Working Paper Series
ISSN 1177-777X

COMPOSITIONAL NONBLOCKING VERIFICATION WITH ALWAYS ENABLED AND SELFLOOP-ONLY EVENTS

Colin Pilbrow

Working Paper: 07/2013
November 19, 2013

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

COMPOSITIONAL NONBLOCKING VERIFICATION WITH ALWAYS ENABLED AND SELFLOOP-ONLY EVENTS

Colin Pilbrow
Department of Computer Science
The University of Waikato
Hamilton, New Zealand
colinpilbrow@gmail.com

November 19, 2013

Abstract

This report proposes to improve compositional nonblocking verification through the use of two special event types: always enabled and selfloop-only events. Compositional verification involves abstraction to simplify parts of a system during verification. Normally, this abstraction is based on the set of events not used in the remainder of the system. Here, it is proposed to exploit more knowledge about the system and abstract events even though they are used in the remainder of the system. This can lead to more simplification than was previously possible. Abstraction rules from previous work are extended to respect the new special events and proofs show these rules still preserve nonblocking. The rules have been implemented in Waters and experimental results demonstrate that these extended simplification rules help verify several industrial-scale discrete event system models while achieving better state-space reduction than before.

Contents

1 Introduction **4**

2 Preliminaries **6**
2.1 Events and Languages 6
2.2 Nondeterministic Automata 6
2.3 The Nonblocking Property 7
2.4 Synchronous Composition 8
2.5 Compositional Verification 10
2.6 Automaton Abstraction 11
2.7 Introducing Special Events 13

3 Simplification Rules **16**
3.1 Always Enabled Events 16
3.1.1 Silent Continuation 16
3.1.2 Only Silent Incoming Rule 17
3.1.3 Limited Certain Conflicts Rule 20
3.2 Other Selfloop-Only events 25

4 Finding Always Enabled and Selfloop-only Events **27**
4.1 Finding Additional Always Enabled Events 27
4.2 Conditionally Always Enabled Events 28

5 Implementation in Waters **32**

6 Experimental Results **35**

7 Conclusions **38**
7.1 Future Work 38

A Silent Continuation Proof **39**

B Limited Certain Conflict Proof **40**

C Selfloop Removal Lemma **42**

D Selfloop-Only Addition Proof **43**

E Conditionally Always Enabled Events Proof **44**

1 Introduction

When working with safety-critical systems, it is important to know that they behave as expected. Safety-critical systems include medical devices and factories where errors are expensive or even deadly. These systems can also be large or complex, making it difficult to determine that they will behave as expected in all situations and be safe for the users. Model Checking is used to prove that a system satisfies certain properties such as controllability and nonblocking. This lets us be more confident that the system is safe. The system is modelled as a set of finite state automata, where each automaton is used to describe different parts of the system. Nonblocking can show that something good will eventually happen. Depending on how the system has been modelled, this can show the system always being able to reach a safe idle state, or coming to completion. That is, it shows the absence of livelocks or deadlocks in the system that would prevent it from reaching a desired state. [6, 21] These desired states are marked during modelling. This is an important property to verify, however, as the models are getting larger and more complex to accurately match the real-world systems, the standard methods for checking nonblocking are not sufficient.

The standard method to check whether a system is nonblocking involves the explicit composition of all the automata involved, and then performing a backtracking search from all marked states to ensure that every state can reach a marked state. Unfortunately the standard method is limited by the *state-space explosion* problem. This is because composing together automata increases the state-space exponentially, and quickly leads to running out of memory. Different methods have been created to help avoid this problem. *Symbolic model checking* has been used successfully to reduce the amount of memory required by representing the state space symbolically rather than enumerating it explicitly [2]. *Compositional verification* [10, 16] is an effective alternative that can be used independently of, or in combination with, symbolic methods. Compositional verification works by simplifying individual automata before each composition, gradually reducing the state space of the system and allowing much larger systems to be verified in the end. Since the state-space increases exponentially when composing, if even a small number of states can be simplified at the start then this can lead to large reductions in the number of states in the final composition. However, when applied to the nonblocking property, finding simplification rules is difficult as it requires very specific abstraction methods. These abstractions must preserve *conflict equivalence* [17]. When simplifying the automata it is important to ensure this does not change the nonblocking property of the system. Various abstraction rules preserving conflict equivalence have been proposed and implemented [10, 19, 22, 23]. These include rules such as Observational Equivalence, Tau-Loop Removal and Certain Conflicts. However, these abstraction rules do not take advantage of the whole system. It will be shown that by creating abstraction rules that use this extra information more simplification becomes possible. Normally, the main way these rules simplify automata is to use special τ events, that are only present on the automaton being simplified. However, there are other events with simplification properties that can be found by looking at the automata that are not being simplified.

This report proposes simplification rules that take into account that certain events are always

enabled or are only selfloops in the automata not being simplified, and shows proofs that the nonblocking property is preserved. These rules have also been implemented to show that this additional information can achieve further state-space reduction.

Part of this report will be published in FTSCS [20], which introduces the abstraction rules and shows experimental results. In addition, this report also includes examples, proofs, and describes how the rules have been implemented.

In the following, section 2 introduces the background of nondeterministic automata, the nonblocking property, conflict equivalence and compositional verification. Sect. 3 presents always enabled, selfloop-only events and the simplification rules that exploit such events, and section 4 shows how these events are found algorithmically. How the simplification rules were implemented is discussed in section 5. Afterwards, section 6 presents the experimental results, and section 7 adds concluding remarks. The appendix contains the proofs for each of the simplification rules.

2 Preliminaries

There are a set of existing basic definitions that will be used in this report. Many of the standard definitions are from [3, 7], while other definitions are taken from various sources and are mentioned individually. This section shall show all the existing definitions used, and the new definitions that have been created for this report are found in section 3.

2.1 Events and Languages

Event sequences and languages are a simple means to describe how a system behaves [6, 21]. Their basic building blocks are *events*, which are taken from a finite *alphabet* $\mathbf{A}$. In addition, two special events are also 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 $\mathbf{A}_{\tau,\omega} = \mathbf{A} \cup \{\tau, \omega\}$.

The τ event is used to define an event present on only a single automaton. It has many properties that are useful for simplification rules. When moving through an automaton these events can be taken silently, without changing the state of any other automata in the system.

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

Each automaton in the system consists of a finite set of states and events. A transition relation is used to show the transitions between states. This contains the transition event and the before and after states. Automata also have initial states that show where the system starts, since these are nondeterministic, multiple initial states are possible. We also model our automata with marked states. These represent desired or safe states of the system. These are used for nonblocking, where we check that they can always be reached.

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

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

The termination event $\omega \notin \mathbf{A}$ denotes completion of a task and does not appear anywhere else but to mark such completions. It is required that states reached by ω do not have any outgoing transitions, i.e., if $x \xrightarrow{\omega} y$ then there does not exist $\sigma \in \mathbf{A}_{\tau,\omega}$ such that $y \xrightarrow{\sigma}$. This ensures that the termination event, if it occurs, is always the final event of any trace. The traditional set of marked states is $Q^{\omega} = \{\, x \in Q \mid x \xrightarrow{\omega} \}$ in this notation. The states in Q^{ω} are the marked states and are shown shaded in the figures of this report 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 $X \overset{s}{\Rightarrow} Y$, $x \Rightarrow y$, and $x \overset{s}{\Rightarrow}$ are defined analogously to $\rightarrow$.

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

Definition 2 Let $G = \langle \mathbf{A}, Q, \rightarrow, Q^{\circ} \rangle$ and $\Upsilon \subseteq \mathbf{A}$. The result of *hiding* Υ from G, written $G \setminus \Upsilon$, is the automaton obtained from G by replacing each transition $x \xrightarrow{\upsilon} y$ with $\upsilon \in \Upsilon$ by $x \xrightarrow{\tau} y$, and removing the events in Υ from $\mathbf{A}$.

2.3 The Nonblocking Property

The *nonblocking* property is an important property in model checking. An automaton is nonblocking if, from every reachable state, a marked state can be reached; otherwise it is *blocking*.

Definition 3 [17] 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 $G \,\|\, H$ is nonblocking.

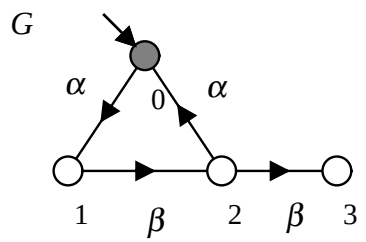

Figure 1: Example of a blocking automaton.

Example 1 The automaton in figure 1 is an example of a blocking automaton. The states of the automaton are represented as circles in the figures of this report. Transitions are shown as arrows

between states, and are labelled with the event of the transition. The shaded circles are marked states and the little arrow entering state 0 shows this is the initial state.

Since there is no sequence of transitions that allows state 3 to reach a marked state, this automaton is blocking.

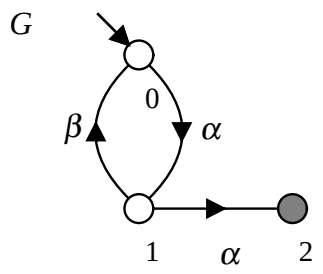

Figure 2: Example of a nonblocking automaton.

Example 2 The automaton in figure 2 is an example of a nonblocking automaton. Although it is possible for the system to cycle between states 0 and 1 infinitely, every state in the system can reach state 2, a marked state. This means the automaton is nonblocking.

2.4 Synchronous Composition

Definition 4 Synchronous composition is used to compose multiple automata together.

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

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

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

Example 3 Finding the synchronous composition of automata $G1$ and $G2$ in figure 3. Start by creating the initial state $(0,0)$ which is the initial states of both $G1$ and $G2$. This state has the

transitions which are enabled in both automata. $0 \xrightarrow{\alpha} 1$ is a transition in both $G1$ and $G2$, so the synchronous composition has a transition $(0,0) \xrightarrow{\alpha} (1,1)$. We then investigate the transitions that are enabled in this state. $1 \xrightarrow{\alpha} 2$ is enabled in $G1$, however α is disabled in state 1 of $G2$, so it is not in the synchronous composition. Event β is enabled in both automata however, so transition $(1,1) \xrightarrow{\beta} (0,2)$ is created in the synchronous composition. This method continues, for each new state (x,y) that is created in the synchronous composition we create new transitions for the events that are enabled in both states x in $G1$ and y in $G2$.

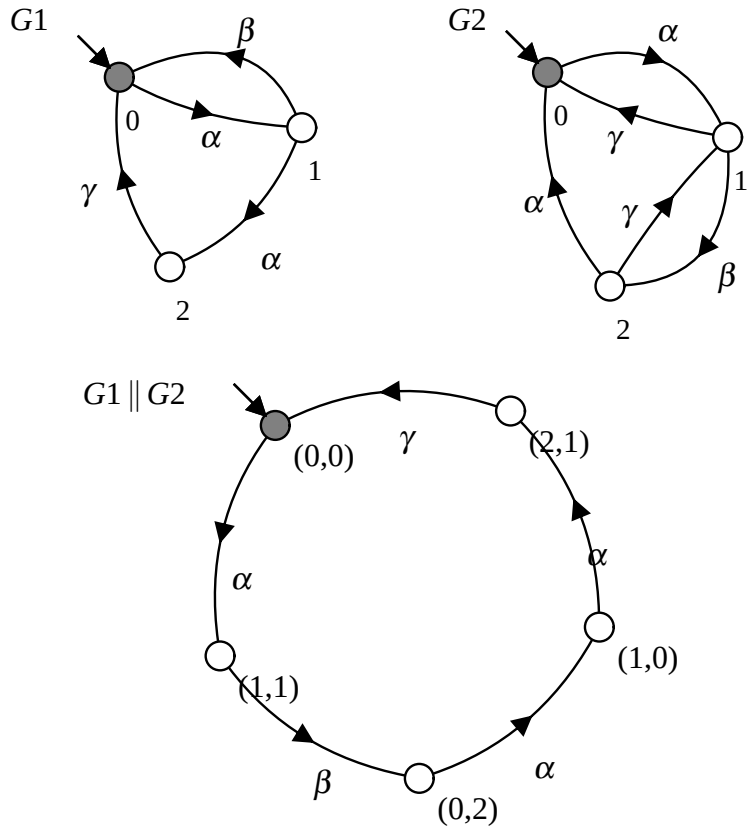

Figure 3: Synchronous Composition of Automata

To reason about conflicts in a compositional way, the notion of *conflict equivalence* is developed in [17]. According to process-algebraic testing theory, two automata are considered as equivalent if they both respond in the same way to tests [8]. 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 5 [17] Two automata G and H are *conflict equivalent*, written $G \simeq_{\text{conf}} H$, if, for any automaton T, $G \parallel T$ is nonblocking if and only if $H \parallel T$ is nonblocking.

