Science of Computer Programming 89 (2014) 23–40

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

An algorithm to test the conflict preorder

Simon Ware*, Robi Malik

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

ARTICLE INFO

Article history:
Received 30 January 2012
Received in revised form 11 June 2013
Accepted 17 September 2013
Available online 3 October 2013

Keywords:
Finite automata
Process algebra
Model checking
Nonblocking

ABSTRACT

This paper proposes a way to effectively compare the potential of processes to cause *conflict*. In discrete event systems theory, two concurrent systems are said to be in conflict if they can get trapped in a situation where they are both waiting or running endlessly, forever unable to complete their common task. The *conflict preorder* is a process-algebraic pre-congruence that compares two processes based on their possible conflicts in combination with other processes. This paper improves on previous theoretical descriptions of the conflict preorder by introducing *less conflicting pairs* as a concrete state-based characterisation. Based on this characterisation, an effective algorithm is presented to determine whether two processes are related according to the conflict preorder.

1. Introduction

A key question in process algebra is how processes can be composed and compared [1,2]. An understanding of what makes processes equivalent is important for several applications, ranging from comparison and minimisation to hierarchical interface design [3] and program construction using abstraction and refinement. Several equivalence relations have been studied, most notably *weak bisimulation* or *observation equivalence* [4], *failures equivalence* [5], and *trace equivalence* [6]. Each equivalence has its own properties, making it suitable for particular applications and verification tasks [2].

This paper proposes a decision procedure for *conflict equivalence* and the associated preorder, which is also known as *weak termination accordance* [7]. Conflict equivalence relates processes based on which other processes they can come into conflict [8,9] with. Two processes are in conflict, if they can reach a state from which termination is no longer possible. This can be because of *deadlock* where neither process is capable of doing anything, or *livelock* where the system continues to run without ever terminating.

Conflict equivalence is introduced in [10] as the best possible process equivalence to reason compositionally about conflicts. It is shown in [10] that conflict equivalence is coarser than weak bisimulation [4], and different from failures and trace equivalence [5], for which decision procedures are known. Weak bisimulation can be decided in cubic complexity [11], while the decision problems for failures and trace equivalence are PSPACE-complete [12].

The process-algebraic theory most closely related to conflict equivalence is *fair testing* [13,14]. The fair testing preorder has got an exponential decision procedure [13], but this procedure cannot be used directly to test the conflict preorder. The essential difference between the conflict and fair testing preorders lies in the capability of the conflict preorder to compare processes that exhibit blocking behaviour, as expressed by the *set of certain conflicts* [10,15,16].

In [17,18], various conflict-preserving rewrite rules are used to simplify processes and check whether or not large systems of concurrent finite automata are free from conflict. While of good use in practice, the rewrite rules are incomplete, and it remains an open question how processes can be normalised or compared for conflict equivalence.

This paper improves on previous results about conflict equivalence and the associated conflict preorder [10], and fair testing [13], by providing a state-based characterisation of the conflict preorder. It proposes *less conflicting pairs* as a more

* Corresponding author.
E-mail addresses: siw4@waikato.ac.nz (S. Ware), robi@waikato.ac.nz (R. Malik).

0167-6423/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.scico.2013.09.006

concrete way to compare processes for their conflicting behaviour than the abstract test-based characterisation using *non-conflicting completions* [10] and the *refusal trees* [13]. Less conflicting pairs give a means to directly compare processes based on their reachable state sets, which leads to an alternative algorithm to test the conflict and fair testing preorders. While still exponential, this algorithm is simpler and has better time complexity than the decision procedure for fair testing [13].

This paper is an extended version of [19]. It contains a more detailed introduction with motivation of compositional nonblocking verification and discussion of the relationship to fair testing, plus a more detailed description of the conflict preorder algorithm with experimental results. Section 2 introduces and motivates the needed terminology of languages, automata, nonblocking, conflict equivalence, and compositional verification. Then Section 3 introduces less conflicting pairs and shows how they can be used to describe certain conflicts and the conflict preorder. Afterwards, Section 4 proposes an algorithm to calculate less conflicting pairs for finite automata, Section 5 describes the implementation of the algorithm and presents the experimental results, and Section 6 adds some concluding remarks.

2. Preliminaries

2.1. Languages and automata

Event sequences and languages are a simple means to describe process behaviours. Their basic building blocks are *events*, which are taken from a finite *alphabet* Σ. Two special events are used, the *silent event* τ and the *termination event* ω. These are never included in an alphabet Σ unless mentioned explicitly.

