Available online at www.sciencedirect.com

SCIENCE DIRECT®

Information and Computation 202 (2005) 191–226

Information and Computation

www.elsevier.com/locate/ic

Using heuristic search for finding deadlocks in concurrent systems

Sara Gradara*, Antonella Santone, Maria Luisa Villani

RCOST—Research Centre on Software Technology, University of Sannio, Palazzo Ex Poste, via Traiano 1, 82100 Benevento, Italy

Received 20 March 2003; revised 12 January 2005
Available online 19 September 2005

Abstract

Model checking is a formal technique for proving the correctness of a system with respect to a desired behavior. This is accomplished by checking whether a structure representing the system (typically a labeled transition system) satisfies a temporal logic formula describing the expected behavior. Model checking has a number of advantages over traditional approaches that are based on simulation and testing: it is completely automatic and when the verification fails it returns a counterexample that can be used to pinpoint the source of the error. Nevertheless, model checking techniques often fail because of the state explosion problem: transition systems grow exponentially with the number of components. The aim of this paper is to attack the state explosion problem that may arise when looking for deadlocks in concurrent systems described through the Calculus of Communicating Systems. We propose to use heuristics-based techniques, namely the A* algorithm, both to guide the search without constructing the complete transition system, and to provide minimal counterexamples. We have realized a prototype tool to evaluate the methodology. Experiments we have conducted on processes of different size show the benefit from using our technique against building the whole state space, or applying some other methods.

Keywords: State explosion; Deadlock; Heuristic search; CCS

* Corresponding author. Fax: +39 0824 50552.
E-mail addresses: gradara@unisannio.it (S. Gradara), santone@unisannio.it (A. Santone), villani@unisannio.it (M.L. Villani).

0890-5401/$ - see front matter © 2005 Elsevier Inc. All rights reserved.
doi:10.1016/j.ic.2005.07.004

1. Introduction

Model checking [17] is a method to formally and automatically verify the correctness of finite-state concurrent and distributed systems. Many tools exist for model checking systems; see, for example, the Concurrency Workbench of New Century (CWB-NC) [20]. In almost all of these tools, the verification problem consists of the following two steps: first, a process specification is given as input to a model builder for producing a finite labeled transition system, and then this transition system is checked against the requirements expressed in suitable temporal logic formulae. Thus, a model checker takes the transition system and the formula, and returns "true" if the formula is verified on the transition system, otherwise it returns a counterexample that can be used to pinpoint the source of the error (see Fig. 1).

Unfortunately this method is often discarded in practice: software verification, in industrial projects, usually relies on techniques such us code inspection and testing [7], while correctness is very seldom formally proved. Although there is a number of potential causes for uncertainty in practical use of such formal method, a primary cause is the so-called *state explosion problem* which can occur in systems with many components that can interacts with each other. Often, not only intrinsically complex concurrent systems, but also not very large ones, are described by a transition system with a prohibitive number of states.

Many approaches have been proposed to address the state explosion problem, including symbolic verification [47], on-the-fly techniques [42], partial-order methods [31,63], abstraction approaches [16], and compositional reasoning [18,39,56]. Although the size of the transition systems that could

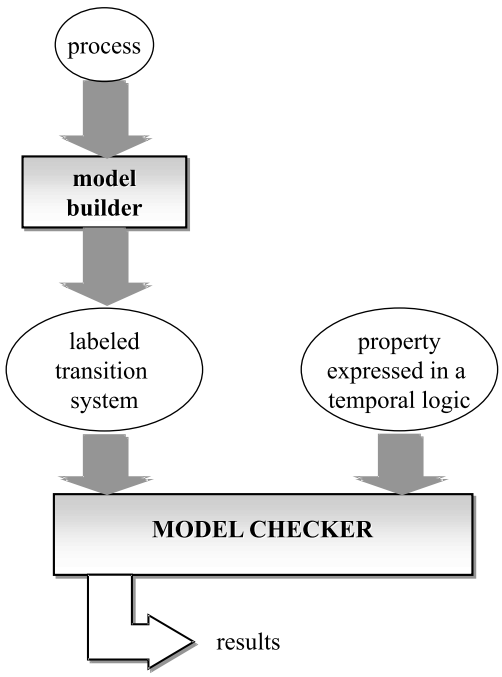

Fig. 1. Model checker.

be verified has been increased by these approaches, many realistic systems are still too large to be handled.

The aim of this paper is to tackle the state explosion problem arising when finding deadlocks in concurrent systems. We consider systems described in the Calculus of Communicating Systems (CCS) [49], which is a mathematical model to describe processes, mostly used in the study of parallelism.

Process algebraic techniques generally feature an elegant set of operators for developing concurrent systems, and so succinct expressions of communicating concurrent processes can be made. Similarly, the emphasis on non-determinism encourages an elegant specification and abstracts away from implementation details. In fact, rich and tractable mathematical models of the semantics of process algebra have been developed; in particular, the CCS model is based upon concepts of equivalence through observation of the external behavior of a specification. As many verification environment tools exist that support CCS, we believe that efforts for studying reduction solutions from CCS specifications are probably worth from both theoretical and application points of view.

Using standard model checkers, deadlock detection in a system *sys* is performed by checking whether the transition system which describes the behavior of *sys* (namely $\mathcal{S}(sys)$) contains a deadlocked state. Thus, the complexity in time and space of deadlock detection is heavily influenced by the size of $\mathcal{S}(sys)$. Although one of the strongest advantages of model checking is the generation of counterexamples when verification fails, traditional model checkers (based on depth-first search algorithms) tend to return very long counterexamples.

As model checking can be seen as a search in a state space, in this paper we propose a method that exploits *heuristics* to explore large state spaces when looking for deadlocked states. Moreover, we would like to produce much more succinct counterexamples, if any. Recently, *heuristic search techniques* [52] have been suggested to deal with the state explosion problem when attempting to find errors. This approach is known as *directed model checking* [28]. Heuristic search is one of the classical techniques widely used in Artificial Intelligence (AI), and has been applied to a wide range of problem-solving tasks including puzzles, two player games, and path finding problems. It exploits information of the specific problem being solved in order to guide the search. A key assumption of heuristic search is that it uses a heuristic evaluation function estimating the distance from a given node to the closest goal node. More precisely, a heuristic evaluation function is a function that maps nodes to numbers. The heuristic search prefers to visit nodes that appear to be better, i.e., nodes with the minimum evaluation. In this way, goal nodes can be found faster. An important property of a heuristic function is the so-called *admissibility property: a heuristic function is admissible if it is a lower bound on the actual cost being estimated.* For example, an admissible heuristic function for a route planning is the Euclidean or airline distance to the goal. This heuristic function satisfies the above property: since the shortest path between two points is a straight line, the road distance must be at least as big as the airline distance. A well-known heuristic search algorithm for solving state-space search problems is A* [40]. The A* algorithm finds a solution with the form of a sequence of operators leading from a start node to a goal node. If the heuristic function satisfies the admissibility property, then A* will find the optimal solution.

In this paper, we use the A* algorithm and propose an admissible heuristic function that suggests to expand first the states that offer the most promising way to deduce that the system has a deadlocked state; in this way we reduce the number of state expansions. The goal is to find a deadlocked state, that is a goal node, but also to find the shortest path leading to it. We believe that it is

important to return the shortest path leading to a deadlock, since that path is examined in order to determine the cause of deadlock. Long error paths can prevent an easy comprehension of the fault. Moreover, it is possible to apply our approach to verify also infinite concurrent systems. We show that a significant space reduction may be obtained by using our method instead of constructing the whole state space. In fact, we have used a prototype tool implementing our algorithm to run several experiments on CCS processes of different size, and compared the results of our method with those of the complete exploration, and the Breath First Search [52].

The methodology proposed in this paper is quite general and can be applied also to verify, for example, multi-threaded JAVA programs: finding deadlocks in multi-threaded programs is difficult, mainly because of the several possible interleaving, any of which may contain a deadlock. We give a method to transform a JAVA program into a CCS process. Once we have the CCS specification of the program, we can apply our method to deadlock detection.

The remainder of the paper is organized as follows: the A* heuristic search algorithm is recalled in Section 2, Section 3 is a review of the basic concepts of CCS, while Section 4 describes our approach. In Section 10, the prototype tool implementing our approach is briefly presented, and the experimental results we obtained by using it are reported in Section 6. The method to transform a multi-threaded Java source code into CCS specifications is defined in Section 7. Finally, comparisons with related works are discussed in Section 8, instead a proof for the admissibility of our heuristics is given in Appendix A.

2. Heuristic search and A* algorithm

One of the most widely used frameworks for problem-solving in AI is the state-space search. A state-space search problem P can be represented in the form:

$$P = (\mathcal{N}, \mathcal{O}, N_0, \mathcal{G}),$$

where

- $\mathcal{N}$ is a set of *nodes*. A node is a complete description of the world for the purposes of problem-solving. For example, in a chess game, the nodes might be the positions of the pieces on the board, while in a route planning, a location.
- $\mathcal{O}$ is a set of (partial) functions $\mathcal{N} \longrightarrow \mathcal{N}$ called *operators* (or *arcs*). An operator transforms one node of the world into another node. In a chess game, the operators might be the legal moves for the pieces given the current board position. In a route planning on a digitized map, the operators could be the moves to a neighboring cell. Each arc has a number associated with it that represents the cost of traversing that arc.
- N_0 is the *start node*, i.e., the state the world is in when problem-solving begins.
- $\mathcal{G}$ is a subset of $\mathcal{N}$, called *goal nodes*. For example, in a route planning the goal nodes would be the desired destination(s).

A *path* is a sequence of connected nodes. A *solution path* is any path whose first node is a designated *start* node and whose last node is one of a designated set of *goal* nodes. The *cost path* is

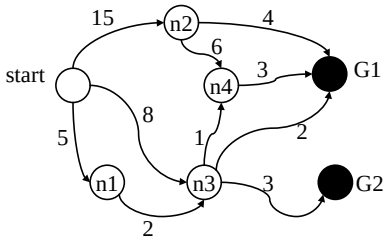

Fig. 2. Graph example.

the sum of the costs of the arcs in the path. A *preferred path* is a path with the lowest possible cost of getting from its first to its last node. The objective of a search is to find a solution path and also to optimize some measure of the cost of the solution. For small search spaces, techniques of exhaustive search can be used, while for very large spaces these techniques are unfeasible. Heuristic search techniques attempt to overcome size problems by using knowledge about the domain to guide the search.

A well-known heuristic search algorithm for solving state-space search problems is A* [52]. A* is an algorithm for finding a path in a graph that leads to a goal node.

Fig. 2 shows a graph with many solution paths (G1 and G2 are goal nodes); for example

$$(start, n2, n4, G1)$$

is a solution path whose cost is 24. For this graph the preferred solution path is

$$(start, n1, n3, G1),$$

whose cost is 9.

In addition to nodes, arcs and costs, A* uses one more kind of data: a number $\widehat{h}(n)$, that is a heuristic evaluation function, associated with each node. Specifically, $\widehat{h}(n)$ is an estimate of a lower bound on the cost of getting from that node to a goal node. The A* algorithm prefers to visit nodes that appear to be better, i.e., nodes with the minimum evaluation, so that goal nodes are found faster. More precisely, at each step A* generates and evaluates the successors of an unexplored node n with the lowest total estimate, $f(n) = g(n) + \widehat{h}(n)$, that is the sum of the distance from the start node to the node n, namely $g(n)$, plus the estimate from the node n to the goal node, i.e., $\widehat{h}(n)$. It stops when a goal node is chosen. A* starts with the initial node and generates all its children nodes. The estimates are usually based on logical or physical knowledge, which otherwise would not be represented in the graph. If the nodes represent cities and the arc costs are railroad miles, then $\widehat{h}(n)$ might be the airline distance from the city n to the goal city; if the nodes are puzzle positions, $\widehat{h}(n)$ might be the minimum number of moves before the puzzle is possibly solved. Fig. 3 outlines the A* algorithm for finding the minimal-cost solution path in a graph. For a more precise description of the algorithm the reader can refer to [52].

