Advances in Symbolic Model Checking Techniques

Written Preliminary Examination II

Abhishek Udupa

audupa@cis.upenn.edu

March 11, 2013

Abstract

A common theme in symbolic model checking is to compute an inductive strengthening of the desired invariant, which forms a proof that no erroneous state can be reached by the system. The original symbolic model checking algorithm computed this inductive strengthening by computing (a hopefully succinct) representation of all reachable states by fixpoint computations and OBDDs. This set of reachable states is, by definition, closed under the transition relation. If no erroneous state is found to be in this set, then it forms an inductive invariant. Symbolic model checking techniques also handle liveness properties using fixpoint computations.

More recently, other techniques for computing inductive strengthenings, which are more efficient in practice, have been proposed. These techniques can broadly be classified as ‘abstraction based’ and ‘resolution based’. In this manuscript, we describe and compare three approaches for computing inductive strengthenings. The first approach computes the inductive strengthening over an abstracted version of the system, refining the abstracted system appropriately when the process fails due to loss of information from abstraction. The second approach uses proofs of unsatisfiability from bounded model checking queries to construct the inductive strengthening. The third approach on the other hand tries to construct the inductive strengthening incrementally by considering individual counterexamples to induction.

1 Introduction

Verification and model checking technologies have made significant strides over the last two decades and are capable of handling some large designs today. However, the state explosion problem, where the size of the state space of the system grows exponentially with the number of stateful components in the design, limits the scalability of the techniques. Explicit state model checkers such as SPIN [20] and Murϕ [16] employ techniques like partial order reduction [21] and symmetry reduction [22] to mitigate the state explosion problem. The key computational effort in model checking is to maintain a set of reachable states. As the state size grows, maintaining this set becomes increasingly difficult.

Symbolic Model Checking techniques using Ordered Binary Decision Diagrams (OBDDs) [10] address the state explosion problem by representing sets of states using Boolean formulas instead of an explicit representation. The original approach [10] computed a fixpoint of the transition relation applied to the initial set of states. This set is a representation of all the reachable states in the system and is thus closed under the transition relation by definition. If no erroneous state is part of this set, then the Boolean formula which represents the set of reachable states forms an inductive invariant.

The idea of using OBDDs to represent sets of states has enabled symbolic model checking techniques to scale to the task of verifying systems with an extremely large number of states [10]. However, although OBDDs often provide succinct representations of Boolean formulas, there exist Boolean formulas which do not have succinct representations [9]. Also, OBDDs are often unpredictable in their space and computational
requirements, depending on the variable order that has been chosen. Unfortunately, the problem of finding an optimal variable ordering has been shown to be hard \cite{9}.

Thus, although symbolic model checking techniques show promise, they are limited by the unpredictable nature of the underlying OBDD representations of sets of states. In this manuscript, we describe and compare three approaches proposed in recent literature that allow symbolic model checking techniques to overcome the limitations imposed by the use of OBDDs. All of them are variations on the common theme of computing an inductive invariant (or an inductive strengthening of an invariant) as a proof of correctness of the systems.

The first approach we describe abstracts the system under consideration. The abstract system can be considered as an over-approximation of the original system. Conceptually, the abstract system exhibits all the behaviors of the original system, but might exhibit some behaviors not exhibited by the original system as well. If the properties to be checked are of a certain class whose violations admit counterexamples, then any spurious counterexample resulting from the process of abstraction are used to refine the abstraction to eliminate at least that spurious counterexample and the process is repeated until either a real counterexample is discovered or the correctness of the abstract system is proved, which in turn implies that the unabstracted system is correct as well. Not surprisingly, this technique is called “Counter-Example Guided Abstract Refinement” (CEGAR) \cite{12}.

Another approach to avoid the dependence on unpredictable OBDD representations are Bounded Model Checking (BMC) techniques. In BMC, the transition system is unrolled to increasingly large degrees, as long as time and resource constraints permit. Each iteration proves that no erroneous state is reachable within $k$ steps from the initial set of states, for increasing values of $k$. While this approach can be used to detect bugs, it is not suited for proving the absence of bugs unless a bound on $k$ is known a-priori. The second technique we describe \cite{23} uses BMC techniques in an interesting way: for each BMC query which results in an unsatisfiable result (i.e., no erroneous state is reachable within $k$ steps from the initial state for some $k$), the algorithm extracts an interpolant from the proof of unsatisfiability. The interpolant is a formula which over-approximates the set of states reachable in one step from the initial state. A BMC query is now repeated with this over-approximation forming the set of initial states, to obtain an over-approximation of states reachable in two steps from the initial state and so on, until either a counterexample is discovered or the sequence of over-approximations converges to a fixpoint, at which time it forms an inductive invariant and a proof of correctness of the system.

The third approach we describe \cite{5, 6}, called ic3, attempts to perform this inductive strengthening in a more focused manner than interpolation based techniques. It attempts to learn small relatively inductive clauses which are similar to the lemmas discovered by a human being attempting a proof. These lemmas are then used to discover further lemmas until an inductive invariant is obtained. One of the interesting aspects of the ic3 approach is that it explores the trade-off between the number and size of satisfiability queries posed to the underlying decision procedure. More precisely, while the interpolant based approach makes a small number of large queries to the underlying solver (whose running time is typically exponential in the size of the queries), the ic3 approach makes a large number of small queries to the underlying solver. This approach works well in practice, especially on benchmarks where the underlying decision procedure is the limiting factor.

The rest of this manuscript is organized as follows: Section 2 reviews some background and establishes the common terminology used in the rest of the manuscript. Section 3 describes the details of the CEGAR algorithm for symbolic model checking. Section 4 describes the interpolation based algorithm, while Section 5 contains a presentation of the ic3 approach to symbolic model checking. We compare the strengths and weaknesses of the approaches along a few key dimensions in Section 6 and conclude in Section 7.

2 Preliminaries

In this section, we provide a brief background of propositional logic, transition systems, temporal logics and symbolic model checking of transition systems as well as define the relevant terminology. The terminology and notation defined in this section will be used uniformly throughout the rest of this manuscript.
2.1 Propositional Logic, CNF and QBF

**Definition 2.1** (Propositional Logic). A formula in propositional logic over a set of Boolean variables $V$ is inductively defined as follows: (1) A Boolean variable $v \in V$ is a formula (2) If $p$ is a formula then the negation of $p$, denoted $\neg p$, is a formula. (3) If $p$ and $q$ are formulas then their disjunction, denoted $p \lor q$, and their conjunction, denoted $p \land q$, are formulas as well.

Other commonly used propositional logic connectives like implication, equivalence, etc. can all be expressed in terms of these basic connectives.

A literal $l$ is either a Boolean variable $v \in V$ or its negation $\neg v$. A clause is a disjunction of literals. Given two clauses $c_1$ and $c_2$, we call $c_1$ a subclause of $c_2$, denoted $c_1 \subseteq c_2$, if every literal which appears in $c_1$ also appears in $c_2$. A formula in conjunctive normal form (CNF) is a conjunction of clauses. $\varphi(V)$ denotes that $\varphi$ is a propositional logic formula over the set of variables $V = \{v_1, v_2, \ldots, v_n\}$. An assignment $A$ associates a truth value from the set $\{\text{True}, \text{False}\}$ with each variable $v_i \in V$. We use $A[v]\equiv \varphi$ to denote the truth value of the formula $\varphi$ on an assignment $A$. We call $A$ a satisfying assignment iff $A[v] \equiv \text{True}$, and is denoted by $A \models \varphi$.

**Definition 2.2** (Quantified Boolean Formula). The Quantified Boolean Formulas (QBF) over the set of Boolean variables $V$ are defined inductively as follows: (1) Every formula $p$ in propositional logic is a quantified Boolean formula. (2) If $p$ is a quantified Boolean formula and $v \in V$, then $\exists v.p$ and $\forall v.p$ are quantified Boolean formulas as well.

The notion of an assignment carries over from its earlier definition in the context of propositional logic. Let $A$ be an assignment over $V$; let $f$ be a quantified Boolean formula over $V$ and let $v \in V$:

- $A \models \exists v. f$ iff $A[v \leftarrow \text{False}] \models f$ or $A[v \leftarrow \text{True}] \models f$.
- $A \models \forall v. f$ iff $A[v \leftarrow \text{False}] \models f$ and $A[v \leftarrow \text{True}] \models f$.

Here the notation $A[v \leftarrow \text{True}]$ ($A[v \leftarrow \text{False}]$) is used to indicate that the value of the variable $v$ in the assignment $A$ is coerced to $\text{True}$ ($\text{False}$) and the values mapped to other variables in $A$ remain unchanged.

We conclude this discussion on propositional logic and QBF by mentioning that satisfiability of a propositional logic formula is decidable and is $\text{NP-complete}$. A decision procedure for the satisfiability of a propositional logic formula is known as a SAT-solver. The satisfiability of a quantified Boolean formula is decidable as well and is $\text{PSpace-complete}$. A decision procedure for QBF is called a QBF-solver.

2.2 Transition Systems

We use the notation for transition systems presented in the text book by Clarke et al. [14], uniformly throughout this manuscript. A model of the system to be verified using model checking techniques is usually represented by a finite, labeled Kripke structure. A labeled Kripke structure is a directed graph with vertices labeled by sets of atomic propositions. Vertices in the Kripke structure represent states and the edges between them represent transitions. One or more states of the Kripke structure are considered to be initial states.

**Definition 2.3** (Kripke Structure). A Kripke structure over a set of atomic propositions $A$ is a tuple $M = (S, R, L, I)$, where $S$ is the set of states, $R \subseteq S \times S$ is the transition relation, $L : S \rightarrow 2^{A}$ is a labeling function and $I \subseteq S$ is a set of initial states.

We assume that $R$ is total. This assumption does not cause any loss of generality, since states with no transitions out of them can be represented as states with a self loop in the transition system. Also, note that the set of atomic propositions $A$ is defined over a set of variables $V$, where each $v \in V$ can take values from a finite domain.$^{[7]}$. This set of variables $V$ is implicitly assumed to be a part of the Kripke structure. Thus, a state $s \in S$ is equivalent to an assignment over $V$ and if $s$ forms a satisfying assignment for a formula $p$, we denote it by $s \models p$.

$^{[7]}$Any finite domain can be mapped to the Boolean domain by an appropriate mapping function, so $V$ can be considered to be a set of Boolean variables.
Definition 2.4 (Paths and Suffixes). A path $\pi$ is an infinite sequence of states, $\pi = (s_0, s_1, \ldots)$, such that $s_0 \in I$ and for all $i \geq 0$, $(s_i, s_{i+1}) \in R$. The suffix $\pi^*$ of a path $\pi$ is defined as $\pi^* = (s_i, s_{i+1}, \ldots)$. Since the transition relation is total, any finite prefix of a path can be extended to a path.

2.2.1 Symbolic Representation of Kripke Structures

An alternative view of a Kripke structure expressed symbolically is that of a tuple $M = (V, R, I)$, where $V = \{v_1, v_2, \ldots, v_n\}$ is a set of variables, where each $v_i$ can take values from the finite domain $D_{v_i}$. The set of all states is thus $S = \Pi_{1 \leq i \leq n} D_{v_i}$. $R(V, V')$ is a transition relation which relates values of variables in the current step to their values in the next step. $V'$ is a set of primed variables $\{v'_1, v'_2, \ldots, v'_n\}$ with each primed variable having the same domain as the corresponding unprimed variable from $V$ and exists solely to distinguish the current values of variables from their values in the next step. Note that $R$ is a formula.

The details of how to represent a relation by a formula can be found in [14] and will not be discussed here. $I$ is a formula that represents the set of initial states. It is easy to see the correspondence between this definition and the earlier definition. A state $s \in S$ is equivalent to an assignment of values to variables in $V$. A set of states can be represented by a symbolic formula $f$, which corresponds to the set of states $\{s \in S : s \vDash f\}$. $R$ is merely the transition relation expressed symbolically, i.e., the transition relation is the set of all pairs of states $(s, s')$ such that $(s, s') \vDash R$. The labeling function $L$ is implicit and can be considered as a mapping from atomic propositions $p$ to symbolic representations of subsets of states $S_{sub} \subseteq S$, with the interpretation that if $p \rightarrow S_{sub}$, then $S_{sub} \Rightarrow p$. We will use the explicit representation of a Kripke structure $M = (S, R, L, I)$ when defining the semantics of temporal logics, but will refer to the symbolic representation $M = (V, R, I)$ when describing the symbolic model checking algorithms later on.

2.2.2 From Programs to Kripke Structures

A Kripke structure is usually used as an abstract representation of a program whose properties are to be checked. Therefore, we now provide an outline of how programs can be translated into symbolic representations of Kripke structures. Consider a finite concurrent program $P$ with a finite set of variables $V = \{v_1, v_2, \ldots, v_n\}$. Where each variable $v_i$ has an associated finite domain $D_{v_i}$. The set of all possible states of $P$ is thus $D = D_{v_1} \times D_{v_2} \times \ldots \times D_{v_n}$. Expressions are constructed from variables in $V$, constants in $D_{v_i}$ and function symbols in the usual way, e.g. $v_1 + 7$. Atomic formulas are constructed using expressions and relation symbols, e.g. $v_1 + 7 < v_2$, and predicates are formed by composing atomic formulas using the usual Boolean operators of negation, disjunction and conjunction. Given a predicate $p$, we let $\text{Atoms}(p)$ denote the atomic formulas appearing in $p$. Further, we assume that the initial values of each variable $v_i$ in $P$ is denoted by $I_i \subseteq D_{v_i}$. The next values of each variable $v_i$ are represented symbolically using a guarded command language, where each guard $g_i^j$ is a predicate and each next value $A_i^j$ is an expression, with the semantics being that the value of $v_i$ is to be updated with the value obtained by evaluating $A_i^j$ in the current state, if the guard $g_i^j$ evaluates to $\text{True}$ in the current state. We call such a block, which gives the initial values $I_i$ and the updates for each variable $v_i$ as the transition block for $v_i$.