Example 4 Figure 4 contains automata G and H which can be shown to be not conflict equivalent using test automaton T. Note that if states 1 and 2 in G are merged together we get automaton H, so this example shows that simplification of automata is not as easy as simply merging together any two states. T is an example of an automaton such that $G \parallel T$ and $H \parallel T$ do not have the same nonblocking property. $G \parallel T$ is blocking since it has no marked states, and every state in $H \parallel T$ can reach the marked state $(2,3)$, so it is nonblocking. Because of this $G \simeq_{\text{conf}} H$ is not true, G is not conflict equivalent to H.

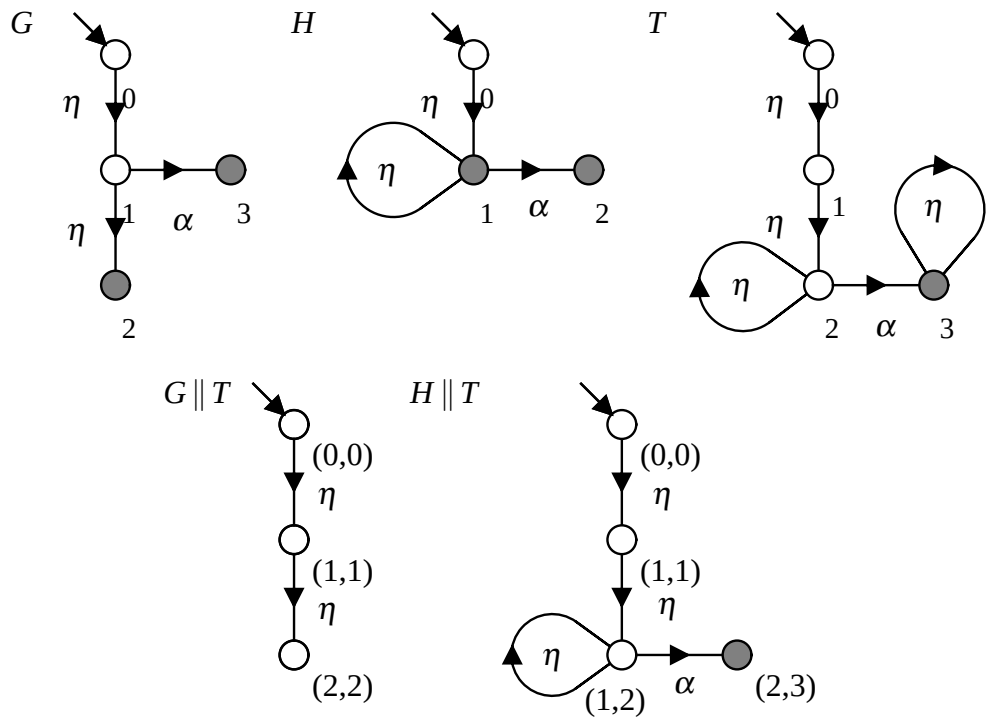

Figure 4: Example of automata G and H that are not conflict equivalent

Example 5 Figure 5 contains automata G and H which are not conflict equivalent. This figure shows the two outgoing α transitions being merged into a single transition. Note that $0 \xrightarrow{\alpha} 2 \xrightarrow{\beta}$ is not possible in G, but $0 \xrightarrow{\alpha\beta}$ is possible in H. This can be used to help find the test automaton T. $G \parallel T$ is blocking, since state $(2,1)$ cannot reach a marked state, while $H \parallel T$ is nonblocking, since every state can reach a marked state. This means that G is not conflict equivalent to H.

2.5 Compositional Verification

When verifying whether a composed system of automata

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

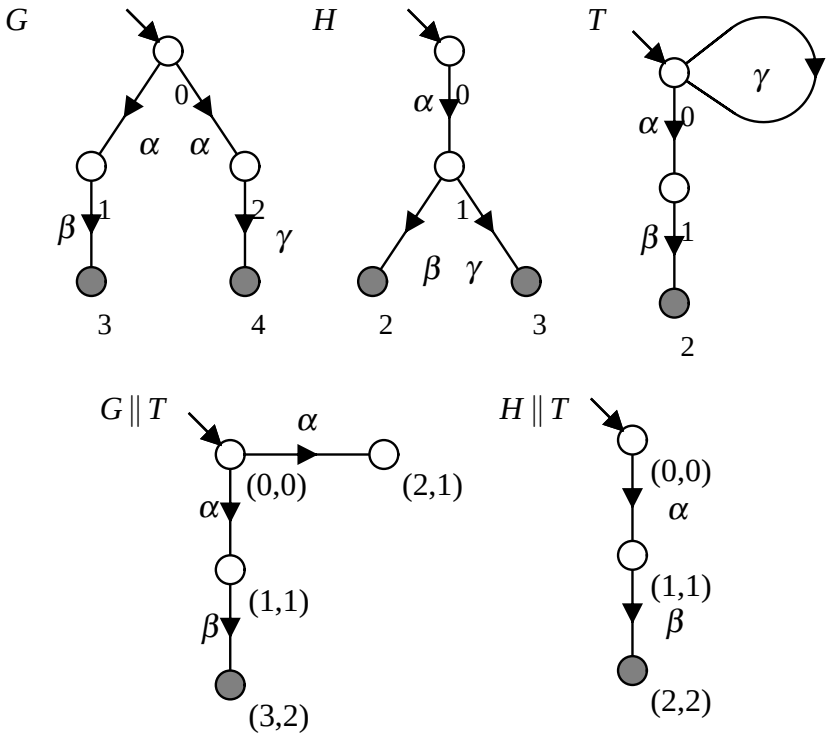

Figure 5: Example of automata G and H that are not conflict equivalent

is nonblocking, compositional methods [10, 16, 23] avoid building the full synchronous composition immediately. Instead, individual automata G_i are simplified and replaced by smaller conflict equivalent automata $G_i' \simeq_{\text{conf}} G_i$. If no simplification is possible, a subsystem of automata $(G_j)_{j \in J}$ is selected and replaced by its synchronous composition, which then may be simplified.

The soundness of this approach is justified by the *congruence* properties [17] of conflict equivalence. For example, if G_1 in (2) is replaced by $G_1' \simeq_{\text{conf}} G_1$, then by considering $T = G_2 \parallel \cdots \parallel G_n$ in definition 5, it follows that the abstracted system $G_1' \parallel T = G_1' \parallel G_2 \parallel \cdots \parallel G_n$ is nonblocking if and only if the original system (2) is.

2.6 Automaton Abstraction

A common method to simplify an automaton is to construct its *quotient* modulo an equivalence relation. Certain states are identified as equivalent and merged. 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 6 Let $G = \langle \mathbf{A}, Q, \rightarrow, Q^\circ \rangle$ be an automaton, and let $\sim \subseteq Q \times Q$ be an equivalence relation. The *quotient automaton* $G/{\sim}$ of G with respect to $\sim$ is $G/{\sim} = \langle \mathbf{A}, Q/{\sim}, \rightarrow/{\sim}, \tilde{Q}^\circ \rangle$, where $\tilde{Q}^\circ = \{\, [x^\circ] \mid x^\circ \in Q^\circ \,\}$ and $\rightarrow/{\sim} = \{\, ([x], \sigma, [y]) \mid x \xrightarrow{\sigma} y \,\}$.

The states of the quotient automaton are classes of equivalent states of the original automaton. A common equivalence relation to construct such a quotient automaton is *observation equivalence* or *weak bisimulation* [18].

Definition 7 [18] Let $G = \langle \mathbf{A}, Q, \rightarrow, Q^\circ \rangle$ be an automaton. A relation $\approx \subseteq Q \times Q$ is an *observation equivalence* relation on G if, for all states $x_1, x_2 \in Q$ such that $x_1 \approx x_2$ and all traces $s \in \mathbf{A}^*_\omega$ the following conditions hold:

1. if $x_1 \overset{s}{\Rightarrow} y_1$ for some $y_1 \in Q$, then there exists $y_2 \in Q$ such that $y_1 \approx y_2$ 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 $y_1 \approx y_2$ and $x_1 \overset{s}{\Rightarrow} y_1$.

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

Proposition 1 [16] Let G be an automaton, and let $\approx$ be an observation equivalence relation on G. Then $G \simeq_{\text{conf}} G/{\approx}$.

A special case of observation equivalence-based abstraction is *τ-loop removal*. If two states are mutually connected by sequences of τ-transitions, it follows from definition 7 that these states are observation equivalent, so by proposition 1 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 8 Let $G = \langle \mathbf{A}, Q, \rightarrow, Q^\circ \rangle$ be an automaton. G is *τ-loop free*, if for every path $x \xrightarrow{t} x$ with $t \in \{\tau\}^*$ it holds that $t = \varepsilon$.

While τ-loop removal and observation equivalence are easy to compute and produce good abstractions, it is known that there are conflict equivalent automata that are not observation equivalent. Several other relations are considered for conflict equivalence [10, 16].

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 [10]. Firstly, it follows immediately from definition 6 that every path between in an automaton also links the corresponding classes in its quotient automaton.

Lemma 2 [10] 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 an observation equivalence relation, and another possibility is *incoming equivalence* [10].

Definition 9 [10] 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 $\sigma \in \mathbf{A}$ 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 the exactly same source states. (This is different from reverse observation equivalence, which accepts *equivalent* rather than identical states.)

The additional requirement of incoming equivalence is enough to establish a converse of lemma 2 and makes it possible to lift paths in the quotient back to paths in the original automaton.

Lemma 3 [10] 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 $\sigma_i \in \mathbf{A}_\tau$ 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 $x_0 \overset{\sigma_1}{\Rightarrow} x_1 \overset{\sigma_2}{\Rightarrow} \cdots \overset{\sigma_n}{\Rightarrow} x_n$ is a path in G.
2. If $G/{\sim} \overset{s}{\Rightarrow} \tilde{x}$ for some $s \in \mathbf{A}^*$, then there exists $x \in \tilde{x}$ such that $G \overset{s}{\Rightarrow} x$.

2.7 Introducing Special Events

Previous approaches for compositional nonblocking verification [10,16,23] make no assumption about the remainder $T = G_2 \parallel \cdots \parallel G_n$ of the system apart from its event set. Typically, G_1 has some *local* events, i.e., events used only by G_1. The local events are abstracted using hiding, i.e., they are replaced by the silent event τ. Conflict equivalence uses the silent event τ as a placeholder for events not used elsewhere, and in this setting is the coarsest conflict-preserving abstraction.

