Software Model Checking via IC3

Alessandro Cimatti and Alberto Griggio*

Fondazione Bruno Kessler – IRST
{cimatti,griggio}@fbk.eu

Abstract. IC3 is a recently proposed verification technique for the analysis of sequential circuits. IC3 incrementally overapproximates the state space, refuting potential violations to the property at hand by constructing relative inductive blocking clauses. The algorithm relies on aggressive use of Boolean satisfiability (SAT) techniques, and has demonstrated impressive effectiveness.

In this paper, we present the first investigation of IC3 in the setting of software verification. We first generalize it from SAT to Satisfiability Modulo Theories (SMT), thus enabling the direct analysis of programs after an encoding in form of symbolic transition systems. Second, to leverage the Control-Flow Graph (CFG) of the program being analyzed, we adapt the "linear" search style of IC3 to a tree-like search. Third, we cast this approach in the framework of lazy abstraction with interpolants, and optimize it by using interpolants extracted from proofs, when useful.

The experimental results demonstrate the great potential of IC3, and the effectiveness of the proposed optimizations.

1 Introduction

Aaron Bradley [6] has recently proposed IC3, a novel technique for the verification of reachability properties in hardware designs. The technique has been immediately generating strong interest: it has been generalized to deal with liveness properties [5], and to incremental reasoning [7]. A rational reconstruction of IC3, referred to as Property Driven Reachability (PDR), is presented in [12], together with an efficient implementation: an experimental evaluation shows that IC3 is superior to any other single solver used in the hardware model checking competition.See also [23] for an overview.

The technique has several appealing aspects. First, different from bounded model checking, k-induction or interpolation, it does not require unrolling the transition relation for more than one step. Second, reasoning is highly localized to restricted sets of clauses, and driven by the property being analyzed. Finally, the method leverages the power of modern incremental SAT solvers, able to efficiently solve huge numbers of small problems.

In this paper, we investigate the applicability of IC3 to software model checking. We follow three subsequent steps. We first generalize IC3 from the purely Boolean case [6], based on SAT, to the case of Satisfiability Modulo Theory (SMT) [1]. The

* Supported by Provincia Autonoma di Trento and the European Community's FP7/2007-2013 under grant agreement Marie Curie FP7 - PCOFUND-GA-2008-226070 "progetto Trentino", project ADAPTATION.

characterizing feature of the generalization is the computation of (underapproximations of) the preimage of potential bug states. This allows us to deal with software modeled as a (fully) symbolic transition system, expressed by means of first order formulae.

The second step is motivated by the consideration that the fully symbolic representation does not exploit the control flow graph (CFG) of the program. Thus, we adapt IC3, that is "linear" in nature, to the case of a tree, which is the Abstract Reachability Tree (ART) resulting from the unwinding of the CFG. This technique, that we refer to as TREE-IC3, exploits the disjunctive partitioning of the software, implicit in the CFG.

The third step stems from the consideration that TREE-IC3 can be seen as a form of lazy abstraction with interpolants [18]: the clauses produced by IC3 are in fact interpolants at the various control points of the ART. From this, we obtain another optimization, by integrating interpolation within IC3. With proof-based interpolation, once the path being analyzed is shown to be unfeasible with one SMT call, it is possible to obtain interpolants for each control point, at a low cost. The key problem with interpolation is that the behaviour is quite "unstable", and interpolants can sometimes diverge. On the other hand, IC3 often requires a huge number of individual calls to converge, and may be computationally expensive, especially in the SMT case, although it rarely suffers from a memory blow-up. The idea is then to obtain clause sets for IC3 from proof-based interpolation, in the cases where this is not too costly.

We carried out a thorough set of experiments, evaluating the merits of the three approaches described above, and comparing with other techniques for software model checking. The results show that the explicit management of the CFG is often superior to a symbolic encoding, and that the hybrid computation of clauses can sometimes yield significant speed-ups. A comparison with other approaches shows that TREE-IC3 can compete with mature techniques such as predicate abstraction, and lazy abstraction with interpolants.

This work has two key elements of novelty. The seminal IC3 [6] and all the extensions we are aware of [12,7,5] address the problem for fully symbolic transition systems at the bit level. This paper is the first one to lift IC3 from SAT to SMT, and also the first one to adapt IC3 to exploit the availability of the CFG.

This paper is structured as follows. In Sec. 2 we present some background. In Sec. 3 we describe the SMT generalization of IC3. In Sec. 4 we present TREE-IC3, and in Sec. 5 TREE-IC3+ITP, the hybrid approach using interpolants extracted from proofs. In Sec. 6 we experimentally evaluate the approach. In Sec. 7 we discuss related work. Finally, in Sec. 8 we draw some conclusions and outline lines of future research.

2 Background and Notation

Our setting is standard first order logic. We use the standard notions of theory, satisfiability, validity, logical consequence. We denote formulas with φ, ψ, I, T, P, variables with x, y, and sets of variables with X, Y. Unless otherwise specified, we work on quantifier-free formulas, and we refer to 0-arity predicates as Boolean variables, and to 0-arity uninterpreted functions as (theory) variables. A literal is an atom or its negation. A *clause* is a disjunction of literals, whereas a *cube* is a conjunction of literals. If s is a cube $l_1 \wedge \ldots \wedge l_n$, with $\neg s$ we denote the clause $\neg l_1 \vee \ldots \vee \neg l_n$, and vice versa. A formula is in conjunctive normal form (CNF) if it is a conjunction of clauses, and in disjunctive normal form (DNF) if it is a disjunction of cubes. With a little abuse of notation, we might sometimes denote formulas in CNF $C_1 \wedge \ldots \wedge C_n$ as sets of clauses $\{C_1, \ldots, C_n\}$, and vice versa. If $X_1, \ldots, X_n$ are a sets of variables and φ is a formula, we might write $\varphi(X_1, \ldots, X_n)$ to indicate that all the variables occurring in φ are elements of $\bigcup_i X_i$. For each variable x, we assume that there exists a corresponding variable x' (the *primed version* of x). If X is a set of variables, X' is the set obtained by replacing each element x with its primed version. Given a formula φ, φ' is the formula obtained by adding a prime to each variable occurring in φ, and $\varphi^{\langle n \rangle}$ is the formula obtained by adding n primes to each of its variables. Given a theory $\mathcal{T}$, we write $\varphi \models_{\mathcal{T}} \psi$ (or simply $\varphi \models \psi$) to denote that the formula ψ is a logical consequence of φ in the theory $\mathcal{T}$. Given a first-order formula φ, we call the *Boolean skeleton* of φ the propositional formula obtained by replacing each theory atom in φ with a fresh Boolean variable.

