Working Paper Series
ISSN 1177-777X

# THE LANGUAGE OF CERTAIN CONFLICTS OF A NONDETERMINISTIC PROCESS

**Robi Malik**

Working Paper: 05/2010
July 7, 2010

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

# The Language of Certain Conflicts of a Nondeterministic Process

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

July 7, 2010

**Abstract**

The language of certain conflicts is the most general set of behaviours of a nondeterministic process, which certainly lead to a livelock or deadlock when accepted by another process running in parallel. It is of great use in model checking to detect livelocks or deadlocks in very large systems, and in process-algebra to obtain abstractions preserving livelock and deadlock. Unfortunately, the language of certain conflicts is difficult to compute and has only been approximated in previous work. This paper presents an effective algorithm to calculate the language of certain conflicts for any given nondeterministic finite-state process and discusses its properties. The algorithm is shown to be correct and of exponential complexity.

## 1 Introduction

Blocking or conflicts are common faults in concurrent programs that can be very subtle and hard to detect. This includes both the possibility of deadlock, where processes are stuck and unable to continue at all, and livelock, where processes continue to run forever without achieving any further progress. Conflicts have long been studied in the field of discrete event systems [8, 22], which is applied to modelling of complex safety-critical systems [4, 5]. To improve the reliability of such systems, techniques are needed to detect the presence or verify the absence of conflicts in models of an ever increasing size.

In discrete-events theory [8, 22], the absence of conflicts is formalised using the *nonblocking* property, which is used very successfully for *synthesis* [8, 22]. A lot of research has been conducted to study the compositional semantics [15, 17] of nonblocking and its verification [11, 20]. Existing model checking techniques [10] can be used to verify nonblocking, but they are limited by the state space explosion problem.

Alternatives to mitigate this problem include *incremental* or *modular verification*, which attempt to find properties of a large system by analysing only certain parts of it, or *compositional verification*, where individual system components are *abstracted* or *simplified* before or while being composed. Such techniques are very effective for safety properties [1, 6, 2]; deadlock-preserving abstractions [12] and more general temporal logic properties [10] have also been considered.

Unfortunately, most standard abstraction techniques do not preserve the nonconflicting property, and only more recently compositional verification has been used successfully to verify nonconflicting [11]. While these results look very promising, it is still difficult to find good conflict-preserving abstractions, and several questions remain open. One powerful abstraction relies on the *language* or *set of certain conflicts* [16], which is the most general set of traces of a subsystem that are guaranteed to cause blocking in any environment. If traces of certain conflicts are detected early, this can considerably reduce the efforts of verification.

The set of certain conflicts is shown in [17] to be the one distinguishing feature that separates conflict equivalence from *fair testing* [7, 19]. In [16], the set of certain conflicts is used to detect conflicts in large systems of synchronised processes, and this idea is further extended in [11] to compute conflict-preserving abstractions. However, the existing works do not calculate the set of certain conflicts accurately, and only use approximations that are easy to compute.

This paper discusses an algorithm to compute the set of certain conflicts accurately for any nondeterministic process. Section 2 summarises necessary notation from automata theory and process-algebra. Then section 3 introduces the language of certain conflicts with some of its properties, and section 4 presents a fixed-point algorithm to compute it for any given finite-state machine, and proves the correctness of this algorithm. Afterwards, section 5 shows how the results can be used to simplify nondeterministic processes in a conflict-preserving way, and section 6 adds some concluding remarks.

## 2 Notation and Preliminaries

This section introduces the notations used throughout this paper. Processes are represented as labelled transition systems, with the possibility of nondeterminism, which naturally arises from abstraction and hiding operations [13, 18, 23]. Process behaviour is described using languages, with notations taken from the background of discrete event systems and automata theory [22, 14].

### 2.1 Alphabets and Languages

Traces and languages are a simple means to describe process behaviours. Their basic building blocks are *actions*, which are taken from a finite *alphabet* $\mathbf{A}$. Two special actions are used, the *silent action* $\tau$ and the *termination action* $\omega$. These are never included in an alphabet $\mathbf{A}$ unless mentioned explicitly. To include them, $\mathbf{A}_\omega = \mathbf{A} \cup \{\omega\}$ and $\mathbf{A}_{\omega,\tau} = \mathbf{A} \cup \{\omega, \tau\}$ are used.

$\mathbf{A}^*$ denotes the set of all finite *strings* or *traces* of the form $\alpha_1\alpha_2 \ldots \alpha_k$ of actions from $\mathbf{A}$, including the *empty trace* $\varepsilon$. A *language* over $\mathbf{A}$ is any subset $\mathcal{L} \subseteq \mathbf{A}^*$. The *concatenation* of two traces $s, t \in \mathbf{A}^*$ is written as $st$. Traces, alphabets, and languages can also be catenated, e.g., $s\mathcal{L} = \{\, st \mid t \in \mathcal{L} \,\}$. Trace $s$ is a *prefix* of $t$, denoted $s \sqsubseteq t$, if there exists a trace $u$ such that $t = su$. The *prefix-closure* $\overline{\mathcal{L}}$ of a language $\mathcal{L} \subseteq \mathbf{A}^*_{\omega,\tau}$ is the set of all prefixes of traces in $\mathcal{L}$, i.e., $\overline{\mathcal{L}} = \{\, s \in \mathbf{A}^*_{\omega,\tau} \mid s \sqsubseteq t \text{ for some } t \in \mathcal{L} \,\}$. A language $\mathcal{L}$ is *prefix-closed* if $\overline{\mathcal{L}} = \mathcal{L}$.

### 2.2 Processes

In the context of this paper, processes are modelled as nondeterministic *labelled transition systems* $P = \langle \mathbf{A}, Q, \rightarrow, Q^\circ \rangle$, where $\mathbf{A}$ is the alphabet of actions, $Q$ is the set of *states*, $\rightarrow \subseteq Q \times \mathbf{A}_{\omega,\tau} \times Q$ is the *transition relation*, and $Q^\circ \subseteq Q$ is the (possibly empty) set of *initial states*.

The transition relation is written in infix notation $x \xrightarrow{\alpha} y$, and it is extended to traces in $\mathbf{A}^*_{\omega,\tau}$ by letting $x \xrightarrow{\varepsilon} x$ for all $x \in Q$ and $x \xrightarrow{s\alpha} z$ if $x \xrightarrow{s} y \xrightarrow{\alpha} z$ for some state $y \in Q$. For state sets $Q_1, Q_2 \subseteq Q$, the notation $Q_1 \xrightarrow{s} Q_2$ denotes the existence of $x_1 \in Q_1$ and $x_2 \in Q_2$ such that $x_1 \xrightarrow{s} x_2$. Also, $x \rightarrow y$ denotes the existence of a trace $s \in \mathbf{A}^*_{\omega,\tau}$ such that $x \xrightarrow{s} y$, and $x \xrightarrow{s}$ denotes the existence of a state $y \in Q$ such that $x \xrightarrow{s} y$.

Processes use the termination action $\omega$ to indicate successful termination, and the transition relation must satisfy the additional requirement that, whenever $x \xrightarrow{\omega} y$, it does not hold that $y \rightarrow$. The traditional set of *marked* or *terminal* states [14] then can be defined as $Q^\omega = \{\, x \in Q \mid x \xrightarrow{\omega} \}$.

To support hiding of silent actions, 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 $s' \in \mathbf{A}^*_{\omega,\tau}$ such that $x \xrightarrow{s'} y$ and $s$ is obtained from $s'$ by removing all silent ($\tau$) actions. Notations such as $Q_1 \overset{s}{\Rightarrow} Q_2$, $x \Rightarrow y$, and $x \overset{s}{\Rightarrow}$ are defined analogously to $\rightarrow$.