Now, given a program $P$ in the form described above and a temporal logic specification $\varphi$, we let $\text{Atoms}(P)$ denote the set of atomic formulas that appear in the condition blocks of $P$ or in $\varphi$. We can construct a symbolic representation of a Kripke structure $M = (V, R, I)$ as follows:

1. The set of variables maps directly to the set $V$ of variables in the Kripke structure.
2. The symbolic representation of the transition relation can be used to construct the symbolic transition relation $R$, which relates the current values of variables to their values in the next step. e.g., Suppose the guard is $v_1 > 3$ and the next value is $v_1 - 1$, then we can represent this particular transition as $v_1 > 3 \land v_1' = v_1 - 1$. Multiple updates for $v_1$ under different guards are expressed as disjunctions.
3. Updates to different variables are conjuncted to obtain the formula for $R$.

We let $I = I_1 \times I_2 \times \ldots \times I_n$ to be the set of initial states, which can again be represented as symbolic formulas from the program text.
2.3 Temporal Logic

Temporal logics are usually an extension of propositional logic used to describe properties of state transition systems or Kripke structures. We briefly discuss a temporal logic called CTL* and some useful fragments of CTL*. The presentation is along the lines of the presentation in the text book by Clarke et al. [13].

CTL* is an extension of propositional logic which allows for path quantifiers and temporal operators. The path quantifiers are the “for every path” operator A and the “there exists a path” operator E. The temporal operators and their intuitive semantics are as follows:

- $Xp$ — The “next time” operator: $p$ holds in the next time step.
- $Fp$ — The “future” or “eventually” operator: $p$ holds in some future time step.
- $Gp$ — The “global” operator: $p$ holds in all future time steps.
- $pUq$ — The “until” operator: $p$ holds until the time step that $q$ holds and $p$ must hold eventually.
- $pRq$ — The “release” operator: $q$ holds at least until the first time step that $p$ holds.

2.3.1 CTL* Syntax

Formulas in CTL* can either be state formulas — formulas that are true in a specific state — or path formulas — formulas which are true along a specific path. Recall that $A$ is the set of atomic propositions. The syntax of CTL* formulas over $A$ is defined inductively as follows:

1. If $p \in A$, then $p$ is a state formula.
2. If $p$ and $q$ are state formulas, then so are $\neg p$, $p \land q$ and $p \lor q$.
3. If $f$ is a path formula then $Ef$ and $Af$ are state formulas.
4. If $f$ is a state formula then $f$ is also a path formula.
5. If $f$ and $g$ are path formulas, then $\neg f$, $f \land g$, $f \lor g$, $Xf$, $Ff$, $Gf$, $fUg$ and $fRg$ are also path formulas.

2.3.2 CTL* Semantics

We define the semantics of CTL* with respect to a Kripke structure $M = (S, R, L, I)$. Given a state $s \in S$, a path $\pi$ in $M$, path formulas $f$ and $g$ and state formulas $p$ and $q$ we use $M, s \models p$ to denote that $p$ holds in the state $s$ and $M, \pi \models f$ to denote that the path formula $f$ holds along the path $\pi$ in $M$. Finally, we say that $M \models \varphi$ iff for all $s_i \in I$, $M, s_i \models \varphi$. The semantics of CTL* are described in Figure [1]

2.3.3 Some Useful Fragments of CTL*

LTL: LTL (or linear temporal logic) is the logic obtained by restricting CTL* formulas to the form $Af$, where $f$ is a path formula in which only atomic propositions are permitted as state subformulas. Thus a path formula in LTL is either (1) An atomic proposition $p \in A$, or (2) A formula of the form $\neg f$, $f \land g$, $Xf$, $Ff$, $Gf$, $fUg$ or $fRg$, where $f$ and $g$ are path formulas.

CTL: Formulas in CTL are characterized by having each of the temporal operators ($X$, $F$, $G$, $U$ and $R$) be immediately preceded by a path quantifier (E or A).

ACTL*: ACTL* is the logic obtained by allowing only universal path quantifiers in formulas, and allowing negations only on atomic propositions. An important feature of ACTL* is the existence of counter-examples. A counter-example to an ACTL* formula is simply a path $\pi$ in the Kripke structure, such that $\pi$ falsifies the formula.

ACTL: ACTL is obtained by only allowing the universal path quantifier and allowing negations only on atomic propositions, in addition to the restrictions on the logic CTL. Like ACTL*, ACTL is also characterized by the existence of counter-examples for formulas which are falsifiable.
1. \( M,s \models p \iff p \in L(s) \).
2. \( M,s \models \neg p \iff M,s \models p \).
3. \( M,s \models p \lor q \iff M,s \models p \lor M,s \models q \).
4. \( M,s \models p \land q \iff M,s \models p \) and \( M,s \models q \).
5. \( M,s \models Ef \iff \text{there exists a path } \pi \text{ beginning at } s, \text{ such that } M,\pi \models f \).
6. \( M,\pi \models Af \iff \text{for every path } \pi \text{ beginning at } s, M,\pi \models f \).
7. \( M,\pi \models f \iff M,s \models f \) where \( s \) is the first state in the path \( \pi \).
8. \( M,\pi \models \neg f \iff M,\pi \notmodels f. \)
9. \( M,\pi \models f \lor g \iff M,\pi \models f \) or \( M,\pi \models g. \)
10. \( M,\pi \models f \land g \iff M,\pi \models f \) and \( M,\pi \models g. \)
11. \( M,\pi \models Xf \iff M,\pi^1 \models f. \)
12. \( M,\pi \models Ff \iff \text{there exists } k \geq 0, \text{ such that } M,\pi^k \models f. \)
13. \( M,\pi \models Gf \iff \text{for all } i \geq 0, M,\pi^i \models f. \)
14. \( M,\pi \models f U g \iff \text{there exists } k \geq 0, \text{ such that } M,\pi^k \models g \) and for all \( 0 \leq j < k, M,\pi^j \models f. \)
15. \( M,\pi \models f R g \iff \text{for all } j \geq 0, \text{ if for every } i < j, M,\pi^i \notmodels f, \text{ then } M,\pi^j \models g. \)

Figure 1: Semantics of CTL* formulas

2.4 The Model Checking Problem

**Definition 2.5 (The model checking problem).** Given a Kripke structure \( M = (S,R,L,I) \) and a specification \( \varphi \) in a temporal logic such as CTL*, the model checking problem is to determine for all initial states \( s_i \in I \), whether \( M,s_i \models \varphi. \)

An explicit state model checker performs the model checking directly on an explicit representation of the given Kripke structure. We do not discuss explicit state model checking in detail in this manuscript and restrict our attention to symbolic model checking algorithms in the rest of this subsection.

2.4.1 Symbolic Model Checking

Symbolic model checking algorithms check properties of a Kripke structure represented symbolically. These algorithms use symbolic formulas to represent reachable sets of states. Specifically, the sets of states reachable in \( k \geq 0 \) steps from the initial states are represented using symbolic formulas. The formulas are often encoded as Ordered Binary Decision Diagrams (OBDDs) [9]. OBDDs usually provide a succinct notation for Boolean formulas, but can be unpredictable with respect to their succinctness [12]. A useful feature of OBDDs is that they provide a canonical representation of Boolean formulas, making equivalence and subsumption checks efficiently computable. We only discuss model checking problems over finite domains, so we restrict our attention to the Boolean domain and observe that any finite domain can be mapped to the Boolean domain with an appropriate mapping function.

**Image and Pre-image computation.** The Image of a set of states \( S \) with respect to a transition relation \( R \) is defined as \( \text{Img}(S,R) := \{ t : \exists s.R(s,t) \land s \in S \} \). i.e., the image of a set of states \( S \) is precisely the set of states which the transition system can reach in one step if the transition system is in any one of the states in the set \( S \). The Pre-image of a set of states \( S \) is defined as \( \text{PreImg}(S,R) := \{ t : \exists s.R(t,s) \land s \in S \} \).

**Model Checking Safety Properties.** A safety property is of the form \( AGf \), where \( f \) does not contain any temporal operators or path quantifiers. Intuitively, a safety property states that “something bad does not happen”. Safety properties can be checked by repeatedly applying the \( \text{Img} \) operator to the set \( I \) until fixpoint. Intuitively, this represents all the reachable states of the system. Let the formula representing this fixpoint be \( F_{\text{fix}} \). Now the system violates the safety property iff \( \neg f \land F_{\text{fix}} \) is satisfiable. We also refer to a safety property as an invariant.

**Model Checking Liveness Properties.** Intuitively, A liveness property states that “something good eventually happens”. Checking a Kripke structure for liveness properties is usually done by using fixpoint
representations of the properties. We do not discuss the details in this manuscript. A detailed description can be found in the text book by Clarke, et al. [14]. Recent work has also shown how liveness checking can be reduced to safety checking [4].

3 CEGAR for Symbolic Model Checking

In this section, we describe a framework for (1) Abstracting a Kripke structure and (2) Iteratively refining the abstraction using counter-examples until the abstraction is fine enough to prove or refute the property to be checked. The framework was proposed by Clarke, et al. [12] and is called Counter-Example Guided Abstraction Refinement (CEGAR).

3.1 Overview of the CEGAR Approach

Figure 2 provides an overview of the CEGAR approach to symbolic model checking. Given a Kripke structure \(M\) and a specification as an ACTL* formula \(\varphi\), the goal is to check if \(M \models \varphi\). The steps in the CEGAR approach to this end are as follows:

1. **Generate the Initial Abstraction.** Informally, an abstraction \(\hat{M}\) of a Kripke structure \(M\) allows for a superset of behaviors that \(M\) allows. In other words, every path \(\pi\) in \(M\) has a corresponding path in \(\hat{M}\), but the converse need not hold. Thus, it is simple to see that any counterexample in \(M\) will have a corresponding counterexample in \(\hat{M}\), however a counterexample in \(\hat{M}\) could be a spurious counterexample, if it does not have a corresponding path in \(M\). Section 3.3 describes the process of computing the initial abstraction in detail.

2. **Model Check the Abstract Kripke Structure.** Suppose \(\hat{M} \models \varphi\), then this implies that \(M \models \varphi\) and the process terminates reporting a success. On the other hand if \(\hat{M} \not\models \varphi\), then the check reveals an abstract counterexample \(\hat{T}\). Recall that a characteristic feature of ACTL* specification is the existence of counterexamples. Thus if \(\hat{M} \not\models \varphi\) the existence of a counterexample to the specification \(\varphi\) is guaranteed to exist since \(\varphi\) is an ACTL* formula. If \(\hat{T}\) corresponds to a counterexample in \(M\) as well, then we have discovered an actual counterexample and the process terminates, reporting that \(M \not\models \varphi\). If \(\hat{T}\) is a spurious counterexample, the process proceeds to the next step.

3. **Refine the Abstract Kripke Structure.** In this step, the abstract Kripke structure \(\hat{M}\) is refined to obtain \(\hat{M}_{\text{ref}}\), which has the property that it does not admit at least the spurious counterexample \(\hat{T}\). Once a refinement has been computed, the refined Kripke structure is model checked again. This process is iterated until the refinement is sufficient to prove that \(M \models \varphi\) or a real counterexample is found. Section 3.4 describes this iterative process in detail.

The rest of this Section is organized as follows: Section 3.2 discusses some preliminaries and introduces new terminology. Section 3.3 describes how the initial abstraction is computed. Section 3.4 describes the iterative
refinement process and Section 3.5 summarizes the results of using the CEGAR approach to model check some industrial strength systems.

3.2 Preliminaries and Definitions

3.2.1 Simulation and ACTL* Specifications

Definition 3.1 (Simulation). Given two Kripke structures \( M = (S, R, L, I) \) and \( M' = (S', R', L', I') \), over the sets of atomic propositions \( \mathcal{A} \) and \( \mathcal{A}' \) respectively, with \( \mathcal{A}' \subseteq \mathcal{A} \), a relation \( H \subseteq S \times S' \) is a simulation relation iff for all \( (s, s') \in H \), the following conditions hold: (1) \( L(s) \cap \mathcal{A}' = L'(s') \). (2) For each state \( s_1 \) such that \( (s, s_1) \in R \), there exists a state \( s'_1 \) with the property that \( (s'_1, s_1') \in R' \) and \( (s_1, s_1') \in H \). \( M' \) is said to simulate \( M \), denoted \( M \preceq M' \), if there exists a simulation relation \( H \) such that for each initial state \( s_0 \) in \( M \), there exists an initial state \( s'_0 \) in \( M' \) such that \( (s_0, s'_0) \in H \).

Intuitively, if \( M \preceq M' \), then \( M' \) is an over-approximation of \( M \), in that, for every path in \( M \) there is a corresponding path in \( M' \) as well, but \( M' \) may contain paths which do not correspond to any path in \( M \). This notion is made precise in the following theorem by Clarke, et al. [13].

Theorem 3.1. For every ACTL* formula \( \varphi \) over a set of atomic propositions \( \mathcal{A}' \), if \( M \preceq M' \) and \( M' \models \varphi \) then \( M \models \varphi \).

3.2.2 Abstraction

Intuitively, abstraction amounts to partitioning the states of a Kripke structure into clusters and treating the clusters as abstract states. An abstraction function \( h \) is a surjection \( h : S \to \hat{S} \), where \( \hat{S} \) is the set of abstract states. Note that \( h \) induces an equivalence relation \( \equiv_h \) as follows: If \( d \) and \( e \) are states in \( S \), then, \( d \equiv_h e \iff h(d) = h(e) \).

The abstract Kripke structure \( \hat{M} = (\hat{S}, \hat{R}, \hat{L}, \hat{I}) \) generated from \( M = (S, R, L, I) \), using the abstraction function \( h \) is defined as follows:

1. \( \hat{I}(\hat{d}) \iff \exists d(h(d) = \hat{d} \land I(d)) \).
2. \( \hat{R}(\hat{d}_1, \hat{d}_2) \iff \exists d_1, d_2 (h(d_1) = \hat{d}_1 \land h(d_2) = \hat{d}_2 \land R(d_1, d_2)) \).
3. \( \hat{L}(\hat{d}) = \cup_{h(d') = \hat{d}} L(d') \).

We use the notation \( \hat{M}_h \) to denote the abstract Kripke structure \( \hat{M} \) has been generated using the abstraction function \( h \).

Definition 3.2 (Appropriate abstraction functions). Given a specification \( \varphi \) and a set of states \( S \), an abstraction function \( h \) is called appropriate for \( \varphi \) if for all states \( d, e \in S \), where \( d \equiv_h e \), and for all atomic subformulas \( f \) of \( \varphi \), it holds that \( d \models f \iff e \models f \).