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

In addition to the existing definitions, the following definitions have been created for *always enabled* events and *selfloop-only* events.

Definition 10 Let $G = \langle \mathbf{A}, Q, \rightarrow, Q^\circ \rangle$ be an automaton. An event $\sigma \in \mathbf{A}$ is *always enabled* in G, if for every state $x \in Q$ it holds that $x \overset{\sigma}{\Rightarrow}$.

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

Definition 11 Let $G = \langle \mathbf{A}, Q, \rightarrow, Q^\circ \rangle$ be an automaton. An event $\sigma \in \mathbf{A}$ is *selfloop-only* in G, if for every transition $x \xrightarrow{\sigma} y$ it holds that $x = y$.

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

Definition 12 Let G and H be two automata, and let $\mathbf{E}$ and $\mathbf{S}$ be two sets of events. G and H are *conflict equivalent* with respect to $\mathbf{E}$ and $\mathbf{S}$, written $G \simeq_{\mathbf{E},\mathbf{S}} H$, if for every automaton T such that $\mathbf{E}$ is a set of always enabled events in T and $\mathbf{S}$ is a set of selfloop-only in T, it holds that $G \parallel T$ is nonblocking if and only if $H \parallel T$ is nonblocking.

This definition only considers tests T where events in $\mathbf{E}$ are always enabled and events in $\mathbf{S}$ are selfloop-only. It is clear that standard conflict equivalence implies conflict equivalence with respect to $\mathbf{E}$ and $\mathbf{S}$, while the opposite is not always the case. The following result is immediate from the definition.

Proposition 4 Let G and H be two automata.

1. $G \simeq_{\text{conf}} H$ if and only if $G \simeq_{\emptyset,\emptyset} H$.
2. If $\mathbf{E} \subseteq \mathbf{E}'$ and $\mathbf{S} \subseteq \mathbf{S}'$ then $G \simeq_{\mathbf{E},\mathbf{S}} H$ implies $G \simeq_{\mathbf{E}',\mathbf{S}'} H$.

As conflict equivalence with respect to $\mathbf{E}$ and $\mathbf{S}$ considers less tests T than standard conflict equivalence, it is clear that it considers more automata as equivalent. The modified equivalence is coarser and has the potential to achieve better abstraction.

Example 6 Automata G and H in figure 6 are *not* conflict equivalent as demonstrated by the test automaton T. This is because $G \parallel T$ is blocking while $H \parallel T$ is not. $G \parallel T$ is blocking because the state $(1,0)$ is reachable by τ from the initial state $(0,0)$, and $(1,0)$ is a blocking 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 since both states can reach a marked state.

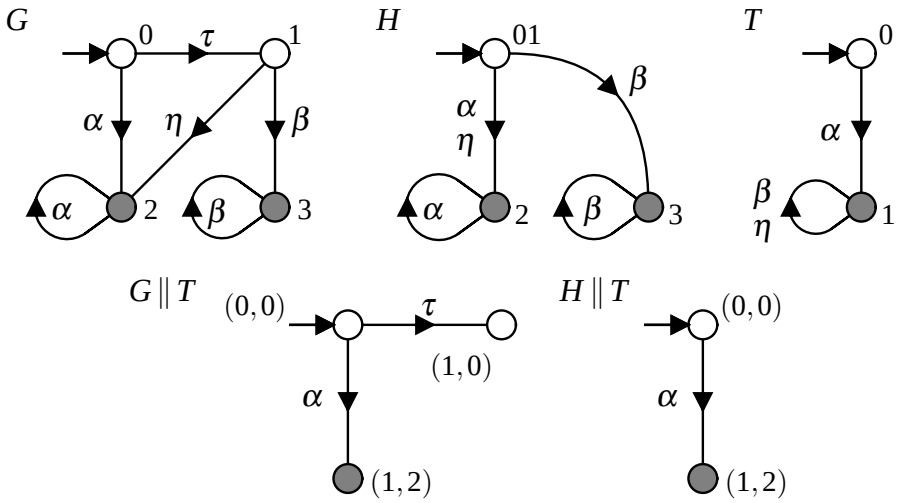

Figure 6: Two automata G and H such that $G \simeq_{\{\eta\},\emptyset} H$ but not $G \simeq_{\text{conf}} H$.

Note that η is not always enabled in T since $0 \overset{\eta}{\Rightarrow}_T$ does not hold. In composition with any test T that has η always enabled, G will be able to continue from state 1 using η, and H will be able to continue from state 01. It follows from proposition 5 below that $G \simeq_{\{\eta\},\emptyset} H$.

Example 7 $G \simeq_{\{\eta\},\emptyset} H$ is not true in Figure 4. This can be shown using T, because every state in T had an outgoing η transition, making η an always enabled event in T.

Based on example 6, if during compositional verification, G in figure 6 is one of the automata in the system (2), and it is known that η is an always enabled event in all automata except G, then G can be replaced by H to simplify the verification task.

3 Simplification Rules

In this section I shall discuss the new and extended rules that have been created with the special events I have found. Before any simplification rules are performed, we have already determined which events are special. How this is done is discussed in section 4. Extending the simplification rules lets them be applied in more places, which leads to more simplification of the automata. Although many existing simplification rules were investigated, only the following three were found to be able to be extended with always enabled events.

3.1 Always Enabled Events

3.1.1 Silent Continuation

Silent Continuation [10] is a rule used to simplify long chains of τ transitions into a single τ transition. This is because the automaton can move silently along the chain without changing the state of any other automata in the system. I found that changing the rule to include τ chains that end with an always enabled event lets this rule simplify more states, while still preserving nonblocking.

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

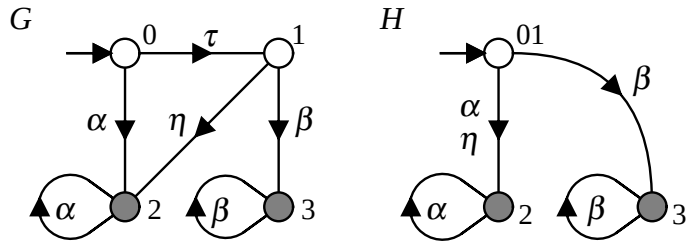

Figure 7: Silent Continuation Rule used to simplify automaton G to automaton H.

Example 8 Consider automaton G in figure 7 with $\mathbf{E} = \{\eta\}$. States 0 and 1 are both "initial" since they both can be reached silently from the initial state 0. This is enough to satisfy $\sim_{\text{inc}}$ in this case, since neither state is reachable by any event other than τ. Moreover, G has no τ-loops, state 0 has an outgoing τ-transition, and state 1 has an outgoing always enabled event η. Thus, by the Silent Continuation Rule, states 0 and 1 in G are conflict equivalent and can be merged into state 01 as shown in H.

Algorithm 1 Silent Continuation Implementation

```
if |{τ} ∪ E| = 0 then
    stop
end if
for all State s do
    if s -τ-> or s -η-> then
        CalculateIncomingEquivalence(s)
        add s to IncomingEquivalenceHashMap
    end if
end for
Merge states in IncomingEquivalenceHashMap which are incoming equivalent
RemoveUnreachableStates()
```

IncomingEquivalenceHashMap is used to store states, and groups incoming equivalent states together. *RemoveUnreachableStates*() removes the states in the automaton that cannot be reached by any sequence of transitions.

Proposition 5 Let $G = \langle \mathbf{A}, Q, \rightarrow_G, Q^\circ \rangle$ be a τ-loop free automaton, let $\mathbf{E} \subseteq \mathbf{A}$, and let $\sim \subseteq Q \times Q$ be an equivalence relation such that $\sim \subseteq \sim_{\text{inc}}$, and for all $x, y \in Q$ such that $x \sim y$ it holds that either $x = y$ or both x and y have an outgoing η-transition for some $\eta \in \mathbf{E} \cup \{\tau\}$. Then $G \simeq_{\mathbf{E},\emptyset} G/\sim$.
The proof is found in Appendix A.

Prop. 5 confirms that the nonblocking property of the system is preserved under the generalised silent continuation rule, provided that **E** is a set of always enabled events for the remainder of the system.

Algorithm 1 is an extension of the algorithm that already exists in Waters. When the simplification rule is run, the automaton is known to be τ-loop free and τ and **E** have already been found. To simplify the automaton we must find which states that have an outgoing τ or always enabled event are incoming equivalent. An existing algorithm can be used to find incoming equivalent states. These states can now be merged because of the Silent Continuation rule, making the automata smaller. After simplification, there may be unreachable states that can be removed to save memory.

3.1.2 Only Silent Incoming Rule

The Only Silent Incoming Rule [10] is a combination of observation equivalence and the Silent Continuation Rule. If a state has only incoming τ transitions we can split it into multiple states using observational equivalence. If this state had an outgoing τ transition, then we can now apply Silent Continuation. Since the Silent Continuation Rule has been generalised to use always enabled events, the Only Silent Incoming Rule can be as well.

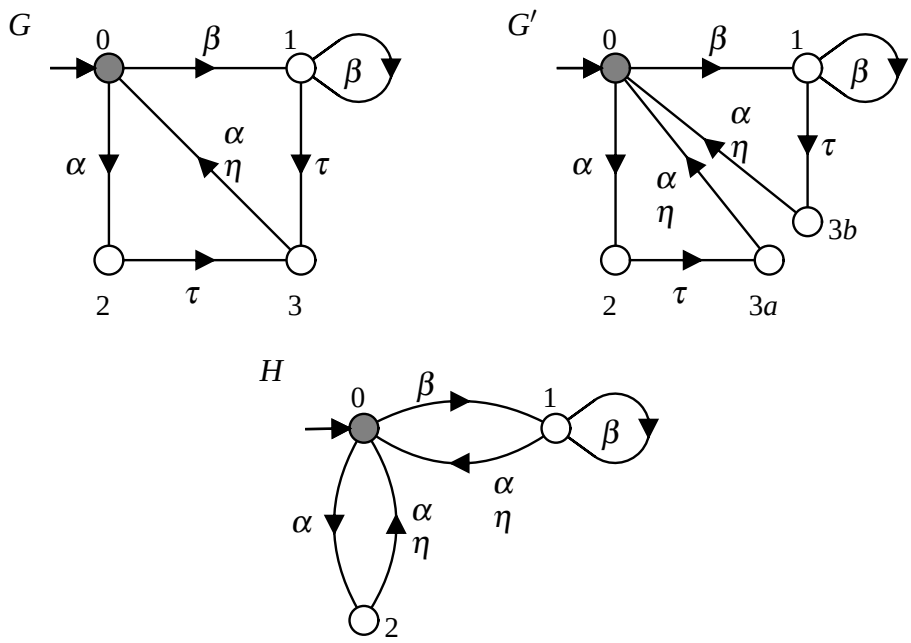

Figure 8: Example of application of the Only Silent Incoming Rule.

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

Example 9

In figure 8 it holds that $G \simeq_{\{\eta\},\emptyset} H$. State 3 in G has only τ-transitions incoming and the always enabled event η outgoing. This state can be removed in two steps. First, state 3 is split into two observation equivalent states $3a$ and $3b$ in G', and afterwards the Silent Continuation Rule is applied to merge these states into their incoming equivalent predecessors, resulting in H.

