Heuristic Search for Equivalence Checking

Nicoletta De Francesco[a], Giuseppe Lettieri[a], Antonella Santone[b], Gigliola Vaglini[a]

[a]*Dipartimento di Ingegneria dell'Informazione, University of Pisa, Pisa, Italy*
[b]*Dipartimento di Ingegneria, University of Sannio, Benevento, Italy*

Abstract

Equivalence checking plays a crucial role in formal verification and model checking. It is a natural relation to use for matching a system implementation with its specification. In this paper we present an efficient procedure, based on heuristic search, for checking well-know bisimulation equivalences for concurrent systems specified using process algebras. An heuristic mechanism for the exploration of the search space is used, in order to avoid the construction of the complete state graph. The method is evaluated on several examples exploiting a prototype that shows that a considerable reduction of the state space size can be achieved.

Keywords: heuristic searches, equivalence checking, formal methods.

---

1. Introduction

Verification of concurrent systems is often carried out by means of the analysis of the state space generated by the system. A very used method is *model checking* that is a technique originating in works by Clarke and Emerson [1] and Queille and Sifakis [2] from the early 1980s. It is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model. An alternative 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. Depending on the type of relation that is chosen, this verification technique is called *refinement* or *equivalence checking* [3].

The verification of concurrent systems based on state exploration suffers 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 on the order of millions or billions of states. When the number of states is too large to fit in a computer's main memory, automated verification quickly breaks down.

Email addresses: `n.defrancesco@unipi.it` (Nicoletta De Francesco), `g.lettieri@iet.unipi.it` (Giuseppe Lettieri), `santone@unisannio.it` (Antonella Santone), `g.vaglini@iet.unipi.it` (Gigliola Vaglini)

Several approaches have been developed to solve or reduce the state explosion problem. They are general methods that typically are used to reduce the state space explosion while verifying a set of properties. Among them, reduction techniques based on process equivalences [4, 5], symbolic model checking techniques [6], on-the-fly techniques [7], heuristic searches [8, 9, 10], local model checking approaches [11], partial order techniques [12, 13, 14], compositional techniques [15, 16], and abstraction approaches [17, 18].

We propose to check equivalence of concurrent systems by applying *heuristic search* techniques on AND/OR graphs. Heuristic search [19] 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 and path finding problems. 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. 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* [20, 19] is the most widely known algorithm that requires the AND/OR graph to be acyclic, while S2 [21], which will be used here, is an algorithm designed to work 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 optimality of the solution supplied by the algorithm is guaranteed by the property of admissibility of the heuristic function, i.e., the function never overestimates the distance to the goal.

In this paper, concurrent systems are specified by means of process algebras and the equivalence checking of processes is formalised as a search problem on AND/OR graphs. Then an equivalence checking procedure is presented that is based on S2 and uses admissible heuristic functions for weak and strong equivalences. To evaluate the method, the heuristic functions are syntactically defined considering a specific process algebra: the Calculus of Communicating Systems (CCS) [22]. 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 not equivalent states. We believe that it is important to return the minimal graph, since that graph will be examined in order to determine the source of the error. Big graphs can prevent an easy comprehension of the fault. This approach extends traditional techniques to efficiently explore the search space. The heuristic overcomes the bottleneck of the exhaustive exploration of the global state graph of the two systems. Moreover, it is possible to apply our approach also in verifying infinite concurrent systems.

To show that a significant space reduction may be obtained with respect to other approaches, a prototype tool is built implementing the presented method and several experiments are carried on processes of different sizes. As far as we know, this is the first attempt to exploit process algebra-based heuristics for equivalence checking in concurrent systems. A preliminary version of the results presented in this paper can be found in [23], where the simulation relation defined by Milner [22] has been considered.

The paper is organised as follows: in Section 2 the basic concepts of behavioral equivalence and of the heuristic search algorithms are recalled. Section 3 describes our approach. In Section 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 Section 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 $(p', \alpha) \in s(p)$, we say that p can perform the action α and reach the process p', and we write $p \xrightarrow{\alpha} p'$.

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.*

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 (strong equivalence)**.** *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 $p \overset{\epsilon}{\Longrightarrow} q$ 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 $p \overset{\alpha}{\Longrightarrow} q$ iff there are processes p' and q' such that

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

Thus, $p \overset{\alpha}{\Longrightarrow} q$ 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. *(weak equivalence). 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) $p \xrightarrow{\alpha} r'$ *implies* $\exists q'$ *such that* $q \overset{\widehat{\alpha}}{\Longrightarrow} q'$ *with* $p'\ \mathcal{B}\ q'$*; and*
  (ii) $q \xrightarrow{\alpha} q'$ *implies* $\exists p'$ *such that* $p \overset{\widehat{\alpha}}{\Longrightarrow} p'$ *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).*

2.1. Heuristic search: AND/OR Graphs and Algorithm S2

In this section we briefly review AND/OR graphs and the heuristic search algorithm S2 [21] 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 graph $G = (N, A)$ if $N' \subseteq N$ and $A' \subseteq A$. 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.

An AND/OR graph G is a directed graph with a special node s, called the *start* (or *root*) *node*, and a nonempty 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.

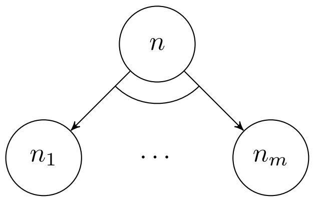

Figure 1: An AND node.

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*;

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.

Table 1: The S2 algorithm.

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 [20, 19] only works with AND/OR graphs, both explicit and implicit, that do not contain cycles; for cyclic AND/OR graphs we have algorithm REV* [24], which only works on explicit graphs, and algorithms $\mathrm{CFC}_{\mathrm{REV}^*}$ [25] and S2 [21] 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* [26] removes this limitation in the context of Markov decision problems.

In this paper we use the algorithm S2 reproduced in Table 1. 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

```
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 $j = \mathrm{argmin}_{1 \le i \le r}\{c(q, q_i) + h(q_i)\}$ 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 $h(s) = \infty$.
```

Table 2: The Bottom_Up(List OPEN) procedure.

(i). This map is updated during step (ii), together with the node costs. Step (ii) uses the procedure Bottom_Up defined in Table 2. 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 Table 1) satisfies the so-called *admissibility* condition, i.e. $\widehat{h}$ is optimistic. More formally:

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

$$\widehat{h}(n) \le h(n).$$

3. The method

In this section we explain the basis of our approach to strong equivalence checking. In the sub-section 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 subsection 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 subsection 3.3 we apply the method to the CCS language. In subsection 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 the in subsection 3.5 the heuristic function for weak equivalence is defined. Finally, subsection 3.6 shows how our approach can be also applied to infinite concurrent systems.

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 $t\colon N \to \{AND, OR\}$.*

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) it is required *at least* one successor 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 labelling as non terminal leaves.

Now it is defined 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 $t^\partial\colon N \to \{AND, OR\}$ 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 $1 \le i \le k$, 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 $1 \le i < k$; (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 4-uples $\langle r, s, \gamma, u\rangle$ 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 3, 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

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

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:

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

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

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

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

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

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

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

Table 3: The operators.

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 3. Then $p \sim q$ iff $T(p,q)$ has a solution.*

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

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

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 3) 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 $T(p, q)$. $\square$

3.2. Heuristic search for strong equivalence

The AND/OR structure $T(p, q)$ built in the previous section can be readily interpreted as an AND/OR graph, so that the heuristic search algorithm S2 can be applied to it. However, algorithm S2 searches for solutions of AND/OR graphs as defined in Definition 2.3, while we are interested in the solutions of AND/OR structures as defined in Definition 3.2.

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 and, as an additional result, the heuristic search has found 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.

The other differences between Definition 2.3 and Definition 3.2 are minor. Note that Definition 2.3 is more restrictive than Definition 3.2, so that any solution found by S2 is automatically a solution of $T^{\partial}(p, q)$. In the other direction, assume D is an acyclic solution of $T^{\partial}(p, q)$. Since we are assuming that the processes are 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 are terminal nodes, 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 still a solution of $T^{\partial}(p,q)$ according to Definition 3.2, but it is also a solution according to Definition 2.3. Therefore S2 finds a solution iff $T^{\partial}(p,q)$ has a solution.