The notion of appropriateness carries over naturally to sets of formulas; An abstraction function \( h \) is appropriate for a set \( \mathcal{F} \) of formulas, if \( h \) is appropriate for every \( f \in \mathcal{F} \).

Theorem 3.2. Let \( h \) be an appropriate abstraction function for an ACTL* specification \( \varphi \) and let \( M \) be defined over the atomic propositions in \( \varphi \). Then \( \hat{M}_h \models \varphi \Rightarrow M \models \varphi \).

Proof. First, we observe that \( M \preceq \hat{M}_h \), by choosing the simulation relation \( H = \{(s, h(s)) : s \in S \} \). Given that \( M \preceq \hat{M}_h \), from Theorem 3.1 we have that \( \hat{M}_h \models \varphi \Rightarrow M \models \varphi \).

From Theorem 3.2 we can conclude that the abstract Kripke structure \( \hat{M}_h \), generated using an appropriate abstraction function \( h \), does not admit false positives. Note however, that \( \hat{M}_h \) could admit false negatives or spurious counterexamples.
Definition 3.3 (F-equivalent). Let F be a set of formulas. Given two states d, e ∈ S, d and e are F-equivalent, denoted d ≡F e, if for all f ∈ F, we have d |= f ⇔ e |= f. The abstraction function corresponding to the equivalence relation ≡F is denoted hF.

Lemma 3.3. Let ϕ be an ACTL* specification. For any set F, such that F ⊇ Atoms(ϕ), hF is an appropriate abstraction function for ϕ.

Proof. Suppose hF is not appropriate for ϕ. Then there must exist states d and e such that d ≡F e, and d |= f, e ̸|= f, where f ∈ Atoms(ϕ). But Atoms(ϕ) ⊆ F, contradicting the definition of ≡F.

3.3 Computing the Initial Abstraction

Consider a program P, with a finite set of variables V = {v1, v2, . . . , vn}, with each variable vi having an associated domain Di, as described in Section 2.2.2. We wish to construct an initial abstract Kripke structure directly from the program text. Recall that given a program P and an ACTL* specification ϕ, Atoms(P) is the set of atomic formulas that appear in the guards of P or in ϕ. Suppose we let the initial abstraction function hinit = hAtoms(P), then from Lemma 3.3 we have that hinit is appropriate for ϕ.

Intuitively, generating the initial abstraction using hinit preserves the “control flow skeleton” of the program P. However, the domain of hinit is D = Dv1 × Dv2 × . . . × Dvn. Since the domain could be very large in practice, representing hinit directly could become intractable.

To overcome this limitation, composite abstraction functions, which have been proposed in earlier literature [13], are used to the extent possible.

Definition 3.4 (Composite Abstraction Function). Suppose the set of states S of a Kripke structure are obtained as a product of the smaller domains D1 × D2 × . . . × Dn and we define the surjections hi : Di → ˆDi, then function h : S → ˆS, where ˆS = ˆD1 × ˆD2 × . . . × ˆDn and h(d1, d2, . . . , dn) = (ˆh1(d1), ˆh2(d2), . . . , ˆhn(dn)) is the composite abstraction function with components ˆh1, ˆh2, . . . , ˆhn.

The advantage of representing h as a composite abstraction function is that h can be represented more compactly if the domains of each hi are small. Although not every abstraction function can be represented as a composite abstraction function, the CEGAR approach presents a technique by which hinit can be represented as a composite abstraction function.

The idea is to partition the set of n program variables V = {v1, v2, . . . , vn} into m variable clusters VC1, VC2, . . . , VCM, where the domain of each variable cluster VCi is DVCi = ∏v∈VCi Dv. Once the variables have been partitioned, the abstraction function hinit is then computed as a composite abstraction function with surjections on each DVCi, as component abstraction functions. The only question that remains then is of how to partition the variables into clusters.

Given an atomic formula f, let var(f) be the variables appearing in f. We say that two atomic formulas f and g interfere if var(f) ∩ var(g) ̸= ∅. Now, given the set Atoms(P), let ≡I be the equivalence relation on Atoms(P) obtained by taking the reflexive, transitive closure of the interference relation. The equivalence class of a formula f ∈ Atoms(P) is called the formula cluster of f and is denoted by [f]. Now, given two formulas f1, f2 ∈ Atoms(P), [f1] = [f2] ⇔ var(f1) ∩ var(f2) ̸= ∅. The equivalence relation ≡I on Atoms(P) induces a dual equivalence relation ≡V on the set of variables V as follows:

\[ v_i \equiv_V v_j \text{ if and only if there exist formulas } f_1, f_2 \in \text{Atoms}(P), \text{ such that } [f_1] = [f_2] \text{ and } v_i \in \text{var}(f_1) \text{ and } v_j \in \text{var}(f_2). \]

Thus, the variable clusters {VC1, VC2, . . . , VCM} are given by the equivalence classes induced by ≡V. Let {FC1, FC2, . . . , FCM} be the corresponding set of formula clusters induced by ≡I. The composite abstraction function h has components h1, h2, . . . , hm, where hi : DVCi → ˆDVCi is defined as follows:

\[ h_i(d_1, d_2, . . . , d_k) = h_i(e_1, e_2, . . . , e_k) \text{ iff } \bigwedge_{f \in FC_i} (d_1, d_2, . . . , d_k) \models f \Leftrightarrow (e_1, e_2, . . . , e_k) \models f. \]
Intuitively, two values in a variable cluster $VC_i$ are in the same equivalence class if they cannot be distinguished by atomic formulas in the corresponding formula cluster $FC_i$. The following theorem states that the abstract Kripke structure $\tilde{M}_h$ generated by $h$ constructed as above is isomorphic to the abstract Kripke structure $\tilde{M}_{init}$, generated by $h_{init}$.

**Theorem 3.4.** $\tilde{M}_h$ is isomorphic to $\tilde{M}_{init}$

**Proof.** We will show that for any two states $d$ and $e$ in $D$, $d \equiv_{init} e$ iff $d \equiv_h e$. Let $\pi_{VC_i}$ be a function that maps a tuple $d \in D$ to the subtuple corresponding to $VC_i$. Consider any $f \in FC_i$, then by definition $f$ depends only on variables in $VC_i$. Thus, $d \models f$ iff $\pi_{VC_i}(d) \models f$. Also, since by definition the variable clusters and the formula clusters form a partition, we can conclude that for all $f \in FC_i$, $f$ does not depend on variables from any other variable cluster $VC_j, j \neq i$. Thus, by using the appropriate definitions we obtain:

$$d \equiv_{init} e \iff \bigwedge_{f \in \text{Atoms}(P)} d \models f \iff e \models f$$

$$\iff \bigwedge_{1 \leq i \leq m} \bigwedge_{f \in FC_i} \pi_{VC_i}(d) \models f \iff \pi_{VC_i}(e) \models f$$

$$\iff d \equiv_h e$$

Thus $\tilde{M}_h$ is isomorphic to $\tilde{M}_{init}$, and we conclude the discussion on how to construct an appropriate initial abstraction function.

### 3.4 Model Checking and Refining the Abstraction

We now discuss the problem of model checking an abstract Kripke structure $\tilde{M}$, generated for a program $P$, from an abstraction function $h$ which is appropriate for an ACTL* specification $\varphi$. Applying standard symbolic model checking techniques using OBDDs on $\tilde{M}$, suppose $\tilde{M} \models \varphi$, then from Theorem 3.2 we can conclude that the unabstracted Kripke structure $M$ corresponding to $P$ satisfies $\varphi$ as well. On the other hand if $\tilde{M} \not\models \varphi$, then there exists an abstract counterexample $\tilde{T}$. We then need to ascertain whether $\tilde{T}$ is spurious or not. If $\tilde{T}$ is not spurious, then it is clear that $M \not\models \varphi$, but if $\tilde{T}$ is spurious, then we need to refine $\tilde{M}$ to eliminate $\tilde{T}$. We describe each of these processes in detail in this section.

#### 3.4.1 Identification of Spurious Counterexamples

An abstract counterexample $\tilde{T}$ can be a path or a loop. Generally, witnesses to violation of safety properties are finite paths, while witnesses to violation of liveness properties involve loops. We deal with identifying path and loop counterexamples separately.

**Identifying Spurious Path Counterexamples.** Suppose $\tilde{T}$ is a path counterexample consisting of the sequence of abstract states $(\tilde{s}_1, \tilde{s}_2, \ldots, \tilde{s}_n)$. Given an abstract state $\tilde{s}$, We define $h^{-1}(\tilde{s}) = \{s \in D : h(s) = \tilde{s}\}$. We extend $h^{-1}$ to sequences in the natural way:

$$h^{-1}(\tilde{T}) = \{ (s_1, s_2, \ldots, s_n) : \bigwedge_{i=1}^{n} h(s_i) = \tilde{s}_i \land I(s_1) \land \bigwedge_{i=1}^{n-1} R(s_i, s_{i+1}) \}$$

where $R$ and $I$ are the transition relation and the set of initial states of the unabstracted Kripke structure $M$ respectively. In other words, $h^{-1}(\tilde{T})$ is a set of concrete paths corresponding to $\tilde{T}$.

$h^{-1}(\tilde{T})$ can be computed as follows: We let $S_1 = h^{-1}(\tilde{s}_1) \cap I$. For $i \in [2, n]$, we let $S_i = \text{Img}(S_{i-1}, R) \cap h^{-1}(\tilde{s}_i)$. This sequence of sets $S_i$ can be computed symbolically using OBDDs.
Theorem 3.5. The following are equivalent:

(i) The path $\hat{T}$ corresponds to a concrete counterexample.

(ii) The set of concrete paths $h^{-1}(\hat{T})$ is non-empty.

(iii) For all $1 \leq i \leq n$, $S_i \neq \emptyset$.

Proof. (i) $\Rightarrow$ (ii). Suppose that $\hat{T}$ corresponds to a concrete counterexample $T = \langle s_1, s_2, \ldots, s_n \rangle$. Then, it must be the case that $h(s_i) = \tilde{s_i}$, for $1 \leq i \leq n$. Further, since $T$ is a trace in the concrete Kripke structure, $s_1 \in I$ and $R(s_i, s_{i+1})$ for all $1 \leq i < n$. Combining this with the definition of $h^{-1}(\hat{T})$ we get $T \in h^{-1}(\hat{T})$, and thus $h^{-1}(\hat{T}) \neq \emptyset$.

(ii) $\Rightarrow$ (iii). Suppose that $h^{-1}(\hat{T}) \neq \emptyset$. Then there exists a concrete path $T = \langle s_1, s_2, \ldots, s_n \rangle \in h^{-1}(\hat{T})$ and $h(s_i) = \tilde{s_i}$ for $1 \leq i \leq n$ and $s_1 \in I$. Now, since $s_1 \in h^{-1}(\tilde{s_1})$, we have established that $S_1 = h^{-1}(\tilde{s_1}) \cap I$ is non-empty. Assume that $s_i \in S_i$, for $i > 1$ as the induction hypothesis. Then, by the definition of $h^{-1}(\hat{T})$, there must exist a state $s_{i+1}$ such that $R(s_i, s_{i+1})$, and $s_{i+1} \in h^{-1}(\tilde{s_{i+1}})$. The first observation gives us that $s_{i+1} \in Img(S_i, R)$. Since $S_{i+1} = Img(S_i, R) \cap h^{-1}(\tilde{s_{i+1}})$, we have that $S_{i+1} \neq \emptyset$, completing the inductive proof.

(iii) $\Rightarrow$ (i). Suppose that $S_i \neq \emptyset$ for all $1 \leq i \leq n$. Assume that $s_n \in S_n$, we construct a concrete counterexample trace backwards inductively. Assume that $s_i \in S_i$ corresponds to a concrete state in $M$. Then from the definition of $S_i$, we must have that $s_i \in h^{-1}(\tilde{s_i})$ and $s_i \in Img(S_{i-1}, R)$. This implies that there must exist some state $s_{i-1} \in S_{i-1}$ such that $R(s_{i-1}, s_i)$. Now, by the definition of $S_{i-1}$, we have that $S_{i-1} \subseteq h^{-1}(\tilde{s_{i-1}})$, which implies that $s_{i-1} \in h^{-1}(\tilde{s_{i-1}})$, thus establishing that $s_{i-1}$ is indeed a concrete state in $M$. Finally, we have that $s_1 \in S_1$ and from the definition of $S_1$, we conclude that $s_1 \in I$ and $s_1 \in h^{-1}(\tilde{s_1})$, concluding the proof. \hfill \square

From Theorem 3.5, we conclude that if $\hat{T}$ does not correspond to a concrete counterexample that there exists a minimal $k$, $2 \leq k \leq n$, such that $S_k = \emptyset$. We call the states in $S_{k-1}$ as dead-end states, which will be used to refine the abstraction as we will discuss in Section 3.4. On the other hand if $\hat{T}$ corresponds to a real counterexample, we simply report it and stop.

Identifying Spurious Loop Counterexamples. We now consider the case when $\hat{T}$ includes a loop and is of the form $\langle \tilde{s_1}, \tilde{s_2}, \ldots, \tilde{s_l}, \tilde{s}_{i+1}, \tilde{s}_{i+2}, \ldots, \tilde{s_n} \rangle^\omega$. That is, the loop includes a non-repeating prefix called a stem $(\langle \tilde{s_1}, \ldots, \tilde{s_l} \rangle)$ and a suffix which repeats infinitely called a lasso $(\langle \tilde{s_{i+1}}, \tilde{s_{i+2}}, \ldots, \tilde{s_n} \rangle^\omega)$. Abstract loop counterexamples are more complicated to handle than path counterexamples: (1) An abstract loop can correspond to multiple concrete loops of different sizes. (2) Each of the concrete loops may start at different stages of the unwinding of the lasso part of the loop counterexample (3) In general, an exponential number of unwindings may be required before the unwinding becomes periodic. i.e., unwinding as many times as the least common multiple of the size of the individual concrete loops is guaranteed to be sufficient for the unwinding to become periodic.