An important property holds: A* returns a minimal-cost solution path provided that the heuristic estimate function $\widehat{h}$ satisfies the so-called *admissibility* condition, i.e., $\widehat{h}$ is optimistic, and f is a non-decreasing function. More formally:

(1) Let $g(n)$ be the cost of a path from *start* to node n and get $g(start) = 0$. Let OPEN be a list of nodes that initially contains only the *start* node. Calculate the estimate $\widehat{h}(start)$.
(2) Select the node n on the list OPEN such that the quantity $f(n) = (g(n) + \widehat{h}(n))$ is the smallest. In presence of two or more nodes with the same $f(n)$, just select any of them. If n is a goal node, then the path to n is a preferred solution path and its cost is $g(n)$. If there are no OPEN nodes, there is no solution path in the graph.
(3) Remove n from OPEN. Find all the successors of n. For each successor s, let $g(s) = g(n) + (\text{cost on arc form } n \text{ to } s)$. Calculate the estimate $\widehat{h}(s)$ and add s to OPEN if $\widehat{h}(s)$ is not ∞.
(4) Go to step 2.

Fig. 3. The A* algorithm.

Definition 2.1 (*admissibility*)**.** A heuristic estimate function $\widehat{h}$ defined on the nodes of a graph G is admissible if for each node n in G,

$$\widehat{h}(n) \leqslant h(n),$$

where $h(n)$ is the actual cost of a preferred path from n to a goal node.

3. The calculus of communicating systems

We briefly recall the main concepts about CCS [49] that is widely used in the specification of concurrent and distributed systems. The syntax of a *process* is the following:

$$p ::= nil | x | \alpha.p | p + p | p | p | p \backslash L | p[f],$$

where α ranges over a finite set of actions $\mathcal{A} = \{\tau, a, \overline{a}, b, \overline{b}, ...\}$. The action $\tau \in \mathcal{A}$ is called the *internal action*. The set of *visible actions*, $\mathcal{V}$, ranged over by l, is defined as $\mathcal{A} - \{\tau\}$. The set L, in processes of the form $p\backslash L$, is a set of actions such that $L \subseteq \mathcal{V}$; while the relabeling function f, in processes of the form $p[f]$, is a total function, $f : \mathcal{A} \to \mathcal{A}$, such that the constraint $f(\tau) = \tau$ is respected. Each action $l \in \mathcal{V}$ (resp. $\overline{l} \in \mathcal{V}$) has a *complementary action* $\overline{l}$ (resp. l). Given $L \subseteq \mathcal{V}$, with $\overline{L}$ we denote the set $\{\overline{l} | l \in L\}$. Variable x ranges over a set of *constant* names: each constant x is defined by a constant definition $x \stackrel{\text{def}}{=} p$, where p is called the *body* of x. We denote the set of all processes by $\mathcal{P}$.

Given a set $\mathcal{D}$ of constant definitions, the standard *operational semantics* is given by a relation $\longrightarrow_{\mathcal{D}} \subseteq \mathcal{P} \times \mathcal{A} \times \mathcal{P}$. $\longrightarrow_{\mathcal{D}}$ ($\longrightarrow$ for short) is the least relation defined by the rules in Fig. 4.

We now informally explain the semantics of a CCS process. Note that there is no rule for the process *nil*, which thus cannot perform any action. In rule **Act** the process $\alpha.p$ can perform the action α to become the process p. The rule **Sum** states that p and q are alternative choices for the behavior of $p + q$. The operator | expresses the parallel execution. The rule **Par** shows how processes in a parallel composition can behave autonomously: if the process p performs α and becomes p', then

$$\textbf{Act}\quad \frac{}{\alpha.p \xrightarrow{\alpha} p} \qquad \textbf{Sum}\quad \frac{p \xrightarrow{\alpha} p'}{p+q \xrightarrow{\alpha} p'}\ \text{(and symmetric)}$$

$$\textbf{Con}\quad \frac{p \xrightarrow{\alpha} p'}{x \xrightarrow{\alpha} p'}\ x \stackrel{\text{def}}{=} p \in \mathcal{D} \qquad \textbf{Par}\quad \frac{p \xrightarrow{\alpha} p'}{p|q \xrightarrow{\alpha} p'|q}\ \text{(and symmetric)}$$

$$\textbf{Com}\quad \frac{p \xrightarrow{l} p',\ q \xrightarrow{\bar{l}} q'}{p|q \xrightarrow{\tau} p'|q'} \qquad \textbf{Rel}\quad \frac{p \xrightarrow{\alpha} p'}{p[f] \xrightarrow{f(\alpha)} p'[f]}$$

$$\textbf{Res}\quad \frac{p \xrightarrow{\alpha} p'}{p\backslash L \xrightarrow{\alpha} p'\backslash L}\ \alpha \notin (L \cup \overline{L})$$

Fig. 4. Standard operational semantics of CCS.

$p|q$ performs α and becomes $p'|q$ (similarly for q). When rule **Com** is used, we say that a *handshake* occurs. A handshake occurs only if two processes can simultaneously execute the complementary actions; a handshake corresponds to an internal communication (the action τ). The operator $\backslash L$, in rule **Res**, prevents the actions in $(L \cup \overline{L})$ to be done: if p can perform α to become p', then $p\backslash L$ can perform α to become $p'\backslash L$ only if $\alpha \notin L \cup \overline{L}$. In rule **Rel**, the operator $[f]$ renames actions by means of the relabeling function f: if p can perform α to become p', then $p[f]$ can perform $f(\alpha)$ to become $p'[f]$. Finally, a constant x behaves as p if $x \stackrel{\text{def}}{=} p$ as stated in rule **Con**. Roughly speaking, the Con rule states that a process behaves like its definition. As an example, let us consider the process $x \stackrel{\text{def}}{=} a.b.x$. From the Act rule, the process $a.b.x$ may move to the state $b.x$ after the action a. Thus, from the Con rule, we also have: $x \xrightarrow{a} b.x$.

Given a process p, a constant x of p is said to be *guarded in p* if x is contained in a sub-process of p of the form $\alpha.q$, where q is a process. A process p is *guarded* if every constant of p is guarded in p, it is *unguarded* otherwise. In the following, $Unfold^{x}(p)$ is the process obtained replacing each unguarded constant x by its definition. For example, if $x \stackrel{\text{def}}{=} \overline{a}.x$, $Unfold^{x}((a.b.x|x)\backslash\{a\})$ is the process $(a.b.x|\overline{a}.x)\backslash\{a\}$.

A *(labeled) transition system* is a quadruple $\mathcal{T} = (S, L, \longrightarrow, s)$, where S is a set of states, L is a set of transition labels (actions), $s \in S$ is the initial state, and $\longrightarrow \subseteq S \times L \times S$ is the transition relation. If $(s, \alpha, s') \in \longrightarrow$, we write $s \xrightarrow{\alpha} s'$.

If $\delta \in L^*$ and $\delta = \alpha_1 \cdots \alpha_n, n \geqslant 1$, $p \xrightarrow{\delta} q$ means $p \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} q$; moreover $p \xrightarrow{\lambda} p$, where λ is the empty sequence. Furthermore, $\mathcal{D}er(p) = \{q | p \xrightarrow{\delta} q\}$ denotes the set of the derivatives of p. Given a CCS process p, the *standard transition system* for p is defined as $\mathcal{S}(p) = (\mathcal{D}er(p), \mathcal{A}, \longrightarrow_{\mathcal{D}}, p)$.

In the following, $p \not\longrightarrow$ denotes that no p', α exist such that $p \xrightarrow{\alpha} p'$,while $p \not\xrightarrow{\alpha}$ denotes that no p' exists such that $p \xrightarrow{\alpha} p'$.

Given a process p, $\mathcal{F}irst(p) = \{\alpha \in \mathcal{A} | \exists p' \text{ s.t. } p \xrightarrow{\alpha} p'\}$ denotes the set of all the first actions that p can perform. It can be syntactically defined as the least solution of the following recursive definition:

Definition 3.1 (*First action*)

$$
\begin{aligned}
\mathcal{F}irst(nil) &= \emptyset \\
\mathcal{F}irst(\alpha.p) &= \{\alpha\} \\
\mathcal{F}irst(p+q) &= \mathcal{F}irst(p) \cup \mathcal{F}irst(q) \\
\mathcal{F}irst(p\backslash L) &= \mathcal{F}irst(p) - (L \cup \overline{L}) \\
\mathcal{F}irst(x) &= \mathcal{F}irst(p) \quad \text{if } x \stackrel{\text{def}}{=} p \\
\mathcal{F}irst(p[f]) &= \{f(\alpha) | \alpha \in \mathcal{F}irst(p)\} \\
\mathcal{F}irst(p|q) &= \begin{cases} \mathcal{F}irst(p) \cup \mathcal{F}irst(q) \cup \{\tau\} & \text{if } \exists \alpha \in \mathcal{F}irst(p) \text{ and } \exists \overline{\alpha} \in \mathcal{F}irst(q) \\ \mathcal{F}irst(p) \cup \mathcal{F}irst(q) & \text{otherwise} \end{cases}
\end{aligned}
$$

Now the notion of deadlock is defined.

Definition 3.2 (*deadlock*)**.** Let p be a CCS process.
p is *deadlocked* if $p \not\longrightarrow$;
p is deadlock sensitive if and only if $q \in \mathcal{D}er(p)$ exists such that $q \not\longrightarrow$;
p is deadlock free if and only if it is not deadlock sensitive.

Note that p is deadlock sensitive if and only if $\mathcal{S}(p)$ contains at least a deadlocked state corresponding to a deadlocked process.

Remark 3.1. From now on, without loss of generality, we consider only parallel compositions of the form $(q_1| \cdots |q_n)$, such that each process q_i, $i \in [1..n]$ does not contain the parallel operator. Moreover, for each process $q = (q_1| \cdots |q_n)$ we assume that if an action α belongs to the sort[1] of q_i, with $i \in [1..n]$ and $\overline{\alpha}$ belongs to the sort of q_j with $j \in [1..n]$ and $i \neq j$, then the process q occurs under a restriction set L such that $L \cup \overline{L}$ contains α. If both α and $\overline{\alpha}$ appear in a process, it is reasonable to assume that they are communication actions.

4. The approach

Many automatic techniques for verifying concurrent systems are based on the representation of the concurrent system by means of a transition system. State explosion is one of the most serious problems of these techniques: even small or medium size systems may be described by a transition system with a prohibitive number of states. This problem mostly occurs in systems with many components that interact with each other. The idea of our approach is to reduce the state explosion

[1] The sort of a CCS process p is the alphabet of p. For the precise definition, see [49].

problem when searching for a deadlocked state. More precisely, given a process p, we want to find a deadlocked state in $\mathcal{S}(p)$, without generating the whole transition system. We begin by over-viewing our approach by means of the following simple working example.

Example 4.1.

Consider the following CCS process:

$$p = a.(b.c.x + d.e.a.d.y) + b.d.e.nil + c.d.nil$$
$$x \stackrel{\text{def}}{=} c.x$$
$$y \stackrel{\text{def}}{=} d.y$$

and suppose that we want to check whether p is deadlock sensitive. The standard model checkers (see, for example, that one used by the CWB-NC tool [20]) generate the transition system for p, $\mathcal{S}(p)$, which has 10 states and 12 transitions (see Fig. 5). Then, the deadlock property is expressed by means of a temporal logic formula that is checked on $\mathcal{S}(p)$. It holds that p is deadlock sensitive. Thus, the complexity in time and space of deadlock detection is heavily influenced by the size of $\mathcal{S}(p)$.

With our approach, we would like to stop the construction of the standard transition system when we can deduce that the system is deadlock sensitive. Moreover, we want to find the shortest path leading to a deadlocked state. We point out that finding the shortest path leading to a deadlock is useful since that path is examined to pinpoint the source of the error. Short paths facilitate the comprehension of the fault. To achieve this result, we use a heuristic function which suggests that the most promising states for reaching a deadlock are expanded first; in this way, we reduce the number of state expansions.

For instance, consider the process p in Example 4.1. To conclude that there is a deadlocked state, we can just generate either the states sequence $\langle p, s_2, s_5, nil\rangle$ or $\langle p, s_3, nil\rangle$: it is more efficient to perform an action of one of the last two processes of the choice. Since we want to find the shortest path leading to a deadlock, we perform the first action of the last process of the choice and we return the path $\langle p, s_3, nil\rangle$. In the following subsection, we formally explain our method.

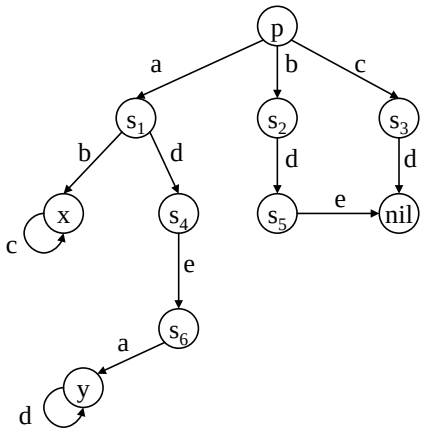

Fig. 5. The transition system for p.