We represent a program by a *control-flow graph* (CFG). A CFG $A = (L, G)$ consists of a set L of program locations, which model the program counter pc, and a set $G \subseteq L \times \mathit{Ops} \times L$ of control-flow edges, which model the operations that are executed when control flows from one program location to another. The set of variables that occur in operations from Ops is denoted by X. We use first-order formulas for modeling operations: each operation $o \in \mathit{Ops}$ has an associated first-order formula $T_o(X, X')$ modeling the effect of performing the operation o. A *program* $\Pi = (A, \mathsf{pc}_0, \mathsf{pc}_E)$ consists of a CFG $A = (L, G)$, an initial program location $\mathsf{pc}_0 \in L$ (the program entry), and a target program location $\mathsf{pc}_E \in L$ (the error location). A *path* π is a sequence $(\mathsf{pc}_0, op_0, \mathsf{pc}_1), (\mathsf{pc}_1, op_1, \mathsf{pc}_2), \ldots, (\mathsf{pc}_{n-1}, op_{n-1}, \mathsf{pc}_n)$, representing a syntactical walk through the CFG. The path π is *feasible* iff the formula $\bigwedge_i {T_{op_i}}^{\langle i \rangle}$ is satisfiable. When π is not feasible, we say it is *spurious*. A program is *safe* when all the paths leading to pc_E are not feasible.

Given a program Π, an *abstract reachability tree* (ART) for Π is a tree $\mathcal{A}$ over (V, E) such that: (i) V is a set of triples $(\mathsf{pc}, \varphi, h)$, where $\mathsf{pc} \in L$ is a location in the CFG of Π, φ is a formula over X, and $h \in \mathbb{N}$ is a unique identifier; (ii) the root of $\mathcal{A}$ is $(\mathsf{pc}_0, \top, 1)$; (iii) for every non-leaf node $v \overset{\text{def}}{=} (\mathsf{pc}_i, \varphi, h) \in V$, for every control-flow edge $(\mathsf{pc}_i, op, \mathsf{pc}_j) \in G$, v has a child node (pc_j, ψ, k) such that $\varphi \wedge T_{op} \models \psi'$ and $k > h$. In what follows, we might denote with $\mathsf{pc}_i \rightsquigarrow \mathsf{pc}_j$ any path in an ART from a node $(\mathsf{pc}_i, \varphi, h)$ to a descendant node (pc_j, ψ, k). Intuitively, an ART represents an unwinding of the CFG of a program performed in an abstract state space. If $v \overset{\text{def}}{=} (\mathsf{pc}_i, \varphi, h)$ is a node, φ is the *abstract state formula* of v. A node $v_1 \overset{\text{def}}{=} (\mathsf{pc}_i, \psi, k)$ in an ART $\mathcal{A}$ is *covered* if either: (i) there exists another node $v_2 \overset{\text{def}}{=} (\mathsf{pc}_i, \varphi, h)$ in $\mathcal{A}$ such that $h < k$, $\psi \models \varphi$, and v_2 is not itself covered; or (ii) v_1 has a proper ancestor for which (i) holds. $\mathcal{A}$ is *complete* if all its leaves are either covered or their abstract state formula is equivalent to $\perp$. $\mathcal{A}$ is *safe* if and only if it is complete and, for all nodes $(\mathsf{pc}_E, \varphi, h) \in V$, $\varphi \models \perp$. If a program Π has a safe ART, then Π is safe [16,18].

Given a set X of (state) variables, a *transition system* S over X can be described symbolically with two formulas: $I_S(X)$, representing the initial states of the system, and $T_S(X, X')$, representing its transition relation. Given a program Π, a corresponding transition system S_Π can be obtained by encoding symbolically the CFG (L, G)

of Π. This can be done by: (i) adding one special element x_{pc}, with domain L, to the set X of variables; (ii) setting $I_{S_\Pi} \stackrel{\text{def}}{=} (x_{\mathsf{pc}} = \mathsf{pc}_0)$; and (iii) setting $T_{S_\Pi} \stackrel{\text{def}}{=} \bigvee_{(\mathsf{pc}_i, op, \mathsf{pc}_j) \in G} (x_{\mathsf{pc}} = \mathsf{pc}_i) \wedge T_{op} \wedge (x'_{\mathsf{pc}} = \mathsf{pc}_j)$.

Given S_Π, the safety of the program Π can be established by proving that all the reachable states of S_Π are a subset of the states symbolically described by the formula $P \stackrel{\text{def}}{=} \neg(x_{\mathsf{pc}} = \mathsf{pc}_E)$. In this case, we say that S_Π satisfies the invariant property P.

3 IC3 with SMT

High-Level Description of IC3. Let X be a set of Boolean variables, and let S be a given Boolean transition system described symbolically by $I(X)$ and $T(X, X')$. Let $P(X)$ describe a set of good states. The objective is to prove that all the reachable states of S are good. (Conversely, $\neg P(X)$ represents a set of "bad" states, and the objective is to show that there exists no sequence of transitions from states in $I(X)$ to states in $\neg P(X)$.) The IC3 algorithm tries to prove that S satisfies P by finding a formula $F(X)$ such that: (i) $I(X) \models F(X)$; (ii) $F(X) \wedge T(X, X') \models F(X')$; and (iii) $F(X) \models P(X)$.

In order to construct F, which is an inductive invariant, IC3 maintains a sequence of formulas (called *trace*, following [12]) $F_0(X), \ldots, F_k(X)$ such that:

- $F_0 = I$;
- for all $i > 0$, F_i is a set of clauses;
- $F_{i+1} \subseteq F_i$ (thus, $F_i \models F_{i+1}$);
- $F_i(X) \wedge T(X, X') \models F_{i+1}(X')$;
- for all $i < k$, $F_i \models P$;

The algorithm proceeds incrementally, by alternating two phases[1]: a blocking phase, and a propagation phase. In the *blocking* phase, the trace is analyzed to prove that no intersection between F_k and $\neg P(X)$ is possible. If such intersection cannot be disproved on the current trace, the property is violated and a counterexample can be reconstructed. During the blocking phase, the trace is enriched with additional clauses, that can be seen as strengthening the approximation of the reachable state space. At the end of the blocking phase, if no violation is found, $F_k \models P$.

The *propagation* phase tries to extend the trace with a new formula F_{k+1}, moving forward the clauses from preceding F_i. If, during this process, two consecutive elements of the trace (called *frames*) become identical (i.e. $F_i = F_{i+1}$), then a fixpoint is reached, and IC3 can terminate with F_i being an inductive invariant proving the property.