Clarke, et al. \cite{clarke1987}, however prove that $\min$ unwindings of the lasso part of the loop are sufficient to check if a loop counterexample $\hat{T}$ corresponds to a concrete counterexample or not, where $\min$ is the minimum size of all abstract states in the loop, i.e., $\min = \min_{1 \leq i \leq n} |h^{-1}(\tilde{s_i})|$. We let $\hat{T}_{\text{unwind}}$ denote the abstract path $\langle \tilde{s_1}, \tilde{s_2}, \ldots, \tilde{s_l}, \tilde{s}_{i+1}, \tilde{s}_{i+2}, \ldots, \tilde{s_n} \rangle^{\min+1}$, i.e., the path obtained by unwinding the lasso part of $\hat{T}$ $\min$ times.

Theorem 3.6. The following are equivalent:

(i) $\hat{T}$ corresponds to a concrete loop counterexample.

(ii) $h^{-1}(\hat{T}_{\text{unwind}})$ is not empty.

Proof. The proof relies on the proof of Theorem 3.5. The idea is to reduce a loop counterexample to a path counterexample, after which Theorem 3.5 can be applied. Suppose $\hat{T} = \langle \tilde{s_1}, \tilde{s_2}, \ldots, \tilde{s_l}, \tilde{s}_{i+1}, \tilde{s}_{i+2}, \ldots, \tilde{s_n} \rangle^\omega$, and $\hat{T}_{\text{unwind}}$ is as defined earlier. We represent the sets of finite $R$-paths corresponding to $\hat{T}_{\text{unwind}}$ as:
is spurious then there must exist a set counterexamples. Recall from the construction of sets refining a spurious counterexample using the techniques described in Section 3.4.1, we now turn our attention to

\begin{algorithm}
\textbf{Algorithm 1: PolyRefine} \\
1. for \( j := 1 \) to \( m \) do \\
2. \( \equiv'_j := \equiv_j \) \\
3. for every \( a, b \in E_j \) do \\
4. if \( \text{proj}(S_D, j, a) \neq \text{proj}(S_D, j, b) \) then \\
5. \( \equiv'_j := \equiv_j \setminus \{(a, b)\} \)
\end{algorithm}

\[(S_0^0, S_1^0, \ldots, S_i^0, S_i^1, S_{i+1}^1, \ldots, S_n^1, S_0^2, S_1^2, \ldots, S_i^2, S_{i+1}^2, \ldots, S_n^2, \ldots, S_i^{min+1}, S_{i+1}^{min+1}, \ldots, S_n^{min+1})\]

where the sets \( S_j^k \) are defined as follows:

1. \( S_0^0 = h^{-1}(\hat{s}_1) \cap I \).
2. \( S_j^k = \text{Img}(S_{j-1}^k, R) \cap h^{-1}(\hat{s}_j) \), for \( 0 \leq k \leq \text{min} + 1 \), \( 1 < j \neq i + 1 \leq \text{min} \).
3. \( S_1^{i+1} = \text{Img}(S_0^i, R) \cap h^{-1}(\hat{s}_{i+1}) \).
4. \( S_k^{i+1} = \text{Img}(S_n^{i-1}, R) \cap h^{-1}(\hat{s}_{i+1}) \), for \( 1 < k \leq \text{min} + 1 \).

It is easy to see that the definition the sets \( S_j^k \) correspond to a set of \( R \)-paths, much like their corresponding definitions in the proof of Theorem 3.5. So, the only thing left to prove is that \( T_{\text{unwind}} \) corresponds to a loop counterexample if and only if \( T_{\text{unwind}} \) corresponds to a path counterexample.

Suppose that \( T_{\text{unwind}} \) corresponds to a path counterexample, then by Theorem 3.5 we have that \( S_j^k \neq \emptyset \) for all \( j, k \). Let \( m \) be the index of the abstract state with the minimum size \((|h^{-1}(s_m)| = \text{min})\), \( i + 1 \leq m \leq n \). Then we claim that some concrete state must repeat in the \( k + 1 \) sets \( S_m^k \), \( 1 \leq k \leq \text{min} + 1 \). The proof of this claim is easy to see from the pigeonhole principle, since each \( S_m^k \) can contain at most \( \text{min} \) elements (since each \( S_m^k \) is defined by \( h^{-1} \)) and it is impossible for \( \text{min} + 1 \) such sets to contain elements all distinct from one another. Thus, if \( T_{\text{unwind}} \) corresponds to a path counterexample then it also corresponds to a loop counterexample.

On the other hand if \( T_{\text{unwind}} \) does not correspond to a path counterexample, then some \( S_j^k \) must be empty from Theorem 3.5. Thus \( T_{\text{unwind}} \) cannot correspond to a loop counterexample either.

3.4.2 Refining the Abstraction

Once we have identified a counterexample obtained by model checking the abstract Kripke structure as a spurious counterexample using the techniques described in Section 3.4.1, we now turn our attention to refining the abstraction to eliminate at least the spurious counterexample obtained. We first consider path counterexamples. Recall from the construction of sets \( S_i \) described in Section 3.4.1 that if a counterexample \( \hat{T} \) is spurious then there must exist a set \( S_i, 1 \leq i < n \) such that \( S_i \subseteq h^{-1}(\hat{s}_i) \) and \( \text{Img}(S_i, R) \cap h^{-1}(\hat{s}_{i+1}) = \emptyset \).

The states in \( S_i \) are reachable from the initial set of states. There is no transition from any state in \( S_i \) to a state in \( h^{-1}(\hat{s}_{i+1}) \). However, there exists at least one transition from a state in \( h^{-1}(\hat{s}_i) \) to a state in \( h^{-1}(\hat{s}_{i+1}) \), since there is a transition from \( \hat{s}_i \) to \( \hat{s}_{i+1} \) in the abstract Kripke structure. We can partition \( h^{-1}(\hat{s}_i) \) into three subsets:

1. Dead-end states denoted by \( S_D \). This is simply the set \( S_i \).
2. Bad states denoted by \( S_B \). This is defined as the set \( \{s \in h^{-1}(\hat{s}_i) : \exists s' \in h^{-1}(\hat{s}_{i+1}), R(s, s')\} \).
3. Irrelevant states denoted by \( S_I \), which is defined as the set \( h^{-1}(\hat{s}_i) \setminus (S_D \cup S_B) \).

Intuitively, the states in \( S_D \) are reachable from the set of initial states, while the states in \( S_B \) are not reachable from the set of initial states. However, because the abstraction function maps some states in \( S_D \) and \( S_B \) to
the same equivalence class, we observe a spurious counterexample. Hence, the refined abstraction needs to separate the sets \( S_D \) and \( S_B \).

Recall that the abstraction function was defined in terms of equivalence classes and that \( h^{-1}(\tilde{s}_i) \) is an equivalence class of the equivalence relation \( \equiv \) which has the form \( E_i \times E_2 \times \ldots \times E_m \), where each \( E_i \) is an equivalence class of the component equivalence relation \( \equiv_i \). Thus, refining \( \equiv \) involves partitioning the equivalence classes \( E_j \), which essentially amounts to refining \( \equiv_j \). Formally \( \equiv' \) is a refinement of \( \equiv \) if for all \( 1 \leq j \leq m \), \( \equiv'_j \subseteq \equiv_j \). We denote this refinement relation as \( \equiv' \subseteq \equiv \).

The size of the refinement is the number of new equivalence classes. Ideally, we would like a refinement with the smallest size or the coarsest refinement. However, the following theorem from [12] states that it is intractable to find the coarsest refinement.

**Theorem 3.7.** The problem of finding the coarsest refinement is NP-HARD.

Clarke et al., thus propose a heuristic algorithm for refining the abstraction, which is shown in Algorithm [1].

The algorithm assumes a composite abstraction function with equivalence relations \( \equiv_1, \equiv_2, \ldots, \equiv_m \) over the domain \( D = D_1 \times D_2 \times \ldots \times D_m \). The algorithm uses the projection function, which is defined as follows:

Given a set \( X \subseteq D \), an index \( j \in [1, m] \) and an element \( a \in D_j \), the projection set \( \text{proj}(X, j, a) \) is defined as:

\[
\text{proj}(X, j, a) = \{(d_1, d_2, \ldots, d_j-1, d_{j+1}, \ldots, d_m) : (d_1, d_2, \ldots, d_{j-1}, a, d_{j+1}, \ldots, d_m) \in X\}
\]

We assume initially that \( S_1 = \emptyset \) and show that if \( \text{proj}(S_D, j, a) \neq \text{proj}(S_D, j, b) \), then it is necessary that any refinement \( \equiv'_j \) be such that \( a \neq'_j b \).

**Lemma 3.8.** Suppose \( S_1 = \emptyset \). If there exist \( a, b \in D_j \) such that \( \text{proj}(S_D, j, a) \neq \text{proj}(S_D, j, b) \) then every refinement \( \equiv' \subseteq \equiv \) must distinguish \( a \) and \( b \), i.e., \( a \neq'_j b \).

**Proof.** Suppose that there exist \( a, b \in D_j \) such that \( \text{proj}(S_D, j, a) \neq \text{proj}(S_D, j, b) \), but \( a \equiv'_j b \). We prove that \( \equiv'_j \) does not separate \( S_D \) and \( S_B \). Assume that there exists \((d_1, \ldots, d_{j-1}, a, d_{j+1}, \ldots, d_m) \in \text{proj}(S_D, j, a)\), but \((d_1, \ldots, d_{j-1}, d_{j+1}, \ldots, d_m) \notin \text{proj}(S_D, j, b)\). Then by the definition of the projection function the state \( s_1 = (d_1, \ldots, d_{j-1}, a, d_{j+1}, \ldots, d_m) \in S_D \), and the state \( s_2 = (d_1, \ldots, d_{j-1}, b, d_{j+1}, \ldots, d_m) \notin S_D \). Since \( S_D \), \( S_B \) and \( S_1 \) form a partition of \( h^{-1}(\tilde{s}_i) \) and since \( S_1 = \emptyset \), we conclude that \( s_2 \in S_B \). However, since \( a \equiv'_j b \), it follows that \( s_1 \equiv'_j s_2 \) and thus \( \equiv'_j \) does not separate \( S_D \) and \( S_B \).

**Lemma 3.8** gives a necessary condition that the refinement needs to satisfy. Now we show that the necessary condition is also a sufficient condition.

**Lemma 3.9.** When \( S_1 = \emptyset \), the relation \( \equiv' \) computed by the algorithm PolyRefine is an equivalence relation which refines \( \equiv \) and also separates \( S_D \) and \( S_B \).

**Proof.** The proof that \( \equiv' \) is indeed an equivalence relation follows from the definition of \( \equiv'_j \) as computed by the algorithm PolyRefine and is omitted. We focus on the proof that \( \equiv' \) is a correct refinement, i.e., for all states \( d \in S_D \) and \( b \in S_B \), it holds that \( d \equiv' b \). Suppose that \( b \in S_B \) and \( d \in S_D \) and \( b \equiv d \). We know that \( b \equiv d \) since they correspond to the abstract failure state \( \tilde{s}_i \). Suppose \( b \) and \( d \) have the form \( b = (b_1, b_2, \ldots, b_m) \) and \( d = (d_1, d_2, \ldots, d_m) \), we construct the sequence of states \( x_1, x_2, \ldots, x_{m+1} \) as follows:

\[
\begin{align*}
x_1 &= (b_1, b_2, \ldots, b_m) = b \\
x_2 &= (d_1, b_2, \ldots, b_m) \\
& \vdots \\
x_m &= (d_1, d_2, \ldots, d_{m-1}, b_m) \\
x_{m+1} &= (d_1, d_2, \ldots, d_m) = d
\end{align*}
\]

Consider \( x_1 = b \) and \( x_2 \). Since \( b \equiv d \), we know that \( b_1 \equiv d_1 \). But by the definition of \( \equiv' \) this implies that \( \text{proj}(S_D, 1, b_1) = \text{proj}(S_D, 1, d_1) \), i.e., for all \( d_1, d_2, \ldots, d_m \), it holds that
Proceeding in this manner, we obtain similar equivalences for any two neighboring states \( x_i \) and \( x_{i+1} \), and finally, we end up with \( b = x_1 \in S_D \iff d = x_{m+1} \in S_D \). But \( d \in S_D \) and \( b \in S_B \), which is a contradiction.

The Algorithm PolyRefine computes a correct refinement even when \( S_I \neq \emptyset \), since it that case, the refinement separates \( S_D \) and \( S_B \cup S_I \). In this case, however, the refinement computed by the Algorithm PolyRefine need not necessarily be minimal.

This completes the description of the CEGAR algorithm applied to symbolic model checking. We note that the technique is sound, since no spurious counterexamples are ever reported. Also, the completeness of the technique follows from the fact that the refinement process must eventually terminate, since each equivalence class must have at least one element.

### 3.5 Summary of Experimental Results for CEGAR

Clarke, et al. compare the performance of an implementation of the CEGAR approach implemented in NuSMV \[11\] with the performance of an implementation of NuSMV which performs cone of influence (COI) reduction \[14\]. Two sets of experiments were performed: (1) The first set of experiments were conducted on four benchmark designs and three industrial designs. (2) The other was performed on an industrial design of a multimedia processor from Fujitsu.

#### 3.5.1 Summary of Results on the Benchmark Set

The total running time of the CEGAR algorithm to verify all the properties was much lower than the running time of NuSMV with COI techniques for every benchmark. In terms of space, although the CEGAR algorithm required 50% more memory than the COI algorithm to compute the initial abstraction, it often required an order of magnitude less memory during the model checking phase.

The results further indicate that the CEGAR algorithm is outperformed by the COI reduction technique on only one benchmark-property pair, in terms of both space and time requirements.

#### 3.5.2 Summary of Results on the Fujitsu Design

The authors also verified a multimedia processor developed by Fujitsu as part of the evaluation of the CEGAR algorithm. The design is quite large and complicated and comprises of about 9,500 lines of NuSMV code. The details are mentioned in the paper by Clarke, et al. \[12\].

While both COI reduction techniques as well as an improved version of NuSMV \[29\] could not verify the Fujitsu design, the CEGAR algorithm could verify the design in three refinement steps. The verification process with CEGAR also led to the discovery of a previously unknown bug in the design.

