Softw Syst Model
DOI 10.1007/s10270-014-0416-2

REGULAR PAPER

Heuristic search for equivalence checking

Nicoletta De Francesco · Giuseppe Lettieri · Antonella Santone · Gigliola Vaglini

Received: 1 July 2013 / Revised: 12 March 2014 / Accepted: 19 April 2014

Abstract Equivalence checking plays a crucial role in formal verification since it is a natural relation for expressing the matching of a system implementation against its specification. In this paper, we present an efficient procedure, based on heuristic search, for checking well-known bisimulation equivalences for concurrent systems specified through process algebras. The method tries to improve, with respect to other solutions, both the memory occupation and the time required for proving the equivalence of systems. A prototype has been developed to evaluate the approach on several examples of concurrent system specifications.

Keywords Heuristic search algorithms · Bisimulation · Concurrent systems · Model checking

1 Introduction

A suitable way for establishing desirable properties of a model is by showing that it is behaviorally related to another model that is known to have those properties; this kind of verification is often carried out by means of the analysis of the state space generated by the system. Depending on the type of relation between systems that is chosen, this verification technique is called *refinement* or *equivalence checking* [1]. *Model checking* is an alternative way (the technique originate in works by Clarke and Emerson [2] and Queille and Sifakis [3] from the early 1980s) to pursue the same result. Both these methods suffer from the so-called *state explosion problem*: the parallelism between the processes of the system leads to a number of reachable states which may become very large, in some cases in the order of millions or billions of states. To solve or reduce the state explosion problem several approaches can be followed: in the literature, there are reduction techniques based on process equivalences [4,5], symbolic model checking techniques [6], on-the-fly techniques [7], heuristic searches [8–10], local model checking approaches [11], partial order techniques [12–14], compositional techniques [15–17], and abstraction approaches [18–21]. More precisely, in [9,10], heuristic search has been used only to find deadlocks in current systems and in [8] to accelerate finding errors, while our goal is to use heuristic search to accelerate verification. In fact, in this paper, we propose to verify properties of concurrent systems without the breaking down of the automated verification caused by a too large memory occupation and with an efficiency in pursuing the result better than existing approaches. We propose to check equivalence of concurrent systems by applying *heuristic search* techniques on AND/OR graphs. A key assumption of heuristic search is that a *utility* or *cost* can be assigned to each state to guide the search by suggesting the next state to expand; in this way the most promising paths are considered first. Heuristic search [22] is one of the classical techniques in Artificial Intelligence and has been applied to a wide range of problem-solving tasks including puzzles, two player games

Communicated by Prof. Robert France.

N. De Francesco · G. Lettieri · G. Vaglini
Dipartimento di Ingegneria dell'Informazione, University of Pisa, Pisa, Italy
e-mail: n.defrancesco@unipi.it

G. Lettieri
e-mail: g.lettieri@iet.unipi.it

G. Vaglini
e-mail: g.vaglini@iet.unipi.it

A. Santone (✉)
Dipartimento di Ingegneria, University of Sannio, Benevento, Italy
e-mail: santone@unisannio.it

Published online: 21 May 2014

and path finding problems. The approach extends traditional on-the-fly techniques to efficiently explore the search space, since the heuristics overcomes the bottleneck of the exhaustive exploration of the global state graph of the two systems. There are several heuristic search algorithms for AND/OR graphs: a main difference among them is whether they tolerate cyclic AND/OR graphs or not. AO* [22,23] is the most widely known algorithm that requires the AND/OR graph to be acyclic, while S2 [24], which will be used here, is an algorithm designed to work also on cyclic AND/OR graphs. In any case, the algorithms expand the graph incrementally, starting from the initial node; an heuristic function assigns a cost to each node and is used to guide the expansion. The solution supplied by the algorithm is optimal only if the heuristic function is admissible, i.e., the function never overestimates the distance to the goal.

We consider concurrent systems specified by means of process algebras and formalize the equivalence checking of processes as a search problem on AND/OR graphs. Then, a procedure is presented that is based on S2 and uses admissible heuristic functions for weak and strong equivalences. The method is completely automated, i.e., there is no need for user intervention or manual effort. The goal is to check equivalence between processes, but also to find the minimal sub-graph leading to two non-equivalent states, since that graph will be examined in order to determine the source of the error and big graphs can prevent an easy comprehension of the fault.

To show that a significant space reduction without losing efficiency may be obtained with respect to other approaches, a prototype tool has been built implementing the presented method, and several experiments are carried on processes of different sizes. The heuristic functions are syntactically defined considering a specific process algebra: the Calculus of Communicating Systems (CCS) [25]. As far as we know, this is the first complete methodology to exploit process algebra-based heuristics in equivalence checking of concurrent systems. A very preliminary version can be found in [26], where the simulation relation defined by Milner [25] was considered; obviously simulation is a simpler relation than bisimulation; moreover, the algorithm used in [26] did not consider cyclic AND/OR graphs and no actual tool was built to perform some kind of evaluation of the proposed methodology, consequently the presented heuristic function was not tuned on the basis of the experimental results.

The paper is organized as follows: in Sect. 2, the basic concepts of behavioral equivalence and of the heuristic search algorithms are recalled. Section 3 describes our approach. In Sect. 4, the prototype tool implementing the approach is briefly presented, and the experimental results obtained are reported. Finally, comparisons with related works are discussed in Sect. 5.

2 Preliminaries

To develop the method in a language independent way, we assume a set of processes Δ, a set of actions Θ and a function σ that maps each $p \in \Delta$ to a finite set $\{(p_1, \alpha_1), \ldots, (p_n, \alpha_n)\} \subseteq \Delta \times \Theta$. If $(p', \alpha) \in \sigma(p)$, we say that p can perform the action α and reach the process p', and we write $p \xrightarrow{\alpha} p'$. If $p \xrightarrow{\alpha_1} p_1 \cdots \xrightarrow{\alpha_n} p_n$, the process p_i (for $i \in [1..n]$) is called *derivative* of p.

2.1 Behavioral equivalence

Process algebras can be used to describe both implementations of processes and specifications of their expected behaviors. Therefore, they support the so-called single language approach to process theory, that is, the approach in which a single language is used to describe both actual processes and their specifications. An important ingredient of these languages is therefore a notion of behavioral equivalence. One process description, say SYS, may describe an implementation, and another, say SPEC, may describe a specification of the expected behavior. This approach to program verification is also sometimes called *implementation verification.* Behavioral equivalence can be used also for other purposes, like for example for clone detection [27,28].

In the following, we introduce well-known notions of behavioral equivalence which describe how processes (i.e. systems) match each other's behavior. Milner introduces strong and weak equivalences. Strong equivalence is a kind of invariant relation between process that is preserved by actions as stated by the following definition.

Definition 2.1 Let p and q be two processes.

- A strong bisimulation, $\mathcal{B}$, is a binary relation on Δ such that $p\ \mathcal{B}\ q$ implies:
  - (i) $p \xrightarrow{\alpha} p'$ implies $\exists q'$ such that $q \xrightarrow{\alpha} q'$ with $p'\ \mathcal{B}\ q'$; and
  - (ii) $q \xrightarrow{\alpha} q'$ implies $\exists p'$ such that $p \xrightarrow{\alpha} p'$ with $p'\ \mathcal{B}\ q'$
- p and q are strongly equivalent ($p \sim q$) iff there exists a strong bisimulation $\mathcal{B}$ containing the pair (p, q).

The idea underlying the definition of the weak equivalence is that an action of a process can now be matched by a sequence of action from the other that has the same "observational content" (i.e. ignoring internal actions) and leads to a state that is equivalent to that reached by the first process. In order to define the weak equivalence, we assume there exists a special action $\tau \in \Theta$, which we interpret as a silent, internal action, and we introduce the following transition relation that ignores it.

Let p and q be processes in Δ. We write $p \stackrel{\epsilon}{\Longrightarrow} q$ if and only if there is a (possibly empty) sequence of τ actions that leads from p to q. (If the sequence is empty, then $p = q$.) For each action α, we write $p \stackrel{\alpha}{\Longrightarrow} q$ iff there are processes p' and q' such that

$$p \stackrel{\epsilon}{\Longrightarrow} p' \stackrel{\alpha}{\Longrightarrow} q' \stackrel{\epsilon}{\Longrightarrow} q.$$

Thus, $p \stackrel{\alpha}{\Longrightarrow} q$ holds if p can reach q by performing an α action, possibly preceded and followed by sequences of τ actions. For each action α, we use $\widehat{\alpha}$ to stand for ϵ if $\alpha = \tau$, and for α otherwise.

Definition 2.2 Let p and q be two processes.

- A weak bisimulation, $\mathcal{B}$, is a binary relation on Δ such that $p \mathcal{B} q$ implies:
  - (i) $p \stackrel{\widehat{\alpha}}{\Longrightarrow} r'$ implies $\exists q'$ such that $q \stackrel{\widehat{\alpha}}{\Longrightarrow} q'$ with $p' \mathcal{B} q'$; and
  - (ii) $q \stackrel{\widehat{\alpha}}{\Longrightarrow} q'$ implies $\exists p'$ such that $p \stackrel{\widehat{\alpha}}{\Longrightarrow} p'$ with $p' \mathcal{B} q'$
- p and q are weak equivalent ($p \approx q$) iff there exists a weak bisimulation $\mathcal{B}$ containing the pair (p, q).

2.2 Heuristic search: AND/OR graphs and algorithm S2

In this section, we briefly review AND/OR graphs and the heuristic search algorithm S2 [24] for solving problems formalized as AND/OR graphs.

First, we establish some terminology for graphs in general. A (directed) graph G is a pair (N, A) where N is a set of nodes and $A \subseteq N \times N$ is the set of arcs. If $n \in N$ is a node of G, then

$$s_G(n) = \{\, m \in N \mid (n, m) \in A \,\}$$

is the set of successors of n in G. A graph $G' = (N', A')$ is a *subgraph* of $G = (N, A)$ if $N' \subseteq N$ and $A' \subseteq N' \times N'$ with $A' \subseteq A$. A graph $G = (N, A)$ is finite if both N and A are finite sets. Let $G = (N, A)$ be a graph and $s, d \in N$ be two nodes (not necessarily distinct). A path from s to d is a sequence of nodes $n_1, \ldots, n_k$ with $k > 1$ such that $s = n_1$, $d = n_k$ and $(n_i, n_{i+1}) \in A$ for each $1 \leq i < k$. A path from a node to the same node is a *cycle*. A graph is acyclic if it contains no cycles, and cyclic otherwise.

A problem can be formalized in terms of a graph, for example as follows. A common problem-solving strategy consists of decomposing a problem P into subproblems, so that either all or just one of these subproblems need to be solved in order to obtain a solution for P. Each problem can be represented as a node in a directed graph, where arcs express the decomposition relationship between problems and subproblems. The two kinds of decomposition give rise to two kinds of nodes: AND nodes and OR nodes.

