Automatic Abstraction Techniques for Propositional μ-calculus Model Checking*

Abelardo Pardo and Gary D. Hachtel

University of Colorado
ECEN Campus Box 425, Boulder, CO, 80309, USA
{abel,hachtel}@vlsi.colorado.edu

Abstract. An abstraction/refinement paradigm for the full propositional μ-calculus is presented. No distinction is made between universal or existential fragments. Necessary conditions for conservative verification are provided, along with a fully automatic symbolic model checking abstraction algorithm. 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 subformulas, until the given formula is verified or computational resources are exhausted.

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. Typically, a sequential system is modeled as a collection of interacting subsystems. As new subsystems are added the number of states may grow exponentially. This is known as the *state explosion* problem. This problem has become the focus of intense research in the last few years. Symbolic techniques based on BDDs (Reduced Ordered Binary Decision Diagrams [3]), such as symbolic model checking [5, 16] have significantly increased the size of the systems which can be verified. However, there are numerous examples for which additional techniques are required.

Abstract interpretation [7] is one approach to alleviate the state explosion problem. An *abstracted* system is obtained from a given *concrete* system by modeling groups of states as a single state. In contrast, BDD-based techniques tend to stay in the given state space, while providing a more compact representation. In [15] 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 [6], 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 [13], 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

* Work supported by NSF/DARPA grant MIP-94-22268 and SRC contract 95-DJ-560.

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. These methods used only upper bound approximations of the underlying Kripke structure.

In [14], 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 ∀CTL. In [11] 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 required 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.

In this paper, novel techniques are presented for generic, BDD-based, fully automated abstraction for μ-calculus model checking. A complete procedure for automatically refining the approximations is included. This procedure is "lazy", in the sense, that it refines approximations in sub-formulas only where necessary. The "goal set" refinement method differs from the error trace refinement methods of [13] and [1] in that it is uniformly applicable to upper and lower bounds and to all μ-calculus formulas, whereas error traces are available only for upper bounds applied to universal CTL formulas or language containment. However, a superficial resemblance between the two can be discerned for some sub-formulas.

The plan of the paper is as follows. In Section 2 an overview of the propositional μ-calculus is given, and a labeled operational graph is defined. Section 3 lays the foundation for conservative μ-calculus symbolic model checking. A condition is given, in terms of the operational graph labels, to determine whether an over-approximation or under-approximation is required at the level of any sub-formula. In Section 4, a fully automatic BDD based model checking procedure is presented. The algorithm has two phases. First, it is shown how the given formula is tentatively verified with an initial abstraction. Then, it is shown how the algorithm recursively traverses every sub-formula and checks if the current approximation can be refined to achieve exact verification.

2 Propositional μ-calculus and Symbolic Model Checking

The propositional μ-calculus was introduced in [12]. It consists of propositional modal logic with a least fixed point operator. This calculus has been shown to be strictly more expressive than conventional temporal logics such as CTL, LTL and CTL* [8,9]. Also, it can express the language containment relation between two deterministic ω-automata.

2.1 Syntax and Semantics of the Propositional μ-calculus

Let us denote by A a set of atomic propositions with $p \in A$ and let X denote a set of variables with $x \in X$. The syntax of a formula ϕ is defined by

$$\phi ::= \phi_1 \wedge \phi_2 \mid \neg\phi \mid \mathbf{EX}\phi \mid \mu x.\phi \mid p \mid x. \qquad (1)$$

In order to insure well-definedness of the semantics, each occurrence of the variable x in the formula $\mu x.\phi$ must be within the scope of an even number of negations. This restriction ensures monotonicity and therefore existence of the fixed point.

A state satisfies the formula $\mathbf{EX}\phi$ if ϕ is satisfied in one of its immediate successors. We will denote by $L\mu$ the set of closed formulas obtained with the sets A and X and the grammar in Equation 1.

Definition 1. A Kripke structure is a tuple $M = (S, R, A, \lambda)$ in which S is a set of states, $R \subseteq S \times S$ is a transition relation, A is a set of atomic propositions, and $\lambda : A \rightarrow 2^S$ is a labeling function that returns the set of states in S labeled with a given atomic proposition.