4.1. Finding deadlocked states using A*

In this section, we apply the heuristic search A* described in Section 2 to verify CCS processes. We must define a heuristic function that guides the construction of a minimal-cost solution path. For this, we have considered both the safe and critical points for a possible deadlock in order for the function to preserve admissibility but also be informative. Given a process p, the function $\widehat{h}(p)$ associates a non-negative value with p, called the *$\widehat{h}$-value of p*. Roughly speaking, that value approximates the least number of actions which must be performed before reaching a deadlocked state.

Finding deadlocked states in the transition system for a CCS process p can be seen as a search problem:

$$P = (\mathcal{N}, \mathcal{O}, N_0, \mathcal{G}),$$

where $\mathcal{N}$ is the set of the states of $\mathcal{S}(p)$, $\mathcal{O}$ is the standard operational semantics $\longrightarrow$, N_0 is the process p and $\mathcal{G}$ is the set of all the states $s \in \mathcal{D}er(p)$, such that $s \not\longrightarrow$. We set all cost arcs to 1. This implies that finding the minimal-cost solution path means finding the shortest solution path.

We formally define the function $\widehat{h}$.

Definition 4.1 ($\widehat{h}(p)$)**.** Let p be a CCS process, $\mathcal{L}$ a set of visible actions, and C a set of constants. First, we define the auxiliary function $\widehat{h}$ with three arguments, $\widehat{h}(p, \mathcal{L}, C)$, inductively on p, as in Fig. 6. Then, $\widehat{h}(p)$ is defined as: $\widehat{h}(p) = \widehat{h}(p, \emptyset, \emptyset)$.

The heuristic function $\widehat{h}(p, \mathcal{L}, C)$ is parametric with respect to a *restriction environment* $\mathcal{L}$ ($\mathcal{L} \subseteq \mathcal{V}$), which keeps the set of actions on which some restriction holds. The function is initially applied to a process with $\mathcal{L} = \emptyset$. The current environment $\mathcal{L}$ is modified when the function is applied to $p\backslash L$ (*Rule R5*): in this case the actions in $L \cup \overline{L}$ are added to $\mathcal{L}$. Note that we expand the body of each constant x only once (*Rule R7*), this is because each constant already expanded is stored in C. Initially, C is equal to the empty set.

For $p = nil$ (*Rule R1*) the function $\widehat{h}$ returns 0 as this is a deadlock by definition.

When applied to $\alpha.p$ (*Rule R2*), the function returns 0 if α is a restricted action (i.e., $\alpha \in \mathcal{L}$), otherwise we recursively apply the function to find, if any, an action in $\mathcal{L}$. Roughly speaking, if α is restricted by $\mathcal{L}$ then $\alpha.p$ could not be able to move; thus, we optimistically return 0.

When the choice of two processes is encountered (*Rule R3*), the minimum number of actions between the two components is returned.

Now, consider the parallel composition of processes (*Rule R4*). First, we unfold once the unguarded constants occurring in the parallel composition. This can be done storing the unfolded constants in C. Moreover, in the case where all the parallel components are guarded, if:

- there exists an independent component of the parallel composition, i.e., a process that can perform a non-restricted action ($p_i = \alpha.q$ and $\alpha \notin \mathcal{L}$); or
- all components can perform only restricted actions and there exists one and only one pair of processes that can communicate on a restricted action (remember Remark 3.1), and this unique pair has the form $\alpha.q, \overline{\alpha}.r$.

$$\mathbf{R1}.\quad \widehat{h}(nil,\mathcal{L},C) = 0$$

$$\mathbf{R2}.\quad \widehat{h}(\alpha.p,\mathcal{L},C) = \begin{cases} 0 & \text{if } \alpha \in \mathcal{L} \\ 1+\widehat{h}(p,\mathcal{L},C) & \text{otherwise} \end{cases}$$

$$\mathbf{R3}.\quad \widehat{h}(p_1+p_2,\mathcal{L},C) = \min(\widehat{h}(p_1,\mathcal{L},C),\widehat{h}(p_2,\mathcal{L},C))$$

$$\mathbf{R4}.\quad \widehat{h}(p_1|\cdots|p_n,\mathcal{L},C) =$$

$$\begin{cases} \widehat{h}(\mathit{Unfold}^x(p_1|\cdots|p_n),\mathcal{L},C\cup\{x\}) & \text{if there exists an unguarded constant } x \\ & \text{in } p_1|\cdots|p_n \text{ with } x\notin C \\ 1+\widehat{h}(p_1|\cdots|q|\cdots|p_n,\mathcal{L},C) & \text{if } p_i \text{ is guarded, } i\in[1..n], \text{ and there} \\ & \text{exists } i\in[1..n] \text{ s.t. } p_i=\alpha.q, \text{ and } \alpha\notin\mathcal{L} \\ 1+\widehat{h}(p_1|\cdots|q|\cdots|r|\cdots|p_n,\mathcal{L},C) & \text{if } p_i \text{ is guarded, } \mathcal{F}\mathit{irst}(p_i)\subseteq\mathcal{L}, \\ & \text{for all } i\in[i..n], \text{ and there exists a unique} \\ & \alpha\in\mathcal{L} \text{ s.t. } p_i=\alpha.q,\ p_j=\overline{\alpha}.r, \text{ with} \\ & \alpha,\overline{\alpha}\notin\mathcal{F}\mathit{irst}(p_k) \text{ for all } k\neq i\neq j \\ & \text{for some } \alpha\in\mathcal{L} \\ \sum_{i=1}^{n}\widehat{h}(p_i,\mathcal{L},\emptyset) & \text{otherwise} \end{cases}$$

$$\mathbf{R5}.\quad \widehat{h}(p\backslash L,\mathcal{L},C) = \widehat{h}(p,\mathcal{L}\cup L\cup\overline{L},C)$$

$$\mathbf{R6}.\quad \widehat{h}(p[f],\mathcal{L},C) = \widehat{h}(p,f^{-1}(\mathcal{L}),C)$$

$$\mathbf{R7}.\quad \widehat{h}(x,\mathcal{L},C) = \begin{cases} \infty & \text{if } x\in C \\ \widehat{h}(p,\mathcal{L},C\cup\{x\})) & \text{if } x\notin C \text{ and } x\stackrel{\text{def}}{=}p \end{cases}$$

Fig. 6. The $\widehat{h}$ function.

then the estimated number of actions to a deadlocked state is 1 plus the value returned by a recursive application of the function. In all the other cases, the sum of the number of actions by each parallel process p_i is returned. We note that this is computed starting again from C equals to the empty set so that we are able to apply the definition of $\widehat{h}$ on a constant process in case this occurs in the definition of p_i.

The following example clarifies why we stop the calculation in presence of more than one communication actions. Let us consider the process:

$$q = (a.b.nil|\overline{a}.nil|\overline{a}.\overline{b}.nil)\backslash\{a,b\}. \tag{1}$$

In this case, two pairs of processes exist, which can communicate on the restricted action a: $(a.b.nil,\overline{a}.nil)$ and $(a.b.nil,\overline{a}.\overline{b}.nil)$. With the first pair, after the execution of one action, the τ action, we reach a deadlocked state, while, with the second one, a deadlocked state is reached after

two τ actions. Since the function $\widehat{h}$ must be admissible (for the optimality of A*), in this case, we return 0. Clearly, it is possible to investigate a more accurate function.

When considering a relabeled process (*Rule R6*), we must take as set of actions the set $f^{-1}(\rho) = \{\alpha | f(\alpha) \in \rho\}$, since now the interesting actions are also those relabeled by f into actions in $\mathcal{L}$.

Finally (*Rule R7*), we return ∞ when we encounter a constant already expanded (more precisely, when we encounter a constant x such that $x \in C$). In this case, no action in $\mathcal{L}$ has been found, and this means that the state under consideration is safe: a state that can perform actions in $\mathcal{L}$ is more promising. For example, suppose that there are two states $c.x$ (where $x \stackrel{\text{def}}{=} c.x$) and $a.nil$. It holds that the $\widehat{h}$-value associated to $a.nil$ is equal to 1, which is less than that associated to $c.x$, which is equal to ∞. Thus, expanding first $a.nil$ is more promising for finding a deadlocked state than expanding first $c.x$. Actually, when we reach a state whose $\widehat{h}$-value is ∞, we will never expand that state, because surely it will be deadlock free (see Lemma 4.1).

A consequence of the definition of $\widehat{h}$ is that, while considering a process q, any sub-process r of q is evaluated in an environment which is the union of the restriction sets of all the restriction contexts containing that occurrence of r. For example, if

$$q = \Big((a.b.x)\backslash\{e\} | (\overline{b}.y + (c.y)\backslash\{d\})\Big)\backslash\{b\}$$

$a.b.x$ is evaluated under the environment $\{e, b\} \cup \{\overline{e}, \overline{b}\}$ and $c.y$ is evaluated under the environment $\{d, b\} \cup \{\overline{d}, \overline{b}\}$.

Recall the process p of Example 4.1. It holds, for example, that $\widehat{h}(s_3) = 1$ (where $s_3 = d.nil$); this means that to reach a deadlocked state we have to perform at least 1 action.

If $\widehat{h}(p) = n$, then at least n actions must be performed before reaching a deadlock. Clearly, it is possible that a bigger number of actions is required. Recall the CCS process q in (1). It holds that $\widehat{h}(q) = 0$. This means that optimistically to reach a deadlock no action has to be performed. Actually, we have to perform one action: a communication between a of the first process and $\overline{a}$ of the second one.

It is worth noting that our heuristic function $\widehat{h}$ has an important property: it is easy to compute being syntactically defined.

The following lemma describes some properties of the heuristic function $\widehat{h}$. It asserts that if a process is a deadlocked state, then its $\widehat{h}$-value is equal to zero. Moreover, given a process p, the situation where for each path in $\mathcal{S}(p)$ there is a state whose $\widehat{h}$-value is equal to ∞, would guarantee that p is deadlock free.

Lemma 4.1. *Let p be a CCS process. It holds that*:

(1) p deadlocked state implies $\widehat{h}(p) = 0$;
(2) p deadlock free if for all $s \in \mathcal{D}er(p)$, there exists $t \in \mathcal{D}er(s)$ s.t. $\widehat{h}(t) = \infty$.

Proof. See Appendix A.

From (2) it follows that $\widehat{h}(p) = \infty$ implies p deadlock free.

Notice that the converse of (1) of the above lemma is not true in general. For this, we consider the following example:

$$z \stackrel{\text{def}}{=} ((a.\overline{c}.x|c.y)\backslash\{c\}|\overline{a}.nil)\backslash\{a\},$$
$$x \stackrel{\text{def}}{=} \overline{c}.x,$$
$$y \stackrel{\text{def}}{=} c.y.$$

It holds that $\widehat{h}(z) = 0$, but z is deadlock free. □

The following lemma guarantees that the heuristic function always terminates.

Lemma 4.2. *Let p be a CCS process and s be a state of $\mathcal{S}(p)$. It holds that $\widehat{h}(s)$ always returns a value.*

Proof Sketch. The proof follows by considering that we stop the calculation of the $\widehat{h}$-value when we encounter one of the following:

- a deadlocked state (Rule R1);
- a restricted action (Rule R2);
- a state with more than one possible communication action (Rule R4); and
- a constant already expanded (Rule R7). □

The following theorem states that the heuristic function $\widehat{h}$, defined in Definition 4.1 is admissible, i.e., it never overestimates the actual cost.

Theorem 4.3. *Let p be a CCS process and s be a state of $\mathcal{S}(p)$. It holds that*

$$\widehat{h}(s) \leqslant h(s),$$

where $h(s)$ is the actual cost of a preferred path from s to a goal node.

Proof. See Appendix A. □

Let us define a state evaluation function $f(s) = g(s) + \widehat{h}(s)$, where the cost function g is the distance of the node representing the state s from the start node. Since $\widehat{h}$ is admissible, A* returns a minimal-cost solution graph using f that is a non-decreasing function [52].

Recall the process p of Example 4.1 and let us apply our method. Note that, using A* (see Fig. 3) we stop the node expansion when we select a node s, in the list OPEN, which is deadlocked, that is $s \not\longrightarrow$.

