ON THE SET OF CERTAIN CONFLICTS OF A GIVEN LANGUAGE

Robi Malik

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

Abstract: Two concurrent processes are said to be in conflict if they can get trapped in a situation where they both are waiting or running endlessly, forever unable to complete their common task. In the design of reactive systems, this is a common fault which can be very subtle and hard to detect. This paper studies conflicts in more detail and characterises the most general set of behaviours of a process which certainly leads to a conflict when accepted by another process running in parallel. It shows how this set of certain conflicts can be used to simplify the automatic detection of conflicts and thus the verification of reactive systems.

Keywords: Discrete event systems, large-scale systems, formal verification, deadlock, termination.

1. INTRODUCTION

Conflicts are a common fault in the design of concurrent programs, and have long been studied in the context of discrete event systems (Dietrich *et al.*, 2002; Ramadge and Wonham, 1989; Wong *et al.*, 2000). Two concurrent processes are in conflict if they can reach a state, from which no terminal state can be reached anymore. 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.

Since discrete event systems are used to model complex, safety-critical systems (Brandin and Charbonnier, 1994; Leduc and Wonham, 1995; Malik, 2003), tools to detect conflicts in very large systems are needed. Existing model checking techniques (Clarke *et al.*, 1999) can be used, but are limited by the state explosion problem. This paper studies the special nature of conflicts in deeper detail and proposes a method to facilitate the automatic detection of conflicts in modular systems.

A common way of fighting state explosion is to exploit modular system structures (Leduc *et al.*, 2000; Wong and Wonham, 1998). Often it suffices to analyse only subsystems of a large system to establish that the entire system satisfies a property of interest (Åkesson *et al.*, 2002; Brandin *et al.*, 2004). Yet, the nature of conflict is more sophisticated: even if each individual subsystem is free from conflict, the system composed from these subsystems may be in conflict (Wong *et al.*, 2000).

In this situation, it is possible to use abstractions of subsystems that preserve possible conflicts with other subsystems. Structural (Leduc *et al.*, 2000; Lee and Wong, 2002) and priority-based (Wong *et al.*, 2000) approaches have been suggested. These techniques give some insight how systems can be designed to be free from conflict, but they rely on design carried out by users, and therefore cannot be performed automatically. Other techniques have been discovered in the context of model checking (Clarke *et al.*, 1999; Dams *et al.*, 1997), but they are not tailored for the special needs of conflict analysis and therefore are limited in power or cannot be applied directly.

Recently, a heuristic algorithm for detecting a special case of conflicts has been developed (Malik, 2003). The experience gained from this algorithm can be used to find abstractions of discrete event systems that preserve all possible conflicts. This paper introduces the set of certain conflicts of a language, a set of behaviours that can always be treated specially for verification purposes, considerably improving performance.

Section 2 introduces the necessary notations from supervisory control theory (Ramadge and Wonham, 1989; Wonham, 1999) including conflicts. Section 3 defines the set of certain conflicts and shows how it can simplify the task of checking a complex system for conflicts. Section 4 presents a fixed point algorithm to compute the set of certain conflicts, and section 5 gives some concluding remarks.

2. NOTATION AND PRELIMINARIES

2.1 Languages

Event sequences and languages are a simple means to describe discrete system behaviours. Their basic building blocks are *events*, which are taken from a finite *alphabet* Σ. Then, Σ^* denotes the set of all finite *strings* of the form $\sigma_1\sigma_2\cdots\sigma_k$ of events from Σ, including the *empty string* ε. A *language* over Σ is any subset $\mathcal{L} \subseteq \Sigma^*$.

The *catenation* of two strings $s, t \in \Sigma^*$ is written as st. A string $t \in \Sigma^*$ is called a *prefix* of $s \in \Sigma^*$, written $t \sqsubseteq s$, if $s = tu$ for some $u \in \Sigma^*$. The *prefix-closure* $\overline{\mathcal{L}}$ of a language $\mathcal{L} \subseteq \Sigma^*$ is the set of all prefixes of strings in $\mathcal{L}$, i.e. $\overline{\mathcal{L}} = \{\, t \in \Sigma^* \mid t \sqsubseteq s \text{ for some } s \in \mathcal{L} \,\}$. A language $\mathcal{L}$ is called *prefix-closed* if $\mathcal{L} = \overline{\mathcal{L}}$.