The set of all processes with action alphabet $\mathbf{A}$ is denoted by $\Pi_\mathbf{A}$. The transition relation is also defined for processes, denoting by $P \overset{s}{\Rightarrow} P'$ that process $P \in \Pi_\mathbf{A}$ evolves into $P' \in \Pi_\mathbf{A}$ by executing actions $s \in \mathbf{A}^*_\omega$. This is defined as $\langle \mathbf{A}, Q, \rightarrow, Q^\circ \rangle \overset{s}{\Rightarrow} (\mathbf{A}, Q, \rightarrow, \{x\})$ for each $x \in Q$ such that $Q^\circ \overset{s}{\Rightarrow} x$. The notation $P \overset{s}{\Rightarrow}$ means that $P \overset{s}{\Rightarrow} P'$ for some $P' \in \Pi_\mathbf{A}$.

The possible behaviours of a process are defined by the set of traces it can execute. The *language* $\mathcal{L}(P)$ and the *success language* $\mathcal{M}(P)$ of $P \in \Pi_\mathbf{A}$ are

$$\mathcal{L}(P) = \{\, s \in \mathbf{A}^* \cup \mathbf{A}^*\omega \mid P \overset{s}{\Rightarrow} \} \quad \text{and} \tag{1}$$

$$\mathcal{M}(P) = \{\, s \in \mathbf{A}^*\omega \mid P \overset{s}{\Rightarrow} \} \ . \tag{2}$$

$\mathcal{L}(P)$ contains all complete or incomplete traces that can be executed by a process. This is a prefix-closed language. In contrast, $\mathcal{M}(P)$ contains only traces ending with $\omega$, i.e., only those traces that lead to successful termination.

A process $P = \langle \mathbf{A}, Q, \rightarrow, Q^\circ \rangle$ is *deterministic* if it has at most one initial state, i.e., $|Q^\circ| \leq 1$, if $x \xrightarrow{\alpha} y_1$ and $x \xrightarrow{\alpha} y_2$ always implies $y_1 = y_2$, and if $P$ has no transitions labelled $\tau$. Given a possibly nondeterministic process $P$, the well-known *subset construction* can be used to obtain a language-equivalent deterministic process [14]. More precisely,

$$\det(P) = \langle \mathbf{A}, \mathbb{P}(Q), \rightarrow_{\det}, \{Q^\circ\} \setminus \{\emptyset\} \rangle \ , \tag{3}$$

where $X \xrightarrow{\alpha}_{\det} Y$ for $X, Y \subseteq Q$ if and only if $Y = \{\, y \in Q \mid X \overset{\alpha}{\Rightarrow} y \,\}$ and $Y \neq \emptyset$, is a deterministic process such that $\mathcal{L}(\det(P)) = \mathcal{L}(P)$ and $\mathcal{M}(\det(P)) = \mathcal{M}(P)$.

## 2.3 Synchronous Product

When several processes are running in parallel, lock-step synchronisation in the style of [13] is used. The *synchronous product* $P_1 \parallel P_2$ of two processes $P_1 = \langle \mathbf{A}, Q_1, \rightarrow_1, Q_1^\circ \rangle$ and $P_2 = \langle \mathbf{A}, Q_2, \rightarrow_2, Q_2^\circ \rangle$ both using the action alphabet $\mathbf{A}$, is

$$P_1 \parallel P_2 \;=\; \langle \mathbf{A}, Q_1 \times Q_2, \rightarrow, Q_1^\circ \times Q_2^\circ \rangle \tag{4}$$

where

- $(x_1, x_2) \xrightarrow{\alpha} (y_1, y_2)$, if $\alpha \in \mathbf{A}_\omega$, $x_1 \xrightarrow{\alpha}_1 y_1$, $x_2 \xrightarrow{\alpha}_2 y_2$;
- $(x_1, x_2) \xrightarrow{\tau} (y_1, x_2)$, if $x_1 \xrightarrow{\tau}_1 y_1$;
- $(x_1, x_2) \xrightarrow{\tau} (x_1, y_2)$, if $x_2 \xrightarrow{\tau}_2 y_2$.

It is a well-known property of synchronous composition that the language of the synchronous product is the intersection of the languages of the composed processes, i.e., $\mathcal{L}(P_1 \parallel P_2) = \mathcal{L}(P_1) \cap \mathcal{L}(P_2)$ and $\mathcal{M}(P_1 \parallel P_2) = \mathcal{M}(P_1) \cap \mathcal{M}(P_2)$ [22].

## 2.4 Conflicts

Given a process $P \in \Pi_\mathbf{A}$, it is desirable that every trace in $\mathcal{L}(P)$ can be completed to a trace in $\mathcal{M}(P)$, otherwise $P$ may become unable to terminate. In discrete event systems theory, a process that may become unable to terminate is called *blocking*. This concept becomes more interesting when several processes are running in parallel—in this case the term *conflicting* is used instead [22, 17].

**Definition 1** A process $P \in \Pi_\mathbf{A}$ is *nonblocking*, if for every trace $s \in \mathbf{A}^*$ and every $P' \in \Pi_\mathbf{A}$ such that $P \overset{s}{\Rightarrow} P'$, there exists a continuation $t \in \mathbf{A}^*$ such that $P' \overset{t\omega}{\Rightarrow}$. Otherwise $P$ is *blocking*.

**Definition 2** Two processes $P_1, P_2 \in \Pi_\mathbf{A}$ are *nonconflicting* if $P_1 \parallel P_2$ is nonblocking. Otherwise they are *conflicting*.

In order to be nonblocking, or nonconflicting, it is sufficient that a terminal state *can* be reached in every possible situation. For finite-state systems, this is equivalent to termination under an implicit *strong fairness* assumption stating that "whenever a transition can occur infinitely often, it occurs infinitely often" [3].

**Example 1** Consider the two processes in Fig. 1. For the sake of graphical simplicity, terminal states, i.e., states with outgoing $\omega$-transitions, are shaded in the figures of this paper instead of explicitly showing $\omega$-transitions.

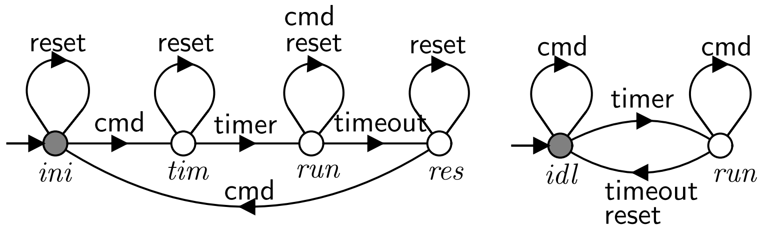

Fig. 1: Two example processes.

Both processes in this example are nonblocking. Although both can theoretically execute an infinite sequence of cmd actions from their *run* states without ever terminating, strong fairness requires that other transitions are taken eventually, leading the system to a terminal state.

In spite of the apparent simplicity of the concept, conflicts are difficult to analyse in a modular way [25]. For purposes of *model checking* [9, 10], the property of two processes being nonconflicting can be expressed by a CTL formula such as

$$\mathbf{AG}\,\mathbf{EF}\ \mathit{terminal\_state}, \tag{5}$$

where *terminal_state* is a propositional formula identifying the states in which both processes are in their terminal states. This formula is neither in $\forall\mathrm{CTL}^*$ nor in $\exists\mathrm{CTL}^*$, which explains why many known abstraction techniques [10] cannot be used for this kind of property.