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) [22], which is an algebra suitable for modelling and analysing processes. The reader can refer to [22] for further details. The syntax of *processes* is the following:

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

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 $x \overset{\text{def}}{=} p$. Given $L \subseteq \mathcal{V}$, with L° we denote the set $\{\, \overline{l}, l \mid l \in L \,\}$. 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.

The standard *operational semantics* [22] 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 4 (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 $\delta = \alpha_1 \dots \alpha_n$, $n \geq 1$, 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.

Given a process p, we define $\mathcal{F}(p) = \{\, \alpha \in Act \mid \exists p' \text{ s.t. } p \xrightarrow{\alpha} p' \,\}$ as 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:

$$
\begin{array}{llll}
\textbf{Act} & \dfrac{}{\alpha.p \xrightarrow{\alpha} p} & \textbf{Sum} & \dfrac{p \xrightarrow{\alpha} p'}{p + q \xrightarrow{\alpha} p'} \\[2ex]
\textbf{Con} & \dfrac{p \xrightarrow{\alpha} p'}{x \xrightarrow{\alpha} p'}\ x \overset{\text{def}}{=} p & \textbf{Par} & \dfrac{p \xrightarrow{\alpha} p'}{p \,|\, q \xrightarrow{\alpha} p' \,|\, q} \\[2ex]
\textbf{Com} & \dfrac{p \xrightarrow{l} p',\ q \xrightarrow{\bar{l}} q'}{p \,|\, q \xrightarrow{\tau} p' \,|\, q'} & \textbf{Rel} & \dfrac{p \xrightarrow{\alpha} p'}{p[f] \xrightarrow{f(\alpha)} p'[f]}
\end{array}
$$

$$
\textbf{Res} \quad \dfrac{p \xrightarrow{\alpha} p'}{p \backslash L \xrightarrow{\alpha} p' \backslash L}\ \alpha \notin L^\circ
$$

Table 4: Operational semantics of CCS.

Definition 3.4 (First actions)**.**

$$
\begin{aligned}
\mathcal{F}(nil) &= \varnothing; \\
\mathcal{F}(\alpha.p) &= \{\alpha\}; \\
\mathcal{F}(p+q) &= \mathcal{F}(p) \cup \mathcal{F}(q); \\
\mathcal{F}(p \backslash L) &= \mathcal{F}(p) - L^\circ; \\
\mathcal{F}(x) &= \mathcal{F}(p) \quad \textit{if } x \overset{\text{def}}{=} p; \\
\mathcal{F}(p[f]) &= \{\, f(\alpha) \mid \alpha \in \mathcal{F}(p)\, \}; \\
\mathcal{F}(p \,|\, q) &= \begin{cases} \mathcal{F}(p) \cup \mathcal{F}(q) \cup \{\tau\} & \textit{if } \exists \alpha \in \mathcal{F}(p), \overline{\alpha} \in \mathcal{F}(q), \\ \mathcal{F}(p) \cup \mathcal{F}(q) & \textit{otherwise.} \end{cases}
\end{aligned}
$$

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° contains α. If both α and $\overline{\alpha}$ appear in a process, it is reasonable to assume that they are communication actions.*

In this paper we use CCS without the relabelling 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. $\sigma(p) = \{\, (p', \alpha) \mid p \xrightarrow{\alpha} p' \,\} = \{(p_1, \alpha_1), \ldots, (p_n, \alpha_n)\}$.

[1]The sort of a CCS process p is the alphabet of p. For the precise definition see [22].

$$\begin{tikzcd}
 & & & & & \langle p, q, \top, \lambda\rangle \arrow[dlll] \arrow[drrr] & & & & & & \\
 & & \langle p, q, \bot, 1\rangle \arrow[dl] \arrow[drr] & & & & & & \langle p, q, \bot, 2\rangle \arrow[dl] \arrow[drr] & & & \\
 & \langle nil, q, b, 2\rangle \arrow[d] & & & \langle k.d.f.nil, q, c, 2\rangle \arrow[d] & & & \langle p, c.nil, b, 1\rangle \arrow[d] & & & \langle p, k.d.nil, c, 1\rangle \arrow[d] & \\
 & \langle c.nil, nil, \top, \lambda\rangle \arrow[dl] \arrow[dr] & & & \langle k.d.f.nil, k.d.nil, \top, \lambda\rangle \arrow[dl] \arrow[dr] & & & \langle nil, c.nil, \top, \lambda\rangle \arrow[dl] \arrow[dr] & & & \langle k.d.f.nil, k.d.nil, \top, \lambda\rangle \arrow[dl] \arrow[dr] & \\
\cdots & & \cdots & \cdots & & \cdots & \cdots & & \cdots & \cdots & & \cdots
\end{tikzcd}$$

Figure 2: The AND/OR structure $T(p, q)$.

Example 3.1. *Consider the following CCS processes:*

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

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

The AND/OR structure $T(p, q)$ generated starting from $\langle p, q, \top, \lambda\rangle$, using the operators of Table 3, is sketched in Fig. 2. For simplicity, only the first four levels have been shown in detail.

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, such function is called $\widehat{h}$ and is aimed at working on the AND/OR structure dual to that built by the operators of Table 3 and then at looking for a state containing two not bisimilar processes.

Definition 3.5 ($\widehat{h}(\langle p, q, \gamma, u\rangle)$: 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 four arguments, namely $\widehat{h}_x(p, q, A_p, A_q)$, where $x \in \{\top, \bot\} \cup Act$, $A_p, A_q \subseteq \mathcal{V}$. The three functions are defined in Table 5 and

Table 6, while

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

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 non negative value with each node n of the structure, called $\widehat{h}$-value of n: roughly speaking, that value approximates the number of arcs of the structure that must be crossed to establish that p and q are not bisimilar. It is worth noting that $\widehat{h}$ is easy to be computed, since it depends only on the syntax of the processes and not on their semantics.

The auxiliary functions are parametric with respect to the restriction environments A_p and A_q, which contain the set of actions on which some restriction holds for the processes p and q, respectively. The functions are initially applied with $A_p = A_q = \varnothing$. The current environment A_p is modified when the function is applied to $p\backslash L$: in this case the actions in L° are added to A_p.

The first auxiliary function is $\widehat{h}_{\top}$ that, applied to the node n, holds the minimum value between the values computed for the successors of n plus 1. Since $\widehat{h}$ must be admissible, we choose the minimum between such two values, while 1 is the cost of each arc connecting n to its successors.

The second and third auxiliary functions are inductively defined on the syntactic structure of p: both of them are aiming at reaching a derivative of p of the form $\alpha.p'$. When a derivative of p of the form $\alpha.p'$ is found, $\widehat{h}_{\perp}$ switches the control to the function $\widehat{h}_{\alpha}$; in the other cases it keeps the control while analyzing the type of the operator at the top level of p. The function halts without calling $\widehat{h}_{\alpha}$ with value 0 when a derivative is found that might not be able to perform any action; on the other hand, the returned value is ∞ when both processes have reached a derivative with no possible further move.

We analyse the function $\widehat{h}_{\perp}$ more in detail:

- **Rule R1** applies to a process $p = \mathit{nil}$ and returns ∞ if the process q too is able to perform no move (i.e. the set of first actions of q decreased with the actions belonging to a restricted environment is equal to the empty set), in fact, in this case, the processes are surely bisimilar; otherwise the rule returns 0 since a possible action of q will be not matched by p.

- **Rule R2** applies when $p = \alpha.p'$ and optimistically returns 0 if α is restricted by A_p; otherwise it passes the control to $\widehat{h}_{\alpha}$ trying to check whether q is able to perform the same action; the final value computed by this call is increased of 1, since a possible successor node of p exists for which an arc with cost equal to 1 will be added to the AND/OR structure.