Projection is used to hide events that are not of interest. For $\Omega \subseteq \Sigma$, the *natural projection* $P_\Omega \colon \Sigma^* \to \Omega^*$ removes all events not in Ω from a string. This is extended to languages by $P_\Omega(\mathcal{L}) = \{\, P_\Omega(s) \mid s \in \mathcal{L} \,\}$. The *inverse projection* adds events not in Ω to strings of a language in all possible ways, $P_\Omega^{-1}(\mathcal{L}) = \{\, s \in \Sigma^* \mid P_\Omega(s) \in \mathcal{L} \,\}$.

2.2 Discrete Event Systems

A *discrete event system (DES)* over Σ is a pair $\mathbf{G} = (\mathcal{L}, \mathcal{M})$, where $\mathcal{L} \subseteq \Sigma^*$ is a prefix-closed language, and $\mathcal{M} \subseteq \mathcal{L}$ (Cassandras and Lafortune, 1999). The language $\mathcal{L}$ contains all possible sequences of events that can be executed by the system, while $\mathcal{M}$ contains only event sequences corresponding to completed tasks. A DES $\mathbf{G}$ is completely characterised by these two languages, which are also referred to as $\mathcal{L}(\mathbf{G})$ and $\mathcal{M}(\mathbf{G})$, respectively.

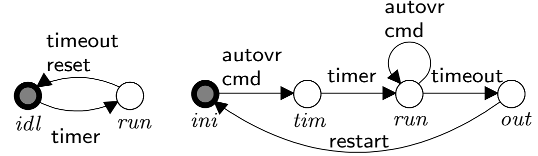

Fig. 1. Two DES represented as state transition diagrams.

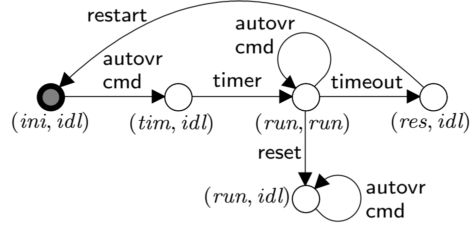

Fig. 2. The synchronous product of the DES in fig. 1.

Discrete event systems can be represented visually using state transition diagrams. Fig. 1 shows two examples of this representation. The left DES represents a timer which, after having been started by event timer, can either produce a timeout or be reset before being started again. Initial states have a thick border, and terminal states are coloured gray. In this example, the states *idl* and *ini* are both initial and terminal states.

2.3 Synchronous Product

When multiple DES run in parallel, lock-step synchronisation is used. In order to execute an event in a certain state, all participating DES must be able to accept that event (Hoare, 1985). It follows that the behaviour $\mathbf{G}_1 \parallel \mathbf{G}_2$ of two DES running in parallel is characterised by the intersection of their languages (Wonham, 1999).

This kind of synchronisation requires $\mathbf{G}_1$ and $\mathbf{G}_2$ to use the same event alphabet Σ. Otherwise, the event alphabet Σ_1 of $\mathbf{G}_1$, e.g., can be extended to the common alphabet $\Sigma = \Sigma_1 \cup \Sigma_2$ using inverse projection. The events not considered in $\mathbf{G}_1$ are assumed to be possible in every state and not to cause any state change.

Given such an extended DES, it is interesting which events are actually used to characterise the behaviour, and which may just have been introduced by the above procedure. An event set $\Omega \subseteq \Sigma$ is called *adequate* for $\mathcal{L} \subseteq \Sigma^*$ if $P_\Omega^{-1}(P_\Omega(\mathcal{L})) = \mathcal{L}$, i.e., if the language can be reconstructed from its projection. Ω is called adequate for a DES $\mathbf{G}$, if Ω is adequate for $\mathcal{L}(\mathbf{G})$ and $\mathcal{M}(\mathbf{G})$.

In the state transition diagram representation, the synchronous product of DES is obtained from the Cartesian product of all involved state spaces. Fig. 2 shows the synchronous product of the two DES in fig. 1, after extending both DES to use the

events autovr, cmd, reset, and restart which occur in only one of the original DES.

