Incremental CTL Model Checking Using BDD Subsetting*

Abelardo Pardo
Mentor Graphics Corporation
267 Boston Road, Suite 2
Billerica, MA, 01862, USA
abelardo_pardo@mentorg.com

Gary D. Hachtel
University of Colorado
ECEN Campus Box 425
Boulder, CO, 80309, USA
hachtel@colorado.edu

Abstract

An automatic abstraction/refinement algorithm for symbolic CTL model checking is presented. Conservative model checking is thus done for the full CTL language–no restriction is made to the universal or existential fragments. The algorithm begins with conservative verification of an initial abstraction. If the conclusion is negative, it derives a "goal set" of states which require further resolution. It then successively refines, with respect to this goal set, the approximations made in the sub-formulas, until the given formula is verified or computational resources are exhausted. This method applies uniformly to the abstractions based in over-approximation as well as under-approximations of the model. Both the refinement and the abstraction procedures are based in BDD-subsetting. Note that refinement procedures which are based on error traces, are limited to over-approximation on the universal fragment (or for language containment), whereas the goal set method is applicable to all consistent approximations, and for all CTL formulas.

1 Introduction

The success of formal verification in detecting incorrect designs has been proven over the last decade. However, limitations on the size of verifiable problems continue to be a serious drawback. Symbolic techniques based on BDDs (Reduced Ordered Binary Decision Diagrams [3]), such as symbolic model checking [4, 13] have significantly increased the size of the systems which can be verified. However, industrial strength examples, model checking based on automatic conservative approximation is required [10, 9, 14].

The main idea behind abstraction is to interpret the behavior of a system in a abstracted (and therefore simplified) system with a more compact representation. An abstraction can be seen as a relation between two systems. On one hand, the original system has the complete description of its behavior, whereas its abstraction preserves some of that behavior and abstracts the rest. The key issue is to know in advance what parts of a design are relevant, and how to automatically abstract the rest of it. In most cases, the problem in the abstract domain is solved conservatively, that is, only when the formula is proved true it can be concluded true in the original system.

In this paper we provide an algorithm to perform CTL symbolic model checking based on the generic abstraction paradigm of [14]. We explore the effect of two kinds of abstraction: (1) BDD subsetting in the pervasive pre-image computations, and (2) gross initial approximation of the overall partitioned transition relation, followed by restoration of the exact relation. We emphasize, that this excludes a vast panoply of alternative automatic abstraction methods. However, the experimental results show that even with this extremely limited set of abstractions, excellent results can be obtained for CTL model checking.

2 Automatic Abstraction

Abstraction has been seen as an effective technique to simplify the systems to be verified to the extent that they can be handled by conventional symbolic techniques. The main challenge for abstraction is the trade-off between the degree of automation and its effectiveness.

In [12], Long proposed a conservative abstraction paradigm that preserved the validity of the logic $\forall$CTL. This method was one-sided, in that it considered only upper bound approximations of the underlying Kripke structure. In [5], a procedure for approximate traversal of large systems was presented, based on automatic state space decomposition. This technique was similarly one-sided, and applied only to reachability analysis. Neither of these methods provided a procedure for automatically refining the approximation until verification was conclusive.

In [8], Kurshan described an abstraction paradigm called "localization reduction" in the context of ω-regular language containment based on reducing the parts of a system that are *irrelevant* with respect to the task being verified. A systematic procedure was sketched which refined the approximations based on error trace analysis. A related iterative approach to abstraction in language containment verification was presented in [1]. In contrast to Kurshan's method, this was BDD-based, and the refinement considered sets of error traces. However, the details of iterating their method to a definite conclusion were omitted.

In [10], a method was given for conservative CTL model checking. Although this approach used both upper and lower bounds, and included a complete procedure for refining the initial approximation, it was limited to $\forall$CTL. In [7] Kelb et al. proposed an abstraction mechanism for μ-calculus model checking based on two novel approximations, which were called universal and existential. The process re-

*This work was supported in part by NSF/DARPA grant MIP-9115432 and SRC contract 95-DJ-560.

quired the intervention of the user. Although this approach applied to the full μ-calculus, and used both upper and lower bounds, only one (very interesting) type of approximation was used, and no automatic refinement procedure was given.