- **Rule R3** applies when $p = p_1 + p_2$. Two cases are possible. When q is not of the form $q = q_1 + q_2$, $\widehat{h}_\perp$ is again applied to q and each sub-component $p_i, i \in \{1, 2\}$ of p, trying to obtain derivative of the form $p_i = \alpha.p'$; the minimum number of the computed values is returned as value of $\widehat{h}_\perp$ applied to p and q. When q too is of the form $q = q_1 + q_2$, the minimum is required among all the combinations of the sub-processes of p and q. To retrieve the *correct* values computed by such successive invocations of $\widehat{h}_\perp$, the operator *cross* is used that takes into account only the values supplied by the *correct* combinations. Consider, for example, $p = p_1 + p_2$ and $q = q_1 + q_2$ with $p_1 = a.c.nil, p_2 = b.d.nil$ and $q_1 = a.c.nil, q_2 = b.d.nil$. It is necessary to compare p_1 against both q_1 and q_2, but, if p_1 is discovered bisimilar to one of two's, for example q_1, the result of the comparison of p_1 and q_2 must not be considered.

- **Rule R4** applies to $p = p_1|p_2$ and examines the form of p_1 and p_2: only if both the sub-processes are of the form $\alpha_i.r_i$, $i \in \{1, 2\}$, it is possible to call $\widehat{h}_{\alpha_j}$ on q and $r_j|\alpha_k.r_k$ (i.e. the derivative of p after the execution of α_j). If both α_i are possible, both values returned by $\widehat{h}_{\alpha_j}$ are considered and the minimum is assigned as value of $\widehat{h}_\perp$, while 1 is the cost of the first action. It is also possible that α_i be a communication between p_1 and p_2 (recall the Remark II.1), in this case its cost is again 1, since the action τ must be executed, but there is only one derivative of p to be considered.

- **Rule R5** applies to $p = p'\backslash L$ and simply adds the set of actions L° to A_p. So, when considering any sub-process r of p, its environment takes account of the union of all the restriction contexts containing that occurrence of r. For example, if

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

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

- **Rule R6** applies to $p = x$ and always returns 0; this rule can be refined by further investigating the structure of the process bound with the constant x; the way in which this refinement could be defined can be easily understood from the previous rules, we do not deepen here the definition of Rule R5 since we are mainly interested into the approach.

The third auxiliary function behaves as $\widehat{h}_\perp$ except that the occurrence of the particular action α_i is required in the derivatives of the form $\alpha_i.r_i$ in **R2** and **R4**, and not simply that such derivative exists. Moreover, $\widehat{h}_\alpha$ passes the control to $\widehat{h}_\top$ when the processes can perform the required action in **R2** and **R4**. Also **R1** is different since it is required that the first process be able to perform α (that is the action that the other process has previously performed).

$$\widehat{h}_{\top}(p,q,A_p,A_q) = 1 + \min(\widehat{h}_{\perp}(p,q,A_p,A_q), \widehat{h}_{\perp}(q,p,A_p,A_q))$$

$$\widehat{h}_{\perp}(nil,q,A_p,A_q) = \begin{cases} \infty & \text{if } \mathcal{F}(q) - A_q = \varnothing, \\ 0 & \text{otherwise;} \end{cases} \qquad \textbf{R1.}$$

$$\widehat{h}_{\perp}(\alpha.p',q,A_p,A_q) = \begin{cases} 1 + \widehat{h}_{\alpha}(q,p',A_q,A_p) & \text{if } \alpha \notin A_p, \\ 0 & \text{otherwise;} \end{cases} \qquad \textbf{R2.}$$

$$\widehat{h}_{\perp}(p_1+p_2,q,A_p,A_q) = \begin{cases} \mathit{cross}\{\widehat{h}_{\perp}(p_i,q_j,A_p,A_q)\}_{1\le i,j\le 2} & \text{if } q = q_1+q_2, \\ \min(\widehat{h}_{\perp}(p_1,q,A_p,A_q), \widehat{h}_{\perp}(p_2,q,A_p,A_q)) & \text{otherwise;} \end{cases} \qquad \textbf{R3.}$$

$$\widehat{h}_{\perp}(p_1 \,|\, p_2,q,A_p,A_q) = \begin{cases} 1 + \min(\widehat{h}_{\alpha_1}(q, r_1 \,|\, \alpha_2.r_2, A_q, A_p), \widehat{h}_{\alpha_2}(q, \alpha_1.r_1 \,|\, r_2, A_q, A_p) \\ \qquad \text{if } p_1|p_2 = \alpha_1.r_1 \,|\, \alpha_2.r_2,\ \alpha_1 \notin A_p,\ \alpha_2 \notin A_p, \\ 1 + \widehat{h}_{\tau}(q, r_1 \,|\, r_2, A_q, A_p) \qquad \text{if } p_1|p_2 = \alpha_1.r_1 \,|\, \overline{\alpha}_1.r_2, \\ 1 + \widehat{h}_{\alpha_1}(q, r_1 \,|\, \alpha_2.r_2, A_q, A_p) \\ \qquad \text{if } p_1|p_2 = \alpha_1.r_1 \,|\, \alpha_2.r_2,\ \alpha_1 \notin A_p,\ \alpha_2 \in A_p, \\ 1 + \widehat{h}_{\alpha_2}(q, \alpha_1.r_1 \,|\, r_2, A_q, A_p) \\ \qquad \text{if } p_1|p_2 = \alpha_1.r_1 \,|\, \alpha_2.r_2,\ \alpha_1 \in A_p,\ \alpha_2 \notin A_p, \\ 0 \qquad \text{otherwise;} \end{cases} \qquad \textbf{R4.}$$

$$\widehat{h}_{\perp}(p' \backslash L,q,A_p,A_q) = \begin{cases} \widehat{h}_{\perp}(p',q,A_p \cup L^{\circ},A_q) \end{cases} \qquad \textbf{R5.}$$

$$\widehat{h}_{\perp}(x,q,A_p,A_q) = 0 \qquad \textbf{R6.}$$

where

$$\mathit{cross}\{x_{ij}\}_{1\le i,j\le 2} = \begin{cases} \infty & \text{if } x_{11} = x_{22} = \infty \text{ or } x_{12} = x_{21} = \infty, \\ \min\{x_{ij}\}_{1\le i,j\le 2} & \text{otherwise.} \end{cases}$$

Table 5: The $\widehat{h}_{\perp}$ function and the $\widehat{h}_{\top}$ function for strong equivalence.

$$\widehat{h}_\alpha(nil, q, A_p, A_q) = 0 \qquad \textbf{R1.}$$

$$\widehat{h}_\alpha(\beta.p', q, A_p, A_q) = \begin{cases} 1 + \widehat{h}_\top(p', q, A_p, A_q) & \text{if } \beta = \alpha,\ \beta \notin A_p \\ 0 & \text{otherwise;} \end{cases} \qquad \textbf{R2.}$$

$$\widehat{h}_\alpha(p_1 + p_2, q, A_p, A_q) = \begin{cases} cross\{\widehat{h}_\alpha(p_i, q_j, A_p, A_q)\}_{1\le i,j\le 2} & \text{if } q = q_1 + q_2, \\ \min(\widehat{h}_\alpha(p_1, q, A_p, A_q), \widehat{h}_\alpha(p_2, q, A_p, A_q)) & \text{otherwise;} \end{cases} \qquad \textbf{R3.}$$

$$\widehat{h}_\alpha(p_1 \,|\, p_2, q, A_p, A_q) = \begin{cases} 2 + \widehat{h}_\top(r_1|\alpha.r_2, q, A_p, A_q) + \widehat{h}_\top(\alpha.r_1|r_2, q, A_p, A_q) \\ \qquad \text{if } p_1|p_2 = \alpha.r_1|\alpha.r_2,\ \alpha \notin A_p, \\ 1 + \widehat{h}_\top(r_1|r_2, q, A_p, A_q) \qquad \text{if } p_1|p_2 = \alpha_1.r_1|\overline{\alpha}_1.r_2,\ \alpha = \tau, \\ 1 + \widehat{h}_\top(\alpha_1.r_1|r_2, q, A_p, A_q) \\ \qquad \text{if } p_1|p_2 = \alpha_1.r_1|\alpha.r_2,\ \alpha_1 \neq \alpha,\ \alpha \notin A_p, \\ 1 + \widehat{h}_\top(r_1|\alpha_2.r_2, q, A_p, A_q) \\ \qquad \text{if } p_1|p_2 = \alpha.r_1|\alpha_2.r_2,\ \alpha \notin A_p,\ \alpha_2 \neq \alpha, \\ 0 \qquad \text{otherwise;} \end{cases} \qquad \textbf{R4.}$$

$$\widehat{h}_\alpha(p'\backslash L, q, A_p, A_q) = \widehat{h}_\alpha(p', q, A_p \cup L^\circ, A_q); \qquad \textbf{R5.}$$

$$\widehat{h}_\alpha(x, q, \alpha, A_p, A_q) = 0 \qquad \textbf{R6.}$$

Table 6: The $\widehat{h}_\alpha$ function for strong equivalence.