2.4 Conflicts

Given a DES $\mathbf{G}$, it is desirable that every behaviour possible in $\mathcal{L}(\mathbf{G})$ can be extended to a complete task in $\mathcal{M}(\mathbf{G})$. A DES satisfying this property is called *nonblocking*; otherwise it is said to be *blocking* (Wonham, 1999). The DES in fig. 2 is blocking, because no terminal state can be reached from state (run, idl).

Blocking becomes more interesting when multiple DES are running in parallel—in this case the term *conflicting* is used instead of blocking. Two DES $\mathbf{G}_1$ and $\mathbf{G}_2$ are called *nonconflicting* if $\mathbf{G}_1 \parallel \mathbf{G}_2$ is nonblocking, i.e.,

$$\mathcal{L}(\mathbf{G}_1) \cap \mathcal{L}(\mathbf{G}_2) \;=\; \overline{\mathcal{M}(\mathbf{G}_1) \cap \mathcal{M}(\mathbf{G}_2)}. \qquad (1)$$

Otherwise $\mathbf{G}_1$ and $\mathbf{G}_2$ are called *conflicting*. The above (1) extends the traditional definition of nonconflicting languages (Wonham, 1999) to potentially blocking DES.

Conflicts are difficult to analyse in a modular way (Wong *et al.*, 2000). For example, although each of the two DES in fig. 1 is nonblocking, they are conflicting when run in parallel (fig. 2). In computation tree logic (CTL) (Clarke *et al.*, 1999), the property of being nonblocking or nonconflicting is expressed as

$$\mathbf{AG}\,\mathbf{EF}\; marked_state, \qquad (2)$$

where *marked_state* is a propositional formula identifying the states in which all involved DES have completed their tasks. This formula is neither in $\forall$CTL* nor in $\exists$CTL*, which explains why many known abstraction techniques (Clarke *et al.*, 1999) cannot be used for this kind of property.

3. FINDING CONFLICTS IN LARGE SYSTEMS

3.1 The Problem

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

$$\mathbf{G}_1 \parallel \mathbf{G}_2 \parallel \cdots \parallel \mathbf{G}_n\,.$$

The check is done by exploring all reachable states and checking whether a terminal state can be reached from every reachable state. Using symbolic representations such as BDDs (Bryant, 1986) or IDDs (Zhang and Wonham, 2002), this approach has been used to analyse very large models. Yet, the technique always remains limited by the amount of memory available to store representations of the synchronous product.

An alternative approach avoids building the entire synchronous product by analysing smaller subsystems first. Assume, for example, that the blocking DES in fig. 2 is the first component $\mathbf{G}_1$ of a large system of concurrent DES. If it can be shown that all components of the system can execute the event sequence cmd timer reset, e.g., then it already follows that the entire system is blocking. In order to determine whether this is the case without constructing the entire synchronous product, an *incremental language inclusion check* (Brandin *et al.*, 2004) can be used.

Further to the example, assume that the events timer and reset are known never to be disabled by any of the other components of the system. In this case, the system is known to be blocking if $\mathbf{G}_1$ can ever enter state (tim, idl): if $\mathbf{G}_1$ is in this state, the entire system can execute the event sequence timer reset and put $\mathbf{G}_1$ into the blocking state (run, idl).

Therefore, the states (tim, idl), (run, run), and (run, idl) are considered as states of *certain conflicts* of $\mathbf{G}_1$: any system that is to be nonconflicting with $\mathbf{G}_1$ has to prevent $\mathbf{G}_1$ from entering these states. In the following, these ideas are presented in a more formal way using languages.

3.2 The Set of Certain Conflicts

If a DES $\mathbf{G}$ interacts with other systems, then its event set Σ can be split into a set Υ of events that are used only by $\mathbf{G}$ and are irrelevant to all other systems, and a set Ω of events that can be shared between $\mathbf{G}$ and other systems.

$$\Sigma \;=\; \Omega \,\dot\cup\, \Upsilon \qquad (3)$$

As seen in the above example, if $\mathbf{G}$ is blocking, then there exists a set of strings accepted by $\mathbf{G}$ which, when accepted by another DES $\mathbf{H}$ running in parallel with $\mathbf{G}$, causes $\mathbf{G}$ and $\mathbf{H}$ to be conflicting. These strings constitute the set of certain conflicts.