Proposition 6 Let $G = \langle \mathbf{A}, Q, \rightarrow_G, Q^\circ \rangle$ be a τ-loop free automaton, and let $\mathbf{E} \subseteq \mathbf{A}$. Let $q \in Q$ such that $q \xrightarrow{\eta}_G$ for some $\eta \in \mathbf{E} \cup \{\tau\}$, and for each transition $x \xrightarrow{\sigma}_G q$ it holds that $\sigma = \tau$. Further, let $H = \langle \mathbf{A}, Q, \rightarrow_H, Q^\circ \rangle$ with

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

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

It is shown in [10] that the Only Silent Incoming Rule can be expressed as a combination of observation equivalence and the Silent Continuation Rule as suggested in example 9. The same argument can be used to prove proposition 6.

Algorithm 2 Only Silent Incoming Implementation

```
for all State x do
    if there does not exist x -σ-> with σ = {τ} ∪ E then
        add x to keepSet
    end if
    for all Transition x -σ-> y with σ ≠ τ do
        add y to keepSet
    end for
end for
if |keepSet| = |States| then
    stop
else
    for all State source do
        for all Transition source =τ=> y do
            if y ∉ keepSet then
                add y to targetSet
            end if
        end for
        for all target ∈ targetSet do
            for all Transition target -σ-> z do
                create Transition source -σ-> z
            end for
            delete Transition source -τ-> target
        end for
    end for
    RemoveUnreachableStates()
end if
```

The Only Silent Incoming Rule removes states, however it often increases the number of transitions. Yet, it usually improves the structure of the automaton such that it allows other rules to be applied.

Algorithm 2 is a simplified version of the implemented algorithm, which also handles generalised nonblocking [15] and the Silent Incoming rule, which does not require every incoming transition to be τ.

Although the Only Silent Incoming rule has been shown as the application of Observational Equivalence followed by Silent Continuation, this algorithm is a shortcut. We do not need to split any states into two as shown in example 9, as that is expensive and unnecessary.

This algorithm first finds the states that will be kept these are the states that cannot be removed using Only Silent Incoming. However, any states which are not found to be kept are those which satisfy the requirements of Only Silent Incoming and may be removed. The source states where

the outgoing transitions of the removed state will be copied to are then found. After moving the outgoing transitions, the incoming τ transition from the source state to the target state is removed. When all the incoming τ transitions are removed the state will become unreachable, and it is removed in the final step.

Figure 9 shows algorithm 2 being applied. State 3 is found to be the only state that is not kept. We then loop over each of the states. We set state 1 to be the source state, since it is the first state with an outgoing τ to state 3. In $G2$ we have copied each of the outgoing transitions from state 3 to state 1. The τ transition between them is also removed. We then choose state 2 to be the next source state. In $G3$ the outgoing transitions are copied from state 3 to state 2, and the τ transition removed. After looping through each of the states we can now remove unreachable states. Since state 3 has no incoming transitions, it is unreachable, and so can be removed resulting in $G4$.

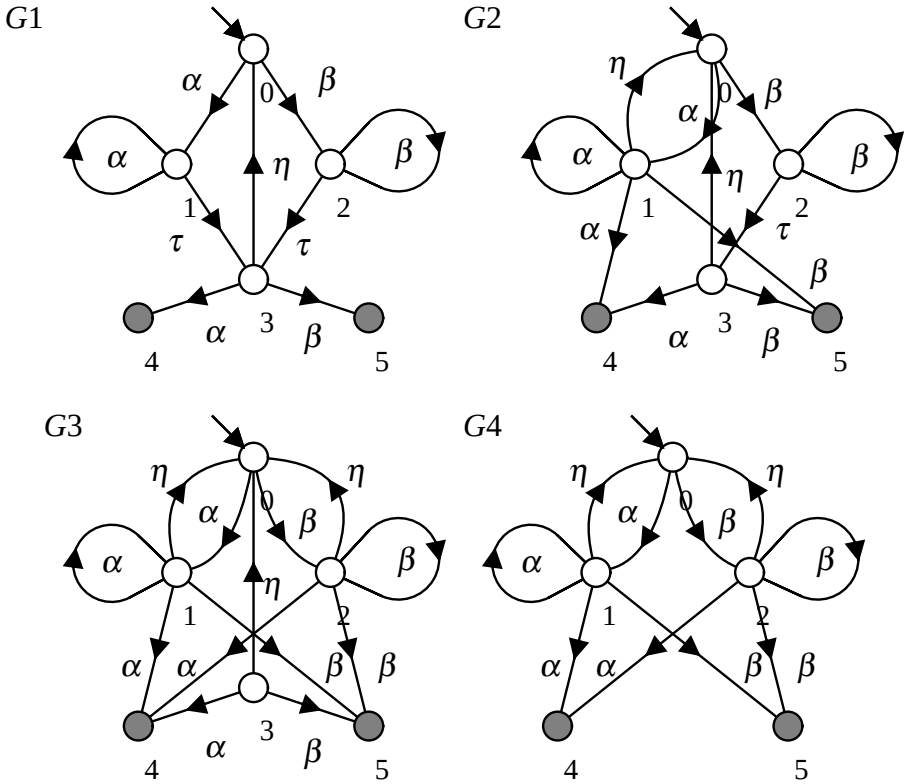

Figure 9: Example of implementation of the Only Silent Incoming Rule.

3.1.3 Limited Certain Conflicts Rule

Some automata contain blocking states, i.e., states from where it is not possible to reach any state with an ω-transition. If one automaton in a synchronous composition enters a blocking state, then the composition is blocking. We can also look at the states with transitions entering

a blocking state. If we know these transitions are enabled, then the state will be able to enter the blocking state. If a state has an always enabled transition to a blocking state then it is also a blocking state, as it is always possible for it to reach a state where it cannot reach any state with an ω-transition. We can use this to find more blocking states. When a blocking state is found, all it's outgoing transitions are removed and it is merged into the other blocking states.

Every automaton is associated with a language of *certain conflicts* [13], which characterises exactly the traces that cause blocking in every possible context. It is possible to calculate all states of certain conflicts and construct an abstraction that replaces all certain conflicts by a single state. Unfortunately, the algorithm to do this is exponential in the number of states of the automaton to be simplified [14].

To reduce the complexity, the Limited Certain Conflicts Rule [10] approximates the set of certain conflicts. If a state has a τ-transition to a blocking state, then the source state also is a state of certain conflicts. This can be extended to include always enabled events, because if an always enabled transition takes an automaton to a blocking state, then nothing can disable this transition and the system is necessarily blocking.

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

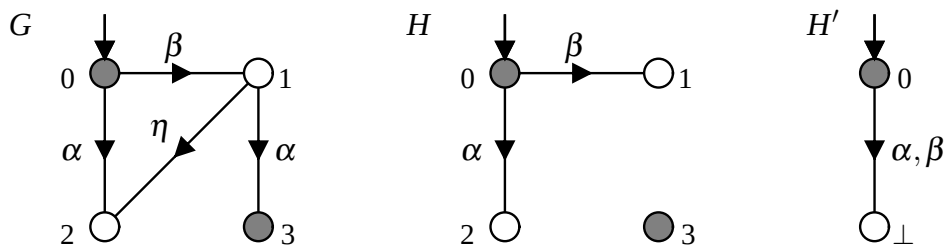

Figure 10: Example of application of the Limited Certain Conflicts Rule.

Example 10 Consider automaton G in figure 10 with $\mathbf{E} = \{\eta\}$. State 2 is already blocking, and states 1 has an always enabled η-transition to the blocking state 2. All transitions from this state are removed. This results in automaton H. Now state 3 is unreachable and can be removed, and states 1 and 2 can be merged using observation equivalence to create H'. It holds that $G \simeq_{\emptyset,\{\eta\}} H \simeq_{\text{conf}} H'$.

Proposition 7 Let $G = \langle \mathbf{A}, Q, \rightarrow_G, Q^\circ \rangle$ be an automaton and $\mathbf{E} \subseteq \mathbf{A}$, let $q \in Q$ be a blocking state, and let $p \xrightarrow{\eta} q$ for some $\eta \in \mathbf{E} \cup \{\tau\}$. Furthermore, let $H = \langle \mathbf{A}, Q, \rightarrow_H, Q^\circ \rangle$ where $\rightarrow_H = \{ (x, \sigma, y) \in \rightarrow \mid x \neq p \}$. Then $G \simeq_{\mathbf{E},\emptyset} H$.
The proof is found in Appendix B.

Prop. 7 confirms that a state with a τ or always enabled event transitions to some other blocking state can also be made blocking, by deleting all outgoing transitions (including ω) from it. The Limited Certain Conflicts Rule should be applied repeatedly, as the deletion of transitions may introduce new blocking states and thus new certain conflicts.

The original Limited Certain Conflicts Rule [10] also allows the removal of nondeterministic transitions: if a transition $p \xrightarrow{\alpha} q$ enters a blocking state q then all other α-transitions from state p can be removed. This aspect of the rule is not changed by always enabled events.

Example 11 Description of how Limited Certain Conflicts has been implemented in Waters.

A major part of nonblocking verification is, once a system has been found to be blocking, giving a sequence of events that may be taken to reach a blocking state. This has not yet been discussed in this report, as the only time it was encountered was when implementing the Limited Certain Conflict rule. Since this rule involves finding additional blocking states using τ and nondeterministic transitions, it has been implemented such that it is easier to find the original blocking state. This is done by giving each state a depth value based on how far it is from the original blocking state.

Algorithm 3 shows how the depth is calculated for each state when extended to include always enabled events. This example shows how it has been applied to figure 11.

Firstly, find all coreachable states. These are the states that can reach a marked state through some sequence of transitions. State S_6 is the only state that cannot reach a marked state, so it is given depth 0, while the rest are given depth -1. Current depth is set to 1. There is an η transition from S_5 to S_6, so state S_5 is blocking and given depth 1. Current depth is set to 2. $S_3 \xrightarrow{\alpha} S_5$, and S_5 has just been found to be blocking. So $S_3 \xrightarrow{\alpha} S_4$ is removed, and now S_3 cannot reach the marked state, so it becomes blocking with depth 2. Current depth is set to 3. $S_1 \xrightarrow{\eta} S_3$ so S_1 becomes blocking with depth 3. Current depth is set to 4. $S_0 \xrightarrow{\alpha} S_1$, so $S_0 \xrightarrow{\alpha} S_2$ is removed. However, since S_0 is marked, it is still found to be coreachable so remains at depth -1.

Finally, blocking states have transitions removed and are merged into state $\perp$. Unreachable states are removed. This results in automaton H. The depths may be used later to show S_6 was a blocking state the system could reach.

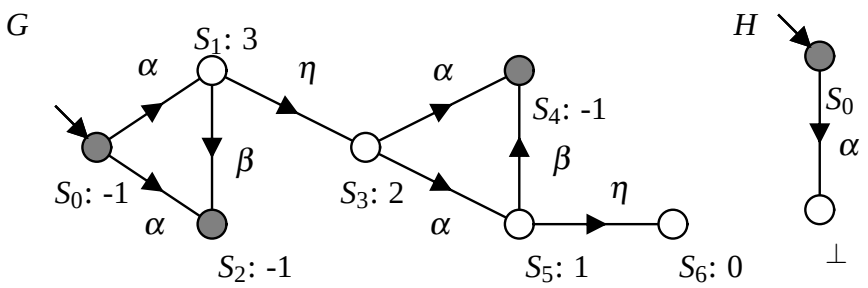

Figure 11: Depths of automaton states after Limited Certain Conflicts is applied.