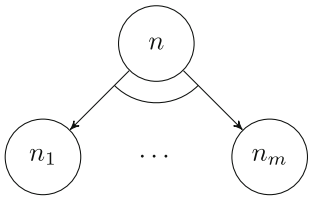

Fig. 1 An AND node

An AND/OR graph G is a directed graph with a special node s, called the *start* (or *root*) *node*, and a (possibly empty) set of *terminal leaf nodes* denoted as $t, t_1, \ldots$. The start node s represents the given problem to be solved, while the terminal leaf nodes correspond to subproblems with known solutions. The nonterminal nodes of G are of three types: OR, AND, and *nonterminal leaf*. An OR node is solved if one of its immediate subproblems is solved, while an AND node is solved only when every one of its immediate subproblems is solved. A nonterminal leaf node has no successors and is unsolvable. AND nodes with at least two successors are recognized from OR nodes by connecting the subproblems arcs by a line, like in Fig. 1.

Given an AND/OR graph G, a solution of G is represented by an AND/OR subgraph, called *solution (sub)graph of G* with the characteristics given below.

Definition 2.3 A finite subgraph D of an AND/OR graph G is a solution subgraph of G if it is acyclic and:

(i) the start node of G is in D;
(ii) if n is an OR node in G and n is in D, then exactly one of the immediate successors of n in G is in D;
(iii) if n is an AND node in G and n is in D, then all the immediate successors of n in G are in D;
(iv) every maximal path in D ends in a terminal leaf node.

Each solution subgraph can obtain a cost through a cost function that assigns a cost to each arc: each directed arc (m, n) in the graph has a discrete cost $c(m, n) > 0$.

Definition 2.4 Let D be a solution graph and n a node in D, the cost of n in D, denoted $h(n)$ is defined as follows:

(i) $h(n) = 0$ if n is a terminal leaf node;
(ii) $h(n) = \infty$ if n is a nonterminal leaf node;
(iii) $h(n) = c(n, p) + h(p)$ if n is an OR node and p is its immediate successor;
(iv) $h(n) = \sum_{i=1}^{k} [c(n, p_i) + h(p_i)]$ if n is an AND node with immediate successors $p_1, p_2, \ldots, p_k$.

A solution graph of an AND/OR graph G is a *minimal-cost solution graph* if the cost of its root is the minimum over the cost of the roots of all the solution graphs of G. The goal of any search algorithm for AND/OR graphs is to find a minimal-cost solution graph.

Since in most domains, the AND/OR graph G is unknown in advance, it is not supplied explicitly to a search algorithm. We refer to G as the *implicit graph*; it is specified implicitly by a start node s and a successor function. The search algorithm works on an *explicit graph* G', which initially consists of the start node s. The start node is then expanded, that is, all the immediate successors of s are added to the explicit graph G'. At any moment, the explicit graph has a number of *tip nodes*, which are nodes with no successors in the explicit graph, and the search algorithm chooses one of these tip nodes for expansion. In this way, more and more nodes and arcs get added to the explicit graph, until finally it has one or more solution graphs as subgraphs. One of these solution graphs is then output by the search algorithm.

Heuristic search algorithms use an heuristic estimate function $\widehat{h}$, which can be viewed as an estimate of h, to direct the search and to restrict the number of nodes expanded within acceptable limits. Thus, the heuristic search can find an optimal solution graph without evaluating the entire state space. There are several heuristic search algorithms for AND/OR graphs. The algorithms differ on the kind of AND/OR graphs they accept as input and the solution subgraphs they produce as output. The classic AO* algorithm [22,23] only works with AND/OR graphs, both explicit and implicit, that do not contain cycles; for cyclic AND/OR graphs we have algorithm REV* [29], which only works on explicit graphs, and algorithms $\text{CFC}_{\text{REV}*}$ [30] and S2 [24] that accept implicit AND/OR graphs. All these algorithms only search for solution subgraphs that do not themselves contain cycles, and thus adhere to Definition 2.3. Algorithm LAO* [31] removes this limitation in the context of Markov decision problems.

In this paper, we use the algorithm S2 reproduced in Fig. 2. The algorithm mainly consists of two iterated steps: (i) expand the most promising tip node of the explicit graph (step **S2.2.1**); (ii) update of the computed node costs (step **S2.2.3**). The algorithm uses a map *front* to select the node to expand in step (i). This map is updated during step (ii), together with the node costs. Step (ii) uses the procedure Bottom_Up defined in Fig. 3. The algorithm terminates with success if an acyclic solution exists, otherwise it terminates with failure. When the acyclic solution exists, it also returns the cost of the solution.

An important property holds: S2 returns a minimal-cost solution graph if the heuristic estimate function $\widehat{h}$ (used in step **S2.2.1** of Fig. 2) satisfies the so-called *admissibility* condition, i.e. $\widehat{h}$ is optimistic. More formally:

Definition 2.5 (*admissibility*) Let G be an AND/OR graph and D a minimal solution subgraph of G. A heuristic estimate function $\widehat{h}$ defined on the nodes of G is admissible if for each node n in G

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

where $h(n)$ is the cost of n in D.

3 The method

In this section, we explain the basis of our approach to strong equivalence checking. In Subsect. 3.1, *AND/OR structures* are defined as a slight modification of the concept of AND/OR graph more suitable to the problem of equivalence checking. Given two processes p and q, we build an AND/OR structure that has a solution if and only if p and q are bisimilar (Theorem 3.2). In Subsect. 3.2, we show how algorithms for heuristic search on AND/OR graphs can be used to find solutions of AND/OR structures. This allows for the method to be implemented in practice. In Subsect. 3.3, we apply the method to the CCS language. In Subsect. 3.4, the heuristic function to be used in the search for strong equivalence of CCS processes is described and its admissibility is proved, while in the Subsect. 3.5, the heuristic function for weak equivalence is defined.

Fig. 2 The S2 algorithm

S2.1 Create an explicit graph G' consisting solely of the start node s. Set $front(s) = s$. If s is a terminal leaf set $h(s) = 0$; else if s is a nonterminal leaf set $h(s) = \infty$.

S2.2 While ($front(s)$ is not a terminal leaf and $h(s) \neq \infty$) do:

- **S2.2.1** Let $n = front(s)$. Expand n and add all its children $n_1, \ldots, n_k$ to G'. For each newly occurring node n_i in G' set $front(n_i) = n_i$. If n_i is a terminal leaf set $h(n_i) = 0$; else if n_i is nonterminal leaf set $h(n_i) = \infty$; else set $h(n_i) = \widehat{h}(n_i)$.
- **S2.2.2** Create a list OPEN containing all the tip nodes of G'. Label all the nodes in OPEN as eligible and initial.
- **S2.2.3** Call Bottom_Up(OPEN).

S2.3 If $front(s)$ is a terminal leaf node, output $h(s)$ and terminate with SUCCESS; else terminate with FAILURE.

Fig. 3 The Bottom_Up(List OPEN) procedure

B1 Initialize a list, CLOSED, to nil.

B2 While (OPEN contains an eligible node and $s \notin$ CLOSED) do:

B2.1 Select an eligible node q from OPEN that has minimum h-value.

B2.2 If q is not an initial node, then do the following: Let $q_1, q_2, \ldots, q_r$ be the children of q in G'' which are in CLOSED. If q is an OR node let $j = \text{argmin}_{1 \le i \le r}\{c(q, q_i) + h(q_i)\}$ and set $front(q) = front(q_j)$. If q is an AND node let q_j be the leftmost child of q whose front is not a terminal leaf and set $front(q) = front(q_j)$. (Use q_1 if no such q_j exists).

B2.3 Put q in CLOSED. Let $p_1, \ldots, p_k$ be the parents of q in G'. For each p_i do:

* (If p_i is an OR node) if $p_i \notin$ OPEN$\cup$CLOSED set $h(p_i) = h(q) + c(p_i, q)$, put p_i in OPEN and mark it eligible; else if p_i is already in OPEN with $h(p_i) > h(q) + c(p_i, q)$, set $h(p_i) = h(q) + c(p_i, q)$.
* (If p_i is an AND node) if $p_i \notin$ OPEN, put it in OPEN and set $h(p_i) = h(q) + c(p_i, q)$; else set $h(p_i) = h(p_i) + c(p_i, q) + h(q)$. If all children of p_i are in CLOSED, mark p_i as eligible.

B3 Remove any remaining nodes from OPEN.

B4 If $s \notin$ CLOSED set $h(s) = \infty$.

3.1 AND/OR structures

AND/OR structures are related to AND/OR graphs, but differ slightly in the way the terminal nodes and the solution subgraphs are defined. These differences allow AND/OR structures to have *duals* which are again AND/OR structures. More importantly, the existence of solutions for an AND/OR structure can be related to the existence of solutions for its dual (Theorem 3.1).

Definition 3.1 (*AND/OR structure*) An AND/OR structure is a triple $\langle G, s, t\rangle$, where $G = (N, R)$ is a directed graph, $s \in N$ is the start node, and $t : N \to \{AND, OR\}$.

Nodes in $t^{-1}(\{AND\})$ are called AND nodes and nodes in $t^{-1}(\{OR\})$ are called OR nodes. Terminal and non terminal leaves are defined as special cases of AND and OR leaves (i.e., nodes with no successors): AND leaves are terminal, while OR leaves are non terminal.

The AND/OR structure $\langle G, s, t\rangle$ is finite/cyclic/acyclic if G is respectively finite/cyclic/acyclic.

Definition 3.2 (*Solution*) Let $T = \langle G, s, t\rangle$ be an AND/OR structure. A subgraph $D = (N', R')$ of G is a *solution* of T if:

(i) $s \in N'$;
(ii) if $n \in N'$ and $t(n) = AND$, then $s_D(n) = s_G(n)$;
(iii) if $n \in N'$ and $t(n) = OR$ then $s_D(n) \neq \varnothing$.

In point (iii) one successor *at least* is required for OR nodes, in contrast with the *exactly one* successor required in Definition 2.3. Moreover, the solution is not required to be acyclic and the maximal paths are not required to end in terminal leaves. Note that OR nodes which have no successors in G cannot belong to any solution, hence their labeling as non terminal leaves.

We now define the notion of dual of an AND/OR structure that is obtained by switching the type of all the nodes of the original structure.

Definition 3.3 (*Dual*) If $T = \langle (N, R), s, t\rangle$ is an AND/OR structure, its *dual*, denoted T^∂, is the AND/OR structure $\langle (N, R), s, t^\partial\rangle$, where $t^\partial : N \to \{AND, OR\}$ is defined as

$$t^\partial(n) = \begin{cases} AND & \text{if } t(n) = OR, \\ OR & \text{if } t(n) = AND. \end{cases}$$

Note that the terminal leaves of the original structure became non terminal leaves of the dual structure, while non terminal leaves of the original become terminal in the dual.