Definition 1. Let $\mathbf{G}$ be a DES over Σ, and $\Omega \subseteq \Sigma$. A string $s \in \Omega^*$ is said to be Ω-*blocking* for $\mathbf{G}$, if, for every DES $\mathbf{H}$ over Σ such that Ω is adequate for $\mathbf{H}$ and $s \in \mathcal{L}(\mathbf{H})$, the DES $\mathbf{G}$ and $\mathbf{H}$ are conflicting. Otherwise s is said to be Ω-*nonblocking* for $\mathbf{G}$.

To be Ω-blocking is a property of strings—for s to be Ω-blocking, *every* DES $\mathbf{H}$ with the mentioned properties has to be conflicting with $\mathbf{G}$.

Definition 2. For a DES $\mathbf{G}$ over Σ, and $\Omega \subseteq \Sigma$,

$$\begin{aligned}\textsc{Conf}(\mathbf{G},\Omega) &= \{\, s \in \Omega^* \mid \\ &\qquad s \text{ is } \Omega\text{-blocking for } \mathbf{G}\,\};\\ \textsc{NConf}(\mathbf{G},\Omega) &= \{\, s \in \Omega^* \mid \\ &\qquad s \text{ is } \Omega\text{-nonblocking for } \mathbf{G}\,\}.\end{aligned}$$

$\textsc{Conf}(\mathbf{G},\Omega)$ is the set of *certain conflicts* of $\mathbf{G}$ with respect to Ω. Its complement $\textsc{NConf}(\mathbf{G},\Omega)$ is the set of all strings s of shared events for which there exists another system which accepts s and is nonconflicting with $\mathbf{G}$.

Let $\mathbf{G}_1$ be the DES in fig. 2, and assume $\Upsilon = \{\mathsf{reset}, \mathsf{timer}\}$. Then every DES $\mathbf{H}$ that has the events in Υ selflooped in all states and accepts autovr or cmd must be conflicting with $\mathbf{G}_1$. Thus,

$$\textsc{Conf}(\mathbf{G}_1,\Omega) \;=\; \{\mathsf{autovr}, \mathsf{cmd}\}\,\Omega^*. \qquad (4)$$

The following result shows how the set of certain conflicts can simplify conflict analysis: another system can only be nonconflicting with a DES $\mathbf{G}$ if the behaviour of that system remains confined by $\textsc{NConf}(\mathbf{G},\Omega)$. This gives a necessary condition for complex systems to be nonconflicting.

Proposition 3. Let $\mathbf{G}$ and $\mathbf{H}$ be two DES over Σ, and let $\Omega \subseteq \Sigma$ be adequate for $\mathbf{H}$. If $\mathbf{G}$ and $\mathbf{H}$ are nonconflicting then

$$P_\Omega(\mathcal{L}(\mathbf{H})) \;\subseteq\; \textsc{NConf}(\mathbf{G},\Omega). \qquad (5)$$

Proof. Assume $\mathbf{G}$ and $\mathbf{H}$ are nonconflicting, and let $s \in P_\Omega(\mathcal{L}(\mathbf{H}))$. Since Ω is adequate for $\mathbf{H}$, it follows that $s \in P_\Omega^{-1}(P_\Omega(\mathcal{L}(\mathbf{H}))) = \mathcal{L}(\mathbf{H})$. Hence, s is Ω-nonblocking for $\mathbf{G}$, and $s \in \textsc{NConf}(\mathbf{G},\Omega)$ by definition. □

Consider once more the example subsystem $\mathbf{G}_1$ in fig. 2, which is part of a larger system $\mathbf{G}_1 \parallel \cdots \parallel \mathbf{G}_n$. The events in $\Upsilon = \{\mathsf{timer}, \mathsf{reset}\}$ are not used in $\mathbf{G}_2, \ldots, \mathbf{G}_n$. Therefore, the above result says that, if the behaviour of $\mathbf{G}_2, \ldots, \mathbf{G}_n$ can be shown not to be contained in the language $\textsc{NConf}(\mathbf{G}_1,\Omega)$, i.e. if it is not the case that

$$\mathcal{L}(\mathbf{G}_2 \parallel \cdots \parallel \mathbf{G}_n) \;\subseteq\; P_\Omega^{-1}(\textsc{NConf}(\mathbf{G}_1,\Omega)), \quad (6)$$

then the entire system must be conflicting.