## 3 The Language of Certain Conflicts

An objective of this research is to find efficient algorithms to determine whether a large system of concurrent processes is blocking or not. The straightforward approach to do this is to construct and examine a synchronous product such as

$$P_1 \parallel P_2 \parallel \cdots \parallel P_n \ . \tag{6}$$

The check is done by exploring all reachable states and checking whether a terminal state can be reached from every reachable state. This can be done using CTL model checking, and models of substantial size can be analysed if the state space is represented symbolically [10]. Yet, the technique always remains limited by the amount of memory available to store representations of the synchronous product.

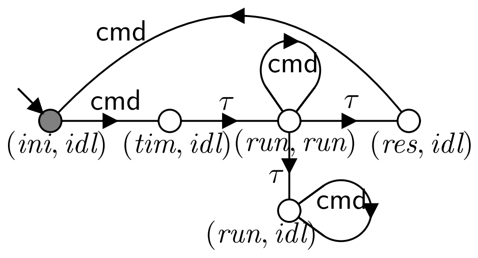

Fig. 2: The synchronous product of the automata in Fig. 1, after hiding the actions timer, timeout, and reset.

As an alternative that avoids the full construction of the synchronous product, *incremental* or *modular* verification try to analyse parts of the system (6) and make conclusions about properties of the entire system. This approach works well for safety properties [1, 6, 2], but it is problematic for nonblocking.

**Example 2** The processes in Fig. 1 are part of a large *central locking system* [16], and these are the only components using the actions timer, timeout, and reset. Therefore, in a first step to analyse the system, the two processes are composed, and the now local actions are *hidden*, i.e., replaced by $\tau$. Figure 2 shows the synchronous product of the two processes in Fig. 1 after hiding these actions. Although each of the processes composed is nonblocking on its own, their synchronous product is blocking, because no terminal state can be reached from state $(run, idl)$. Thus, the *central locking system* includes a blocking subsystem. But this does not necessarily mean that the entire system is blocking. Some other component may disable the action cmd and thereby remove the problem.

Although the presence of a blocking subsystem does not necessarily mean that the entire system is blocking, a blocking subsystem can still be used to detect blocking in certain cases [16]. If it can be shown in the above example that all components of the system can execute the action cmd, this is enough to enable the process in Fig. 2 to reach state $(run, idl)$, so the composed system is blocking.

Therefore, trace cmd is considered as a trace of *certain conflicts* for Fig. 2. Any system that is to be nonconflicting with this process has to prevent cmd from occurring. The following definition first appears in [16].

**Definition 3** For $P \in \Pi_{\mathbf{A}}$, write

$$\textsc{Conf}(P) = \{\, s \in \mathbf{A}_\omega^* \mid \text{For every } T \in \Pi_{\mathbf{A}} \text{ such that } T \overset{s}{\Rightarrow},\ P \parallel T \text{ is blocking} \,\} \ ; \quad (7)$$

$$\textsc{NConf}(P) = \{\, s \in \mathbf{A}_\omega^* \mid \text{There exists } T \in \Pi_{\mathbf{A}} \text{ such that } T \overset{s}{\Rightarrow} \text{ and } P \parallel T \text{ is nonblocking} \,\} \ . \quad (8)$$

$\textsc{Conf}(P)$ is the set of *certain conflicts* of $P$. It contains all traces that, when possible in the environment, necessarily cause blocking. Its complement $\textsc{NConf}(P)$ is the most general behaviour of processes that are to be nonconflicting with $P$. Clearly, if $P$ is nonblocking, then $\textsc{Conf}(P) = \emptyset$ and $\textsc{NConf}(P) = \mathbf{A}_\omega^*$. The set of certain conflicts becomes more interesting for blocking processes.

**Example 3** Let $P$ be the process in Fig. 2. If $P$ is composed with any system that can ever execute action $\mathsf{cmd}$, then $P$ may silently enter the state $(\mathit{run}, \mathit{idl})$ and thus prevent the entire system from ever reaching a terminal state. Therefore, $\textsc{Conf}(P) = \mathsf{cmd}\, \mathbf{A}_\omega^*$.

The set of certain conflicts is introduced in [16], where it is used in combination with an *incremental language inclusion check* [6] to detect conflicts in large systems of synchronised processes. Its relationship to *conflict equivalence* is discussed in [17, 11]. The remainder of this section explains some additional properties of the set of certain conflicts.

Even if a process is nondeterministic, its set of certain conflicts is a *language*. If one trace can lead a process to a blocking state, that trace needs to be disabled to prevent blocking. It is a trace of certain conflicts, no matter how many other nonblocking states can be reached by the same trace.

The language of certain conflicts of a process $P$ is not necessarily a subset of its language $\mathcal{L}(P)$. For example, any trace that starts with $\mathsf{cmd}$ is a trace of certain conflicts of the process in Fig. 2, but not all traces starting with $\mathsf{cmd}$ are accepted by this process. In fact, if $s$ is a trace of certain conflicts, then so is any extension $st$. In consequence, the complement of the set of certain conflicts, the set $\textsc{NConf}(P)$, is prefix-closed.

**Lemma 1** The following holds for all $P \in \Pi_{\mathbf{A}}$:

(i) $\textsc{Conf}(P) = \textsc{Conf}(P)\mathbf{A}_\omega^*$;

(ii) $\textsc{NConf}(P) = \overline{\textsc{NConf}(P)}$.

**Proof.** (i) $\textsc{Conf}(P) \subseteq \textsc{Conf}(P)\mathbf{A}_\omega^*$ is trivial. For the converse inclusion, let $s \in \textsc{Conf}(P)\mathbf{A}_\omega^*$. Then there exists $s' \sqsubseteq s$ such that $s' \in \textsc{Conf}(P)$. To see that $s \in \textsc{Conf}(P)$, let $T \in \Pi_\mathbf{A}$ such that $P \parallel T \overset{s}{\Rightarrow}$. Then clearly $P \parallel T \overset{s'}{\Rightarrow}$, and since $s' \in \textsc{Conf}(P)$, it follows that $P \parallel T$ is blocking. Since $T$ was chosen arbitrarily, it holds that $s \in \textsc{Conf}(P)$.

(ii) $\textsc{NConf}(P) \subseteq \overline{\textsc{NConf}(P)}$ is trivial. For the converse inclusion, let $s \in \overline{\textsc{NConf}(P)}$. Then there exists $t \in \mathbf{A}_\omega^*$ such that $st \in \textsc{NConf}(P)$. Thus, $st \notin \textsc{Conf}(P) = \textsc{Conf}(P)\mathbf{A}_\omega^*$ by (i). It follows that $s \notin \textsc{Conf}(P)$, or equivalently $s \in \textsc{NConf}(P)$. □

Although the set of certain conflicts of a process $P$ is not a subset of the language of $P$, every trace of certain conflicts needs to have an *explanation* in the language of $P$. That is, if $s$ is a trace of certain conflicts, then there is a prefix of $s$, which is accepted by $P$ and which also is a trace of certain conflicts.

**Lemma 2** Let $P \in \Pi_\mathbf{A}$. For every trace $s \in \textsc{Conf}(P)$, there exists a prefix $s' \sqsubseteq s$ such that $s' \in \textsc{Conf}(P) \cap \mathcal{L}(P)$.

**Proof.** First note that, if $\mathcal{L}(P) = \emptyset$ (i.e., $P$ has no initial state), then $P \parallel T$ is nonblocking for every test $T \in \Pi_\mathbf{A}$, i.e., $\textsc{Conf}(P) = \emptyset$. Therefore, $s \in \textsc{Conf}(P)$ implies $\mathcal{L}(P) \neq \emptyset$.