Definition 2. Given a Kripke structure $M = (S, R, A, \lambda)$ and a set of states $S_1 \subseteq S$, we define the image function $Img(R, S_2)$ as the set of states S_1 such that $\forall s \in S_1, \exists s' \in S_2 \ni (s', s) \in R$. We also define the reverse image function $Pre(R, S_1)$ as the set of states S_2 such that $\forall s \in S_2, \exists s' \in S_1 \ni (s, s') \in R$.

Note that both Img and Pre are monotonic functions, that is, if we replace either operand by a subset or a superset, we obtain a subset or a superset of the exact result.

Given the set of variables X and a state space S, we define an *environment* as a function $e : X \rightarrow 2^S$. Given a Kripke structure M the semantics of $\phi \in L$ are defined as the function Sat that given a formula and an environment, it returns the set of states satisfying the formula. When necessary, we will write Sat_M to make the intended Kripke structure explicit. Figure 1 shows the algorithm to compute this function. Emerson *et al.* proved that model checking problem for this calculus is exponential in the alternation depth of the formula [9]. However, the depth of most useful temporal formulas is two or less.

```
funct $Sat(v, e) \equiv$
  case $TypeOf(v)$ do
    and$(v_1, v_2)$: return $EvalAnd(v_1, v_2, e)$;
    not$(v_1)$: return $EvalNot(v_1, e)$;
    EX$(v_1)$: return $EvalEX(v_1, e)$;
    $\mu x.v_1$: return $Eval\mu x(v_1, e)$;
    Atom$(p)$: return $\lambda(p)$;
    Variable$(x)$: return $e(x)$;
  end case
end
funct $EvalAnd(v_1, v_2, e) \equiv$
  return $Sat(v_1, e) \cap Sat(v_2, e)$;
end

funct $EvalNot(v_1, e) \equiv$
  return $\overline{Sat(v_1, e)}$;
end
funct $EvalEX(v_1, e) \equiv$
  return $Pre(R, Sat(v_1, e))$;
end
funct $Eval\mu x(v, e) \equiv$
  $x := ReadVariable(v)$;
  $result := \emptyset$;
  do
    $e(x) := result$;
    $result := Sat(v_1, e)$;
  while $(result \neq e(x))$ od;
  return $result$;
end
```

Fig. 1. μ-Calculus Model Checking Algorithm.

Given a Kripke structure M, a state s satisfies the formula ϕ, denoted by $(M, s) \models \phi$, if and only if $s \in Sat(\phi, e_\perp)$, where $e_\perp$ is the environment that maps every variable to the empty set.

2.2 Graph Representation of a Formula

The verification algorithm of Figure 1 uses a graph representation based on the parse graph.

Definition 3. We define the *labeled operational graph* of the formula $\phi \in L\mu$ as $G = (V, E, P)$. V is a set of vertices. Each vertex is of the type $\{\mathbf{and}, \mathbf{not}, \mathbf{EX}, \mu x\} \cup \{\mathbf{Atom}(p) \ni p \in A\} \cup \{\mathbf{Variable}(x) \ni x \in X\}$ and represents a sub-formula of ϕ. We will refer to a vertex or the sub-formula it represents interchangeably. $(v_1, v_2) \in E$ if v_2 is a direct sub-formula of v_1. We will denote by $top_\phi \in V$ the vertex representing ϕ. Every vertex v is labeled with a *polarity* which is the number of vertices of type **not** that are traversed in the path from v to top_ϕ excluding v itself. P is a function $P : V \rightarrow \{+, -\}$ such that $P(v) = +$ if v has odd polarity and $P(v) = -$ if v has even polarity.

Identical sub-graphs with identical polarity labeling represent common sub-formulas and therefore are shared.

2.3 BDDs and Symbolic Model Checking

BDDs [3] provide a canonical and efficient representation of boolean functions. Furthermore, although BDD size is exponential in its worse case, practical cases have been shown to present very compact representations. By a *symbolic model checking* algorithm we refer to an algorithm in which boolean functions are represented by BDDs [5].

In order to manipulate BDDs it is necessary to encode the state space with boolean variables. Let us denote by $\mathbf{u}$ the array of boolean variables required to encode the state space S of a given Kripke structure. Following this notation, we will represent a set in the state space S as a boolean function $S(\mathbf{u})$ such that, if $s \in S$ then $S(\beta(s)) = 1$ where $\beta(s)$ is the binary encoding of s. Analogously, a relation R will be represented as a boolean function $R(\mathbf{u}, \mathbf{w})$. Henceforth, whenever a set or a relation is mentioned we will be referring to its symbolic representation.