Algorithm 3 Limited Certain Conflicts Implementation

```
newBlocking = false
FindCoreachableStates()
for all States s do
    if Coreachable(s) = true then
        depth(s) = −1
    else
        depth(s) = 0
        newBlocking = true
    end if
end for
currentDepth = 0
while newBlocking = true do
    newBlocking = false
    currentDepth++
    for all Transitions x -σ-> y where σ ∈ {τ} ∪ E do
        if depth(x) = −1 and depth(y) ≥ 0 then
            blocking(x)
            depth(x) = currentDepth
            newBlocking = true
        end if
    end for
    currentDepth++
    for all Transitions x -σ-> y do
        if depth(x) = −1 and depth(y) ≥ 0 then
            delete Transitions x -σ->
            add Transition x -σ-> y
            FindCoreachableStates()
            if Coreachable(x) = false then
                blocking(x)
                depth(x) = currentDepth
                newBlocking = true
            end if
        end if
    end for
end while
```

FindCoreachableStates() performs a backwards search from all marked states, and is used to find states which can reach a marked state. *blocking*(s) is used on states that have been found to be blocking. It removes all outgoing transitions and markings from state s.

Algorithm 3 finds the blocking states of an automaton using Limited Certain Conflicts with Always Enabled Events. Each of the blocking states is given a depth here, and when a state is found to be blocking the outgoing transitions are removed and it is merged into $\perp$ to save memory. Coreachable states are states that can reach a marked state through some sequence of transitions. They have depth -1. The blocking states found in the first coreachability search have depth 0. States with τ or always enabled events entering a blocking state are blocking, and have odd depth. If a state has multiple transitions of the same event outgoing to different states this is nondeterministic. If a nondeterministic event transition enters a blocking state, then the other transitions of this event on this state are removed. If this causes the state to become blocking then it has even depth.

3.2 Other Selfloop-Only events

Another special type of event are events that are selfloop-only in every automata except the one being simplified. Selfloops with this event can be added or removed freely to the automaton being simplified. This can save memory by removing transitions and can be applied in many places to let other rules be applied.

To verify nonblocking, we check if every state in the final synchronous composition of all automata can reach a marked state. Selfloops in the final synchronous composition have no effect on the blocking nature of the system, since any path between two states passes the same states if all selfloops are removed from the path. So, the final synchronous composition is nonblocking if and only if it is nonblocking with all selfloops removed.

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

If an event only appears on selfloops in all automata, then it can be removed entirely. This is because the event never changes the state of any automata, and so cannot affect nonblocking.

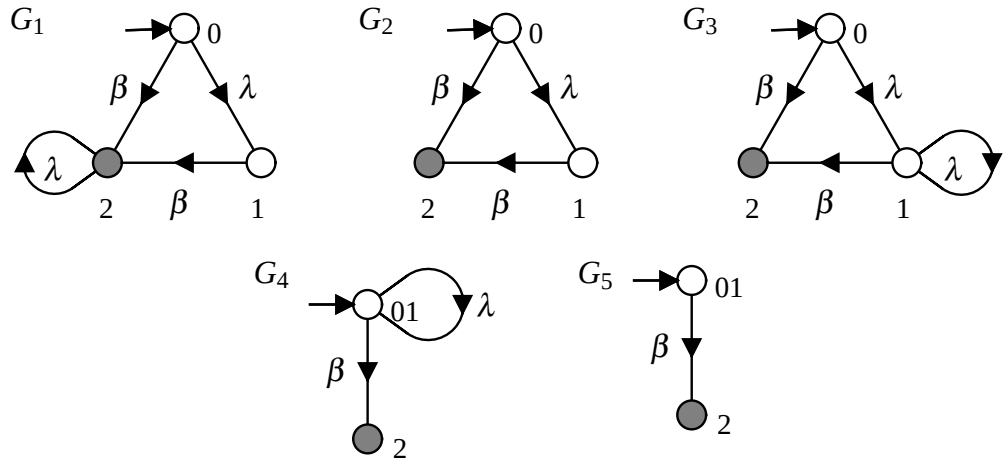

Figure 12: Example of the removal and addition of selfloops.

Example 12 Figure 12 shows a sequence of conflict-preserving changes to an automaton containing the selfloop-only event λ. First, the λ-selfloop in G_1 is removed to create G_2. In G_2, states 0 and 1 are close to observation equivalent, as they both have a β-transition to state 2; however 0 has a λ-transition to 1 and 1 does not. Yet, it is possible to add a λ-selfloop to state 1 and create G_3. Now states 0 and 1 are observation equivalent and can be merged to create G_4. Finally, the λ-selfloop in G_4 is removed to create G_5.

Prop. 8 below confirms that the Selfloop Removal Rule preserves conflict equivalence when selfloop-only events are considered.

Proposition 8 Let $G = \langle \mathbf{A}, Q, \rightarrow_G, Q^\circ \rangle$ and $H = \langle \mathbf{A}, Q, \rightarrow_H, Q^\circ \rangle$ be automata with $\rightarrow_H = \rightarrow_G \cup \{(q, \lambda, q)\}$ for some $\lambda \in \mathbf{A}$. Then $G \simeq_{\emptyset,\{\lambda\}} H$.
The proof is found in Appendix D.

Prop. 8 shows that the addition of a single selfloop preserves conflict equivalence. it can be applied in reverse to remove selfloops, and it can be applied repeatedly to add or remove several selfloops in an automaton or in the entire system.

Example 13 Figure 13 shows that selfloop-only event transitions can be removed if they are parallel to a τ transition. After λ has been added to state 1 in G_2, it is easy to see that $0 \overset{\lambda}{\Rightarrow} 1$ is possible with or without the $0 \xrightarrow{\lambda} 1$ transition. This means it is redundant, and can be removed to get automaton G_3. This automaton can be simplified further by removing the λ-selfloop transition, resulting in G_4.

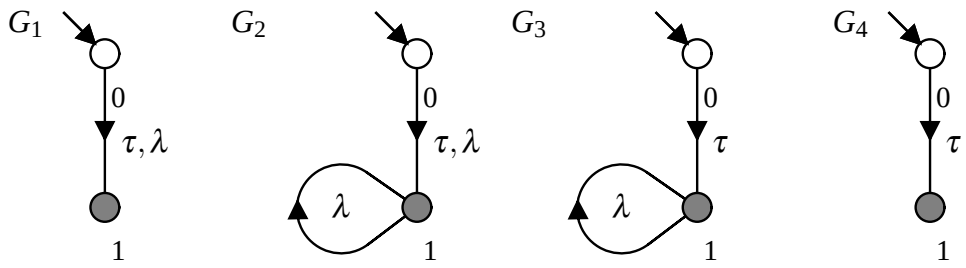

Figure 13: Example of the removing redundant selfloop-only transition.

The implementation in section 6 uses selfloop removal whenever applicable to delete as many selfloops as possible. When creating an automaton, selfloop transitions are not created if the event is recognised to be Other Selfloop-Only. In addition, observation equivalence has been modified to assume the presence of selfloops for all selfloop-only events in all states, so as to achieve the best possible state-space reduction.

4 Finding Always Enabled and Selfloop-only Events

Before any of the extended rules can be used to simplify an automaton, we need to know which events are always enabled and selfloop-only. This section discusses how these events may be found.

Assume the system (2) encountered during compositional verification is

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

and automaton G_1 is to be simplified.

An event must be always enabled or selfloop-only in all the automata not being simplified, $T = G_2 \parallel \cdots \parallel G_n$. For each component automaton G_i, such events are easy to detect based on definition 10 and 11 in Section 2.7. An always enabled event is enabled on every state in the automaton, and a selfloop-only event is only present in the automaton as selfloop transitions. Using these definitions it can also be seen that these properties carry over to the synchronous product, which means we do not need to search for these properties in the synchronous product if we already know they are satisfied in the individual automata.

4.1 Finding Additional Always Enabled Events

When searching for always enabled events, it is often possible to find additional events that satify the definition if a more sophisticated method is used.

As we have seen in 3.1.3, many automata contain blocking states. These states have no outgoing transitions, and any event would be found to be not always enabled in any automaton with a blocking state if using the simple search above. However, as adding a selfloop to a blocking state cannot change whether the system is nonblocking or not, we can imagine a selfloop on the blocking state of the event we are searching for. This makes it possible to find always enabled events in automata with blocking states.

We can also use τ transitions to find more always enabled events, as the definition states that in each state it holds that $\overset{\eta}{\Rightarrow}$ rather than $\overset{\eta}{\rightarrow}$. This means that we can do any number of silent τ transitions to reach a state that has η enabled, rather than needing to have η enabled in every state. In addition, this method can be made even more powerful if redundant tau transitions are added to an automaton. This increases the number of tau transitions, and so we can find many more always enabled events.

Example 14 Consider automaton G in figure 14. It can be seen that $0 \overset{\eta}{\rightarrow}$ and $2 \overset{\eta}{\rightarrow}$ but state 1 has no outgoing η transition. However, $1 \overset{\tau}{\rightarrow} 2 \overset{\eta}{\rightarrow}$, so $1 \overset{\eta}{\Rightarrow}$ Therefore η can be considered as an always enabled event in G since $\overset{\eta}{\Rightarrow}$ is true in each state.

Example 15 Consider automaton G in figure 15. It clearly holds that $0 \overset{\eta}{\rightarrow}$, and $1 \overset{\tau}{\rightarrow} 0 \overset{\eta}{\rightarrow}$ and thus $1 \overset{\eta}{\Rightarrow}$. Although η is not enabled in state $\perp$, this state is a blocking state and the set of

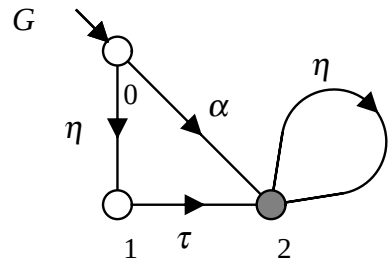

Figure 14: Finding an always enabled event with $\overset{\eta}{\Rightarrow}$

enabled events for blocking states is irrelevant—it is known [14] that G is conflict equivalent to G'. Therefore η can be considered as an always enabled event in G' and thus also in G.

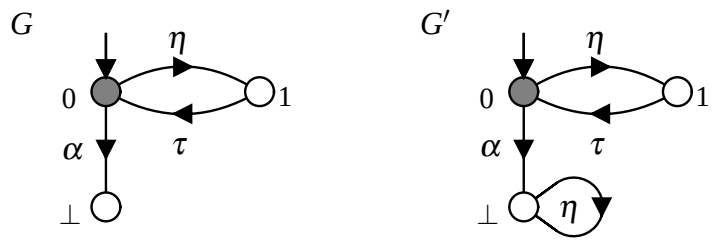

Figure 15: Finding an always enabled event with dump states.

4.2 Conditionally Always Enabled Events

Conditionally always enabled events can be used for the simplification rules but because of how they are defined many more of these events can be found. Many states may not have event η enabled, however if they are not possible in the current state of the automaton being simplified this does not matter. An event is conditionally always enabled if the environment T enables it in all states where it is possible in the automaton G to be simplified.

Definition 13 Let $G = \langle \mathbf{A}, Q_G, \rightarrow_G, Q_G^\circ \rangle$ and $T = \langle \mathbf{A}, Q_T, \rightarrow_T, Q_T^\circ \rangle$ be automata. An event $\sigma \in \mathbf{A}$ is *conditionally always enabled* for G in T, if for all $s \in \mathbf{A}^*$ such that $Q_G^\circ \overset{s\sigma}{\Rightarrow}_G$ and all states $x_T \in Q_T$ such that $Q_T^\circ \overset{s}{\Rightarrow}_T x_T$, it holds that $x_T \overset{\sigma}{\Rightarrow}_T$.

The following proposition 9 shows that the result of compositional nonblocking verification is also preserved with events that are only conditionally always enabled.