The utility of the notion of dual comes from the following Theorem, which permits to look for acyclic solutions even if the original problem may admit cyclic solutions.

Theorem 3.1 *A finite AND/OR structure has no solutions iff its dual has an acyclic solution.*

Proof Let $T = \langle G, s, t\rangle$ be a finite AND/OR structure and let T^∂ be the dual of T. We prove the $\Leftarrow$ direction first. The proof is by contradiction, so let $C = (N', R')$ be an

acyclic solution of T^∂ and assume that there exists a solution $D = (N'', R'')$ of T. Let $n_1, n_2, \ldots$ be a topological sorting of the nodes of C with $n_1 = s$ and such that if $(n_i, n_j) \in R'$ then $i < j$. Note that $n_1 = s \in N''$. We claim that if a non-leaf n_i is in N'', then there is a $j > i$ such that also n_j is in N''. Indeed, if $t(n_i) = AND$ then n_i is an OR node in T^∂, thus C must contain a successor node, which must be n_j for some $j > i$. Then surely $n_j \in N''$, since D must contain all successors of n_i. If, instead, $t(n_i) = OR$, then D must contain a successor node m. Since n_i is an AND node in T^∂, m must appear as n_j for some $j > i$. If we iterate this reasoning, we must arrive at a terminal leaf of T^∂ which is in D, but this is a contradiction, since terminal leaves of T^∂ are non terminal leaves of T and cannot belong to a solution.

Now let us prove the $\Longrightarrow$ direction. We know that G has no solutions. This implies that any subgraph of G that contains s will either contain an AND node without containing at least one of its successors, or it will contain an OR node, but none of its successors. We use this property to build an acyclic solution for T^∂ in the following way: we build a sequence $D_1, \ldots, D_k$ of subgraphs of T and a parallel sequence $C_1, \ldots, C_k$ of subgraphs of T^∂ such that: (i) for each $1 \leq i \leq k$, subgraph C_i is acyclic and the set of nodes of C_i and D_i form a partition of the nodes of G; (ii) s is contained in D_i for all $1 \leq i < k$; (iii) s is not contained in D_k. Then, C_k will be the required solution. To build the two sequences, start with $D_1 = G$. Then, D_1 must contain at least a non terminal leaf, be it n_1. Let $C_1 = (\{n_1\}, \varnothing)$ and build $D_2 = D_1 - n_1$, i.e., D_1 after the removal of n_1 and all arcs incident on n_1. If $n_1 = s$ we are done, otherwise we can continue. Assume we have already built sequences $D_1, \ldots, D_i$ and $C_1, \ldots, C_i$ satisfying properties (i) and (ii) above. If D_i does not contain s we are done, otherwise D_i must contain an AND node with at least one successor in C_i, or an OR node with all of its successors in C_i. Let n_{i+1} be any such node. Let $D_{i+1} = D_i - n_{i+1}$ and build C_{i+1} from C_i by adding node n_{i+1} and all the arcs from n_{i+1} to nodes already in C_i. Since C_i was assumed to by acyclic, so is C_{i+1}. We can iterate the process and, since G is finite, property (iii) will eventually hold. □

Let p and q be two processes in Δ: an AND/OR structure can be built that has a solution iff p and q are bisimilar. The idea is to check the requirements of Definitions 2.1 and 2.2 by letting p and q move in alternating turns.

Let $T(p, q) = \langle G, s, t\rangle$ with $G = (N, R)$. The nodes contained in N are tuples of 4 elements $\langle r, s, \gamma, u\rangle$ where

- r is a derivative of p;
- s is a derivative of q;
- $\gamma \in \{\top, \bot\} \cup Act$;
- $u \in \{1, 2, \lambda\}$.

We assume that $\{\top, \bot\} \cap Act = \varnothing$ and that $u = \lambda$ iff $\gamma = \top$. When $\gamma = \top$ it is the turn of both r and s to move; when $\gamma = \bot$ then r has to move if $u = 1$, while s has to move if $u = 2$; finally, when $\gamma = \alpha \in Act$ then r has to move if $u = 1$ and s has to move if $u = 2$, but, in both cases, α has to be performed (the idea is that α is the action that the other process has performed in the previous turn).

The map t of $T(p, q)$ only depends on γ and is defined as

$$t(\langle r, s, \gamma, u\rangle) = \begin{cases} AND & \text{if } \gamma = \top \text{ or } \gamma = \bot; \\ OR & \text{if } \gamma \in Act. \end{cases}$$

The graph G of $T(p, q)$ is obtained by repeated application of the operators given in Table 1, starting with a graph containing the node $\langle p, q, \top, \lambda\rangle$ and no arcs. The operators generate the outgoing arcs and the successor nodes of each node; if $(n, n') \in R$, we write $n \longrightarrow n'$. The operators with the form

$$\frac{\textit{premise}}{n \longrightarrow n_1 \text{ and } \cdots \text{ and } n \longrightarrow n_m}$$

where *premise* is the antecedent, possibly empty, of the rule, generate all the outgoing arcs and successor nodes of the AND node n. On the other hand, the operators with the form:

$$\frac{\textit{premise}}{n \longrightarrow n_1 \text{ or } \cdots \text{ or } n \longrightarrow n_m}$$

generate all the outgoing arcs and successor nodes of the OR node n. Finally, the start node s of $T(p, q)$ is $\langle p, q, \top, \lambda\rangle$.

The rule $\mathbf{op}_1$ transforms the initial node, which is an AND node, into the two successors nodes with $\gamma = \bot$ and $u = 1$ and $u = 2$, respectively. The rule $\mathbf{op}_2$ points out the possible moves (α_i) of p when $u = 1$ and $\gamma = \bot$; in this way an AND node can be connected with its successor nodes in the graph, all such successors have $\gamma = \alpha_i$ and $u = 2$; roughly speaking if p can move performing an action α_i and reaches the process p_i then it is the turn of q to move with the same action α_i. The rule $\mathbf{op}_2'$ is similar to $\mathbf{op}_2$ applied when it is the turn of q to move. In rule $\mathbf{op}_3$, the process p must simulate the action α performed by q, while in rule $\mathbf{op}_3'$, it is the process q that must simulate the action α performed by p. In both cases, an OR node can be connected with its successor nodes in the graph, all such successors have $\gamma = \top$ and $u = \lambda$, i.e. nodes that can be transformed only through the operator $\mathbf{op}_1$ like the initial node.

The following theorem shows that finding a solution of $T(p, q)$ is equivalent to checking whether p and q are strongly bisimilar.

Theorem 3.2 *Let p and q be two processes in Δ and consider the AND/OR structure $T(p, q)$ generated starting from $\langle p, q, \top, \lambda\rangle$ using the operators of Table 1. Then $p \sim q$ iff $T(p, q)$ has a solution.*

Table 1 The operators

$$\mathbf{op}_1 \quad \frac{}{\langle p,q,\top,\lambda\rangle \longrightarrow \langle p,q,\bot,1\rangle \text{ and } \langle p,q,\top,\lambda\rangle \longrightarrow \langle p,q,\bot,2\rangle}$$

$$\mathbf{op}_2 \quad \frac{\sigma(p)=\{(p_1,\alpha_1),\ldots,(p_n,\alpha_n)\}\neq\varnothing}{\langle p,q,\bot,1\rangle \longrightarrow \langle p_1,q,\alpha_1,2\rangle \text{ and } \cdots \text{ and } \langle p,q,\bot,1\rangle \longrightarrow \langle p_n,q,\alpha_n,2\rangle}$$

$$\mathbf{op}'_2 \quad \frac{\sigma(q)=\{(q_1,\alpha_1),\ldots,(q_n,\alpha_n)\}\neq\varnothing}{\langle p,q,\bot,2\rangle \longrightarrow \langle p,q_1,\alpha_1,1\rangle \text{ and } \cdots \text{ and } \langle p,q,\bot,2\rangle \longrightarrow \langle p,q_n,\alpha_n,1\rangle}$$

$$\mathbf{op}_3 \quad \frac{\sigma(p)=\{(p_1,\alpha_1),\ldots,(p_n,\alpha_n)\}\neq\varnothing,\ \alpha_i=\alpha\ \forall i\in[1..n]}{\langle p,q,\alpha,1\rangle \longrightarrow \langle p_1,q,\top,\lambda\rangle \text{ or } \cdots \text{ or } \langle p,q,\alpha,1\rangle \longrightarrow \langle p_n,q,\top,\lambda\rangle}$$

$$\mathbf{op}'_3 \quad \frac{\sigma(q)=\{(q_1,\alpha_1),\ldots,(q_n,\alpha_n)\}\neq\varnothing,\ \alpha_i=\alpha\ \forall i\in[1..n]}{\langle p,q,\alpha,2\rangle \longrightarrow \langle p,q_1\top,\lambda\rangle \text{ or } \cdots \text{ or } \langle p,q,\alpha,2\rangle \longrightarrow \langle p,q_n,\top,\lambda\rangle}$$

Proof Let us prove the $\Leftarrow$ direction first. Take a solution $D=(N,R)$ of $T(p,q)$ and consider the relation

$$S=\{\,(p',q') \mid \langle p',q',\top,\lambda\rangle \in N\,\}. \qquad (*)$$

Clearly $(p,q)\in S$. We claim that S is a strong bisimulation, thus proving $p\sim q$. Indeed, take any $(p',q')\in S$. Thus, the AND node $\langle p',q',\top,\lambda\rangle$ is in N and, since D is a solution, both its successors $\langle p',q',\bot,1\rangle$ and $\langle q',p',\bot,2\rangle$ (as given by operator $\mathbf{op}_1$ in Table 1) are also in N (according to Definition 3.2). If $p'\xrightarrow{\alpha}p''$, then node $\langle p',q',\bot,1\rangle$ will produce processes in Δ node $\langle p'',q',\alpha,2\rangle$ through operator $\mathbf{op}_2$. Since $\langle p',q',\bot,1\rangle$ is and AND node in D and D is a solution, then node $\langle p'',q',\alpha,2\rangle$ will also be in D. Now, node $\langle p'',q',\alpha,2\rangle$ is an OR node, thus D must contain at least on successor for it (Definition 3.2(iii)). Since successors of node $\langle p'',q',\alpha,2\rangle$ are produced by operator $\mathbf{op}'_3$, this means that $q'\xrightarrow{\alpha}q''$ must hold for some q''. Moreover, the successor of node $\langle p'',q',\alpha,2\rangle$ will be node $\langle p'',q'',\top,\lambda\rangle$. Since this latter node must be in D, then (p'',q'') must be in S according to $(*)$. This proves point (i) of Definition 2.1. Point (ii) is proved symmetrically.