Let us now consider the lower level details of IC3. For $i > 0$, F_i represents an over-approximation of the states of S reachable in i transition steps or less. The distinguishing feature of IC3 is that such sets of clauses are constructed incrementally, starting from cubes representing sets of states that can reach a bad state in zero or more

[1] We follow the formulation of IC3 given in [12], which is slightly different from the original one of Bradley given in [6]. Moreover, for brevity we have to omit several important details, for which we refer to the two papers cited above.

```
bool IC3-prove($I$, $T$, $P$):
1.  trace = [$I$]  # first elem of trace is init formula
2.  trace.push()  # add a new frame to the trace
3.  while True:
        # blocking phase
4.      while there exists a cube $c$ s.t. trace.last() $\wedge T \wedge c$ is satisfiable and $c \models \neg P$:
5.          recursively block the pair $(c, \text{trace.size()} - 1)$
6.          if a pair $(p, 0)$ is generated:
7.              return False  # counterexample found

        # propagation phase
8.      trace.push()
9.      for $i = 1$ to trace.size() $- 1$:
10.         for each clause $c \in$ trace[i]:
11.             if trace[i] $\wedge c \wedge T \wedge \neg c'$ is unsatisfiable:
12.                 add $c$ to trace[i+1]
13.         if trace[i] == trace[i+1]:
14.             return True  # property proved
```

Fig. 1. High-level description of IC3 (following [12])

transition steps. More specifically, in the blocking phase, IC3 maintains a set of pairs (s, i), where s is a cube representing a set of states that can lead to a bad state, and $i > 0$ is a position in the current trace. New clauses to be added to (some of the frames in) the current trace are derived by (recursively) proving that a set s of a pair (s, i) is unreachable starting from the formula F_{i-1}. This is done by checking the satisfiability of the formula:

$$F_{i-1} \wedge \neg s \wedge T \wedge s'. \tag{1}$$

If (1) is unsatisfiable, and s does not intersect the initial states I of the system, then $\neg s$ is *inductive relative to* F_{i-1}, and IC3 strengthens F_i by adding $\neg s$ to it[2], thus *blocking* the bad state s at i. If, instead, (1) is satisfiable, then the overapproximation F_{i-1} is not strong enough to show that s is unreachable. In this case, let p be a cube representing a subset of the states in $F_{i-1} \wedge \neg s$ such that all the states in p lead to a state in s' in one transition step. Then, IC3 continues by trying to show that p is not reachable in one step from F_{i-2} (that is, it tries to block the pair $(p, i-1)$). This procedure continues recursively, possibly generating other pairs to block at earlier points in the trace, until either IC3 generates a pair $(q, 0)$, meaning that the system does not satisfy the property, or the trace is eventually strengthened so that the original pair (s, i) can be blocked. Figure 1 reports the pseudo-code for the full IC3 algorithm, including more details on the propagation phase.

Extension to SMT. In its original formulation, IC3 works on finite-state systems, with Boolean state variables and propositional logic formulas, using a SAT solver as its reasoning engine. However, for modeling programs it is often more convenient to reason at a higher level of abstraction, using (decidable) fragments of first-order logic and SAT modulo theories (SMT).

[2] In fact, $\neg s$ is actually *generalized* before being added to F_i. Although this is quite important for the effectiveness of IC3, here for simplicity we shall not discuss this.

Most of the machinery of IC3 can be lifted from SAT to SMT in a straightforward way, by simply replacing the underlying SAT engine with an SMT solver. From the point of view of IC3, in fact, it is enough to reason at the level of the Boolean skeleton of formulas, simply letting the SMT solver cope with the interpretation of the theory atoms. There is, however, one crucial step in which IC3 must be made theory-aware, as reasoning at the Boolean-skeleton level does not work. This happens in the blocking phase, when trying to block a pair (s, i). If the formula (1) is satisfiable, then a new pair $(p, i-1)$ has to be generated such that p is a cube in the *preimage of* s *wrt.* T. In the purely-Boolean case, p can be obtained from the model μ of (1) generated by the SAT solver, by simply dropping the primed variables occurring in μ.[3] This cannot be done in general in the first-order case, where the relationship between the current state variables X and their primed version X' is encoded in the theory atoms, which in general cannot be partitioned into a primed and an unprimed set.

A first (and rather naïve) solution would be to consider the theory model for the state variables X generated by the SMT solver. However, for infinite-state systems this would lead IC3 to exclude only a single point at a time. This will most likely be impractical: being the state space infinite, there would be a high chance that the blocking phase will diverge.

For theories admitting quantifier elimination, a better alternative is to compute an exact preimage of s. This means to existentially quantify the variables X' in (1), eliminate the quantifiers, and then convert the result in DNF. This will generate a set of cubes $\{p_j\}_j$ which in turn generate a set of pairs $\{(p_j, i-1)\}_j$ to be blocked at $i-1$. The drawback of the second solution is that for many important theories, even when it is possible, quantifier elimination may be a very expensive operation.

We notice that the two solutions above are just the two extremes of a range of possibilities: in fact, any procedure that is able to compute an under-approximation of the exact preimage can be used. Depending on the theory, several trade-offs between precision and computational cost can be explored, ranging from single points in the state space to a precise enumeration of all the cubes in the preimage. In what follows, we shall assume that we have a procedure APPROX-PREIMAGE for computing such under-approximations, and present our algorithms in a general context. We shall discuss our current implementation, which uses the theory of Linear Rational Arithmetic, in §6.

Discussion. We conclude this Section by pointing out that the ideas underlying IC3 are nontrivial even in the Boolean case. At a very high level, the correctness is based on the invariants ensured by the blocking and propagation phases. Termination follows from the finiteness of the state space being analyzed, and from the fact that at each step at least one more new state is explored. A more in depth justification is out of the scope of this paper. The interested reader is referred to [6,23,12].

In the case of SMT, we notice that the invariants of the traces also hold in the SMT case, so that the argument for the finite case can be applied. This ensures partial correctness. On the other hand, the reachability problem being undecidable for infinite-state transition systems, it is impossible to guarantee termination. This might be due to the

[3] For efficiency, the result has to be generalized by dropping irrelevant variables, but this is not important for the discussion here.

failure in the blocking phase to eliminate all the counterexamples, for the given trace length, or to the failure to reach a fixpoint in the propagation phase.

4 Tree-Based IC3

We now present an adaptation of IC3 from symbolic transition systems to a CFG-represented program. The search proceeds in an "explicit-symbolic" approach, similarly to the lazy abstraction approach [16]. The CFG is unwound into an ART (Abstract Reachability Tree), following a DFS strategy. Each node of the tree is associated with a location, and a set of clauses.

The algorithm starts by finding an abstract path to the error location. Then, it applies a procedure that mimics the blocking phase of IC3 on the sets of clauses of the path.