Σ^* denotes the set of all finite *traces* of the form $\sigma_1\sigma_2\cdots\sigma_n$ of events from Σ, including the *empty trace* ε. The *length* of trace s is denoted by $|s|$. A subset $L \subseteq \Sigma^*$ is called a *language*. The *concatenation* of two traces $s, t \in \Sigma^*$ is written as st, and a trace s is called a *prefix* of t, written $s \sqsubseteq t$, if $t = su$ for some trace u. A language $L \subseteq \Sigma^*$ is *prefix-closed*, if $s \in L$ and $r \sqsubseteq s$ implies $r \in L$.

In this paper, process behaviour is modelled using nondeterministic *labelled transition systems* or *automata* $A = \langle \Sigma, Q, \rightarrow, Q^\circ \rangle$, where Σ is a finite alphabet of *events*, Q is a set of *states*, $\rightarrow \subseteq Q \times (\Sigma \cup \{\tau, \omega\}) \times Q$ is the *state transition relation*, and $Q^\circ \subseteq Q$ is the set of *initial states*. A is called *finite* if its alphabet Σ and state set Q are finite.

The transition relation is written in infix notation $x \xrightarrow{\sigma} y$, and extended to traces by letting $x \xrightarrow{\varepsilon} x$ for all $x \in Q$, and $x \xrightarrow{s\sigma} y$ if $x \xrightarrow{s} z \xrightarrow{\sigma} y$ for some $z \in Q$. The transition relation must satisfy the additional requirement that, whenever $x \xrightarrow{\omega} y$, there does not exist any outgoing transition from y. The automaton A is *deterministic* if $|Q^\circ| \leqslant 1$ and the transition relation contains no transitions labelled τ, and if $x \xrightarrow{\sigma} y_1$ and $x \xrightarrow{\sigma} y_2$ always implies $y_1 = y_2$.

To support silent transitions, $x \overset{s}{\Rightarrow} y$, with $s \in (\Sigma \cup \{\omega\})^*$, denotes the existence of a trace $t \in (\Sigma \cup \{\omega, \tau\})^*$ such that $x \xrightarrow{t} y$, and s is obtained from t by deleting all τ events. For a state set $X \subseteq Q$ and a state $y \in Q$, the expression $X \overset{s}{\Rightarrow} y$ denotes the existence of $x \in X$ such that $x \overset{s}{\Rightarrow} y$, and $A \overset{s}{\Rightarrow} y$ means that $Q^\circ \overset{s}{\Rightarrow} y$. Furthermore, $x \Rightarrow y$ denotes the existence of a trace s such that $x \overset{s}{\Rightarrow} y$, and $x \overset{s}{\Rightarrow}$ denotes the existence of a state $y \in Q$ such that $x \overset{s}{\Rightarrow} y$. For a state, state set, or automaton $\mathbf{X}$, the *language* and the *marked language* are

$$\mathbf{L}(\mathbf{X}) = \bigl\{ s \in \bigl(\Sigma \cup \{\omega\}\bigr)^* \bigm| \mathbf{X} \overset{s}{\Rightarrow} \bigr\} \quad \text{and} \quad \mathbf{L}^\omega(\mathbf{X}) = \mathbf{L}(\mathbf{X}) \cap \Sigma^*\omega. \tag{1}$$

Every prefix-closed language L is recognised by an automaton A such that $\mathbf{L}(A) = L$, but only *regular* languages are recognised by a finite automaton [6].

When two automata are running in parallel, lock-step synchronisation in the style of [5] is used.

Definition 1. Let $A = \langle \Sigma_A, Q_A, \rightarrow_A, Q_A^\circ \rangle$ and $B = \langle \Sigma_B, Q_B, \rightarrow_B, Q_B^\circ \rangle$ be two automata. The *synchronous composition* of A and B is

$$A \parallel B = \bigl\langle \Sigma_A \cup \Sigma_B, Q_A \times Q_B, \rightarrow, Q_A^\circ \times Q_B^\circ \bigr\rangle, \tag{2}$$

where

$$\begin{aligned}
(x_A, x_B) &\xrightarrow{\sigma} (y_A, y_B) && \text{if } \sigma \in (\Sigma_A \cap \Sigma_B) \cup \{\omega\},\ x_A \xrightarrow{\sigma}_A y_A, \text{ and } x_B \xrightarrow{\sigma}_B y_B;\\
(x_A, x_B) &\xrightarrow{\sigma} (y_A, x_B) && \text{if } \sigma \in (\Sigma_A \setminus \Sigma_B) \cup \{\tau\} \text{ and } x_A \xrightarrow{\sigma}_A y_A;\\
(x_A, x_B) &\xrightarrow{\sigma} (x_A, y_B) && \text{if } \sigma \in (\Sigma_B \setminus \Sigma_A) \cup \{\tau\} \text{ and } x_B \xrightarrow{\sigma}_B y_B.
\end{aligned}$$

In synchronous composition, shared events (including ω) must be executed by all automata together, while events used by only one of the composed automata and silent (τ) events are executed independently.

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

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