GreASE: A Tool for Efficient “Nonequivalence” Checking

NICOLETTA DE FRANCESCO and GIUSEPPE LETTIERI, University of Pisa
ANTONELLA SANTONE, University of Sannio
GIGLIOLA VAGLINI, University of Pisa

Equivalence checking plays a crucial role in formal verification to ensure the correctness of concurrent systems. However, this method cannot be scaled as easily with the increasing complexity of systems due to the state explosion problem. This article presents an efficient procedure, based on heuristic search, for checking Milner’s strong and weak equivalence; to achieve higher efficiency, we actually search for a difference between two processes to be discovered as soon as possible, thus the heuristics aims to find a counterexample, even if not the minimum one, to prove nonequivalence. The presented algorithm builds the system state graph on-the-fly, during the checking, and the heuristics promotes the construction of the more promising subgraph. The heuristic function is syntax based, but the approach can be applied to different specification languages such as CCS, LOTOS, and CSP, provided that the language semantics is based on the concept of transition. The algorithm to explore the search space of the problem is based on a greedy technique; GreASE (Greedy Algorithm for System Equivalence), the tool supporting the approach, is used to evaluate the achieved reduction of both state-space size and time with respect to other verification environments.

Categories and Subject Descriptors: D.2.4 [**Software/Program Verification**]

General Terms: Theory, Verification

Additional Key Words and Phrases: Formal methods, heuristic searches, equivalence checking

ACM Reference Format:
Nicoletta de Francesco, Giuseppe Lettieri, Antonella Santone, and Gigliola Vaglini. 2014. GreASE: A tool for efficient “nonequivalence” checking. ACM Trans. Softw. Eng. Methodol. 23, 3, Article 24 (May 2014), 26 pages.
DOI: http://dx.doi.org/10.1145/2560563

1. INTRODUCTION

Verification of concurrent systems is often carried out by means of the analysis of the state space generated by the system. A common method is *model checking*, which is a technique originating in works by Clarke and Emerson [2008] and Queille and Sifakis [1982] 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 of) the system. An alternative way for establishing desirable properties of a model is by showing that it is behaviorally related to another model known to have those properties. Depending on the type of relation that is chosen, this verification technique is called *refinement* or *equivalence checking* [Parashkevov and Yantchev 1996]. There are two kinds of equivalences, defined by Milner [1989]: the strong equivalence, which means that the processes are not distinguishable; the weak

Authors’ addresses: N. de Francesco, G. Lettieri, Dipartimento di Ingegneria dell’ Informazione, University of Pisa, Italy; A. Santone (corresponding author), Dipartimento di Ingegneria, University of Sannio, Benevento, Italy; email: santone@unisannio.it; G. Vaglini, Dipartimento di Ingegneria dell’ Informazione, University of Pisa, Italy.

DOI: http://dx.doi.org/10.1145/2560563

equivalence, which defines a notion of observability for the processes and thus their equivalence is defined only with respect to the observable actions.

Equivalence checking, as well as model checking, suffers from the so-called *state explosion problem*: the parallelism between the processes leads to a number of reachable states which may become very large until breaking down automated verification.

Several approaches have been developed to solve or reduce the state explosion problem, among them, reduction techniques based on process equivalences [Bouajjani et al. 1990; Graf et al. 1996], symbolic model-checking techniques [McMillan 1993], on-the-fly techniques [Jard and Jéron 1991], local model-checking approaches [Stirling and Walker 1991], partial-order techniques [Godefroid 1996; Peled 1993; Valmari 1992], compositional techniques [Santone et al. 2013; Clarke et al. 1989; Santone 2002], and abstraction approaches [Clarke et al. 1994; Santone and Vaglini 2012; De Francesco et al. 1998].

Heuristic search [Pearl 1984] 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 cycles in the AND/OR graphs, or require the graphs to be acyclic. The latter class includes the AO* algorithm [Mahanti and Bagchi 1985; Pearl 1984], while the former includes the S2 algorithm [Mahanti et al. 2003]. In any case, the graph is expanded incrementally during the execution of the algorithm, starting from the initial node; a heuristic function assigns a cost to each node not yet considered.

This article proposes to check equivalence of concurrent systems by applying *heuristic search* techniques on AND/OR graphs; we consider that both the problem graph and the solution subgraphs can be cyclic, but this is not a problem in the search because we look for a node whose subgraph cannot be a solution (and then an acyclic subgraph); in some sense we search for a solution of the nonequivalence-checking problem. Moreover, since it is sometime better to find a solution as soon as possible rather than to repeatedly discard solutions until the optimum is found, optimality of the solution is not really an issue from the point of view of the method efficiency. Discarding optimality, a greedy approach to the problem can be followed: at any step, the next node to expand is chosen only on the basis of the foreseen cost of its expansion. With this approach the heuristic function can be very simple even if surprisingly effective.

First, we consider concurrent systems specified by means of a general process-algebra-based specification language and formalize the nonequivalence checking as a search problem on AND/OR graphs. Then we present the procedure based on the greedy algorithm. The method is completely automated, that is, there is no need for user intervention or manual effort. The heuristic function exploits the specification language syntax, so the method can be applied to different languages by only modifying the function. GreASE (Greedy Algorithm for System Equivalence) is the tool supporting the approach that has been used to verify a sample of process specifications, in order to evaluate the method. The experiments show that a considerable reduction of both state-space size and time can be achieved with respect to other checking algorithms. Our approach can be easily integrated in the Concurrency WorkBench of New Century tool (CWB-NC) [Cleaveland and Sims 1996], but also in other verification environments as, for example, the Construction and Analysis of Distributed Processes tool (CADP). Moreover, the set of checked equivalences can be enlarged to include, for example, the ρ-equivalence (defined in Barbuti et al. [1999]) and the $\tau * \alpha$-equivalence (defined in Fernandez et al. [1996]).

The article is organized as follows: in Section 2 the basic concepts of behavioral equivalence are recalled. Section 3 presents our greedy algorithm approach. In Section 4 the prototype tool implementing the approach is briefly presented, and the experimental results we obtained by using it are reported. Finally, comparisons with related work are made in Section 5, and our conclusions and future work are presented in Section 6.

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 $\{(p_1, \alpha_1), \dots, (p_n, \alpha_n)\} \subseteq \Delta \times \Theta$. If $(p', \alpha) \in \sigma(p)$, we say that p can perform the action α and reach the process p', and we write $p \xrightarrow{\alpha} p'$. In the following, given $p \in \Delta$, the process reachable from p by σ is called *σ-derivative* of 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.* Behavioral equivalence can be used also for other purposes, like, for example, for clone detection [Cuomo et al. 2013].

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 processes 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.

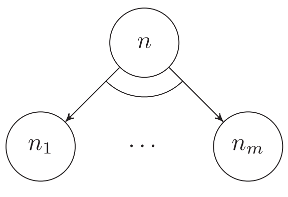

Fig. 1. An AND node.

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 weakly equivalent ($p \approx q$) iff there exists a weak bisimulation $\mathcal{B}$ containing the pair (p, q).

3. THE APPROACH

In this section we first recall the general definition of AND/OR graphs, then, in Section 3.1, we propose a greedy algorithm to visit an AND/OR graph and find a suitable solution of the search problem represented by the graph. The algorithm uses a heuristic function to be supported in finding a solution as soon as possible. The heuristic function first computes a measure of the actions to be performed by each process separately, and then compares the two values: the more they differ, the faster the expansion of the corresponding processes should lead to a failure.

It is well known that a common problem-solving strategy consists of decomposing a problem P into subproblems. We can introduce two kinds of decomposition: a decomposition where all of the subproblems need to be solved in order to obtain a solution for P, and a decomposition where solving just one of the subproblems is sufficient. So 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. In this way a problem can be formalized in terms of a graph whose precise definition follows.

An AND/OR graph G is a directed graph with two types of special nodes: the *start* (or *root*) *node*, called s, and a 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 other nodes of G are nonterminal nodes and 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 visually distinguished from OR nodes by connecting the subproblem arcs by a line, like in Figure 1.

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

Definition 3.1. A finite subgraph SG of an AND/OR graph G is a solution subgraph if:

(i) the start node of G is in SG;
(ii) if n is an OR node in G and n is in SG, then exactly one of the immediate successors of n in G is in SG;

(iii) if n is an AND node in G and n is in SG, then all the immediate successors of n in G are in SG;
(iv) every acyclic maximal path in SG ends in a terminal leaf node.

Note that the preceding definition differs from the classical one [Mahanti and Bagchi 1985] since we allow cyclic paths in the solution graph. Cyclic solutions are generally disallowed in other applications of AND/OR graphs, since they normally imply a failure to reduce the original problem to solvable subproblems, but this is not an issue in our context. Other uses of cyclical solutions of AND/OR graphs can be found in Hansen and Zilberstein [2001].

We also need the notion of a *counterexample subgraph* of an AND/OR graph.

Definition 3.2. A finite acyclic subgraph C of an AND/OR graph G is a counterexample subgraph if:

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

Note that, since C is acyclic, it must contain some nonterminal leaf node.

LEMMA 3.3. *An AND/OR graph that contains a counterexample subgraph cannot contain any solution subgraph.*

PROOF. Let G be an AND/OR graph and assume by contradiction that G contains both a counterexample subgraph C and a solution subgraph SG. Since C is acyclic, we can sort its nodes topologically so that if a node n is an ancestor of a different node m in C, then $n < m$ in the ordering. Note that the start node of C is also in SG. Now, using Definitions 3.1 and 3.2 we can find a sequence of nodes $n_1 < n_2 < \cdots$ common to both C and SG. Since C is finite, this sequence must end in a nonterminal leaf, and therefore SG cannot be a solution subgraph. □

3.1. The Greedy Algorithm on AND/OR Graphs

In most domains the AND/OR graph G representing the problem is unknown in advance, so it cannot be supplied explicitly to the search algorithm. Instead, a distinction is drawn between the *explicit graph* G' and the *implicit graph* G. The implicit graph G is specified by a start node s and a successor function. The search algorithm works on the explicit graph G', which initially consists of the start node s only. The start node is then expanded using the successor function of G to obtain a set of nodes and arcs that are added to G'. At any instant, the explicit graph G' has a number of nodes yet to be expanded, called *tip nodes*. The search algorithm chooses one of these tip nodes for expansion. In this manner, more and more nodes and arcs get added to the explicit graph, until it finally has one or more solution graphs as subgraphs. One of these solution graphs is then output by the search algorithm.