Now let us prove the $\Longrightarrow$ direction. Assume we are given a bisimulation $\mathcal{B}$ such that $(p,q)\in\mathcal{B}$. We first use $\mathcal{B}$ to build and AND/OR structure $D(\mathcal{B})$. We build $D(\mathcal{B})$ in two steps:

A. For each $(p',q')\in\mathcal{B}$ we add to $D(p,q)$ the AND nodes $\langle p',q',\top,\lambda\rangle$, $\langle p',q',\bot,1\rangle$ and $\langle p',q',\bot,2\rangle$, with the appropriate arcs.
B. for each $p'\xrightarrow{\alpha}p''$ such that $(p',q')\in\mathcal{B}$ for some q', we use point (ii) of Definition 2.1 to find q'' such that $q'\xrightarrow{\alpha}q''$ and $(p'',q'')\in\mathcal{B}$. Then, we add to $D(p,q)$ the OR node $\langle p'',q',\alpha,2\rangle$, with an arc coming from node $\langle p',q',\bot,1\rangle$ and an arc going to node $\langle p'',q'',\top,\lambda\rangle$ (these latter two nodes were added in step A). We operate analogously for each $q'\xrightarrow{\alpha}q''$ such that $(p',q')\in\mathcal{B}$ for some p'.

It is now easy to show that $D(\mathcal{B})$ is a solution of $T(p,q)$. □

3.2 Heuristic search for strong equivalence

When trying to use the S2 algorithm to prove strong equivalence we face two problems: i) the S2 algorithm works on AND/OR graphs, rather then AND/OR structures; ii) the solutions found by the S2 algorithm follow Definition 2.3, while we are interested in solutions that follow Definition 3.2.

Problem (i), however, is only apparent. Indeed, any AND/OR structure $\langle G,s,t\rangle$ can also be interpreted as an AND/OR graph G with the same root node s. The AND nodes of the AND/OR graph are the AND nodes of the AND/OR structure with at least one successor, while the terminal leaves of the AND/OR graph are the AND nodes of the AND/OR structure with no successors, and similarly for OR nodes and nonterminal leaves.

For problem (ii), we first note that the main difference between the two kinds of solutions is that Definition 2.3 requires acyclicity, while Definition 3.2 does not. Indeed, it is easy to find bisimilar processes p and q such that $T(p,q)$ only contains cyclic solutions: consider for example two processes p and q such that $p\xrightarrow{a}p$ and $q\xrightarrow{a}q$.

To cope with this problem we use Theorem 3.1. More precisely, given two processes p and q, an heuristic search on $T^{\partial}(p,q)$, the dual of $T(p,q)$, can be performed. If the search terminates with failure, then p and q are bisimilar. If, instead, the search terminates with success, then p and q are not bisimilar. Essentially, we try prove/disprove bisimilarity by looking for counterexamples, which are always acyclic.

The other differences between Definition 2.3 and Definition 3.2 are taken into account by the following Theorem.

Theorem 3.3 *Let $T=\langle G,s,t\rangle$ be a finite AND/OR strucuture also interpreted as an AND/OR graph. Then, T has an acyclic solution according to Definition 2.3 iff it also has a solution according to Definition 3.2.*

Proof One direction is trivial, since Definition 2.3 is more restrictive than Definition 3.2, and therefore, any solution in the AND/OR graph sense is also a solution in the AND/OR structure sense. In the other direction, assume D is an acyclic solution of T according to Definition 3.2. Since T is finite and D is acyclic, the maximal paths in D must end in a node with no successors. This must be an AND node, since D is a solution. But AND nodes with no successors in an AND/OR structure are terminal nodes in the corresponding AND/OR graph, so point (iv) of Definition 2.3 is satisfied. Now, D may fail to be a solution in the AND/OR graph sense only if some of the OR nodes it contains have more than one successor in D. However, we can build a subgraph D' of D in which we keep all nodes, all outgoing arcs of the AND nodes, and exactly one outgoing arc for each OR node. Then, D' is a solution according to Definition 2.3. □

Therefore, S2 applied to $T^{\partial}(p,q)$ finds a solution iff $T^{\partial}(p,q)$ has a solution iff p and q are not strongly bisimilar. Note that the proof also shows that any solution found by $S2$ can be readily interpreted as a solution of $T^{\partial}(p,q)$, i.e., as a counterexample of $T(p,q)$. Moreover, the heuristic search founds a counterexample of minimal cost. The cost of each arc is 1, so that the cost of the counterexample is related to the number of actions performed by the two processes.

3.3 Application of the method to CCS processes

In this section, we apply our method to the CCS language specification. Thus, we briefly recall the Calculus of Communicating Systems (CCS) [25], which is an algebra suitable for modeling and analyzing processes. The reader can refer to [25] for further details. The syntax of *processes* is the following:

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

where α ranges over a finite set of actions $Act = \{\tau, a, \overline{a}, b, \overline{b}, ...\}$. Input actions are labeled with "non-barred" names, e.g. a, while output actions are "barred", e.g. $\overline{a}$. The action $\tau \in Act$ is called *internal action*. The set L, in processes with the form $p\backslash L$, ranges over sets of *visible actions* ($\mathcal{V} = Act - \{\tau\}$), f ranges over functions from actions to actions, while x ranges over a set of *constant* names: each constant x is defined by a constant definition $x \stackrel{\text{def}}{=} p$. Given $L \subseteq \mathcal{V}$, with L°, we denote the set $\{\bar{l}, l \mid l \in L\}$. We call $\mathcal{P}$ the processes generated by p.

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, we consider only guarded processes.

Table 2 Operational semantics of CCS

Act	$\dfrac{}{\alpha.p \xrightarrow{\alpha} p}$	**Sum**	$\dfrac{p \xrightarrow{\alpha} p'}{p+q \xrightarrow{\alpha} p'}$
Con	$\dfrac{p \xrightarrow{\alpha} p'}{x \xrightarrow{\alpha} p'}\ x \stackrel{\text{def}}{=} p$	**Par**	$\dfrac{p \xrightarrow{\alpha} p'}{p \mid q \xrightarrow{\alpha} p' \mid q}$
Com	$\dfrac{p \xrightarrow{l} p',\ q \xrightarrow{\bar{l}} q'}{p \mid q \xrightarrow{\tau} p' \mid q'}$	**Rel**	$\dfrac{p \xrightarrow{\alpha} p'}{p[f] \xrightarrow{f(\alpha)} p'[f]}$
Res	$\dfrac{p \xrightarrow{\alpha} p'}{p\backslash L \xrightarrow{\alpha} p'\backslash L}\ \alpha \notin L^{\circ}$		

The standard *operational semantics* [25] is given by a relation $\longrightarrow \subseteq \mathcal{P} \times Act \times \mathcal{P}$, which is the least relation defined by the rules in Table 2 (we omit the symmetric rule of **Sum** and **Par**).

A *(labeled) transition system* is a quadruple $(\mathcal{S}, Act, \longrightarrow, p)$, where $\mathcal{S}$ is a set of states, Act is a set of transition labels (actions), $p \in \mathcal{S}$ is the initial state, and $\longrightarrow \subseteq \mathcal{S} \times Act \times \mathcal{S}$ is the transition relation. If $(p, \alpha, q) \in \longrightarrow$, we write $p \xrightarrow{\alpha} q$.

If $\delta \in Act^*$ and $\delta = \alpha_1 \ldots \alpha_n$, $n \geq 1$, we write $p \xrightarrow{\delta} q$ to mean $p \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} q$. Moreover, $p \xrightarrow{\lambda} p$, where λ is the empty sequence. Given $p \in \mathcal{S}$, with $\mathcal{R}(p) = \{q \mid p \xrightarrow{\delta} q\}$ we denote the set of the states reachable from p by $\longrightarrow$, also called *derivatives* of p. When p has a finite number of syntactically different derivatives, p is called finite state, or simply finite.

Given a CCS process p, the *standard transition system* for p is defined as $\mathcal{S}(p) = (\mathcal{R}(p), Act, \longrightarrow, p)$. Note that, with abuse of notation, we use $\longrightarrow$ for denoting both the operational semantics and the transition relation among the states of the transition system.

In this paper, we use CCS without the relabeling operator. Note that this is not a restriction since the calculus is still Turing equivalent.

To build the AND/OR structure for CCS processes, we take Δ as the set of CCS processes $\mathcal{P}$, and the σ function described in the previous sub-section, is rephrased using the standard operational semantic, i.e. $\sigma(p) = \{(p', \alpha) \mid p \xrightarrow{\alpha} p'\} = \{(p_1, \alpha_1), \ldots, (p_n, \alpha_n)\}$.

Example 3.1 Consider the following CCS processes:

$$p \stackrel{\text{def}}{=} b.nil + c.k.d.f.nil$$

$$q \stackrel{\text{def}}{=} b.c.nil + c.k.d.nil$$

The AND/OR structure $T(p,q)$ generated starting from $\langle p, q, \top, \lambda \rangle$, using the operators of Table 1, is sketched in Fig. 4. For simplicity, only the first four levels have been shown in detail.

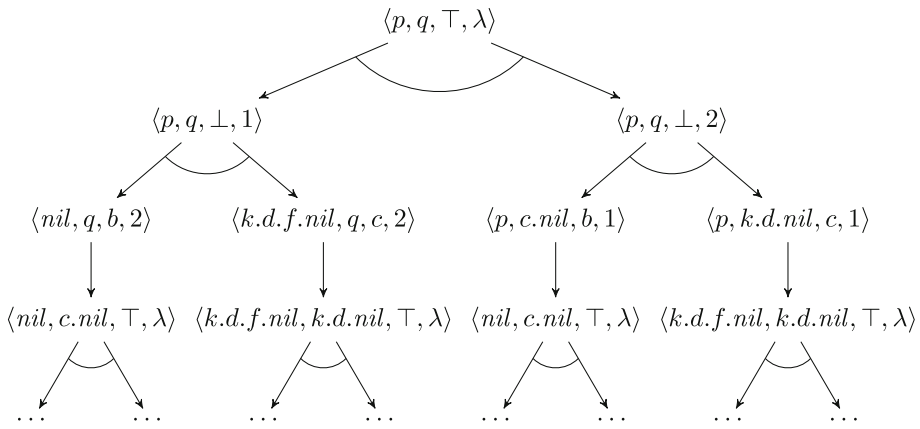

Fig. 4 The AND/OR structure $T(p, q)$

Table 3 Auxiliary functions used by the heuristics

$$\widehat{h}_\top(p,q,i) = \begin{cases} 0 & \text{if } stop(p,q,i) \\ 1 + \min(\widehat{h}_\bot(p,q,i+1), \widehat{h}_\bot(q,p,i+1)) & \text{otherwise} \end{cases}$$

$$\widehat{h}_\bot(p,q,i) = \begin{cases} 0 & \text{if } stop(p,q,i) \\ \min\{\, 1 + \widehat{h}_\alpha(p',q,i+1) \mid p \xrightarrow{\alpha} p' \,\} & \text{otherwise} \end{cases}$$

$$\widehat{h}_\alpha(p,q,i) = \begin{cases} 0 & \text{if } stop(p,q,i) \\ \sum\{\, 1 + \widehat{h}_\top(p,q',i+1) \mid q \xrightarrow{\alpha} q' \,\} & \text{otherwise} \end{cases}$$

where

$stop(p, q, i) \equiv (i > M)$ or not $wf(p)$ or not $wf(q)$

3.4 The heuristic function for checking strong equivalence

In order to apply an heuristic search, an heuristic function over nodes has to be defined. This function is called $\widehat{h}$ and it is aimed at working during the construction of the AND/OR graph by means of the operators of Table 1.