There are three important differences. First, the clauses associated to a node are implicitly conditioned to the corresponding control location: the clause $\neg(x_{\mathsf{pc}} = \mathsf{pc}_i) \vee c$ in the fully symbolic setting simply becomes c in a node associated with control location pc_i. This also means that the logical characterization of a node being unreachable, expressed by the clause $\neg(x_{\mathsf{pc}} = \mathsf{pc}_E)$ in the fully symbolic setting, is now the empty clause. Second, in each formula T_i characterizing a transition, the start and end control locations are not explicitly represented, but rather implicitly assumed. Finally, the most important difference is in the inductiveness check performed when constructing the IC3 trace. When checking whether a cube c is blocked by a set of clauses F_{i-1}, we cannot use the relative inductiveness check of (1). This is because that would not be sound in our setting, since we are using different transition formulas T_i at different i steps (corresponding to the edge formulas in the abstract error path). Therefore, we replace (1) with the weaker check

$$F_{i-1} \wedge T_{i-1} \models \neg c' \tag{2}$$

which allows us to construct a correct ART (satisfying points (i)–(iii) of the definition on page 279.) We observe that, because of this difference, the requirement that $F_{i+1} \subseteq F_i$ is not enforced in TREE-IC3.

With this adaptation, the blocking phase tries to produce the clauses necessary to refute the abstract path. When the blocking phase is successful, it must generate an empty clause at some point. In case of failure to refute the path, the property is violated and a counterexample is produced[4]. If sufficient information can be devised to refute the abstract path to the error location, the algorithm backtracks to the deepest node that is not inconsistent (i.e. is not associated with the empty clause). The pseudo-code of this modified blocking phase, which we call TREE-IC3-BLOCK-PATH, is reported in Figure 2.

Then, a new node is selected and expanded, with a process that is similar in nature to the forward propagation phase of IC3. For each expanded node, the clauses of the ancestor are tested for forward propagation, in order to ensure the invariant that the clauses of an abstract node overapproximate the image of the predecessor clauses. More specifically, for each clause c, we check whether $F_i \wedge (x_{\mathsf{pc}} = \mathsf{pc}_i) \rightarrow c \wedge T_{op}$ entails $(x'_{\mathsf{pc}} = \mathsf{pc}_{i+1}) \rightarrow c'$.

[4] The counterexample has exactly the same length as the abstract path. This is a key difference with respect to the case of the fully symbolic IC3.

```
procedure TREE-IC3-BLOCK-PATH ($\pi \overset{\text{def}}{=} (\mathsf{pc}_0, \top, 1) \rightsquigarrow \ldots (\mathsf{pc}_i, \varphi_i, \cdot) \ldots \rightsquigarrow (\mathsf{pc}_E, \varphi_n, \cdot)$):
    # $T_1 \ldots T_{n-1}$ are the edge formulas of $\pi$
    # initialize the trace with the clauses attached to the nodes in $\pi$
1.  $F = [\top, \ldots, \varphi_i, \ldots, \varphi_{n-1}]$
2.  while not exists $j$ in $1 \ldots n-1$ s.t. $F[j] \wedge T_j \models \bot$:
3.      q = []
4.      for each bad in APPROX-PREIMAGE ($\varphi_{n-1} \wedge T_{n-1}$):
5.          q.push((bad, $n-1$))  # bad is a cube in the preimage of $T_{n-1}$
6.      while q is not empty:
7.          $c, j$ = q.top()
8.          if $j = 0$: compute and return a counterexample trace  # $\pi$ is a feasible error trace
9.          if $F[j-1] \wedge T_{j-1} \models \neg c'$:
10.             q.pop()  # c is blocked, discard the proof obligation
11.             $g$ = generalization of $\neg c$ s.t. $F[j-1] \wedge T_{j-1} \models g'$
12.             $F[j] = F[j] \wedge g$
13.         else:
14.             for each $p$ in APPROX-PREIMAGE ($F[j-1] \wedge T_{j-1} \wedge c'$):
15.                 q.push(($p, j-1$))
16.  return $F$  # $\pi$ is blocked
```

Fig. 2. Modified blocking phase of TREE-IC3 for refuting a spurious error path

A significant difference with respect to IC3 is in the way the fix point is handled. In IC3 the fix point is detected globally, by comparing two subsequent formulae in the trace. Here, as standard in lazy abstraction, we close each path of the ART being generated.

Whenever a new node v' is expanded, it is checked against previously generated nodes v having the same location. If the set of states of v' is contained in the states of some previously generated node v, then v' is covered, and it can be closed[5].

In order to maximize the probability of coverage, the IC3-like forward propagation phase is complemented by another form of forward propagation: whenever a loop is encountered (i.e. the node v' being expanded has the same location of one of its ancestors v), then each of the clauses of v is tested to see if it also holds in v'. Let $v, v_1, \ldots, v_k, v'$ be the path from v to v'. For each clause c in v, we check if the symbolic encoding of the path $v \rightsquigarrow v'$, strengthened with the clauses in each v_i, entails c in v'.

It is easy to see that this may result in a stronger set of clauses for v', because the analysis is carried out on the concrete path from v to v', that retains all the available information. Simple forward propagation would not be able to achieve the same result, because of the limited strength of the clauses on the intermediate nodes v_i. Intuitively, this means that the clauses in v_i may be compatible with (too weak to block) the paths that violate the clauses of v that also hold in v'. Thus, simply strengthening v' would break the invariant that, in each node, the abstract state formula overapproximates the image of the abstract state formula of its parent node (point (iii) in the definition of ART, §2). In order to restore the situation, the v_i nodes must be strengthened. Let $C_{v,v'}$