Consider the following two CCS processes

$$\begin{aligned} p =& (c.k.nil \,|\, \overline{c}.nil)\backslash\{c\} \\ q =& (d.h.nil \,|\, \overline{d}.nil)\backslash\{d\} \end{aligned}$$

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

$$\begin{aligned} &\widehat{h}_\bot(p, q, \varnothing, \varnothing) \\ &\quad = \widehat{h}_\bot(c.k.nil \,|\, \overline{c}.nil, q, \{c, \overline{c}\}, \varnothing) && \text{(Dfn. } \widehat{h}_\bot\text{: } \textbf{Rule R5}) \\ &\quad = 1 + \widehat{h}_\tau(q, k.nil \,|\, nil, \varnothing, \{c, \overline{c}\}) && \text{(Dfn. } \widehat{h}_\bot\text{: } \textbf{Rule R4}) \\ &\quad = 1 + \widehat{h}_\tau(d.h.nil \,|\, \overline{d}.nil, k.nil \,|\, nil, \{d, \overline{d}\}, \{c, \overline{c}\}) && \text{(Dfn. } \widehat{h}_\alpha\text{: } \textbf{Rule R5}) \\ &\quad = 1 + 1 + \widehat{h}_\top(h.nil \,|\, nil, k.nil \,|\, nil, \{d, \overline{d}\}, \{c, \overline{c}\}) && \text{(Dfn. } \widehat{h}_\bot\text{: } \textbf{Rule R4}) \\ &\quad = 2 + 1 + \min(n_1, n_2) && \text{(Dfn. } \widehat{h}_\top) \end{aligned}$$

where

$$\begin{aligned} n_1 &= \widehat{h}_\bot(h.nil \,|\, nil, k.nil \,|\, nil, \{d, \overline{d}\}, \{c, \overline{c}\}); \quad \text{and} \\ n_2 &= \widehat{h}_\bot(k.nil \,|\, nil, h.nil \,|\, nil, \{c, \overline{c}\}, \{d, \overline{d}\}). \end{aligned}$$

Finally,

$$\begin{aligned} n_1 &= \widehat{h}_\bot(h.nil \,|\, nil, k.nil \,|\, nil, \{d, \overline{d}\}, \{c, \overline{c}\}) = 0, && \text{(Dfn. } \widehat{h}_\bot\text{: } \textbf{Rule R4}) \\ n_2 &= \widehat{h}_\bot(k.nil \,|\, nil, h.nil \,|\, nil, \{c, \overline{c}\}, \{d, \overline{d}\}) = 0. && \text{(Dfn. } \widehat{h}_\bot\text{: } \textbf{Rule R4}) \end{aligned}$$

Since $\widehat{h}_\bot(p, q, \varnothing, \varnothing) = 3$, 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 a bigger number of actions be required. Consider, for example, the node $n = \langle p, q, \bot, 1\rangle$, where $p = \big((a.nil + d.b.nil) \,|\, \overline{a}.nil\big)\backslash\{a\}$ and $q = \big((a.nil + d.k.nil) \,|\, \overline{a}.nil\big)\backslash\{a\}$. It holds that $\widehat{h}(n) = 0$, but, to reach two non bisimilar sub-processes of p and q at least the two d actions must be performed.

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

Theorem 3.3. *Let $D = (N, R)$ be a solution graph. It holds that*

$$\forall n \in N,\ \widehat{h}(n) \leq h(n)$$

where $h(n)$ is the actual cost of n.

$$\begin{tikzcd}[column sep=tiny]
 & & & \langle p, q, \top, \lambda\rangle \arrow[dll, "\widehat{h}=0"'] \arrow[drr, thick, "\widehat{h}=0"] & & & \\
 & \langle p, q, \bot, 1\rangle \arrow[dl, "\widehat{h}=0"'] \arrow[dr, "\widehat{h}=0"] & & & & \langle p, q, \bot, 2\rangle \arrow[dl, thick, "\widehat{h}=0"'] \arrow[dr, "\widehat{h}=0"] & \\
\langle nil, q, b, 2\rangle \arrow[d, "\widehat{h}=1"'] & & \langle k.d.f.nil, q, c, 2\rangle \arrow[d, "\widehat{h}=7"] & & \langle p, c.nil, b, 1\rangle \arrow[d, thick, "\widehat{h}=1"'] & & \langle p, k.d.nil, c, 1\rangle \arrow[d, "\widehat{h}=7"] \\
\langle c.nil, nil, \top, \lambda\rangle & & \langle k.d.f.nil, k.d.nil, \top, \lambda\rangle & & \langle nil, c.nil, \top, \lambda\rangle \arrow[dl] \arrow[dr, thick, "\widehat{h}=1"] & & \langle k.d.f.nil, k.d.nil, \top, \lambda\rangle \\
 & & & \langle nil, c.nil, \bot, 1\rangle & & \langle nil, c.nil, \bot, 2\rangle \arrow[d, thick] & \\
 & & & & & \langle nil, nil, c, 1\rangle &
\end{tikzcd}$$

Figure 3: An example.

Proof. Let $n = \langle p, q, \gamma, k\rangle$. First we prove that $\widehat{h}_\bot(p, q, A_p, A_q) \leq h(n)$, with $Ap \subseteq \mathcal{V}$, $A_q \subseteq \mathcal{V}$, $\gamma = \bot$ and $k = 1$. Therefore $n = \langle p, q, \bot, 1\rangle$. The proof is made by induction on the structure of the process p. The admissibility of other auxiliary heuristic functions can be proved similarly. 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 p_1 and p_2.

$\underline{p = \alpha.p_1}$. According to the definition of the auxiliary heuristic function $\widehat{h}_\bot$ (see Rule R2 in Table 5), two cases exist: $\alpha \notin A_p$ and $\alpha \in A_p$. In the first case, $\widehat{h}_\bot(n) = 1 + \widehat{h}_\alpha(q, p_1, A_q, A_p)$. Using the operators of Table 3, it holds that $n = \langle p, q, \bot, 1\rangle \longrightarrow \langle p_1, q, \alpha, 2\rangle$, thus $h(n) = 1 + h(\langle p_1, q, \alpha, 2\rangle)$ (according to Definition 2.4). The thesis follows by inductive hypothesis, that is $\widehat{h}_\alpha(\langle q, p, \alpha, 2\rangle) \leq h(\langle p_1, q, \alpha, 2\rangle))$. Remember that by Definition 3.5, $\widehat{h}(\langle q, p, \alpha, 2\rangle) = \widehat{h}_\alpha(q, p, \varnothing, \varnothing)$. In the second case, α is a restricted action and $\widehat{h}_\bot(n) = 0$. Consider now the actual moves of p. If p can perform an action, then $h(n) \geq 1$, which is greater than zero: the theorem is obviously true. Otherwise, if p can perform no actions, then $h(n) = 0$, which is equal to $\widehat{h}_\bot(n)$: also in this case the theorem is true.