Let $s \in \textsc{Conf}(P)$, and let $s' \sqsubseteq s$ be the longest prefix of $s$ contained in $\mathcal{L}(P)$, i.e., $s' \in \mathcal{L}(P)$ and for all $t \in \mathcal{L}(P)$ such that $t \sqsubseteq s$ it holds that $t \sqsubseteq s'$. Such a trace $s'$ exists because $\mathcal{L}(P) \neq \emptyset$ and therefore $\varepsilon \in \mathcal{L}(P)$.

It is enough to show that $s' \in \textsc{Conf}(P)$. Therefore, let $T \in \Pi_\mathbf{A}$ such that $T \overset{s'}{\Rightarrow}$. Construct a deterministic process $T_s$ such that

$$\mathcal{L}(T_s) = \mathcal{L}(T) \cup \overline{\{s\}} \quad \text{and} \quad \mathcal{M}(T_s) = \mathcal{M}(T)\ . \tag{9}$$

Clearly $T_s \overset{s}{\Rightarrow}$, and since $s \in \textsc{Conf}(P)$, it follows that $P \parallel T_s$ is blocking. Thus, there exists $t \in \mathbf{A}^*$ such that $P \parallel T_s \overset{t}{\Rightarrow} P' \parallel T_s'$ and $\mathcal{M}(P') \cap \mathcal{M}(T_s') = \emptyset$. Note that, if $t \sqsubseteq s$ then also $t \sqsubseteq s'$ by construction of $s'$ (since $P \overset{t}{\Rightarrow}$), and therefore $T \overset{t}{\Rightarrow}$. It follows that $P \parallel T \overset{t}{\Rightarrow} P' \parallel T'$ for some $T' \in \Pi_\mathbf{A}$, and $\mathcal{M}(T') = \mathcal{M}(T_s')$ by construction of $T_s$. Thus, $\mathcal{M}(P') \cap \mathcal{M}(T') = \mathcal{M}(P') \cap \mathcal{M}(T_s') = \emptyset$, i.e, $P \parallel T$ is blocking. Since $T$ was chosen arbitrarily, it follows that $s' \in \textsc{Conf}(P)$. □

## 4 Algorithm

While the language of certain conflicts provides a powerful tool when verifying nonblocking, it is only approximated by the method in [11]. A language-based fixpoint construction to compute the language of certain conflicts of a deterministic process is outlined in [16], but no analysis of correctness or complexity is given, and it is unclear how the method is applicable to nondeterministic processes. This section develops a state-based description of an algorithm to compute the language of certain conflicts for any finite-state nondeterministic process, proves its correctness, and analyses its space and time complexity.

### 4.1 Introduction and Example

A first approach to find the language of certain conflicts of a process is to identify all *blocking states*, i.e., all the states from which no terminal state is reachable. Any trace that leads to a blocking state definitely is a trace of certain conflicts. But these are not the only certain conflicts.

Consider process $P$ in Fig. 3. State 4 is blocking and all traces leading to it, i.e., all traces in $\alpha\{\alpha, \beta\}^*\gamma$, are traces of certain conflicts. Furthermore, $P$ may enter state 7 after execution of trace $\alpha\alpha\beta$, where the only possibility to terminate is by executing action $\gamma$. This requires the execution of $\alpha\alpha\beta\gamma$, which is a trace of certain conflicts as explained above. If some other process accepts $\alpha\alpha\beta$, in order to be nonconflicting with $P$, it also needs to accept $\alpha\alpha\beta\gamma$, but by doing so it already is conflicting with $P$. In consequence, $\alpha\alpha\beta$ also is a trace of certain conflicts.

The example also shows that the language of certain conflicts of a process $P$ cannot always be represented using only the states of $P$. Clearly, states 4 and 7 are states of certain conflicts. This entails that the $\beta$-transition from state 6 to 7 must be disabled to avoid certain conflicts. Thus, after execution of trace $\alpha\alpha$, action $\beta$ must be disabled. This includes the removal of some $\beta$-transitions originating from state 1 and 2, but it needs to be distinguished whether these states have been entered by executing $\alpha\alpha$ or some other trace.

Therefore, to compute the set of certain conflicts accurately, states may need to be split depending on the history of actions by which they have been reached. This can be achieved using subset construction [14], by constructing the synchronous product $P \parallel \det(P)$. In this way, every state of $P$ is paired with the set of alternative states $P$ may be in.

This idea leads to the algorithm shown in Fig. 4. After construction of

$P \,\|\, \det(P)$, all blocking states are identified and removed from the composed process. In addition, if a composed state $(r, R)$ is removed, where $r$ is a state of $P$ and $R$ is the set of alternative states $P$ may be in, then all other composed states that involve this set $R$ of alternatives are also removed.

Consider again process $P$ in Fig. 3. The figure also shows its determinised version $\det(P)$ and the composition $P \,\|\, \det(P)$. For brevity, state sets from subset construction are written as 123, e.g., to represent the set $\{1, 2, 3\}$. States $(4, 4)$ and $(4, 45)$ are blocking in $P \,\|\, \det(P)$. Therefore, all state pairs with 4 or 45 in the second component are removed in step 6 of the algorithm. This leads to the removal of $(5, 45)$. In consequence, $(7, 157)$ becomes blocking, and three more states $(1, 157)$, $(5, 157)$, and $(7, 157)$ are removed in the next iteration. The resultant process is nonblocking and can execute only traces that are *not* certain conflicts. The states removed are crossed out in the figure.

## 4.2 Proof of Correctness

In this section, the correctness of the algorithm in Fig. 4 is established formally. This is done using a fixpoint operator that describes the intermediate results in each step of the algorithm.

**Definition 4** Let $P = \langle \mathbf{A}, Q, \rightarrow, Q^\circ \rangle$ and $X \subseteq Q$. The *restriction* of $P$ by $X$ is

$$P_{|X} = \langle \mathbf{A}, X, \rightarrow_{|X}, Q^\circ \cap X \rangle \ , \tag{10}$$

where

$$\rightarrow_{|X} = \{\, (x, \alpha, y) \mid x \xrightarrow{\alpha} y \text{ and } x, y \in X \,\} \ . \tag{11}$$

**Definition 5** For $P = \langle \mathbf{A}, Q, \rightarrow, Q^\circ \rangle$, define the mapping

$$\begin{aligned} \Theta_P : \ \mathbb{P}(\mathbb{P}(Q)) \ &\rightarrow \ \mathbb{P}(\mathbb{P}(Q)); \qquad (12) \\ X \ &\mapsto \ \{\, R \in X \mid \text{For all } r \in R, \text{ there exists } t \in \mathbf{A}^* \text{ such that} \\ &\qquad (r, R) \overset{t\omega}{\Rightarrow} \text{ in } P \,\|\, \det(P)_{|X} \,\} \ . \end{aligned}$$

The function $\Theta_P$ operates on subsets of the state set of $\det(P)$ in a way that captures the behaviour of the algorithm in Fig. 4. When applied repeatedly, starting with the entire state set as follows,

$$X_0 = \mathbb{P}(Q) \qquad \text{and} \qquad X_{i+1} = \Theta_P(X_i) \ , \tag{13}$$

the process $P_i$ in step $i$ of the algorithm is equal to $P \,\|\, \det(P)_{|X_i}$. To answer the question whether the iteration terminates, it is first confirmed that the operator $\Theta_P$ is *monotonic*.

