An Iterative Approach to Language Containment

Felice Balarin* Alberto L. Sangiovanni-Vincentelli

Department of Electrical Engineering and Computer Science
University of California, Berkeley, CA 94720

Abstract. We propose an iterative approach to formal verification by language containment. We start with some initial abstraction and then iteratively refine it, guided by the failure report from the verification tool. We show that the procedure will terminate, propose a series of heuristic aimed at reducing the size of BDD's used in the computation, and formulate several open problems that could improve efficiency of the procedure. Finally, we present and discuss some initial experimental results.

1 Introduction

The size of finite-state systems that can be verified by formal methods has increased significantly since the introduction of implicit state enumeration techniques based on fixed-point computations and binary decision diagrams (BDD's) [2, 5]. Still, many practical systems are out of reach of these methods. Moreover, the size of systems to be verified increases rapidly with the progress of VLSI technologies.

Many researchers and practitioners believe that one of the keys to managing such complex systems is the use of abstractions and reductions, where one tries to verify a complex systems by verifying its simplified version [6, 3].

We make a distinction between two kinds of simplifications: *exact* and *conservative*. Exact simplifications preserve all aspects of system behavior,[1] therefore the original system is verified *if and only if* the simplified system is. They can be applied to virtually any verification formalism, but it is often hard to find an exact simplification of reasonable size.

On the other hand, conservative simplification preserve enough information of the system behavior to guarantee that the original system is verified *if* the simplified system is. However, if the simplified system is not verified, the original system might or might not satisfy the required property. Conservative approximations exist only for some formalisms like language containment [6], simulation [1], $\forall CTL$ and $\forall CTL*$ model checking [3], but they can often lead to much

* Supported by SRC under grant # 93-DC-008.

[1] Here, we use the term "behavior" informally, since the exact meaning varies with verification formalisms.

larger reductions in size than exact simplification. We use the word *abstraction* and the phrase *conservative simplification* interchangeably.

The obvious problem is how to proceed when the abstracted system is not verified. In practice, the user would analyze a failure report from a verification tool to decide whether the failure is inherent in the original system, or the consequence of some oversimplification. If the latter applies, the user would usually modify the current abstraction of the system, resulting in a slightly more complex system and repeat the verification process. Kurshan [7] observed that it is often necessary to go through several of these iterations before the final decision can be made. No attempts have been made to automate this process, even though this significant designer involvement is seen as one of the obstacles to wider acceptance of formal verification techniques in the design community.

The main purpose of this paper is to describe one of the possible ways to automate this iterative process. Although ideas similar to those presented here could easily be applied to any formalism allowing abstractions, many details are dependent on the fact that inside an iteration loop we are using a language containment tool based on the theory of L-automata and implemented with BDD's. For example, in the case of tools based on explicit state enumeration, the complexity of the system is measured very well by the number of reachable states. That is not necessarily true in case of BDD-based tools. In fact, some of the heuristic abstractions described in the following sections result in a system with more reachable states, but smaller BDD representations.

The rest of this paper is organized as follows. In Sect. 2 we discuss previous results significant to our work. In Sect. 3 we propose a heuristic iterative verification algorithm and provide justifications for our choices of heuristics. Finally, we discuss initial experimental results in Sect. 4.

2 Preliminaries

We represent a system to be verified as a collection of n communicating finite-state subsystems called L-processes [6]. With every L-process (say P_i) we associate variables x_i (a present state variable) and y_i (a next state variable) both ranging over some finite non-empty set V_i (a set of states) and a variable o_i (an output variable) ranging over some other finite non-empty set O_i (a set of observables). Let r be a variable ranging over $D_r = \{true, false\}$ and let a variable $o = (o_1, \ldots, o_n)$ be a n-tuple containing all output variables, ranging over the set $O = O_1 \times \ldots \times O_n$. The L-process P_i can then be specified with Boolean functions $I_i(x_i)$ (a characteristic function of the initial states), $T_i(x_i, y_i, o, r)$ (a characteristic function of the transition relation, or just "transition relation") and a set of Boolean functions $Z_i = \{Z_i^1(x_i), \ldots, Z_i^{k_i}(x_i)\}$ (a set of cycle sets).

