance of the individual abstraction rules. The piecharts show, for each abstraction rule, the total number of states removed and the total runtime over all experiments with the **MustL/MinS** candidate selection strategy and with all types of special events enabled. This data suggests that most states are removed by weak observation equivalence abstraction, which also contributes to most of the runtime. A few states become unreachable and are removed after processing of blocked and failing events, while the τ-Loop Removal Rule, the Only Silent Incoming Rule, the Only Silent Outgoing Rule, and particularly the Limited Certain Conflicts Rule remove a substantial number of states. These rules are simpler and run faster than weak observation equivalence, and their ability to reduce the automata before starting the high-complexity partitioning rules can significantly reduce runtime. The other two partitioning rules, namely Incoming Equivalence, which is the combination of the Enabled Continuation and Active Events Rules, and Reverse Observation Equivalence take less time and remove fewer states than weak observation equivalence, which probably is due to the fact that they are called on the smaller automata resulting from weak observation equivalence.

Overall, the experiments show that compositional verification can check the nonblocking property of systems with up to 10^{17} states in a matter of seconds. The size of the final synchronous composition does not increase significantly by the use of special events in the experiments, while it decreases in several cases. This is encouraging because the size of the final synchronous composition is the main limiting factor in compositional verification. Moreover, the large **profisafe** problems [37] can only be verified compositionally with selfloop-only events. By adding special events to the available tools, it becomes possible to solve problems that are not solvable otherwise.

7. Conclusions

It has been shown how compositional nonblocking verification of discrete event systems can be enhanced by taking into account additional information about the context in which an automaton to be abstracted is used. *Always enabled* events, *selfloop-only* events, *failing* events, and *blocked* events are easy to discover and help to produce smaller abstractions. Experimental results demonstrate that these special events can make it possible to verify the nonblocking property of more complex models.

In future work, it is of interest how special events can be used in other applications. It is likely that similar results can be obtained for *compositional synthesis* [42]. Furthermore, always enabled and selfloop-only events naturally arise from particular guard and update formulas of *extended finite-state machines* [31, 43], so it is likely that they can help with the analysis of such systems as well.

References

[1] C. G. Cassandras, S. Lafortune, Introduction to Discrete Event Systems, 2nd Edition, Springer Science & Business Media, New York, NY, USA, 2008.

[2] P. J. G. Ramadge, W. M. Wonham, The control of discrete event systems, Proceedings of the IEEE 77 (1) (1989) 81–98. doi:10.1109/5.21072.

[3] C. Baier, J.-P. Katoen, Principles of Model Checking, MIT Press, 2008.

[4] E. M. Clarke, D. E. Long, K. L. McMillan, Compositional model checking, in: Proceedings of the 5th IEEE Symposium on Logic in Computer Science, 1989, pp. 353–362.

[5] S. Graf, B. Steffen, Compositional minimization of finite state systems, in: Proceedings of the 1990 Workshop on Computer-Aided Verification, Vol. 531 of LNCS, Springer, 1990, pp. 186–196. doi:10.1007/BFb0023732.

[6] A. Valmari, Compositionality in state space verification methods, in: Proceedings of the 18th International Conference on Application and Theory of Petri Nets, Vol. 1091 of LNCS, Springer, 1996, pp. 29–56.

[7] D. Dams, O. Grumberg, R. Gerth, Abstract interpretation of reactive systems: Abstractions preserving $\forall$CTL*, $\exists$CTL* and CTL*, in: E.-R. Olderog (Ed.), Proceedings of IFIP WG2.1/WG2.2/WG2.3 Working Conference on Programming Concepts, Methods and Calculi (PROCOMET), IFIP Transactions, Elsevier Science Publisher (North-Holland), Amsterdam, The Netherlands, 1994, pp. 573–592.

[8] H. Flordal, R. Malik, Compositional verification in supervisory control, SIAM Journal of Control and Optimization 48 (3) (2009) 1914–1938. doi:10.1137/070695526.

[9] R. Malik, D. Streader, S. Reeves, Conflicts and fair testing, International Journal of Foundations of Computer Science 17 (4) (2006) 797–813. doi:10.1142/S012905410600411X.

[10] P. N. Pena, J. E. R. Cury, S. Lafortune, Verification of nonconflict of supervisors using abstractions, IEEE Transactions on Automatic Control 54 (12) (2009) 2803–2815.

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

[12] R. Malik, R. Leduc, Compositional nonblocking verification using generalised nonblocking abstractions, IEEE Transactions on Automatic Control 58 (8) (2013) 1–13. doi:10.1109/TAC.2013.2248255.

[13] S. Ware, R. Malik, Conflict-preserving abstraction of discrete event systems using annotated automata, Discrete Event Dynamic Systems: Theory and Applications 22 (4) (2012) 451–477. doi:10.1007/s10626-012-0133-3.

[14] S. Ware, R. Malik, Compositional verification of the generalized nonblocking property using abstraction and canonical automata, International Journal of Foundations of Computer Science 24 (8) (2013) 1183–1208. doi:10.1142/S0129054113500287.

[15] C. Pilbrow, R. Malik, Compositional nonblocking verification with always enabled events and selfloop-only events, in: Proceedings of the 2nd International Workshop on Formal Techniques for Safety-Critical Systems, FTSCS 2013, 2013, pp. 147–162.