Our work is mainly based on the automatic abstraction paradigm presented by Pardo et al. in [14], thus a brief summary of it is given. The paradigm provides the sufficient conditions for an approximation to provide a conservative abstraction.

The formulas are represented by its parse tree, where each vertex represents a sub-formula. We will refer indistinctly to a sub-formula or to the vertex representing it. The model checking procedure traverses this tree in postorder computing the set of states that satisfy each of the sub-formulas. In this paradigm, each node in the tree is labeled with a *polarity* which denotes the type of approximation (either an over or under-approximation) that can be produced at that level in the verification.

The paradigm follows a "lazy" strategy, in the sense that it creates an initial crude approximation and keeps refining it until the result is conclusive. This refinement process is guided by a derived "goal set" of states that need further resolution. The "goal set" refinement method bears a superficial resemblance to the error trace refinement methods of [8] and [1]. However, the method of [14] is uniformly applicable to over and under-approximations whereas error traces are available only for upper bounds applied to universal CTL formulas or language containment.

In the context of reachability analysis of sequential systems Ravi et al. presented in [15] a set of techniques to simplify the representation of a set as a BDD while creating either a superset or a subset.

The work presented in this paper combines these two mechanisms of goal-set based abstraction/refinement and consistent approximation by BDD subsetting into one single implementation. Our primary objective is to explore how BDD simplification techniques can be used in the context of automatic abstraction for CTL model checking.

3 CTL Symbolic Model Checking

CTL (Computational Tree Logic) is a modal logic proposed by Clarke et al. in [6]. Predicates in this logic enable reasoning about the behavior of a given model with respect to a set of atomic propositions over time. Examples of atomic proposition predicates are `reset = 1` or `busId = 0x24`. Let us assume a set of atomic propositions A. The set of CTL formulas ϕ are those generated by the following grammar:

$$\phi ::= p \in A | \neg\phi \mid \phi_1 \wedge \phi_2 | \mathbf{EX}(\phi_1) | \mathbf{EG}(\phi_1) | \mathbf{E}(\phi_1 \mathbf{U} \phi_2).$$

The semantics of CTL is defined with respect to infinite sequences of states in a Kripke structure.

Definition 3.1 *Let a Kripke structure be a 5-tuple $K = (S, Q, R, A, \lambda)$ in which S is a set of states, $Q \subseteq S$ is the set of initial states, $R \subseteq S \times S$ is a transition relation, A is a set of atomic propositions, and $\lambda : A \to 2^S$ is the labeling function that returns the set of states in which an atomic proposition is satisfied.*

Definition 3.2 *An infinite path π in a Kripke structure $K = (S, Q, R, A, \lambda)$ is a sequence of states $s_1, s_2, \ldots$ such that $s_1 \in Q$ and $\forall i \in \mathbb{N}^+, (s_i, s_{i+1}) \in R$.*

Given a Kripke structure $K = (S, Q, R, A, \lambda)$ and a set of atomic propositions A, we write $(K, s) \models \phi$ to denote that the state s of the structure K satisfies the CTL formula ϕ. Let $\pi = s_1, s_2, \ldots$ be an infinite path. The semantics of CTL formulas is defined as:

- $(K, s) \models p$, where $p \in A$ iff $s \in \lambda(p)$.
- $(K, s) \models \neg\phi$ iff s does not satisfy the formula ϕ in K.
- $(K, s) \models (\phi_1 \wedge \phi_2)$ iff s satisfies both ϕ_1 and ϕ_2.
- $(K, s) \models \mathbf{EX}(\phi_1)$ iff $\exists\pi$ such that $(K, s_2) \models \phi_1$.
- $(K, s) \models \mathbf{EG}(\phi_1)$ iff $\exists\pi$ such that all states satisfy ϕ_1.
- $(K, s) \models \mathbf{E}(\phi_1 \mathbf{U} \phi_2)$ iff $\exists\pi$ such that there exists a state s' which satisfies ϕ_2 and every state from s_1 until s' (excluding s') satisfies ϕ_1.

Different techniques have been studied to improve the efficiency of the model checking algorithm. Among them, the one that provided a significant breakthrough was the manipulation of states symbolically with BDDs [4]. In symbolic model checking, the size of the BDD representation of the sets of states manipulated has a big impact on the efficiency of the algorithm.