We interpret $I_i(x_i)$ and cycle sets in Z_i as sets of states, and we interpret transition relation as an edge labeled graph. Nodes of the graph are elements of the state space V_i and edge labels are elements of $O \times D_r$. If T_i is satisfied for $x_i = a$, $y_i = b$, $o = c$ and some value of r, we say that there exists an edge from a to b labeled with c. We require that for every valuation a, b and c of x_i, y_i

and o, the expression $T_i(a, b, c, false)$ is satisfied only if $T_i(a, b, c, true)$ is. Edges for which $T_i(a, b, c, true)$ is satisfied and $T_i(a, b, c, false)$ is not, are called *recur* edges.

A language of an L-process P_i (denoted by $\mathcal{L}(P_i)$) contains all infinite sequences of element of observable space O that have an *accepting* run. A run is said to be accepting if it starts in some of the initial states and it does not cross any of the recur edges infinitely often nor it remains in some of the cycle sets infinitely often.

Kurshan [6] has defined a product $\otimes$ of L-processes that satisfies:

$$\mathcal{L}(P_1 \otimes \ldots \otimes P_n) = \bigcap_{i=1}^{n} \mathcal{L}(P_i). \tag{1}$$

We think of the language as the possible behaviors of the system. The task of formal verification is to prove that all of these behaviors are acceptable, i.e. that the language of the system to be verified is contained in some other language that defines a set of acceptable behaviors. Kurshan [6] has shown that under some mild conditions, it is possible to reduce the language containment problem to the check of emptiness of the language of L-process $P = \bigotimes_{i=0}^{n} P_i$, where $P_1, \ldots, P_n$ are components of the system to be verified and P_0 (also called a *task*) is an L-process whose language defines a complement of acceptable behaviors. In the rest of this paper we will assume that x_i and y_i are present and next state variables of the process P_i and that $x = (x_0, x_1, \ldots, x_n)$ and $y = (y_0, y_1, \ldots, y_n)$ are present and next state variables of the process P. Obviously, the language emptiness problem lends itself to conservative simplifications. Any L-process R satisfying:

$$\mathcal{L}(P) \subseteq \mathcal{L}(R) \tag{2}$$

is an abstraction of P. The challenge is to find R which is small, satisfies (2) by construction, and yet close enough to P such that its language is empty if $\mathcal{L}(P)$ is.

One way of generating R that is likely to be[2] smaller than P and satisfies (2) is to compute partial product $\bigotimes_{i \in I} P_i$ for some $I \subset \{0, \ldots, n\}$. We say that processes in I are active and others are ignored. An equivalent interpretation of this abstraction is that we have replaced every process P_j, $j \notin I$ with a process that has the same states but unrestricted transition between any two states (transition relation equals *true*), all states designated as initial (characteristic function of initial states equals *true*) and no cycle sets. The following lemma states that this is indeed the conservative simplification.

Lemma 1. *Let $P_0, \ldots, P_n$ be L-processes and let $I \subset \{0, \ldots, n\}$. Then:*

$$\mathcal{L}(\bigotimes_{i=0}^{n} P_i) \subseteq \mathcal{L}(\bigotimes_{i \in I} P_i).$$

[2] In all examples we tried this did result in a smaller BDD. However, it is not true in general case. In fact, it is very easy to construct degenerate examples where this is not the case.

Proof. $\mathcal{L}(\bigotimes_{i=0}^{n} P_i) = \mathcal{L}(\bigotimes_{i\in I} P_i) \cap \mathcal{L}(\bigotimes_{i\in\{0,\ldots,n\}-I} P_i) \subseteq \mathcal{L}(\bigotimes_{i\in I} P_i)$. □

The second kind of abstractions we use is closely related to the language containment algorithm, which we now describe only in as much details as necessary for this paper. The language containment algorithm can be divided into three main steps:

1. *Computing the product*, i.e. in this step we compute $I(x) = \prod_{i=0}^{n} I_i(x_i)$, $T(x,y,o,r) = \prod_{i=0}^{n} T_i(x_i,y_i,o,r)$ and $Z = \bigcup_{i=0}^{n} Z_i$.
2. *Removing the outputs*, i.e. in this step we compute:

$$G(x,y,r) = \exists o : T(x,y,o,r). \tag{3}$$

   We say that $G(x,y,r)$ is the *graph* of the system. We can interpret this operation as removing labels form the edges in graph defined by $T(x,y,o,r)$.
3. *Search for a bad run*, i.e. in this step we search the graph of the system $G(x,y,r)$ for an accepting run. If such a run exists the language is not empty and verification tools usually report one such run.