Each step of the A* algorithm with our $\widehat{h}$ function is shown in Fig. 7.

In Fig. 8 the box near each state s_i contains the $\widehat{h}$-value of s_i (i.e., $\widehat{h}(s_i)$), while its f-value (equals to $\widehat{h}(s_i) + g(s_i)$) is included in the circle. For example, it holds that $\widehat{h}(s_3) = 1$ (where $s_3 = d.nil$); this means that to reach a deadlocked state we have to perform at least one action. First we expand the node p: three states are generated: s_1, s_2 and s_3 (see Fig. 8A). In both figures, 8A and 8B corresponding, respectively, to step II and step III of Fig. 7, the shaded nodes represent the states in *OPEN*. We remark the assumption that each arc has a cost equals to 1. Let us compute the f values of the generated nodes. We have: $f(s_1) = g(s_1) + \widehat{h}(s_1) = 1 + \infty = \infty$, $f(s_2) = g(s_2) + \widehat{h}(s_2) = 1 + 2 = 3$

Step	*List*	*Selected node*	*Goal?/path*
I	$OPEN = [p]$	p	no
II	$OPEN = [s_2, s_3]$	s_3	no
III	$OPEN = [s_2, nil]$	nil	yes/$\langle p, s_3, nil\rangle$

Fig. 7. Steps of A* for the process p.

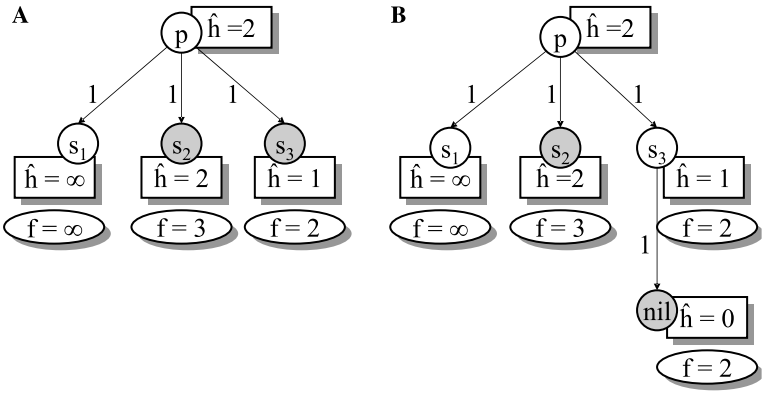

Fig. 8. An example of application of A* with the heuristic function $\widehat{h}$.

and $f(s_3) = g(s_3) + \widehat{h}(s_3) = 1 + 1 = 2$. The infinite f value of s_1 means that this is a safe node and so it does not need to be expanded. Hence only s_2 and s_3 are inserted in the OPEN list. In step II, we choose to expand s_3 that has the least f-value, generating nil, which is inserted in the *OPEN* list (see Fig. 8B). Since $f(nil) = 2 + 0 = 2$, we select that node and as it is a goal node, we stop the construction of the transition system. Therefore, we obtain the shortest path leading to a deadlocked state; this path is $\langle p, s_3, nil\rangle$: we reach a deadlocked state after the execution of two actions. Note that, for this simple example, we have obtained a great reduction: only 4 states and 3 transitions have been generated, while the standard transition system for p has 10 states and 12 transitions (see Fig. 5).

In general, every approach aiming at reducing the state space while generating the transition system may be argued looking at the trade-off between the number of states and the cost of a single transition, which the approach provides. With our method, some states of the standard transition system are only encountered for the calculation of the $\widehat{h}$-value of some state, and not considered later when applying the A* search algorithm. For example, the state $b.nil$ of $\mathcal{S}(a.nil + a.c.b.nil)$, not included in our transition system, occurs in the calculation of the $\widehat{h}$-value of the state $a.nil + a.c.b.nil$. The advantage of considering these states only in the calculation of the $\widehat{h}$-value is that the memory allocated for a calculation can be deallocated once generated the transition, and reallocated for the generation of another transition. In general, the recursive definition of $\widehat{h}$ may weigh on the time complexity of the whole method, but this fact is secondary to our final objective of memory space reduction.

Finally, we point out that if $f = \widehat{h}$, then the A* algorithm represents a greedy search [52]. This type of search may have the advantage of better performances but used alone does not guarantee to find a minimal-cost solution. On the other hand, if we take $f = g$, then our algorithm becomes a Breath First Search (BFS) algorithm, i.e., at each stage, the states at the highest level are expanded first, without any further information. With the BFS alone, the minimality of the counterexample is guaranteed at a possible cost of a more expensive search. A comparison of these techniques is discussed in Section 6.

4.2. Infinite CCS processes

To prove, by model checking, that a CCS process p is deadlock sensitive, the transition system for p, $\mathcal{S}(p)$, is built. Obviously, this approach cannot be straightforwardly used if the process has an infinite representation. In this section we show that our approach can be applied also in these situations.

The following theorem states that if a CCS process p is deadlock sensitive, our approach always finds a deadlocked state even if the process p is infinite.

Theorem 4.4. *Let p be a CCS process. If p is deadlock sensitive, then A* returns a path leading to a deadlocked state*.

Proof. Follows from the completeness of A* [52]. □

Note that no assumption is made on p being finite in the statement of Theorem 4.4.

As an example, let us consider the following CCS process:

$$x \stackrel{\text{def}}{=} (a.nil|b.x) + c.d.nil.$$

Let us suppose that we want to check whether x is deadlock sensitive. Then, $\mathcal{S}(x)$ is infinite. Almost all the existing verification environments (see for example [12,20]) are based on an internal finite-state representation of the processes. A very common condition, required by many verification environments, is the following:

> *the parallel and relabeling operators are allowed inside the body of a process name as long as no process name occurs in the arguments* [46].

Thus, the above environments cannot accept the CCS process x and so it is not possible to find deadlocks in $\mathcal{S}(x)$. Instead, with our approach we can deduce that x is deadlock sensitive. In fact, with A* we prefer to move the second process of the choice, reaching, after the execution of two actions, a deadlocked state. Other approaches for verification of infinite systems can be found in [23,48].

4.3. Distinction between correct termination and deadlock

In the previous section, no distinction is made between correct termination and deadlock. For example, the process $(a.b.nil|\overline{b}.\overline{a}.nil)\{a, b\}$ is a deadlocked process while the process $a.b.nil$ is well terminating. However, for this distinction to be taken into account it is enough to slightly modify

the definition of the heuristics (see Definition 4.1). In particular, we need to modify rule **R1** and the last case of rule **R4**. In Fig. 9, the new definition of $\widehat{h}$ is given.

First we introduce a new symbol ∞^t to represent the correct termination. Its arithmetic behavior is similar to ∞; this means that, for example, if n is a natural number, $n + \infty^t$ is ∞^t, and $\min(n, \infty^t)$ is n.

When we encounter *nil* (Rule **R1**) we return ∞^t, since we have reached a correct termination and so no deadlock has been found. Moreover, in Rule **R4** the last case has been changed as follows: if the $\widehat{h}$-value of each process of the parallel is ∞^t, that is, each of them is well terminating, so is their composition, and therefore we return ∞^t; if not, then a deadlock is possible and this would not be caused by the well-terminating components, if any. Hence we return the sum of the $\widehat{h}$-values of the remaining processes as this could be a finite value.

$$
\begin{array}{lll}
\textbf{R1}. & \widehat{h}(nil, \mathcal{L}, C) & = \infty^t \\
\textbf{R2}. & \widehat{h}(\alpha.p, \mathcal{L}, C) & = \begin{cases} 0 & \text{if } \alpha \in \mathcal{L} \\ 1 + \widehat{h}(p, \mathcal{L}, C) & \text{otherwise} \end{cases} \\
\textbf{R3}. & \widehat{h}(p_1 + p_2, \mathcal{L}, C) & = \min(\widehat{h}(p_1, \mathcal{L}, C), \widehat{h}(p_2, \mathcal{L}, C)) \\
\textbf{R4}. & \widehat{h}(p_1 | \cdots | p_n, \mathcal{L}, C) & = \\
\end{array}
$$

$$
\begin{cases}
\widehat{h}(\mathit{Unfold}^x(p_1 | \cdots | p_n), \mathcal{L}, C \cup \{x\}) & \text{if there exists an unguarded constant } x \text{ in } p_1 | \cdots | p_n \text{ with } x \notin C \\
1 + \widehat{h}(p_1 | \cdots | q | \cdots | p_n, \mathcal{L}, C) & \text{if } p_i \text{ is guarded, } i \in [1..n], \text{ and there exists } i \in [1..n] \text{ s.t. } p_i = \alpha.q, \text{ and } \alpha \notin \mathcal{L} \\
1 + \widehat{h}(p_1 | \cdots | q | \cdots | r | \cdots | p_n, \mathcal{L}, C) & \text{if } p_i \text{ is guarded, } \mathcal{F}\mathit{irst}(p_i) \subseteq \mathcal{L}, \text{ for all } i \in [i..n], \text{ and there exists a unique } \alpha \in \mathcal{L} \text{ s.t. } p_i = \alpha.q,\ p_j = \overline{\alpha}.r, \text{ with } \alpha, \overline{\alpha} \notin \mathcal{F}\mathit{irst}(p_k) \text{ for all } k \neq i \neq j \text{ for some } \alpha \in \mathcal{L} \\
\begin{cases}
\infty^t & \text{if } \widehat{h}(p_i, \mathcal{L}, \emptyset) = \infty^t, \forall i \\
\displaystyle\sum_{\substack{i=1, \\ \widehat{h}(p_i, \mathcal{L}, \emptyset) \neq \infty^t}}^{n} \widehat{h}(p_i, \mathcal{L}, \emptyset) & \text{otherwise}
\end{cases} & \text{otherwise}
\end{cases}
$$

$$
\begin{array}{lll}
\textbf{R5}. & \widehat{h}(p \backslash L, \mathcal{L}, C) & = \widehat{h}(p, \mathcal{L} \cup L \cup \overline{L}, C) \\
\textbf{R6}. & \widehat{h}(p[f], \mathcal{L}, C) & = \widehat{h}(p, f^{-1}(\mathcal{L}), C) \\
\textbf{R7}. & \widehat{h}(x, \mathcal{L}, C) & = \begin{cases} \infty & \text{if } x \in C \\ \widehat{h}(p, \mathcal{L}, C \cup \{x\})) & \text{if } x \notin C \text{ and } x \stackrel{\text{def}}{=} p \end{cases}
\end{array}
$$

Fig. 9. The new $\widehat{h}$ function.

Consider the following two CCS processes:
$p = (a.b.nil|c.d.nil)\backslash\{d\}$,
$q = (a.b.nil|\bar{a}.d.nil)\backslash\{a\}$.
The process p is deadlock sensitive, while q is well terminating. Applying $\widehat{h}$ to p we obtain 3: in fact after the execution of three actions we can reach the deadlocked state $p = (nil|d.nil)\backslash\{d\}$. The value of $\widehat{h}$ on q is ∞^t: this means that the process is well terminating.

5. DELFIN

A prototype tool, named DELFIN (DEadLock FINder), has been implemented in order to test and validate the methodology on bigger examples. The tool is written in Java, it is freely available and can be requested from gradara@unisannio.it. The tool provides a windows user interface easy to use, as shown in Fig. 10. The CCS specifications of processes may be loaded from files selected through a file system browser, and the exploration strategy is chosen through a specific tool option. Namely, it is possible to activate the CWB-NC environment (`CWB` button), the A*-based search, the BFS, or a Greedy search. The result of the search with the trace to the deadlock, if any, and the information on the size of the transition system built will appear on a different panel.

6. Experimental results

This section reports the experimental results obtained by running DELFIN on several CCS processes. The experiments were produced on an Intel Pentium III Xeon with a 902 MHz processor and 2 GB of RAM running Microsoft Windows Server 2003 SE.