The algorithm presented in this paper uses 2-sided BDD reduction techniques like the 1-sided ones presented in [15] to gradually increase the accuracy of the verification. Conventional symbolic model checking algorithms provide either an exact or an approximate computation. Our algorithm explores the space of possible approximations while looking for a conclusive verification result.

4 The Abstraction/Refinement Algorithm

The algorithm consists of two stages. Let us assume that a certain CTL formula ϕ is to be verified. The first stage amounts to obtaining a *crude* approximation of the verification problem. This approximation will have the property of being conservative in the sense defined in Section 1. In the second stage, the verification process is recursively analyzed to obtain a measure of how the current result needs to be refined to achieve the conclusive verification of the formula. This measure will be a set of states which will be called the *goal set*. This refinement process involves the recomputation of certain parts of the verification but with an increased level of accuracy.

In order to apply a BDD based approximation at any point in the model checking algorithm, each temporal operator is expressed in terms of conjunction, negation, preimage computation (denoted by *PreImage*) and fix-point iteration (denoted by *lfp*). This translation is based on the equivalences $\mathbf{EG}(\phi_1) = \neg \mathit{lfp}(X, \neg(\phi_1 \wedge \mathit{PreImage}(\neg X)))$ and $\mathbf{E}(\phi_1 \mathbf{U} \phi_2) = \mathit{lfp}(X, \neg\phi_2 \wedge \neg(\phi_1 \wedge \mathit{PreImage}(X)))$, where $\mathit{lfp}(X, \phi)$ represents a least fix-point iteration of the formula ϕ over the variable X, and ϕ represents both a CTL formula and the set of states s such that $(K, s) \models \phi$. Figure 5 shows an example of the tree representation of a formula.

4.1 Initial Abstraction

In principle, the algorithm may approximate the evaluation of any type of sub-formula. Our objective is then to introduce the approximations gradually in the parts of the verification that are more time consuming. From empirical data, we have observed that most of the verification time is spent in the pre-image computation.

Let us assume that a relation R and a set of states S are given to compute the operation $\mathit{PreImage}(R, S)$. The algorithm uses two types of abstractions: (1) Apply BDD subsetting to the set S and reduce its size by a constant factor. This technique can be applied to produce either over-approximations or under-approximations. (2) Assuming that the relation R is given as a product of sub-relations, the algorithm chooses a subset of the sub-relations as an over-approximation of R. This type of abstraction is used only when an over-approximation is required.

At every evaluation step, the algorithm annotates if an approximation method has been used. In the second stage, these approximation will be refined in order to obtain a conclusive result. Due to the simplifications in this initial evaluation, the answer to the verification process is obtained usually in a negligible amount of CPU time.

4.2 Refining the Approximation

The refinement stage of the algorithm is applied only if the result of the initial approximation is inconclusive. In our implementation, the initial approximation is very significant, and consequently, in almost all the cases, the verification problem remains inconclusive.

In the second stage we have the choice of applying gradual refinements to the different approximations. That is, whenever an approximation needs to be refined, it can be replaced by a better approximation and the computation is repeated. In the implementation discussed here, the refinement of an operation is done directly to the exact computation.

The motivation for this decision is based on experimental data obtained by exploring the trade-off in time between approximate and exact computations. Experimental results showed that in the case of gradual refinements, the number of approximate evaluations was significantly higher, and more importantly, this number outweighed the fact that the approximation required much less computation time to be performed. As a consequence, the heuristic that has been implemented attempts to detect these situations.

The overall refinement algorithm has an operand named the *goal set* that *guides* the process towards the resolution of the verification problem (i.e a conclusive result.) The goal set of a sub-formula ϕ (denoted by f_ϕ contains the set of states by which the approximation has to be modified in order to progress towards a conclusive result. If the operation has been over-approximated, the set contains the states that need to be removed from the result. Conversely, if an operation has been under-approximated, the set contains the set of states that need to be included in the result. The rules to propagate the goal set through the different sub-formulas were given in [14].