3 Conservative Abstraction

Abstract interpretation is a paradigm first introduced by Cousot *et al.* [7] in the context of static analysis of programs. The main idea is to interpret the behavior of a system in a different *abstracted* (and therefore simplified) system with fewer states. In the context of symbolic model checking, the complexity of the algorithms depends no longer on the number of states but on its representation as BDDs. An abstracted system therefore must be simplified so as to provide more compact BDD representations of the sets appearing in the verification algorithm.

The abstraction techniques presented in this paper assume that the state space in the concrete and abstract systems are the same. The simplification is based on taking supersets and subsets of a given set with a more compact representation.

3.1 Conservative μ-calculus Model Checking

In our abstraction, the concrete and abstract system share the same state space. Thus, we state the conservativeness property in terms of an approximation $\widehat{Sat}$ of the function Sat.

Definition 4. Given a Kripke structure M, we say the function $\widehat{Sat}$ provides a conservative interpretation if and only if $\forall \phi \in L\mu$, $\widehat{Sat}(\phi, e_\perp) \subseteq Sat(\phi, e_\perp)$.

If the conservative verification algorithm proves the formula true, we can conclude that the formula is true in the concrete system. However, if the formula is proved false, no conclusion can be drawn in the concrete system. This definition can be reversed to provide conclusive verification when a formula is false, providing a reliable false result. "Reliable positive" conservativeness is assumed in the sequel. However, the techniques presented apply dually for both cases.

Let us denote by I the set of initial states of the system represented by the Kripke structure M. The system satisfies the formula ϕ if and only if $I \subseteq Sat(\phi, e_\perp)$. Due to the conservativeness property $I \subseteq \widehat{Sat}(\phi, e_\perp) \Rightarrow I \subseteq Sat(\phi, e_\perp)$.

Lemma 5. *Let us consider an operational graph $G = (V, E, P)$ and two vertices v_1, v_2 such that $(v_1, v_2) \in E$. Let us assume that in the computation of $Sat(v_1, e)$ the evaluation of v_2 has been approximated by $\widehat{Sat}(v_2, e)$. If v_1 is of type* **not** *and $\widehat{Sat}(v_2, e)$ is a superset (subset), then the computed $Sat(v_1, e)$ is a subset (superset) of the exact result. If v_1 is of type* **and**, **EX**, *or μx and $\widehat{Sat}(v_2, e)$ is a superset (subset), then $Sat(v_1, e)$ also is a superset (subset) of the exact result.*

Proof. The first part of the lemma is trivial by set complementation. If v_1 is of type **and** the lemma is true by monotonicity of boolean conjunction. If v_1 is of type **EX** then by monotonicity of the Pre function the lemma holds. If v_1 is of type μx the lemma is proved by induction over the number of symbols in the formula represented by v_2. If the length is 1 the lemma is trivially true. Let us assume the lemma is true for formulas of length up to n. The approximation is reflected in the value of the variable x. By definition this variable must be within the scope of an even number of negations. By the induction hypothesis, approximation of the vertex v_2 is consistently of the same type throughout the fixed point iteration, thus the lemma holds. □

Theorem 6. *Let us consider a Kripke structure M, a formula $\phi \in L\mu$, and its labeled operational graph $G = (V, E, P)$. Any function $\widehat{Sat}$ such that for every vertex $v \in V$*

$$\begin{aligned} Sat(v, e) &\subseteq \widehat{Sat}(v, e) \quad \text{if } P(v) = + \\ Sat(v, e) &\supseteq \widehat{Sat}(v, e) \quad \text{if } P(v) = - \end{aligned} \tag{2}$$

provides a conservative interpretation of ϕ in M.

Proof. By Lemma 5, all the vertices propagate the direction of the approximation except those of type **not** which switch the direction. By Definition 4, the set $Sat(top_\phi, e)$ must be under-approximated. The approximation taken at any sub-formula must be such that when propagated to the top vertex it translates into a subset of $Sat(top_\phi, e)$. This condition is guaranteed by the function P. □

This result is advantageous in the verification process. For example, there is no need to distinguish between universal and existential sub-formulas. Also, the approximations can be produced at the level of any sub-formula.