The following proposition states another sufficient condition for a conservative approximation.

Proposition 2. *Let $G(x,y,r)$ be a graph of the system, and let $H(x,y,r)$ be some Boolean function that covers G. Then, the absence of a bad cycle in the graph interpretation of $H(x,y,r)$ implies the absence of a bad cycle in $G(x,y,r)$.*

Proof. Since $H(x,y,r)$ covers $G(x,y,r)$ every edge in G appears also in H, and every non-recur edge in G is also a non-recur edge in H. Therefore, any bad cycle in G (i.e. a cycle not containing any of the recur edges and not contained in any of the cycle sets) exists and is also bad in H, and any path from some of the initial states to such a cycle in G is also a path in H. □

The following result is a well known fact in Boolean theory, so we state it here without a proof. It offers a convenient way to compute an abstraction satisfying the condition in Proposition 2.

Lemma 3. *Let $G(x,y,r)$ be as defined by (3) and let:*

$$H(x,y,r) = \prod_{i=0}^{n} (\exists o : T_i(x_i,y_i,o,r)). \tag{4}$$

Then, $H(x,y,r)$ covers $G(x,y,r)$.

We can interpret this abstraction as ignoring the communication between subsystems, since we remove labels from edges before checking whether they can be simultaneously satisfied. The heuristic argument for computing (4) instead of (3) is that all the intermediate results in (4) have fewer variables in its

support than the intermediate result $(\prod_{i=0}^{n} T_i(x_i, y_i, o, r))$ in (3). This is likely to reduce the size of the intermediate BDD's. In fact the intermediate result $(\prod_{i=0}^{n} T_i(x_i, y_i, o, r))$ is usually the single largest BDD created throughout the language containment algorithm.

The second heuristic argument follows from the well known fact that the BDD size of the product of two Boolean functions is bounded from above by the product of the sizes of BDD's representing each function, but if they have disjoint supports under appropriate ordering the bound can be strengthen to the sum of the sizes of BDD's. This suggests that the BDD representing (4) is likely to be smaller than BDD representing (3), since intermediate results $(\exists o : T_i(x_i, y_i, o, r))$ in (4) have only r as a common support.

As is the case with the other proposed abstractions, these heuristic arguments do not hold in general, and counter-examples are easily constructed. However, our (admittedly limited) experience supports them without exceptions.

It is interesting to note that both of these abstractions result in a graph with the same number of nodes as the original one, but with more edges, therefore possibly more reachable states. So, these abstractions are not at all suitable for a verification tool based on explicit state enumeration, but they should result in a smaller BDD representation in most cases.

3 Verification Algorithm

Our verification algorithm is a special case of the following four-step general iterative approach to formal verification:

Initial abstraction: Choose an initial abstraction.

Verification: Try to verify the task. If the verification is successful, terminate with success. Otherwise, go to the next step.

Failure analysis: Analyze the failure report from the verification tool and determine whether the failure is inherent in the original system or the failure is due to the oversimplification. If the former is true, terminate with failure. If the latter is true, go to the next step.

Refinement: Refine the abstraction in a way that a reported failure is eliminated. Go to the verification step.

This general procedure is applicable to any formal verification techniques that allows conservative simplifications. The specific algorithm will depend on the technique, but also on the heuristic choices of initial abstraction, failure report and refinement procedures.

The algorithm we propose for iterative language containment (ILC algorithm from here on) is shown in Fig. 1. In the following paragraphs we describe and justify the choices we have made in that algorithm.