The example discussed above was taken from a faulty model of an actual automotive central locking system (KorSys project), which consists of 54 components and has a reachable state space of 7.5×10^8 states. The two components in fig. 1 that are responsible for the conflict were found and combined automatically, by appropriate heuristics, and an incremental language inclusion check quickly revealed that the event cmd can indeed be executed by the entire system, after executing

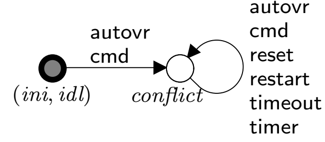

Fig. 3. A DES which is conflict equivalent to fig. 2.

only one other event. The error was that the event reset in the timer model of fig. 1 had been forgotten when modelling the rest of the system. This example demonstrates that proposition 3 provides a very powerful means to detect conflicts in large systems.

3.3 Simplifying Subsystems

If condition (6) is true, it remains unknown whether the composed system $\mathbf{G}_1 \parallel \cdots \parallel \mathbf{G}_n$ is conflicting or not. In this case, the information about the certain conflicts of $\mathbf{G}_1$ can still be used to simplify conflict analysis. The component $\mathbf{G}_1$ can be replaced by a simpler model $\mathbf{G}_1'$, and then the system $\mathbf{G}_1' \parallel \mathbf{G}_2 \parallel \cdots \parallel \mathbf{G}_n$ can be considered for analysis.

The prerequisite for such an approach to work is that the replacement $\mathbf{G}_1'$ preserves all possibilities of $\mathbf{G}_1$ being in conflict with the rest of the system. This suggests to use conflicts to define a testing semantics along the lines of (Brinksma *et al.*, 1995).

Definition 4. Two DES $\mathbf{G}_1$ and $\mathbf{G}_2$ over Σ are called *conflict equivalent* with respect to $\Omega \subseteq \Sigma$ if, for every DES $\mathbf{H}$ such that Ω is adequate for $\mathbf{H}$, $\mathbf{G}_1$ and $\mathbf{H}$ are nonconflicting if and only if $\mathbf{G}_2$ and $\mathbf{H}$ are nonconflicting.

The DES in fig. 3 is conflict equivalent to the DES in fig. 2: any DES $\mathbf{H}$, which does not use the events timer and reset, is conflicting with either of these two DES if and only if it ever accepts one of the events autovr or cmd. Therefore, fig. 2 can be replaced by the smaller fig. 3 before starting a conflict check of the entire system. Furthermore, other components of the system may be replaced in a similar way.

The simplification of fig. 2 to fig. 3 was based on the observation that, as far as conflicts are concerned, all the strings that bear a certain conflict for a DES can be treated in the same way. In a state transition diagram, this can be done by collapsing all states representing certain conflicts into a single state where all events are possible.

The following two results justify this idea. Proposition 5 states that strings from the set of certain conflicts can be added to the language $\mathcal{L}(\mathbf{G})$ of a DES without affecting its possible conflicts.

Proposition 6 states that certain conflicts do not represent proper completed tasks, and therefore can be removed safely from the language $\mathcal{M}(\mathbf{G})$. Fig. 3 can be obtained from fig. 2 by first applying proposition 6 and then proposition 5.

Proposition 5. Let $\mathbf{G}$ be a DES over Σ, and $\Omega \subseteq \Sigma$. Then $\mathbf{G}$ is conflict equivalent to $\mathbf{G}'$ with respect to Ω, where

$$\mathcal{L}(\mathbf{G}') = \mathcal{L}(\mathbf{G}) \cup P_\Omega^{-1}(\textsc{Conf}(\mathbf{G},\Omega));$$
$$\mathcal{M}(\mathbf{G}') = \mathcal{M}(\mathbf{G}).$$

Proof. Let $\mathbf{H}$ be a DES over Σ such that Ω is adequate for $\mathbf{H}$.