Table I shows the greedy algorithm proposed in this article for the heuristic search in AND/OR graphs. The algorithm receives as input the start node s, the operators used to expand a node and the heuristic function $\widehat{h}$. It builds an explicit graph G' by expanding a new node for each iteration of cycle 2. The next node to be expanded is chosen according to a strategy based on the value returned by the heuristic function. Note that the correctness of the algorithm (on finite processes) is not affected by the chosen strategy. The strategy, however, should favor nodes with higher values of the

Table I. The Greedy Algorithm

```
(1) Let $G' = (\{s\}, \varnothing)$. If $s$ is a non-terminal leaf mark it as FAILED.
(2) While $s$ is not marked as FAILED:
    (a) If the current psg has no tip nodes exit with SUCCESS.
    (b) Otherwise, let $n$ be the tip node of the current psg supplied by the chosen strategy on the
        basis of the $\widehat{h}$ value. Add all successors of $n$ to $G'$. For each successor $m$ of $n$, add an arc
        from $n$ to $m$. If any successor is a non-terminal leaf, mark it as FAILED.
    (c) Let $Z = \{n\}$.
    (d) While $Z$ is not empty;
        i. extract any node $m$ from $Z$.
        ii. —($m$ is an AND node?) If any successor of $m$ is marked as FAILED, mark $m$ as FAILED,
              otherwise mark all outgoing arcs of $m$.
            —($m$ is an OR node?) If all successors of $m$ are marked as FAILED mark $m$ as FAILED.
              Otherwise, consider the successors of $m$ that are not marked as FAILED and mark
              the arc going to the one with the greatest value of $\widehat{h}$.
        iii. If $m$ has been marked as FAILED, unmark all outgoing arcs of $m$ and add to $Z$ all
              predecessors of $m$ along marked arcs.
(3) Exit with FAILURE.
```

heuristic function, otherwise the inconsistency with the other use of the heuristic function in step (ii) would likely affect performance negatively.

Each arc in G' may be either marked or unmarked. At each iteration, marked arcs individuate the possible solution graph (*psg*) that has been built up to that moment. More precisely, by *current psg* we denote the subgraph of G' reachable from s by following marked arcs. The tip nodes of the current *psg* can only be OR nodes or AND nodes, since arcs entering nonterminal leaves cannot be marked.

In order to guarantee that a solution is found whenever there is one, the algorithm backtracks when it finds that the current *psg* cannot be completed. After a node is expanded, the algorithm enters loop (d) where it decides whether it can expand the current *psg* by marking outgoing arcs of the newly expanded node, or if it has to backtrack instead. The backtracking is performed by unmarking arcs in step (iii) and propagating back the message along the *psg*. Nodes are definitively marked as FAILED as soon as the algorithm discovers that there is no solution graph below them. Nodes marked as FAILED need not be reexamined if they are ever visited again, either by backtracking or because they are a common successor of other visited nodes.

The algorithm exits successfully from step (a) when it has not found any tip node in a visit of the current *psg*. Note that this allows for *psg* with cycles, as we already noted in the comment following Definition 3.1.

We now give a justification for the correctness of the algorithm.

THEOREM 3.4. *If the input implicit graph is finite, the algorithm of Table I terminates after a finite number of steps.*

PROOF. Note that an arc that has been unmarked in step (iii) cannot be marked again. Since new nodes are added to Z only if at least one arc had been unmarked before, cycle (d) must terminate. Step (b) expands a new node at every iteration of cycle 2, so also this cycle must terminate. □

To prove Theorem 3.6 later, we need the following lemma. Here, by "AND/OR subgraph (of an AND/OR graph G) starting at (node) n" we mean the AND/OR graph obtained by taking n as a start node, and then adding all G's nodes that are reachable from n, together with all arcs between them.

LEMMA 3.5. *If the algorithm of Table I marks a node n as FAILED, then the AND/OR subgraph starting at n contains a counterexample subgraph.*

PROOF. This proof is adapted from Mahanti et al. [2003]. The proof is by induction on the number of nodes marked as FAILED during a given run of the algorithm. The first

FAILED node, be it n, must be a nonterminal leaf, so the base step is trivially true, since $C(n)$ is obviously a counterexample. In the induction step, the newly marked node, n, may be either a nonterminal leaf, an AND node, or an OR node. The nonterminal leaf case is again trivial. The AND node case is easy: at least one of its successor nodes, be it m, must have been marked as FAILED in a previous step. By the induction hypothesis, the AND/OR subgraph starting at m contains a counterexample subgraph $C(m)$. We can obtain the counterexample subgraph $C(n)$ that we need by taking the whole of $C(m)$ and connecting the node n by an arc to m. The OR node case needs some care to avoid creating cycles in the graph: by the induction hypothesis, all successors of n, call them $n_1, n_2, \dots, n_k$, have already been marked as FAILED, and there exists a counterexample subgraph $C(n_i)$, $1 \leq i \leq k$, for each of them. We build $C(n)$ as follows. Let $C(n)_p$ be the partial counterexample built until a time (we start from $C(n)_p$ empty). Moreover, let CG be a set of counterexample subgraphs (we start with $CG = \{C(n_i) \mid 1 \leq i \leq k\}$):

(1) Add to $C(n)_p$ the node n_1, namely the root of $C(n_1)$, and all its outgoing arcs; let n_{1_1}, $\dots$, n_{1_r} be the successor nodes of n_1, and the subgraphs rooted at such nodes are counterexamples too, call them $C(n_{1_1}), \dots, C(n_{1_r})$.
(2) Cancel $C(n_1)$ from CG and add each $C(n_{1_z})$, $1 \leq z \leq r$ to CG only if n_{1_z} is not already a node of $C(n)_p$.
(3) Repeat steps 1 and 2 until CG is empty.
(4) Add n and all its outgoing arcs to $C(n)_p$ so obtaining $C(n)$.

This process guarantees that the subgraph that we have built is acyclic. □

Now we can prove the main theorem.

THEOREM 3.6. *If the algorithm of Table I terminates with SUCCESS, then the input implicit graph contains a solution graph. If the algorithm terminates with FAILURE, the implicit graph contains no solution graph.*

PROOF. Cycle (d) ensures that no marked arc can point to a node marked as FAILED. On the other end, AND nodes not marked as FAILED have all their outgoing arcs marked, and OR nodes not marked as FAILED have exactly one outgoing arc marked. This ensures that the algorithm terminates in step (a) only if it has actually found a solution graph, thus proving the first claim. For the second claim, note that the algorithm terminates with FAILURE only when the start node is marked as FAILED. By Lemma 3.5, this means that the implicit graph contains a counterexample subgraph. Then Lemma 3.3 completes the proof. □

3.2. AND/OR Graph for Strong and Weak Equivalences Checking

Consider two processes, p and q: to check the requirements of Definition 2.1 and Definition 2.2, an AND/OR graph is built representing the fact that p and q move in alternating turns as long as they can perform the same move. Thus the AND/OR graph has a solution iff p and q are strongly or weakly bisimilar, since the construction halts with nonterminal leaves only when there is a move of p that cannot be matched by q, or vice versa.

We denote by $G(p, q, \rightsquigarrow)$ the implicit AND/OR graph built from p, q and the transition relation $\rightsquigarrow$, which can be either $\longrightarrow$ for strong equivalence, or $\Longrightarrow$ for weak equivalence. The nodes of $G(p, q, \rightsquigarrow)$ are 4-tuples $\langle r, s, \gamma, u \rangle$ where:

—r is a $\rightsquigarrow$-derivative of p;
—s is a $\rightsquigarrow$-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 $\gamma = \top$. The first (respectively, second) argument of the 4-tuple is always a state reachable from the process p (respectively, q). The combination of the third argument of the 4-tuple with the fourth one indicates who is the next process that has to move and which action has to be performed. In particular, 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). For example, consider the set of processes $\Delta = \{p, p_1, q, q_1\}$; the set of actions $\Theta = \{a\}$; and the function σ such that $p \xrightarrow{a} p_1$ and $q \xrightarrow{a} q_1$.

$\langle p, q, \top, \lambda\rangle$ means:

"both the process p and the process q can move; in the previous rounds each process has matched the actions of the other process."

$\langle p, q, \bot, 1\rangle$ means:

"it is the turn of the process p to move with any action."

$\langle p_1, q, a, 2\rangle$ means:

"now it is the turn of the process q to move with the action a, which is the action that the other process has performed in the previous round becoming the process p_1."

Let $n = \langle r, s, \gamma, u\rangle$ be a node of $G(p, q, \rightsquigarrow)$. Let us say that the *current process* of n is r if $u = 1$, or s if $u = 2$. Node n is an AND node, OR node, terminal, or nonterminal leaf according to the value of γ and the following rules:

—when $\gamma = \top$, node n is an AND node if $r \neq s$, otherwise it is terminal leaf;
—when $\gamma = \bot$, node n is an AND node if its current process has any $\rightsquigarrow$-derivative, otherwise it is a terminal leaf;
—when $\gamma = \alpha$, node n is an OR node if its current process has a $\overset{\alpha}{\rightsquigarrow}$ transition, otherwise it is a nonterminal leaf.

The start node of $G(p, q, \rightsquigarrow)$ is $\langle p, q, \top, \lambda\rangle$. The successor function is given by the operators in Table II. The operators generate the outgoing arcs and the successor nodes of each node. 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.

The premises of the rules are general and able to manage both strong and weak equivalence depending on the particular relation that $\rightsquigarrow$ represents.