3.1 Initial Abstraction

Our choice of the initial abstraction is based on the observation that many of the interesting properties are expressed in terms of of the output variables of

input $P_0, \ldots, P_n$ /* L-processes, where $P_i = (I_i(x_i), T_i(x_i, y_i, o, r), Z_i)$ */
input Act /* a set containing indexes of the task and of all */
/* processes whose output variables are in the */
/* support of the task's transition relation */
begin
step 1: $I(x) = \prod_{i \in Act} I_i(x_i)$; $G(x,y,r) = \exists o : \prod_{i \in Act} T_i(x_i, y_i, o, r)$; $Z = \bigcup_{i \in Act} Z_i$;
while not STOP **do**
step 2: **if** "bad run" does not exist **then**
STOP /* the task is verified */
step 3: **else if** "bad edges" $e(x,y,r)$ exist **then**
step 4: $G(x,y,r) = G(x,y,r) * \overline{e(x,y,r)}$
step 5: **else if** there exists $New \subseteq \overline{Act}$ that eliminates the bad run **then**
step 6: $Act = Act \cup New$; $I(x) = I(x) * \prod_{k \in New} I_k(x_k)$; $Z = Z \cup \bigcup_{k \in New} Z_k$;
$G(x,y,r) = G(x,y,r) * \prod_{k \in New} (\exists o : T_k(x_k, y_k, o, r))$;
else
STOP /* the task is not verified */
end if
end while
end

Fig. 1. ILC – iterative language containment algorithm

only a few subsystems, and that for some of the properties the behavior of some subsystems is irrelevant. Therefore, as an initial abstraction, in step 1 of the ILC algorithm, we ignore all the subsystems that are not in the *Act* set, i.e. all except the task and those processes that have their output variables in the support of the task's transition relation. By choosing this initial abstraction and an appropriate refinement step, we will never consider a subsystem that is irrelevant to the property to be verified.

3.2 Verification

In step 2 of the ILC algorithm a verification procedure is performed. For this step we use a tool described in [5]. Similarly to COSPAN [4], in case of failure the tool reports one "bad run", i.e. an initialized sequence of states that does not cross any of the recur edges infinitely often, nor it remains forever in any of the cycle sets. A bad run can be represented by a final prefix, called a "bad path", and a final, but infinitely often repeated suffix called a "bad cycle". The choice of the cycle and the path is done by the tool and is basically arbitrary.

We stress the importance of the failure report for our algorithm. If the task is

not satisfied than there always (in any iteration) exists a failure report to show it. However, the tool might ignore that failure for many iterations, reporting instead "false" failures. At the moment, there are no heuristics (let alone exact procedures) which would guide the tool in the search of the "real" failure.

3.3 Failure Analysis

The task of the failure analysis step is determined by the failure report and the types of abstraction used. As indicated previously, there are two abstractions used: ignoring communications (step 6 of the ILC algorithm) and ignoring subsystems (step 1 of the ILC algorithm). In our algorithm we first analyze the former in step 3, and then, if no violations are found, we analyze the latter in step 5 of the ILC algorithm.

If the communication between subsystems is ignored while making a product the result is a graph with same nodes and more edges than the exact graph. Since all other information (initial states, recur edges and cycle sets) is exact one should only check that every edge in the report is the "real" edge, i.e. that the product of labels of all component edges is not *false*. Care should be taken not to consider recur edges while analyzing a bad cycle.

Let $e_{ik}(x_i, y_i, r)$ be a characteristic function of the i-th component of the k-th edge in the failure report, i.e. if the k-th edge is $((x_0)_k, \ldots, (x_n)_k) \longrightarrow ((x_0)_{k+1}, \ldots, (x_n)_{k+1})$ let:

$$e_{ik}(x_i, y_i, r) \stackrel{\text{def}}{=} (x_i = (x_i)_k) * (y_i = (x_i)_{k+1}) * F,$$

where F is *true* if the k-th edge is in a bad path, and F is $(r = false)$ if the k-th edge is in a bad cycle. The expression $(r = false)$ ensures that we will not consider any recur edges in the bad cycle. If we can find some k and some $I \subseteq Act$ such that:

$$lab_k(o) = \prod_{i \in I} \exists (x_i, y_i, r) : (T_i(x_i, y_i, o, r) * e_{ik}(x_i, y_i, r)), \tag{5}$$

evaluates to $false$, we can compute:

$$e(x, y, r) = \prod_{i \in I} e_{ik}(x_i, y_i, r), \tag{6}$$

a characteristic function of the set of "bad edges", as required in step 3 of the ILC algorithm. The justification for the name "bad edges" comes from the following proposition.