[5] In fact, it is also required that there will be no cycles in the covering-uncovering interplay. This requirement is a bit technical, and discussed in detail in [18]. In the definition of covered node, in §2, identifiers to nodes are intended to enforce this requirement.

```
ART UNWINDING:
if $v \overset{\text{def}}{=} (\mathsf{pc}_i, \varphi, h)$ is an uncovered leaf:
  for all edges $(\mathsf{pc}_i, op, \mathsf{pc}_j)$ in the CFG:
    add $v_j \overset{\text{def}}{=} (\mathsf{pc}_j, \top, k)$ with $k > h$ as a child
                                   of $v$ in the ART

PATH BLOCKING:
if $v_E \overset{\text{def}}{=} (\mathsf{pc}_E, \varphi, h)$ is a leaf with $\varphi \not\models \bot$:
  apply TREE-IC3-BLOCK-PATH (Fig. 2) to
    the ART path $\pi \overset{\text{def}}{=} (\mathsf{pc}_0, \top, 1) \rightsquigarrow v_E$
  if IC3 returns a counterexample: return UNSAFE
  otherwise:
    let $F_1, \ldots, F_E$ be the sets of clauses
    computed by IC3 for the formulas
                        $T_{op_1}, \ldots, T_{op_E}$ of $\pi$
    for each node $v_i \overset{\text{def}}{=} (\mathsf{pc}_i, \varphi_i, h_i) \in \pi$,
   for each clause $c_j$ in the corresponding $F_i$:

      if $\varphi_i \not\models c_j$, then:
        add $c_j$ to $\varphi_i$
        uncover all the nodes covered by $v_i$

NODE COVERING:
if $v_1 \overset{\text{def}}{=} (\mathsf{pc}_i, \psi, k)$ is uncovered, and there exists
$v_2 \overset{\text{def}}{=} (\mathsf{pc}_i, \varphi, h)$ with $k > h$ and $\psi \models \varphi$, then:
  mark $v_1$ as covered by $v_2$
  uncover all the nodes $v_j \overset{\text{def}}{=} (\mathsf{pc}_i, \psi_j, k_j)$ covered by $v_1$

STRENGTHENING:
let $v_1 \overset{\text{def}}{=} (\mathsf{pc}_i, \varphi, h_1)$ and $v_2 \overset{\text{def}}{=} (\mathsf{pc}_k, \psi, h_2)$ be two
uncovered nodes s.t. there is a path $\pi \overset{\text{def}}{=} v_1 \rightsquigarrow v_2$,
and let $\phi_\pi \overset{\text{def}}{=} \bigwedge_{j=0}^{n} T_{op_j}{}^{\langle j \rangle}$ be the formula for $\pi$
let $C_{v_1,v_2} = \emptyset$
for each $c_j \in \varphi$:
  if $\psi \not\models c_j$ and $\varphi^{\langle 0 \rangle} \wedge \phi_\pi \models c_i{}^{\langle n \rangle}$:
    add $c_j$ to $C_{v_1,v_2}$
if $C_{v_1,v_2} \neq \emptyset$:
  refute $\neg C_{v_1,v_2}$ using TREE-IC3-BLOCK-PATH along $\pi$
  for each node $v_j \overset{\text{def}}{=} (\mathsf{pc}_j, \varphi_j, h_j) \in \pi$:
    add all the clauses $c \in F_j$ computed by IC3 s.t. $\varphi_j \not\models c$
    if $\varphi_j$ changes, uncover all the nodes covered by $v_j$
  add $C_{v_1,v_2}$ to $\psi$, and uncover all the nodes covered by $v_2$
```

Fig. 3. High-level description of the basic building blocks of TREE-IC3

be the set of clauses of v that also hold in v'. Before adding $C_{v,v'}$ to v', we strengthen the v_i nodes with the information necessary to block the violation of $C_{v,v'}$ in v'. This is done by "tricking" the blocking phase, using the negation of $C_{v,v'}$ as conjecture: the clauses deduced in the process of refuting $\neg C_{v,v'}$ can be added to strengthen each v_i. After this, $C_{v,v'}$ is added to v'.

Notice that whenever a node v is strengthened, then each node v' that had been covered by v must be re-opened. In fact, after the strengthening, the set of states of v shrinks, thus the set of states of v', that was previously covered, might no longer be contained.

A high-level view of the basic steps of TREE-IC3 is reported in Figure 3. We shall describe the actual strategy that we have implemented for applying these steps in §6. (Notice that the forward propagation that is performed when a node is expanded is just a special case of the more general strengthening procedure, in which the path between the two nodes involved consists of a single edge, and as such does not require a call to IC3 for strengthening the intermediate nodes.)

Comparison with IC3. When the fully symbolic IC3 analyzes a program (in form of symbolic transition system), some literals represent the location in the control flow that is "active". This information, that is implicit in the position in the ART, becomes direct part of the clauses. There is the possibility for clauses to be present at frames where the corresponding location can not be reached, and that are thus irrelevant. Another advantage of the TREE-IC3 approach is that the program is disjunctively partitioned, transition by transition, and thus the SMT solver is manipulating simpler and smaller formulae. On the other hand, the symbolic representation gives the ability to implicitly "replicate" the same clause over many control locations – in particular, when no control location is relevant in the clause, it means that it holds for all the control locations. Moreover, using a symbolic representation of the program as a transition formula allows to exploit relative inductiveness, which is crucial for the performance of the original IC3 (on hardware designs). As already mentioned above, relative inductiveness cannot be directly applied in our setting, because we use a disjunctively-partitioned representation. In our experiments (§6), we show that the benefits of a CFG-guided exploration significantly outweigh this drawback in the verification of sequential programs.

5 Hybrid Tree IC3

It can be observed that the sequence of sets of clauses generated by the Tree-based IC3 for refuting a spurious abstract error path can be seen as an *interpolant* for the path, in the sense used by McMillan in his "lazy abstraction with interpolants" algorithm [18].[6] Recalling the definition of [18], given a sequence of formulas $\Gamma \overset{\text{def}}{=} \varphi_1, \dots, \varphi_n$, an interpolant is a sequence of formulas $I_0, \dots, I_n$ such that: (i) $I_0 \equiv \top$ and $I_n \equiv \bot$; (ii) for all $1 \leq i \leq n$, $I_{i-1} \wedge \varphi_i \models I_i$; (iii) for all $1 \leq i \leq n$, I_i contains only variables that are shared between $\varphi_1 \wedge \dots \wedge \varphi_i$ and $\varphi_{i+1} \wedge \dots \wedge \varphi_n$. Consider now a program path $\mathsf{pc}_0 \rightsquigarrow \mathsf{pc}_n$, and its corresponding sequence of edge formulas $T_{op_1}, \dots, T_{op_n}$ (where T_{op_i} is the formula attached to the edge $(\mathsf{pc}_{i-1}, op_i, \mathsf{pc}_i)$). Then, it easy to see that the trace $F_0, \dots, F_n$ generated by IC3 in refuting such path immediately satisfies points (i) and (ii) above by definition, and, if we consider the sequence ${T_{op_1}}^{\langle 0 \rangle}, \dots, {T_{op_n}}^{\langle n-1 \rangle}$, then ${F_0}^{\langle 0 \rangle}, \dots, {F_n}^{\langle n \rangle}$ satisfies also point (iii).

Under this view, the TREE-IC3 algorithm described in the previous section can be seen as an instance of the lazy abstraction with interpolants algorithm of [18], in which however interpolants are constructed in a very different way. In the algorithm of [18], interpolants are constructed from proofs of unsatisfiability generated by the SMT solver in refuting spurious error paths; as such, the generated interpolants might have a complex Boolean structure, which depends on the structure of the proof generated by the SMT solver. Moreover, they are typically large and possibly very redundant. In the iterative process of expanding and refining ART nodes, it is often the case that interpolants become larger and larger, causing the algorithm to diverge. In fact, in our experiments we have seen several cases in which the interpolant-based algorithm quickly runs out of memory. On the other hand, when the interpolants are "good", the algorithm is quite fast, since interpolants can be quickly generated using a single call to the SMT solver for each spurious error path.