$\underline{p = r_1 | r_2}$. According to the definition of the auxiliary heuristic function $\widehat{h}_\bot$ (see Rule R4 in Table 5), five cases may occur.

1. suppose that $r_1 = \alpha_1.p_1$ and $r_2 = \alpha_2.p_2$ and both actions α_1, α_2 are not

restricted actions. It holds that

$$\widehat{h}_\perp(n) = 1 + \min(\widehat{h}_{\alpha_1}(q, p_1|\alpha_2.p_2, A_q, A_p), \widehat{h}_{\alpha_2}(q, \alpha_1.r_1|r_2, A_q, A_p))$$

Consider now the possible moves of n. Recall the assumption made in Section 3.3 (Remark 3.1). With this assumption it is not possible to apply, at the same time, both **Com** rule and **Par** rule (see Table 4), involving a same action. Therefore, using the operators of Table 3, it holds that $n = \langle p, q, \perp, 1\rangle \longrightarrow \langle p_1|\alpha_2.p_2, q, \alpha_1, 2\rangle$ and $n = \langle p, q, \perp, 1\rangle \longrightarrow \langle \alpha_1.p_1|p_2, q, \alpha_2, 2\rangle$. Now, node n is an OR node, thus D must contain at least one successor. Therefore, either $h(n) = 1 + h(\langle p_1|\alpha_2.p_2, q, \alpha_1, 2\rangle)$ or $h(n) = 1 + h(\langle \alpha_1.p_1|p_2, q, \alpha_2, 2\rangle)$ (according to Definition 2.4). Since $\widehat{h}_\perp(n)$ chooses the minimum of the two values, the thesis follows by inductive hypothesis.

2. Now, suppose that $r_1 = \alpha_1.p_1$ and $r_2 = \alpha_2.p_2$, both actions α_1, α_2 are restricted actions and are complementary actions, i.e. $\alpha_1 = \overline{\alpha}_2$. It holds that

$$\widehat{h}_\perp(n) = 1 + \widehat{h}_\tau(q, p_1|p_2, A_q, A_p)$$

Using the operators of Table 3, $n = \langle p, q, \perp, 1\rangle \longrightarrow \langle p_1|p_2, q, \tau, 2\rangle$. The successors of n is in D, therefore, $h(n) = 1 + h(\langle p_1|p_2, q, \tau, 2\rangle)$ (according to definition of h 2.4). The thesis follows by inductive hypothesis.

3. All the other cases can be proved in a similar way.

□

Example 3.2. *Recall the CCS processes p and q of Example 3.1 and let us apply our method to check whether p and q are strong bisimilar. Using the CWB-NC the result is:*

```
cwb-nc> eq -S bisim p q
Building automaton...
States: 9
Transitions: 10
Done building automaton.
FALSE...
p satisfies:
        [b][c]ff
q does not.
```

Thus, the complexity in space of equivalence checking is heavily influenced by the size of $\mathcal{S}(p)$ and $\mathcal{S}(q)$. It holds that both the transition systems have five states. Applying our approach, the dual of the AND/OR structure of Fig. 2 is constructed. As shown in Fig. 3, we stop the generation of states of the standard transition systems of both processes when we can deduce that they are not bisimilar. Moreover, since our heuristic is admissible as stated by Theorem 3.3, we find the minimal-cost solution graph leading to two not bisimilar states. We *point out that finding the minimal-cost solution graph is useful since that graph is examined to pinpoint the source of the error. Little graphs facilitate the comprehension of the fault. In this simple example, since all the AND nodes have at least one successor, the solution graph is actually a path (the bold line in figure) of cost equal to five. In Fig. 3, the* $\widehat{h}$ *value of each node is reported. For example, it holds that* $\widehat{h}(\langle nil, c.nil, \top, \lambda\rangle = 1$. *Note that, the node* $\widehat{h}(\langle nil, c.nil, \bot, 1\rangle$ *is a nonterminal leaf node, while the node* $\widehat{h}(\langle nil, nil, c, 1\rangle$ *is a terminal one. With our approach only six states are generated: p, q, c.nil, k.d.f.nil, k.d.nil and nil, obtaining a reduction of three states. With this little example we have provided evidence of the reduction of the state space that may result when applying our methodology.*

3.5. The heuristic function for checking weak equivalence

In this section we slightly modify the heuristic function defined in the previous section to manage the weak equivalence. The three auxiliary functions $\widehat{h}_\top$, $\widehat{h}_\bot$ and $\widehat{h}_\alpha$ are shown in Tables 7 and 8. The novelty of these new functions is that the silent action τ is skipped when counting the number of arcs of the AND/OR structure that must be crossed to establish that p and q are not weak bisimilar. The AND/OR structure to check weak equivalence is built using the operators of Table 3 where the σ function is rephrased using the transition relation $\Longrightarrow$ i.e., $\sigma(p) = \{\, (p', \alpha) \mid p \overset{\alpha}{\Longrightarrow} p' \,\} = \{(p_1, \alpha_1), \ldots, (p_n, \alpha_n)\}$. Skipping the τ action is obtained by modifying the rules R2, R3 and R4 of Table 7. Similarly for the rules in Table 8. Note that $\widehat{h}_\alpha$ is never called with $\alpha = \tau$, since the τ action is irrelevant for checking equivalence. Theorem 3.3 stating the admissibility of the heuristic function for strong equivalence still holds also for the new heuristic function for checking weak equivalence. The proof easily follows by Theorem 3.3 since the new auxiliary functions never increments the $\widehat{h}$-value of each node.

3.6. Infinite CCS processes

In many approaches, in order to disprove that a CCS process p is equivalent to another process q, the whole transition systems for p and for q are built and then almost all the existing verification environments (see for example [27, 28]) are based on an internal finite state representation of the processes. For this reason, a very common requirement in these environments is the following:

> *the parallel and relabelling operators are allowed inside the body of a process name as long as no process name occurs in the arguments* [29].

Obviously, these approaches cannot be used when the transition systems of the processes to check are infinite. Instead our method can be applied also in these situations.

Consider the following CCS processes:

$$x \overset{\text{def}}{=} (a.b.nil \,|\, b.c.x) + c.d.nil$$
$$y \overset{\text{def}}{=} (a.b.nil \,|\, b.c.y) + c.e.nil$$

$$\widehat{h}_\top(p,q,A_p,A_q) = 1+\min(\widehat{h}_\perp(p,q,A_p,A_q),\widehat{h}_\perp(q,p,A_p,A_q))$$

$$\widehat{h}_\perp(nil,q,A_p,A_q) = \begin{cases} \infty & \text{if } \mathcal{F}(q)-A_q=\varnothing, \\ 0 & \text{otherwise;} \end{cases} \qquad \textbf{R1.}$$

$$\widehat{h}_\perp(\alpha.p',q,A_p,A_q) = \begin{cases} 1+\widehat{h}_\alpha(q,p',A_q,A_p) & \text{if } \alpha\notin A_p\cup\{\tau\}, \\ \widehat{h}_\perp(p',q,A_p,A_q) & \text{if } \alpha=\tau, \\ 0 & \text{otherwise;} \end{cases} \qquad \textbf{R2.}$$

$$\widehat{h}_\perp(p_1+p_2,q,A_p,A_q) =$$
$$\begin{cases} 0 & \text{if } \tau\in\mathcal{F}(p_1+p_2)\cup\mathcal{F}(q), \\ \mathit{cross}\{\widehat{h}_\perp(p_i,q_j,A_p,A_q)\}_{1\le i,j\le 2} & \text{if } q=q_1+q_2 \text{ and} \\ & \tau\notin\mathcal{F}(p_1+p_2)\cup\mathcal{F}(q), \\ \min(\widehat{h}_\perp(p_1,q,A_p,A_q),\widehat{h}_\perp(p_2,q,A_p,A_q)) & \text{otherwise;} \end{cases} \qquad \textbf{R3.}$$