In the remaining of the section, for the sake of clarity, it will be assumed that all trivial cases have already been explored. The different algorithms presented in this section represent the situation in which an operation needs to be refined and the refinement is non-trivial. Also, we will denote by $Sat(v)$ the set of states that satisfy the formula represented by v and by $Sat'(v)$ an approximation of this set.

4.2.1 Refinement of the Conjunction Operation

Let us assume a formula v of type conjunction with sub-formulas v_1 and v_2. The refinement procedure receives the Kripke structure K, v, and the goal set f_v. Figure 1 shows the pseudo-code for the refinement algorithm of the conjunction. The function $P(v)$ returns the polarity labeling of the formula v denoting if an over-approximation or an under-approximation is required. The algorithm returns the BDD representation of the remaining goal set after refinement.

```
funct $RefineConjunction(K, v, f_v)$: set ≡
  if (The conjunction has been approximated) then
    return $TestSatisfaction(v, f_v)$
  endif
  $f_{v_1} = SubFormulaRefine(K, v_1, f_v)$
  if $(P(v) = +)$ then
    $result = SubFormulaRefine(K, v_2, f_{v_1})$
  else
    $f_{v_2} = SubFormulaRefine(K, v_2, f_v)$
    $result = f_{v_1} \vee f_{v_2}$
  endif
  $Sat'(v) = Sat'(v_1) \wedge Sat'(v_2)$
  return $result$
end
```

Figure 1: Refinement algorithm for the conjunction

If the conjunction has been approximated due to the size of the result then no further refinement attempt is made and the set of states that remain to be refined is returned by the function *TestSatisfaction*.

If the conjunction has not been approximated, the refinement needs to be propagated recursively to the sub-formulas. The sub-formula to be refined first is chosen trying to minimize the potential amount of refinement. The implemented heuristic checks first if any of the sub-formulas contains the variable of a fix-point. In that case the other sub-formula is chosen. If none of them contain the variable of a fix-point, the sub-formula with the smaller depth is chosen.

If the node must be over-approximated, the formula v_2 has to be refined with the goal set containing the states that could not be refined in v_1. The set denoted by f_{v_1} contains precisely these states. The set to be returned at the level of formula v is precisely the result of the refinement of v_2. If the node must be under-approximated, then both refinements of v_1 and v_2 have to be done with the goal set f_v.

After these refinements have been completed, the conjunction needs to be re-evaluated.

4.2.2 Refinement of the Pre-image Operation

The refinement procedure for the pre-image operation departs slightly from the common structure of the algorithm. It is assumed that a refinement fails, and the algorithm directly computes the propagation of the goal set f_v as $Image(R, f_v)$. The reason for this heuristic is totally based on experimental results. In all the examples in which this refinement was required, the image computation of the goal set was computed very efficiently. This design decision can be thought of as speculative execution. Figure 2 shows the pseudo-code for the algorithm just described.

```
funct $RefinePreImage(K, v, f_v)$: set ≡
  $f_{v_1} = Image(R, f_v)$
  $SubformulaRefine(K, v_1, f_{v_1})$
  if (There has been some refinement in $v_1$) then
    $ComputeExactPreImage(v)$
  endif
  $result = TestSatisfaction(v, f_v)$
  return $result$
end
```

Figure 2: Refinement algorithm for the pre-image

The algorithm first propagates the goal set to the sub-formula v_1. No exact pre-image computation is performed to check that this propagation is required. If this propagation is indeed required, then the procedure saves the computation of an exact pre-image. If this propagation is not required, the image computation is performed but the refinement will succeed in the immediate sub-formula.

After the recursive call, if the refinement of the sub-formula succeeds (the function returns the empty goal set), the exact pre-image computation is then performed.

4.2.3 Refinement of the Fix-point Operation

The approximation of the fix-point computation has several important effects. There is no relation between the number of iterations in the exact computation and the number of iterations of an approximation. The refinement procedure may refine several or all of the iterations in the sequence. If the last iteration is refined, the convergence criteria may not hold, and the fix-point needs to be iterated further. After convergence is restored, the result of the fix-point might have changed, and therefore, the refinement process needs to be started again.