The rule $\mathbf{op}_1$ transforms the initial node, which is an AND node, into the two successor nodes with $\gamma = \bot$ and $u = 1$ and $u = 2$, respectively, stating that both processes can start to move. The rule $\mathbf{op}_2$ is applied when it is the turn of the process p to move, thus, the rule points out the possible moves (α_i) of p; in this way an AND node can be connected with its successor nodes in the graph. All such successors have $\gamma = \alpha_i$ and

Table II. General Operators

$\mathbf{op}_1$	$\dfrac{}{\langle p, q, \top, \lambda\rangle \longrightarrow \langle p, q, \bot, 1\rangle \ \textbf{and}\ \langle p, q, \top, \lambda\rangle \longrightarrow \langle p, q, \bot, 2\rangle}$
$\mathbf{op}_2$	$\dfrac{p \overset{\alpha_1}{\rightsquigarrow} p_1, \ldots, p \overset{\alpha_n}{\rightsquigarrow} p_n,\ n \geq 1}{\langle p, q, \bot, 1\rangle \longrightarrow \langle p_1, q, \alpha_1, 2\rangle \ \textbf{and}\ \cdots\ \textbf{and}\ \langle p, q, \bot, 1\rangle \longrightarrow \langle p_n, q, \alpha_n, 2\rangle}$
$\mathbf{op}'_2$	$\dfrac{q \overset{\alpha_1}{\rightsquigarrow} q_1, \ldots, q \overset{\alpha_n}{\rightsquigarrow} q_n,\ n \geq 1}{\langle p, q, \bot, 2\rangle \longrightarrow \langle p, q_1, \alpha_1, 1\rangle \ \textbf{and}\ \cdots\ \textbf{and}\ \langle p, q, \bot, 2\rangle \longrightarrow \langle p, q_n, \alpha_n, 1\rangle}$
$\mathbf{op}_3$	$\dfrac{p \overset{\alpha}{\rightsquigarrow} p_1, \ldots, p \overset{\alpha}{\rightsquigarrow} p_n,\ n \geq 1}{\langle p, q, \alpha, 1\rangle \longrightarrow \langle p_1, q, \top, \lambda\rangle \ \textbf{or}\ \cdots\ \textbf{or}\ \langle p, q, \alpha, 1\rangle \longrightarrow \langle p_n, q, \top, \lambda\rangle}$
$\mathbf{op}'_3$	$\dfrac{q \overset{\alpha}{\rightsquigarrow} q_1, \ldots, q \overset{\alpha}{\rightsquigarrow} q_n,\ n \geq 1}{\langle p, q, \alpha, 2\rangle \longrightarrow \langle p, q_1 \top, \lambda\rangle \ \textbf{or}\ \cdots\ \textbf{or}\ \langle p, q, \alpha, 2\rangle \longrightarrow \langle p, q_n, \top, \lambda\rangle}$

$u = 2$, stating that in the next turn the process q has to move matching the action α_i. 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 in the previous round, while in rule $\mathbf{op}'_3$, it is the process q that must simulate the action α performed by p in the previous round. In both cases an OR node can be connected with its successor nodes in the graph, all such successors having $\gamma = \top$ and $u = \lambda$, that is, nodes that can be transformed only through the operator $\mathbf{op}_1$ like the initial node.

Example 3.7. Consider the set of processes $\Delta = \{p, p_1, q, q_1, q_2, q_3\}$; the set of actions $\Theta = \{a, b\}$; and the function σ such that

$$p \xrightarrow{a} p_1;$$

$$q \xrightarrow{a} q_1; \quad q \xrightarrow{b} q_2; \quad q_1 \xrightarrow{b} q_3; \quad q_2 \xrightarrow{a} q_3.$$

The AND/OR graph $G(p, q, \longrightarrow)$ generated starting from $\langle p, q, \top, \lambda\rangle$, using the operators of Table II, is shown in Figure 2. Nonterminal leaf nodes are underlined.

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

Theorem 3.8. *Let p and q be two finite CCS processes and consider the AND/OR graph $G(p, q, \rightsquigarrow)$ generated starting from $\langle p, q, \top, \lambda\rangle$ using the operators of Table II. Then:*

(i) *$p \sim q$ iff $G(p, q, \longrightarrow)$ has a solution;*
(ii) *$p \approx q$ iff $G(p, q, \Longrightarrow)$ has a solution.*

Proof. We prove point (i) only, since point (ii) can be proved similarly.

Let us prove the $\Leftarrow$ direction first. Take a solution SG of $G(p, q, \longrightarrow)$ and consider the relation

$$S = \{(s, t) \mid \langle s, t, \top, \lambda\rangle \text{ is a node of } SG\}.$$

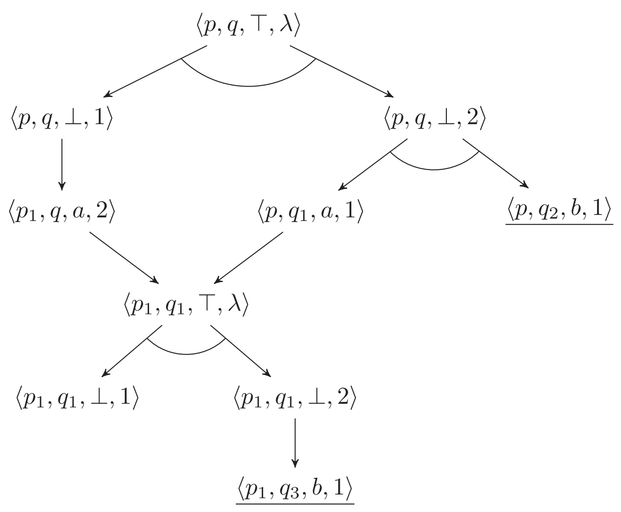
$\langle p, q, \top, \lambda\rangle$
$\langle p, q, \bot, 1\rangle$
$\langle p, q, \bot, 2\rangle$
$\langle p_1, q, a, 2\rangle$
$\langle p, q_1, a, 1\rangle$
$\langle p, q_2, b, 1\rangle$
$\langle p_1, q_1, \top, \lambda\rangle$
$\langle p_1, q_1, \bot, 1\rangle$
$\langle p_1, q_1, \bot, 2\rangle$
$\langle p_1, q_3, b, 1\rangle$

Fig. 2. The AND/OR graph $G(p, q, \longrightarrow)$.

We claim that S is a strong bisimulation. Indeed, consider any $(s, t) \in S$. If $s = t$, then clearly Definition 2.1 is satisfied, otherwise assume $s \xrightarrow{\alpha} s'$ for some action α and process s'. By Table II, the AND/OR graph will contain the path

$$\langle s, t, \top, \lambda\rangle \to \langle s', t, \bot, 1\rangle \to \langle s', t, \alpha, 2\rangle.$$

Since the first two nodes are AND nodes, the first one is in SG and SG is a solution, SG must contain all of them. The last node is an OR node, therefore SG will also contain one of its successor nodes, call it n. Node n can only have been obtained via rule $\mathbf{op}'_3$ in Table II. Therefore, we must have $t \xrightarrow{\alpha} t'$ and $n = \langle s', t', \top, \lambda\rangle$ for some t', which also means that (s', t') is in S. This proves point (i) of Definition 2.1. Point (ii) is proved symmetrically, and this shows that S is indeed a strong bisimulation. Since clearly $(p, q) \in S$, this proves that $p \sim q$ holds.

Now let us prove the $\Rightarrow$ direction. Assume we are given a strong bisimulation $\mathcal{B}$ such that $(p, q) \in \mathcal{B}$. First, we build an AND/OR graph SG in three steps.

(A) For each $(s, t) \in \mathcal{B}$, add the AND nodes $\langle s, t, \top, \lambda\rangle$, $\langle s, t, \bot, 1\rangle$ and $\langle s, t, \bot, 2\rangle$ with the appropriate arcs.
(B) For each $s \xrightarrow{\alpha} s'$ such that $(s, t) \in \mathcal{B}$ for some t, find t' such that $t \xrightarrow{\alpha} t'$ and $(s', t') \in \mathcal{B}$ (such a t' must exist because of Definition 2.1), then add the OR node $\langle s', t, \alpha, 2\rangle$ with an arc coming from $\langle s, t, \bot, 1\rangle$ and an arc going to $\langle s', t', \top, \lambda\rangle$ (these latter two nodes were added in step A). Operate analogously for each $t \xrightarrow{\alpha} t'$ such that $(s, t) \in \mathcal{B}$ for some s.
(C) Note that step A has added the node $\langle p, q, \top, \lambda\rangle$. Now remove all nodes (together with their incident arcs) that are not reachable from it.

These steps make SG be a subgraph of $G(p, q, \longrightarrow)$ which contains the start node, all successors of all AND nodes, and one successor for each OR node. Noncyclic maximal paths in SG must end in AND nodes without successors, which are terminal leaves in our model. Therefore, SG is a solution. $\square$

Table III. Operational Semantics of CCS

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

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) [Milner 1989], which is an algebra suitable for modelling and analysing processes. The reader can refer to Milner [1989] for further details. The syntax of *processes* is

$$p ::= 0 \mid \alpha.p \mid p+p \mid p\,|\,p \mid p\backslash L \mid p[f] \mid x,$$

where α ranges over a finite set of actions $Act = \{\tau, a, \bar{a}, b, \bar{b}, \ldots\}$. Input actions are labeled with “nonbarred” names, for example, a, while output actions are “barred”, for instance, $\bar{a}$. 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 $\{\bar{l}, l \mid l \in L\}$. We call $\mathcal{P}$ the processes generated by p.

The standard *operational semantics* [Milner 1989] 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 III (we omit the symmetric rule of **Sum** and **Par**).

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

If $\delta \in Act^*$ and $\delta = \alpha_1 \ldots \alpha_n$, $n \geq 1$, we write $p \xrightarrow{\delta} q$ to mean $p \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} q$. Moreover $p \xrightarrow{\lambda} p$, where λ is the empty sequence. Given $p \in \mathcal{S}$, with $\mathcal{R}(p) = \{q \mid p \xrightarrow{\delta} q\}$ we denote the set of the states reachable from p by zero or more $\longrightarrow$, 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.