Proposition 4. *Let $I \subseteq Act$ be such that $lab_k(o)$, as defined by* (5)*, evaluates to false for some k, and let $e(x, y, r)$ be as defined by* (6)*. Then:*

a) *$e(x, y, r)$ intersects with the present graph of the system,*
b) *$e(x, y, r)$ does not intersect with the exact graph of the system.*

Proof.

a) If the k-th edge is in the bad path, $e(x, y, r)$ contains at least that edge, which must also be contained in the present graph of the system. If the k-th edge is in the bad cycle, $e(x, y, r)$ must at least intersect that edge, because the k-th edge must be non-recur, i.e. not dependent on r. Since that edge is contained in the present graph of the system, $e(x, y, r)$ intersects the present graph.

b) If the k-th edge is in the bad path, since $lab_k(o) = false$ there can be no edge in the exact graph of the system with components $(x_i)_k \rightarrow (x_i)_{k+1}$ $i \in I$, because no element of the observable space can satisfy all their label simultaneously. If the k-th edge is in the bad cycle, there can be no non-recur edge in the exact graph of the system with components $(x_i)_k \rightarrow (x_i)_{k+1}$ because no element of the observable space can satisfy all their label simultaneously. If there is a recur edge with these components, it is by definition covered by $(r = true)$, so $e(x, y, r)$ (which is covered by $(r = false)$) does not intersect it.

□

The proof of part a) shows not only that $e(x, y, r)$ intersects with the present graph of the system, but also with the failure report. Therefore, the reported bad run is no longer a run after the graph of the system is updated in step 4 of the ILC algorithm.

Proposition 4 shows that the criterion $lab_k(o) = false$ is correct in a sense that corresponding $e(x, y, r)$ never contains edges that are in the exact graph of the system. The following proposition shows that it is also complete, in a sense that if an I satisfying $lab_k(o) = false$ can not be found, the intersection of the languages of active processes is indeed not empty.

Proposition 5. *Let $I = Act$ in (5) and let a sequence $a = a_1, \ldots, a_k, \ldots$ (where $a_k \in O$), be such that $lab_k(a_k) \neq false$ for every k in a bad run. Then $a \in \mathcal{L}(\bigotimes_{i \in Act} P_i)$.*

Proof. It suffices to show that a bad run is a run of a_k's in $\bigotimes_{i \in Act} P_i$ and that a bad cycle does not contain any of the recur edges of active processes. Indeed, if the k-th edge is in the bad path:

$$lab_k(a_k) = \prod_{i \in I} \exists r : T_i((x_i)_k, (x_i)_{k+1}, a_k, r) \neq false,$$

implies that there exists an edge between $(x_i)_k$ and $(x_i)_{k+1}$ labeled with a_k, which in turn implies that a bad path is a (finite) prefix of the run of a. Similarly, if the k-th edge is in the bad cycle:

$$lab_k(a_k) = \prod_{i \in I} T_i((x_i)_k, (x_i)_{k+1}, a_k, false)) \neq false,$$

implies that there exists a non-recur edge between $(x_i)_k$ and $(x_i)_{k+1}$ labeled with a_k. Thus, a bad cycle is an (infinitely often repeated) suffix of the run of a, which does not contain any recur edges. □

If this phase of failure analysis reveals no oversimplifications, we move to the next one in step 5 of the ILC algorithm. First, we construct a simple L-process X satisfying $\mathcal{L}(X) \subseteq \mathcal{L}(\bigotimes_{i \in Act} P_i)$. We set a graph of X to be exactly the graph of the reported bad run (i.e. "bad path" + "bad cycle"), and we set the label of the k-th edge to be $lab_k(o)$ as defined by (5).