Because of these considerations, the refinement process of a fix-point operation is divided into two levels. At the internal stage, one of the individual iterations is refined. In the outer stage, a post-processing of the sequence is required to guarantee convergence. Figure 3 shows the pseudo code for the overall procedure.

```
funct $RefineFixPoint(K, v, f_v)$: set $\equiv$
  $NewIterations = True$; $f'_v = f_v$
  while ($NewIterations \wedge f'_v \neq \emptyset$) do
    $RefineIteration(K, v, f'_v, lastIteration)$
    $RestoreContainmentInIterations(v)$
    $PruneIterations(v)$
    $NewIterations = IterateFixPoint(v)$
    $f'_v = TestSatisfaction(v, f'_v)$
  endwhile
  return $f'_v$
end
```

Figure 3: Refinement algorithm for a fix-point

The main loop of the procedure is executed if there has been new iterations produced or the evaluation of the fix-point can be further refined.

Let us denote by $(\mu_1, \ldots, \mu_n)$ the set of fix-point iterations computed so far, and $\mu' = (\mu'_1, \ldots, \mu'_n)$ the new set of iterations after the loop has been executed once.

The function *RefineIteration* computes the refinement for one specific iteration. If this refinement fails, it proceeds to refine the previous iteration. The main procedure shown starts by refining the last iteration of the sequence. The discussion of the function *RefineIteration* is deferred until later in this section.

The sequence of iterations satisfies the property that $\mu_{i-1} \subseteq \mu_i$. However, this property does not necessarily hold after some of the iterations have been refined. This situation happens when different approximations are produced every time the sub-formula is evaluated. After the refinement, the procedure *RestoreContainmentInIterations* propagates the effect of the local refinement to the entire sequence restoring the containment property. If the iterations have been over-approximated and iteration μ_i has been refined, the containment relation is restored by replacing every iteration $\mu_1, \ldots, \mu_{i-1}$ by its product with μ_i. Conversely, in the case of the refinement of an under-approximation, iterates $\mu_{i+1}, \ldots, \mu_n$ are replaced by its disjunction with μ_i.

The function *PruneIterations* detects early convergence cases in the new sequence of iterations. After the containment relation has been restored, it is possible that two arbitrary iterations in the sequence satisfy $\mu'_i = \mu'_{i+1}$, and they are not at the end of the sequence. In other words, the refinement of the iterations has provided a different fix-point value. In this case the iterations $\mu'_{i+2}, \ldots, \mu'_n$ can be disregarded and μ'_{i+1} is considered the new convergence point.

The procedure *IterateFixPoint* assures that the convergence of the fix-point is restored after refinement. This procedure simply re-evaluates the formula introducing approximations as in the first stage of the overall algorithm.

After the local refinement step has finished, the overall algorithm computes the new goal set. These steps are repeated until no further refinement can be applied, the refinement process has succeeded completely, or the refinement process did not produce any changes in the sequence of iterations. Figure 4 shows the pseudo-code for the procedure to refine one iteration.

```
funct $RefineIteration(K, v, f_v, index) \equiv$
  $x = ReadFixPointVariable(v)$
  $e(x) = ReadIteration(v, index - 1)$
  $SubFormulaEvaluate(K, v)$
  $result = SubFormulaRefine(K, v_1, f_v)$
  if (No further refinement is appropriate) then
    $StoreNewIteration(v_1, index)$
    return $result$
  endif
  $f'_v = ReadPropagatedGoalSet(v)$
  $RefineIteration(K, v, f'_v, e, index - 1)$
  $e(x) = ReadIteration(v, index - 1)$
  $SubFormulaEvaluateExact(K, v)$
  $StoreNewIteration(v_1, index)$
  return $TestSatisfaction(v, f_v)$
end
```

Figure 4: Refinement algorithm for a fix-point iteration

The procedure receives as a parameter the index of the iteration to be processed. In the first part of the procedure, the sub-formula is re-evaluated with the value of the previous iteration in order to replicate in all the vertices of the sub-formula the values that were present at evaluation time. This re-evaluation is the main reason to use a table storing previous results in the sub-formulas.

Once the formula has been re-evaluated, the refinement is performed by calling the function *SubFormulaRefine*. After this function returns, there are several possible situations:

- The set *result* is empty, the refinement of the sub-formula has succeeded completely at refining the goal set f_v. In this case no further computation is required.
- The set *result* is non-empty, the formula can still be refined, but this refinement will not change the value of the verification. In this case a conclusive result has been reached with an approximated result.
- The set *result* is non-empty, but the formula has been evaluated with no approximation. In this case the result is conclusive.