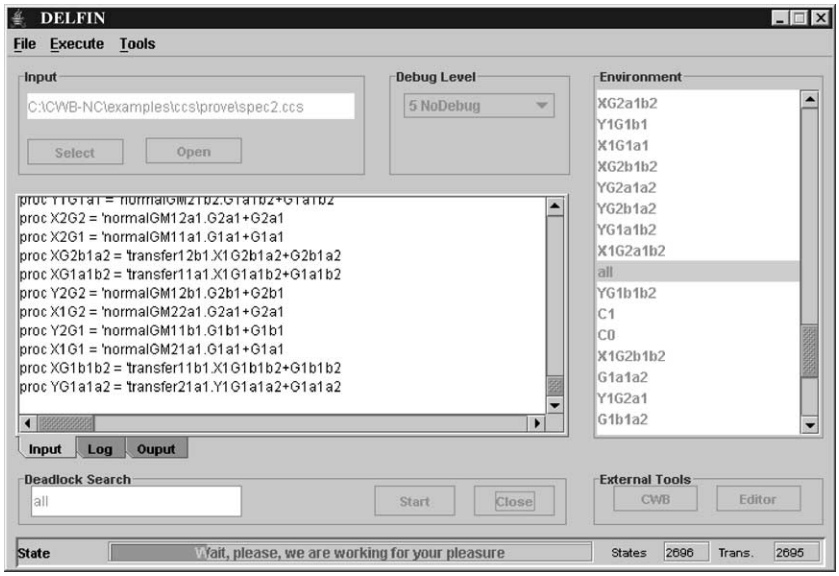

Fig. 10. The user interface of DELFIN.

Two kinds of experiments were made: first we took one well-known example, namely the *dining philosophers* problem, and made several runs of its CCS representation, each time increasing the process size, i.e., the number of philosophers. This example is used as benchmark in almost all the related works. Then, we considered some other deadlock sensitive processes that we found in literature. All of these processes were verified using three different search algorithms, namely: full (depth-first) search, performed by the CWB-NC tool [20], and A* and BFS, performed by the DELFIN tool. The CWB-NC comparison values were generated using version 1.2 of the model checker.

6.1. The dining philosophers

A well-known incorrect solution of the dining philosophers problem is described by a deadlock sensitive CCS process. In this solution, when a philosopher gets hungry, he can pick up his left fork and then the one at his right; if he has both forks, he eats, and then puts the forks down in the same order as he had picked them up.

Table 1 shows the results obtained by running DELFIN on different instances of the dining philosophers problem. More precisely, for different values of `n` (number of philosophers), the table shows the number of states of the standard transition system built by the CWB-NC, the number of states generated if using the BFS strategy, the number of states with A* and, in the last two columns, the state-space reduction of A* with respect to CWB-NC and BFS, respectively. Note that in this example the results achieved with A* are much better than those of the CWB-NC and BFS. In fact, the last does not scale because of the symmetry of the transition system of the dining philosophers process.

We point out that with our methodology we can obtain a great reduction even if the actions of our system are almost all communication actions. Consider the dining philosophers problem; all the actions are actions on which synchronization may occur. Nevertheless, we have obtained a great state-space reduction. Since communication actions are relevant to deadlock detection, in several reduction techniques (see for example [18,26]), communication actions cannot be deleted from the specifications without affecting the deadlock possibilities. Thus, for instance, for the dining philosophers problem only a little reduction is obtained with those methods.

Table 1
Results for the dining philosophers

n	CWB-NC States	BFS States	A* States	State-space reduction A* vs. CWB-NC %	State-space reduction A* vs. BFS %
2	22	12	11	50.00	8.33
3	106	50	36	66.04	28.00
4	506	199	111	78.06	44.22
5	2404	809	358	85.11	55.75
6	11410	3343	1448	87.31	56.69
7	54142	13950	7697	85.78	44.82
8	256898	58636	36728	85.70	37.36
9	__	247613	170632	__	31.08

6.2. CCS deadlock sensitive processes

We selected from the literature various well-known deadlock sensitive systems:

- Alternating Bit Protocol (ABP), a simple version of the protocol consisting of a sender and receiver process communicating over a medium, also modeled as a process. We considered the deadlock sensitive version of the protocol (ABP-safe), written by Cleaveland [19], which is included in the examples of the CWB-NC distribution.
- Mail System, a specification of a mail system, devised by Brebman [13].
- Context Management Application Service Element (CM-ASE), a model of the Application Layer of the Aeronautical Telecommunications Network, developed by Gurov and Kapron [33].
- Transmition Control Protocol (TCP), a specification of the TCP/IP protocol [51]. TCP is defined as a parallel composition of a client, a channel and a server. The channel is defined as a parallel composition of a channel from client to server and a channel from server to client.
- Multicast Protocol for Mobile Computing (MPMC), a protocol for reliable multicast in distributed mobile systems, presented in [1,2].

The results are reported in Table 2.

With these examples we have provided some experimental evidence of the reduction of the state space that may result when applying our methodology instead of the standard model checkers where all the possible states of a process are considered. As the $\hat{h}$ heuristics is admissible, the number of the generated states will always be less than the actual one, according to the specific process. This is always true, except the worst case of expanding all the states, e.g., a process whose state transition graph consists of only one branch and no constant process is encountered. In fact, the presence of constant processes may cause a reduction of the state space even in the case of deadlock-free processes.

In Tables 1 and 2, we highlight the difference between the performance of A* and BFS from our experiments, as both strategies try to find the shortest path leading to a deadlock. The penalty to pay with the BFS is that the cost of the search is exponential in the depth of the optimal solution, and so a common believe in the Artificial Intelligence community is that combining BFS with some heuristics, like in A*, better performances may be achieved in most cases [52]. This, of course, depends on the added-value that the heuristics provides for the process at hand. These thoughts are reflected in the results of our experiments. In fact, we note that the figures related to process MPMC in Table 2 would suggest that, for that kind of process, BFS could be preferred to A*. The reason is that nothing can be said whenever several states exist with the same f-value.

Table 2
Results for some deadlock sensitive systems.

Process	CWB-NC states	BFS states	A* states	State-space reduction A* vs. CWB-NC %	State-space reduction A* vs. BFS %
ABP Protocol	49	18	15	69.39	16.67
Mail System	409	276	254	37.90	7.97
CM-ASE	791	91	89	88.75	2.20
TCP	64	61	57	10.84	6.56
MPMC	4815	3361	3505	27.21	−4.28

Currently, we apply a FIFO strategy too in these cases, but, clearly, the order of state expansion may not be the same as that followed by BFS when the states are at the same level in the transition system. However, it should be noted that, in general, the order of the nodes generated after each expansion is undecidable, and so usually a BFS gives different results on strong equivalent CCS processes.

7. Using the method to check deadlocks in Java programs

The results obtained by applying our approach to some CCS example processes, as the *dining philosophers* problem, are very encouraging. The possible reduction of the state space is significant and this suggests that our technique can be useful to identify deadlocks in concurrent programs. To this aim, we need a way of representing the source code in the CCS specifications. We have analyzed programs written in Java [35] and defined a Java-to-CCS transform operator that is described in this section. Operators on other programming languages could be easily defined following the same idea. A similar work on Java programs is presented in [15]. However, our method seems to be more systematic, the translation rules are modular and have been thought with the aim of performing them automatically. In fact, we have realized a prototype tool, named Java2 CCS that currently only implements the translation rules presented in this paper.

7.1. Threads synchronization in Java

The essential Java constructs for concurrent programming are briefly recalled in this section. The reader can refer to [41] for further details.

The *critical section* of a program is a piece of code executed by different concurrent threads to manipulate a shared resource. Therefore synchronization should be explicitly programmed to maintain data consistency and avoid deadlocks. The Java specifications language provides mechanisms to solve this problem using the concept of *monitor*. A monitor is a critical section which is guarded by a mutual-exclusion semaphore associated with any object. Only one thread at the time can own the *lock* (or mutex) of an object. If a second thread tries to acquire the lock of the same object, it will be blocked until the owning thread releases the lock. The monitor is realized in Java by the word *synchronized*, which applies to the object and may refer to a method or a section inside a method. A Java class may contain one or more synchronized methods whose execution is guarded by the availability of the object lock.

Mechanisms for threads to wait for, or to be notified of, a certain value of the shared resource (*condition variable*) are also provided by the language. The Java class Object offers methods as *wait( )* and *notify( )* to this aim that need to be included in synchronized sections of the resource class. The execution of the *wait( )* method causes the lock of the shared object to be released and the calling thread to be blocked until the *notify( )* method is executed on the same object by some other thread. Each object has a *wait set* associated with it collecting all threads that are waiting to be notified. Whenever a notify action is issued and the lock of the object is released, one of the blocked threads leaves the wait set and may proceed with its execution by trying first to re-acquire the lock of the object. It should be noted that a *notify( )* method executed with an empty wait set does not have any effect.

7.2. Transforming a Java program into a CCS process

In this section, we present our methodology for describing Java programs through CCS. Here, we focus on multi-threading programs as we are interested in detecting possible deadlocks from the associated CCS. We also show an application of the methodology on a simple example.

The methodology is based on the following two assumptions:

- the number of objects in the program is statically defined; and
- each set of variable values is finite. This can be achieved through suitable abstraction schemes like for example data abstraction [22,27] or predicate abstraction [4,24]. Abstraction techniques have to be applied to make model checking feasible, since most software programs have an infinite number of states. In this paper, we concentrate on the boolean data abstraction, which has been proven to be still quite efficient in this context.

The first assumption has as a consequence that (non-static) state variables and methods of a class are replicated for each object and translated singularly into CCS processes. The second constraint is to assure a finite number of states for both variables and methods using them.

All objects, variables, and methods have an index. Namely,

- $\mathcal{V}$ is the index set for variables; and
- $\mathcal{O}$ for objects and methods.

Variables include object state variables and local variables passed to, or used inside, a method. Threads are special objects whose indexes belong to the set $\mathcal{TH}$, with $\mathcal{TH} \subset \mathcal{O}$. Methods and the Java concepts for synchronization that we have formalized, such as the object lock and wait set, have the same identifier as the object to which they refer.

Before describing the Java-to-CCS transformation we introduce the following operator to the CCS algebra, to express the execution order of two processes.

Definition 7.1 (*The sequence operator* ;)**.** Let p and q be two CCS processes. The sequence operator ; is defined as:

$$p\,;q \stackrel{\text{def}}{=} p\{q/nil\}$$

with the effect of replacing each occurrence of nil in p, and in the bodies of all the contained constants, with the process q.

In the remaining of this section, we define a Java-to-CCS transform operator $\mathcal{T}$ that directly applies to the Java code of a program and translates it into CCS process specifications.

7.3. Definition of variables and methods processes

The definitions of the processes corresponding to each variable and method are shown in Fig. 11.

$$\mathcal{T}(boolean\ VAR_i) = VAR\mathtt{ff}_i \stackrel{\text{def}}{=} \overline{isFalse_i}.VAR\mathtt{ff}_i + setFalse_i.VAR\mathtt{ff}_i + setTrue_i.VAR\mathtt{tt}_i$$

$$VAR\mathtt{tt}_i \stackrel{\text{def}}{=} \overline{isTrue_i}.VAR\mathtt{tt}_i + setTrue_i.VAR\mathtt{tt}_i + setFalse_i.VAR\mathtt{ff}_i$$

$$\mathcal{T}(main()\{I\}) = MAIN \stackrel{\text{def}}{=} \mathcal{T}(I)$$

$$\mathcal{T}(methodName_i()\{I\}) = METHODNAME_i \stackrel{\text{def}}{=} callMethodName_i.\mathcal{T}(I); \overline{returnMethodName_i}.METHODNAME_i$$

$$\mathcal{T}(run_i()\{I\}) = RUN_i \stackrel{\text{def}}{=} callRun_i.\mathcal{T}(I)$$

Fig. 11. Definitions of variables and methods.

Boolean variable definition. Each time, the boolean variable VAR_i may be seen as the process corresponding to its current value. We have defined two processes, $VAR\ \mathtt{ff}_i$ and $VAR\mathtt{tt}_i$, accordingly. For example, the branch $\overline{isFalse_i}$. $VAR\ \mathtt{ff}_i$ represents the false state of the variable; this state will change to true after the execution of the action $setTrue_i$ and the process will continue as $VAR\ \mathtt{tt}_i$.
Main definition. The method *main*() is translated as the process *MAIN*, where $\mathcal{T}$ is applied to the instructions contained in *I* according to the rules of Fig. 12.
Methods definition. Every method corresponds to a process waiting to be invoked through the action $callMethodName_i$.

The action $\overline{returnMethodName_i}$ causes the control being returned to the method caller. Then, the process goes back to the starting action so that it can be invoked again. A special case is the process RUN_i, corresponding to the method $run_i()$ of a thread object, which can only be executed once.

7.4. Instructions

The translations of the main Java instructions are contained in Fig. 12.