An approximation may be created not only on the transition relation of the system inside the function *EvalEX* but on any type of vertex. For example, in a vertex v of type **and** such that $P(v) = +$, we may evaluate only one of the operands and return it as the result since it is a valid superset of the exact *Sat* function. Further, this paradigm allows for an incremental approach to verification. First, an initial approximation of all the vertices is obtained. Second, a set of vertices is chosen in which the approximations are to be refined. This is the essence of the algorithm presented in Section 4.

Theorem 6 also provides a condition that is local for every vertex. Two vertices of the same type representing different sub-formulas need not have the same kind of approximation. For example, we may provide different estimations of the transition relation R in the evaluation of different **EX** vertices. This property may be considered an advantage if we have a local measure of how the approximation must be refined locally to increase its exactness globally.

4 An Automatic Abstraction and Refinement Algorithm

In this section a novel verification algorithm is presented that exploits the advantages of the above paradigm. The conservativeness property is preserved by the local approximations made at relevant vertices. The automatic procedure successively refines these approximations until the formula is decided or resources are exhausted.

The proposed algorithm has as one of its parameters, a limit on the size of the BDD representation of the intermediate results. At the end of the computation either the formula is proved true, the formula is proved false or the memory limit is reached and therefore no conclusion can be drawn.

The algorithm has two phases: Creation of the initial approximation, and successive refinement. In the first step, the algorithm traverses the labeled operational graph and obtains an approximation for every evaluation of the function *Sat* at every vertex. To guarantee conservativeness, the type of approximation is determined by the condition in Theorem 6. Once the first abstraction is obtained, if the formula is proved false, the graph is traversed by depth first search to detect vertices whose approximation can be refined. At a given vertex, the algorithm attempts to improve the approximation with respect to a given "goal set". This set is obtained by propagating to other vertices in the graph the condition to achieve verification in the top vertex.

4.1 Incremental Approximations

The elementary units of the algorithm are the evaluation functions shown in Figure 1. If they satisfy the following two properties, we will call them *incremental approximations*. First, they must return a superset or subset of the exact result according to the polarity. Second, note that the BDD size limit may be reached at any point in the computation. Thus, we require that all partial results constitute valid approximations.

The generality of this paradigm leaves the choice of the type of approximation techniques open. A candidate set of techniques are provided which meet the requirements of our paradigm. In principle, any other technique that complies with the above conditions can be used.

We now show how to generate the two types of approximations. Since the negation of a function represented by a BDD is a constant time operation that has no effect on its size, the function *EvalNot* is computed exactly.

Conjunction Operation: The function *EvalAnd* takes the conjunction of the result of evaluating its two sub-formulas. We rely on the monotonicity property of this function to provide the required approximations. A superset is obtained by over-approximating either of its operands. A BDD representing the characteristic function of a set may be reduced in size by either adding or subtracting elements to that set. In the case of an over-approximation the operands are simplified by adding elements to the sets. In a vertex of type **and** the algorithm checks if the size of either of the operands exceeds its limit. If so, the operand(s) are simplified before the conjunction is taken.

If an under-approximation is required we rely on the following decomposition

$$a(\mathbf{u}) \wedge b(\mathbf{u}) = (u_i \wedge (a(\mathbf{u}) \wedge b(\mathbf{u}))_{u_i}) \vee (\neg u_i \wedge (a(\mathbf{u}) \wedge b(\mathbf{u}))_{\neg u_i}) \qquad (3)$$

where the subscript symbolizes the cofactor operation with respect to a variable. This decomposition is applied recursively. At each recursive step, both operands are cofactored with respect to a variable, thus reducing the size of the candidate conjunction. When enough reduction has been achieved, the conjunction is returned as a valid subset of the exact result.

Fixed Point Computation: The partial results μ_i obtained while iterating the least fixed point satisfy $\emptyset = \mu_0 \subseteq \ldots \subseteq \mu_\infty$. If an under-approximation is required, any set μ_i constitutes a valid result. In this case, the algorithm iterates for an arbitrary number of steps and returns the result without reaching convergence. This scheme also provides a natural way to refine the approximation. If the approximation previously computed needs to be refined, it is enough to apply several additional iterations, since each of them creates a superset of the previous result.

The other possible scenario is that the fixed point vertex has $P(v) = +$ and therefore an over-approximation is required. In this case the fixed point iteration has to reach convergence before returning a correct approximation. At each iteration of the fixed point, the algorithm monitors the size of the representation of μ_i. Whenever this size reaches a certain limit, it applies a BDD simplification procedure that reduces its size while creating a superset. When convergence is reached, the result is guaranteed to be an over-approximation of the real result.