Consider now the case of TREE-IC3. Here, the interpolants generated, being sets of clauses, have a very regular Boolean structure, and experiments have shown that they are often more compact than those generated from proofs, and as such do not cause blow-ups in memory. Furthermore, another important advantage of having interpolants (that is, abstract state formulas in the ART) in the form of sets of clauses is that this allows to perform strengthening of nodes (see §4) at the level of granularity of individual

[6] In fact, a similar observation has been done already for the fully-symbolic IC3 [12,23].

clauses. In [18], strengthening (called "forced covering" there) is an "all or nothing" operation: either the whole abstract state formula φ_P holds at a descendant node φ_N, or no strengthening is performed. As our experimental evaluation in §6 will show, the capability of performing clause-by-clause strengthening is very important for performance.

A drawback of TREE-IC3 is that the construction of the interpolants is typically more expensive than with the proof-based approach, since it requires many (albeit simpler) calls to the SMT solver for each spurious path, and it also requires many potentially-expensive calls to the APPROX-PREIMAGE procedure needed for generalizing IC3 to SMT (see §3). As a solution, we propose a hybrid approach that combines TREE-IC3 with proof-based interpolant generation, in order to get the benefits of both. The main idea of this new algorithm, which we call TREE-IC3+ITP, is that of generating the sets of clauses in the trace of TREE-IC3 starting from the proof-based interpolants, when such interpolants are "good". More specifically, given an abstract error path $\mathsf{pc}_0 \rightsquigarrow \mathsf{pc}_E$, before invoking IC3 on it, we generate an interpolant $I_0, \ldots, I_n$ (for the corresponding edge formulas $T_{op_1}, \ldots, T_{op_n}$) with the efficient proof-based procedures available in interpolating SMT solvers (see e.g. [9]); then, we try to generate clauses from each I_i by converting them to CNF, using an *equivalence-preserving procedure* (and not, as usual, a satisfiability-preserving one), aborting the computation if this process generates too many clauses. Only when this procedure fails, we fall back to generating sets of clauses with the more expensive IC3. This allows us to keep the performance advantage of the proof-based interpolation method when the generated interpolants are "good", while still benefiting from the advantages of a clause-based representation of abstract states outlined above. Despite its simplicity, in fact, this hybrid algorithm turns out to be quite effective in practice, as our experiments in the next section show.

6 Implementation and Experiments

We have implemented the algorithms described in the previous sections on top of the MATHSAT5 SMT solver [14] and the KRATOS software model checker [8]. In this section, we experimentally evaluate their performance.

6.1 Implementation Details

Generalization of IC3 to SMT. Our current implementation uses the theory of Linear Rational Arithmetic (LRA) for modeling program operations. LRA is well supported by MATHSAT5, which implements efficient algorithms for both satisfiability checking and interpolation modulo this theory [11,9]. Moreover (and more importantly), using LRA allows us to implement a simple and not-too-expensive APPROX-PREIMAGE procedure for computing under-approximations of preimages, as required for generalizing IC3 to SMT (see §3). Given a bad cube s and a transition formula $T(X, X')$, the exact preimage of s wrt. T can be computed by converting $s' \wedge T$ to a DNF $\bigvee_i m_i$ and then projecting each of the cubes m_i over the current-state variables X: $\bigvee_i \exists X'.(m_i)$. Then, an under-approximation can be constructed by simply picking only a subset of the projections of the cubes m_i of the DNF. In our implementation, we use the All-SMT-based algorithm of [20] to construct the DNF lazily, and in order to keep the cost of the computation relatively low we under-approximate by simply stopping after the first cube.

Implementation of IC3. In general, our implementation of IC3 follows the description given in [12] (called PDR there). In order to be implemented efficiently, IC3 requires a very tight integration with the underlying SAT (or SMT) solver, and the details of such integration are sometimes crucial for performance. Therefore, here we precisely outline the differences wrt. the description given in [12]. In particular, besides the obvious one of using an SMT solver instead of a SAT solver, the two main differences are:

- For simplicity, we use a single solver rather than a different solver per frame, as suggested in [12]. Moreover, since MATHSAT5 supports both an incremental interface, through which clauses added and removed in a stack-based manner, and an assumptions-based interface, we use a mixture of both for efficiently querying the solver: we use assumptions for activating and deactivating the clauses of the initial states, transition relation, bad states and those of the individual frames, as described in detail in [12], whereas we use the push/pop interface for temporarily adding a clause to the solver for checking whether such clause is inductive (relative to the previously-generated ones). This allows us to avoid the need of periodically cleaning old activation literals as described in [12].
- For reducing bad cubes that must be blocked during the execution of IC3, we exploit the *dual-rail encoding* typically used in Symbolic Trajectory Evaluation [22]. We do not apply ternary simulation through the transition relation, as suggested in [12] for the Boolean case, as we found the former to be much more efficient than the latter. This is possibly because the data structures that we use for representing formulas are relatively naïve and inefficient.

Implementation of Tree-Based IC3. We adopt the "Large-Block" encoding [3] of the control-flow graph of the program under analysis, which collapses loop-free subparts of the original CFG into a single edge, in order to take full advantage of the power of the underlying SMT solver of efficiently reasoning with disjunctions. Currently, we do not handle pointers or recursive functions, and we inline all the function calls so as to obtain a single CFG.

For the construction of the abstract reachability tree, we adopt the "DFS" strategy described by McMillan in [18]. When constructing the ART, we apply strengthening systematically to each uncovered node n which has a proper ancestor p tagged with the same program location, by adding to n all the clauses of p that hold after the execution of the path $p \rightsquigarrow n$, and strengthening the intermediate nodes accordingly.

Finally, in the "hybrid" version, we use a threshold on the size of the CNF conversion for deciding whether to use interpolants for computing the sets of clauses for refuting a spurious path: we compute the sequence of interpolants, and we try to convert them to CNF, aborting the process when the formulas become larger than k times the size of the interpolants (k being a configurable parameter set to 5 in the experiments).[7]

[7] We remark that here we need an *equivalent*, and not just equisatisfiable, CNF representation. Therefore, such conversion might result in an exponential blow-up in the size of the formula.

6.2 Benchmarks and Evaluation

For our evaluation, we use a set of 98 benchmark C programs from the literature, originating from different domains (e.g. device drivers, communication protocols, SystemC designs, and textbook algorithms), most of which have been used in several previous works on software model checking, The set includes the benchmarks used in the first software verification competition (`http://sv-comp.sosy-lab.org`) that can be handled by our implementation. About one third of the programs contain bugs.

All the benchmarks, tools and scripts needed for reproducing the experiments are available at `http://es.fbk.eu/people/griggio/papers/cav12--ic3smt.tar.bz2` The experiments have been run on a Linux machine with a 2.6GHz CPU, using a time limit of 1200 seconds and a memory limit of 2GB.