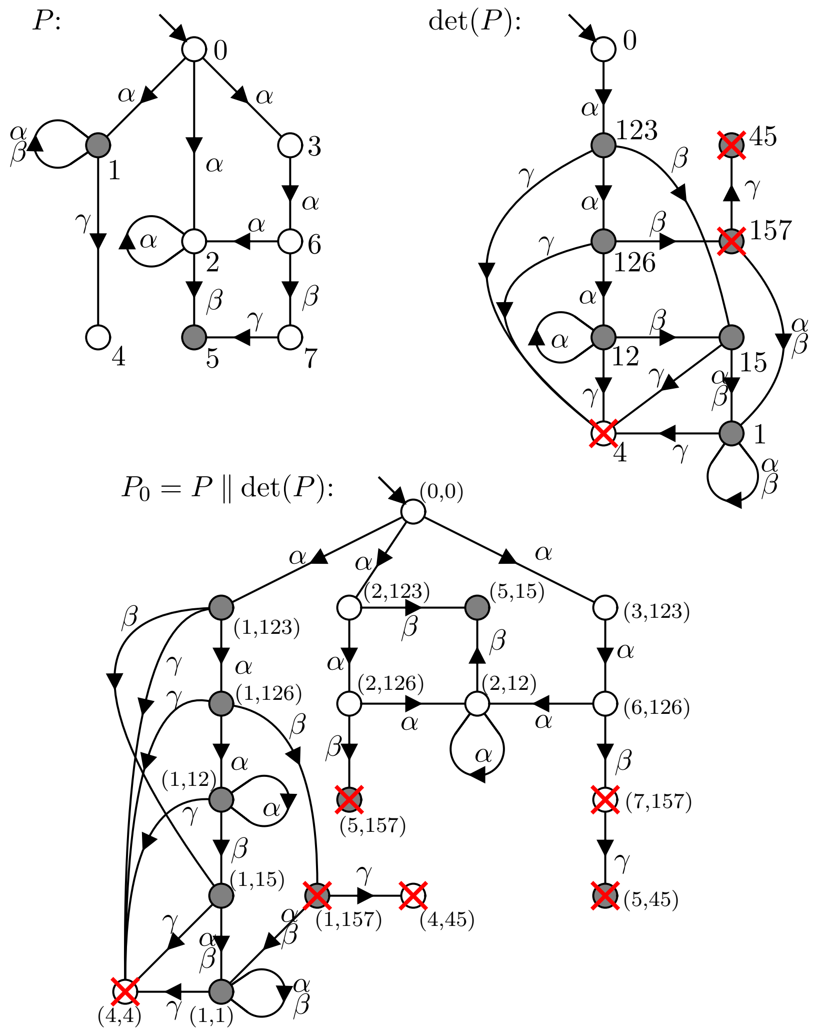

Fig. 3: Working example showing the computation of certain conflicts of process $P$. After removal of the crossed out states, the languages of $\det(P)$ and $P \parallel \det(P)$ both are equal to $\text{NC}\textsc{onf}(P) \cap \mathcal{L}(P)$.

```
1. Input $P = \langle \mathbf{A}, Q, \rightarrow, Q^\circ \rangle$.
2. Apply subset construction to obtain $\det(P)$.
3. Set $i := 0$ and $P_0 := P \parallel \det(P)$.
   (The state space of $P_0$ consists of pairs $(r, R)$ with $r \in Q$ and $R \subseteq Q$.)
4. while $P_i$ is blocking do
5.    Let $B_i$ be the set of blocking states in $P_i$.
6.    Construct $P_{i+1}$ from $P_i$ by removing all states $(r, R)$ such that
      $(r', R) \in B_i$ for some $r' \in Q$.
7.    Set $i := i + 1$.
8. end
   (The language of $P_i$ now equals $\mathrm{NC\small{ONF}}(P) \cap \mathcal{L}(P)$.)
```

Fig. 4: Algorithm to compute $\mathrm{NC\small{ONF}}(P) \cap \mathcal{L}(P)$.

**Lemma 3** Let $P = \langle \mathbf{A}, Q, \rightarrow, Q^\circ \rangle$ and $X, Y \subseteq Q$. If $X \subseteq Y$ then $\Theta_P(X) \subseteq \Theta_P(Y)$.

**Proof.** Let $X, Y \subseteq Q$ such that $X \subseteq Y$, and assume $R \in \Theta_P(X)$. It immediately follows that $R \in X \subseteq Y$. Furthermore, for $r \in R$, by assumption $R \in \Theta_P(X)$ there exists $t \in \mathbf{A}^*$ such that $(r, R) \stackrel{t\omega}{\Rightarrow}$ in $P \parallel \det(P)_{|X}$. Given $X \subseteq Y$, it then also holds that $(r, R) \stackrel{t\omega}{\Rightarrow}$ in $P \parallel \det(P)_{|Y}$. This implies $R \in \Theta_P(Y)$. □

As $\Theta_P$ is a monotonic operator on the lattice of subsets of the state set of $\det(P)$, it follows by the Knaster-Tarski theorem [24] that $\Theta_P$ has a greatest fixpoint $\hat{X} = \mathrm{gfp}(\Theta_P)$. That is, $\hat{X}$ is a fixpoint of $\Theta_P$, i.e., $\Theta_P(\hat{X}) = \hat{X}$, and every other fixpoint of $\Theta_P$ is a subset of $\hat{X}$. If the state set is finite, the iteration (13) converges on this fixpoint in a finite number of steps,

$$\mathrm{gfp}\,\Theta_P \;=\; \hat{X} \;=\; X_n \qquad \text{for some } n \in \mathbb{N}_0 \ . \tag{14}$$

Unfortunately, $\Theta_P$ is not continuous. If the state set is infinite, the greatest fixpoint still exists, but it may be not a limit of the sequence (13).

It is next established that the greatest fixpoint of $\Theta_P$ does indeed describe the set of certain conflicts. More precisely, the greatest fixpoint $\hat{X}$ satisfies

$$\mathcal{L}(\det(P)_{|\hat{X}}) \;=\; \mathrm{NC\small{ONF}}(P) \cap \mathcal{L}(P) \ . \tag{15}$$

To prove this result, it is not enough to follow the standard approach show that exactly the fixpoints of $\Theta_P$ lead to subsets of the language $\mathrm{NC\small{ONF}}(P) \cap$

$\mathcal{L}(P)$, because it also needs to be established that this language can be represented using a subautomaton of $\det(P)$. Still, the first step of the proof is to establish that every fixpoint of $\Theta_P$ leads to a sublanguage of $\text{NC}\textsc{onf}(P)$.

**Lemma 4** Let $P = \langle \mathbf{A}, Q, \rightarrow, Q^\circ \rangle$, and let $X \subseteq Q$ be a pre-fixpoint of $\Theta_P$, i.e., $X \subseteq \Theta_P(X)$. Then $\mathcal{L}(\det(P)_{|X}) \subseteq \text{NC}\textsc{onf}(P)$.

**Proof.** Let $X \subseteq \Theta_P(X)$, and first note that $P \parallel \det(P)_{|X}$ is nonblocking. To see this, let $P \parallel \det(P)_{|X} \overset{s}{\Rightarrow} (r, R)$. Then $R \in X \subseteq \Theta_P(X)$ and since $P \overset{s}{\Rightarrow} r$ and $\det(P) \overset{s}{\Rightarrow} R$ also $r \in R$. By definition of $\Theta_P$, there exists $t \in \mathbf{A}^*$ such that $(r, R) \overset{t\omega}{\Rightarrow}$ in $P \parallel \det(P)_{|X}$. Therefore, $P \parallel \det(P)_{|X}$ is nonblocking.