Proposition 9 Let G, H, and T be automata, and let $\mathbf{E}$ and $\mathbf{S}$ be event sets such that $G \simeq_{\mathbf{E},\mathbf{S}} H$, and $\mathbf{E}$ is a set of conditionally always enabled for G in T and for H in T, and $\mathbf{S}$ is a set of self-

loop-only for T. Then $G \parallel T$ is nonblocking if and only if $H \parallel T$ is nonblocking.
The proof is found in Appendix E

There is an additional condition that must be satisfied for abstraction rules using conditionally always enabled events. That is that the events must still be conditionally always enabled after abstraction. If no new conditionally always enabled events have been added to the language during abstraction this will always be true, but if for some reason this is done, the new events must be verified to be conditionally always enabled in the automaton after abstraction. This means that in some cases the limited certain conflict rule cannot be applied backwards. Investigation of figure 10 shows that automaton H cannot be 'simplified' to automaton G if η is conditionally always enabled for H but not for G. However, G can still be abstracted to H and the limited certain conflicts rule can always be done in reverse if η is an always enabled event.

Example 16 Finding conditionally always enabled events for $G1$ in $G2$ in figure 16. In this example we are trying to simplify automaton $G1$ somehow, and are looking for any events that are conditionally always enabled and may be used in some abstraction rules here.

Firstly, observe that α is enabled in states 0 and 1 in $G1$. This event will be conditionally always enabled in $G2$ if we α is enabled in every state in $G2$ where the state of $G1$ is 0 or 1. To find these states we can observe $G1 \parallel G2$. Since $G1 \parallel G2$ contains states (0,0), (1,1), (0,2) and (1,0) we must check that α is enabled in states 0, 1 and 2 in $G2$. However, since α is not enabled in 1, it cannot be conditionally always enabled for $G1$ in $G2$. That is, $Q^{\circ}_{G1} \overset{\alpha,\alpha}{\Longrightarrow}$ is enabled, but $Q^{\circ}_{G2} \overset{\alpha,\alpha}{\Longrightarrow}$ is not.

Next, observe that β is enabled in state 1 in $G1$. So, since the synchronous product contains states (1,1) and (1,0) we must check that β is enabled in states 1 and 0 in $G2$. But β is not enabled in state 0 in $G2$ so it cannot be conditionally always enabled for $G1$ in $G2$. $Q^{\circ}_{G1} \overset{\alpha\beta\alpha\beta}{\Longrightarrow}$ is enabled, but $Q^{\circ}_{G2} \overset{\alpha\beta\alpha\beta}{\Longrightarrow}$ is not.

Finally we check γ. γ is enabled in state 2 in $G1$. $G1 \parallel G2$ contains state (2,1) and γ is enabled in state 1 in $G2$. This means γ is a conditionally always enabled event for $G1$ in $G2$.

Example 17 Finding conditionally always enabled events for $G2$ in $G1$ in figure 16. In this example we are trying to simplify automaton $G2$.

Firstly, observe that α is enabled in states 0 and 2 in $G2$. The synchronous product contains states (0,0), (1,0) and (0,2), so states 0 and 1 in $G1$ are checked. Both states enable α, so α is a conditionally always enabled event for $G2$ in $G1$.

Next, observe that β is enabled only in state 1 in $G2$. Using $G1 \parallel G2$, we see that we must check that β is enabled in states 1 and 2 in $G1$. However, since β is not enabled in state 2, it cannot be a conditionally always enabled event for $G2$. $Q^{\circ}_{G2} \overset{\alpha\beta\alpha\alpha\beta}{\Longrightarrow}$ is enabled, but $Q^{\circ}_{G1} \overset{\alpha\beta\alpha\alpha\beta}{\Longrightarrow}$ is not.

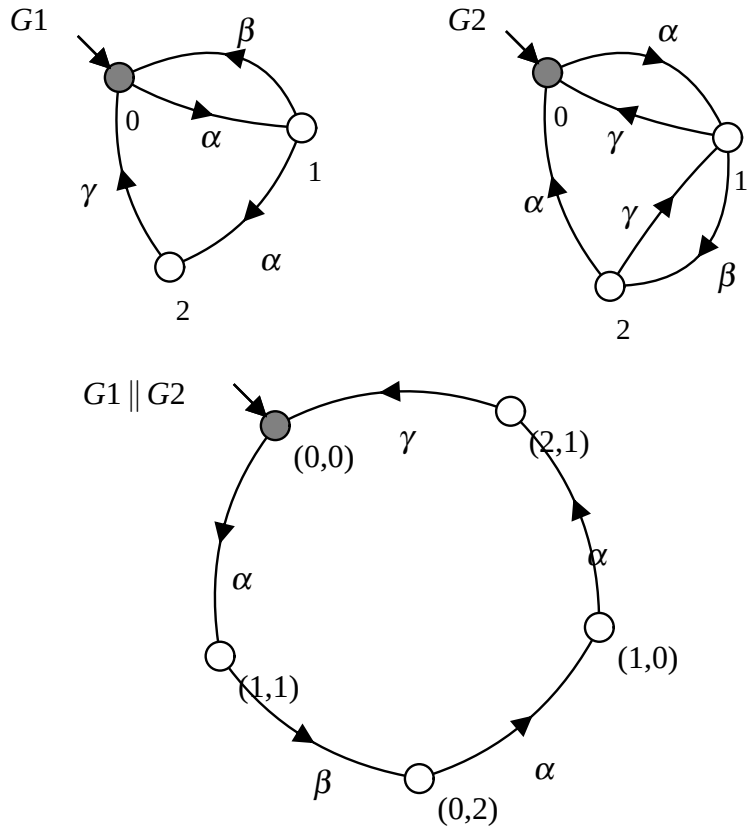

Figure 16: Finding a conditionally always enabled event in G2

Finally, observe that γ is enabled in states 1 and 2 in $G2$. The synchronous product has states (1,1), (2,1) and (0,2). However, γ is not enabled in states 0 and 1 in $G1$. So it cannot be a conditionally always enabled event for $G2$. $Q^{\circ}_{G2} \overset{\alpha\gamma}{\Rightarrow}$ is enabled, but $Q^{\circ}_{G1} \overset{\alpha\gamma}{\Rightarrow}$ is not.

Note that none of these events are always enabled in either automaton. However we have found that γ is conditionally always enabled for $G1$ in $G2$ and α is conditionally always enabled for $G2$ in $G1$. This is clearly a more powerful method for finding special events that can be used in the extended simplification rules, leading to more possible simplification.

Conditionally always enabled events can be used like general always enabled events, but they are more difficult to find. To check the condition of definition 13, it is necessary to explore the state space of $G \| T$, which has the same complexity as a nonblocking check. Yet, the condition is similar to *controllability* [6], which can often be verified quickly by an *incremental controllability check* [4]. The incremental algorithm gradually composes some of the automata of the system (4) until it can be ascertained whether or not a given event is conditionally always enabled. It many cases, it gives a positive or negative answer after composing only a few automata.

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

5 Implementation in Waters

This section discusses how the new simplification rules have been implemented and tested in the Model Checking program Waters.

Waters, *The Waikato Analysis Tool for Events in Reactive Systems*, was developed by the Formal Methods Group at the University of Waikato, and later combined with Supremica, developed by the Department of Signals and Systems at Chalmers University of Technology in Gothenburg, Sweden [1]. It is a tool to model and analyse finite-state machine models. The compositional nonblocking verification algorithm and many simplification rules had already been implemented here. These included the rules that I have now extended such as Silent Continuation and Silent Incoming [16]. I have also needed to add support for always enabled and selfloop-only events.

Before investigating the code, I spent some time creating models of various systems and looking at the examples included with the program. There are hundreds of automata included, which may be used as examples or for testing. By looking at these I found how compositional modelling works, how the automata are composed and then simplified, and how difficult it is to tell if a system will be blocking even after studying it closely. The state-space explosion problem also became clear, as the synchronous composition of most of the example systems were far too large to display graphically.

I then started work on the code, starting with the simplification rules. Each of the simplification rules were in a separate class, so by cloning the classes for the rules I was extending changes could be made without breaking other parts of the system. I quickly found that the code needed to be a lot more complex then the short rule it was performing. Also, while each class had a short explanation, the details were largely uncommented making it difficult to see what was happening. So I began by adding comments at each line to help show the purpose of different parts of the code. Since I was only extending the rule at this point, I did not want to change the structure of the code significantly, instead only changing what was necessary for the rule to respect the new special events. At this point I had not added a way to find the special events, as this was more complicated than changing existing code. I could test that the simplication rules were working properly without this adding possible errors.

To test the new rules, I tested that they simplified automata correctly. Many pairs of before and after automata were created for this purpose. By creating a specific automaton and the automaton that would result after applying the rule I could see if the rule was working as intended. When modelling these test automata I could say which events were always enabled or selfloop-only, removing the need to search for them. I modelled the test automata to test certain complex cases where the rule would be applied, for example when the automata was nondeterministic or could be simplified multiple times. There were also a large number of existing test automata that could be applied, this was important to ensure adding the support for special events hadn't broken any existing code.

After I was satisfied the extended rules would simplify automata as expected I could move deeper into the code. To add support for finding special events I needed to understand a lot more of the structure of the program, particularly the compositional nonblocking algorithm. An im-

portant part of this algorithm is the order automata are chosen to be composed. This is important as it can greatly change the size of the final synchronous composition. There are many factors to consider, including the number of states and transitions in the composition. It is also valuable to have a large number of τ and always enabled events, as these are used when simplifying the automata. Which automata are composed together are chosen heuristically, using an existing two-step approach [10]. In the first step, some *candidate* sets of automata are formed, and in the second a most promising candidate is selected. For each event σ in the model, a candidate is formed consisting of all automata with σ in their alphabet. This is used to increase the number of τ events in the composition. Among these candidates, the candidate with the smallest estimated number of states after abstraction is selected. The estimate is obtained by multiplying the product of the state numbers of the automata forming the candidate with the ratio of the numbers of events in the synchronous composition of the candidate after and before removing any local events. This strategy is called **MustL/MinS** [10, 16]. A new heuristic was written to try maximise the number of always enabled events, however this had no significant improvement over the existing heuristic that was instead chosen.

After identification of a candidate, its automata are composed, and then a sequence of abstraction rules is applied to simplify it. First, τ-loops (definition 8) and observation equivalent redundant transitions [9] are removed from the automaton. This is followed by the Only Silent Incoming Rule (proposition 6), the Only Silent Outgoing Rule [10], the Limited Certain Conflicts Rule (proposition 7), Observation Equivalence (proposition 1), the Non-α Determinisation Rule [16], the Active Events Rule [10], and the Silent Continuation Rule (proposition 5).

When finding special events, the main mistake I wanted to avoid was incorrect identification, as this would quickly lead to simplification that did not preserve nonblocking. I also wanted to ensure that I was finding as many special events as possible in order to maximise the possible simplification. Although my first attempts did not find many special events, I could still change the structure of the compositional verification algorithm to use them. I could then run tests and see nothing had been broken and how much simplification was being added.

The result of my improvements to the algorithm is that during simplification, all selfloops with selfloop-only events are deleted, and observation equivalence and the removal of observation equivalent redundant transitions exploit selfloop-only events for further simplification. Furthermore, the Only Silent Incoming Rule, the Limited Certain Conflicts Rule, and the Silent Continuation Rule take always enabled events into account.

In addition to the small automata used to test single rules, Waters also contains a large test suite. The test suite includes complex industrial models and case studies from various application areas such as manufacturing systems, communication protocols, and automotive electronics. Included are all models used in [16] with at least $5 \cdot 10^7$ reachable states that have been used for experimental results.