If n is a node in the AND/OR graph, $\widehat{h}$ should be an estimate of $h(n)$ (see Definition 2.4). In our method, $h(n)$ is the cost of the minimal counterexample sub-graph having n as a start node. Note that, if n has no counterexample below it, then $h(n) = \infty$.

Algorithm S2 is only guaranteed to work and find the minimal counterexample if the heuristics $\widehat{h}$ is admissible according to Definition 2.5. To meet the admissibility requisite, the heuristics we propose performs a limited look-ahead in the (yet to be constructed) AND/OR graph, in some of those cases where this can be done efficiently. The lookahead is limited, since this computation is still subject to combinatorial explosion and, if not truncated somewhere, it can easily make the cost of computing the heuristics overcame any benefit we may gain from it.

Definition 3.4 ($\widehat{h}(\langle p, q, \gamma, u\rangle)$*: the heuristic function*) $\widehat{h}(\langle p, q, \gamma, u\rangle)$ uses three auxiliary functions, one for each value of the γ component of the node:

- $\widehat{h}_\top$, when γ is equal to $\top$;
- $\widehat{h}_\bot$, when γ is equal to $\bot$;
- $\widehat{h}_\alpha$, when γ is equal to α; the latter is actually a family of functions one for each $\alpha \in Act$.

Each auxiliary function has three arguments: two CCS processes and a natural number. The three functions are defined in Table 3, while

$$\widehat{h}(\langle p,q,\gamma,u\rangle) = \begin{cases} \widehat{h}_\top(p,q,0) & \text{if } \gamma = \top; \\ \widehat{h}_\bot(p,q,0) & \text{if } \gamma = \bot, u = 1; \\ \widehat{h}_\bot(q,p,0) & \text{if } \gamma = \bot, u = 2; \\ \widehat{h}_\alpha(q,p,0) & \text{if } \gamma = \alpha, u = 1; \\ \widehat{h}_\alpha(p,q,0) & \text{if } \gamma = \alpha, u = 2. \end{cases}$$

The heuristic function guides the construction of the AND/OR structure aiming to find a node containing two not bisimilar states; in fact, $\widehat{h}(n)$ associates a nonnegative value with each node n of the structure, called the $\widehat{h}$-value of n: roughly speaking, which value approximates the number of arcs of the structure that must be crossed to establish that p and q are not bisimilar.

The auxiliary functions in Table 3 mimic a visit in the (not yet generated) AND/OR graph below node n. The visit is stopped when either the distance from n exceeds a fixed maximum M, or when the transitions of the CCS processes can no longer be obtained by inspecting their textual representation in node n. This latter constraint is checked by

Table 4 Restricted syntax for CCS processes

$Lin ::= nil \mid \alpha.p$	(where p is any CCS process)
$Sum ::= Lin \mid Lin + Sum$	
$Par ::= Sum \mid Sum'' \mid'' Par$	
$Res ::= Par\backslash L$	(where L is a set of actions)

the *wf* predicate. If p is the textual representation of a CCS process, we define $wf(p)$ to be true iff p can be generated by the *Res* production in the grammar defined in Table 4. It turn out that our heuristic returns a value different from zero for processes generated by this grammar. An example of a process generated by *Res* is

$$(\underline{a}.b.X + \underline{d}.e.Y \mid \underline{c}.Y \mid \underline{\bar{a}}.X)\backslash\{a\}.$$

In the process expression above, we have marked all the possible first actions (in the case of a and $\bar{a}$, these will actually produce τ). For each marked action, we can obtain the corresponding derivative of the process by keeping the same textual representation and changing the markings. E.g., if we let the process perform action d, we obtain

$$(a.b.X + d.\underline{e}.Y \mid \underline{c}.Y \mid \bar{a}.X)\backslash\{a\}$$

and we can use this as an equivalent representation of the actual derivative, i.e., $(e.Y \mid c.Y \mid \bar{a}.X)\backslash\{a\}$. Note that performing action c produces a derivative that can no longer be obtained from the grammar in Table 4. In fact, we choose to stop the lookahead when we meet a process constant, since the expansion of the constant may require a (possibly costly) process rewriting. Note, however, that this would be too restrictive at the beginning of the computation, since it is very common to have CCS processes defined as a parallel composition of constants. Therefore, when computing $\widehat{h}(\langle p, q, \gamma, u\rangle)$, we actually do a first pass in which we replace all constants with their definitions.

The actual expressions of the auxiliary functions are derived by Definition 2.4 and the operators of Table 1. Please keep in mind that we set the cost of each arc to 1, and that AND nodes and OR nodes are exchanged w.r.t. Table 1, since we are visiting the dual graph in order to look for counterexamples (see Sect. 3.2). To address special cases, we assume $\min \varnothing = \infty$ and $\sum \varnothing = 0$.

Consider the following two CCS processes

$$p = (c.k.nil \mid \overline{c}.nil)\backslash\{c\}$$
$$q = (d.h.nil \mid \overline{d}.nil)\backslash\{d\}$$

Let $n = \langle p, q, \bot, 1\rangle$, it holds that $\widehat{h}_\bot(p, q, 0) = 3$. In fact,

$$\begin{aligned}
\widehat{h}_\bot(p, q, 0) &= 1 + \widehat{h}_\tau((k.nil \mid nil)\backslash\{c\}, q, 1) && \text{Dfn. } \widehat{h}_\bot \\
&= 1 + 1 + \widehat{h}_\top((k.nil \mid nil)\backslash\{c\}, (h.nil \mid nil)\backslash\{d\}, 2) && \text{Dfn. } \widehat{h}_\alpha \\
&= 1 + 1 + \min(n_1, n_2) && \text{Dfn. } \widehat{h}_\top
\end{aligned}$$

where

$$n_1 = \widehat{h}_\bot((k.nil \mid nil)\backslash\{c\}, (h.nil \mid nil)\backslash\{d\}, 3); \quad \text{and}$$
$$n_2 = \widehat{h}_\bot((h.nil \mid nil)\backslash\{d\}, (k.nil \mid nil)\backslash\{c\}, 3).$$

Finally,

$$\begin{aligned}
n_1 &= 1 + (\widehat{h}_k((nil \mid nil)\backslash\{c\}, (h.nil \mid nil)\backslash\{d\}, 4) = 0), && \text{Dfn. } \widehat{h}_\bot \\
n_2 &= 1 + (\widehat{h}_h((nil \mid nil)\backslash\{d\}, (k.nil \mid nil)\backslash\{c\}, 4) = 0). && \text{Dfn. } \widehat{h}_\bot
\end{aligned}$$

Since $\widehat{h}_\bot(p, q, 0) = 3$, to reach two non bisimilar states we have to cross 3 arcs: the two arcs corresponding to the τ actions and the arc connecting the node $\top$ with a node $\bot$. Then, the value of $\widehat{h}(n)$ could be interpreted as a lower bound to the number of actions that have to be performed by both processes before reaching a node in which they are discovered not bisimilar. Nevertheless, this is an optimistic point of view: in fact, it is possible that

The following theorem states that the heuristic function is admissible, i.e., it never overestimates the actual cost.

Theorem 3.4 *The heuristic function $\widehat{h}$ of Definition 3.4 is admissible.*

Proof (See Definition 2.5) Let p and q be any two CSS processes. The AND/OR graph G, on which nodes the heuristics is defined, is $T^\partial(p, q)$. Let D be a minimal solution subgraph of G. We need to show that $\widehat{h}(n) \leq h(n)$ for each n in G, where $h(n)$ is the cost of n in D as defined in Definition 2.4.

Note that D is acyclic. Take any topological sort of the nodes in D, such that n comes before m whenever m is a successor of n in D. The proof is by induction on the number of nodes that come after any given node in the topological sort.

Preliminarily, note that $\widehat{h}_\top(p', q', j) \leq \widehat{h}_\top(p', q', i)$ whenever $i \leq j$, for any processes p' and q'. The same holds for $\widehat{h}_\bot$ and $\widehat{h}_\alpha$.

The base case of the induction is for the last node in the sort. This must be a terminal leaf n, with $h(n) = 0$. A terminal leaf is an AND node of $T^\partial(p, q)$, i.e., an OR node of $T(p, q)$, with no successors. Therefore, n must have the form $\langle \alpha, p', q', u\rangle$ for some processes p', q', $\alpha \in Act$ and $u \in \{1, 2\}$. Assume, without loss of generality, that $u = 2$. Then, since the node must have no successors, process q' must be unable to perform action α. We will have $\widehat{h}(n) = h_\alpha(p', q', 0) = 0$, either because $stop(p', q', 0)$ is true, or by computing $\sum \varnothing = 0$.

Now assume that $\widehat{h}(m) \leq h(m)$ for all nodes m that follow a node n in the topological sort, whenever n has at most k nodes after it, and consider a node n' followed by $k+1$ nodes.

Now, n' has the form $\langle \gamma, p', q', u \rangle$ for some p' and q', with $\gamma \in \{\top, \bot\} \cup Act$ and $u \in \{1, 2, \lambda\}$. We consider the possible cases for γ in turn.

Assume $\gamma = \top$. Then $u = \lambda$, node n' is an OR node of $T^{\partial}(p, q)$ and it has two successors after it: $m' = \langle \bot, p', q', 1 \rangle$ and $m'' = \langle \bot, p', q', 2 \rangle$. According to Definition 2.4, $h(n)$ is either $1 + h(m')$ or $1 + h(m'')$. By induction hypotesis, $\widehat{h}(m') \leq h(m')$ and $\widehat{h}(m'') \leq h(m'')$. Note that $\widehat{h}(m') = \widehat{h}_{\bot}(p', q', 0)$ and $\widehat{h}(m'') = \widehat{h}_{\bot}(q', p', 0)$. By looking at the definition of $\widehat{h}_{\top}$, we can see that we will have $\widehat{h}(n) = 0$ if $stop(p', q', 0)$ is true, or the following chain of inequalities will hold:

$$\begin{aligned}
\widehat{h}(n') &= \widehat{h}_{\top}(p', q', 0) \\
&= 1 + \min\left(\widehat{h}_{\bot}(p', q', 1), \widehat{h}_{\bot}(q', p', 1)\right) \\
&\leq 1 + \min\left(\widehat{h}_{\bot}(p', q', 0), \widehat{h}_{\bot}(q', p', 0)\right) \\
&= 1 + \min\left(\widehat{h}(m'), \widehat{h}(m'')\right) \\
&\leq 1 + \min\left(h(m'), h(m'')\right) \\
&\leq h(n').
\end{aligned}$$

The arguments for $\gamma = \bot$ and $\gamma \in Act$ are similar. □

3.5 The heuristic function for checking weak equivalence

The heuristics we use for weak equivalence is obtained from the heuristics for strong equivalence, by replacing $\longrightarrow$ with $\Longrightarrow$ in the defintions of $\widehat{h}_{\bot}$ and $\widehat{h}_{\alpha}$ in Table 3. A result analogous to Theorem 3.4 also holds for this case and can be proved similarly.