Now let $u \in \mathcal{L}(\det(P)_{|X})$, According to the above, $\det(P)_{|X}$ is a process accepting $u$ that is nonconflicting with $P$, i.e., $u \in \text{NC}\textsc{onf}(P)$. □

For the second inclusion of (15), it is proved that the iteration (13) only produces supersets of the language $\text{NC}\textsc{onf}(P) \cap \mathcal{L}(P)$. This is done inductively using the following lemma. Part (ii) is only needed for the infinite-state case.

**Lemma 5** Let $P = \langle \mathbf{A}, Q, \rightarrow, Q^\circ \rangle$.

(i) For $X \subseteq Q$, if $\text{NC}\textsc{onf}(P) \cap \mathcal{L}(P) \subseteq \mathcal{L}(\det(P)_{|X})$, then $\text{NC}\textsc{onf}(P) \cap \mathcal{L}(P) \subseteq \mathcal{L}(\det(P)_{|\Theta_P(X)})$.

(ii) Let $(X_i)_{i \in I}$ be a family of subsets of $Q$, and let $X = \bigcap_{i \in I} X_i$. If for each $i \in I$ it holds that $\text{NC}\textsc{onf}(P) \cap \mathcal{L}(P) \subseteq \mathcal{L}(\det(P)_{|X_i})$, then $\text{NC}\textsc{onf}(P) \cap \mathcal{L}(P) \subseteq \mathcal{L}(\det(P)_{|X})$.

**Proof.** (i) Let $\text{NC}\textsc{onf}(P) \cap \mathcal{L}(P) \subseteq \mathcal{L}(\det(P)_{|X})$ and $s \in \text{NC}\textsc{onf}(P) \cap \mathcal{L}(P)$. By assumption, $s \in \mathcal{L}(\det(P)_{|X})$. Write $s = \alpha_1 \ldots \alpha_n$ and

$$Q^\circ = R_0 \xrightarrow{\alpha_1} R_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_n} R_n \quad \text{in } \det(P)_{|X} \; . \tag{16}$$

It is enough to show $R_k \in \Theta_P(X)$ for $k = 0, \ldots, n$.

To see this, first note that $R_k \in X$ since $\det(P)_{|X} \rightarrow R_k$. Second let $s_k = \alpha_1 \ldots \alpha_k$. Then $s_k \sqsubseteq s$ and thus

$$s_k \in \overline{\text{NC}\textsc{onf}(P) \cap \mathcal{L}(P)} \subseteq \overline{\text{NC}\textsc{onf}(P)} = \text{NC}\textsc{onf}(P) \tag{17}$$

by lemma 1. Therefore, $s_k \in \text{NC}\textsc{onf}(P)$ and there exists $T \in \Pi_{\mathbf{A}}$ such that $T \overset{s_k}{\Rightarrow}$ and $P \parallel T$ is nonblocking. Now let $r \in R_k$. Then $P \parallel T \overset{s_k}{\Rightarrow} (r, r_T)$ for some state $r_T$ of $T$, and since $P \parallel T$ is nonblocking, there exists

$t \in \mathbf{A}^*$ such that $(r, r_T) \overset{t\omega}{\Rightarrow}$. Then $T \overset{s_k t\omega}{\Longrightarrow}$ and $s_k t\omega \in \text{NC}\textsc{onf}(P)$ since $P \parallel T$ is nonblocking, and $P \overset{s_k t\omega}{\Longrightarrow}$, i.e., $s_k t\omega \in \mathcal{L}(P)$. By the assumption $\text{NC}\textsc{onf}(P) \cap \mathcal{L}(P) \subseteq \mathcal{L}(\det(P)_{|X})$, it follows that $s_k t\omega \in \mathcal{L}(\det(P)_{|X})$. Since $\det(P)$ is deterministic, $P \parallel \det(P)_{|X} \overset{s_k}{\Rightarrow} (r, R_k) \overset{t\omega}{\Rightarrow}$ and thus $R_k \in \Theta_P(X)$ for all $k = 0, \ldots, n$.

(ii) Let $X = \bigcap_{i \in I} X_i$, and let $s \in \text{NC}\textsc{onf}(P) \cap \mathcal{L}(P) \subseteq \mathcal{L}(\det(P)_{|X_i})$ for each $i \in I$. Since $\det(P)$ is deterministic, there exists only one path accepting $s$ in $\det(P)$, say $Q^\circ = R_0 \overset{\alpha_1}{\rightarrow} \cdots \overset{\alpha_n}{\rightarrow} R_n$. This path must exist in each $\det(P)_{|X_i}$, i.e., $R_0, \ldots, R_n \in X_i$ for each $i \in I$. Then $R_0, \ldots, R_n \in \bigcap_{i \in I} X_i = X$, which implies $s \in \mathcal{L}(\det(P)_{|X})$. □

**Proposition 6** Let $P \in \Pi_{\mathbf{A}}$, and let $\hat{X} = \operatorname{gfp} \Theta_P$. Then

$$\mathcal{L}(\det(P)_{|\hat{X}}) \;=\; \text{NC}\textsc{onf}(P) \cap \mathcal{L}(P) \;. \tag{18}$$

**Proof.** Since the greatest fixpoint is a pre-fixpoint of $\Theta_P$, it follows from lemma 4 that $\mathcal{L}(\det(P)_{|\hat{X}}) \subseteq \text{NC}\textsc{onf}(P)$. Then, since $\mathcal{L}(\det(P)_{|\hat{X}}) \subseteq \mathcal{L}(\det(P)) = \mathcal{L}(P)$, it follows that $\mathcal{L}(\det(P)_{|\hat{X}}) \subseteq \text{NC}\textsc{onf}(P) \cap \mathcal{L}(P)$.

For the converse inclusion, given the monotonicity of $\Theta_P$ (lemma 3) it follows from the Knaster-Tarski theorem [24] that $\hat{X} = \operatorname{gfp} \Theta_P = \Theta_P^\nu(\mathbb{P}(Q))$ for some ordinal $\nu$. Noting that $\mathcal{L}(\det(P)_{|\mathbb{P}(Q)}) = \mathcal{L}(\det(P)) = \mathcal{L}(P) \supseteq \text{NC}\textsc{onf}(P) \cap \mathcal{L}(P)$, it follows by transfinite induction from lemma 5 that $\mathcal{L}(\det(P)_{|\hat{X}}) = \mathcal{L}(\det(P)_{|\Theta_P^\nu(\mathbb{P}(Q))}) \supseteq \text{NC}\textsc{onf}(P) \cap \mathcal{L}(P)$. □

This confirms that the iteration (13) indeed converges in the finite-state case and yields the language $\text{NC}\textsc{onf}(P) \cap \mathcal{L}(P)$. As this is not the actual set of certain conflicts, the following final proposition shows how to obtain the set of certain conflicts from the algorithm result.

**Proposition 7** Let $P \in \Pi_{\mathbf{A}}$, and let $\mathcal{N} = \text{NC}\textsc{onf}(P) \cap \mathcal{L}(P)$. Then

$$\textsc{Conf}(P) \;=\; (\mathcal{L}(P) \setminus \mathcal{N})\mathbf{A}_\omega^* \;. \tag{19}$$

**Proof.** First, let $s \in \textsc{Conf}(P)$. By lemma 2, there exists $s' \sqsubseteq s$ such that $s' \in \textsc{Conf}(P) \cap \mathcal{L}(P)$. Then $s' \in \mathcal{L}(P)$ and $s' \notin \text{NC}\textsc{onf}(P) \supseteq \mathcal{N}$, which implies $s' \in \mathcal{L}(P) \setminus \mathcal{N}$ and $s \in (\mathcal{L}(P) \setminus \mathcal{N})\mathbf{A}_\omega^*$.