If none of the three conditions is satisfied, the refinement is applied to the previous iteration.

5 An Example

In this section a simple example is presented to illustrate the techniques presented in the previous section. The system models a three element token ring network. Each element has a binary state variable which denotes if the token is present or not. Let us denote by $state[i]$ the value of the ith state component. The formula that will be verified is expressed in CTL as

$$\mathbf{AG}(state[0] = 1 \Rightarrow \mathbf{AX}(state[0] = 0)), \qquad (1)$$

where $\mathbf{AG}(\phi) = \neg\mathbf{E}(True\mathbf{U}\neg\phi)$ and $\mathbf{AX}(\phi) = \neg\mathbf{EX}(\neg\phi)$. Intuitively, this formula verifies that whenever a state element has the token, it will not have it at the next time step.

Let us denote the possible states of the system by the decimal interpretation of the state elements being $state[0]$ the least significant bit. Figure 5 shows the state transition graph of the system as well as the parse tree corresponding to the CTL formula.

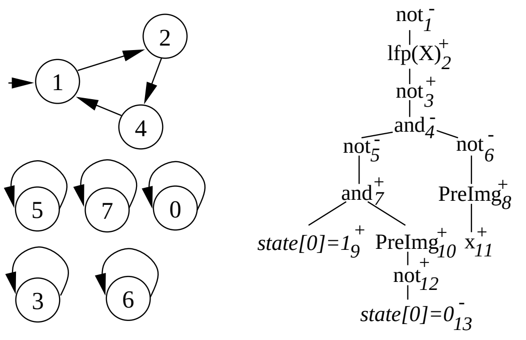

Figure 5: State graph and formula of the token ring example

Each vertex in the tree has been labeled with a unique number in order to refer to its verification result as well as the type of approximation provided (+ for over and − for under-approximation). We further assume that only the pre-image computations in vertices 8 and 10 are approximated.

Due to the size of the example, instead of considering the ordinary BDD subsetting as the approximation technique, we have approximated the computation by letting the behavior of the memory elements in $state[1]$ and $state[2]$ be free at all times.

After the first approximation has been obtained, the value of the function Sat for some of the vertices in the tree are

$$\begin{array}{ll} Sat(v_{13}) = \{0,2,4,6\} & Sat(v_5) = \{0,2,4,6\} \\ Sat(v_{12}) = \{1,3,5,7\} & Sat(v_2^1) = \emptyset \\ Sat(v_{10}) = S & Sat(v_2^2) = \{1,3,5,7\} \\ Sat(v_9) = \{1,3,5,7\} & Sat(v_2^3) = S \\ Sat(v_7) = \{1,3,5,7\} & Sat(v_2^4) = S, \end{array}$$

where S denotes the set of states $\{0,\ldots,7\}$.

The formula v_i in the table denotes the vertex labeled with the sub-index i in Figure 5. The values $v_2^1, v_2^2, v_2^3, v_2^4$ represent the sequence of iterations in the fix-point.

After the initial approximation, the result is the empty set, and the formula is declared false. Since some formulas have been approximated, this result is conservative. The refinement process is then started in v_1 with goal set $f_{v_1} = \{1\}$. The refinement process propagates until vertex v_4. The sub-formula chosen for refinement is v_5 because it does not contain any fix-point variable. For the same reason the refinement propagates from v_7 to v_9. Once in v_9 with $f_{v_9} = \{1\}$ the refinement fails because $f_{v_9} \neq \emptyset$ and the operation has been computed exactly. The program then proceeds to refine v_{10}.

At the level of v_{12}, $f_{v_{12}} = \{2\}$, and since v_{12} has been evaluated exactly, and $2 \notin Sat(v_{12})$ the refinement succeeds, and the pre-image in v_{10} is computed exactly. As a result, $Sat'(v_{10}) = \{2,3,4,5,6,7\}$, and since $f_{v_{10}} = \{1\}$, the refinement of v_{10} has succeeded. This new result propagates through vertices v_7 and v_5, such that $Sat'(v_7) = \{3,5,7\}$ and $Sat'(v_5) = \{0,1,2,4,6\}$.