***Pre* Computation:** The *Pre* operation is often responsible for the increase in size during the verification process. Even though both the operands and the final result may have a compact representation, the intermediate results may go beyond the computational limit. One method that significantly minimizes this effect is to manipulate the transition relation as a conjunction of relational blocks [4]. The reverse image is now obtained by successive steps of conjunction and variable existential abstraction. Several heuristics have been develop to compute the way the relation is broken into blocks and the order of the blocks so to minimize the size of the intermediate results (i.e. [10]).

The proposed *EvalEX* function modifies such reverse image computation in two different ways. The first method takes the conjunction and existential abstraction of the relational blocks until a certain limit in the size of the intermediate result is reached. At that point, no more conjunctions are taken and all the remaining variables are quantified, yielding a superset. The second method amounts to sub-setting or super-setting C in $Pre(R, C)$.

If a vertex of type **EX** has $P(v) = -$ then the algorithm must provide an under-estimation. The algorithm builds its result incrementally, and it is a modification of the one proposed in [17]. With this scheme, the algorithm builds the reverse image by recursive cofactoring the set C. If at any point during this recursion, the intermediate results grow too large, the algorithm returns the current partial result.

4.2 Refining the Abstraction

We assume that for every vertex v an initial approximation denoted by Sat_v has been obtained, and the verification of the formula is false. We describe the approximation of a vertex with respect to a set f_v. If the vertex has been over-approximated, the refinement amounts to producing a new result such that f_v is not included in it. If the vertex has been under-approximated, the refinement amounts to computing a new set which includes f_v.

Figure 2 shows the pseudo code for the generic procedure to refine the approximation in a given vertex. The specifics depend on the type of vertex and are explained subsequently. This procedure is preliminary and may be substantially improved, so only the main ideas behind it are presented.

```
funct $RefineVertex(v, f_v)$:boolean ≡
  if ($f_v = \emptyset$) then return TRUE;
  if ($Sat_v$ is an approximation) then
    $(Sat'_v, f'_v) := RefineApproximation(Sat_v, f_v)$;
    if ($f'_v = \emptyset$) return TRUE;
    $f_v := f'_v$;
  endif
  Sort Sub-formulas;
  foreach (Sub-formula $v_i$) do
    $f_{v_i} := PropagateGoalSet(v, v_i)$;
    $result_i = RefineVertex(v_i, f_{v_i})$;
  od
  if ($RefinementInSubFormulas(result)$) then
    $Sat'_v := ReEvaluate(v)$;
    if ($(P(v) = +) \wedge (f_v \cap Sat'_v = \emptyset)$) return TRUE;
    if ($(P(v) = -) \wedge (f_v \subseteq Sat'_v)$) return TRUE;
  endif
  return FALSE;
end
```

Fig. 2. Algorithm to Refine the Approximation in a Vertex.

Definition 7. Let us assume that an evaluation of the function Sat at a vertex v has been approximated. For a given set of states f_v the refinement of Sat_v with respect to f_v computed in the procedure of Figure 2 is successful if the new approximation Sat'_v satisfies

$$\begin{aligned} Sat'_v \subseteq Sat_v \setminus f_v \ \ \text{if } P(v) = + \\ Sat_v \cup f_v \subseteq Sat'_v \ \ \text{if } P(v) = -. \end{aligned} \tag{4}$$

In other words, if Sat_v has been over-approximated, the refinement attempts to exclude the elements of f_v from the approximation. Conversely, if Sat_v has been under-approximated, the refinement attempts to increase the set to include the elements in f_v. Note that since the algorithm works with memory limits, it is possible that in the attempt to refine the approximation, a limit is reached and the refinement fails. The procedure in Figure 2 returns TRUE if the new refinement satisfies Equation 4 and FALSE otherwise.

When the procedure is applied to a generic vertex v, there are two possible scenarios. The initial approximation Sat_v has been either computed in the vertex itself, or it has been propagated from the approximation of any of its sub-formulas.

If the approximation has been produced in the vertex itself, the proper incremental approximation procedure described in Section 4.1 is re-executed. However, this time the approximation process is modified so as to include or exclude the set f_v from the result. For example, in a vertex of type EX the initial result has been over-approximated by considering a subset of relational blocks. The refinement algorithm computes a new approximation but this time considering the relational blocks that exclude the set f_v from the result.