For our evaluation, we tested the following algorithms/configurations:

IC3 is the fully symbolic version of IC3, in which the CFG is encoded symbolically in the transition relation using an auxiliary variable representing the program counter;[8]

TREE-IC3 is the CFG-based version of IC3, as described in §4;

TREE-IC3+ITP is the hybrid algorithm of §5, in which "good" interpolants are used for computing sets of inductive clauses;

TREE-IC3+ITP-MONO is a variant of TREE-IC3+ITP in which strengthening of nodes is performed "monolithically" by checking whether *all* the clauses of an ancestor node p hold at a descendant node n, instead of checking the clauses individually. This configuration mimics the forced coverage procedure applied in the lazy abstraction with interpolants algorithm of [18];

TREE-ITP is an implementation of the lazy abstraction with interpolants algorithm of [18];

KRATOS is an implementation of lazy predicate abstraction with interpolation-based refinement, the default algorithm used by the KRATOS software model checker [8]. (We recall that the difference with TREE-ITP is that in standard lazy predicate abstraction interpolants are used only as a source of new predicates, and abstract states are computed using Boolean abstraction rather than using interpolants directly.)

All the implementations use the same front-end for parsing the C program and computing its CFG, and they all use the same SMT solver (MATHSAT5 [14]) as a back-end reasoning engine for all satisfiability checks and interpolation queries. Moreover, all the tree-based algorithms use the same depth-first strategy for constructing the ART, and all the IC3-based ones use the same settings for IC3. This makes it possible to compare the merits of the various algorithms (over the benchmark instances) without being affected by potential differences in the implementation of other parts of the systems which are orthogonal to the evaluation.

Moreover, in addition to comparing the different algorithms within the same implementation framework, we also compared our best algorithm with the following software model checkers:

[8] More precisely, we encode the program counter variable using $\lceil \log_2 n \rceil$ Boolean variables, where n is the number of locations in the CFG.

CPAchecker [4], which uses an algorithm based on lazy predicate abstraction [16] (like Kratos). CPAchecker was the winner of the first software verification competition.[9]

Wolverine [17], an implementation of the lazy abstraction with interpolants algorithm [18] (like Tree-ITP). [10]

6.3 Results

The results of the evaluation are summarized in Figure 4. The scatter plots in the top row show the comparisons of the various configurations of IC3 proposed in the previous Sections: first, we compare the fully-symbolic IC3 with Tree-IC3, in order to evaluate the benefits of exploiting the CFG of the program; then, we evaluate the effect of using interpolants for computing sets of inductive clauses (Tree-IC3+ITP) wrt. "plain" Tree-IC3; third, we evaluate the impact of the fine-grained strengthening that is possible when using a conjunctively-partitioned representation for abstract states, by comparing Tree-IC3+ITP with Tree-IC3+ITP-mono. The rest of the plots show instead the comparison of our best configuration, Tree-IC3+ITP, with alternative algorithms and implementations. A summary of the performance results for all the algorithms/-configurations is reported in the table, showing the number of instances successfully checked within the timeout, and the total execution time for the solved instances.

From the plots and the table of Figure 4, we can draw the following conclusions:

- All the techniques proposed in this paper lead to significant improvements to IC3, with Tree-IC3 solving 17 more instances than IC3 (and being up to two orders of magnitude faster), and Tree-IC3+ITP solving 11 more instances than Tree-IC3;
- On relatively-easy problems, the IC3-based algorithms are generally more expensive than the alternative techniques; in particular, the tools based on predicate abstraction (Kratos and CPAchecker) perform very well in terms of execution time. However, on harder benchmarks Tree-IC3+ITP seems to be more robust than the competitors. This is particularly evident for Tree-ITP and CPAchecker, which run out of memory in 25 and 34 cases respectively, [11] whereas this never happens with Kratos and Tree-IC3+ITP.
- The ability to perform clause-by-clause strengthening is very important for the performance of Tree-IC3+ITP: when using a "monolithic" approach, Tree-IC3+ITP (Tree-IC3+ITP-mono) is not only almost always slower, but it also solves 13 instances less. However, we notice that even without it, Tree-IC3+ITP-mono behaves better than Tree-ITP, and in particular it is significantly more robust in terms of memory consumption.

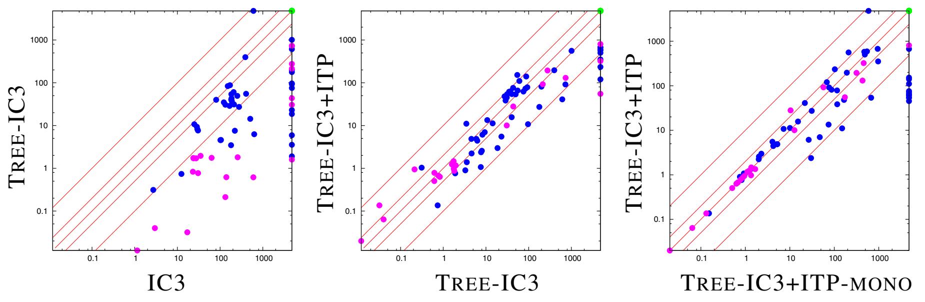
7 Related Work

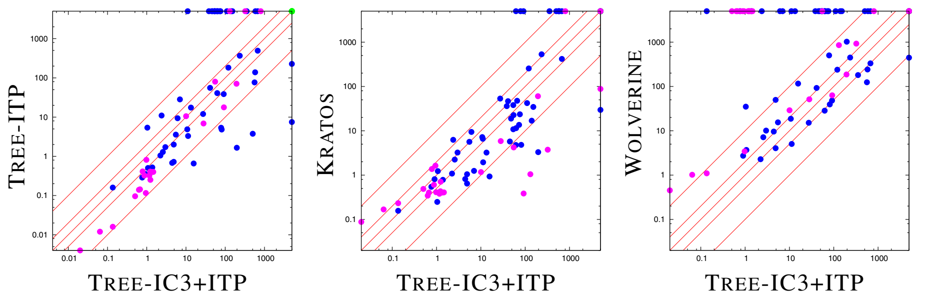
Besides all the IC3-related approaches in the hardware domain [6,5,7,12], as already stated in §5 the work that is most closely-related to ours is the "lazy abstraction with

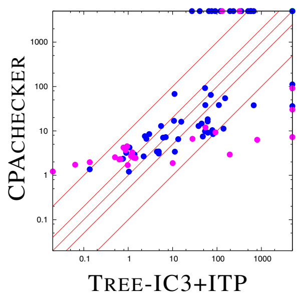
[9] See `http://sv-comp.sosy-lab.org/results/index.php`

[10] It would have been interesting to include also the Impact tool of [18] in the comparison; however, the tool is no longer available.

[11] Notice that with CPAchecker this happens even when increasing the memory limit to 4GB.