The refinement then returns to v_4 and proceeds with v_6. In this case the refinement propagates until formula v_{11}, in which $f_{v_{11}} = \{2\}$ and the refinement propagates to the previous iteration.

The algorithm returns to v_2 and starts the refinement of iteration 3 with goal set $f_{v_2^3} = \{2\}$. Now the refinement is required only in v_6. This refinement succeeds and as a result, $v_2^3 = \{3,4,5,7\}$. Note that element 2 no longer belongs to the set. This refinement prompts the re-evaluation of v_2^4 and now $Sat'(v_2^4) = \{2,3,4,5,7\}$. As a consequence of these refinements the containment property among the iterations no longer holds. The containment is restored and the new sequence $(\emptyset, \{3,5,7\}, \{3,4,5,7\}, \{2,3,4,5,7\})$ is obtained.

This new sequence is no longer convergence so the algorithm keeps iterating until convergence is reached again. After the procedure *IterateFixPoint* is executed, two more iterations are added to the sequence, namely $Sat(v_2^5) = Sat(v_2^6) = S$.

Since the sequence of iterations has changed, the refinement process starts again. The process just described is repeated, including the restoration of the containment relation and the pruning of the iterations. As a result, the new obtained sequence is $(\emptyset, \{3,5,7\}, \{3,5,7\})$.

At this point $Sat(v_2) = \{3,5,7\}$ which satisfies the refinement of $f_{v_2} = \{1\}$, and the formula is proved true.

6 Experimental Results

The algorithm described has been implemented within the framework provided by VIS [2]. The results have been obtained in a Sun UltraSparc workstation with 170Mhz and 192Mb of RAM memory. For three examples we have compared the execution time using the incremental approach with that obtained with the model checker in VIS 1.2.

Production Cell: This system is a control circuit with 45 registers [11] for which 15 formulas have been verified. Reachability analysis has been used as a don't care condition in the verification process. Table 1 summarizes the results.

The left half of the table shows the time and number of pre-image operations taken by VIS. The right part shows the execution time with the new procedure as well as the number of the pre-image, approximate pre-image and exact image operations. The column labeled "Ratio" shows the ratio between the execution time with VIS and the incremental approach.

As it can be seen, there are certain instances in the specification in which the formula is proved in the first stage. This means that the initial crude approximation of our algorithm sufficed to prove the formula correct. The overall speedup factor is almost 2.

Ethernet model: The second example is a model of the Ethernet protocol to communicate a set of processors. It has 118 memory elements. The definition of this system

	VIS		Incremental		
Form.	*Pre*	Time	$Pre/\widehat{Pre}/Img$	Time	Ratio
1	43	17.2	27/39/27	21.1	0.82
2	83	17.9	39/33/73	20.1	0.89
3	84	18.8	0/9/0	6.4	2.94
4	91	35.7	43/34/58	30.5	1.17
5	92	36.2	0/9/0	7.1	5.10
6	83	28.2	39/26/20	22.2	1.27
7	84	28.4	0/5/0	6.5	4.37
8	83	13.3	39/31/21	13.8	0.96
9	84	13.3	0/4/0	6.0	2.22
10	83	22.3	39/25/17	17.3	1.29
11	84	22.0	0/4/0	6.1	3.61
12	83	28.2	39/22/29	21.2	1.33
13	84	27.6	0/5/0	6.3	4.38
14	83	28.1	20/32/32	28.7	0.98
15	84	27.9	0/6/0	6.1	4.57
Total		414.8		219.4	1.89

Table 1: Verification of the production cell

includes several parameters that can be used to scale up the size of the design.

Table 2 shows the results for two version of the system with different internal parameters that have a significant impact in the execution time. In this case our algorithm did not verify any formulas with the initial approximation only, however, we see a consistent reduction in the number of pre-image computations. In some cases our method achieves speedups of up to a factor of 5.