Second, let $s \in (\mathcal{L}(P) \setminus \mathcal{N})\mathbf{A}_\omega^*$. Then there exists $s' \sqsubseteq s$ such that $s' \in \mathcal{L}(P) \setminus \mathcal{N}$. Thus, $s' \in \mathcal{L}(P)$ and $s' \notin \mathcal{N} = \text{NC}\textsc{onf}(P) \cap \mathcal{L}(P)$. Since $s \in \mathcal{L}(P)$, the latter means that $s' \notin \text{NC}\textsc{onf}(P)$, i.e., $s' \in \textsc{Conf}(P)$ and therefore $s \in \textsc{Conf}(P)\mathbf{A}_\omega^* = \textsc{Conf}(P)$ by lemma 1. □

This result implies that, after completion of the algorithm in Fig. 4, the set of certain conflicts can be constructed directly as the set of all traces leading to states of $\det(P)$ deleted by the algorithm, plus all possible extensions of such traces.

## 4.3 Complexity

This section briefly discusses the complexity of the algorithm to compute the set of certain conflicts in the finite-state case and shows that its space and time complexity is exponential in the number of states of the given process. If the result is to be given as a deterministic finite-state machine, this also is the lower limit of complexity.

Given an input process $P = \langle \mathbf{A}, Q, \rightarrow, Q^\circ \rangle$, the algorithm in Fig. 4 needs to search the state space of $P_0 = P \parallel \det(P)$, so its space complexity is determined by the number of states of that process. It is

$$O(|Q| \cdot 2^{|Q|}) \ . \tag{20}$$

For the time complexity, note that each iteration in the algorithm involves the computation of the set of blocking states of a restriction of $P_0 = P \parallel \det(P)$. This can be done using standard graph search algorithms, visiting all transitions of $P_0$ at most once. For each action and each state, there may be up to $|Q|$ outgoing transitions in $P$, but at most one outgoing transition in the deterministic process $\det(P)$. Therefore, $P_0$ can have up to $|\mathbf{A}||Q|$ outgoing transitions from each of its up to $|Q| \cdot 2^{|Q|}$ states, in total up to $|\mathbf{A}||Q|^2 \cdot 2^{|Q|}$ transitions. Each iteration except the last involves the removal of at least one state of $\det(P)$, so there can be at most $2^{|Q|} + 1$ iterations. This gives a worst-case time complexity of

$$O(|\mathbf{A}| \cdot |Q|^2 \cdot 4^{|Q|}) \ . \tag{21}$$

To address the question whether there can be better algorithms to compute the set of certain conflicts, it is worth noting that by proposition 7 the language of certain conflicts of $P$ can be represented as a deterministic process using a subset of the states of $\det(P)$. Thus, the language $\textsc{Conf}(P)$ can be represented as a deterministic process with at most $2^{|Q|}$ states, which is only slightly less than (20).

On the other hand, if the process $P$ is nonblocking then $\textsc{NConf}(P) = \mathbf{A}_\omega^*$, so the language computed by the algorithm is $\textsc{NConf}(P) \cap \mathcal{L}(P) = \mathcal{L}(P)$, and it is well-known that the worst-case for a deterministic automaton accepting the same language as a given nondeterministic process $P$ has

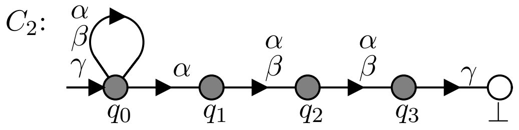

Fig. 5: Example to show that the language of certain conflicts may have a smallest deterministic recogniser of exponential size.

an exponential number of states [14]. The example used to establish this complexity bound for language determinisation can be modified to apply to the language of certain conflicts.

**Example 4** Consider a family of processes $C_k = \langle \mathbf{A}, Q_k, \rightarrow_k, \{q_0\}\rangle$, $k \in \mathbb{N}_0$, where $\mathbf{A} = \{\alpha, \beta, \gamma\}$, $Q_k = \{q_0, \ldots, q_{k+1}, \perp\}$, and $\rightarrow_k$ consists of transitions $q_0 \xrightarrow{\sigma}_k q_0$ for $\sigma \in \mathbf{A}$; $q_i \xrightarrow{\alpha}_k q_{i+1}$ for $i = 0, \ldots, k$; $q_i \xrightarrow{\beta}_k q_{i+1}$ for $i = 1, \ldots, k$; $q_{k+1} \xrightarrow{\gamma}_k \perp$; and $q_i \xrightarrow{\omega}_k \perp$ for $i = 0, \ldots, k+1$. Process $C_2$ is shown in Fig. 5.

The set of certain conflicts of $C_k$ is $\textsc{Conf}(C_k) = \mathbf{A}^*\alpha\{\alpha,\beta\}^k\gamma\mathbf{A}^*_\omega$. That is, certain conflicts are reached if action $\gamma$ occurs when $\alpha$ has occurred $k+1$ steps before. While this language has a nondeterministic recogniser with $k+2$ states, it is known that it cannot have a deterministic recogniser with less than $2^k$ states [14].

This shows that an exponential number of states for deterministic representations of the set of certain conflicts can be necessary. There may be smaller nondeterministic representations, and it is unknown to the author whether they are guaranteed to exist or at what computational cost they can be found.

## 5 Simplifying Subsystems

This section shows one way how the language of certain conflicts in general, and the results of the algorithm of the previous section in particular, can be used for compositional nonblocking verification [11]. In compositional verification, parts of system of interacting components such as (6) are replaced by a simpler, yet *equivalent*, process. The appropriate notion of equivalence for nonblocking verification is *conflict equivalence* as introduced in [17].

**Definition 6** Processes $P_1, P_2 \in \Pi_\mathbf{A}$ are called *conflict equivalent*, written $P_1 \simeq_{\text{conf}} P_2$, if for every process $T \in \Pi_\mathbf{A}$, it holds that $P_1 \parallel T$ is nonblocking if and only of $P_2 \parallel T$ is nonblocking.

The properties of conflict equivalence are discussed in [17]. The equivalence is coarser than *observation equivalence* [18] and different from *language equivalence*, *failures equivalence* [13] and *failure trace equivalence* [21, 15]. The process-algebraic equivalence most similar to it is *fair testing equivalence* [7, 19].

Given a process $P$, there are several ways to obtain conflict equivalent abstractions [11]. The language of certain conflicts provides a powerful tool in this regard, because it identifies all potentially blocking behaviours, and these can all be treated alike. Abstraction along this line is suggested in [16] for deterministic processes, and the following proposition extends the idea to arbitrary nondeterministic processes.

**Proposition 8** Let $P \in \Pi_{\mathbf{A}}$. Construct a deterministic process $C_P \in \Pi_{\mathbf{A}}$ such that

$$\begin{aligned} \mathcal{L}(C_P) &= \textrm{NC{\scriptsize ONF}}(P) \cup (\textrm{NC{\scriptsize ONF}}(P)\mathbf{A} \cap \mathcal{L}(P)) \cup \{\varepsilon\} \; ; && (22) \\ \mathcal{M}(C_P) &= \textrm{NC{\scriptsize ONF}}(P) \cap \mathcal{M}(P) \; . && (23) \end{aligned}$$

Then $P \simeq_{\mathrm{conf}} P \parallel C_P$.