If the new approximation succeeds with respect to the whole set f_v, the procedure returns TRUE. If not, the set f'_v contains the elements of f_v that were not refined from Sat_v. If no further approximation has been produced in v the whole set f_v is recursively propagated to the sub-formulas.

The sub-formulas of v are scheduled for refinement with criteria based on the size of the BDDs and the depth. The depth of a formula is defined as the length of the longest path to a leaf vertex. The propagation of the set f_v to the sub-formulas is different depending on the type of v.

- **Negation Vertex**: The vertex of type **not** has a single sub-formula v_1 and the set f_v is propagated such that $f_{v_1} = f_v$.
- **Conjunction Vertex**: Let us assume that v has sub-formulas v_1 and v_2 already sorted by increasing depth. If $P(v) = +$ then $f_{v_1} = f_v$ and $f_{v_2} = Sat'_{v_1} \cap f_v$. If $P(v) = -$ then $f_{v_1} = f_v$ and $f_{v_2} = f_v$.
- **EX Vertex**: In this vertex, $Sat_v = Pre(R, Sat_{v_1})$. The set f_{v_1} is propagated such that $f_{v_1} = Img(R, f_v)$.
- **Fixed Point Vertex**: For this type of vertex the algorithm has stored the sequence of intermediate results $\mu_1, \ldots, \mu_m$. The refinement process is applied at first to the set μ_m. If the approximations cannot be improved, the refinement keeps propagating to the sub-formula and eventually the refinement is applied to the set μ_{m-1}. The propagation of the set f_v is $f_{v_1} = f_v$.
- **Variable(x) Vertex**: Although this vertex does not have any sub-formulas, the refinement process may be propagated through it. Intuitively, if a vertex of this type needs to be refined, that refinement refers in fact to the temporary result produced by the fixed point that binds the variable x. Therefore, the refinement

process is propagated to the vertex representing the fixed point sub-formula. The refinement of the fixed point vertex refers now to the set μ_{i-1} where μ_i is the set that has been considered for refinement the last. Independently of the value of $P(v)$, the propagation is such that $f_{v_1} = f_v$.

The vertex of type **Atom**(p) constitutes the trivial case of the recursive procedure, and since no approximations are computed, the procedure returns FALSE with no further computation.

After the algorithm recursively refined the approximations in the sub-formulas, it checks if v needs to be re-evaluated again. For example, in a vertex of type μx with $P(v) = +$ if the last set computed in the fixed point has been refined, the fixed point has to be iterated until it reaches convergence again. In the case of a vertex of type **and** it simply re-computes the conjunction.

To guarantee the correctness of the procedure in Figure 2 we need to prove that the refinement process succeeds in v if the propagated refinement succeeds in the sub-formulas.

Proposition 8. *Given a vertex v with sub-formulas v_1 (and v_2 when applicable), for every sub-formula v_i, $RefineVertex(v_i, f_{v_i}) = TRUE \Rightarrow RefineVertex(v, f_v) = TRUE$.*

Proof. If v is of type **not**, since no approximation is made, $Sat_v = \neg Sat_{v_1}$ and $Sat'_v = \neg Sat'_{v_1}$. For $P(v) = +$, the polarity of v_1 is $P(v_1) = -$ and therefore $Sat_{v_1} \cup f_{v_1} \subseteq Sat'_{v_1} \Rightarrow Sat'_v \subseteq Sat_v \setminus f_v$. If $P(v) = -$ then $P(v_1) = +$ and since $Sat'_{v_1} \subseteq Sat_{v_1} \setminus f_v$ then $Sat_v \cup f_v \subseteq Sat'_v$.

If v is of type **and** and $P(v) = -$, then the proposition holds because if the refinement process succeeds in both sub-formulas, then $f_v \subseteq Sat'_{v_1}$ and $f_v \subseteq Sat'_{v_2}$. Since $Sat'_v = Sat'_{v_1} \cap Sat'_{v_2}$ then $f_v \subseteq Sat'_v$. If $P(v) = +$ the refinement of f_v propagates to both formulas. If the process succeeds in both sub-formulas then $f_v \nsubseteq Sat'_{v_1}$ and $f_v \nsubseteq Sat_{v_2}$ and therefore $f_v \nsubseteq Sat'_v$.