First assume that $\mathbf{G}'$ and $\mathbf{H}$ are conflicting. Then there exists $s \in \mathcal{L}(\mathbf{G}') \cap \mathcal{L}(\mathbf{H})$ such that $s \notin \overline{\mathcal{M}(\mathbf{G}') \cap \mathcal{M}(\mathbf{H})} = \overline{\mathcal{M}(\mathbf{G}) \cap \mathcal{M}(\mathbf{H})}$ since $\mathcal{M}(\mathbf{G}') = \mathcal{M}(\mathbf{G})$. If $s \in \mathcal{L}(\mathbf{G})$, it follows immediately that $\mathbf{G}$ and $\mathbf{H}$ are conflicting. Otherwise, if $s \notin \mathcal{L}(\mathbf{G})$, note that $s \in \mathcal{L}(\mathbf{G}')$ and therefore $s \in P_\Omega^{-1}(\textsc{Conf}(\mathbf{G},\Omega))$. Then $s' = P_\Omega(s) \in \textsc{Conf}(\mathbf{G},\Omega)$, and $s' \in \mathcal{L}(\mathbf{H})$ since $s \in \mathcal{L}(\mathbf{H})$ and Ω is adequate for $\mathbf{H}$. By definition of $\textsc{Conf}(\mathbf{G},\Omega)$, $\mathbf{G}$ and $\mathbf{H}$ are conflicting.

Now assume that $\mathbf{G}'$ and $\mathbf{H}$ are nonconflicting. To prove that $\mathbf{G}$ and $\mathbf{H}$ are nonconflicting, let $s \in \mathcal{L}(\mathbf{G}) \cap \mathcal{L}(\mathbf{H}) \subseteq \mathcal{L}(\mathbf{G}') \cap \mathcal{L}(\mathbf{H})$. Since $\mathbf{G}'$ and $\mathbf{H}$ are nonconflicting, there exists $t \in \Sigma^*$ such that $st \in \mathcal{M}(\mathbf{G}') \cap \mathcal{M}(\mathbf{H}) = \mathcal{M}(\mathbf{G}) \cap \mathcal{M}(\mathbf{H})$. Thus, $\mathbf{G}$ and $\mathbf{H}$ are nonconflicting. $\square$

Proposition 6. Let $\mathbf{G}$ be a DES over Σ, and $\Omega \subseteq \Sigma$. Then $\mathbf{G}$ is conflict equivalent to $\mathbf{G}'$ with respect to Ω, where

$$\mathcal{L}(\mathbf{G}') = \mathcal{L}(\mathbf{G});$$
$$\mathcal{M}(\mathbf{G}') = \mathcal{M}(\mathbf{G}) \cap P_\Omega^{-1}(\textsc{NConf}(\mathbf{G},\Omega)).$$

Proof. Let $\mathbf{H}$ be a DES over Σ such that Ω is adequate for $\mathbf{H}$.

First assume that $\mathbf{G}$ and $\mathbf{H}$ are nonconflicting. To prove that $\mathbf{G}'$ and $\mathbf{H}$ are nonconflicting, let $s \in \mathcal{L}(\mathbf{G}') \cap \mathcal{L}(\mathbf{H}) = \mathcal{L}(\mathbf{G}) \cap \mathcal{L}(\mathbf{H})$. Since $\mathbf{G}$ and $\mathbf{H}$ are nonconflicting, there exists $t \in \Sigma^*$ such that $st \in \mathcal{M}(\mathbf{G}) \cap \mathcal{M}(\mathbf{H}) \subseteq \mathcal{L}(\mathbf{H})$. Since Ω is adequate for $\mathbf{H}$, $P_\Omega(st) \in \mathcal{L}(\mathbf{H})$ is Ω-nonblocking for $\mathbf{G}$. By definition of $\textsc{NConf}(\mathbf{G},\Omega)$ it follows that $P_\Omega(st) \in \textsc{NConf}(\mathbf{G},\Omega)$. Thus, $st \in \mathcal{M}(\mathbf{G}) \cap P_\Omega^{-1}(\textsc{NConf}(\mathbf{G},\Omega)) \cap \mathcal{M}(\mathbf{H}) = \mathcal{M}(\mathbf{G}') \cap \mathcal{M}(\mathbf{H})$, i.e., $\mathbf{G}'$ and $\mathbf{H}$ are nonconflicting.