	VIS		Incremental		
Form.	*Pre*	Time	$Pre/\widehat{Pre}/Img$	Time	Ratio
13.1	45	416	24/18/22	345	1.20
13.2	45	344	23/16/14	89	3.85
13.3	61	486	27/45/21	632	0.77
13.4	61	479	27/28/21	339	1.41
13.5	47	346	21/16/20	384	0.90
13.6	47	314	22/17/21	152	2.07
Total		2414.6		1944.1	1.24
23.1	65	1056	31/26/25	1939	0.54
23.2	65	946	35/23/27	480	1.97
23.3	71	7911	55/133/76	1956	4.04
23.4	71	7884	24/33/23	1593	4.95
23.5	67	1794	35/23/27	1514	1.18
23.6	67	956	32/22/26	571	1.67
Total		20578.2		8056.2	2.55

Table 2: Verification of the Ethernet system

Landing Gear Control: The third example analyzed is a system controlling the landing gear of an aircraft. The system contains 231 memory elements. The only formula verified has been the resetability condition, which is defined as the possibility of driving the system from any state to its initial state. The execution results showed 22626 seconds for VIS compared with 11383 seconds for the incremental algorithm.

7 Conclusions

We have presented a novel incremental algorithm to perform CTL symbolic model checking. The idea is to initially create an approximation of the verification and gradually refine it until the formula is proved conclusively. The approximations are created by reducing the size of the BDDs while creating a superset or a subset of the original sets.

This paradigm makes no assumption about the structure of the system, so it can be applied automatically. The experimental results have shown that for three systems, there are instances in which the algorithm is significantly more efficient than conventional exact model checking. The presented procedures are straightforwardly extended to the past tense of the μ-calculus by simply adding Image to the logic.

As for future directions, we plan to extend the type of abstractions that can be used in this paradigm in order to obtain the improvements needed to handle designs with a memory element count around $10,000$. We feel that this work constitutes a first step in that direction. Techniques likes the one presented in [10] are applicable in this paradigm. Furthermore, there has been significant advances in the area of BDD subsetting since [15], and not all them have been incorporated into our paradigm.

References

[1] F. Balarin and A. L. Sangiovanni-Vincentelli. An iterative approach to language containment. In C. Courcoubetis, editor, *Fifth Conference on Computer Aided Verification (CAV '93)*. Springer-Verlag, Berlin, 1993. LNCS 697.

[2] R. K. Brayton et al. VIS: A system for verification and synthesis. In T. Henzinger and R. Alur, editors, *Eigth Conference on Computer Aided Verification (CAV'96)*, pages 428–432. Springer-Verlag, Rutgers Univ., 1996. LNCS 1102.

[3] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[4] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *Proceedings of the Design Automation Conference*, pages 46–51, June 1990.

[5] H. Cho, G. D. Hachtel, E. Macii, B. Plessier, and F. Somenzi. Algorithms for approximate FSM traversal based on state space decomposition. In *Proceedings of the Design Automation Conference*, pages 25–30, Dallas, TX, June 1993.

[6] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proceedings Workshop on Logics of Programs*, pages 52–71, Berlin, 1981. Springer-Verlag. LNCS 131.

[7] P. Kelb, D. Dams, and R. Gerth. Practical symbolic model checking of the full μ-calculus using compositional abstractions. Technical Report 95-31, Department of Computing Science, Eindhoven University of Technology, 1995.

[8] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes.* Princeton University Press, 1994.

[9] R. P. Kurshan. Formal verification in a commercial setting. In *Proceedings of the Design Automation Conference*, pages 258–262, Anaheim, CA, June 1997.

[10] W. Lee, A. Pardo, J. Jang, G. Hachtel, and F. Somenzi. Tearing based abstraction for CTL model checking. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 76–81, San Jose, CA, November 1996.

[11] Thomas Lindner. *Case Study "Production Cell": A Comparative Study in Formal Software Development*, chapter 2, pages 9,21. FZI, 1994.

[12] D. E. Long. *Model Checking, Abstraction, and Compositional Verification.* PhD thesis, CMU, July 1993.

[13] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Boston, MA, 1994.

[14] Abelardo Pardo and Gary Hachtel. Automatic abstraction techniques for propositional μ-calculus model checking. In *9th Conference on Computer Aided Verification (CAV'97)*. Springer-Verlag, June 1997.

[15] K. Ravi and F. Somenzi. High-density reachability analysis. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 154–158, San Jose, CA, November 1995.