Remark 3.9. 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 $\bar{\alpha}$ 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

[1]The sort of a CCS process p is the alphabet of p. For the precise definition, see Milner [1989].

$$\begin{tikzcd}
 & & & & & \langle p, q, \top, \lambda\rangle \arrow[dlll] \arrow[drrr] & & & & & & \\
 & & \langle p, q, \bot, 1\rangle \arrow[dl] \arrow[drr] & & & & & & \langle p, q, \bot, 2\rangle \arrow[dl] \arrow[drr] & & & \\
 & \langle 0, q, b, 2\rangle \arrow[d] & & & \langle k.d.f.0, q, c, 2\rangle \arrow[d] & & & \langle p, c.0, b, 1\rangle \arrow[d] & & & \langle p, k.d.0, c, 1\rangle \arrow[d] & \\
 & \langle 0, c.0, \top, \lambda\rangle \arrow[dl] \arrow[dr] & & & \langle k.d.f.0, k.d.0, \top, \lambda\rangle \arrow[dl] \arrow[dr] & & & \langle 0, c.0, \top, \lambda\rangle \arrow[dl] \arrow[dr] & & & \langle k.d.f.0, k.d.0, \top, \lambda\rangle \arrow[dl] \arrow[dr] & \\
\cdots & & \cdots & \cdots & & \cdots & \cdots & & \cdots & \cdots & & \cdots
\end{tikzcd}$$

Fig. 3. The AND/OR graph $G(p, q, \longrightarrow)$.

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 article 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 graph for CCS processes, we take Δ as the set of CCS processes $\mathcal{P}$, and the σ function, described in the previous section, is rephrased using the standard operational semantic, namely, $\sigma(p) = \{(p', \alpha) \mid p \xrightarrow{\alpha} p'\} = \{(p_1, \alpha_1), \ldots, (p_n, \alpha_n)\}$.

Example 3.10. Consider the following CCS processes.

$$p \overset{\text{def}}{=} b.0 + c.k.d.f.0$$

$$q \overset{\text{def}}{=} b.c.0 + c.k.d.0$$

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

3.4. Heuristic Functions

In order to apply the greedy algorithm, a heuristic function over nodes has to be defined. Such a function is called $\widehat{h}$ and is aimed at working during the construction of the AND/OR graph by means of the operators of Table II: $\widehat{h}$ should suggest the state containing two not bisimilar processes (with respect to a particular equivalence) so that this state can be included in the graph as soon as possible. The function assigns a value to each node n of the graph, called $\widehat{h}$-value of n: roughly speaking, that value is a measure of the degree of dissimilarity of the two processes in n. In this section, we present two heuristics functions, $\widehat{h}_{nc}$ and $\widehat{h}_{com}$, able to manage both strong and weak equivalence and equivalently easy to be computed since they depend only on the syntax of the processes and not on their semantics. Both functions consider that a structural difference between the processes can be a good index of their nonequivalence; obviously, for example, two not strongly equivalent processes are also structurally different, and two structurally equal processes are also strongly equivalent. Nevertheless, two not structurally equal processes can be strongly equivalent; in this sense, $\widehat{h}$ is a heuristics. The two functions have a different behavior regarding communication actions: $\widehat{h}_{nc}$

Table IV. The $\widehat{h}_{\mathcal{V}}$ Function

$\widehat{h}_{\mathcal{V}}(0, S)$	$= 0$	**R1.**
$\widehat{h}_{\mathcal{V}}(\alpha.p', S)$	$= \begin{cases} \widehat{h}_{\mathcal{V}}(p', S) & \text{if } \alpha \in S, \\ 1 + \widehat{h}_{\mathcal{V}}(p', S) & \text{otherwise;} \end{cases}$	**R2.**
$\widehat{h}_{\mathcal{V}}(p_1 + p_2, S)$	$= \max(\widehat{h}_{\mathcal{V}}(p_1, S), \widehat{h}_{\mathcal{V}}(p_2, S))$	**R3.**
$\widehat{h}_{\mathcal{V}}(p_1 \mid p_2, S)$	$= \widehat{h}_{\mathcal{V}}(p_1, S) + \widehat{h}_{\mathcal{V}}(p_2, S)$	**R4.**
$\widehat{h}_{\mathcal{V}}(p' \backslash L, S)$	$= \begin{cases} \widehat{h}_{\mathcal{V}}(p', S \cup L^{\circ}) & \text{if } S \neq \emptyset, \\ \widehat{h}_{\mathcal{V}}(p', S) & \text{otherwise;} \end{cases}$	**R5.**
$\widehat{h}_{\mathcal{V}}(x, S)$	$= 0$	**R6.**

completely ignores the communication actions, while $\widehat{h}_{com}$ considers also these ones. In some sense, the latter function can be seen as a refinement of the former: when two processes exhibit the same number of visible actions, we can try to distinguish them by considering also their communication capabilities. We describe here two possible heuristic functions to show that the syntactic structure of the processes can supply a lot of information on their semantic equivalence. This information can be very simple, even if very effective, as in the case of the function $\widehat{h}_{nc}$; it can be more complex, as in the case of $\widehat{h}_{com}$, but at the same low computing cost. It is also possible to think of more punctual functions to investigate, for example, the structure of the variable definitions, this at a higher computing cost.

3.4.1. Heuristics Based on Noncommunication Actions: $\widehat{h}_{nc}$. Since the primary cause of nonequivalence is due to a different number of visible actions, the first heuristic only gives an approximate measure of the number of such actions in a process. The function $\widehat{h}_{nc}$ uses some auxiliary functions:

—$\widehat{h}_{\mathcal{V}}$, where $\mathcal{V}$ is the set of visible actions;
—$\widehat{h}_{\alpha}$, where $\alpha \in Act$; it is actually a family of functions one for each $\alpha \in Act$;
—$\Xi(p, q) = \begin{cases} \widehat{h}_{\mathcal{V}}(p, S) - \widehat{h}_{\mathcal{V}}(q, S) & \text{if } \widehat{h}_{\mathcal{V}}(p, S) \neq \widehat{h}_{\mathcal{V}}(q, S), \\ 0 & \text{otherwise.} \end{cases}$

$\widehat{h}_{\mathcal{V}}(p, S)$ and $\widehat{h}_{\alpha}(p, S)$ have two arguments: $p \in \mathcal{P}$ and $S \subseteq \mathcal{V}$ and are inductively defined on the process p as shown in Table IV and Table V, where S contains the actions that can be skipped depending of the type of equivalence. Ξ is applied to two processes, p and q, and suggests which process has to be chosen to perform the next move. If the number of visible actions that p can perform (i.e., $\widehat{h}_{\mathcal{V}}(p, S)$) is bigger that those of q (i.e., $\widehat{h}_{\mathcal{V}}(q, S)$), it is better to move p first, q otherwise. In fact, our point is that, when the process that has more visible actions moves first, the other process can have difficulty to mimic the performed action. In this way, a failure could be reached more quickly.

Definition 3.11 (*$\widehat{h}_{nc}$: Heuristic Function*). Let $n = \langle p, q, \gamma, u \rangle$ be a node.

$$\widehat{h}_{nc}(n) = \begin{cases} |\Xi(p, q)| & \text{if } \gamma = \top, \\ \Xi(p, q) & \text{if } \gamma = \bot \text{ and } u = 1, \\ \Xi(q, p) & \text{if } \gamma = \bot \text{ and } u = 2, \\ \widehat{h}_{\alpha}(p, S) & \text{if } \gamma = \alpha \text{ and } u = 1, \\ \widehat{h}_{\alpha}(q, S) & \text{if } \gamma = \alpha \text{ and } u = 2. \end{cases}$$

Table V. The $\widehat{h}_\alpha$ Function

$\widehat{h}_\alpha(0, S)$	$= 0$	**R1.**
$\widehat{h}_\alpha(\beta.p', S)$	$= \begin{cases} \widehat{h}_\alpha(p', S) & \text{if } \beta = \alpha \text{ or } \beta \in S \\ 1 + \widehat{h}_\alpha(p', S) & \text{otherwise;} \end{cases}$	**R2.**
$\widehat{h}_\alpha(p_1 + p_2, S)$	$= \max(\widehat{h}_\alpha(p_1, S), \widehat{h}_\alpha(p_2, S))$	**R3.**
$\widehat{h}_\alpha(p_1 \mid p_2, S)$	$= \widehat{h}_\alpha(p_1, S) + \widehat{h}_\alpha(p_2, S)$	**R4.**
$\widehat{h}_\alpha(p' \backslash L, S)$	$= \begin{cases} \widehat{h}_\alpha(p', S \cup L^\circ) & \text{if } S \neq \emptyset, \\ \widehat{h}_\alpha(p', S) & \text{otherwise;} \end{cases}$	**R5.**
$\widehat{h}_\alpha(x, S)$	$= 0$	**R6.**

Initially:

—$S = \varnothing$ for the strong equivalence, since all the actions are important;
—$S = \{\tau\}$ for the weak equivalence, since the silent action can be ignored. In this case, S is also augmented, when a restriction operator $p' \backslash L$ is encountered, by the actions contained in the set L°.

When the method is applied to check different equivalences, S will be differently initialized.

To clarify the use of the heuristic function, consider the following two nodes, between which the next one to be expanded must be chosen

$$n_1 = \langle p, q, \top, \lambda \rangle;$$
$$n_2 = \langle p', q, \top, \lambda \rangle;$$

with $p = b.0 \mid a.0$, $p' = b.0$ and $q = b.0$. Moreover, suppose that we are looking for strong bisimilarity, so all actions must be considered and then $S = \varnothing$. The value of $\widehat{h}_{\mathcal{V}}(p, S)$ is 2, while $\widehat{h}_{\mathcal{V}}(p', S) = 1$ and $\widehat{h}_{\mathcal{V}}(q, S) = 1$. Thus, $|\Xi(p, q)|$ is equal to 1, while $|\Xi(p', q)| = 0$. From this fact we deduce that the expansion of n_1 is more promising for proving nonbisimilarity. Alternatively, when the nodes are the following.

$$n_1 = \langle p, q, \perp, 2 \rangle$$
$$n_2 = \langle q, p, \perp, 2 \rangle$$

Since it is the turn of the second process to move in both nodes (the last component of the nodes is equal to 2), we shall look for the actual values of $\widehat{h}_{nc}$. It turns out that $\widehat{h}_{nc}(n_1) = \Xi(q, p) = -1$ and $\widehat{h}_{nc}(n_2) = \Xi(p, q) = 1$. From this difference of values, we deduce that the expansion of n_2 is more promising; in fact, if p performs action a, it can be immediately observed that q cannot mimic such action. A similar behavior can be followed when $u = 1$ and it is the turn of the first process to move.