$$\widehat{h}_\perp(p_1\,|\,p_2,q,A_p,A_q) =$$
$$\begin{cases} 1+\min(\widehat{h}_{\alpha_1}(q,r_1\,|\,\alpha_2.r_2,A_q,A_p),\widehat{h}_{\alpha_2}(q,\alpha_1.r_1\,|\,r_2,A_q,A_p) \\ \qquad \text{if } p_1|p_2=\alpha_1.r_1\,|\,\alpha_2.r_2,\ \alpha_1\notin A_p\cup\{\tau\},\ \alpha_2\notin A_p\cup\{\tau\}, \\ \widehat{h}_\perp(r_1\,|\,r_2,q,A_p,A_q) \qquad \text{if } p_1|p_2=\alpha_1.r_1\,|\,\overline{\alpha}_1.r_2, \\ 1+\widehat{h}_{\alpha_1}(q,r_1\,|\,\alpha_2.r_2,A_q,A_p) \\ \qquad \text{if } p_1|p_2=\alpha_1.r_1\,|\,\alpha_2.r_2,\ \alpha_1\notin A_p\cup\{\tau\},\ \alpha_2\in A_p, \\ 1+\widehat{h}_{\alpha_2}(q,\alpha_1.r_1\,|\,r_2,A_q,A_p) \\ \qquad \text{if } p_1|p_2=\alpha_1.r_1\,|\,\alpha_2.r_2,\ \alpha_1\in A_p,\ \alpha_2\notin A_p\cup\{\tau\}, \\ 0 \qquad \text{otherwise;} \end{cases} \qquad \textbf{R4.}$$

$$\widehat{h}_\perp(p'\backslash L,q,A_p,A_q) = \Big\{\widehat{h}_\perp(p',q,A_p\cup L^\circ,A_q) \qquad \textbf{R5.}$$

$$\widehat{h}_\perp(x,q,A_p,A_q) = 0 \qquad \textbf{R6.}$$

where

$$\mathit{cross}\{x_{ij}\}_{1\le i,j\le 2} = \begin{cases} \infty & \text{if } x_{11}=x_{22}=\infty \text{ or } x_{12}=x_{21}=\infty, \\ \min\{x_{ij}\}_{1\le i,j\le 2} & \text{otherwise.} \end{cases}$$

Table 7: The $\widehat{h}_\perp$ function and the $\widehat{h}_\top$ function for weak equivalence.

$$\widehat{h}_\alpha(nil, q, A_p, A_q) = 0 \qquad \textbf{R1.}$$

$$\widehat{h}_\alpha(\beta.p', q, A_p, A_q) = \begin{cases} 1 + \widehat{h}_\top(p', q, A_p, A_q) & \text{if } \beta = \alpha,\ \beta \notin A_p \cup \{\tau\}, \\ \widehat{h}_\alpha(p', q, A_p, A_q) & \text{if } \beta = \tau, \\ 0 & \text{otherwise;} \end{cases} \qquad \textbf{R2.}$$

$$\widehat{h}_\alpha(p_1 + p_2, q, A_p, A_q) =$$
$$\begin{cases} 0 & \text{if } \tau \in \mathcal{F}(p_1 + p_2) \cup \mathcal{F}(q), \\ cross\{\widehat{h}_\alpha(p_i, q_j, A_p, A_q)\}_{1 \le i,j \le 2} & \text{if } q = q_1 + q_2 \text{ and} \\ & \tau \notin \mathcal{F}(p_1 + p_2) \cup \mathcal{F}(q), \\ \min(\widehat{h}_\alpha(p_1, q, A_p, A_q), \widehat{h}_\alpha(p_2, q, A_p, A_q)) & \text{otherwise;} \end{cases} \qquad \textbf{R3.}$$

$$\widehat{h}_\alpha(p_1 \,|\, p_2, q, A_p, A_q) =$$
$$\begin{cases} 2 + \widehat{h}_\top(r_1|\alpha.r_2, q, A_p, A_q) + \widehat{h}_\top(\alpha.r_1|r_2, q, A_p, A_q) \\ \qquad \text{if } p_1|p_2 = \alpha.r_1|\alpha.r_2,\ \alpha \notin A_p \cup \{\tau\}, \\ 1 + \widehat{h}_\top(\alpha_1.r_1|r_2, q, A_p, A_q) \\ \qquad \text{if } p_1|p_2 = \alpha_1.r_1|\alpha.r_2,\ \alpha_1 \neq \alpha,\ \alpha \notin A_p \cup \{\tau\}, \\ 1 + \widehat{h}_\top(r_1|\alpha_2.r_2, q, A_p, A_q) \\ \qquad \text{if } p_1|p_2 = \alpha.r_1|\alpha_2.r_2,\ \alpha \notin A_p \cup \{\tau\},\ \alpha_2 \neq \alpha, \\ 0 \qquad \text{otherwise;} \end{cases} \qquad \textbf{R4.}$$

$$\widehat{h}_\alpha(p' \backslash L, q, A_p, A_q) = \widehat{h}_\alpha(p', q, A_p \cup L^\circ, A_q); \qquad \textbf{R5.}$$

$$\widehat{h}_\alpha(x, q, \alpha, A_p, A_q) = 0 \qquad \textbf{R6.}$$

Table 8: The $\widehat{h}_\alpha$ function for weak equivalence.

It holds that $\mathcal{S}(x)$ and $\mathcal{S}(y)$ are infinite: we choice to move first the second branch of the $+$ operator both in x and y, since the value returned by the heuristic function in these cases is lesser than that returned by the other branches, and, after having generated 17 states, a counterexample of cost equal to 5 is produced. Other approaches for verification of infinite systems can be found in [30, 31].

4. Experimental Results

In this section we present and discuss our experience with using the tool to check both strong and weak equivalence of several well-known processes. Our aim is to evaluate the performances of the approach presented in Section 3 and compare it against both the CWB-NC and the CADP tool [32]. 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 several instances of the dining philosophers example. In the first solution, when a philosopher gets hungry, he can, without any control, pick up his left fork first, then his right one; if he can eat, then puts the forks down in the same order that he had picked them up. In the second solution, shown in [33], 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 solutions are not strongly equivalent.

Table 9 shows the number of generated nodes resulting from the CWB-NC and using our approach, 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 solutions of the dining philosophers, namely p and q. Finally, the last column indicates the cost of the counterexample. We may see the significant reduction of the state-space that results when applying our heuristic function. 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 10 our approach is compared with the CADP tool. Again we consider the dining philosophers example and we check the strong equivalence. Since CADP tool uses LOTOS [34] as specification language, all the CCS specifications have been equivalently translated in LOTOS. Moreover, the memory usage is considered instead of the generated states. The table shows both the memory consumption (KB) and the runtime performance (sec) of our approach and of the CADP tool.

Both the classic CWB-NC environment and of the CADP environment use efficient algorithms, and, in particular, in the CADP, great attention is devoted to implementation efficiency issues. Note that the execution time of our tool for 5 philosophers was about 11 seconds. This is quite high, considering the scale of the process: anyway, we found that it is mostly due to the overhead of the use of S2 algorithm. Also the authors of S2 have found that the algorithm

n	our approach	CWB-NC	dimension		state space	cost of
	gen	gen	p	q	reduction	counterexample
2	60	156	74	82	62 %	14
3	163	1072	639	433	85 %	14
4	348	7212	5510	1702	95 %	14
5	664	53815	47496	6319	99 %	14
6	1170	-	-	-	-	14
8	3070	-	-	-	-	14
12	13770	-	-	-	-	14
16	43676	-	-	-	-	14
20	111700	-	-	-	-	14

Table 9: Results on the dining philosophers (CWB-NC - strong equivalence)

n	our approach	CADP	our approach	CADP
	memory	memory	time	time
2	10000	51600	0.04	0.60
3	13856	57840	0.20	0.66
4	27648	62672	2.13	0.77
5	64640	67824	11.5	1.01

Table 10: Results on the dining philosophers (CADP - strong equivalence)

is much slower than the competing algorithm $\mathrm{CFC}_{\mathrm{REV}*}$, 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 more complex behavior. We recall that our aim was checking equivalences trying to save as memory space as possible and to obtain the minimum counterexample in the case of two non equivalent processes; thus we reasonably choose S2 as search algorithm and consequently defined an admissible heuristic function. However, in general, the choice of saving space cannot be compatible with the needs of obtaining low execution time; this is true when the two processes are similar, but also in the worst cases of non similar processes. This approach has the advantage that a good heuristic function should help to obtain both memory saving (less nodes to explore) and low execution time (less work to do).