Next, we try to find a subset New of ignored processes such that the intersection of $\mathcal{L}(\bigotimes_{i \in New} P_i)$ and $\mathcal{L}(X)$ is empty. If such a subset cannot be found, the intersection of languages of all processes is not empty, and we terminate the verification procedure with failure. Otherwise, we go to the refinement phase in step 6 of the ILC algorithm. To find a set New we can reuse language containment algorithm, but finding it can be as difficult as the original verification problem. We propose the following heuristics to avoid this complexity:[3]

1. first, try to find a single process P_k such that $\mathcal{L}(X) \cap \mathcal{L}(P_k) = \emptyset$,
2. if that fails, order ignored processes in a way that every process has at least one variable in common support either with X or with some process preceding it (in that order),
3. compute cumulative product $X \otimes P_1 \otimes \ldots$ adding one ignored process at a time in order chosen in the previous step,
4. repeat the previous step until one of the following conditions is satisfied:
   (a) *the language of the cumulative product becomes empty*: in this case let New contain all ignored processes in the cumulative product,
   (b) *the cumulative product contains all ignored processes and its language is not empty*: in this case terminate with failure,
   (c) *the cumulative product is too big (i.e. occupied memory exceeds some given limit)*: in this case let New contain all ignored processes in the cumulative product (we defer eliminating the failure report until subsequent iterations).

3.4 Refinement

Depending on the results of failure analysis there are two different refinement problems: deleting certain edges from the current abstraction (step 4) and including a previously ignored subsystem into the current abstraction (step 6 of the ILC algorithm).

We delete some bad edges by executing step 4 of the ILC algorithm, where $e(x, y, r)$ is as defined in (6) and $I \subseteq Act$ in (6) is such that (5) evaluates to $false$. The smaller I is, more edges will be deleted, hence we will converge faster towards the exact graph.

We propose a two-step method of finding a suitable I. First, we evaluate (5) iteratively for ever increasing subset of Act, starting with a singleton, adding one new element in each iteration, and stopping as soon as (5) evaluates to $false$. Say that this happens for some $I' \subseteq Act$. Then, we try all two-element sets $\{l, m\} \subseteq I'$. If some of those makes (5) $false$ we use it in (6), otherwise

[3] The choice of these heuristics were influenced by discussions with R.P. Kurshan.

we use I'. In principle, one could then try all triples, quadruples, etc., but we conjecture that it would actually increase total running time.

The second refinment task occurs when we find a subset New of ignored processes that eliminates the reported failure. In that case, we include New in the set of active processes, and update initial states, cycle sets and transition relation as indicated in step 6 of the ILC algorithm. This amounts to including processes in New in the current abstraction, but ignoring the communication between them and other active processes. This way, we only need to update the graph of the system from the previous iteration. Since this is also true for the other refinement procedure, we do not ever compute the full transition relation of the system.

On the other hand, the reported failure will not be eliminated in the updated system. This is important only in terms of efficiency, since the failure will eventually be eliminated in subsequent iterations, but their number may be large. To define a refinement step which retains some of the mentioned advantages, but also eliminates the reported failure is another interesting open problem.

3.5 Convergence and Correctness

In showing the correctness of the ILC algorithm we assume that the algorithm that searches for a bad run in step 2 is correct. Also, we assume that in step 6 a set New is not empty unless the failure report is valid despite the abstractions. In our implementation, for both of these steps, we use previously developed language containment algorithm.

Proposition 6. *The ILC algorithm is correct.*

Proof. To show that the algorithm can not terminate with a false success we need to show that at every point in the algorithm the description of the system is an abstraction of the exact system. Indeed:

1. by Lemma 1 $(I(x), G(x, y, r), Z)$ computed in step 1 is an abstraction of the exact system,
2. by Proposition 4b the updated graph of the system in step 4 contains the exact graph if the one before the update did so, hence by Proposition 2 it is an abstraction of the exact system,
3. by Lemma 3 and Proposition 2 $(I(x), G(x, y, r), Z)$ updated in step 6 are an abstraction of $\bigotimes_{i \in Act \cup New} P_i$ if before the update they were an abstraction of $\bigotimes_{i \in Act} P_i$, and by Lemma 1 $\bigotimes_{i \in Act \cup New} P_i$ is an abstraction of the exact system.

Now, we make an inductive argument on the number of iteration, using part 1 as a base case, and parts 2 and 3 as an inductive step.