**Proof.** Let $T \in \Pi_{\mathbf{A}}$ be an arbitrary process.

First, let $P \parallel T$ be nonblocking, and let $P \parallel C_P \parallel T \overset{s}{\Rightarrow} P' \parallel C'_P \parallel T'$. Since $P \parallel T$ is nonblocking, there exists $t \in \mathbf{A}^*$ such that $P' \parallel T' \overset{t\omega}{\Rightarrow}$. Then $st\omega \in \mathcal{M}(P)$ and $st\omega \in \mathcal{M}(T)$, and since $P \parallel T$ is nonblocking, it follows that $st\omega \in \textrm{NC{\scriptsize ONF}}(P) \cap \mathcal{M}(P) = \mathcal{M}(C_P)$. Since $C_P$ is deterministic, it also holds that $P \parallel C_P \parallel T \overset{s}{\Rightarrow} P' \parallel C'_P \parallel T' \overset{t\omega}{\Rightarrow}$, i.e., $P \parallel C_P \parallel T$ is nonblocking.

Conversely, let $P \parallel C_P \parallel T$ be nonblocking, and let $P \parallel T \overset{s}{\Rightarrow} P' \parallel T'$. Let $s' \sqsubseteq s$ be the longest prefix of $s$ such that $s' \in \mathcal{L}(C_P)$. Note that $s'$ exists because $\varepsilon \in \mathcal{L}(C_P)$. Then $C_P \parallel T \overset{s'}{\Rightarrow}$, and since $P \parallel (C_P \parallel T)$ is nonblocking, it follows that $s' \in \textrm{NC{\scriptsize ONF}}(P)$. Then it holds that $s' = s$: if $s'$ is a proper prefix of $s$, then $s'\alpha \sqsubseteq s$ for some $\alpha \in \mathbf{A}$, and given $s \in \mathcal{L}(P)$ it follows that $s'\alpha \in \textrm{NC{\scriptsize ONF}}(P)\mathbf{A} \cap \mathcal{L}(P) \subseteq \mathcal{L}(C_P)$ by (22), but $s'$ was chosen to be the longest prefix of $s$ such that $s' \in \mathcal{L}(C_P)$. Therefore, $s = s'$ and thus $C_P \overset{s}{\Rightarrow}$, which implies $P \parallel C_P \parallel T \overset{s}{\Rightarrow} P' \parallel C'_P \parallel T'$ for some $C'_P \in \Pi_{\mathbf{A}}$. Since $P \parallel C_P \parallel T$ is nonblocking, there exists $t \in \mathbf{A}^*$ such that $P' \parallel C'_P \parallel T' \overset{t\omega}{\Rightarrow}$. Then it follows that $P' \parallel T' \overset{t\omega}{\Rightarrow}$, i.e., $P \parallel T$ is nonblocking. $\square$

According to proposition 8, an arbitrary process $P$ can be replaced by the conflict equivalent abstraction $P \parallel C_P$, effectively merging all traces of certain conflicts into a single state.

The addition of the empty trace in (22) is only needed to cover the case $\text{NC}\textsc{onf}(P) = \emptyset$. In this case, $\mathcal{L}(C_P) = \{\varepsilon\}$ after addition of the empty trace, and $\mathcal{M}(C_P) = \emptyset$, ensuring that $C_P$ is blocking in combination with any other nonempty process.

The abstraction $P \parallel C_P$ in proposition 8 can be obtained directly from the results of the algorithm in Fig. 4. It is enough to add a new blocking state $\perp$ to the result $P_n = P \parallel \det(P)_{|X_n}$ and redirect all transitions leading to the states deleted from $P \parallel \det(P)$ to $\perp$.

## 6 Conclusions

An algorithm to compute the *language of certain conflicts* of a nondeterministic finite-state process is presented and analysed, and it is shown how the result of this algorithm can be used to compute conflict-preserving abstractions of a nondeterministic process by identifying all behaviours that may lead to livelock or deadlock in some environment. Despite its exponential complexity, this algorithm can improve on existing methods that merely approximate the language of certain conflicts: by simplifying only small components in a large verification problem, a substantial reduction of the overall state space can be achieved.

## References

[1] K. Åkesson, H. Flordal, and M. Fabian. Exploiting modularity for synthesis and verification of supervisors. In *Proc. 15th IFAC World Congress on Automatic Control*, Barcelona, Spain, 2002.

[2] Rajeev Alur, Luca de Alfaro, Thomas A. Henzinger, and Freddy Y. C. Mang. Automating modular verification. In *Proc. Int. Conf. Concurrency Theory, CONCUR '99*, LNCS, pages 82–97. Springer, 1999.

[3] A. Arnold. *Finite Transitions Systems: Semantics of Communicating Systems.* Prentice-Hall, 1994.

[4] S. Balemi, G. J. Hoffmann, P. Gyugyi, H. Wong-Toi, and G. F. Franklin. Supervisory control of a rapid thermal multiprocessor. *IEEE Trans. Automat. Contr.*, 38(7):1040–1059, July 1993.

[5] Bertil Brandin and François Charbonnier. The supervisory control of the automated manufacturing system of the AIP. In *Proc. Rensse-*

*laer's 4th Int. Conf. Computer Integrated Manufacturing and Automation Technology*, pages 319–324, Troy, NY, USA, 1994.

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

[7] Ed Brinksma, Arend Rensink, and Walter Vogler. Fair testing. In Insup Lee and Scott A. Smolka, editors, *Proc. 6th Int. Conf. Concurrency Theory, CONCUR '95*, volume 962 of *LNCS*, pages 313–327, Philadelphia, PA, USA, 1995. Springer.

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

[9] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Programming Languages and Systems*, 8(2):244–263, April 1986.

[10] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking.* MIT Press, 1999.

[11] Hugo Flordal and Robi Malik. Modular nonblocking verification using conflict equivalence. In *Proc. 8th Int. Workshop on Discrete Event Systems, WODES '06*, pages 100–106, Ann Arbor, MI, USA, July 2006.

[12] Cédric Fournet, Tony Hoare, Sriram K. Rajamani, and Jakob Rehof. Stuck-free conformance. In *Proc. 16th Int. Conf. Computer Aided Verification, CAV 2004*, volume 3114 of *LNCS*, pages 242–254, Boston, MA, USA, July 2004. Springer.

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

[14] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 2001.

[15] R. Kumar and M. A. Shayman. Non-blocking supervisory control of nondeterministic discrete event systems. In *Proc. American Control Conf.*, pages 1089–1093, Baltimore, MD, USA, 1994.

[16] 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.

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

[18] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer, 1980.

[19] V. Natarajan and Rance Cleaveland. Divergence and fair testing. In *Proc. 22nd Int. Colloquium on Automata, Languages, and Programming, ICALP '95*, pages 648–659, 1995.

[20] Patrícia N. Pena, José E. R. Cury, and Stéphane Lafortune. New results on testing modularity of local supervisors using abstractions. In *Proc. 11th IEEE Int. Conf. Emerging Technologies and Factory Automation, ETFA '06*, pages 950–956, Prague, Czech Republic, September 2006.

[21] Iain Phillips. Refusal testing. *Theoretical Comput. Sci.*, 50:241–284, 1987.

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

[23] A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall, 1997.

[24] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5(2):285–309, 1955.

[25] K. C. Wong, J. G. Thistle, R. P. Malhame, and H.-H. Hoang. Supervisory control of distributed systems: Conflict resolution. *Discrete Event Dyn. Syst.*, 10:131–186, 2000.