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 [35] due two Dekker;
  - q: the CCS specification of the mutual exclusion algorithms [35] due two 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 $p \not\sim q$.

- 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 [36].
  - q: a new specification of a mail system, always produced by Gordon Brebner [36].
  - 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 [37, 38, 39].
  - 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 11 (comparison with CWB-NC) and Table 12 (comparison with CADP), while Table 13 shows the results of our approach against CADP for checking weak equivalence.

From the point of view of our initial goal, with these examples we have provided some experimental evidence of the reduction of the state space that may result when applying our methodology with respect to CWB-NC. Note that in some cases we obtain a reduction more than 85%. However, the tests performed so far show that our approach based on heuristic searches is comparable to the

case study	our approach gen	CWB-NC gen	dimension p	q	state space reduction	cost of counterexample
DEK-PET	39	256	165	91	85 %	17
BUFF	12	17	7	10	29 %	5
XOR	12	18	8	10	33 %	8
MUTUAL	4508	8704	6912	6912	48%	20
MAIL	99	1434	1025	409	93 %	17
BRP	1518	1518	759	759	0 %	strong bisimilar

Table 11: Results for other systems (CWB-NC - strong equivalence)

case study	our approach memory	CADP memory	our approach time	CADP time
DEK-PET	9664	50848	0.00	0.32
BUFF	8528	44784	0.08	0.3
XOR	8592	46464	0.00	0.40
MUTUAL	42528	63392	9.22	0.70
MAIL	10816	53008	0.01	1.44

Table 12: Results for other systems (CADP - strong equivalence)

case study	our approach memory	CADP memory	our approach time	CADP time	weak bisimilar?
DEK-PET	67264	73472	12.45	0.45	no
BUFF	9280	46544	0.00	0.32	yes
XOR	9264	47584	0.00	0.32	yes
MUTUAL	108064	105712	131.11	2.47	yes
MAIL	4513	106960	3.21	0.46	no

Table 13: Results for other systems (CADP - weak equivalence)

CADP tool in terms of memory space reduction. On the contrary, its performance, with particular reference to the scalability of the analysis, is lower. For the time being, our impression is that our technique looks promising for further refinements of the method. For example, we investigated the use of more accurate heuristic functions (and more complex); 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. On the other hand, the exploitation of heuristic searches for checking equivalence is a novel approach, which, in our knowledge, has never used before. As said before and as evidenced by the experiments, the S2 algorithm for detecting processes similarities has lower performance especially due to the Bottom_Up step of Table 2. However, we think that our 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 not 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 not equivalent states. This structure allows the user to understand and navigate through the counterexample better, especially in a setting with highly non-deterministic automata where counterexamples may become rather complicated. For example, consider the following CCS processes:

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

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

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

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

The graphical structure representation returned by our approach is shown in Figure 4.

5. Conclusion and Related Work

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

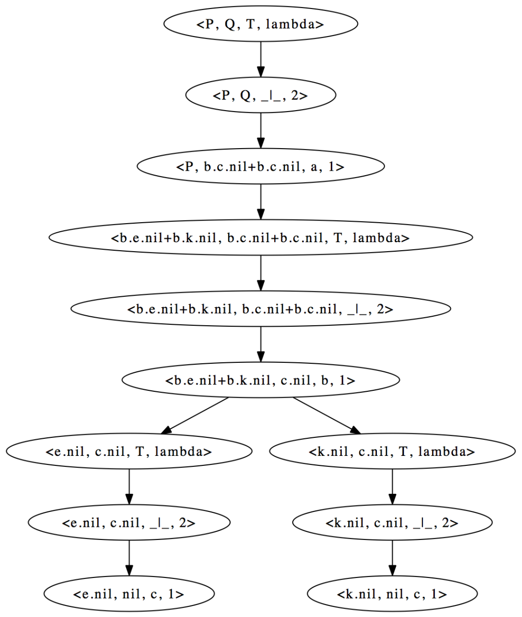

Figure 4: Our counterexample.

- 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, by S2, we can always find minimal graph leading to two not 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 practise 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 the 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 [40]. The other family of algorithms is based on a Cartesian product traversal from the initial state [41, 42]. 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 [43, 44], 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 [45] 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 [46] and by Gentilini, Piazza and Policriti [47]. 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 generalised 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 [48] is one of the original ones. 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 [49] genetic algorithms are used to exploit heuristics for guiding a search in large state spaces towards errors like deadlocks and assertion violations. In [50] heuristics have been used for real-time model checking in UPPAAL. In [51, 52] heuristic search has been combined with on-the-fly techniques, while in [53, 54] with symbolic model checking. Other works, as for example [55, 56], used heuristic search to accelerate finding errors, while in [57] heuristic search is used to accelerate verification.

As a future work we intend apply this approach also for other equivalences, as for example weak equivalence and ρ-equivalence, introduced in [17], 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] E. M. Clarke, E. A. Emerson, Design and synthesis of synchronization skeletons using branching time temporal logic, in: O. Grumberg, H. Veith (Eds.), 25 Years of Model Checking, volume 5000 of *Lecture Notes in Computer Science*, Springer, 2008, pp. 196–215.

[2] J.-P. Queille, J. Sifakis, Specification and verification of concurrent systems in cesar, in: M. Dezani-Ciancaglini, U. Montanari (Eds.), Symposium on Programming, volume 137 of *Lecture Notes in Computer Science*, Springer, 1982, pp. 337–351.

[3] A. N. Parashkevov, J. Yantchev, Arc - a tool for efficient refinement and equivalence checking for csp, in: In IEEE Int. Conf. on Algorithms and Architectures for Parallel Processing ICA3PP '96, pp. 68–75.

[4] A. Bouajjani, J.-C. Fernandez, N. Halbwachs, Minimal model generation, in: [58], pp. 197–203.

[5] S. Graf, B. Steffen, G. Lüttgen, Compositional minimisation of finite state systems using interface specifications, Formal Asp. Comput. 8 (1996) 607–616.

[6] K. L. McMillan, Symbolic model checking, Kluwer, 1993.

[7] C. Jard, T. Jéron, Bounded-memory algorithms for verification on-the-fly, in: [59], pp. 192–202.

[8] S. Edelkamp, V. Schuppan, D. Bosnacki, A. Wijs, A. Fehnker, H. Aljazzar, Survey on directed model checking, in: MoChArt, pp. 65–89.

[9] S. Gradara, A. Santone, M. L. Villani, Using heuristic search for finding deadlocks in concurrent systems, Inf. Comput. 202 (2005) 191–226.

[10] S. Gradara, A. Santone, M. L. Villani, Delfin$^+$: An efficient deadlock detection tool for ccs processes, J. Comput. Syst. Sci. 72 (2006) 1397–1412.

[11] C. Stirling, D. Walker, Local model checking in the modal mu-calculus, Theor. Comput. Sci. 89 (1991) 161–177.

[12] P. Godefroid, Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem, volume 1032 of *Lecture Notes in Computer Science*, Springer, 1996.

[13] D. Peled, All from one, one for all: on model checking using representatives, in: [60], pp. 409–423.

[14] A. Valmari, A stubborn attack on state explosion, Formal Methods in System Design 1 (1992) 297–322.

[15] E. M. Clarke, D. E. Long, K. L. McMillan, Compositional model checking, in: LICS, pp. 353–362.

[16] A. Santone, Automatic verification of concurrent systems using a formula-based compositional approach, Acta Inf. 38 (2002) 531–564.

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

[18] E. M. Clarke, O. Grumberg, D. E. Long, Model checking and abstraction, ACM Trans. Program. Lang. Syst. 16 (1994) 1512–1542.