### 4 Interpolation and SAT-based Model Checking

We now turn our attention to a SAT-based technique for symbolic model checking proposed by McMillan \[23\]. While the CEGAR approach described in Section 3 attempted to mitigate the problems caused by the unpredictability of OBDDs by model checking a simpler, abstracted version of the state space, the interpolation based approach described in this section attempts to tackle the problem by doing away with OBDDs altogether and maintaining over-approximations of reachable states by means of interpolants. The key innovation in the interpolant based approach to symbolic model checking is to leverage the proofs obtained from a SAT solver to Bounded Model Checking (BMC) \[13\] queries in order to produce a strengthening of the safety property to be checked. The objective of this strengthening process is that the strengthened property becomes inductive. Now if this strengthened property is such that it also excludes erroneous states, then from the inductive nature of the strengthening, it comprises a proof that the Kripke structure under question satisfies said property.
4.1 Preliminaries and Definitions

Recall the definitions of a clause and a literal from Section 2. A clause is called non-tautological if it does not contain a literal and its negation. A set of clauses $C$ is satisfiable when there exists a truth assignment to the Boolean variables which make all the clauses in the set TRUE. Such an assignment is called a satisfying assignment as mentioned in Section 2.

**Definition 4.1 (Resolvent).** Given two clauses $c_1 = \forall v \land A$ and $c_2 = \neg \forall v \land B$, the resolvent of $c_1$ and $c_2$ is the clause $A \lor B$ provided that $A \lor B$ is non-tautological.

Any two clauses can have at most one resolvent. The resolvent (if it exists) is a clause implied by $c_1$ and $c_2$ and is exactly $\exists \forall v. c_1 \land c_2$. $v$ is called the pivot variable in this case.

**Definition 4.2 (Proof of Unsatisfiability).** A proof of unsatisfiability $\Pi$ for a set of clauses $C$ is a directed acyclic graph $(V_\Pi, E_\Pi)$ where $V_\Pi$ is a set of clauses such that:

- for every vertex $c \in V_\Pi$, either
  - $c \in C$ and $c$ is a root, or
  - $c$ has exactly two predecessors $c_1$ and $c_2$ such that $c$ is the resolvent of $c_1$ and $c_2$, and
- The empty clause is the unique leaf.

**Theorem 4.1.** If there exists a proof of unsatisfiability for a clause set $C$ then $C$ is unsatisfiable.

*Proof.* The proof follows from the definition of a resolvent. We note that a resolvent of $c_1$ and $c_2$ is implied by $c_1 \land c_2$. Consider the leaf of the proof of unsatisfiability $\Pi$, which is FALSE by definition and is a resolvent of a pair of clauses $c_1$ and $c_2$, which themselves could be either be resolvents of other clauses, or clauses in the set $C$. Proceeding in this manner, we have that FALSE is implied by some set of clauses $C_{\text{sub}} \subseteq C$, which completes the proof.

**Definition 4.3 (Interpolant).** Given two sets of clauses $A$ and $B$ and a proof of unsatisfiability of the clause set $A \cup B$, an interpolant for the pair of clause sets $(A, B)$ is the a formula $P$ with the following properties:

- $A$ implies $P$.
- $P \land B$ is unsatisfiable.
- $P$ refers to only the common variables of $A$ and $B$, i.e., every variable which appears in $P$ also appears in both $A$ and $B$.

Suppose we are given a pair of clause sets $(A, B)$ and a proof of unsatisfiability $\Pi$ of $A \cup B$. With respect to the clause sets $(A, B)$, we call a variable global if it appears in both $A$ and $B$ and local to $A$ if it only appears in $A$. The notion of global and local carries over naturally to literals. Given a clause $c$, we let $g(c)$ denote the disjunction of the global literals in $c$ and $l(c)$ denote the disjunction of literals local to $A$. With these definitions in mind, we now describe how to extract an interpolant from a proof of unsatisfiability of a pair of clause sets $(A, B)$.

**Definition 4.4 (Π Interpolant).** Let $(A, B)$ be a pair of clause sets and let $\Pi$ be a proof of the unsatisfiability of $A \cup B$. The unique leaf vertex of $\Pi$ is FALSE by definition. For all vertices $c \in V_\Pi$ let $p(c)$ be a Boolean formula with the following properties:

- If $c$ is a root, then
  - If $c \in A$ then $p(c) = g(c)$.
  - $p(c) = \text{TRUE}$ otherwise.
- Otherwise, let $c_1, c_2$ be the predecessors of $c$ in $\Pi$ and let $v$ be their pivot variable:
Theorem 4.2. For all pairs of clause sets \((A, B)\) and \(\Pi\) a proof of unsatisfiability of \(A \cup B\), \(\text{Itp}(\Pi, A, B)\) is an interpolant for \((A, B)\).

Proof. Given a clause set \(A\) We let \(\text{Vars}(A)\) denote the set of Boolean variables which appear in the clauses in \(A\). Given a Boolean variable \(v \in \text{Vars}(A)\), we let \(\text{Clauses}(v, A)\) denote the set of clauses in the clause set \(A\) in which \(v\) or \(\neg v\) appears. The proof proceeds by an induction on the size of \(\text{Vars}(A) \setminus \text{Vars}(B)\), i.e., the number of local variables in \(A\). Suppose \(|\text{Vars}(A) \setminus \text{Vars}(B)| = 0\), then from Definition 4.4, \(p(\text{False})\) is simply a conjunction of the clauses in \(A\) and is a valid interpolant. Assume that Definition 4.4 correctly computes an interpolant for all \(k \leq n\) where \(n = |\text{Vars}(A) \setminus \text{Vars}(B)|\). Now, consider the case where \(|\text{Vars}(A) \setminus \text{Vars}(B)| = n + 1\). Pick a variable \(v \in \text{Vars}(A) \setminus \text{Vars}(B)\). Obviously, \(v\) is local to \(A\). There are two cases to consider:

- \(v\) appears in \(A\) with only one polarity. In this case, \(v\) is unconstrained and all clauses in \(\text{Clauses}(v, A)\) can be satisfied. These clauses will thus not be a part of the proof of unsatisfiability. Thus, in the proof of unsatisfiability \(|\text{Vars}(A) \setminus \text{Vars}(B)| \leq n\), and the proof follows.

- \(v\) appears in \(A\) with both polarities. Since we are free to choose the variable under question, we choose the variable \(v \in \text{Vars}(A) \setminus \text{Vars}(B)\) which gets resolved first in \(\Pi\). Consider the resolution step at which \(v\) gets resolved, yielding the clause \(c\) with predecessors \(c_1\) and \(c_2\). The proof of unsatisfiability of clause \(c\) along with the unresolved clauses now form a set of clauses with fewer than \(n + 1\) local variables in the clause set \(A\). Because \(v\) never forms part of the interpolant, this concludes the proof. \(\square\)

4.2 Interpolation and Model Checking

Having defined what an interpolant is and how to obtain an interpolant from a proof of unsatisfiability, we now focus on how interpolants can be used in symbolic model checking of safety properties. As mentioned earlier, liveness checking can be reduced to safety checking \([4]\) and will not be covered separately. Again, the system will be represented symbolically as a Kripke structure \(M = (V, R, I)\), where the transition relation is expressed as a propositional logic formula over the set of unprimed and primed variables as described in Section 2.2.2. Thus \(R\) is a Boolean formula over \(V = \{v_1, v_2, \ldots, v_n\}\) and \(V' = \{v'_1, v'_2, \ldots, v'_n\}\). \(I\) is a Boolean formula over \(V\) and the specification \(\varphi\) is also a Boolean formula over \(V\). The model checking problem is to check if \(\neg \varphi\) is reachable from the set of initial states \(I\).

For the rest of this manuscript we assume that \(V\) consists of only Boolean valued variables. As mentioned earlier, any finite domain \(D\) can be mapped to the Boolean domain using an appropriate mapping function. Thus, this assumption does not cause any loss of generality.

The key intuition in using interpolation in symbolic model checking is that an interpolant can be extracted from the proof of unsatisfiability of an unsuccessful BMC query. A BMC query consists of initial constraints \(I\), a set of transition constraints \(R\) and a set of final constraints \(F\). The final constraint encodes that no erroneous state is reachable in \(k\) steps from the initial set of states \(I\). These are obtained by unrolling the transition relation \(R\) some \(k\) number of times and renaming variables appropriately at each stage of unrolling. This is depicted in Figure 3.
Figure 4: Computation of Interpolant as an Over-approximation of Reachable States.

Suppose that a BMC query results in the SAT-solver providing a proof that no error states are reachable within $k$ steps. From the proof of unsatisfiability, we derive an interpolant $P$ for $(A, B)$, where the clause sets $A$ and $B$ are as shown in Figure 4. Because $P$ is implied by the initial condition and a one step transition constraint, it follows that $P$ is true in every state reachable from the initial states in one step. i.e., $P$ is an over-approximation of the set of states reachable in one step. Further, since $P$ and $B$ are unsatisfiable, it follows that no state in $P$ can reach an error state in $k - 1$ steps. This over-approximate image operation can be iterated to obtain an over-approximation of states reachable in $2, 3, \ldots$ steps. It might lead us to falsely conclude that an erroneous state is reachable, but by increasing $k$ in such situations, the algorithm will eventually find a true counterexample or be able to prove that no erroneous state is reachable by inductively strengthening $\varphi$. This intuition is formalized in Section 4.2.1.

4.2.1 The Basic Interpolation Based Algorithm

Consider a Kripke structure $M = (V, R, I)$ where $R$ and $I$ are represented symbolically as mentioned earlier, $V$ is a set of variables, $V = \{v_1, v_2, \ldots, v_n\}$ and $\varphi$ is an invariant. Consider a bounded path of length $k$, $s_0, s_1, \ldots, s_k$, where $s_i \in S, 0 \leq i \leq k$ in $M$. A bounded path is defined similarly to a path as in Section 2, except that it need not be of infinite length, i.e., interpreting states as assignments as mentioned earlier, $I(s_0)$ holds and $R(s_i, s_{i+1})$ holds for $0 \leq i < k$.

In BMC, we translate the existence of a run of $M$ which violates the invariant $\varphi$ into a Boolean satisfiability problem. We introduce a new set of variables $W_i = \{w_{i1}, w_{i2}, \ldots, w_{in}\}$, for $0 \leq i \leq k$. And given a propositional formula $P$, over the set of variables $V$, we let $P(W)$ denote the formula where the corresponding instances of $v_i$ are replaced with $w_i$ in $P$. Now a run of length $k$ which violates $\varphi$ exists if and only if the following formula is satisfiable.

$$BMC_k = I(W_0) \land \left( \bigwedge_{0 \leq i < k} R(W_i, W_{i+1}) \right) \land \left( \bigvee_{0 \leq i \leq k} \neg \varphi(W_i) \right)$$

For convenience of exposition, we parametrize this formula by the Boolean formula characterizing the set of initial states and split this formula into two parts: one representing prefixes of length one and another representing the suffixes of length $k$ as follows:

$$\text{Pref}(I) = I(W_{-1}) \land T(W_{-1}, W_0)$$

The set of suffixes of length $k$ are encoded in the following formula, again parametrized by the set of initial states:

$$\text{Suff}_k(I) = \left( \bigwedge_{0 \leq i < k} R(W_i, W_{i+1}) \right) \land \left( \bigvee_{0 \leq i \leq k} \neg \varphi(W_i) \right)$$

With the definitions in place, we now present the algorithm for symbolic model checking using interpolation in Algorithm 2. The algorithm first checks if there exists a run of length zero. If not, it sets the initial
Algorithm 2: FiniteRun

**Input:** A Kripke structure $M = (S, R, L, I)$, An invariant $\varphi$ and $k > 0$

1. If $I \land \neg \varphi$ is satisfiable then
   - return True
2. $R := I$
3. While TRUE do
   4. $I' := R$
   5. $A := \text{Cnf(Pref}(I'))$
   6. $B := \text{Cnf(Suff}_k(I'))$
   7. If $A \cup B$ is satisfiable then
      8. If $R = I$ then
         - return True
      9. Else
         - Abort
   10. Else
      11. Let $\Pi$ be a proof of unsatisfiability of $A \cup B$
      12. $P := \text{Itp}(\Pi, A, B)$
      13. $R' := P(V/W_0)$
      14. If $R' \Rightarrow R$ then
         15. return False
      16. $R := R \lor R'$

approximation states reachable in one step ($R$) to the set of initial states $I$. We then compute CNF representations of $A$ — the set of states reachable in one step from $R$ — and $B$ — which is the set of $k$ suffixes. If $A \cup B$ is satisfiable and $R = I$, then we have discovered a true counterexample. If $R \neq I$, then the process of over-approximating the set of states reachable in some number of steps from the initial state might have led to an erroneous state to be reachable, which would not have been possible otherwise. Because it is difficult to determine if the erroneous state is indeed reachable, we simply abort and the result is inconclusive.

On the other hand, if $A \cup B$ is unsatisfiable, we extract a $\Pi$-interpolant $P$ from the proof of unsatisfiability of $A \cup B$. Since $P$ is a formula implied by $A$, which is the set of states reachable in one step from $I$, and $P$ and $B$ are unsatisfiable, we conclude that no state in $P$ can reach a state which violates the invariant in $k$ steps or less. We then obtain a new approximation of the set of states reachable in one step from the initial state as $R \lor P(V/W_0)$ by substituting variables from $V$ for variables from $W_0$. If a fixpoint is reached, then $R$ is an inductive invariant and we have proved that $M$ can never reach a state which violates $\varphi$. If a fixed point is not reached, we iterate with the new over-approximation $R$.

**Theorem 4.3.** For $k > 0$, if the Algorithm FiniteRun does not abort, then it returns True if and only if there exists a reachable state in $M$ which violates $\varphi$.

**Proof.** First, whenever FiniteRun returns True, it is easy to see that either there exists a run of length zero, or a run of length $k + 1$. Now, if FiniteRun returns False, then:

1. $I \Rightarrow R$. This is easy to see since $R$ starts off being initialized by $I$ and is progressively weakened by adding more disjuncts.