4 Experimental results

In this section, we present and discuss our experience with using a C++ tool[1] implementing the presented approach to check both strong and weak equivalence of several well-known processes. Our aim is to evaluate the performances of the approach presented in Sect. 3 and compare it against both the CWB-NC [32] and the CADP tool [33]. Experiments were executed on a 64 bit, 2.67 GHz Intel i5 CPU equipped with 8 GiB of RAM and running Gentoo Linux.

First, we consider the effect of the M constants in Table 3. This constant measures the amount of lookahead performed by the heuristics function. High values of M improve the heuristics, but also increase the time required to compute it. Therefore, we expect that incrementing M should be beneficial only up to a certain value. We consider a particularly problematic example (MUTUAL 8, described below) and use our tool to check for weak equivalence with different values of M. In Table 5, we show the running time, number of generated process states, and percentage of the running time spent computing the heuristics, for values of M ranging from 0 to 8. The data largely confirm our expectations: the running time decreases at first, but then it start to increase again for $M \geq 7$. The increase is due to the high computational cost of the heuristics, which becomes very significant starting at $M = 5$. Note that the number of generated process states continues to decrease, but at decreasing rate: it is essentially constant for $M \leq 6$. In the rest of the experiments, we select $M = 4$.

Table 5 Results on MUTUAL 8 (weak equivalence)

M	Time (s)	States	Heu (%)
0	99.7	3,335	0.3
1	79.2	3,302	0.6
2	64.7	3,089	1.6
3	51.7	3,068	4.1
4	45.8	2,866	6.3
5	43.0	2,810	14.2
6	42.0	2,698	29.4
7	48.2	2,698	40.5
8	66.2	2,684	58.9

Now, we consider different instances of the dining philosophers problem. In the first version, when a philosopher gets hungry, he can, without any control, pick up his left fork first, then his right one; after having eaten, he puts the forks down in the same order that he had picked them up. In the second version, shown in [34], when a philosopher gets hungry, he tries to sit at the table, but an usher keeps at least one philosopher from sitting. Only after having sit, a philosopher can pick up his left fork and then his right one; he eats and then puts the forks down in the same order that he picked them up. We prove that the two versions are not strongly equivalent.

Table 6 shows the number of generated nodes and the running time, measured in seconds, resulting from the CWB-NC and our approach, respectively, where column n indicates the number of philosophers. The column "dimension" shows the number of states of the standard transition system of the two CCS processes specifying the two versions of the dining philosophers, namely p and q. Finally, the last column indicates the cost of the counterexample. We may see a significant reduction of both state space and time when applying our approach. In fact, for $n = 6$, the CWB-NC tool was not able give an answer, while with our methodology we managed to check equivalence up to a configuration of 20 philosophers.

In Table 7, our approach is compared with the CADP tool. Again, we consider the dining philosophers example and check strong equivalence. Since CADP tool uses LOTOS [35] as specification language, all the CCS specifications have been equivalently translated in LOTOS. Moreover, the

[1] http://www2.ing.unipi.it/~a080224/grease/.

Table 6 Results on the dining philosophers (CWB-NC—strong equivalence)

n	Our approach		CWB-NC		Dimension		State-space reduction (%)	Cost of counterexample
	Gen	Time	Gen	Time	p	q		
2	41	0.00	156	0.10	74	82	73	14
3	57	0.01	1,072	0.14	639	433	94	14
4	125	0.05	7,212	1.53	5,510	1,702	98	14
5	203	0.15	53,815	19.89	47,496	6,319	99	14
6	305	0.39	–	–	–	–	–	14
8	597	1.78	–	–	–	–	–	14
12	1,687	18.20	–	–	–	–	–	14
16	3,600	92.73	–	–	–	–	–	14
20	6,472	337.96	–	–	–	–	–	14

Table 7 Results on the dining philosophers (CADP—strong equivalence)

n	Our approach memory	CADP memory	Our approach time	CADP time
2	10,480	51,600	0.00	0.60
3	11,616	57,840	0.01	0.66
4	14,432	62,672	0.05	0.77
5	20,112	67,824	0.15	1.01
6	29,408	72,992	0.39	1.69
8	71,664	87,890	1.78	6.97
12	375,392	–	18.20	–
16	1,372,048	–	92.73	–
20	3,749,648	–	337.96	–

memory usage is considered instead of the generated states. The table shows both the memory consumption (KB) and the runtime performance (sec) of both approaches: again our method obtains better results. Actually, we set a timeout of 15 minutes during the checking of 12 philosophers trying to dine.

The goodness of the proposed heuristic function is estimated by the comparison of the results of using our approach with the function defined in the previous section, called $\widehat{h}$, and with an heuristic function always equal to zero, from now on called $\widehat{h}_0$. Essentially, by means of $\widehat{h}_0$, we simulate a breath-first search strategy. The results of checking strong equivalence between the two versions of dining philosophers are shown in Table 8, where the second and third columns show the number of states (gen) generated using $\widehat{h}_0$ and $\widehat{h}$, respectively; moreover, the fourth and fifth column show the time results in terms of seconds. Is easy to see that the non informative heuristic $\widehat{h}_0$ has both the disadvantages of a higher computational cost and a less effectiveness. In fact, the search with the non informative heuristics is not able to give an answer for 16 dining philosophers after having waited more than 15 minutes, while we manage the checking of strong equivalence up to a configuration of 20 philosophers through $\widehat{h}$.

For a more complete evaluation of our approach, we select from the literature a sample of well-known systems. In all examples, we prove the equivalence between two processes, namely p and q.

- DEK-PET
  - p: the CCS specification of the mutual exclusion algorithms [36] due to Dekker.
  - q: the CCS specification of the mutual exclusion algorithms [36] due to Peterson.
  - It holds that $p \not\sim q$.
- BUFF:
  - p: a buffer of capacity 2.
  - q: the implementation of the buffer obtained by composing in parallel 2 copies of a buffer cell.
  - It holds that $p \not\sim q$.
- XOR:
  - p: specification of an XOR of 3 inputs.
  - q: the implementation obtained using two XOR gates of 2 inputs each one.
  - It holds that $p \not\sim q$.

Table 8 Results on the dining philosophers ($\widehat{h}$ vs $\widehat{h}_0$)

n	Gen		Time	
	Our approach $\widehat{h}_0$	Our approach $\widehat{h}$	Our approach $\widehat{h}_0$	Our approach $\widehat{h}$
2	60	41	0.00	0.00
3	163	57	0.01	0.01
4	348	125	0.07	0.05
5	664	203	0.26	0.15
6	1,170	305	0.79	0.39
8	3,070	597	5.15	1.78
12	13,768	1,687	107.32	18.20
16	–	3,600	–	92.73
20	–	6,472	–	337.96

Table 9 Results for other systems (CWB-NC—strong equivalence)

Case study	Our approach		CWB-NC		Dimension		State-space reduction (%)	Cost of counterexample
	Gen	Time	Gen	Time	p	q		
DEK-PET	36	0.00	256	0.02	165	91	86	17
BUFF	7	0.00	17	0.01	7	10	58	5
XOR	8	0.00	18	0.01	8	10	55	8
MUTUAL	1,532	0.03	8,704	2.89	6,912	6,912	82	20
MAIL	68	0.00	1,434	0.24	1,025	409	95	17
BRP	1,518	0.01	1,518	0.60	759	759	0	Strong bisimilar

- MUTUAL:
  - p: a system handling the requests of a resource shared by 8 processes. It presents two alternative choices between a server based on a round robin scheduling and a server based on mutual exclusion.
  - q: similar to p with the round robin scheduling changed.
  - It holds that $p \not\sim q$.
- MAIL:
  - p: an old specification of a mail system, devised by Gordon Brebner [37].
  - q: a new specification of a mail system, always produced by Gordon Brebner [37].
  - It holds that $p \not\sim q$.
- BRP: Philips Bounded Retransmission Protocol (BRP): the Bounded Retransmission Protocol used by the Philips Company in one of its products [38–40].
  - p and q are two similar specifications of the protocol.
  - It holds that $p \sim q$.

The results of all runs are reported in Table 9 (comparison with CWB-NC) and Table 10 (comparison with CADP), while Table 11 shows the results of our approach against CADP when checking weak equivalence. The tables show the number of generated nodes/memory usage and the running time, measured in seconds. As we can see in Table 9, no space reduction is obtained for BRP since our approach is only successful when applied to non-equivalence checking. Our tool is generally much faster than CWB-NC, but this is mainly a fault of CWB-NC implementation. On BRP, CADP takes several seconds to complete, while our tool completes in a fraction of a second; this result, however, may be due to the fact that the BRP LOTOS specification (coming from CADP's demos) is slightly different from our CCS specification. Accordingly, we have omitted the BRP results from Table 10. For what regards weak equivalence, while its memory occupation is higher, CADP shows a better performance, in particular, the execution time in the MUTUAL case study is about 3 s against 50 s of our tool. In fact, CADP uses very efficient algorithms, and great attention is devoted to implementation efficiency issues. Actually, for the MUTUAL case, also when checking strong equivalence, CADP has better time performance. This is mainly due to the structure of the two CCS processes that are modeled with a very low level of structural difference The high execution time is also due to the overhead introduced by the S2 algorithm. For the MUTAL case study, through measurements of the time we can confirm that the computational complexity of our approach mainly depends on the S2 algorithm. In fact, the Bottom_Up step of

Table 10 Results for other systems (CADP—strong equivalence)

Case study	Our approach memory	CADP memory	Our approach time	CADP time
DEK-PET	10,416	50,848	0.00	0.32
BUFF	9,856	44,784	0.00	0.30
XOR	9,904	46,464	0.00	0.40
MUTUAL	41,440	63,392	8.03	0.70
MAIL	91,504	53,008	0.00	1.44

Table 11 Results for other systems (CADP—weak equivalence)

Case study	Our approach memory	CADP memory	Our approach time	CADP time	Weak bisimilar?
DEK-PET	45,808	73,472	4.11	0.45	No
BUFF	10,048	46,544	0.00	0.32	Yes
XOR	10,048	47,584	0.00	0.32	Yes
MUTUAL	107,168	105,712	49.12	2.47	No
MAIL	14,272	106,960	0.02	0.46	No

Fig. 3 requires more than seventy percent (70 %) of the time for detecting processes similarities, while the time required to compute the heuristic function is negligible. Also the authors of S2 have found that the algorithm is much slower than the competing algorithm CFC_{REV^*}, for example, but we found that S2 is relatively easy to explain and it is proved to reach the minimum counterexample (this is the main reason of our choice). Similar algorithms in literature with better performances pay the price of a very complex behavior. However, the objective of our paper is to propose a heuristic-based methodology to check non-equivalence regardless of the used algorithm and our main aim is checking equivalences trying to save as memory space as possible and obtain the minimum counterexample in the case of not equivalence. It is worth noting that the presented approach has the advantage that a good heuristic function can help to obtain both memory saving (less nodes to explore) and low execution time (less work to do).