If v is of type **EX** then $Sat_v = Pre(R, Sat_{v_1})$. If $P(v) = +$ the success of the refinement at the sub-formula v_1 implies $Sat'_{v_1} \subseteq Sat_{v_1} \setminus f_{v_1}$. Since $f_{v_1} = Img(R, f_v)$ there is no pair of states s_1, s_2 such that $s_1 \in f_v$, $s_2 \in Sat'_{v_1}$ and $(s_1, s_2) \in R$. Therefore $Pre(R, Sat'_{v_1}) \subseteq Sat_v \setminus f_v$. The proof when $P(v) = -$ is analogous.

When v is of type μx the theorem holds independently of the polarity. The fixed point does not perform any type of computation over the result obtained from its sub-formula. Thus the refinement of v_1 propagates to v.

If v is of type **Variable**(x) it is true that $Sat_v = Sat_{v_1}$ then $Sat'_v = Sat'_{v_1}$. Therefore if the refinement succeeds in v_1 it also succeeds in v. □

Theorem 9. *For a given vertex v, an approximation Sat_v, and a set f_v, the algorithm of Figure 2 returns TRUE if Equation 4 is satisfied and FALSE otherwise.*

Proof. If the approximation Sat_v has been computed locally in v (independently of the result of its sub-formulas) the first part of the algorithm guarantees that the elements of f_v are included or excluded from Sat_v depending on $P(v)$. Those elements that could not be refined from f_v are propagated to the sub-formulas of v following the rules discussed above and the theorem holds because of Proposition 8. □

The initial refinement set f_v is obtained from the top vertex in the graph. Since we assumed that our verification procedure is conservative and the initial approximation proved the formula false, then $I \nsubseteq Sat(top_\phi, e_\perp)$. Since $P(top_\phi) = -$, if the set $Sat(top_\phi, e_\perp)$ is increased by $I \setminus Sat(top_\phi, e_\perp)$ the verification is successful. Therefore the refinement process starts with the function call $RefineVertex(top_\phi, I \setminus Sat(top_\phi, e_\perp))$.

5 Conclusions and Future Work

We have presented a general abstraction/refinement paradigm for propositional μ-calculus. It provides general conditions to obtain conservative abstract interpretations of a system. These conditions are local to the verification process in each sub-formula of the given formula. Also, a fully automatic symbolic model checking abstraction algorithm was presented. The algorithm includes a set of techniques for providing incremental approximations for every type of sub-formula. Also included is a procedure for gradually refining these approximations, until the formula is verified or resources are exhausted.

The algorithm and techniques described above are being implemented inside the framework provided by VIS [2]. The first prototype of the algorithm correctly verified small examples, but after finishing its implementation, we plan to apply it to examples that are known to be hard to verify with conventional techniques.

In the future, we plan to enrich the set of approximation techniques provided in the algorithm. In particular, generic BDD sub-setting and super-setting may be enhanced by adapting them to the context of refining a previous approximation. We are also working on efficient techniques for computing *Pre* and *Img* when domain and co-domain constraint sets are given.

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 University, 1996. LNCS 1102.
3. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug. 1986.
4. J. R. Burch, E. M. Clarke, and D. E. Long. Representing circuits more efficiently in symbolic model checking. In *Proceedings of the Design Automation Conference*, pages 403–407, San Francisco, CA, June 1991.
5. 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.
6. 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.
7. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by constructions or approximation of fixpoints. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 238–250, 1977.
8. M. Dam. CTL* and ECTL* as fragments of the modal μ-calculus. *Theoretical Computer Science*, 126:77–97, 1994.
9. E. A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings of the First Annual Symposium of Logic in Computer Science*, pages 267–278, June 1986.

10. D. Geist and I. Beer. Efficient model checking by automated ordering of transition relation parititons. In D. L. Dill, editor, *Sixth Conference on Computer Aided Verification (CAV'94)*, pages 299–310, Berlin, 1994. Springer-Verlag. LNCS 818.
11. 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.
12. D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.
13. R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes.* Princeton University Press, Princeton, NJ, 1994.
14. 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, 1996.
15. D. E. Long. *Model Checking, Abstraction, and Compositional Verification.* PhD thesis, Carnegie-Mellon University, July 1993.
16. K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Boston, MA, 1994.
17. C. Pixley, S.-W. Jeong, and G. D. Hachtel. Exact calculation of synchronization sequences based on binary decision diagrams. In *Proceedings of the Design Automation Conference*, pages 620–623, Anaheim, CA, June 1992.