Sequence of two Java instructions. The execution order of two Java instructions is expressed by the use of the operator ; (introduced in Definition 7.1) to sequentialize the translated processes. We assume that these instructions are contained in the definition of some $methodName_i()$.

$$\mathcal{T}(I_1; I_2) = \mathcal{T}(I_1); \mathcal{T}(I_2)$$

$$\mathcal{T}(VAR_i = true;) = \overline{setTrue_i}.nil$$

$$\mathcal{T}(while\ (VAR_i == true)\ I) = WHILE$$

where

$$WHILE \stackrel{\text{def}}{=} isTrue_i.\mathcal{T}(I); WHILE + isFalse_i.nil$$

$$\mathcal{T}(if\ (VAR_i == true)\ I_1; I_2) = IF$$

where

$$IF \stackrel{\text{def}}{=} isTrue_i.\mathcal{T}(I_1) + isFalse_i.\mathcal{T}(I_2)$$

$$\mathcal{T}(methodName_i();) = \overline{callMethodName_i}.returnMethodName_i.nil$$

$$\mathcal{T}(run_i();) = \overline{callRun_i}.nil$$

$$\mathcal{T}(synchronized\ (this)\{I\}) = \overline{lock_i}.\mathcal{T}(I); \overline{unlock_i}.nil$$

Fig. 12. Instructions.

The translation of the assignment operations and of the “while" and “if" constructs reflects their meaning in the programming theory. In particular, they are synchronized with some variable process.
Assignment operations. The assignment of the true value to the variable VAR_i is performed by a process that executes the action $\overline{setTrue_i}$ and then terminates. The assignment of the false value is defined similarly.
While construct. The $WHILE$ process simulates the while control construct. Namely, the current value of VAR_i is checked through the actions $isTrue_i$ and $isFalse_i$: depending on what action is performed, the process corresponding to the block instruction I is activated or not.
If construct. The IF process simulates the if control construct interrogating the current value of VAR_i: either the translation of the block instruction I_1 or that corresponding to I_2 is executed, depending on the performed action ($isTrue_i$ or $isFalse_i$, respectively).
Method call. The call to the method “$methodName_i$” through the object i is performed by the action $\overline{callMethodName_i}$. The control to the caller process is returned through the action $returnMethodName_i$. Instead, for the $run_i()$ method the control is immediately returned to allow for a parallel execution of the processes.

Java synchronized block or method. Before translating I, the lock of the object i must be acquired, through the action $\overline{lock_i}$; the action $\overline{unlock_i}$ follows the translation of I to release the lock (see Fig. 13).

7.5. Concurrency constructs and methods

The translation of the Java constructs and methods for concurrency is shown in Fig. 13. Namely, the process $LOCK$ is to describe the *monitor* concept, the processes $WAIT$ and $NOTIFY$ correspond to the *wait( )* and *notify( )* methods, and the processes $WAITSET$ represent the *wait-set* management associated with any object. A careful analysis of the Java language specifications has been made in order to represent these constructs.

Lock. This process simulates the lock and unlock mechanisms of the object i allowing concurrent threads to execute synchronized instructions after the action $lock_i$.

$$LOCK_i \stackrel{\text{def}}{=} lock_i.unlock_i.LOCK_i$$

$$WAIT_i \stackrel{\text{def}}{=} callWait_i.\overline{insert_i}.\overline{unlock_i}.resume_i.\overline{lock_i}.\overline{returnWait_i}.WAIT_i$$

$$WAIT_i \stackrel{\text{def}}{=} callWait_i.\overline{insert_i}.\overline{unlock_i}.resume_i.\overline{returnWait_i}.WAIT_i$$

$$\begin{aligned} NOTIFY_i \stackrel{\text{def}}{=}\ & callNotify_i. \\ & (someoneInQueue_i.\overline{resume_i}.\overline{remove_i}.\overline{returnNotify_i}.NOTIFY_i + \\ & noneInQueue_i.\overline{returnNotify_i}.NOTIFY_i) \end{aligned}$$

$$\begin{aligned} WAITSET(0)_i \stackrel{\text{def}}{=}\ & insert_i.WAITSET_i(1) + \\ & \overline{noneInQueue_i}.WAITSET(0)_i \end{aligned}$$

Let j be such that $0 < j < n-1$. Then

$$\begin{aligned} WAITSET(j)_i \stackrel{\text{def}}{=}\ & remove_i.WAITSET(j-1)_i + \\ & \overline{someoneInQueue_i}.WAITSET(j)_i + \\ & insert_i.WAITSET(j+1)_i \\ WAITSET(n-1)_i \stackrel{\text{def}}{=}\ & remove_i.WAITSET(n-2)_i + \\ & \overline{someoneInQueue_i}.WAITSET(n-1)_i \end{aligned}$$

Fig. 13. Definitions to handle synchronization.

Wait. The call to the method *wait( )* always occurs in a synchronized block. This causes the thread being inserted in the wait-set of the object through the action $\overline{insert_i}$ that will be caught by one of the $WAITSET(j)_i$ processes, depending on the number j $(0 < j < n)$ of the waiting threads. Then, the object lock is released through the action $\overline{unlock_i}$, so that the object i may be used again. Then, $WAIT_i$ waits for the action $\overline{resume_i}$ from the process $NOTIFY_i$ and executes the action $\overline{lock_i}$ before returning the control and accepting a new invocation.
Notify. The call to the method *notify( )* sets free one of the threads in the wait-set of the object i, if any. The actions $someoneInQueue_i$ and $noneInQueue_i$, communicating with the $\overline{someoneInQueue_i}$ and $\overline{noneInQueue_i}$ actions of the $WAITSET(j)_i$ process definitions, show the state of the wait-set. If this is empty, nothing is done, whereas if the wait-set contains at least one thread, the actions $\overline{resume_i}$ (caught by the $WAIT_i$ process) and $\overline{remove_i}$ will resume and remove from the wait-set one of the waiting threads.
Wait-set. We assume that the program consists of at most n threads. So the wait-set associated with the shared object i may contain at most $n-1$ threads at the time. The translation of this concept consists of n $WAITSET$ processes. Namely, $WAITSET(0)_i$ represents an empty wait-set, while $WAITSET(n-1)_i$ represents a full wait-set. The insertion of a thread in the wait-set is required through the action $\overline{insert_i}$ performed by the process $WAIT_i$.

7.6. The final CCS process

The final CCS process describing a Java program is obtained through the proposed methodology as a parallel of the $MAIN$ process and all the RUN, VAR, $LOCK$, $WAIT$, $NOTIFY$, $WAITSET$ and $METHODNAME$ processes, restricted to all the actions used for communicating among such processes.

As previously defined, $\mathcal{V}$, $\mathcal{O}$ and $\mathcal{TH}$ are the index sets of variables, objects (including threads), and threads. Moreover, let $\mathcal{N}$ be the set of methods names.

Given the set $\mathcal{A} = \{1, ..., s\}$, we define

$$|_{i\in\mathcal{A}}\ (process_i) = (process_1| \ ... \ |process_s)$$

and, if $k > 0$,

$$(|\ process_i)^{(k)} = (process_i| \ ..... \ |process_i).$$

With the proposed notation, the CCS process describing a Java program is the following:

$$\begin{aligned}\mathcal{P} = \big(&MAIN|\\ &|_{i\in\mathcal{TH}}\ (run_i)|\\ &|_{i\in\mathcal{V}}\ (VAR_i)|\\ &|_{i\in\mathcal{O}-\mathcal{TH}}\ (LOCK_i|(WAIT_i)^{(|\mathcal{TH}|)}|NOTIFY_i|WAITSET_i(0)|\\ &|_{i\in\mathcal{O},MethodX\in\mathcal{N}}\ (MethodX_i)^{(|\mathcal{TH}|)}\big) \setminus \mathcal{R},\end{aligned}$$

where $\mathcal{R}$ is the restriction set of the whole process, including all the actions used for internal communication among the parallel processes:

$$\mathcal{R} = \bigcup_{i\in\mathcal{V}} \{isTrue_i, isFalse_i, setTrue_i, setFalse_i\} \cup \bigcup_{i\in\mathcal{O}-\mathcal{TH}} \{lock_i, unlock_i, callWait_i, returnWait_i, callNotify_i, returnNotify_i\} \cup \bigcup_{i\in\mathcal{O}-\mathcal{TH}} \{insert_i, resume_i, someoneInQueue_i, NoneInQueue_i\} \cup \bigcup_{i\in\mathcal{O},MethodX\in\mathcal{N}} \{callMethodX_i, returnMethodX_i\}.$$

It is worth noting that since the default Java initialization of a boolean variable is the false value, the CCS translation of the variable is the $VARff_i$ process. Moreover, any method of the ith object may be invoked by more than one thread at the time. This fact is expressed by considering as many copies of the process $methodName_i$ as the number of the potential caller threads, i.e., $|\mathcal{TH}|$ at most. This number could be exactly determined by performing a static analysis of the source code, thus reducing the number of states of the process.

The proof of the correctness of the translation is ongoing and discussed in [36].

7.7. An example

We consider the classical example of the dining philosophers Java program with deadlock, of which we present the code and the CCS translation. When a philosopher gets hungry, he tries to pick up his left fork first, and then the one at his right. After eating, both forks are released.

```
class Philosopher extends Thread {
    private int id;
    private Fork leftFork;
    private Fork rightFork;

    public Philosopher(int i, Fork lf, Fork rf){
        id=i;
        leftFork=lf;
        rightFork=rf;
        }

    public void run(){
        while (true){
                leftFork.getFork();
                rightFork.getFork();
                leftFork.putFork();
                rightFork.putFork();
        }
    }
}
```

Fig. 14. The dining philosophers Java program (I).

```
class Fork {
    private int i;
    private boolean isBusy = false;

    public Fork(int j){
        i=j;
    }

    public synchronized void getFork(){
        while (isBusy==true){
                try{
                    wait();
                }catch(InterruptedException e)
                      {System.out.println("Interrupted");}
       }
       isBusy=true;
    }

    public synchronized void putFork(){
        isBusy=false;
        notify();
    }
}

public class DiningPhilosophers {
    public static void main(String [] args) throws Exception
    {
        Fork f1= new Fork(1);
        Fork f2= new Fork(2);
        Philosopher ph1 = new Philosopher(1,f1, f2);
        Philosopher ph2 = new Philosopher(2,f2,f1);
        ph1.start();
        ph2.start();
    }
}
```

Fig. 15. The dining philosophers Java program (II).

For simplicity, we use only two concurrent threads representing the philosophers and two forks as the shared resources. The code is shown in Figs. 14 and 15. The resulting CCS process is shown in Fig. 16.

8. Conclusion and related work

The aim of this work is to reduce the state explosion problem in the context of model checking. Thus, we have proposed a method that uses the A* algorithm for deadlock detec-

For $i = 1, 2$ we have

$$proc\ GETFORK_i \stackrel{\text{def}}{=} callGetfork_i.\overline{lock_i}.Xi$$

where

$$proc\ Xi \stackrel{\text{def}}{=} isTrue_i.\overline{callWait_i}.returnWait_i.Xi + isFalse_i.\overline{setTrue_i}.\overline{unlock_i}.\overline{returnGetfork_i}.GETFORK_i$$

$$proc\ PUTFORK_i \stackrel{\text{def}}{=} callPutfork_i.\overline{lock_i}.\overline{setFalse_i}.\overline{callNotify_i}.returnNotify_i.\overline{unlock_i}.\overline{returnPutfork_i}.PUTFORK_i$$

$$proc\ RUN_3 \stackrel{\text{def}}{=} callRun_3.WHILETrue_3; nil$$

where

$$proc\ WHILETrue_3 \stackrel{\text{def}}{=} callGetfork_1.returnGetfork_1.\overline{callGetfork_2}.returnGetfork_2.\overline{callPutfork_1}.returnPutfork_1.\overline{callPutfork_2}.returnPutfork_2.WHILETrue_3$$

$$proc\ RUN_4 \stackrel{\text{def}}{=} callRun_4.WHILETrue_4; nil$$

where

$$proc\ WHILETrue_4 \stackrel{\text{def}}{=} callGetfork_2.returnGetfork_2.\overline{callGetfork_1}.returnGetfork_1.\overline{callPutfork_2}.returnPutfork_2.\overline{callPutfork_1}.returnPutfork_1.WHILETrue_4$$

$$proc\ MAIN \stackrel{\text{def}}{=} \overline{callRun_3}.\overline{callRun_4}.nil$$

Fig. 16. The dining philosophers CCS specifications.

tion in concurrent systems described in CCS. The novel contributions of our work are the following.