Finally, when considering nodes with $\gamma = \alpha$, we are in the case in which one process has performed the action α and, by the function $\widehat{h}_\alpha$, we want to approximate the number of visible actions that the other process must perform before and after α, if it exists. In other words, this value indicates the effort to explore the entire process in which α should occur.

Now, we analyse the function $\widehat{h}_{\mathcal{V}}$, shown in Table IV, more in detail.

—*Rule R1* applies to a process $p = 0$ and banally returns 0 as 0 cannot perform any visible actions.
—*Rule R2* applies when $p = \alpha.p'$. If α is not in S, that is, it is a visible action, the function is recursively applied on p', increasing by 1 its value; otherwise it is applied to p' to find, if any, an action not in S.
—*Rule R3* applies when $p = p_1 + p_2$ and the maximum number of actions computed for the two components is returned.
—*Rule R4* applies when $p = p_1|p_2$ and the sum of actions between the two components is returned.
—*Rule R5* applies to $p = p'\backslash L$ and simply adds the set of actions L° to S, only when we are managing weak equivalence (i.e., initially, $S \neq \varnothing$). The reason is that the communication actions will produce the τ action. When strong equivalence is considered, no action can be ignored, and so the function is recursively applied on p' without modifying S.
—*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 function $\widehat{h}_\alpha$, shown in Table V is similar to $\widehat{h}_{\mathcal{V}}$ except when applied to $p = \beta.p'$: in fact, only when $\beta \neq \alpha$ and $\beta \notin S$ this action is counted, since α has been performed by the other process and β can make the two processes different.

Example 3.12. Consider the following CCS processes $p = a.0$ and $q = a.0|b.0$ and apply the method to check whether p and q are weak bisimilar. Using the Concurrency WorkBench of New Century tool[2] (CWB-NC) [Cleaveland and Sims 1996], a well-known tool for model-checking concurrent systems, the result is as follows.

```
cwb-nc> eq -S obseq a.nil  a.nil|b.nil
Building automaton...
States: 6
Transitions: 5
Done building automaton.
Transforming automaton...
Done transforming automaton.
FALSE...
a.nil satisfies:
        [[b]]ff
a.nil|b.nil does not.
Execution time (user,system,gc,real):(0.016,0.000,0.000,0.016)
```

The space complexity of equivalence checking in CWB-NC is heavily influenced by the size of $\mathcal{S}(p)$ and $\mathcal{S}(q)$, since the transition systems for the two processes are always completely built. In fact, the transition system for p has two states, while that for q has four states. On the other hand, our approach builds the AND/OR graph whose nodes represents pairs of states of the transition systems of the two processes, but, as shown in Figure 4, we stop the generation when we can deduce that the processes are not weakly bisimilar. In Figure 4, the $\widehat{h}_{nc}$ value of each node is reported. FAILED nodes are recognized as underlined nodes. This is a small example, but it shows the reduction of

[2]Note that the Concurrency WorkBench of New Century tool uses `nil` instead of 0.

$$\begin{tikzcd}
 & \underline{\langle a.0, a.0 \mid b.0, \top, \lambda\rangle} \arrow[dl, "\widehat{h}_{nc} = -1"'] \arrow[dr, "\widehat{h}_{nc} = 1"] & & \\
\langle a.0, a.0 \mid b.0, \bot, 1\rangle & & \underline{\langle a.0, a.0 \mid b.0, \bot, 2\rangle} \arrow[dl] \arrow[dr] & \\
 & \langle a.0, 0 \mid b.0, a, 1\rangle & & \underline{\langle a.0, a.0 \mid 0, b, 1\rangle}
\end{tikzcd}$$

Fig. 4. An example.

Table VI. The $\widehat{h}_{\mathcal{C}}$ Function

$$\begin{array}{lll}
\widehat{h}_{\mathcal{C}}(0, S) & = 0 & \textbf{R1.} \\
\widehat{h}_{\mathcal{C}}(\beta.p', S) & = \begin{cases} \widehat{h}_{\mathcal{C}}(p', S) & \text{if } \beta \notin S \\ 1 + \widehat{h}_{\mathcal{C}}(p', S) & \text{otherwise;} \end{cases} & \textbf{R2.} \\
\widehat{h}_{\mathcal{C}}(p_1 + p_2, S) & = \max(\widehat{h}_{\mathcal{C}}(p_1, S), \widehat{h}_{\mathcal{C}}(p_2, S)) & \textbf{R3.} \\
\widehat{h}_{\mathcal{C}}(p_1 \mid p_2, S) & = \widehat{h}_{\mathcal{C}}(p_1, S) + \widehat{h}_{\mathcal{C}}(p_2, S) & \textbf{R4.} \\
\widehat{h}_{\mathcal{C}}(p' \backslash L, S) & = \widehat{h}_{\mathcal{C}}(p', S \cup L^{\circ}) & \textbf{R5.} \\
\widehat{h}_{\mathcal{C}}(x, S) & = 0 & \\
 & & \textbf{R6.}
\end{array}$$

the state space that may result when applying our methodology: we generate only four states, p, q, $0 \mid b.0$ and a.0 | 0, against the six states of the CWB-NC.

3.4.2. Heuristic Based on Communication Actions: $\widehat{h}_{com}$. The heuristic proposed in the previous section completely ignores the communication actions. In this section we present the function $\widehat{h}_{com}$ that considers also communication actions. In fact, the parallel composition in conjunction with the restriction is another cause of a nonequivalence. $\widehat{h}_{com}$ uses a further auxiliary function with respect to $\widehat{h}_{nc}$.

—$\widehat{h}_{\mathcal{C}}$, defined in Table VI, counts the number of possible communication actions (i.e., actions contained in some restriction set) that a process can perform. This function is similar to $\widehat{h}_{\mathcal{V}}$ except for *Rule R2* and *Rule R5*. In fact, when applied to $\beta.p'$, if β is in S, that is, it is a restricted action, the function is recursively applied on p', increasing by 1 its value; otherwise it is applied to p'. Moreover, when a restriction operator is found, S is always enlarged by the restricted actions (both for the strong and the weak equivalence).

Also the definition of $\Xi(p, q)$ is slightly modified.

$$\Xi(p, q) = \begin{cases} \widehat{h}_{\mathcal{V}}(p, S) - \widehat{h}_{\mathcal{V}}(q, S) & \text{if } \widehat{h}_{\mathcal{V}}(p, S) \neq \widehat{h}_{\mathcal{V}}(q, S), \\ -(\widehat{h}_{\mathcal{C}}(p, S) - \widehat{h}_{\mathcal{C}}(q, S)) & \text{otherwise} \end{cases}$$

This modification is due to the fact that, when p and q have the same number of visible actions, then we consider also the restricted actions and move first the node that has a less difference between the communication capability of the processes in that node (then the sign of the difference is inverted). In this way, we can more quickly obtain a failure, when a visible action is finally reached that the other process cannot perform.

To clarify this situation, consider the nodes

$$n_1 = \langle p, q, \perp, 1\rangle$$
$$n_2 = \langle z, q, \perp, 1\rangle$$

with

$$\begin{aligned} p &= (c_1.c_2.c_3.b.0 \,|(\bar{c}_1.\bar{c}_2.\bar{c}_3.0)\backslash\{c_1, c_2, c_3\}, \\ q &= (c_1.a.0 \,|\, \bar{c}_1.0)\backslash\{c_1\}; \qquad \text{and} \\ z &= (c_1.c_2.d.0 \,|\, \bar{c}_1.\bar{c}_2.0)\backslash\{c_1, c_2\}. \end{aligned}$$

It turns out that $\widehat{h}_{nc}(n_1) = \widehat{h}_{nc}(n_2) = 0$, since the number of visible actions is the same in all processes. Now, we can put on the field $\widehat{h}_C$ to obtain a different value for the nodes.

We formally define the heuristic function $\widehat{h}_{com}$ over nodes.

Definition 3.13 (*$\widehat{h}_{com}$: Heuristic Function*). Let $n = \langle p, q, \gamma, u\rangle$ be a node. We define $\widehat{h}_{com}(n)$ as

$$\widehat{h}_{com}(n) = \begin{cases} |\Xi(p, q)| & \text{if } \gamma = \top, \\ \Xi(p, q) & \text{if } \gamma = \perp \text{ and } u = 1, \\ \Xi(q, p) & \text{if } \gamma = \perp \text{ and } u = 2, \\ \widehat{h}_\alpha(p, S) + \Xi(p, q) & \text{if } \gamma = \alpha \text{ and } u = 1, \\ \widehat{h}_\alpha(q, S) + \Xi(q, p) & \text{if } \gamma = \alpha \text{ and } u = 2. \end{cases}$$

We obtain $\widehat{h}_{com}(n_1) = -4$ and $\widehat{h}_{com}(n_2) = -2$ and, if we expand first n_2 (the node with a lesser difference in the number of communication capabilities between the examined processes), we will find that, after a communications, q can perform action a and z cannot. Therefore, essentially $\widehat{h}_{com}$ behaves as $\widehat{h}_{nc}$, since the aim of both functions is to reach as quickly as possible the first visible action able to distinguish the two processes.

3.5. 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, Bouali et al. [1994] and Cleaveland and Sims [1996]) 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 [Madelaine and Vergamini 1990].

This means that these environments do not accept processes as the following two

$$x \overset{\text{def}}{=} (a.0 \,|\, c.x) + c.d.0$$
$$y \overset{\text{def}}{=} (a.0 \,|\, c.y) + c.e.0$$

since their standard transition systems are infinite. Nevertheless, for some kind of processes (similar to the previous two) with infinite transition systems, equivalence checking is decidable [Christensen et al. 1995]. We can apply our approach to the preceding processes and we can reach two nonbisimilar states that are on the finite path, and it turns out that after having generated 17 states, we can prove that the two processes are not equivalent. Other approaches for verification of infinite systems can be found in Kucera and Jancar [2006] and Chen et al. [2007].

4. EXPERIMENTAL RESULTS

In this section we present and discuss our experience using the tool, called GreASE (Greedy Algorithm for System Equivalence), implementing the presented approach to check strong and weak equivalence of several well-known CCS processes. The aim is to evaluate the performance of the approach presented in Section 3 and compare it against other formal verification tools for process algebras. Experiments were executed on a 64-bit, 2.67 GHz Intel i5 CPU equipped with 8 GB of RAM and running Gentoo Linux. The tool is freely available for download at the url `http://www2.ing.unipi.it/∼a080224/grease/`.