Now assume that $\mathbf{G}'$ and $\mathbf{H}$ are nonconflicting. To prove that $\mathbf{G}$ and $\mathbf{H}$ are nonconflicting, let $s \in \mathcal{L}(\mathbf{G}) \cap \mathcal{L}(\mathbf{H}) = \mathcal{L}(\mathbf{G}') \cap \mathcal{L}(\mathbf{H})$. Since $\mathbf{G}'$ and $\mathbf{H}$ are nonconflicting, there exists $t \in \Sigma^*$ such that $st \in \mathcal{M}(\mathbf{G}') \cap \mathcal{M}(\mathbf{H}) \subseteq \mathcal{M}(\mathbf{G}) \cap \mathcal{M}(\mathbf{H})$. Thus, $\mathbf{G}$ and $\mathbf{H}$ are nonconflicting. $\square$

4. A FIXED POINT ALGORITHM

This section provides an operational characterisation and an algorithm to compute the set of certain conflicts. The set of certain conflicts can be characterised as a fixed point of the following operator which manipulates languages over Ω.

Definition 7. Let $\mathbf{G}$ be a DES over Σ, and $\Omega \subseteq \Sigma$. Then define the mapping

$$\Xi_{\mathbf{G},\Omega} : 2^{\Omega^*} \to 2^{\Omega^*};$$
$$X \mapsto P_\Omega(\mathcal{L}(\mathbf{G}) \setminus \overline{\mathcal{M}(\mathbf{G}) \setminus P_\Omega^{-1}(X)}\,)\,\Omega^*.$$

This operator is designed to compute the set of certain conflicts of the DES $\mathbf{G}$ iteratively. Iteration starts with the empty language $X_0 = \emptyset$ which is extended repeatedly, until the complete set $\textsc{Conf}(\mathbf{G},\Omega)$ is obtained. The first step yields

$$X_1 = \Xi_{\mathbf{G},\Omega}(\emptyset) = P_\Omega(\mathcal{L}(\mathbf{G}) \setminus \overline{\mathcal{M}(\mathbf{G})})\,\Omega^*,$$

the set of all strings that resemble, considering only the shared events from Ω, strings that cannot be extended to a completed task, plus any continuations of such strings. The set of certain conflicts can be shown to be the least fixed point of this iteration.

Proposition 8. Let $\mathbf{G}$ be a DES over Σ, and $\Omega \subseteq \Sigma$. Then $\textsc{Conf}(\mathbf{G},\Omega) = \operatorname{lfp} \Xi_{\mathbf{G},\Omega}$.

The proof uses the fixed point theory of (Tarski, 1955) and is technical. While the operator $\Xi_{\mathbf{G},\Omega}$ can be shown to be monotonic, it is not necessarily continuous, and therefore the iteration is not guaranteed to converge in all cases.

If the DES to be analysed can be represented by a finite-state transition diagram, then the iteration does converge and terminate in a finite number of steps. In this case, it is convenient to use sets of states X'_i instead of languages X_i in the algorithm. In the example of fig. 2, the iteration terminates with the second step, producing

$$X'_0 = \emptyset;$$
$$X'_1 = \{(run, idl), (run, run), (tim, idl)\};$$
$$X'_2 = X'_1.$$

Not all state transition diagrams can be manipulated in such a simple way. A single state may be reachable via different sequences of events which may need to be treated separately.

Such states can be split on the fly or in a preprocessing step. It is possible to compute a representation $\mathbf{P}$ of the language $P_\Omega(\mathcal{L}(\mathbf{G}))$ first, using subset construction (Hopcroft *et al.*, 2001), and then use the synchronous product $\mathbf{G} \parallel \mathbf{P}$ as input for the algorithm. This shows that the procedure outlined above can indeed be used to compute the set of certain conflicts for a finite-state DES.