---

2The procedure Cnf is used to convert an arbitrary propositional logic formulas into an equisatisfiable CNF formula. This can be achieved without an exponential blow-up in formula size. However, the algorithm is agnostic to the specific technique used to convert arbitrary formulas into CNF and hence we do not delve in the details of such algorithms here.
2. $\mathcal{R}(V) \land R(V, V') \Rightarrow \mathcal{R}(V')$. To prove this, consider the last iteration of \textsc{FiniteRun}, when it returns \textit{false} on detecting a fixpoint. In the last iteration, we have that $\mathcal{R}'(V')$ is implied by $\mathcal{R}(V) \land R(V, V')$. This is because $\mathcal{R}'$ is essentially the interpolant of $A \cup B$, where $A$ is $\mathcal{R} \land R(V, V')$, with some renaming of variables. Thus when $\mathcal{R}'$ itself implies $\mathcal{R}$, we get $\mathcal{R}'(V) \land R(V, V') \Rightarrow \mathcal{R}'(V')$, establishing $\mathcal{R}'$ as an inductive invariant.

3. $\mathcal{R} \land \neg \varphi$ is unsatisfiable. This follows by observing that initially $\mathcal{R} = I$ and $I \land \neg \varphi$ is unsatisfiable. In each iteration $\mathcal{R}$ is weakened by adding $\mathcal{R}'$ as a disjunct. We know that $\mathcal{R}' \land \neg \varphi$ is unsatisfiable from the definition of $\text{Suff}_k$. Thus $\mathcal{R} \land \neg \varphi$ is always unsatisfiable throughout the execution of \textsc{FiniteRun}.

\textbf{Theorem 4.4.} For every finite Kripke structure $M$, there exists a $k$ such that \textsc{FiniteRun} applied on $M$ terminates.

\textit{Proof.} Suppose $M$ has a counterexample to the invariant $\varphi$ of length $k$. Then it is obvious that \textsc{FiniteRun} terminates and reports the counterexample. On the other hand, if $M$ does not admit a violation of $\varphi$, then we notice that $\mathcal{R}$ progressively weakens in each iteration. This process cannot go on forever and must reach a fixpoint. If \textsc{FiniteRun} aborts for a given value of $k$, then it must eventually terminate for a larger value of $k$. Further, $k$ never needs to be greater than the \textit{diameter} of the Kripke structure $M$, which is defined as the length of the longest shortest simple path between a state $s_I \in I$ and any other state $s \in S$.

\section*{4.2.2 Optimizations}

We focus only on the optimizations discussed by McMillan \cite{MCMILLAN} which are \textit{sound}. The work also discusses some optimizations which are inherently unsound.

\textbf{Simplifying Interpolants.} McMillan \cite{MCMILLAN} discovered that interpolants are usually highly redundant, containing subformulas that are syntactically distinct, but logically equivalent. The implementation attempts to simplify interpolants by building BDDs up-to a fixed size and checking for equivalence.

\textbf{Caching of CNF formulas.} In the Algorithm \textsc{FiniteRun}, the formula $\text{Suff}_k(I)$ is invariant across iterations. It would be more efficient to cache the CNF representation of $\text{Suff}_k(I)$ than recomputing it each time. However, this was not implemented because it needed significant modifications to the SAT solver.

\section*{4.3 Interpolation Based Model Checking in Practice}

McMillan \cite{MCMILLAN} compares the interpolation based approach with two approaches: (1) An earlier work by the same author of using proofs of unsatisfiability to obtain abstractions of the systems \cite{MCMILLAN5}, we call this approach \textit{proof based abstraction} and (2) An approach by Baumgartner, \textit{et al.} \cite{BAUMGARTNER}, which exploits the structure of circuits of models with special properties to bound the depth of the search space.

\subsection*{4.3.1 Interpolation on the PicoJava II Microprocessor Benchmarks}

A set of 20 properties from the verification of a microprocessor were used to compare the strengths and weaknesses of the interpolation based approach. Standard symbolic model checking algorithms such as those implemented in NuSMV could not verify any of these properties within the time limit of 30 minutes. The interpolation based method was successfully able to verify 19 of the 20 properties. Compared to the proof based abstraction techniques, the interpolation based method was faster in verifying all but a few properties.

\subsection*{4.3.2 Interpolation on the IBM Gigahertz Processor Benchmarks}

The author compares the interpolation based method against Baumgartner’s techniques \cite{BAUMGARTNER} on a set of benchmark problems derived from the IBM Gigahertz processor. The author notes that of the 28 properties which could not be verified by Baumgartner’s approach, all but one were verified successfully by the interpolant.
based approach. In a vast majority of the cases, the interpolant based approach verified properties quicker than Baumgartner’s approach.

Comparing interpolant based approaches with proof based abstraction techniques on the IBM Gigahertz processor benchmarks resulted in the interpolant based approach not emerging as a clear winner. The author attributes it to the property of proof based abstraction techniques to find counterexamples quickly to properties that are falsifiable and suggests that a hybrid approach might provide the best performance in such cases.

5 IC3

We now describe another approach for symbolic model checking of safety properties which relies on learning an inductive strengthening of the safety property as well. However, the approach is markedly different from the approach presented in Section 4. The inductive strengthening obtained in the interpolation based approach was a consequence of the over-approximation of states reachable in some number of steps from the initial state reaching a fixpoint. On the other hand, IC3 approach [5, 6] attempts to learn small lemmas that are stepwise inductive relative to known reachability information and propagates these lemmas until an inductive strengthening of the safety property is achieved. Another difference between IC3 and interpolation based approaches is that IC3 issues a large number of very simple SAT queries to the SAT solver, compared to the few large, monolithic queries that interpolation based approaches pose to the SAT solver.

The rest of this section is organized as follows: In Section 5.1 we define the relevant terminology and go over some earlier work essential for understanding IC3. We provide an intuitive and informal description of the IC3 algorithm in Section 5.2 and defer a more formal treatment of the IC3 algorithm to Section 5.3. Section 5.4 describes some optimizations that help in enhancing the scalability and efficiency of the IC3 algorithm. Finally, we present the results of running IC3 to verify real world designs in Section 5.5.

5.1 Background and Preliminaries

In the rest of this section, all formulas are assumed to be over the set $V$ or $V'$ unless otherwise specified. Given a formula $F$, whenever there is no ambiguity, we will use $F$ as a shorthand for $F(V)$ and $F'$ as a shorthand for $F(V')$. We will also use $R$ to represent $R(V, V')$ when there is no ambiguity.

**Definition 5.1** (Inductive Assertion). Given a Kripke structure $M = (V, R, I)$, defined over the Boolean variables $V = \{v_1, v_2, \ldots, v_n\}$, an inductive assertion $F$ is a formula which describes a set of states that (1) includes all initial states of the transition system, i.e., $I \Rightarrow F$ and (2) is closed under the transition relation, i.e., $F \wedge R \Rightarrow F'$. The first condition is called initiation and the second is called consecution.

**Definition 5.2** (Relatively Inductive Assertion). An assertion $F$ is inductive relative to another assertion $G$ if (1) $I \Rightarrow F$ holds and (2) $F \wedge G \wedge R \Rightarrow F'$.

Consider a state $s \in S$ in the Kripke structure $M = (V, R, I)$. Recall that a state can be interpreted as an assignment to each of the Boolean variables $v \in V$. A state is thus a conjunction of literals: If a variable $v \in V$ is assigned the value TRUE in $s$, then it is represented by $v$, otherwise it is represented by $\neg v$ in this conjunction. A state can thus be represented as a Boolean formula. The negation of a state, denoted $\neg s$ represents all states in $S$ except $s$. We observe that $\neg s$ is a clause, by pushing the negation in and replacing conjunction by disjunction.

Previous work [7] demonstrates how to construct a sequence of relatively inductive assertions until it converges to form an inductive strengthening for the required safety property $\varphi$. The approach in the earlier work [7] can be broadly described as follows: Enumerate states that can reach a violation of the property $\varphi$ and conjoin their negations to $\varphi$ until it becomes inductive. The algorithm improves upon the naïve enumeration of states which can reach a state which violates $\varphi$ by inductively generalizing a counterexample. Because the counterexample is a clause, the proposed generalization technique was to find
a Minimal Inductive Subclause for a counterexample. The drawback of this approach is that when it fails to find an inductive generalization for a clause, it falls back to a naïve enumeration of states.

Minimal Inductive Subclauses. We briefly describe how to find minimal inductive subclauses because it is a core procedure for the iC3 algorithm.

Consider a state $s \in S$ which can reach a state which violates $\varphi$ in one step. Now, $\neg s$ is a clause. The idea is to find a minimal subclause of $s$ which is inductive, possibly relative to previously computed reachability information $G$. Such a subclause will not only rule out $s$ but possibly many other states as well. The procedure to find a minimal inductive subclause $d \subseteq \neg s = c_0$ is as follows: If $G \land c_0 \land R \Rightarrow c_0'$ holds, then $c_0$ is itself inductive. If not, then a counterexample to the implication, which is a state, — let’s call it $b$ — exists. We form $c_1 = c_0 \land \neg b$, i.e., $c_1$ contains only the literals common to $c_0$ and $\neg b$. We now iterate over $c_1$ until it converges to an inductive subclause $c_i$. If $I \Rightarrow c_i$ holds then $c_i$ satisfies initiation and consecution and is thus inductive and we set $d = c_i$. Otherwise, $c_0 = \neg s$ has no inductive subclause.

Now, $d$ is an inductive subclause but not necessarily a minimal inductive subclause. To find a minimal inductive subclause, we drop some literal from $d$ to form $d_1$. If $d_1$ is inductive, we recurse on $d_1$. If $d_1$ is not inductive, then we drop some other literal from $d$ to form $d_1$ until we find a literal which can be dropped without losing inductivity or we ascertain that $d$ itself is a minimal inductive subclause.

5.2 An Intuitive, Informal Description of the iC3 Algorithm

Given a Kripke structure $M$ and a safety property or invariant $\varphi$, the iC3 algorithm alternates between lazily extending and refining a sequence of formulas $F_0, F_1, F_2, \ldots, F_k$. Each $F_i$ is an over-approximation of the set of states reachable in at most $i$ steps from the set of initial states. The refinement phases can consist of multiple iterations of refinement and in each iteration one new clause is added to some $F_i$. $0 \leq i \leq k$. The algorithm maintains the following invariants on the sequence $F_0, F_1, \ldots, F_k$:

1. $I \Rightarrow F_0$.
2. $F_i \Rightarrow \varphi$, for $0 \leq i \leq k$. The algorithm thus maintains each $F_i$ as $\varphi$ and a set of clauses which grows as each $F_i$ is refined. We denote the set of clauses associated with $F_i$ as $C(F_i)$, i.e., each $F_i$ is of the form $(\bigwedge_{c \in C(F_i)} c) \land \varphi$.
3. $F_i \Rightarrow F_{i+1}$, for $0 \leq i < k$. In fact, from the previous invariant, we have that $C(F_{i+1}) \subseteq C(F_i)$ for $0 \leq i < k$.
4. $F_i \land R \Rightarrow F'_{i+1}$, for $0 \leq i < k$.

Given the logical structure of the core data structure, the iC3 algorithm first checks if $I \land \neg \varphi$ and $I \land R \land \neg \varphi'$ are satisfiable to detect zero and one step counterexamples respectively. If no such counterexamples exist then $F_0$ is set to $I$ and $F_1$ is set to $\varphi$.

Now, suppose that we have extended the sequence up to some $F_k$ and that the sequence $F_0, F_1, \ldots, F_k$ satisfies the four invariants mentioned above. We check if $F_k \land R \Rightarrow \varphi'$. There are two cases to consider:

- $F_k \land R \Rightarrow \varphi'$ holds. In this case, we extend the sequence to $F_0, F_1, \ldots, F_k, F_{k+1}$, where $F_{k+1}$ is initially set to $\varphi$. We also propagate clauses as follows: for any clause $c \in C(F_i)$, for $0 \leq i \leq k$, we check if $F_i \land R \Rightarrow c'$ holds. If it does, then we set $C(F_{i+1}) = C(F_{i+1}) \cup \{c\}$. If during this propagation step, we discover that $C(F_i) = C(F_{i+1})$ for some $0 \leq i \leq k$, then $F_i$ forms an inductive strengthening of $\varphi$ and is a proof that a state violating $\neg \varphi$ cannot be reached. This scenario is shown in Figure 5(f).

- $F_k \land R \Rightarrow \varphi'$ does not hold. Then there must exist a state $s \in F_k$ which is one step away from violating $\varphi$. We then proceed to find the maximum $j$ such that $\neg s$ is inductive relative to $F_j$. If at all $\varphi$ is invariant, then we claim that $j \geq k - 2$. To see why, suppose that $j < k - 2$, which means that $\neg s$ is not inductive relative to $F_{k-2}$. This implies that $F_{k-2}$ contains a state $t$ which is a predecessor of the state $s$. From invariant (4) above, we thus know that $F_{k-1}$ must contain $s$, from where we can reach a state violating $\varphi$ in one step. This implies that there exists a state $r \in F_k$, such that $r \models \neg \varphi$, contradicting the invariant (2).
Figure 5: An example illustrating the working of the ic3 algorithm.

Having found the maximum \( j \) such that \( \neg s \) is inductive relative to \( F_j \) (if not, we have discovered a counterexample and terminate), we then compute the minimal inductive subclause of \( \neg s \) relative to \( F_j \). Call
Algorithm 3: ic3Main

Input: A Kripke structure $M = (S, R, L, I)$ and an invariant $\varphi$.
Result: True if and only if $M \models \varphi$.

1. if $I \land \neg \varphi$ is satisfiable or $I \land R \land \neg \varphi'$ is satisfiable then
   return False

2. $F_0 := I$
3. $C(F_0) := \emptyset$
4. $F_i := \varphi$, for all $i > 0$
5. $C(F_i) := \emptyset$, for all $i > 0$

6. for $k := 1$ to $\ldots$ do
   7. if STRENGTHEN$(k) = \text{False}$ then
      8. return False
   9. PROPAGATECLAUSES$(k)$
10. if $C(F_i) = C(F_{i+1})$ for some $1 \leq i \leq k$ then
11.   return True

end for

this clause $c$. We then let $C(F_i) = C(F_j) \cup \{c\}$, and also set $C(F_{j+1}) = C(F_{j+1}) \cup \{c\}$, since by the definition of an inductive subclause, we know that $c \land F_j \land R \Rightarrow c'$. Thus $c$ holds in $F_{j+1}$ as well.