For the evaluation, a sample of well-known systems was selected from the literature. In all examples we check that both the strong and weak equivalence between two processes, namely p and q, is proved: typically, p describes the implementation of the system under consideration, while q describes the specification of the expected behavior. For all the following examples it holds that both $p \not\sim q$ and $p \not\approx q$.

Grid. This example is taken from Bradfield and Stirling [1990] and considers two processes on a grid 4×4 of relay stations which allow them to communicate.

Railway system (Crail). We applied our tool to the system specification given in Bruns [1992]. This system describes the British Rail's Solid State Interlocking which is devoted "to adjust, at the request of the signal operator, the setting of signal and points in the railway to permit the safe passage of trains".

Diva. This is a video-on-demand distributed application developed at the University of Naples, called *DIstributed Video Architecture* (DIVA), and it can be operated both in a WAN or a LAN scenario. A CCS specification can be found in Mazzocca et al. [2002].

CM-ASE. Context Management Application Service Element (CM-ASE) is a model of the application layer of the Aeronautical Telecommunications Network, developed by Gurov and Kapron [1998].

Mutual8 and Mutual10. p is a system handling the requests of a resource shared by 8 (respectively, 10) processes. It presents two alternative choices between a server based on a round-robin scheduling and a server based on mutual exclusion. q is similar to p with the round-robin scheduling changed.

Dining philosophers. We consider several instances of the dining philosophers example, $Phil_i$ where i is the number of the philosophers. 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 Bruns [1993], 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 sitting, 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.

The greedy algorithm employs a strategy to choose the tip node of the current psg to be expanded first (step (b) in Table I). We have experimented with three different strategies. The simplest strategy, called GREEDY_NONE, always chooses the tip node with the highest value for the heuristic. The other two strategies, called GREEDY_AND and GREEDY_OR, give different priorities to AND nodes and OR nodes by segregating them in two separate lists. Each list is sorted using the heuristic function, so that nodes with highest values are chosen first. However, the GREEDY_AND strategy always prefers the list of AND nodes and uses the list of OR nodes only when the former is empty, while the GREEDY_OR strategy does the opposite. The rationale of the GREEDY_AND strategy is that AND nodes can be marked as FAILED if any single one of their successors is marked as FAILED (first case of step (ii) in Table I). Instead, we are entitled to mark an OR node as FAILED only if we have already tried all of its successors (second

Table VII. Results for the Analyzed Systems (Weak Equivalence), Comparing Three Greedy Strategies

	GREEDY_AND		GREEDY_OR		GREEDY_NONE	
case study	gen	time	gen	time	gen	time
Grid	14320	1.221	14320	2.001	14320	1.945
Crail	292	0.195	2814	24.572	2315	7.948
Diva	242	0.033	235	0.025	236	0.026
CM-ASE	205	0.044	417	0.147	377	0.114
Mutual8	508	0.067	1091	0.190	1091	0.212
Mutual10	1020	0.183	40389	205.409	40389	194.313
$Phil_2$	88	0.060	127	0.026	127	0.025
$Phil_4$	496	0.100	1061	0.548	1061	0.547
$Phil_6$	1421	0.410	3132	2.589	3132	2.591
$Phil_8$	4940	8.270	6689	8.016	6689	8.026
$Phil_{12}$	13942	47.820	20203	44.777	20203	44.605

case of step (ii) in Table I). Thus, if we expect the AND/OR graph to have no solutions, the algorithm may confirm our expectation faster if we expand AND nodes first. The GREEDY_OR strategy is included for comparison. To make a choice among these strategies, we used all of them to check weak equivalence on the sample of concurrent systems using $\widehat{h}_{nc}$. The results are reported in Table VII and show that GREEDY_AND has the best performance, while GREEDY_OR and GREEDY_NONE are comparable. Note that GREEDY_AND greatly outperforms GREEDY_OR for Mutual10, while for the dining philosophers GREEDY_AND is still better than GREEDY_OR, but by a much slighter margin. This is mainly due to the structure of the CCS processes for the philosophers that are modeled with a high level of symmetry, thus producing tip nodes with the same value of the heuristic function and losing the chance to move towards the most promising path. In the following we always use the GREEDY_AND strategy.

The results obtained by GreASE are validated by comparing them to those obtained by two state-of-the-art tools for process algebras: the Construction and Analysis of Distributed Processes (CADP) tool [Fernandez et al. 1996] and the Concurrency Work-Bench of New Century tool (CWB-NC) [Cleaveland and Sims 1996]. CADP has been chosen as reference because it is a mature tool which is regularly improved. In fact, for what concerns equivalence checking, CADP offers BISIMULATOR which is an equivalence checker that takes as input two graphs to be compared (one represented implicitly using the OPEN/CAESAR environment, the other represented explicitly as a BCG file) and determines whether they are equivalent (modulo a given equivalence relation). BISIMULATOR works on-the-fly, meaning that only those parts of the implicit graph pertinent to verification are explored. The behavior of our algorithm is very related to on-the-fly equivalence-checking algorithms. However, our approach adds the guidance to the search using heuristic functions. Adding heuristics is more simple and natural using AND/OR graphs, widely used in artificial intelligence.

We have evaluated both GreASE performance, that is, the speed at which the tool returns its results (Table VIII) and GreASE scalability, namely, the extent to which the tool can manage increasingly large systems (Table IX). The goodness of the proposed heuristic functions is estimated by comparing the results of the greedy algorithm exploiting both $\widehat{h}_{nc}$ and $\widehat{h}_{com}$ against those obtained using a function, called $\widehat{h}_{rnd}$, that randomly chooses the tip node for the expansion. In all the experiments, 200 independent runs were executed to get statistically significant values. We report the average of the results (column “avg”) and the standard deviation (the column “stddev%”). In Table VIII, the running time is measured in seconds, while in Table IX the second,

Table VIII. Time Results

case study	GreASE with $\widehat{h}_{rnd}$ avg	stddev%	GreASE with $\widehat{h}_{nc}$	GreASE with $\widehat{h}_{com}$	CWB-NC	CADP
WEAK EQUIVALENCE						
Grid	0.53	1.85	1.221	0.727	17.836	2.40
Crail	50.17	85.50	0.195	204.758	2.349	2.25
Diva	0.02	22.70	0.033	0.048	0.071	2.41
CM-ASE	0.04	37.04	0.044	0.087	0.310	2.76
Mutual8	0.02	27.12	0.067	0.022	6.639	3.86
Mutual10	3.11	614.41	0.183	0.204	68.526	13.81
$Phil_2$	0.02	24.94	0.060	0.031	0.0618	1.71
$Phil_4$	0.32	102.01	0.100	0.088	4.817	4.14
$Phil_6$	0.32	161.14	0.410	0.183	-	7.98
$Phil_8$	0.56	197.37	8.270	0.319	-	103.45
$Phil_{12}$	1.11	201.82	47.820	0.712	-	566.32
STRONG EQUIVALENCE						
Grid	0.01	17.88	0.004	0.008	8.589	2.04
Crail	0.01	72.41	0.008	0.008	1.086	1.98
Diva	0.01	21.08	0.015	0.026	0.061	1.05
CM-ASE	0.01	15.02	0.011	0.017	0.221	1.03
Mutual8	0.01	43.57	0.008	0.016	3.804	1.05
Mutual10	0.29	27.46	0.055	0.18	37.675	1.71
$Phil_2$	0.01	28.74	0.004	0.007	0.012	1.56
$Phil_4$	0.01	26.63	0.010	0.008	1.564	1.87
$Phil_6$	0.01	67.21	0.027	0.012	3015.472	2.45
$Phil_8$	0.02	84.78	0.065	0.012	-	3.56
$Phil_{12}$	0.07	102.49	0.252	0.017	-	23.64

third, and forth columns show the number of states generated using $\widehat{h}_{rnd}$, $\widehat{h}_{nc}$ and $\widehat{h}_{com}$, respectively. The fifth column shows the number of nodes generated by the CWB-NC, the last column shows the "size" of the standard transition systems of the processes p and q; the experiments show that the uninformative function $\widehat{h}_{rnd}$ has the advantage of a lower computational cost, but is less effective. In this table the states generated by using CADP are not reported, as this information is not an output of the tool, but evaluating memory consumption we can conclude that our tool and CADP are comparable with respect to memory consumption.

We may see that our approach produces a significant reduction of the state space with respect to the other approaches. In fact, for $Phil_6$ the CWB-NC was not able to give an answer, while we managed to check the strong equivalence up to a configuration of 12 philosophers. Moreover, GreASE has also a better runtime performance with respect to CADP, even when memory consumption is comparable; on the contrary, the greedy algorithm employing either $\widehat{h}_{nc}$ or $\widehat{h}_{com}$ causes a far better memory usage than that caused by the CWB-NC. Finally, we note that $\widehat{h}_{com}$ behaves as $\widehat{h}_{nc}$, when the nodes to compare have different numbers of visible actions at each step of the algorithm; alternatively it is possible that the two processes of each tip node always have the same number of restricted actions; in both cases, the use of $\widehat{h}_{com}$ does not produce a great effect.

As shown in the experiments, our approach has a better performance when strong equivalence is checked with respect to weak equivalence; this occurs since strong equivalence considers all actions (including in each communication), while weak equivalence ignores all the actions for communication.