From the point of view of our initial goal, with these examples we have provided some experimental evidence of the reduction of both state space and running time that may result when applying our methodology with respect to CWB-NC and CADP, when checking strong equivalence. Note that in some cases we obtain a reduction of more than 99 %. However, for weak equivalence, our approach outperforms CWB-NC, while its performance, with particular reference to the scalability of the analysis, is lower than CADP, even if the tests performed so far show that our approach based on heuristic searches is comparable to the CADP tool in terms of memory space reduction. For the time being, our impression is that the technique looks promising via further refinements of the method. For example, we investigated the use of more accurate heuristic functions (and more complex) especially for weak equivalence; in fact, as the accuracy of the heuristics improves, the amount of search required to find a solution and the time for obtaining the solution both decrease. As said before and as evidenced by the experiments, the S2 algorithm for detecting processes similarities has not very good performances especially due to the Bottom_Up step of Fig. 3. However, we think that the heuristic-based approach is a viable solution, due to the availability of several algorithms with better performances. Note that the research of algorithms for cyclic AND/OR graphs is very recent, thus we hope that new more efficient algorithms can be proposed in the future.

In our approach particular attention is paid to the representation of counterexamples. In fact, in formal verification, learning why a system fails or passes a verification task, could be as important as the result itself. The CWB-NC supports a game theoretic representation of counterexamples and, if two systems are non-equivalent, generates a logic formula as diagnostic information. However, often this formula is fairly hard to understand and is inadequate to be used for debugging the model. Our approach returns a graphical AND/OR structure representation, which is the minimal sub-graph leading to two non-equivalent states. This structure allows the user to understand and navigate through the counterexample better, especially in a setting with highly nondeterministic automata where counterexamples may become rather complicated. For example, consider the following CCS processes:

$$p \stackrel{\text{def}}{=} a.(b.e.nil + b.k.nil)$$

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

It holds that $p \not\sim q$. The CWB-NC returns as counterexample:

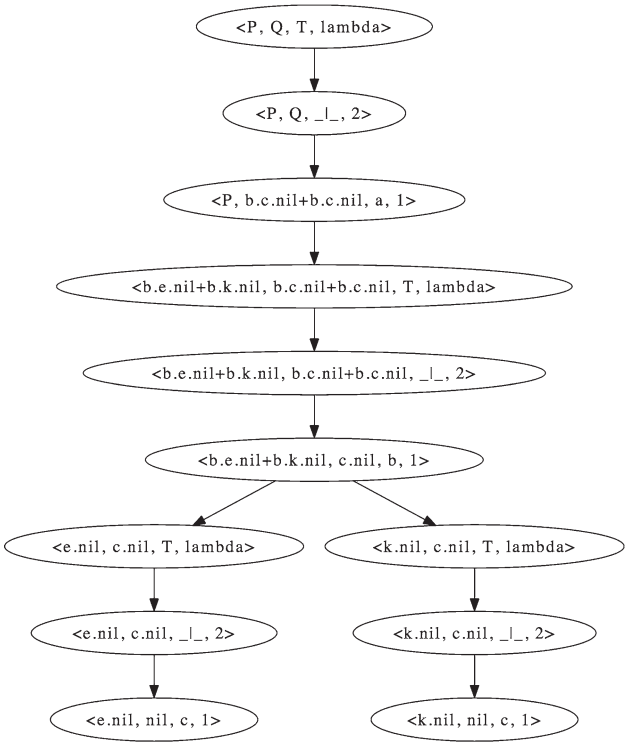

Fig. 5 Our counterexample

```
FALSE...
p satisfies:
<a><b><e>tt
q does not.
```

The graphical structure representation returned by our approach is shown in Fig. 5. Note that CWB-NC returns a logic formula and not the specific execution that causes the false value; moreover, in some cases, the formula can be difficult to understand for users that are unfamiliar with temporal logic-based specifications.

5 Conclusion and related work

A method that uses heuristic searches has been proposed for equivalence checking for concurrent systems. The novel contributions of our work are the following.

- Application of heuristic search for equivalence checking. As far as we know, it is the first attempt to exploit process algebra-based heuristics for equivalence checking in concurrent systems.
- The definition of an admissible heuristic function. In this way, using S2, we can always find minimal graph leading to two non-equivalent states. We believe that it is important to return the minimal graph, since that graph is examined in order to determine the source of the error. Big graphs can prevent an easy comprehension of the fault.
- 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.

The most challenging task when applying automated model checking in practice is to conquer the state explosion problem. Hence, equivalence algorithms with minimal space complexity are of particular interest. Two algorithmic families can be considered to perform equivalence checking. The first one is based on refinement principle: *given an initial partition, find the coarsest partition stable with respect to the transition relation* see for example the algorithm proposed by Paige and Tarjan in [41]. The other family of algorithms is based on a Cartesian product traversal from the initial state [42,43]. These algorithms are both applied on the whole state graph, and they require an explicit enumeration of this state space. This approach leads to the well-known state explosion problem. Classical reduction algorithms already exist [44,45], but they can be applied only when the whole state space has been computed, which limits their interest. A possible solution is to reduce the state graph before performing the check as shown in [46] where symbolic representation of the state space is used. In [4] is presented an algorithm that allows the minimization of the graph during its generation, thus avoiding in part the state explosion problem. The main problems of this algorithm arise from the model itself, a system of communicating automata, where some base automata can be bigger than the full model itself. We avoid this problem since we use an heuristic that guides the generation of states that are still belonging to the standard transition systems of the two CCS processes under consideration.

Other algorithms are the ones by Bustan and Grumberg [47] and by Gentilini, Piazza and Policriti [48]. For an input graph with N states, T transitions and S simulation equivalence classes, the space complexity of both algorithms is $\mathcal{O}(S^2 + N \log S)$. The approach of Gentilini et al. represents the simulation problem as a generalized coarsest partition problem. Our heuristic approach can produce great reduction in space, which becomes the bottleneck as the input graph grows, especially when two processes do not bisimulate each other.

Recently, great interest was shown in combining 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 [49] is one of the original ones. They enhance the bug-finding capability of a model checker using heuristics to search the states that are most likely to lead to an error. In [50], genetic algorithms are used to exploit heuristics for guiding a search in large state spaces toward errors like deadlocks and assertion violations. In [51], heuristics

have been used for real-time model checking in UPPAAL. In [52,53], heuristic search has been combined with on-the-fly techniques, while in [54,55] with symbolic model checking. Other works, as for example [56,57], used heuristic search to accelerate finding errors, while in [58] heuristic search is used to accelerate verification.

As a future work, we intend apply this approach also for other equivalences, as for example ρ-equivalence, introduced in [18], that formally characterizing the notion of "the same behavior with respect to a set ρ of actions": *two transition systems are* ρ-**equivalent** *if a* ρ*-bisimulation relating their initial states exists*. Moreover, we intend to investigate more accurate heuristic functions.