If $j = k - 1$ or $k$, then $c$ was conjoined to $F_k$, eliminating the state $s$ from $F_k$. This scenario is depicted in Figures 5(d) and 5(e). We then check again if $F_k \land R \Rightarrow \varphi'$ holds and repeat the process described. On the other hand, if $j = k - 2$, then $F_k$ still contains $s$. In this case, we proceed as follows: Since $\neg s$ is inductive relative to $F_{k-2}$ and not inductive relative to $F_{k-1}$, we can conclude that there exists a state $t \in F_{k-1}$, such that $t$ is a predecessor of $s$. See Figure 5(a) for a depiction of this scenario. If $\varphi$ is to be invariant, then $t$ itself must not be reachable; otherwise, we can reach a state violating $\varphi$ in two steps from $t$. We then recurse on the state $t$ at level $F_{k-1}$, with the subgoal of eliminating $t$ from $F_k$. If $\varphi$ is invariant, we must eventually conjoin a clause to $C(F_{k-1})$ which eliminates $t$ from $F_{k-1}$, as shown in Figure 5(b) otherwise we will obtain a counterexample. It is possible that in the attempt to eliminate $t$ from $F_{k-1}$, we will have to recur on one or more predecessors $t'$ of $t$ itself and then on $t'$ and so on, until the root cause for the counterexample is eliminated. For every state $q$ considered along this chain, the algorithm also attempts to conjoin minimal inductive subclauses of $q$ to $F_k$ as well as shown in Figure 5(c). While this step is not strictly necessary, it avoids having to consider these states at $F_k$ at a later point in time if and when a counterexample involving these states is encountered at the level $k$.

When the state $s$ has been finally eliminated from $F_k$, by conjoining one or more clauses to $F_k$, we consider if $F_k \land R \Rightarrow \varphi'$ holds again and repeat the steps described so far.

5.3 The ic3 Algorithm

Algorithm 3, Algorithm 4, Algorithm 5, Algorithm 6, Algorithm 7 and Algorithm 8 formally specify the various steps in the ic3 algorithm in a roughly top-down manner. For brevity, the inputs to Algorithm 3 are assumed to be available to all other algorithms described here as well. We now describe each of these algorithms in detail and prove correctness.

5.3.1 The Algorithm ic3Main

The Algorithm ic3Main is the top level of the ic3 algorithm. It first checks for length zero and one counterexamples. If none exist, it then sets up $F_0 = I$ and $F_1 = P$. The invariants that the Algorithm

---

3The detection of a counterexample might require following a chain of predecessors of $t$ until we find that chain to terminate with a state $x \in I$. 

23
Algorithm 4: Strengthen

**Input**: A level $k$

**Result**: False if a counterexample trace has been found, True otherwise

1. try
2. while $F_k \land R \land \neg \varphi$ is satisfiable do
3. $s :=$ Predecessor to the $\neg \varphi$ state extracted from the witness
4. $n :=$ INDUCTIVELYGENERALIZE($s, k - 2, k$)
5. PUSHGENERALIZATION($\{(n + 1, s)\}, k$)
6. return True
7. catch Counterexample:
8. return False

Algorithm 5: PropagateClauses

**Input**: A level $k$

1. for $i := 1$ to $k$ do
2. foreach $c \in \mathcal{C}(F_i)$ do
3. if $F_i \land R \land \neg c'$ is not satisfiable then
4. $\mathcal{C}(F_{i+1}) := \mathcal{C}(F_{i+1}) \cup c$

Algorithm 6: PushGeneralization

**Input**: A set $\text{states}$, where each $p \in S$ is a tuple containing a level and a state. A level $k$

1. while True do
2. $(n, s) :=$ Choose from $\text{states}$, minimizing $n$
3. if $n > k$ then
4. return
5. if $F_n \land R \land s'$ is satisfiable then
6. $p :=$ The predecessor of $s'$ extracted from the witness
7. $m :=$ INDUCTIVELYGENERALIZE($p, n - 2, k$)
8. $\text{states} := \text{states} \cup \{(m + 1, p)\}$
9. else
10. $m :=$ INDUCTIVELYGENERALIZE($s, n, k$)
11. $\text{states} := \text{states} \setminus \{(n, s)\} \cup \{(m + 1, s)\}$

ic3Main maintains are the following:

1. $I \Rightarrow F_i$, for all $i \geq 0$.
2. $F_i \Rightarrow \varphi$, for all $i \geq 0$.
3. $\mathcal{C}(F_{i+1}) \subseteq \mathcal{C}(F_i)$, for all $i > 0$.
4. $F_i \land R \Rightarrow F_{i+1}^\prime$, for all $0 \leq i < k$.
5. $|\mathcal{C}(F_k)| = 0$, for all $i > k$.

Each iteration of the loop first attempts to strengthen the last level, which is initialized to $\varphi$. During the strengthening process if a counterexample is discovered, it returns FALSE immediately. Otherwise, the call
### Algorithm 7: InductivelyGeneralize

**Input**: A state \( s \), a level \( \min \) and a level \( k \).

**Result**: A level between \( \min \) and \( k \) inclusive, at which \( s \) was inductively generalized.

1. if \( \min < 0 \) and \( F_0 \land \neg s \land R \land s' \) is satisfiable then
2. raise CounterExample
3. for \( i := \max(1, \min + 1) \) to \( k \) do
4. if \( F_i \land \neg s \land R \land s' \) is satisfiable then
5. GenerateClause\((s, i-1, k)\)
6. return \( i - 1 \)
7. GenerateClause\((s, k, k)\)
8. return \( k \)

### Algorithm 8: GenerateClause

**Input**: A state \( s \), A level \( i \) and a level \( k \).

1. \( c := \) subclause if \( \neg s \) that is relative to \( F_i \)
2. for \( j := 1 \) to \( i + 1 \) do
3. \( \mathcal{C}(F_j) := \mathcal{C}(F_j) \cup \{c\} \)

To PropagateClauses extends the sequence to an additional levels and propagates relatively inductive clauses from all lower levels to as a level as possible. In this process if two adjacent levels are found to have the same set of clauses, then we have found an inductive strengthening of \( \varphi \) and we return True.

To prove termination, observe that the sequence \( \mathcal{C}(F_0), \mathcal{C}(F_1), \ldots, \mathcal{C}(F_k) \) must be non-decreasing. In fact, they must be monotonically increasing, because otherwise we would terminate. Since the state space is finite, this cannot go on forever and must eventually terminate. The maximum number of iterations of the loop is therefore \( 2|V| + 1 \), where \( V \) is the set of Boolean variables in the Kripke structure.

To prove correctness, observe that from the invariants we have that \( F_i \land R \Rightarrow F_{i+1} \). Now if \( \mathcal{C}(F_i) = \mathcal{C}(F_{i+1}) \) for some \( i \), then we have that \( F_i \land R \Rightarrow F_i', \) i.e., \( F_i \) is inductive. Furthermore, \( F_i \Rightarrow \varphi \), therefore the system can never reach a state which violates \( \varphi \).

#### 5.3.2 The Algorithm Strengthen

The Algorithm Strengthen attempts to strengthen the \( F_k \) so that no state in \( F_k \) can reach a counterexample in one step. It iterates on each counterexample to induction, calling InductivelyGeneralize to inductively generalize the counterexample relative to some \( F_i \) and then attempting to push the generalization to \( F_k \) with a call to PushGeneralization.

To prove termination, we observe that the loop can only iterate \( 2|V| \) times at most, because the number of possible counterexamples is bounded by the size of the state space. Thus if all the called algorithms terminate then Strengthen also terminates.

We claim that when Strengthen terminates, \( F_k \land R \Rightarrow \varphi' \), unless a counterexample is discovered. This is easy to see from the loop termination condition in Algorithm Strengthen.

#### 5.3.3 The Algorithm PropagateClauses

PropagateClauses simply propagates clauses which are inductive relative to \( F_i \) to \( F_{i+1} \) and recurs on \( F_{i+1} \) until \( i = k \). This essentially propagates reachability information from lower levels to higher levels. Thus, a counterexample which has been eliminated at a level \( F_i \) never needs to be considered again at a level \( F_j \), where \( j > i \), as long as the counterexample can be generalized at level \( F_j \) as well.
5.3.4 The Algorithm PushGeneralization

The PushGeneralization algorithm attempts to push a generalization obtained at some level \(i\) to eliminate a state \(s\) from \(F_i\) to the level \(k\), \(k > i\). It achieves this by checking if the generalization obtained at level \(i\) is inductive relative to \(F_k\), if so, then it adds the inductive generalization at level \(k\) and terminates. Otherwise, it recurses on the predecessors of \(s\) in \(F_k\). To guarantee that this recursion terminates even in the presence of cyclic state spaces, a predecessor of a state is always generalized relative to the lowest level it is encountered. This is achieved by maintaining the level of the state along with the state itself in the set of states maintained by the algorithm PushGeneralization.

To prove termination of PushGeneralization, we observe that the size of the set \(states\) maintained by the algorithm is non-decreasing. Further the level associated with a state monotonically increases. Thus, the number of iterations of the loop is bounded by the number of levels \(k\) and the size of the state space \(2^{|\mathcal{V}|}\), proving termination.

5.3.5 The Algorithm InductivelyGeneralize

The Algorithm InductivelyGeneralize simply attempts to inductively generalize \(\neg s\), where \(s\) is a state given to it as input at the highest possible level \(i\) and adds the generalization to all the levels from 1 to \(i + 1\) with a call to GenerateClause. If at all inductive generalization of \(\neg s\) is called for relative \(F_0\) and it is the case that \(\neg s\) is not inductive with respect to \(F_0\), then we have discovered a counterexample. InductivelyGeneralize raises an exception to report this situation.

**Theorem 5.1.** Given a finite Kripke structure \(M = (S, R, L, I)\) and an invariant \(\varphi\), the Algorithm ic3Main always terminates and returns True if and only if \(M \models \varphi\).

**Proof.** The proof follows immediately from the termination and correctness proofs of the constituent algorithms and the proof of correctness and termination of ic3Main itself. \(\square\)

5.4 Optimizations in IC3

We briefly describe some key optimizations which help the IC3 algorithm to be more efficient in practice.

5.4.1 Use of Unsatisfiable Cores

Consider the unsatisfiable query \(F \land c \land R \land \neg c'\). The unsatisfiable core of this query can reveal a clause \(d \subset c\) such that \(F \land c \land R \land \neg d'\) is unsatisfiable. \(d\) can now be used as an inductive subclause if it satisfies initiation. If it does not (due to some initial values being defined as 0), then the IC3 algorithm restores a negative literal from \(c\) corresponding to the variable if available. This optimization can speed up the computation of minimum inductive subclauses.

5.4.2 Cone of Influence Techniques

The implementation of the IC3 algorithm applies reduction based on cone of influence techniques as follows: If a state \(s\) is known to be \(j\) steps away from violating the invariant \(\varphi\), then only the literals corresponding to variables in the \(j\) step cone of influence are considered in the initial clause \(c \subseteq \neg s\). The author claims that the inductive generalization generated from such a clause is more relevant with respect to \(\varphi\), even though \(c\) itself might be inductive only relative to a stronger stepwise assumption than \(\neg s\).

5.4.3 Subsumption

If two clauses \(c_1\) and \(c_2\) at levels \(i\) and \(j\), with \(i \leq j\) are such that \(c_2\) subsumes \(c_1\), i.e., \(c_2 \subseteq c_1\), then \(c_1\) is eliminated. This can have some effect in reducing the memory usage in the clause sets stored or cached in the SAT solver.
5.5 Summary of Experimental Results

We summarize the results of evaluating ic3 on the two benchmarks suites: (1) The Hardware Model Checking Competition (HWMCC) 2008, as reported in the technical report version [5] of the paper by Bradley [6]. (2) The results demonstrated in the published paper [6] on the benchmarks from HWMCC 2010. We summarize both these results because the results presented in the final version of the paper [6] does not include a comparison with other model checkers, but does include results on a parallel version of the algorithm, which the technical report [5] lacks.

5.5.1 Results on the HWMCC 2008 Benchmarks

Although ic3 does manage to successfully solve some of the benchmark problems from HWMCC 2008 which no other solver could solve, and solve some benchmarks quicker and with less memory than other solvers, it does not emerge as a clear winner. This is primarily because ic3 sometimes takes orders of magnitude more time and memory to solve some problems than other solvers.

However, one thing which stands out in the results of ic3 on the HWMCC 2008 benchmarks is the sheer number of SAT queries discharged by the ic3 algorithm. One of the stated design goals of ic3 was to trade-off a few large, complicated queries to the SAT solver for a large number of very simple queries. And the results clearly show that this goal has been achieved, with the ic3 algorithm often executing thousands of SAT queries per second.

There are several techniques by which the performance of ic3 can be further improved: (1) Better SAT solver support for incremental queries. The implementation of ic3 uses ZChaff [27], a dated SAT solver technology. This solver was chosen because of its support for incremental SAT solving. Use of state-of-the-art solvers like MiniSAT could make a huge difference in performance. (2) The algorithm itself is amenable to easy parallelization, unlike other commonly used symbolic model checking algorithms. The performance of a parallel solver is indeed studied in the final version of the technical report and the results are summarized in the next subsection.

5.5.2 Results on the HWMCC 2010 Benchmarks

The paper by Bradley [6] does not provide a comparison with other model checking techniques, instead referring the reader to publicly available results. It does however present runtime and memory statistics of ic3 on a small sample of the HWMCC 2010 benchmarks. Again, the most striking feature in the results is the large number of SAT queries executed each second.

Parallel Implementation. The final version of the paper by Bradley [6] also includes a description of the performance of a parallel implementation of the ic3 algorithm. The parallelism is exploited at the level of counterexamples to induction and generalizing from such counterexamples. A number of worker processes generalize different clauses at various levels. Each time a process generates an inductive generalization, it sends the clause and the level it was generalized at to a server. The server responds with a set of clauses and the levels each one of them was generated at, since the last communication between the process and the server. Duplication of work is avoided by employing randomization in the SAT solver so that it returns reasonably different counterexamples each time.