Table IX. State-Space Results

	GreASE with $\widehat{h}_{rnd}$					size	
case study	avg	stddev%	GreASE with $\widehat{h}_{nc}$	GreASE with $\widehat{h}_{com}$	CWB-NC	p	q
WEAK EQUIVALENCE							
Grid	14320.00	0.00	14320	14320	14320	14318	2
Crail	2516.53	42.94	292	3019	4345	3628	717
Diva	247.14	13.78	242	244	393	225	168
CM-ASE	346.25	25.47	205	298	1260	1259	791
Mutual8	427.53	47.53	508	216	8704	6912	6912
Mutual10	18303.60	36.26	1020	1322	54272	32768	32768
Phil$_2$	155.16	2.36	88	103	156	74	82
Phil$_4$	2589.44	62.71	496	354	7212	5510	1702
Phil$_6$	3909.16	129.58	1421	648	-	-	-
Phil$_8$	5226.07	161.48	4940	878	-	-	-
Phil$_{12}$	6059.27	170.21	13942	1338	-	-	-
STRONG EQUIVALENCE							
Grid	3.48	14.39	4	4	14320	14318	2
Crail	259.93	38.38	130	67	4345	3628	717
Diva	234.13	15.37	228	230	393	225	168
CM-ASE	126.43	22.11	121	93	1260	1259	791
Mutual8	572.00	39.33	181	200	8704	6912	6912
Mutual10	12680.00	21.00	695	2614	54272	32768	32768
Phil$_2$	34.97	21.85	40	26	156	74	82
Phil$_4$	169.49	60.50	139	53	7212	5510	1702
Phil$_6$	472.85	79.08	307	77	454197	409241	44956
Phil$_8$	1122.38	96.27	563	101	-	-	-
Phil$_{12}$	3148.59	108.21	1403	149	-	-	-

In light of this, GreASE obtains good results (sometimes very good results) when compared with CWB-NC and CADP, both in performance and in scalability. Nevertheless, we might observe that, in the Grid example, for weak equivalence no reduction is obtained, since all states have to be generated. However, we gained in time.

Table X analyses the goodness of our heuristic functions comparing them with the random heuristic function from the point of view of the generated graphs and the amount of backtracking needed to reach the goal. In particular, column "nodes" represents the number of nodes of the AND/OR graph, while column "cycles" represents the number of cycles of our greedy algorithm (i.e., point (2) of Table I). Finally, column "bt" represents the percentage of cycles that caused a backtracking. More precisely, it refers to point (2)(d)(iii) of Table I, namely, when an expanded node has been marked has FAILED.

As shown by Table X, both $\widehat{h}_{nc}$ and $\widehat{h}_{com}$ are more efficient with respect to $\widehat{h}_{rnd}$. In fact, they generate an AND/OR graph almost always smaller than that generated using $\widehat{h}_{rnd}$, with a lower amount of backtracking.

5. RELATED WORK

The most challenging task when applying automated model checking in practice is to conquer the state explosion problem. Hence, equivalence algorithms with minimal space complexity are of particular interest. Two algorithmic families can be considered to perform the equivalence checking. The first one is based on the 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 [1987]). The

Table X. AND/OR Graph Results

case study	GreASE with $\widehat{h}_{rnd}$				GreASE with $\widehat{h}_{nc}$			GreASE with $\widehat{h}_{com}$		
	nodes									
	avg	stddev%	cycles	bt%	nodes	cycles	bt%	nodes	cycles	bt%
WEAK EQUIVALENCE										
Grid	14436.36	0.08	13.66	7.32	14492	60	1.67	14492	60	1.67
Crail	265786.15	69.04	27834.26	23.21	1561	907	2.21	390423	76254	10.28
Diva	2027.02	19.00	1108.36	7.37	2388	2022	5.09	2319	1857	6.14
CM-ASE	4964.61	49.12	1120.90	19.64	2653	992	12.80	7252	2165	21.85
Mutual8	970.14	64.36	164.88	11.11	5221	778	23.91	598	112	7.14
Mutual10	111629.81	132.63	23743.75	5.47	11216	1986	16.21	15156	3337	13.28
Phil_2	2737.59	16.33	1404.37	5.28	1234	644	8.85	1816	1071	8.68
Phil_4	29468.51	96.88	6447.18	13.55	15151	2216	13.27	4425	1585	9.78
Phil_6	16348.20	162.31	2138.39	14.83	12450	1500	2.40	6640	1147	8.28
Phil_8	16232.67	200.65	1656.84	14.63	68288	3623	8.23	8400	1174	9.11
Phil_{12}	14220.47	203.95	969.62	14.04	164282	5110	5.83	12092	1210	8.84
STRONG EQUIVALENCE										
Grid	4.48	11.18	2.00	50.00	5	2	50.00	5	2	50.00
Crail	1967.19	75.25	709.99	6.21	342	110	2.73	138	28	14.29
Diva	690.53	20.53	562.57	1.10	679	624	1.76	690	630	1.75
CM-ASE	345.43	28.72	195.90	6.11	369	251	9.16	305	234	8.97
Mutual8	1170.54	51.91	408.59	0.24	262	97	1.03	288	106	0.94
Mutual10	54313.31	26.46	22215.35	0.00	1811	1383	0.07	10116	8822	0.01
Phil_2	71.55	34.20	39.13	6.98	88	45	4.44	48	27	7.41
Phil_4	328.19	80.95	123.50	7.21	296	102	1.96	80	29	10.34
Phil_6	777.17	91.93	213.85	6.62	673	179	1.12	106	30	10.00
Phil_8	1710.65	108.05	375.76	6.59	1270	276	0.72	132	30	10.00
Phil_{12}	4287.44	114.47	669.26	6.03	3316	530	0.38	184	30	10.00

other family of algorithms is based on a Cartesian product traversal from the initial state [Fernandez and Mounier 1991; Godskesen et al. 1989]. 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 [Fernandez 1989; Kanellakis and Smolka 1990], 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 Fernandez et al. [1993], where symbolic representation of the state space is used. In Bouajjani et al. [1990] an algorithm is presented 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 by using a 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 [2003] and by Gentilini et al. [2003]. 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. When two processes do not bisimulate, 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 [1998] is one of the original ones. They enhance the bug-finding capability of a model checker by using a heuristics to search the states that are most likely to lead to an error. In Godefroid and Khurshid [2002] genetic algorithms are used to exploit heuristics for guiding a search in large state spaces towards errors like deadlocks and assertion violations. In Behrmann and Fehnker [2001] heuristics have been used for real-time model checking in UPPAAL. In Alur and Wang [1999] and Möller and Alur [2001] heuristic search has been combined with on-the-fly techniques, and in Edelkamp and Reffel [1998] and Jensen et al. [2002] with symbolic model checking. Other works, for example, Edelkamp et al. [2001] and Groce and Visser [2002], used the heuristic search to accelerate finding errors, while in Santone [2003] it is used to accelerate verification. Many other works exist in symbolic execution on sequential programs to verify code equivalence, see, for example, Ramos and Engler [2011] and Dwyer et al. [2006].

Here we follow the road of using a heuristics in the analysis of AND/OR graphs trying to reduce the state explosion, both using on-fly-techniques and exploiting information on the processes to generate as quickly as possible a counterexample. A similar method, used to reduce the state explosion problem in the analysis of graphs representing concurrent systems, is the Ant Colony Optimization (ACO) [Dorigo and Stuetzle 2004]. The advantage over our method derives from the fact that ants explore different parts of the graph simultaneously and find optimal or near-optimal solutions (i.e., the minimal counterexample). The disadvantage is that the application of ACO is more expensive than greedy methods. For example, in Francesca et al. [2011] the authors proposed the use of ACO to reduce the state explosion that arises when finding deadlocks in complex networks described using Calculus of Communicating Systems (CCS). The analysis of the results obtained with ACO in this case shows that ACO outperforms A* and BFS algorithms, but not the greedy search. In fact, ACO returns better results than greedy in finding the (near) minimal-length path, but finding a short counterexample and finding it quickly are conflicting goals, so we need a trade-off between the advantage of the shortest counterexample and the cost to compute it. Sometimes the optimality of the solution is not worth the loss in efficiency of the search and here we focus mainly on the efficiency.

6. CONCLUSION AND FUTURE WORK

A method that uses heuristic searches has been proposed for equivalence checking for concurrent systems described in CCS. As far as we know, this is the first attempt to exploit process-algebra-based heuristics for equivalence checking in concurrent systems. The novel contributions of the article are:

—application of heuristic search for checking different equivalence relations (both strong and weak equivalence checking);
—defining heuristic functions that are simple and easy to calculate;
—proposing a new greedy algorithm with three different strategies for selecting nodes; and
—providing an experimental evaluation of our approach. As reflected in the experiments we made, the good scalability of our approach has been reached since we have neglected the optimality. Therefore, the efficiency is gained using both a greedy algorithm and heuristic functions.

As a future work we intend apply this approach to other equivalences, such as the ρ-equivalence, introduced in Barbuti et al. [1999], that formally characterizes 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 heuristic functions both more accurate and related to different specification languages. Further, it is our aim to integrate our approach also in other verification environments, for example, CADP [Fernandez et al. 1996].

REFERENCES

Rajeev Alur and Bow-Yaw Wang. 1999. “Next” heuristic for on-the-fly model checking. In *Proceedings of the 10th International Conference on Concurrency (CONCUR'99)*. Jos C. M. Baeten and Sjouke Mauw, Eds., Lecture Notes in Computer Science, vol. 1664, Springer, 98–113.

Roberto Barbuti, Nicoletta De Francesco, Antonella Santone, and Gigliola Vaglini. 1999. Selective mu-calculus and formula-based equivalence of transition systems. *J. Comput. Syst. Sci.* 59, 537–556.

Gerd Behrmann and Ansgar Fehnker. 2001. Efficient guiding towards cost-optimality in uppaal. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*. Tiziana Margaria and Wang Yi, Eds., Lecture Notes in Computer Science, vol. 2031, Springer, 174–188.

Ahmed Bouajjani, Jean-Claude Fernandez, and Nicolas Halbwachs. 1990. Minimal model generation. In *Proceedings of the 2nd International Workshop on Computer Aided Verification (CAV'90)*. Lecture Notes in Computer Science, vol. 531, Springer, 197–203.

Amar Bouali, Stefania Gnesi, and Salvatore Larosa. 1994. JACK: Just another concurrency kit. The intergration projekt. *Bull. EATCS* 54, 207–223.

Julian C. Bradfield and Colin Stirling. 1990. Verifying temporal properties of processes. In *Proceedings of the Conference on Theories of Concurrency: Unification and Extension (CONCUR'90)*. Jos C. M. Baeten and Jan Willem Klop, Eds., Lecture Notes in Computer Science, vol. 458, Springer, 115–125.

Glenn Bruns. 1992. A case study in safety-critical design. In *Proceedings of the International Conference on Computer Aided Verification (CAV'92)*. Gregor von Bochmann and David K. Probst, Eds., Lecture Notes in Computer Science, vol. 663, Springer, 220–233.