I could use these to test the compositional nonblocking verification algorithm with the support for special events added. The main properties to test were that the nonblocking result was the same as expected and that no errors had been introduced. I could also run these tests with the original rules still in place. This meant that when looking at the statistics of what simplification

had occurred, I could see quickly the improvements my rules had compared to the originals. Unfortunately at first I found they only had a very small improvement.

I also saw that the number of always enabled events was much smaller than hoped, so I reinvestigated how they were found. By this point I understood more of the code, and was able to move the search to a place where it was both faster and more effective. I also discovered and implemented the improvements discussed in section 4. All always enabled events were found in the small models that could be investigated carefully, however this is no guarantee that all always enabled events are always found in larger systems. The addition of the conditionally always enabled events algorithm increased the amount of simplification significantly, though at the cost of speed.

6 Experimental Results

Experimental results have been gathered from the implementation in Waters. This data is shown in Table 6.

A number of strategies have been used to show the improvements of the extended always enabled and selfloop-only events rules compared to the original simplification rules.

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

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

For each model, Table 6 shows the total number of reachable states in the synchronous composition (Size) if known, and whether or not the model is nonblocking (Res). Then it shows for each strategy, the number of states in the largest automaton encountered during abstraction (Peak States), the number of states in the synchronous composition explored after abstraction (Final States), and the total verification time (Time). The best result in each category is highlighted in bold.

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

All experiments are run on a standard desktop computer using a single core 3.3 GHz CPU and 8 GB of RAM. The experiments are controlled by state limits. If during abstraction the synchronous composition of a candidate has more than 100,000 states, it is discarded and another candidate is chosen instead. The state limit for the final synchronous composition after abstraction is 10^8 states. If this limit is exceeded, the run is aborted and the corresponding table entries are left blank.

The results in Table 6 demonstrate that compositional verification can check the nonblocking property of systems with up to 10^{14} states in a matter of seconds. The exploitation of always enabled and selfloop-only events reduces the peak or final state numbers in many cases. This is important as these numbers are the limiting factors in compositional verification.

Unfortunately, the runtimes rarely improve as the smaller state numbers are outweighed by the effort to find always enabled and selfloop-only events. The search has to be repeated after each abstraction step, because each abstraction can produce new always enabled or selfloop-only events, and the cost increases with the number of steps and events. Conditionally always enabled

events can produce better abstractions, but it takes a lot of time to find them.

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

A significant result of my work is that the large PROFIsafe models [12] can only be verified compositionally with selfloop-only events. By adding always enabled and selfloop-only events to the available tools, it becomes possible to solve problems that are not solvable otherwise.

Table 1: Experimental Results

Model	Size	Res	**None**			**SL**			**AE**			**SL/AE**			**SL/AE** $\langle 200 \rangle$			**SL/AE** $\langle 1000 \rangle$		
			Peak states	Final states	Time [s]	Peak states	Final states	Time [s]	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	$1.02 \cdot 10^{9}$	yes	1090	5	**1.2**	1090	5	1.3	1090	5	1.3	1090	5	1.2	**892**	5	24.5	**892**	5	31.9
aip0alps	$3.00 \cdot 10^{8}$	no	18	16	**0.2**	18	16	0.2	18	16	0.3	18	16	0.3	18	16	1.4	18	16	1.4
aip0tough	$1.02 \cdot 10^{10}$	no	96049	19833682	82.6	96049	17066874	47.1	96049	19829534	76.2	96049	**17063170**	**46.0**	96049	**17063170**	46.7	96049	**17063170**	105.8
aip1efa$\langle 3 \rangle$	$6.88 \cdot 10^{8}$	yes	40290	1878708	**12.5**	40290	1878708	12.7	40290	1878708	13.2	40290	1878708	13.2	32980	1726127	17.2	**31960**	**1707905**	40.8
aip1efa$\langle 16 \rangle$	$9.50 \cdot 10^{12}$	no	65520	13799628	**21.8**	65520	13799628	22.0	65520	13799628	22.3	65520	13799628	22.2	65520	13799628	28.9	65520	13799628	48.0
aip1efa$\langle 24 \rangle$	$1.83 \cdot 10^{13}$	no	6384	13846773	18.1	6384	13846773	**18.1**	6384	13846773	18.4	6384	13846773	18.2	5313	13846773	23.8	**5292**	13846773	42.3
fencaiwon09	$1.03 \cdot 10^{8}$	yes	10421	105	**2.3**	10421	105	2.4	10421	105	2.4	10421	105	2.4	10421	105	3.5	10421	**78**	6.3
fencaiwon09b	$8.93 \cdot 10^{7}$	no	10421	81	1.9	10421	81	**1.9**	10421	81	2.0	10421	81	1.9	10421	**62**	3.4	10421	**62**	5.7
ftechnik	$1.21 \cdot 10^{8}$	no	172	0	**0.3**	172	0	0.3	172	0	0.4	172	0	0.4	172	0	4.3	172	0	5.4
profisafe_i4		yes				74088	**409**	84.2				**49152**	9864	**67.2**	**49152**	9864	630.2	**49152**	9864	2873.7
profisafe_i5		yes				98304	57888	**68.5**				98304	**12070**	71.9	98304	**12070**	1181.6	98304	**12070**	2969.0
profisafe_i6		yes				55296	**148284**	**51.2**				**52224**	628131	84.9	**52224**	628131	1830.2	**52224**	628131	4179.5
tbed_ctct	$3.94 \cdot 10^{13}$	no	43825	0	**14.1**	43825	0	14.2	43825	0	16.3	43825	0	16.5	43825	0	20.8	43825	0	43.6
tbed_hisc	$5.99 \cdot 10^{12}$	yes	1757	**33**	2.4	1757	**33**	**2.4**	**1705**	**33**	2.6	**1705**	**33**	2.5	**1705**	**33**	23.6	**1705**	138	90.1
tbed_valid	$3.01 \cdot 10^{12}$	yes	50105	3839	**9.5**	50105	3580	9.7	50105	2722	10.3	50105	**2621**	10.0	50105	**2621**	14.6	50105	**2621**	28.3
tip3	$2.27 \cdot 10^{11}$	yes	**6399**	173	**3.1**	**6399**	173	3.2	12303	153	4.5	12303	153	4.5	12303	153	6.0	12303	**149**	6.4
tip3_bad	$5.25 \cdot 10^{10}$	no	**1176**	14	0.9	1254	14	**0.9**	**1176**	**0**	1.1	1231	**0**	1.1	1231	**0**	2.9	1231	**0**	3.7
verriegel3	$9.68 \cdot 10^{8}$	yes	3303	2	1.6	3303	2	**1.3**	3349	2	1.7	3349	2	1.4	**2644**	2	6.0	**2644**	2	9.7
verriegel3b	$1.32 \cdot 10^{9}$	no	**1764**	0	**1.0**	**1764**	0	1.1	1795	0	1.1	1795	0	1.2	1795	0	5.8	1795	0	8.4
verriegel4	$4.59 \cdot 10^{10}$	yes	**2609**	2	**1.3**	**2609**	2	1.4	2644	2	1.3	2644	2	1.5	2644	2	8.6	2644	2	15.4
verriegel4b	$6.26 \cdot 10^{10}$	no	**1764**	0	**1.1**	**1764**	0	1.2	1795	0	1.2	1795	0	1.4	1795	0	8.2	1795	0	13.2
6linka	$2.45 \cdot 10^{14}$	no	64	0	**0.4**	64	0	0.4	64	0	0.4	64	0	0.4	64	0	2.2	64	0	2.7
6linki	$2.75 \cdot 10^{14}$	no	61	0	0.3	61	0	**0.3**	61	0	0.3	61	0	0.3	61	0	1.7	61	0	2.0
6linkp	$4.43 \cdot 10^{14}$	no	32	0	**0.3**	32	0	0.3	32	0	0.3	32	0	0.3	32	0	1.7	32	0	2.0
6linkre	$6.21 \cdot 10^{13}$	no	118	12	**0.5**	118	12	0.5	**106**	**0**	0.5	**106**	**0**	0.5	**106**	**0**	2.3	**106**	**0**	2.7

7 Conclusions

The goal of this project was to create better simplification rules that could be used to simplify systems further and therefore allow the verification of larger systems. This was done by taking into account additional information about the context in which an automaton to be simplified is used. Specifically, *always enabled* and *selfloop-only* events are easy to discover and can be used to simplify automata. These improved simplification rules have been proven to preserve nonblocking and have been implemented in Waters. Experimental results are gathered from testing a set of models that include complex industrial models from multiple areas. The experiments demonstrate that the improved algorithms can simplify these models further and that a previously unanalyzable model can be verified with the improved algorithms.

7.1 Future Work

In future work, it is of interest whether the algorithms to detect and use always enabled and selfloop-only events can be improved.

Conditional τ events are another type of special event that have also been investigated but not yet implemented. These are events which are *both* conditionally always enabled and conditionally selfloop-only. These events are very interesting and potentially powerful, as they can be completely replaced in the automaton being simplified by τ.

Finding Conditionally Always Enabled Transitions is another way of using information from automata other than the one being simplified. These transitions are those which are not disabled by any other automata in the system, similar to conditionally always enabled events. For example, in figure 16 there are four transitions in $G1$. Transitions $0 \xrightarrow{\alpha} 1$ and $2 \xrightarrow{\gamma} 0$ are conditionally always enabled for $G1$ in $G2$. These transitions should be able to be used similarly to conditionally always enabled events in the extended rules of this report, and could possibly be used to build new abstraction rules.

Appendix

A Silent Continuation Proof

Proof. Let T be an automaton such that $\mathbf{E}$ is always enabled for 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 3, there exists $x \in \tilde{x}$ such that $Q^\circ \overset{s}{\Rightarrow} x$.
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 2, this implies $(\tilde{x}, x_T) = ([x], x_T) \overset{t\omega}{\Rightarrow}$, i.e., $G \parallel T$ is nonblocking.
Conversely assume that $(G/\sim) \parallel T$ is nonblocking.
Let $G \parallel T \overset{s}{\Rightarrow} (x, x_T)$. Then, by lemma 2 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 3 and since x is the only state in $[x]$, it follows that $(x, x_T) \overset{t\omega}{\Rightarrow}$.

2. $x \xrightarrow{\eta} y$ 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 2, 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 3, there exists $y' \in [y]$ such that $(y', y_T) \overset{t\omega}{\Rightarrow}$.
   Since $y' \sim_{\text{inc}} y$ and $x \xrightarrow{\eta} y$, it follows that $x \xrightarrow{\eta} y'$.
   Therefore, $(x, x_T) \xrightarrow{\eta} (y', y_T) \overset{t\omega}{\Rightarrow}$.

3. $x \xrightarrow{\tau} y$ for some $y \in Q$.
   Since G is τ-loop free and finite, there exists a state $y' \in Q$ such that $x \overset{\varepsilon}{\Rightarrow} y'$ and $y' \xrightarrow{\tau}$ does not hold.
   If $[y'] = \{y'\}$ then the proof continues as in case 1.
   Otherwise, since $y' \xrightarrow{\tau}$ does not hold, it follows by the properties of $\sim$ that $y' \xrightarrow{\eta}$ for some $\eta \in \mathbf{E}$, and the proof continues as in case 2.

In all three cases, it has been shown that $(x, x_T) \overset{t\omega}{\Rightarrow}$.
Therefore $G \parallel T$ is nonblocking.