References

1. Parashkevov, A.N., Yantchev, J.: Arc—a tool for efficient refinement and equivalence checking for csp. In: IEEE International Conference on Algorithms and Architectures for Parallel Processing ICA3PP'96, pp. 68-75
2. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Grumberg, O., Veith, H. (eds.) 25 Years of Model Checking, vol. 5000 of Lecture Notes in Computer Science, Springer, pp. 196-215 (2008)
3. Queille, J.-P., Sifakis, J.: Specification and verification of concurrent systems in cesar. In: Dezani-Ciancaglini, M., Montanari, U. (eds.) Symposium on Programming, vol. 137 of Lecture Notes in Computer Science, Springer, Berlin, pp. 337–351 (1982)
4. Bouajjani, A., Fernandez, J.-C., Halbwachs, N.: Minimal model generation. In: Clarke, E.M., Kurshan, R.P. (eds.) CAV, vol. 531 of Lecture Notes in Computer Science, Springer, Berlin pp. 197–203 (1990)
5. Graf, S., Steffen, B., Lüttgen, G.: Compositional minimisation of finite state systems using interface specifications. Form. Asp. Comput. **8**, 607–616 (1996)
6. McMillan, K.L.: Symbolic Model Checking. Kluwer, Dordrecht (1993)
7. Jard, C., Jéron, T.: Bounded-memory algorithms for verification on-the-fly, in: [60], pp. 192–202
8. Edelkamp, S., Schuppan, V., Bosnacki, D., Wijs, A., Fehnker, A., Aljazzar, H.: Survey on directed model checking. In: MoChArt, pp. 65-89
9. Gradara, S., Santone, A., Villani, M.L.: Using heuristic search for finding deadlocks in concurrent systems. Inf. Comput. **202**, 191–226 (2005)
10. Gradara, S., Santone, A., Villani, M.L.: An efficient deadlock detection tool for ccs processes. J. Comput. Syst. Sci. **72**, 1397–1412 (2006)
11. Stirling, C., Walker, D.: Local model checking in the modal mu-calculus. Theor. Comput. Sci. **89**, 161–177 (1991)
12. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems—An Approach to the State-Explosion Problem, vol. 1032 of Lecture Notes in Computer Science, Springer, Berlin (1996)
13. Peled, D.: All from one, one for all: on model checking using representatives, in: [59], pp. 409–423
14. Valmari, A.: A stubborn attack on state explosion. Form. Methods Syst. Des. **1**, 297–322 (1992)
15. Santone, A., Vaglini, G., Villani, M.L.: Incremental construction of systems: an efficient characterization of the lacking sub-system. Sci. Comput. Program. **78**, 1346–1367 (2013)
16. Clarke, E.M., Long, D.E., McMillan, K.L.: Compositional model checking. In: LICS, pp. 353-362
17. Santone, A.: Automatic verification of concurrent systems using a formulabased compositional approach. Acta Inf. **38**, 531–564 (2002)
18. Barbuti, R., De Francesco, N., Santone, A., Vaglini, G.: Selective mu-calculus and formula-based equivalence of transition systems. J. Comput. Syst. Sci. **59**, 537–556 (1999)
19. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM Trans. Program. Lang. Syst. **16**, 1512–1542 (1994)
20. Santone, A., Vaglini, G.: Abstract reduction in directed model checking ccs processes. Acta Inf. **49**, 313–341 (2012)
21. De Francesco, N., Santone, A., Vaglini, G.: State space reduction by nonstandard semantics for deadlock analysis. Sci. Comput. Program. **30**, 309–338 (1998)
22. Pearl, J.: Heuristics—Intelligent Search Strategies for Computer Problem Solving. Addison-Wesley series in artificial intelligence. Addison-Wesley (1984)
23. Mahanti, A., Bagchi, A.: And/or graph heuristic search methods. J. ACM **32**, 28–51 (1985)
24. Mahanti, A., Ghose, S., Sadhukhan, S.K.: A framework for searching and/or graphs with cycles, CoRR cs.AI/0305001 (2003)
25. Milner, R.: Communication and Concurrency, PHI Series in Computer Science. Prentice Hall, Englewood Cliffs (1989)
26. Santone, A.: Heuristic for simulation checking. In: Proceedings of the 2011 International Conference on Artificial Intelligence (ICAI 2011), Las Vegas Nevada, USA, CSREA Press (2011)
27. Santone, A.: Clone detection through process algebras and java bytecode. In: IWSC, pp. 73-74
28. Cuomo, A., Santone, A., Villano, U.: Cd-form: a clone detector based on formal methods. Sci. Comput. Program. (2013) doi:10.1016/j.scico.2013.11.022
29. Chakrabarti, P.P.: Algorithms for searching explicit and/or graphs and their applications to problem reduction search. Artif. Intell. **65**, 329–345 (1994)
30. Jiménez, P., Torras, C.: An efficient algorithm for searching implicit and/or graphs with cycles. Artif. Intell. **124**, 1–30 (2000)
31. Hansen, E.A., Zilberstein, S.: Lao*: A heuristic search algorithm that finds solutions with loops. Artif. Intell. **129**, 35–62 (2001)
32. Cleaveland, R., Sims, S.: The ncsu concurrency workbench. In: Alur, R. , Henzinger, T.A. (eds.) CAV, vol. 1102 of Lecture Notes in Computer Science, Springer, Berlin pp. 394-397 (1996)
33. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: Cadp 2010: a toolbox for the construction and analysis of distributed processes. In: TACAS, pp. 372-387
34. Bruns, G.: A practical technique for process abstraction. In: Best, E. (ed.) CONCUR, vol. 715 of Lecture Notes in Computer Science, Springer, Berlin, pp. 37–49 (1993)
35. Bolognesi, T., Brinksma, E.: Introduction to iso specification language lotos. Comput. Netw. ISDN Syst. **14**, 25–59 (1987)
36. Walker, D.J.: Automated analysis of mutual exclusion algorithms using ccs. Form. Asp. Comput. **1**, 273–292 (1989)
37. Brebner, G.J.: A ccs-based investigation of deadlock in a multiprocess electronic mail system. Form. Asp. Comput. **5**, 467–478 (1993)
38. Groote, J.F., van de Pol, J.: A bounded retransmission protocol for large data packets. In: Wirsing, M., Nivat, M. (eds.) AMAST, vol. 1101 of Lecture Notes in Computer Science. Springer, Berlin, pp. 536–550 (1996)
39. Havelund, K., Shankar, N.: Experiments in theorem proving and model checking for protocol verification. In: Gaudel, M.-C., Woodcock, J. (eds.) FME, vol. 1051 of Lecture Notes in Computer Science, Springer, Berlin, pp. 662-681 (1996)
40. Helmink, L., Sellink, M.P.A., Vaandrager, F.W.: Proof-checking a data link protocol. In: Barendregt, H., Nipkow, T. (eds.) TYPES,

vol. 806 of Lecture Notes in Computer Science, Springer, Berlin, pp. 127–165 (1993)
41. Paige, R., Tarjan, R.E.: Three partition refinement algorithms. SIAM J. Comput. **16**, 973–989 (1987)
42. Fernandez, J.-C., Mounier, L.: "On the fly" verification of behavioural equivalences and preorders. In: Larsen K.G., Skou A. (eds.) CAV'91 Proceedings of the 3rd International Workshop on Computer Aided Verification, pp. 181–191. Springer-verlag, London, Uk (1991)
43. Godskesen, J., Larsen, K., Zeeberg, M.: TAV—Tools for Automatic Verification: Users Manual. Department of Mathematics and Computer Science, The University of Aalborg, Institute for Electronic Systems (1989)
44. Fernandez, J.-C.: An implementation of an efficient algorithm for bisimulation equivalence. Sci. Comput. Program. **13**, 219–236 (1989)
45. Kanellakis, P.C., Smolka, S.A.: Ccs expressions, finite state processes, and three problems of equivalence. Inf. Comput. **86**, 43–68 (1990)
46. Fernandez, J.-C., Kerbrat, A., Mounier, L.: Symbolic equivalence checking. In: CAV'93 Proceedings of the 5th International Conference on, Computer Aided Verification, June 28-July 1. Lecture Notes in Computer Science. vol. 697, pp. 85-96. Springer, London, UK (1993)
47. Bustan, D., Grumberg, O.: Simulation-based minimization. ACM Trans. Comput. Log. **4**, 181–206 (2003)
48. Gentilini, R., Piazza, C., Policriti, A.: From bisimulation to simulation: coarsest partition problems. J. Autom. Reason. **31**, 73–103 (2003)
49. Yang, C.H., Dill, D.L.: Validation with guided search of the state space. In: DAC, pp. 599–604
50. Godefroid, P., Khurshid, S.: Exploring very large state spaces using genetic algorithms. In: Katoen, J.-P., Stevens, P. (eds.), TACAS, vol. 2280 of Lecture Notes in Computer Science, Springer, Berlin, pp. 266–280 (2002)
51. Behrmann, G., Fehnker, A.: Efficient guiding towards cost-optimality in uppaal. In: Margaria, T. , Yi, W. (eds.) TACAS, vol. 2031 of Lecture Notes in Computer Science, Springer, Berlin, pp. 174–188 (2001)
52. Alur, R., Wang, B.-Y.: "Next" heuristic for on-the-fly model checking. In: Baeten, J.C.M., Mauw, S. (eds.) CONCUR, vol. 1664 of Lecture Notes in Computer Science, Springer, Berlin, pp. 98-113 (1999)
53. Möller, M.O., Alur, R.: Heuristics for hierarchical partitioning with application to model checking. In: Margaria, T., Melham, T.F. (eds.) CHARME, vol. 2144 of Lecture Notes in Computer Science, Springer, Berlin, pp. 71–85 (2001)
54. Edelkamp, S., Reffel, F.: Obdds in heuristic search. In: Herzog, O., Günter, A. (eds.), KI, vol. 1504 of Lecture Notes in Computer Science, Springer, Berlin, pp. 81–92 (1998)
55. Jensen, R.M., Bryant, R.E., Veloso, M.M.: Seta*: an efficient bdd-based heuristic search algorithm. In: AAAI/IAAI, pp. 668-673
56. Edelkamp, S., Lluch-Lafuente, A., Leue, S.: Directed explicit model checking with hsf-spin. In: Dwyer, M.B. (ed.) SPIN, vol. 2057 of Lecture Notes in Computer Science, Springer, Berlin, pp. 57–79 (2001)
57. Groce, A., Visser, W.: Heuristic model checking for java programs. In: Bosnacki, D., Leue, S. (eds.) SPIN, vol. 2318 of Lecture Notes in Computer Science, Springer, Berlin pp. 242–245 (2002)
58. Santone, A.: Heuristic search + local model checking in selective mucalculus. IEEE Trans. Softw. Eng. **29**, 510–523 (2003)

Nicoletta De Francesco was born in Florence in 1951. In 1974 she obtained the degree in Computer Science (cum laude) from the University of Pisa. From 1981 she was researcher at the Dipartimento di Informatica of the University of Pisa, till 1989. From 1989 till 2000, november, she was associate professor, first at the University of Salerno, and then at the University of Pisa. From december 2000 she is full professor of Computer Engineering at the Dipartimento di Ingegneria della Informazione of the University of Pisa. From 2003 to 2010 she was delegate for teaching of the University of Pisa and from november 2010 she is vice-rector of the University of Pisa. She made her research in the field of the specification and verification of concurrent and distributed systems. Her research interests included the formal verification of systems. After, she moved to the security field and to the application of formal verification techniques to biological systems and languages. She has been involved in several reasearch projects under the financial support of the Italian Ministero dell'Istruzione, dell'Università e della Ricerca (MIUR), the Italian National Research Council (CNR) and the European Community. In 2010 she winned the Computer Journal Wilkes Award.

Giuseppe Lettieri received the Ph.D. degree in Computer Systems Engineering from the University of Pisa, Italy, in 2002. Since 2004 he is a researcher with the Dipartimento di Ingegneria dell'Informazione of the University of Pisa. He has worked on distributed Operating Systems and memory management, then he moved to formal methods, with special interest in model checking and applications of the theory of Abstract Interpretation to Java bytecode verification and secure information flow. Giuseppe Lettieri has been involved in research projects under the finantial support of the Italian Ministero dell'Istruzione, dell'Università e della Ricerca (MIUR) and the Cassa di Risparmio di Pisa.

Antonella Santone was born in Montreal (Canada) on June 13, 1969. She is associate professor at the Department of Engineering of the University of Sannio, Benevento since November 2001. She received the Laurea degree in Computer Science at the University of Pisa, Italy, in April 1993. In September 1997 she received the Ph.D. degree in Computer Systems Engineering at the Dipartimento di Ingegneria della Informazione, Pisa. She has been Assistant Professor at the University of Pisa from November 1998 till October 2001. She was involved in several research activities and projects. Antonella Santone's current research is focused on the application of formal verification methods. Her research interests include formal description techniques, temporal logic, concurrent and distributed systems modelling, heuristic search.

Gigliola Vaglini is full professor of Computer Engineering at the Dipartimento di Ingegneria della Informazione of the University of Pisa. She was born in Pisa in 1952. In 1974 she obtained her Doct. degree in Computer Science from the University of Pisa. She held a post-graduate scholarship from the University of Pisa and from 1981 she was a research assistant at the Dipartimento di Informatica of the University of Pisa. From 1989 till 2002, she was an associate professor, first at the University of Naples, and then at the University of Pisa. The main field of her research activity was the specification and verification of concurrent and distributed systems. In particular, she worked on the model checking of both the formal specifications of concurrent systems and the programs. She has been involved in several reasearch projects under the financial support of the Italian Ministero dell'Istruzione, dell'Università e della Ricerca (MIUR), the Italian National Research Council (CNR) and the European Community.

View publication stats