To show that the algorithm can not terminate with a false failure we need to show that a failure is reported only if a sequence is found that is in the language of all the components $P_0, \ldots, P_n$. Indeed, a failure is reported only if step 4

results in a sequence a which by Proposition 5 is contained in the languages of all the active processes, and by the assumption of correctness of step 5, also in the language of all the ignored processes. □

Proposition 7. *The ILC algorithm terminates.*

Proof. At each iteration we either execute steps 4 or 6 or terminate. Step 6 can not be executed more than n times, because every time it is executed the set Act grows by one new element and $Act \subseteq \{0, \ldots, n\}$. By Proposition 4a updated $G(x, y, r)$ in step 4 contains at least one minterm less than the one before the update, and by Proposition 6 it always contains the exact graph of the system. Therefore, step 4 can also be executed only finitely many times. □

The proof of the Proposition 7 has an unfortunate consequence that the number of steps in the worst case is proportional to the possible number of edges in the exact graph which is exponential in the number of subsystems.

4 Experiments and Conclusions

We have tested our algorithm on two different properties of the well known Dining Philosopher's problem. We have used the solution with an encyclopedia [8] to insure that the system is deadlock and starvation free. All experiments were performed on a 400Mb DEC 5000 workstation.

The first property we have verified is that two neighboring philosophers will never eat at the same time (mutual exclusion). More precisely we have verified this property for the first two philosophers. The results are summarized in Table 1. We could not verify any larger examples due to the memory limit. No comparison is given to the direct approach since it can verify systems with at most several hundred philosophers. For any number of philosophers the algorithm verified the property in one iteration.

Table 1. Results for the mutual exclusion property

philosophers	2,000	4,000	6,000	8,000	10,000	12,000	14,000
reachable states	10^{953}	10^{1908}	10^{2862}	10^{3816}	10^{4771}	10^{5618}	10^{6595}
CPU time [sec]	42.8	160.4	359.1	627.8	981.2	1346.6	1854.4

The other property we have verified is that the first philosopher will not be hungry forever (starvation). Although this property is expressed in terms of outputs of only one philosopher, it is not a local property. In fact, some aspects of the behavior of all philosophers must be included to verify this property. In this case results were not nearly as good as for the mutual exclusion method. In

fact, it performed worse than the direct method with the number of iterations growing rapidly with the number of philosophers.

To summarize, we have presented a general iterative approach to language containment problem and proposed several specific heuristic aimed at reducing the size of BDD's used in computation. We have examined strengths and weaknesses of our approach and formulated several open problems. Our procedure is completely automatic and very efficient for some classes of problems.

Although the proposed heuristics are capable of dealing with some very large systems, they are by no means the definite answer to the state explosion problem. Rather, we envision them as one of the tools available to deal with real-life verification problems. Other tools might include different abstractions, exploration of symmetry and induction. But, we believe that our proposed ideas of automatic failure analysis and automatic modifications will play an important role in any of these approaches.

References

1. S. Bensalem, A. Boujjani, C. Loiseaux, and J. Sifakis. Property preserving simulations. In *Proceeding of the Fourth Workshop on Computer-Aided Verification (CAV '92)*, June 1992.
2. J. R. Burch, Edmund M. Clarke, K. L. McMillan, and David L. Dill. Sequential circuit verification using symbolic model checking. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, 1990.
3. Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. In *Proc. Principles of Programming Languages*, January 1992.
4. Z. Har'El and R. P. Kurshan. Software for analysis of coordination. In *Proceedings of the International Conference on System Science*, pages 382–385, 1988.
5. Ramin Hojati, Herve Touati, R. P. Kurshan, and Robert K. Brayton. Efficient ω-regular language containment. In *Proceeding of the Fourth Workshop on Computer-Aided Verification (CAV '92)*, June 1992.
6. R. P. Kurshan. Analysis of discrete event coordination. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, pages 414–453. Springer-Verlag, 1990. LNCS vol. 430.
7. R. P. Kurshan, 1991. private communications.
8. R. P. Kurshan and K. L. McMillan. A structural induction theorem for processes. In *Proceedings of the 8th ACM Symp. PODC*, 1989.