Algorithm/Tool	**# solved**	**Tot time**
TREE-IC3+ITP	74	7654
KRATOS	66	1898
CPACHECKER	63	988
TREE-IC3	63	5548
TREE-IC3+ITP-MONO	61	7727
TREE-ITP	55	1934
IC3	46	7166
WOLVERINE	37	6160

Fig. 4. Experimental results

interpolants" technique of McMillan [18]. Both TREE-IC3 and TREE-IC3+ITP, in fact, can be seen as instances of the "lazy abstraction with interpolants" algorithm, in which however interpolants are computed using the IC3 algorithm and approximate preimage computations, rather than proofs of unsatisfiability produced by the SMT solver. This in turn leads to interpolants that can be easily partitioned conjunctively, which allows to significantly improve their usefulness in pruning the number of abstract paths that need to be explored (see §5 and §6).

Another approach based on interpolation and explicit exploration of CFGs is described in [19]. In this approach, the search in the CFG is guided by symbolic execution; moreover, a learning procedure inspired by DPLL-based SAT solvers is applied in order to generate new annotations that prevent the exploration of already-visited portions of the search space. Such annotations are obtained from interpolants, generated from proofs. In principle, it should be possible to apply IC3-based ideas similar to those that we have presented also in that context.

Some analogies between the present work and the "DASH" approach described in [2,13], which analyzes programs with a combination of testing and verification, can be seen in the use of approximate preimages and of highly-incremental SMT queries. However, in another sense DASH is somewhat orthogonal to IC3-based techniques, in that the latter could be used as a verification engine for the former. (In fact, although in [2] an approach based on weakest preconditions is used, interpolation is suggested as a potential alternative.)

Finally, the use of a clause-based representation for abstract states bears some similarities with the work in [15]. However, the exploration of the CFG and the whole verification approach is very different, and the two approaches can be considered largely orthogonal. In [15], the CFG is treated as a Boolean formula, whose satisfying assignments, enumerated by a SAT solver, correspond to path programs that are checked with different verification oracles. The invariants computed by such oracles are then used to construct blocking clauses that prevent re-exploration of already-covered parts of the program. Both the fully-symbolic and the tree-based versions of IC3 that we have presented could be used as oracles for checking the path programs and generating invariants for the blocking clauses.

8 Conclusions and Future Work

We have presented an investigation on the application of IC3 to the case of software. We propose three variants: the first one, generalizing IC3 to the case of SMT, provides for the analysis of fully symbolically represented software; the second one, TREE-IC3, relies on an explicit treatment of the CFG; the third one is a hybrid appraoch based on the use of interpolants to improve TREE-IC3.

IC3 is a radically new verification paradigm, and has a great potential for future developments in various directions. First, we intend to investigate further IC3 in the setting of SMT, devising effective procedures for APPROX-PREIMAGE in other relevant theories, and adding low-level optimization techniques, similarly to the highly tuned techniques used in the Boolean case. Second, we intend to investigate the extraction of CFG's from hardware designs, in the same spirit as the transition-by-transition approach [21], and to apply IC3 directly to descriptions in high-level languages such as Verilog or VHDL. Finally, we intend to extend IC3 to richer theories such as bit vectors and arrays, and to the case of networks of hybrid systems [10].

Acknowledgements. We would like to thank Ken McMillan and the anonymous referees for their very helpful comments on early versions of the paper.

References

1. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Satisfiability, vol. 185, pp. 825–885. IOS Press (2009)
2. Beckman, N.E., Nori, A.V., Rajamani, S.K., Simmons, R.J.: Proofs from tests. In: Proc. of ISSTA, pp. 3–14. ACM (2008)

3. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software model checking via Large-Block Encoding. In: Proc. of FMCAD, pp. 25–32. IEEE (2009)
4. Beyer, D., Keremoglu, M.E.: CPACHECKER: A Tool for Configurable Software Verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011)
5. Bradley, A., Somenzi, F., Hassan, Z., Zhang, Y.: An incremental approach to model checking progress properties. In: Proc. of FMCAD (2011)
6. Bradley, A.R.: SAT-Based Model Checking without Unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011)
7. Chokler, H., Ivrii, A., Matsliah, A., Moran, S., Nevo, Z.: Incremenatal formal verification of hardware. In: Proc. of FMCAD (2011)
8. Cimatti, A., Griggio, A., Micheli, A., Narasamdya, I., Roveri, M.: KRATOS – A Software Model Checker for SystemC. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 310–316. Springer, Heidelberg (2011)
9. Cimatti, A., Griggio, A., Sebastiani, R.: Efficient generation of craig interpolants in satisfiability modulo theories. ACM Trans. Comput. Log. 12(1), 7 (2010)
10. Cimatti, A., Mover, S., Tonetta, S.: Efficient Scenario Verification for Hybrid Automata. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 317–332. Springer, Heidelberg (2011)
11. Dutertre, B., de Moura, L.: A Fast Linear-Arithmetic Solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
12. Een, N., Mishchenko, A., Brayton, R.: Efficient implementation of property-directed reachability. In: Proc. of FMCAD (2011)
13. Godefroid, P., Nori, A.V., Rajamani, S.K., Tetali, S.: Compositional may-must program analysis: unleashing the power of alternation. In: Proc. of POPL, pp. 43–56. ACM (2010)
14. Griggio, A.: A Practical Approach to Satisfiability Modulo Linear Integer Arithmetic. JSAT 8 (2012)
15. Harris, W.R., Sankaranarayanan, S., Ivancic, F., Gupta, A.: Program analysis via satisfiability modulo path programs. In: Proc. of POPL, pp. 71–82. ACM (2010)
16. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proc. of POPL, pp. 58–70 (2002)
17. Kroening, D., Weissenbacher, G.: Interpolation-Based Software Verification with WOLVERINE. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 573–578. Springer, Heidelberg (2011)
18. McMillan, K.L.: Lazy Abstraction with Interpolants. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006)
19. McMillan, K.L.: Lazy Annotation for Program Testing and Verification. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 104–118. Springer, Heidelberg (2010)
20. Monniaux, D.: A Quantifier Elimination Algorithm for Linear Real Arithmetic. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 243–257. Springer, Heidelberg (2008)
21. Nguyen, M.D., Stoffel, D., Wedler, M., Kunz, W.: Transition-by-transition FSM traversal for reachability analysis in bounded model checking. In: Proc. of ICCAD. IEEE (2005)
22. Roorda, J.-W., Claessen, K.: SAT-Based Assistance in Abstraction Refinement for Symbolic Trajectory Evaluation. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 175–189. Springer, Heidelberg (2006)
23. Somenzi, F., Bradley, A.: IC3: Where Monolithic and Incremental Meet. In: Proc. of FMCAD (2011)