Glenn Bruns. 1993. A practical technique for process abstraction. In *Proceedings of the International Conference on Concurrency Theory (CONCUR'93)*. Eike Best, Ed., Lecture Notes in Computer Science, vol. 715, Springer, 37–49.

Doron Bustan and Orna Grumberg. 2003. Simulation-based minimazation. *ACM Trans. Comput. Log.* 4, 2, 181–206.

Taolue Chen, Bas Ploeger, Jaco van de Pol, and Tim A. C. Willemse. 2007. Equivalence checking for infinite systems using parameterized boolean equation systems. In *Proceedings of the 18th International Conference on Concurrency Theory (CONCUR'07)*. 120–135.

Søren Christensen, Hans Huttel, and Colin Stirling. 1995. Bisimulation equivalence is decidable for all context-free processes. *Inf. Comput.* 121, 2, 143–148.

Edmund M. Clarke and E. Allen Emerson. 2008. Design and synthesis of synchronization skeletons using branching time temporal logic. In *25 Years of Model Checking*, Orna Grumberg and Helmut Veith, Eds., Lecture Notes in Computer Science, vol. 5000, Springer, 196–215.

Edmund M. Clarke, Orna Grumberg, and David E. Long. 1994. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.* 16, 5, 1512–1542.

Edmund M. Clarke, David E. Long, and Kenneth L. McMillan. 1989. Compositional model checking. In *Proceedings of the 4th Annual Symposium on Logic in Computer Science (LICS'89)*. IEEE Computer Society, 353–362.

Rance Cleaveland and Steve Sims. 1996. The ncsu concurrency workbench. In *Proceedings of the 8th International Conference on Computer Aided Verification (CAV'96)*. Lecture Notes in Computer Science, vol. 1102, Springer, 394–397.

Antonio Cuomo, Antonella Santone, and Umberto Villano. 2013. CD-form: A clone detector based on formal methods. *Sci. Comput. Program.* 1, 2013.

Nicoletta De Francesco, Antonella Santone, and Gigliola Vaglini. 1998. State space reduction by non-standard semantics for deadlock analysis. *Sci. Comput. Program.* 30, 3, 309–338.

Marco Dorigo and Thomas Stuetzle. 2004. *Ant Colony Optimization*. MIT Press.

Matthew B. Dwyer, Suzette Person, and Sebastian Elbaum. 2006. Controlling factors in evaluating path-sensitive error detection techniques. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT'06/FSE'06)*. ACM Press, New York, 92–104.

Stefan Edelkamp, Alberto Lluch-Lafuente, and Stefan Leue. 2001. Directed explicit model checking with hsf-spin. In *Proceedings of the 8th International SPIN Workshop (SPIN'01)*. Matthew B. Dwyer, Ed., Lecture Notes in Computer Science, vol. 2057, Springer, 57–79.

Stefan Edelkamp and Frank Reffel. 1998. OBDDs in heuristic search. In *Proceedings of the 22nd Annual German Conference on Artificial Intelligence: Advances in Artificial Intelligence (KI'98).* Otthein Herzog and Andreas Gunter, Eds., Lecture Notes in Computer Science, vol. 1504, Springer, 81–92.

Jean-Claude Fernandez. 1989. An implementation of an efficient algorithm for bisimulation equivalence. *Sci. Comput. Program.* 13, 1, 219–236.

Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Laurent Mounier, Radu Mateescul, and Mihaela Sighireanu. 1996. CADP - A protocol validation and verification toolbox. In *Proceedings of the 8th International Conference on Computer Aided Verification (CAV'96).* Lecture Notes in Computer Science, vol. 1102, Springer, 437–440.

Jean-Claude Fernandez, Alain Kerbrat, and Laurent Mounier. 1993. Symbolic equivalence checking. In *Proceedings of the 5th International Conference on Computer Aided Verification (CAV'93).* Lecture Notes in Computer Science, vol. 697, Springer, 85–96.

Jean-Claude Fernandez and Laurent Mounier. 1991. “On the fly” verification of behavioural equivalences and preorders. In *Proceedings of the 3rd International Workshop on Computer Aided Verification (CAV'91).* Lecture Notes in Computer Science, vol. 575, Springer, 181–191.

Gianpiero Francesca, Antonella Santone, Gigliola Vaglini, and Maria Luisa Villani. 2011. Ant colony optimization for deadlock detection in concurrent systems. In *Proceedings of the 35th Annual IEEE Computer Software and Applications Conference (COMPSAC'11).* 108–117.

Raffaella Gentilini, Carla Piazza, and Alberto Policriti. 2003. From bisimulation to simulation: Coarsest partition problems. *J. Autom. Reason.* 31, 1, 73–103.

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

Patrice Godefroid and Sarfraz Khurshid. 2002. Exploring very large state spaces using genetic algorithms. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02).* Joost-Pieter Katoen and Perdita Stevens, Eds., Lecture Notes in Computer Science, vol. 2280, Springer, 266–280.

Jens C. Godskesen, Kim G. Larsen, and Michael Zeeberg. 1989. *TAV - Tools for Automatic Verification: Users Manual.* `http://books.google.com/books?id=jiaJYgEACAAJ`.

Susanne Graf, Bernhard Steffen, and Gerald Luttgen. 1996. Compositional minimisation of finite state systems using interface specifications. *Formal Asp. Comput.* 8, 5, 607–616.

Alex Groce and Willem Visser. 2002. Heuristic model checking for java programs. In *Proceedings of the 9th International SPIN Workshop on Model Checking of Software (SPIN'02).* Dragan Bosnacki and Stefan Leue, Eds., Lecture Notes in Computer Science, vol. 2318, Springer, 242–245.

Dilian Gurov and Bruce Kapron. 1998. The context management application server element. `http://webhome.cs.uvic.ca/~bmkapron/ccs.html`.

Eric A. Hansen and Shlomo Zilberstein. 2001. LAO*: A heuristic search algorithm that finds solutions with loops. *Artif. Intell.* 129, 1–2, 35–62.

Claude Jard and Thierry Jeron. 1991. Bounded-memory algorithms for verification on-the-fly. In *Proceedings of the 3rd International Workshop on Computer Aided Verification (CAV'91).* Springer, 192–202.

Rune M. Jensen, Randal E. Bryant, and Manuela M. Veloso. 2002. SetA*: An efficient bdd-based heuristic search algorithm. In *Proceedings of the National Conference on Artificial Intelligence (AAAI/IAAI'02).* 668–673.

Paris C. Kanellakis and Scott A. Smolka. 1990. CCS expressions, finite state processes, and three problems of equivalence. *Inf. Comput.* 86, 1, 43–68.

Antonin Kucera and Petr Jancar. 2006. Equivalence-checking on infinite-state systems: Techniques and results. *Theory Pract. Logic Program.* 6, 3, 227–264.

Eric Madelaine and Didier Vergamini. 1990. Finiteness conditions and structural construction of automata for all process algebras. In *Proceedings of the 2nd International Workshop on Computer Aided Verification (CAV'90).* Lecture Notes in Computer Science, vol. 531, Springer, 353–363.

Ambuj Mahanti and Amitava Bagchi. 1985. AND/OR graph heuristic search methods. *J. ACM* 32, 1, 28–51.

Ambuj Mahanti, Supriyo Ghose, and Samir K. Sadhukhan. 2003. A framework for searching and/or graphs with cycles. CoRR cs.AI/0305001.

Nicola Mazzocca, Antonella Santone, Gigliola Vaglini, and Valeria Vittorini. 2002. Efficient model checking of properties of a distributed application: A multimedia case study. *Softw. Test. Verif. Reliab.* 12, 1, 3–21.

Kenneth L. McMillan. 1993. *Symbolic Model Checking.* Kluwer.

Robin Milner. 1989. *Communication and Concurrency.* Prentice Hall.

M. Oliver Moller and Rajeev Alur. 2001. Heuristics for hierarchical partitioning with application to model checking. In *Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct*

Hardware Design and Verification Methods (CHARME'01). Tiziana Margaria and Thomas F. Melham, Eds., Lecture Notes in Computer Science, vol. 2144, Springer, 71–85.

Robert Paige and Robert Endre Tarjan. 1987. Three partition refinement algorithms. *SIAM J. Comput.* 16, 6, 973–989.

Atanas N. Parashkevov and Jay Yantchev. 1996. ARC - A tool for efficient refinement and equivalence checking for csp. In *Proceedings of the IEEE International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP'96).* 68–75.

Judea Pearl. 1984. *Heuristics - Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

Doron Peled. 1993. All from one, one for all: On model checking using representatives. In *Proceedings of the 5th International Conference on Computer Aided Verification (CAV'93).* Lecture Notes in Computer Science, vol. 697, Springer, 409–423.

Jean-Pierre Queille and Joseph Sifakis. 1982. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*. Mariangiola Dezani-Ciancaglini and Ugo Montanari, Eds., Lecture Notes in Computer Science, vol. 137, Springer, 337–351.

David A. Ramos and Dawson R. Engler. 2011. Practical, low-effort equivalence verification of real code. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11).* 669–685.

Antonella Santone. 2002. Automatic verification of concurrent systems using a formula-based compositional approach. *Acta Inf.* 38, 8, 531–564.

Antonella Santone. 2003. Heuristic search + local model checking in selective mu-calculus. *IEEE Trans. Softw. Engin.* 29, 6, 510–523.

Antonella Santone and Gigliola Vaglini. 2012. Abstract reduction in directed model checking ccs processes. *Acta Inf.* 49, 5, 313–341.

Antonella Santone, Gigliola Vaglini, and Maria Luisa Villani. 2013. Incremental construction of systems: An efficient characterization of the lacking sub-system. *Sci. Comput. Program.* 78, 9, 1346–1367.

Colin Stirling and David Walker. 1991. Local model checking in the modal mu-calculus. *Theor. Comput. Sci.* 89, 1, 161–177.

Antti Valmari. 1992. A stubborn attack on state explosion. *Formal Meth. Syst. Des.* 1, 4, 297–322.

C. Han Yang and David L. Dill. 1998. Validation with guided search of the state space. In *Proceedings of the 35th Annual Design Automation Conference (DAC'98).* 599–604.

Received March 2013; revised September 2013; accepted December 2013