[19] J. Pearl, Heuristics - intelligent search strategies for computer problem solving, Addison-Wesley series in artificial intelligence, Addison-Wesley, 1984.

[20] A. Mahanti, A. Bagchi, And/or graph heuristic search methods, J. ACM 32 (1985) 28–51.

[21] A. Mahanti, S. Ghose, S. K. Sadhukhan, A framework for searching and/or graphs with cycles, CoRR cs.AI/0305001 (2003).

[22] R. Milner, Communication and concurrency, PHI Series in computer science, Prentice Hall, 1989.

[23] A. Santone, Heuristic for simulation checking, in: Proceedings of the 2011 International Conference on Artificial Intelligence (ICAI 2011), Las Vegas Nevada, USA, CSREA Press, 2011.

[24] P. P. Chakrabarti, Algorithms for searching explicit and/or graphs and their applications to problem reduction search, Artif. Intell. 65 (1994) 329–345.

[25] P. Jiménez, C. Torras, An efficient algorithm for searching implicit and/or graphs with cycles, Artif. Intell. 124 (2000) 1–30.

[26] E. A. Hansen, S. Zilberstein, Lao*: A heuristic search algorithm that finds solutions with loops, Artif. Intell. 129 (2001) 35–62.

[27] A. Bouali, S. Gnesi, S. Larosa, Jack: Just another concurrency kit. the intergration projekt, Bulletin of the EATCS 54 (1994) 207–223.

[28] R. Cleaveland, S. Sims, The ncsu concurrency workbench, in: R. Alur, T. A. Henzinger (Eds.), CAV, volume 1102 of *Lecture Notes in Computer Science*, Springer, 1996, pp. 394–397.

[29] E. Madelaine, D. Vergamini, Finiteness conditions and structural construction of automata for all process algebras, in: [58], pp. 353–363.

[30] M. Dam, Compositional proof systems for model checking infinite state processes, in: I. Lee, S. A. Smolka (Eds.), CONCUR, volume 962 of *Lecture Notes in Computer Science*, Springer, 1995, pp. 12–26.

[31] K. L. McMillan, Verification of infinite state systems by compositional model checking, in: L. Pierre, T. Kropf (Eds.), CHARME, volume 1703 of *Lecture Notes in Computer Science*, Springer, 1999, pp. 219–234.

[32] H. Garavel, F. Lang, R. Mateescu, W. Serwe, Cadp 2010: A toolbox for the construction and analysis of distributed processes, in: TACAS, pp. 372–387.

[33] G. Bruns, A practical technique for process abstraction, in: E. Best (Ed.), CONCUR, volume 715 of *Lecture Notes in Computer Science*, Springer, 1993, pp. 37–49.

[34] T. Bolognesi, E. Brinksma, Introduction to iso specification language lotos, Comp. Networks and ISDN Systems 14 (1987) 25–59.

[35] D. J. Walker, Automated analysis of mutual exclusion algorithms using ccs, Formal Asp. Comput. 1 (1989) 273–292.

[36] G. J. Brebner, A ccs-based investigation of deadlock in a multi-process electronic mail system, Formal Asp. Comput. 5 (1993) 467–478.

[37] J. F. Groote, J. van de Pol, A bounded retransmission protocol for large data packets, in: M. Wirsing, M. Nivat (Eds.), AMAST, volume 1101 of *Lecture Notes in Computer Science*, Springer, 1996, pp. 536–550.

[38] K. Havelund, N. Shankar, Experiments in theorem proving and model checking for protocol verification, in: M.-C. Gaudel, J. Woodcock (Eds.), FME, volume 1051 of *Lecture Notes in Computer Science*, Springer, 1996, pp. 662–681.

[39] L. Helmink, M. P. A. Sellink, F. W. Vaandrager, Proof-checking a data link protocol, in: H. Barendregt, T. Nipkow (Eds.), TYPES, volume 806 of *Lecture Notes in Computer Science*, Springer, 1993, pp. 127–165.

[40] R. Paige, R. E. Tarjan, Three partition refinement algorithms, SIAM J. Comput. 16 (1987) 973–989.

[41] J.-C. Fernandez, L. Mounier, “on the fly“ verification of behavioural equivalences and preorders, in: [59], pp. 181–191.

[42] J. Godskesen, K. Larsen, M. Zeeberg, TAV - tools for automatic verification: users manual, Institute for Electronic Systems, Department of Mathematics and Computer Science, The University of Aalborg, 1989.

[43] J.-C. Fernandez, An implementation of an efficient algorithm for bisimulation equivalence, Sci. Comput. Program. 13 (1989) 219–236.

[44] P. C. Kanellakis, S. A. Smolka, Ccs expressions, finite state processes, and three problems of equivalence, Inf. Comput. 86 (1990) 43–68.

[45] J.-C. Fernandez, A. Kerbrat, L. Mounier, Symbolic equivalence checking, in: [60], pp. 85–96.

[46] D. Bustan, O. Grumberg, Simulation-based minimazation, ACM Trans. Comput. Log. 4 (2003) 181–206.

[47] R. Gentilini, C. Piazza, A. Policriti, From bisimulation to simulation: Coarsest partition problems, J. Autom. Reasoning 31 (2003) 73–103.

[48] C. H. Yang, D. L. Dill, Validation with guided search of the state space, in: DAC, pp. 599–604.

[49] P. Godefroid, S. Khurshid, Exploring very large state spaces using genetic algorithms, in: J.-P. Katoen, P. Stevens (Eds.), TACAS, volume 2280 of *Lecture Notes in Computer Science*, Springer, 2002, pp. 266–280.

[50] G. Behrmann, A. Fehnker, Efficient guiding towards cost-optimality in uppaal, in: T. Margaria, W. Yi (Eds.), TACAS, volume 2031 of *Lecture Notes in Computer Science*, Springer, 2001, pp. 174–188.

[51] R. Alur, B.-Y. Wang, "next" heuristic for on-the-fly model checking, in: J. C. M. Baeten, S. Mauw (Eds.), CONCUR, volume 1664 of *Lecture Notes in Computer Science*, Springer, 1999, pp. 98–113.

[52] M. O. Möller, R. Alur, Heuristics for hierarchical partitioning with application to model checking, in: T. Margaria, T. F. Melham (Eds.), CHARME, volume 2144 of *Lecture Notes in Computer Science*, Springer, 2001, pp. 71–85.

[53] S. Edelkamp, F. Reffel, Obdds in heuristic search, in: O. Herzog, A. Günter (Eds.), KI, volume 1504 of *Lecture Notes in Computer Science*, Springer, 1998, pp. 81–92.

[54] R. M. Jensen, R. E. Bryant, M. M. Veloso, Seta*: An efficient bdd-based heuristic search algorithm, in: AAAI/IAAI, pp. 668–673.

[55] S. Edelkamp, A. Lluch-Lafuente, S. Leue, Directed explicit model checking with hsf-spin, in: M. B. Dwyer (Ed.), SPIN, volume 2057 of *Lecture Notes in Computer Science*, Springer, 2001, pp. 57–79.

[56] A. Groce, W. Visser, Heuristic model checking for java programs, in: D. Bosnacki, S. Leue (Eds.), SPIN, volume 2318 of *Lecture Notes in Computer Science*, Springer, 2002, pp. 242–245.

[57] A. Santone, Heuristic search + local model checking in selective mu-calculus, IEEE Trans. Software Eng. 29 (2003) 510–523.

[58] E. M. Clarke, R. P. Kurshan (Eds.), Computer Aided Verification, 2nd International Workshop, CAV '90, New Brunswick, NJ, USA, June 18-21, 1990, Proceedings, volume 531 of *Lecture Notes in Computer Science*, Springer, 1991.

[59] K. G. Larsen, A. Skou (Eds.), Computer Aided Verification, 3rd International Workshop, CAV '91, Aalborg, Denmark, July, 1-4, 1991, Proceedings, volume 575 of *Lecture Notes in Computer Science*, Springer, 1992.

[60] C. Courcoubetis (Ed.), Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings, volume 697 of *Lecture Notes in Computer Science*, Springer, 1993.