[16] C. A. R. Hoare, Communicating Sequential Processes, Prentice-Hall, 1985.

[17] R. De Nicola, M. C. B. Hennessy, Testing equivalences for processes, Theoretical Computer Science 34 (1–2) (1984) 83–133. doi:10.1016/0304-3975(84)90113-0.

[18] R. Tarjan, Depth first search and linear graph algorithms, SIAM Journal of Computing 1 (2) (1972) 146–160.

[19] P. Gohari, W. M. Wonham, On the complexity of supervisory control design in the RW framework, IEEE Transactions on Systems, Man, and Cybernetics 30 (5) (2000) 643–652. doi:10.1109/3477.875441.

[20] R. Milner, Communication and concurrency, Series in Computer Science, Prentice-Hall, 1989.

[21] J.-C. Fernandez, An implementation of an efficient algorithm for bisimulation equivalence, Science of Computer Programming 13 (1990) 219–236.

[22] R. Malik, The language of certain conflicts of a nondeterministic process, Working Paper 05/2010, Department of Computer Science, University of Waikato, Hamilton, New Zealand (2010).
URL `http://hdl.handle.net/10289/4108`

[23] E. Nuutila, Efficient Transitive Closure Computation in Large Digraphs, Vol. 74 of Acta Polytechnica Scandinavica, Mathematics and Computing in Engineering Series, Finnish Academy of Technology, Helsinki, Finland, 1995.

[24] B. A. Brandin, R. Malik, P. Malik, Incremental verification and synthesis of discrete-event systems guided by counter-examples, IEEE Transactions on Control Systems Technology 12 (3) (2004) 387–401. doi:10.1109/TCST.2004.824795.

[25] J. Eloranta, Minimizing the number of transitions with respect to observation equivalence, BIT 31 (4) (1991) 397–419.

[26] T. Bolognesi, S. A. Smolka, Fundamental results for the verification of observational equivalence: a survey, in: H. Rudin, C. H. West (Eds.), Protocol Specification, Testing and Verification VII: Proceedings of IFIP WG6.1 7th International Conference on Protocol Specification, Testing and Verification, North Holland, 1987, pp. 165–179.

[27] Y. Wen, J. Wang, Z. Qi, Reverse observation equivalence between labelled state transition systems, in: Proceedings of the 1st International Colloquium on Theoretical Aspects of Computing, ICTAC '04, 2004, pp. 204–219.

[28] K. Åkesson, M. Fabian, H. Flordal, R. Malik, Supremica—an integrated environment for verification, synthesis and simulation of discrete event systems, in: Proceedings of the 8th International Workshop on Discrete Event Systems, WODES'06, IEEE, 2006, pp. 384–385.

[29] Supremica, the official website for the Supremica project. [link].
URL `http://www.supremica.org`

[30] R. Francis, An implementation of a compositional approach for verifying generalised nonblocking, Working Paper 04/2011, Department of Computer Science, University of Waikato, Hamilton, New Zealand (2011).
URL `http://hdl.handle.net/10289/5312`

[31] S. Mohajerani, R. Malik, M. Fabian, An algorithm for compositional nonblocking verification of extended finite-state machines, in: Proceedings of the 12th International Workshop on Discrete Event Systems, WODES'14, 2014, pp. 376–382.

[32] B. Brandin, F. Charbonnier, The supervisory control of the automated manufacturing system of the AIP, in: Proceedings of Rensselaer's 4th International Conference on Computer Integrated Manufacturing and Automation Technology, IEEE Computer Society Press, 1994, pp. 319–324.

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

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

[35] L. Feng, K. Cai, W. M. Wonham, A structural approach to the non-blocking supervisory control of discrete-event systems, International Journal of Advanced Manufacturing Technology 41 (2009) 1152–1168. doi:10.1007/s00170-008-1555-9.

[36] A. Lötzbeyer, R. Mühlfeld, 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`

[37] R. Malik, R. Mühlfeld, A case study in verification of UML statecharts: the PROFIsafe protocol, Journal of Universal Computer Science 9 (2) (2003) 138–151.

[38] R. J. Leduc, PLC implementation of a DES supervisor for a manufacturing testbed: An implementation perspective, Master's thesis, Department of Electrical Engineering, University of Toronto, ON, Canada (1996).
URL `http://www.cas.mcmaster.ca/~leduc`

[39] A. Hinze, P. Malik, R. Malik, Interaction design for a mobile context-aware system using discrete event modelling, in: Proceedings of the 29th Australasian Computer Science Conference, ACSC '06, Australian Computer Society, 2006, pp. 257–266.

[40] KORSYS Project. [link].
URL `http://www4.in.tum.de/proj/korsys/`

[41] J. Yi, S. Ding, M. T. Zhang, P. van der Meulen, Throughput analysis of linear cluster tools, in: Proceedings of the 3rd International Conference on Automation Science and Engineering, CASE 2007, 2007, pp. 1063–1068.

[42] S. Mohajerani, R. Malik, M. Fabian, A framework for compositional synthesis of modular nonblocking supervisors, IEEE Transactions on Automatic Control 59 (1) (2014) 150–162. doi:10.1109/TAC.2013.2283109.

[43] K. T. Cheng, A. S. Krishnakumar, Automatic functional test generation using the extended finite state machine model, in: Proceedings of the 30th ACM/IEEE Design Automation Conference, 1993, pp. 86–91. doi:10.1145/157485.164585.