- The definition of an admissible heuristic function. In this way, by applying A* we can always find the shortest path leading to a deadlock.
- The heuristics is syntactically defined, i.e., it is only based on the CCS specifications of the process, and the proposed method is completely automatic, thus it does not require user intervention and manual efforts.

- A method is provided in order for our methodology to be applicable also to verify programs, for example Java programs.

The performances of our method have been compared with those of some existing techniques that are generally used. As a future work, we will investigate the use of more accurate heuristic functions; in fact as the accuracy of the heuristics improves, the amount of search required to find a solution and the cost of the resulting solution both decrease. Moreover, we will evaluate the efficiency of the algorithm on relevant problems in software verification. In general, our approach achieves good results since we concentrate on a single property, i.e., deadlock detection.

We now compare our method with some related works aiming to state-space reduction for deadlock analysis. First of all, we recall that static deadlock checking in concurrent systems is an intractable problem [62]: i.e., each algorithm has an exponential complexity, in the general case. Thus, a method can be more efficient than another one in some cases and less efficient in some other cases.

Deadlocks checking from CCS specifications was previously tackled, by one of the authors of this paper, from a semantic point of view [25], by defining a non-standard reducing semantics of CCS and from a syntactic point of view [26], by defining syntactic transformation of a CCS specification; the transformation deletes from the specification occurrences of actions that do not affect the deadlock possibilities of the specification. The approach presented here in many cases may be better. For example, consider the *dining philosophers* example shown in Section 6.1: using the approaches in [25,26], a little reduction is obtained, since almost all actions are restricted. Moreover, after the detection of a deadlock, it can be useful for debugging to examine a path from the initial state to the deadlocked state in order to determine the cause of the deadlock. When using the approach defined in [25,26], this information cannot be retrieved, due to the deletion and modification of the actions occurring in the original program. In this paper, we overcome this problem.

The approaches described in [8,58], start from generating the standard transition system corresponding to a concurrent system, and then they reduce it, obtaining a transition system with fewer states. Such approaches require a lot of memory to store the standard transition system and a lot of time to reduce it.

In [55], to check deadlock freeness of a CCS process, the process is first transformed into a Petri net [54] and then the obtained net is reduced using well-known reduction techniques for Petri nets. Apart the obvious considerations on its low uniformity (a different formalism, Petri nets, is used), the approach has, as the previous ones, the drawback that memory must be allocated to store the whole Petri net corresponding to a CCS process.

Several other approaches have been developed to solve the state explosion problem. They are general methods that can be used to reduce the state-space explosion while verifying a set of properties, included deadlock detection.

Reduction techniques based on process equivalences [11,37] overcome the state explosion problem by using minimization with respect to an equivalence relation, characterized by the fact that a minimal semantically equivalent representation of the global system is constructed. This minimal representation can subsequently be used for all kinds of verification. However the minimization is typically $\mathcal{O}(n \log n)$ or $\mathcal{O}(n^2)$, where n is the number of states of the unreduced graph.

In *symbolic model checking techniques* [47] the state space is represented symbolically by a logical formula captured using a Binary Decision Diagram [14]. This technique has been proved successful especially in verifying hardware designs with regular logical structures. For such systems, BDD representations can reduce the state space from exponential order of the number of state variables to linear. However, similar reductions are not obtained in software specifications whose logical structures are less regular.

In *on-the-fly techniques*, see for example [42], and *local model checking approaches*, see for example [60], in order to verify a property, the whole transition system is not generated, but the property is checked during the generation. Roughly speaking, the idea is that a depth-first traversal of the transition system can be performed, and only the current path has to be stored, while memory is deallocated when verifying another path. These approaches give no hint about which states we must visit first, therefore, in most cases almost all the states must be generated.

The works [31,34,53,63] follow the *partial order* approach to model checking, in which only a representative is considered among all the interleaving of actions generated by a parallel composition. However, this method alone is not effective when almost all actions are communications. As the partial order method is complementary to our approach, a combination of the two can be realized. In [21], partial order and symbolic model checking have been evaluated to solve the state explosion problem during deadlock detection. In [61], a hierarchy for a set of transition systems is defined; based on this hierarchy, subsets of transition systems are composed and minimized (w.r.t. observational equivalence). A different approach for efficient deadlock detection is *simultaneous reachability analysis*, which allows a transition to be associated with a set of concurrent actions [44]. The method defined in this paper is orthogonal with respect to all the methods mentioned above and can be usefully integrated with them.

Abstraction approaches (see for example [16]) reduce the number of states by ignoring some state information. The idea behind the abstraction is that instead of verifying a property φ of a system S, we verify φ of the system S_{abs}, which is an abstraction of the system S. Typically, the abstractions are *conservative*: S_{abs} satisfies φ implies S satisfies φ, but not necessarily the converse. The approach described in this paper does not have this limitation.

Recently, great interest was shown in combining the two areas: model checking and heuristics to guide the exploration of the state graph of a system. In the domain of software validation, the work of Yang and Dill [64] is one of the most original. They enhance the bug-finding capability of a model checker by using heuristics to search the states that are most likely to lead to an error. In [32] genetic algorithms are used to exploit heuristics for guiding a search in large state spaces towards errors like assertion violations. In [6] heuristics have been used for real-time model checking in UPPAAL. In [3,50] heuristic search has been combined with on-the-fly techniques, while in [9,10,30,43] has been combined with symbolic model checking. In [45] heuristic search techniques are combined with partial order reduction methods, but optimality of the path to a deadlocked state is not preserved.

Some experimental evidence is given in literature that heuristic-based search may perform better than the depth-first search technique generally used by standard model checkers. However, the state-space reduction strongly depends on the quality of the estimate. In [38], Groce and Visser explore heuristic model checking of software written in the Java programming language and use heuristics derived from the world of software testing to search for assertion violations and dead-

locks. In particular, they consider structure-based evaluation functions such as branch covering percentage and thread-interleaving. It is worth noting that none of these heuristics is admissible, thus A* lacks of optimality. Moreover, our method is not closely dependent on a programming language since it is defined on a process algebra, i.e., CCS. Therefore it is a quite general methodology and can be applied to every programming language with the only constraint to transform the program into a CCS process or equivalently into a transition system. In Section 7 we have only shown an example of transformation for multi-threaded JAVA program. Also, this transformation is quite general and it can be easily extended to whatever language. In fact, one of the most appealing features of the transformation is its modularity: adding new programming instructions can be achieved by introducing other transformation rules. The correctness of the new rules can be "locally" proved.

In [28,29] some other heuristics for deadlock detection are proposed. Namely, in [28] two estimation functions are experimented for directed model checking of (unknown) error states: one consists of counting the number of active (or non-blocked) processes, and the other is a formula-based heuristic where the deadlock formula is inferred from the user designated dangerous states. In [29] heuristics are only used to improve the trails leading to the (known) error states that may be found through depth-first model checking or other techniques like simulation, test, random walk. In this case, the heuristic may be directly defined through state distance functions, like the Hamming Distance or the Finite State Machine Distance. In this paper, we advocate the use of A* both to explore the state space, through an estimation that is made from the structure of the process (hence more informative), and to retrieve the shortest trail, guaranteed by the admissibility of the heuristic.

In [57], one of the authors of this paper has presented an attempt to combine heuristic search and ideas taken from local model checking, using the selective mu-calculus logic [5]. The method is completely automatic, but it leads to good results only for properties concerning precedence relations between actions. Thus, for some kinds of formula (when the property involves almost all actions such as deadlock freeness), no advantage is obtained.

Acknowledgments

The authors thank Antonio Serino and Enrico Romano for their help with implementation of the tools.

Appendix A

Proof of Lemma 4.1

Let p be a CCS process. It holds that:

(1) p deadlocked state implies $\widehat{h}(p) = 0$;
(2) p deadlock free if for all $s \in \mathcal{D}er(p)$, *there exists* $t \in \mathcal{D}er(s)$ *s.t.* $\widehat{h}(t) = \infty$.

Proof. We prove the first point. The proof is made by induction on the structure of the process p.

Base step. *nil*: straightforward.
Inductive step. Let us assume that the lemma holds for q and r.
$p = \alpha.q$. p is not a deadlocked state, since at least the action α can be performed.
$p = q + r$. By Definition 3.2, it holds that p is a deadlocked state if and only if both q and r are deadlocked states. The thesis holds by inductive hypothesis.
$p = q|r$. Similar to the above case.
$p = q\backslash L$. By Definition 3.2, it holds that p is a deadlocked state if and only if either q is a deadlocked state or q can perform an action in $L \cup \overline{L}$. In the first case the thesis holds by inductive hypothesis, in the second one, the thesis holds by definition of $\widehat{h}$: when an action belongs to a restriction environment, the heuristic function returns 0.
$p = q[f]$. By Definition 3.2, it holds that p is a deadlocked state if and only if q is a deadlocked state. The thesis holds by inductive hypothesis.
$p = x$. Similar to the above case.
The second point can be proved similarly using the Definition 3.2. □

Proof of Theorem 4.3
Let p be a CCS process and s be a state of $\mathcal{S}(p)$. It holds that

$$\widehat{h}(s) \leqslant h(s),$$

where $h(s)$ is the actual cost of a preferred path from s to a goal node. □

Proof. We prove that $\widehat{h}(p, \mathcal{L}, C) \leqslant h(p)$, where $\mathcal{L} \subseteq \mathcal{V}$ and C a set of constants, by induction on the structure of the process p. The interesting cases involve the action prefix and the parallel composition, so we consider only these two cases.
Base step. *nil*: straightforward.
Inductive step. Let us assume that the theorem holds for each q_i with $i \in [1..n]$.
$p = \alpha.q_1$ (Rule R2 in Fig. 6). Two cases exist: $\alpha \notin \mathcal{L}$ and $\alpha \in \mathcal{L}$. In the first case, since $p = \alpha.q_1 \xrightarrow{\alpha} q_1$, it holds that $\widehat{h}(p, \mathcal{L}, C) = 1 + \widehat{h}(q_1, \mathcal{L}, C)$. The thesis follows by inductive hypothesis (i.e., $1 + \widehat{h}(q_1, \mathcal{L}, C) \leqslant 1 + h(q_1)$). In the second case, α is a restricted action and $\widehat{h}(p, \mathcal{L}, C) = 0$. Consider now the real moves of p. If p can perform an action, then $h(p) \geqslant 1$, which is greater than zero: the theorem is obviously true. Otherwise, if p can perform no actions (i.e $p \not\longrightarrow$), then $h(p) = 0$, which is equal to $\widehat{h}(p, \mathcal{L}, C)$: also in this case the theorem is true.
$p = q_1| \cdots |q_n$ (Rule R4 in Fig. 6). Three cases may occur:

(1) suppose that a guarded process q_i exists that can perform an unrestricted action α with the form $q_i = \alpha.r$ and $\alpha \notin \mathcal{L}$ [2] . Remember Remark 3.1. Only **Par** rule (see Fig. 4) can be applied to move q_i. Note that **Com** rule cannot be applied to move q_i; in fact, if $\overline{\alpha}$ can be performed by a process q_j, with $j \neq i$, then $\alpha \in \mathcal{L}$. Thus, $h(p) = 1 + h(q_1| \cdots |q_{i-1}|r|q_{i+1}| \cdots |q_n)$. By Definition 4.1, it holds that $\widehat{h}(p, \mathcal{L}, C) = 1 + \widehat{h}(q_1| \cdots |q_{i-1}|r|q_{i+1}| \cdots |q_n, \mathcal{L}, C)$. The thesis follows by inductive hypothesis.
(2) Suppose that only two different processes q_i and q_j exist that can perform only a restricted action α and $\overline{\alpha}$, respectively. More precisely, $q_i = \alpha.r$, $q_j = \overline{\alpha}.s$ and $\alpha, \overline{\alpha} \notin \mathcal{L}$. Moreover,

[2] If such process does not exist, we unfold an unguarded constant $x \notin C$ in $q_1| \cdots |q_n$, as stated by Definition 4.1.

no other process, different from q_i and q_j, can perform α or $\overline{\alpha}$. It holds (see Fig. 4) that $p \xrightarrow{\tau} q_1|\cdots|r|\cdots|s|\ldots|q_n$. Thus, $h(p) = 1 + h(q_1|\cdots|r|\cdots|s|\ldots|q_n)$. The thesis follows by inductive hypothesis and by Definition 4.1.