5. CONCLUSION

The set of *certain conflicts* for a discrete event system has been characterised as the largest set of behaviours which, when accepted by another system running in parallel, definitely cause a livelock or deadlock. This set can help to detect conflicts in complex models of reactive systems without exploring their complete state space, and to simplify the analysis of conflicts in general.

While the abstraction techniques proposed give very powerful reductions for systems that actually have conflicts, i.e. deadlocks or livelocks, the effect on nonconflicting systems remains limited. Although some simplification is possible, it remains necessary to compose all components of a system in order to prove that it actually is nonconflicting.

In the future, the author plans to study abstraction techniques which also deal with behaviours that are not certain conflicts, and obtain better reductions for nonconflicting systems. The goal is to develop efficient techniques for verifying complex systems to be conflicting or nonconflicting.

REFERENCES

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

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

Brandin, Bertil and François Charbonnier (1994). The supervisory control of the automated manufacturing system of the AIP. In: *Proc. Rensselaer's 4th Int. Conf. Computer Integrated Manufacturing and Automation Technology*. Troy, NY, USA. pp. 319–324.

Brinksma, Ed, Arend Rensink and Walter Vogler (1995). Fair testing. In: *Proc. 6th Int. Conf. Concurrency Theory, CONCUR '95* (Insup Lee and Scott A. Smolka, Eds.). Vol. 962 of *LNCS*. Springer. Philadelphia, PA, USA. pp. 313–327.

Bryant, Randal E. (1986). Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Comput.* **35**(8), 677–691.

Cassandras, C. G. and S. Lafortune (1999). *Introduction to Discrete Event Systems*. Kluwer.

Clarke, Jr., Edmund M., Orna Grumberg and Doron A. Peled (1999). *Model Checking*. MIT Press.

Dams, Dennis, Rob Gerth and Orna Grumberg (1997). Abstract interpretation of reactive systems. *ACM Trans. Programming Languages and Systems* **19**(2), 111–149.

Dietrich, P., R. Malik, W. M. Wonham and B. A. Brandin (2002). Implementation considerations in supervisory control. In: *Synthesis and Control of Discrete Event Systems* (B. Caillaud, P. Darondeau, L. Lavagno and X. Xie, Eds.). Kluwer. pp. 185–201.

Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Prentice-Hall.

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

Leduc, R. J. and W. M. Wonham (1995). PLC implementation of a DES supervisor for a manufacturing testbed. In: *Proc. 33rd Allerton Conf. Communication, Control and Computing*. Monticello, Illinois. pp. 519–528.

Leduc, R. J., B. A. Brandin and W. M. Wonham (2000). Hierarchical interface-based nonblocking verification. In: *Proc. Canadian Conf. Electrical and Computer Engineering*. pp. 1–6.

Lee, Sang-Heon and Kai C. Wong (2002). Structural decentralised control of concurrent discrete-event systems. *European J. Control* **8**, 477–491.

Malik, Petra (2003). From Supervisory Control to Nonblocking Controllers for Discrete Event Systems. PhD thesis. University of Kaiserslautern. Kaiserslautern, Germany.

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

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

Wong, K. C. and W. M. Wonham (1998). Modular control and coordination of discrete-event systems. *Discrete Event Dynamic Systems: Theory and Applications* **8**(3), 247–297.

Wong, K. C., J. G. Thistle, R. P. Malhame and H.-H. Hoang (2000). Supervisory control of distributed systems: Conflict resolution. *Discrete Event Dynamic Systems: Theory and Applications* **10**, 131–186.

Wonham, W. M. (1999). Notes on control of discrete event systems. Systems Control Group, Dept. of Electrical Engineering, University of Toronto, Ontario, Canada; at `http://www.control.utoronto.ca/` under "Research".

Zhang, Z. H. and W. M. Wonham (2002). STCT: An efficient algorithm for supervisory control design. In: *Synthesis and Control of Discrete Event Systems* (B. Caillaud, P. Darondeau, L. Lavagno and X. Xie, Eds.). Kluwer. pp. 77–100.