The parallel version scales linearly on most of the benchmarks it was evaluated with. The author attributes some of the performance gain to a reduction in variance: Discovering key lemmas (inductive clauses) early in the model checking process can avoid a lot of computation. With multiple processes and randomization in the SAT solver, it becomes more likely that such key lemmas will be discovered early by some process and shared with the rest, greatly speeding up the other processes as well. In some cases, the performance saturates beyond a certain level of parallelism, but does not degrade nonetheless.

Finally, in terms of problems that the sequential version of the ic3 algorithm could not solve within the time limit, the author claims that the parallel implementation enabled the solution of twelve additional benchmarks from the HWMCC 2010 contest, given the 900 second time limit.
6 Discussion

Having presented three techniques to mitigate the effects the unpredictable nature of OBDDs in symbolic model checking, we now briefly discuss the relative merits and demerits of each and compare the approaches along various dimensions: (1) The manner in which they compute inductive strengthenings. (2) The underlying decision procedures they use and the manner in which they invoke them. (3) The ease with which these algorithms can be composed with each other and other algorithms in an attempt to get the best of all worlds.

6.1 Inductive Strengthening

In a broad sense, every symbolic model checking algorithm can be considered as an algorithm that attempts to learn an inductive invariant. The set of reachable states itself is closed under the transition relation and thus forms an inductive strengthening of the property $\varphi$ to be checked, provided no state violating $\varphi$ is in the reachable set. This is precisely the approach taken by the original symbolic model checking algorithm, which used OBDDs to represent the set of reachable states. The approaches presented here improve upon this theme in different ways.

Inductive Strengthening in the CEGAR Approach. The CEGAR algorithm also computes an inductive strengthening using the standard symbolic model checking algorithm using OBDDs. However, the innovation lies in the fact that the inductive strengthening is computed not with respect to the original transition system, but an abstracted transition system, whose state space is smaller than the original system. This approach works well as long as the transition system admits an abstraction which, while being simpler, also has an inductive strengthening with respect to the property $\varphi$.

It is not clear how compact the inductive strengthening discovered implicitly by the CEGAR algorithm is with respect to $f_{\text{mi}}$. Also, because the CEGAR algorithm as presented uses OBDDs for the computation of the inductive invariant, it is plausible that the OBDD based representations could blow-up on some problems. However, we note that this is not a fundamental limitation of the approach; As we discuss later in this section, the CEGAR algorithm is not tightly coupled with the actual symbolic model checking algorithm used, and the OBDD based algorithms can easily be replaced with other, more efficient techniques.

Inductive Strengthening in the Interpolant based Approach. The interpolation based algorithm computes an inductive strengthening as well, but more as a consequence, than by intention. The intention of extracting an interpolant from an unsatisfiable BMC query is simply to over-approximate the set of states reachable within a bounded number of steps from the set of initial states. In a broad sense, the interpolation based approach computes the over-approximations of reachable states lazily, as compared to the CEGAR approach, which systematically constructs the over-approximate transition relation. The laziness of over-approximation in the interpolation based approach makes it difficult to use the information contained in a potential counterexample. In fact, it is difficult to even decide whether the erroneous state is indeed reachable when a BMC query with the over-approximation of the set of initial states indicates that an erroneous state is reachable. As such, The interpolation based algorithm does not even attempt to leverage any information obtained from satisfiable BMC queries in such scenarios and simply restarts with a larger value of $k$.

The algorithm itself is rather elegant and simple, but it could deteriorate into BMC for some problems. The author claims a distinction between the length of the longest “shortest path” in the state space and the diameter of the state space to make a claim that the interpolation based approach could require substantially fewer unrollings than BMC. However, most literature defines the diameter of a state space as the length of the longest “shortest path” itself. In the worst case, the interpolation based algorithm could require a degree

\footnote{We focus only on safety properties in this discussion. As mentioned before, a technique by which the problem of checking liveness properties can be reduced to the problem of checking safety properties has been proposed in the literature.}
of unrolling which is equal to the diameter of the state space, in which case, it is essentially performing BMC. As with the CEGAR algorithm, it is not clear how the inductive strengthening computed by the interpolation based algorithm compares to $f_m$.

**Inductive Strengthening in the ic3 Approach.** The ic3 approach attempts to build an inductive strengthening by using counterexamples to induction and generalizing them. The algorithm is an improvement over the earlier work by the author [7], which did not maintain over-approximations of states reachable within 1, 2, …, $k$ steps and attempted to construct an inductive strengthening globally. This approach could not handle scenarios like those described in Figure 5(a) efficiently, since the clause $\neg t$ will not be discovered as the root cause of the violation.

With the optimizations for subsumption in place, it is likely that the ic3 algorithm will find the inductive strengthening with the smallest CNF representation. To see why, observe that every clause added to each level $F_i$ by the ic3 algorithm is necessary to eliminate a state that can eventually reach a counterexample. The only question then is with regard to the quality of the inductive generalizations produced. We suspect that if it can be guaranteed that the best inductive generalizations are produced, then the ic3 algorithm could, in theory, result in computing the inductive strengthening which has the most compact CNF representation. Note however, that this does not say anything about the compactness of the inductive strengthening discovered by the ic3 algorithm in general. It is possible that an exponentially smaller DNF representation exists, which ic3 will be unable to discover because it only attempts to generalize clauses.

### 6.2 Use of Resolution Based Decision Procedures

We now compare the approaches along the dimension of the decision procedures they require and the way they use those decision procedures.

**OBDDs in CEGAR.** The CEGAR algorithm is unique among the algorithms considered in this manuscript in that it does not strictly require the use of any resolution based decision procedure such as a SAT solver. All the fixpoint computations are performed using OBDDs. Once a variable ordering is fixed, OBDDs are a canonical representation of Boolean formulas. So checks like equivalence and subsumption can be performed without the use of resolution based decision procedures. While not having to depend on a SAT solver is an advantage of the implementation of CEGAR as described in the paper [12], the recent, rapid advances in SAT solver technology and the unpredictable nature of OBDDs could cause the CEGAR approach, as it has been described, to be unable to scale to current day designs. Since the paper was published nearly a decade ago, the CEGAR approach has not been evaluated on modern day designs.

**SAT Solvers and Interpolation.** The interpolation based technique for symbolic model checking relies on a resolution based SAT solver for two purposes: (1) To check satisfiability of BMC queries and (2) To provide proofs of unsatisfiability when the BMC query ends up being unsatisfiable. The size of the queries issued by the interpolation based algorithm is typically large: For a $k$-step BMC query, the size of the query is $O(k|\mathcal{R}| + |\mathcal{ITP}|)$, ignoring the dependence on the number of variables and assuming that the property $\varphi$ is small, where $|\mathcal{ITP}|$ is the size of the interpolant, which grows linearly with each iteration of over-approximating the set of reachable states. The number of queries in the worst case can be $O((\text{Dia}(M))^2)$, where $\text{Dia}(M)$ is the diameter of the transition system $M$. To see why this is the case, we observe that there can be at most $k$ queries for an unrolling of size $k$ and no more that $\text{Dia}(M)$ unrollings are necessary.

**SAT Solvers and ic3.** In contrast to the interpolation based approach to symbolic model checking, the ic3 algorithm issues many more queries to the SAT solver. However the size of each query is relatively small. Consider the following analysis: Let $w$ be the size of the final inductive strengthening discovered by the ic3 algorithm. If we assume this to be upper bound on the size of $\mathcal{C}(F_i)$ for every $i$, then the size of each query is $O(w + |\mathcal{R}|)$. While this could be an improvement with respect to interpolation, we note that $w$ itself could be $O(2^{|\mathcal{V}|})$. This is unlikely to happen in practice, because that would imply that no clause could be generalized, which is unlikely to happen given that most designs have a lot of structure in them. We can in fact reduce this further by considering COI based reduction techniques which keep only the relevant parts of the transition relation. Another notable feature is that the size of the queries posed by the ic3 algorithm to
the SAT-solver does not grow linearly with the length of the sequence $F_0, F_1, \ldots, F_k$, unlike the interpolation bases approach. On the other hand, there is no easy way to bound the number of SAT solver queries that the ic3 algorithm might make, apart from the trivial upper bound of $O(2^{|V|})$.

6.3 Composability with other Techniques

We now consider how the algorithms described in this manuscript can complement each other and possibly other algorithms for symbolic model checking as well.

Combining CEGAR with Other Approaches. The CEGAR approach to symbolic model checking is not so much an algorithm as it is a paradigm. The methodology is not closely coupled with either the underlying model checking technology, or the exact representation of Boolean formulas. In fact, any symbolic model checker which is capable of producing counterexamples can be used to model check the abstract Kripke structure. In this context, it would be an interesting experiment to use the CEGAR methodology with the interpolation based algorithm or the ic3 algorithm as the symbolic model checker. Such an experiment would give some feel for whether the approaches exploit complementary aspects of the structure present in transition systems. It is possible that the over-approximation performed by the CEGAR methodology in the abstraction step causes techniques like interpolation and ic3 take longer to converge due to the destruction of some of the structure in the transition system.

The CEGAR methodology is also agnostic to the exact representation used for Boolean formulas. While the implementation described in the paper [12] uses OBDDs to represent Boolean formulas, if it turns out that representing formulas explicitly using a SAT solver to check subsumption is more efficient, then it could be used as well.

The only serious limitation of the CEGAR approach is that it can only be applied to ACTL* specifications which admit counterexamples. It is not obvious how the approach can be made more general so that it can be applied to check general CTL* or $\mu$-calculus specifications, given that it relies heavily on the existence of counterexamples.

Combining Interpolation with Other Approaches. While it is obvious that the interpolation based approach combines well with BMC based approaches, it is not very clear that it can be easily combined with other complementary symbolic model checking techniques. As mentioned earlier, it can be used in conjunction with the CEGAR methodology, but it does suffer from one serious limitation: It can only be applied to safety properties. Although McMillan claims that earlier work has studied how liveness checking can be reduced to safety checking [4], such a reduction often involves doubling the number of variables in the system and can cause the state space of the system to suffer an extremely large blow-up.

Interpolation based techniques cannot be easily combined with the ic3 algorithm because the two algorithms rely on radically different approaches to discovering inductive invariants. Although the outset, the sequence of approximations $F_i$ computed by ic3 might appear analogous to a $k$-step unwinding of the transition relation, it is not so. ic3 initializes each of these approximations to the largest possible over-approximation $\varphi$ and then proceeds to refine them, whereas the interpolation based algorithm starts with the smallest possible over-approximation $\sigma$ $-$ the set of states reachable in $k$-steps and proceeds to weaken the over-approximation based on interpolants.

Interpolation based techniques have been extended to domains other than the Boolean domain in recent work by McMillan, et al. [21, 25], including some infinite domains like the integers. The approach thus holds promise in that it might be useful in the verification of infinite state systems, or in the eventual development of more scalable algorithms for finite state systems.

Combining ic3 with Other Approaches. As mentioned earlier, ic3 can easily be combined with the CEGAR methodology for symbolic model checking. Although the ic3 algorithm described in the paper by Bradley [3, 6] is limited to checking safety properties, we note that the ic3 algorithm has been adapted to check CTL specifications [8] and LTL specifications [19] as well. Although support for full CTL* specifications has not yet been achieved, the ic3 algorithm and its variants can be used as a symbolic model checker in the CEGAR loop for ACTL specifications.
Another interesting possibility with the IC3 methodology is to combine it with explicit state model checking. Using a technique like [28] to quickly and efficiently explore all states reachable up-to a depth-bound \( k \) and using a technique such as Daikon [17] to detect likely invariants from this exploration, it might be possible to generalize unreachable states more efficiently. This is especially true since Bradley mentions in his paper [4] that key lemmas discovered early on can help speed up subsequent analysis. A variant of this approach is also possible where \( k \)-step backward reachability information is represented symbolically using OBDDs. Negation of such reachability information could provide a simple inductive clause that IC3 can use to eliminate counterexamples to induction at level \( k \).

7 Conclusions

We have described three algorithms in recent literature which try to mitigate the effects of the unpredictable resource usage of OBDDs in symbolic model checking. Although abstraction based techniques like CEGAR have had success in the past, current research in the field seems to be directed towards finding inductive strengthenings more efficiently, thereby proving safety of the system under consideration. Other recent work [15] also explores the theme of using induction to prove safety properties.

Going forward, advances in Satisfiability Modulo Theories (SMT) solver technology could be leveraged to obtain more efficient model checking algorithms. The techniques discussed here, all rely on mapping the finite domain of the state space into the Boolean domain and applying purely SAT based reasoning to arrive at inductive invariants. Even if the state space of the system is finite, it might be more efficient to use richer theories — such as the theory of integers — to reason about the state space. Recent literature [1] has proposed using SMT solvers instead of SAT solvers for Bounded Model Checking, but we believe that techniques like IC3 can benefit from SMT solvers as well. In particular, reformulating the inductive generalization which IC3 performs on the Boolean domain so that it can be applied meaningfully in the context of integer domains or real-valued domains could be extremely useful. The utility of such a technique is not restricted to model checking finite state systems; it can be applied to check and infer inductive loop invariants in imperative programs.

In computing an inductive strengthening of the desired invariant, all of the algorithms presented here solve the following second-order formula: \( \exists f. \forall A. \psi(f, A) \), where \( f \) is the desired inductive strengthening, \( A \) is an assignment over the variables of the set \( V \) and \( \psi(f, A) \) is \( f(A) \land \varphi(A) \land R(A, V') \Rightarrow \varphi(V') \). This is precisely the form of the program synthesis problem proposed in recent literature [18]. An interesting avenue of future work would be to explore how the techniques described in this manuscript could be leveraged to develop more efficient algorithms for program synthesis.

We conclude with a reflection on the approaches presented in this manuscript. The heuristics described here have been demonstrated to work very well in practice. Most real-world systems are unlikely to trigger pathological behaviors in the algorithms presented here. Various theoretical hardness results exist for model checking systems with respect to specifications expressed in temporal logics. However, proving a problem hard does not diminish the significance of the problem: the increasing complexity of software and hardware designs and the need to be able to have guarantees about their correctness translate to a continuing desire to improve the performance of model checking algorithms on real-world systems. This makes any heuristics which provide an improvement in the real-world performance of model checking techniques particularly relevant.

References