□

B Limited Certain Conflict Proof

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

$$(x, x_T) = (x^0, x_T^0) \xrightarrow{\sigma_1}_{G\|T} (x^1, x_T^1) \xrightarrow{\sigma_2}_{G\|T} \cdots \xrightarrow{\sigma_n}_{G\|T} (x^n, x_T^n) \xrightarrow{\omega}_{G\|T} \quad . \tag{5}$$

Let $(x^i, x_T^i) \xrightarrow{\sigma_{i+1}}_{G\|T} (x^{i+1}, x_T^{i+1})$ be an arbitrary transition on the path (5).
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 $(x^i, x_T^i) \xrightarrow{\sigma_{i+1}}_{H\|T} (x^i, x_T^{i+1}) = (x^{i+1}, x_T^{i+1})$.
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 $x^i = p$, $\sigma_{i+1} = \eta$, and $x^{i+1} = q$.
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 $(x^i, x_T^i) = (p, x_T^i) \xrightarrow{\eta}_{G\|T} (q, y_T)$.
If $\eta = \tau$, then $(x^i, x_T^i) = (p, x_T^i) \xrightarrow{\tau}_{G\|T} (q, x_T^i)$.
In both cases, it follows that $(x, x_T) \xrightarrow{\sigma_1 \cdots \sigma_{i-1}}_{G\|T} (x^i, x_T^i) = (p, x_T^i) \xrightarrow{\eta}_{G\|T} (q, y_T)$ for some state y_T of T.
However, (q, y_T) is a blocking state because q is a blocking state in G.
Then $G \parallel T$ is blocking in contradiction to the assumption.
It follows that the transition $x^i \xrightarrow{\sigma_i}_G x^{i+1}$ was not removed and is still present in H.
Again it holds that $(x^i, x_T^i) \xrightarrow{\sigma_{i+1}}_{H\|T} (x^{i+1}, x_T^{i+1})$.
Thus, the path (5) 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 $(x, x_T) \xrightarrow{t\omega}$.
Let $t = \sigma_1 \cdots \sigma_n$ and write

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

where x^0 and x_T^0 are initial states of G and T, respectively.
It is shown by induction on $k = 0, \ldots, n$ that $(x^0, x_T^0) \xrightarrow{\sigma_1 \cdots \sigma_k}_{H\|T} (x^k, x_T^k)$.
This is trivial for $k = 0$.

Now assume $(x^0, x_T^0) \xrightarrow{\sigma_1 \cdots \sigma_k}_{H \| T} (x^k, x_T^k)$ 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 $(x^0, x_T^0) \xrightarrow{\sigma_1 \cdots \sigma_k}_{H \| T} (x^k, x_T^k) \xrightarrow{\sigma_{k+1}}_{H \| T} (x^k, x_T^{k+1}) = (x^{k+1}, x_T^{k+1})$.
Otherwise, if $x^k \xrightarrow{\sigma_{k+1}}_G x^{k+1}$, assume that the this transition does not exist in H.
This means that $x^k = p$, and thus by inductive assumption $(x^0, x_T^0) \xrightarrow{\sigma_1 \cdots \sigma_k}_{H \| T} (p, x_T^k)$, where p is a deadlock state in H (with no outgoing transitions by construction, and thus ω never possible).
Then $H \| T$ 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, $(x^0, x_T^0) \xrightarrow{\sigma_1 \cdots \sigma_k}_{H \| T} (x^k, x_T^k) \xrightarrow{\sigma_{k+1}}_{H \| T} (x^{k+1}, x_T^{k+1})$.
Since furthermore G and H have the same initial states, it follows from the induction that $H \| T \xrightarrow{s} (x, x_T)$.
Since $H \| T$ is nonblocking, it follows that $(x, x_T) \xrightarrow{t\omega}_{H \| T}$ for some $t \in \mathbf{A}_\tau^*$.
Since $\rightarrow_H \subseteq \rightarrow_G$, this implies $(x, x_T) \xrightarrow{t\omega}_{G \| T}$.
It follows that $G \| T$ is nonblocking. $\square$

C Selfloop Removal Lemma

The Selfloop-only addition proof uses this lemma to show that selfloop removal does not affect the existence of paths.

Lemma 10 Let $G = \langle \mathbf{A}, Q, \rightarrow_G, Q^\circ \rangle$ and $H = \langle \mathbf{A}, Q, \rightarrow_H, Q^\circ \rangle$ be automata with $\rightarrow_H = \rightarrow_G \cup \{(q, \lambda, q)\}$ for some $\lambda \in \mathbf{A}$.
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
$(x, x_T) = (x^0, x_T^0) \xrightarrow{\sigma_1}_{H\|T} \cdots \xrightarrow{\sigma_n}_{H\|T} (x^n, x_T^n) \xrightarrow{\sigma_{n+1}}_{H\|T} (x^n, x_T^n) = (y, y_T)$,
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 $x^n \xrightarrow{\sigma_{n+1}}_H x^{n+1}$ does not hold, 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, $(x, x_T) = (x^0, x_T^0) \rightarrow_{G\|T} (x^n, x_T^n) = (x^{n+1}, x_T^{n+1})$.
If $x^n \xrightarrow{\sigma_{n+1}}_H x^{n+1}$ is the selfloop $q \xrightarrow{\lambda}_H q$, then $x^{n+1} = x^n$ 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
$(x, x_T) = (x^0, x_T^0) \rightarrow_{G\|T} (x^n, x_T^n) \xrightarrow{\sigma_{n+1}}_{G\|T} (x^{n+1}, x_T^{n+1})$. □

D Selfloop-Only Addition Proof

Proof. Let T be an automaton such that λ is selfloop-only for T.
First assume that $G \parallel T$ is nonblocking.
To see that $H \parallel T$ is nonblocking, let $H \parallel T \xrightarrow{s} (x, x_T)$.
By lemma 10, 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 $(x, x_T) \xrightarrow{t\omega}_{G \parallel T}$.
Since $\rightarrow_G \subseteq \rightarrow_H$, it follows that $(x, x_T) \xrightarrow{t\omega}_{H \parallel T}$, i.e., $H \parallel T$ is nonblocking.
Conversely, assume that $H \parallel T$ is nonblocking.
To see that $G \parallel T$ is nonblocking, let $G \parallel T \xrightarrow{s} (x, x_T)$.
Since $\rightarrow_G \subseteq \rightarrow_H$, 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 $(x, x_T) \xrightarrow{t}_{H \parallel T} (y, y_T) \xrightarrow{\omega}_{H \parallel T}$.
Using lemma 10, it follows that $(x, x_T) \rightarrow_{G \parallel T} (y, y_T)$.
Furthermore, it follows from $y \xrightarrow{\omega}_H$ that $y \xrightarrow{\omega}_G$ because $\lambda \neq \omega$ and $\rightarrow_G$ and $\rightarrow_H$ only differ in a λ-transition.
Thus, $(x, x_T) \rightarrow_{G \parallel T} (y, y_T) \xrightarrow{\omega}_{G \parallel T}$, i.e., $G \parallel T$ is nonblocking. $\square$

E Conditionally Always Enabled Events Proof

Proof. This proof shows how conditionally always enabled events can be used similary to regular always enabled events.

Consider the automata G, H and T such that $G \simeq_{\mathbf{E},\mathbf{S}} H$ and $\mathbf{E}$ is a set of events that are conditionally always enabled for G in T and for H in T and $\mathbf{S}$ is a set of events selfloop only in T.
Construct the automaton T' using T.
For each event $\eta \in \mathbf{E}$, add η-selfloops to all states in T where η is not already enabled.
By construction, the events in $\mathbf{E}$ are always enabled in T'.
So the conflict equivalence rule for always enabled events can be applied using T'.
$G \simeq_{\mathbf{E},\mathbf{S}} H$, if for every automaton T' such that $\mathbf{E}$ is a set of events that are always enabled in T' and $\mathbf{S}$ is a set of selfloop-only in T', it holds that

$$G \parallel T' \text{ is nonblocking} \iff H \parallel T' \text{ is nonblocking.} \tag{7}$$

However, the η selfloops that were added to T to give T' are removed when the synchronous product is taken.
This is because the η selfloops were added to states that did not have η enabled in G or H, by definition of conditionally always enabled events.
This means that $G \parallel T' = G \parallel T$ and $H \parallel T' = H \parallel T$.
Substitution into equation 7 above gives

$$G \parallel T \text{ is nonblocking} \iff H \parallel T \text{ is nonblocking.} \tag{8}$$

□

References

[1] Knut Åkesson, Martin Fabian, Hugo Flordal, and Robi Malik. Supremica—an integrated environment for verification, synthesis and simulation of discrete event systems. In *Proc. 8th Int. Workshop on Discrete Event Systems, WODES '06*, pages 384–385, Ann Arbor, MI, USA, July 2006.

[2] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.

[3] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and Ph. Schnoebelen. *Systems and Software Verification*. Springer, 2001.

[4] Bertil A. Brandin, Robi Malik, and Petra Malik. Incremental verification and synthesis of discrete-event systems guided by counter-examples. *IEEE Trans. Control Syst. Technol.*, 12(3):387–401, May 2004.

[5] Stephen D. Brookes and William C. Rounds. Behavioural equivalence relations induced by programming logics. In *Proc. 16th Int. Colloquium on Automata, Languages, and Programming, ICALP '83*, volume 154 of *LNCS*, pages 97–108. Springer, 1983.

[6] C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Kluwer, September 1999.

[7] C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Springer, 2 edition, 2008.

[8] R. De Nicola and M. C. B. Hennessy. Testing equivalences for processes. *Theoretical Comput. Sci.*, 34(1–2):83–133, November 1984.

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

[10] Hugo Flordal and Robi Malik. Compositional verification in supervisory control. *SIAM J. Control and Optimization*, 48(3):1914–1938, 2009.

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

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

[13] Robi Malik. On the set of certain conflicts of a given language. In *Proc. 7th Int. Workshop on Discrete Event Systems, WODES '04*, pages 277–282, Reims, France, September 2004.

[14] Robi Malik. The language of certain conflicts of a nondeterministic process. Working Paper 05/2010, Dept. of Computer Science, University of Waikato, Hamilton, New Zealand, 2010.

[15] Robi Malik and Ryan Leduc. Generalised nonblocking. In *Proc. 9th Int. Workshop on Discrete Event Systems, WODES '08*, pages 340–345, Göteborg, Sweden, May 2008.

[16] Robi Malik and Ryan Leduc. Compositional nonblocking verification using generalised nonblocking abstractions. *IEEE Trans. Autom. Control*, 58(8):1–13, August 2013.

[17] Robi Malik, David Streader, and Steve Reeves. Conflicts and fair testing. *Int. J. Found. Comput. Sci.*, 17(4):797–813, 2006.

[18] Robin Milner. *Communication and concurrency*. Series in Computer Science. Prentice-Hall, 1989.

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

[20] Colin Pilbrow and Robi Malik. Compositional nonblocking verification with always enabled events and selfloop-only events. In *Proc. 2nd Int. Workshop on Formal Techniques for Safety-Critical Systems, FTSCS 2013*, pages 147–162, Queenstown, New Zealand, 2013.

[21] Peter J. G. Ramadge and W. Murray Wonham. The control of discrete event systems. *Proc. IEEE*, 77(1):81–98, January 1989.

[22] Rong Su, Jan H. van Schuppen, Jacobus E. Rooda, and Albert T. Hofkamp. Nonconflict check by using sequential automaton abstractions based on weak observation equivalence. *Automatica*, 46(6):968–978, June 2010.

[23] Simon Ware and Robi Malik. Conflict-preserving abstraction of discrete event systems using annotated automata. *Discrete Event Dyn. Syst.*, 22(4):451–477, 2012.