(3) In all the other cases, $\widehat{h}(p) = \sum_{i=1}^{n} \widehat{h}(q_i, \mathcal{L}, \emptyset)$. Recall the assumption made in Section 3 (Remark 3.1). With this assumption it is not possible to apply, at the same time, both **Com** rule and **Par** rule (see Fig. 4), involving a same action. Thus, the thesis holds, since the number of actual moves of p is greater or equal to the sum of the $\widehat{h}$-value of the parallel components. □

References

[1] G. Anastasi, A. Bartoli, N. De Francesco, A. Santone, Efficient verification of a multicast protocol for mobile computing, Comput. J. 44 (1) (2001) 21–30.

[2] G. Anastasi, A. Bartoli, F. Spadoni, Group multicast in distributed mobile systems with unreliable wireless network, in: Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems (SRDS'99), Lausanne (CH), October 18–21, 1999.

[3] R. Alur, B.-Y. Wang. "Next" heuristic for on-the-fly model checking, in: Proceedings of the 10th International Conference on Concurrency Theory (CONCUR'99), Lecture Notes in Computer Science, vol. 1664, 1999, pp.98–113.

[4] T. Ball, R. Majumdar, T. Millstein, S.K. Rajamani, Automatic predicate abstraction of C programs, in: Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2001), ACM, New York, 2001, pp. 203–213.

[5] R. Barbuti, N. De Francesco, A. Santone, G. Vaglini, Selective mu-calculus and formula-based abstractions of transition systems, J. Comput. Syst. Sci. 59 (3) (1999) 537–556.

[6] G. Behrmann, A. Fehnker, T.S. Hune, K. Larsen, P. Petterson, J. Romijn, Efficient guiding towards cost-optimality in UPPAAL, in: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01), Lecture Notes in Computer Science, vol. 2031, 2001, pp. 174–188.

[7] B. Beizer, Software Testing Techniques, International Thomson Computer Press, 1990.

[8] S. Bensalem, A. Bouajjani, C. Loiseaux, J. Sifakis, Property preserving simulations, in: Proceedings of the Fourth Workshop on Computer Aided Verification (CAV'92), Lecture Notes in Computer Science, vol. 663, 1992, pp. 260–273.

[9] P. Bertoli, A. Cimatti, J. Slaney, S. Thiebaux, Solving power supply restoration problems with planning via symbolic model-checking, in: Proceedings of the Sixth International Conference on AI Planning and Scheduling (AIPS'02), AAAI Press, New York, 2002, pp. 23–29.

[10] R. Bloem, K. Ravi, F. Somenzi, Efficient decision procedures for model checking of linear time logic properties, in: Proceedings of the 11th International Conference on Computer-Aided Verification (CAV'99), Lecture Notes in Computer Science, vol. 1633, 1999, pp. 222–235.

[11] A. Bouajjani, J.C. Fernandez, N. Halbwachs, Minimal model generation, in: Proceedings of the International Conference on Computer-Aided Verification (CAV'90), Lecture Notes in Computer Science, vol. 531, 1990, pp. 197–203.

[12] A. Bouali, S. Gnesi, S. Larosa, The integration project for the JACK environment, Bull. EATCS 54 (1994) 207–223.

[13] G.J. Brebman, A CCS-based investigation of deadlock in a multi-process electronic mail system, Formal Aspect Comput. 5 (5) (1993) 467–478.

[14] R.E. Bryant, Graph-based algorithms for boolean function manipulation, IEEE Trans. Comput. C-35 (8) (1986) 677–691.

[15] J. Chen, On verifying distributed multithreaded Java programs, in: Proceedings of the 33rd Annual Hawaii International Conference on System Sciences (HICSS-33), 2000.

[16] E.M. Clarke, O. Grumberg, D.E. Long, Model checking and abstraction, ACM Trans. Progr. Languages Syst. 16 (5) (1994) 1512–1542.
[17] E.M. Clarke, O. Grumberg, D. Peled, Model Checking, MIT Press, Cambridge, MA, 2000.
[18] E.M. Clarke, D.E. Long, K.L. McMillan, Compositional model checking, in: Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science, 1989, pp. 353–362.
[19] R. Cleaveland, The Concurrency Workbench of North Carolina, Alternating Bit Protocol (ABP). <www.dcs.ed.ac.uk/home/cwb/Examples/ccs/abp.cwb>.
[20] R. Cleaveland, S. Sims, The NCSU Concurrency Workbench, in: Proceedings of the Eighth International Conference on Computer-Aided Verification (CAV'96), Lecture Notes in Computer Science, vol. 1102, 1996, pp.394–397.
[21] J. Corbett, Evaluating deadlock detection methods for concurrent software, IEEE Trans. Software Eng. 22 (3) (1996) 161–180.
[22] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, H. Zheng, Bandera: extracting finite-state models from Java source code, in: Proceedings of the 22nd International Conference on Software Engineering 2000, ACM Press, New York, 2000, pp. 439–448.
[23] M. Dam, Compositional proof systems for model checking infinite state processes, in: Proceedings of the 6th International Conference on Concurrency Theory (CONCUR'95), Lecture Notes in Computer Science, vol. 962, 1995, pp. 12–26.
[24] S. Das, D.L. Dill, S. Park, Experience with predicate abstraction, in: Proceedings of the 11th International Conference on Computer-Aided Verification (CAV'99), Lecture Notes in Computer Science, vol. 1633, 1999, pp.160–171.
[25] N. De Francesco, A. Santone, G. Vaglini, State space reduction by non-standard semantics for deadlock analysis, Sci. Comput. Progr. 30 (3) (1998) 309–338.
[26] N. De Francesco, A. Santone, Syntactic reductions for efficient deadlock analysis, Softw. Test. Verif. Rel. 12 (3) (2002) 173–186.
[27] J. Dingel, T. Filkorn, Model checking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving, in: Proceedings of the 7th International Conference on Computer Aided Verification (CAV'95), Lecture Notes in Computer Science, vol. 939, 1995, pp. 54–69.
[28] S. Edelkamp, A. Lluch-Lafuente, S. Leue, Directed explicit model checking with HSF-SPIN, in: Proceedings of the 8th International SPIN Workshop on Model Checking Software, Lecture Notes in Computer Science, vol. 2057, 2001, pp. 57–79.
[29] S. Edelkamp, A. Lluch-Lafuente, S. Leue, Trail-directed model checking, Electronic Notes Theoret. Comput. Sci. 55 (2001).
[30] S. Edelkamp, F. Reffel, OBDDs in heuristic search, in: Proceedings of the 22nd Annual German Conference on Artificial Intelligence (KI'98), Lecture Notes in Computer Science, vol. 1504, 1998, pp. 81–92.
[31] P. Godefroid, Partial-order methods for the verification of concurrent systems, in: Lecture Notes in Computer Science, vol. 1032, 1996.
[32] P. Godefroid, S. Khurshid, Exploring very large state spaces using genetic algorithms, in: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02), Lecture Notes in Computer Science, vol. 2280, 2002, pp. 266–280.
[33] D. Gurov, B. Kapron, Modeling the application layer of an air traffic telecommunications network using CCS. <www.cs.uvic.ca/bmkapron/ccs.html>.
[34] P. Godefroid, P. Wolper, Using partial orders for efficient verification of deadlock freedom and safety properties, in: Proceedings of the Third International Conference on Computer-Aided Verification (CAV'91), Lecture Notes in Computer Science, vol. 575, 1991, pp. 332–342.
[35] J. Gosling, B. Joy, G. Steele, G. Bracha, The Java Language Specification, second ed., Addison-Wesley, Reading, MA, 2000, ISBN 0-201-31008-2.
[36] S. Gradara, A. Santone, M.L.Villani, Efficient Verification of Java Programs, Technical Report, TR-RCOST 12/03, April 2003.
[37] S. Graf, B. Steffen, G. Luttgen, Compositional minimization of finite state systems using interface specifications, Formal Aspects Comput. 8 (5) (1996) 607–616.

[38] A. Groce, W. Visser, Heuristic model checking for Java programs, in: Proceedings of the 9th International SPIN Workshop(SPIN'02), Lecture Notes in Computer Science, vol. 2318, 2002, pp. 242–245.
[39] O. Grumberg, D.E. Long, Model checking and modular verification, ACM Trans. Progr. Languages Syst. 16 (3) (1994) 843–871.
[40] P. Hart, D.J. Nilsson, B. Raphael, A formal basis fo heuristic determination of minimum path cost, IEEE Trans. Syst. Sci. Cybern. 100 (4) (1968).
[41] A. Holub, Taming Java Threads, APress, 2000.
[42] C. Jard, T. Jéron, Bounded-memory algorithms for verification on-the-fly, in: Proceedings of the Third International Conference on Computer-Aided Verification (CAV'91), Lecture Notes in Computer Science, vol. 575, 1991, pp. 192–201.
[43] R.M. Jensen, R.E. Bryant, M.M. Veloso, SetA*: An efficient BDD-based heuristic search algorithm, in: Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence (AAAI/IAAI'02), AAAI Press, New York, 2002, pp. 668–673.
[44] B. Karacali, K.C. Tai, Model checking based on simultaneous reachability analysis, in: Proceedings of the SPIN 2000 Conference on Model Checking of Computer Software, Lecture Notes in Computer Science, 2000, pp. 34–53.
[45] A. Lluch-Lafuente, S. Edelkamp, S. Leue, Partial order reduction in directed model checking, in: Proceedings of the 9th International SPIN Workshop on Model Checking Software, Lecture Notes in Computer Science, vol. 2318, 2002, pp. 112–127.
[46] E. Madelaine, D. Vergamini, Finiteness conditions and structural construction of automata for all process algebras, in: Proceedings of 2nd Workshop on Computer-Aided Verification, DIMACS Technical Report 90-31, 1990.
[47] K. McMillan, Symbolic Model Checking, Kluwer Academic Publishers, Boston, 1993.
[48] K. McMillan, Verification of infinite state systems by compositional model checking, in: Proceedings of the 10th Correct Hardware Design and Verification Methods International Conference on Concurrency Theory (CHARME'99), Lecture Notes in Computer Science, vol. 1703, 1999, pp. 219–234.
[49] R. Milner, Communication and Concurrency, Prentice-Hall, Englewood Cliffs, NJ, 1989.
[50] M.O. Moller, R. Alur, Heuristics for hierarchical partitioning with application to model checking, in: Proceedings of the 11th IFIP WG 10.5, Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'01), Lecture Notes in Computer Science, vol. 2144, 2001, pp. 71–85.
[51] J. Park, S. Song, TCP Model checking using Concurrency WorkBench, <www.dcs.warwick.ac.uk/doron/Report.doc>, 2002.
[52] J. Pearl, Heuristics: Intelligent Search Strategies for Computer Problem Solving, Addison-Wesley, Reading, MA, 1984.
[53] D. Peled, All from one, one for all, on model-checking using representatives, in: Proceedings of the Fifth International Conference on Computer-Aided Verification (CAV'93), Lecture Notes in Computer Science, vol. 679, 1993, pp. 409–423.
[54] W. Reisig, Petri nets, EATCS Monogr. Theoret. Comput. Sci. 4 (1985).
[55] P. Rondogiannis, M.H.M. Cheng, Petri-net-based deadlock analysis of process algebra programs, Sci. Comput. Progr. 23 (1994) 55–89.
[56] A. Santone, Automatic verification of concurrent systems using a formula-based compositional approach, Acta Inform. 38 (2) (2002) 531–564.
[57] A. Santone, Heuristic search + local model checking in selective mu-calculus, IEEE Trans. Software Eng. 29 (6) (2003) 510–523.
[58] J. Sifakis, Property preserving homomorphisms of transition systems, in: Logics of Programs, Lecture Notes in Computer Science, vol. 164, 1983.
[60] C. Stirling, D. Walker, Local model checking in the modal mu-calculus, Theoret. Comput. Sci. 89 (1991) 161–177.
[61] K.C. Tai, P.V. Koppol, Hierarchy-based incremental analysis of communication protocols, in: Proceedings of the International Conference on Network Protocols, 1993, pp. 318–325.
[62] R.N. Taylor, Complexity of analyzing the synchronization structure of concurrent programs, Acta Inform. 19 (1983) 57–84.

[63] A. Valmari, A stubborn attack on state explosion, in: Proceedings of the Second International Conference on Computer-Aided Verification (CAV'90), Lecture Notes in Computer Science, vol. 531, 1990, pp. 156–165.
[64] C.H. Yang, D.L. Dill, Validation with guided search of the state space, in: Proceedings of the 35th Conference on Design Automation (DAC'98), 1998, pp. 599–604.