IC3 Software Model Checking on Control Flow Automata

Tim Lange
RWTH Aachen University, Germany
tim.lange@cs.rwth-aachen.de

Martin R. Neuhäusser
Siemens AG, Germany
martin.neuhaeusser@siemens.com

Thomas Noll
RWTH Aachen University, Germany
noll@cs.rwth-aachen.de

Abstract—In recent years, the inductive, incremental verification algorithm IC3 had a major impact on hardware model checking. Also with respect to software model checking, a number of adaptations of Boolean IC3 and combinations with CEGAR and ART-based techniques have been developed. However, most of them exploit the peculiarities of software programs, such as the explicit representation of control flow, only to a limited extent. In this paper, we propose a technique that supports this explicit representation in the form of control flow automata, and integrates it with symbolic reasoning about the data state space of the program. It thus provides a true lifting of IC3 from hardware to software model checking. By evaluating the approach on a number of case studies using a prototypical implementation, we demonstrate that our method shows promising results.

I. INTRODUCTION

IC3 [1] is an incremental algorithm that has originally been designed for verifying invariant properties of finite transition systems. It constructs an over-approximation of the reachable state space by generating Boolean clauses that are inductive relative to stepwise reachability information. During this construction, candidate counterexamples are being disproved using Boolean SAT-solving techniques. This approach has turned out to be highly effective; in fact, it is considered to be one of (if not the) most important contribution of bit-level formal verification of hardware systems for the last decade.

There have been several attempts to lift Boolean IC3 to the domain of software model checking. As this setting usually induces infinite-state systems, more advanced symbolic reasoning techniques are required. The most prominent one is Satisfiability Modulo Theories (SMT). Here, sets of states are symbolically specified by first-order formulas over constraints from the respective theory, and SMT-solving techniques are employed to rule out spurious counterexamples.

One of the first integrations of SMT into IC3 has been presented in [2]. In addition to the generalisation of SAT to SMT solving, it exploits the partitioning of the program’s state space as induced by its control flow graph. This is achieved by unwinding the latter into an Abstract Reachability Tree (ART) in which each node is associated with a control location and a formula, resulting in an “explicit-symbolic” approach named Tree-IC3. Candidate counterexamples are handled by computing under-approximations of pre-images.

The advantages in comparison to Boolean IC3 are twofold. First, Tree-IC3 eliminates the possible redundancy of subformulas partitioning of the control state space, the solver is exposed to simpler and smaller formulas.

On the downside, the key idea underlying IC3, relative inductiveness, cannot be directly applied in this setting due to the partitioned representation that leads to a path-wise unwinding of the transition system. The follow-up publication [3] therefore reverts to a monolithic transition relation, replacing the pre-image computation by (implicit) predicate abstraction. The latter is a standard abstraction technique that partitions the state space according to the equivalence relation induced by a set of predicates. Its implicit variant [4] allows to express abstract transitions without explicitly computing the abstract system. In the IC3 setting, this avoids theory-specific generalisation techniques.

In [5], Horn clauses are employed to represent recursive predicate transformers. Proof obligations are generalised using a specialised interpolation procedure for linear arithmetic. However, the latter again does not exploit relative induction.

In summary, existing work exploits the peculiarities of software programs only to a limited extent to support IC3-style verification. In this paper, we develop an approach that combines the advantage of explicitly handling the control flow of a program, employing a corresponding automata model, with relative inductive reasoning over a symbolic representation of its data space. It thus provides a true lifting of IC3 from hardware to software model checking. We demonstrate the applicability and efficiency of our method by evaluating it on a number of case studies using a prototypical implementation.

The remainder of this paper is organised as follows. We start by introducing some general concepts in Section II. Section III sets the stage for our contribution with a description of the original IC3 algorithm, which is then extended in Section IV by taking control flow automata into account. Results of the experimental evaluation are given in Section V. Section VI concludes the paper with a summary and a description of future work.

II. PRELIMINARIES

A control flow automaton (CFA) $A = (L, G)$ consists of a finite set of locations $L = \{0, \ldots, n\}$, modeling the program counter of a corresponding sequential code, and edges in $G \subseteq L \times FO \times L$ labeled with quantifier-free first-order formulas over the set $Var$ of program variables and their next-state primed forms, $Var’$ [7]. Priming of a formula, $\varphi’$, is
the same as priming every variable in \( \varphi \). Such a formula can either encode a variable assignment, containing primed and unprimed variables, or an assume statement in which case it will only contain unprimed variables. We assume that for any two locations there exists at most one edge between them.

Given a subset of variables \( X \subseteq \text{Var} \), a cube over \( X \) is defined as a conjunction of literals, each literal being a variable or its negation in the propositional case and a theory atom or its negation in the quantifier-free first-order case. The negation of a cube, i.e., a disjunction of literals, is called a clause.

A program \( \mathcal{P} = (A, l_0, l_E) \) consists of a CFA \( A \) representing the control flow, as well as an initial location \( l_0 \in L \) and an error location \( l_E \in L \). This representation allows to encode arbitrary programs and assertions for safety verification. For every assertion that has to be verified at location \( l \), split \( l \) and introduce an edge with the negated assertion to \( l_E \) and an edge with the positive assertion between the split nodes. This way, checking the violation of assertions becomes checking reachability of \( l_E \) in \( A \).

Given two locations \( l_1, l_2 \in L \), we define the transition formula

\[
T_{l_1 \rightarrow l_2} = \begin{cases} \text{false} & \text{if } (l_1, l_2) \in G \\ \phi & \text{otherwise.} \end{cases} \tag{1}
\]

This yields the global transition formula as

\[
T = \bigvee_{(l_1, l_2) \in G} T_{l_1 \rightarrow l_2}. \tag{2}
\]

**Definition 1 (Relative inductiveness):** Given a transition formula \( T \), a formula \( \varphi \) is inductive relative to another formula \( \psi \) if

\[
\psi \wedge \varphi \wedge T \Rightarrow \varphi' \tag{3}
\]

is valid.

Starting from Def. \( I \) we can refine relative inductiveness to consider only a single transition rather than the whole transition relation.

**Definition 2 (Edge-relative inductiveness):** Given a CFA \( A \) and locations \( l_1, l_2 \in L \), a formula \( \varphi \) is edge-relative inductive to another formula \( \psi \) if

\[
\psi \wedge \varphi \wedge T_{l_1 \rightarrow l_2} \Rightarrow \varphi' \tag{4}
\]

is valid.

Note that edge-relative inductiveness does also hold if \( (l_1, t, l_2) \notin G \) for every \( t \). In this case, \( T_{l_1 \rightarrow l_2} = \text{false} \), which makes \( \psi \) hold trivially, i.e., if we are in a state satisfying \( \varphi \) and we cannot leave it via the considered edge, we remain in a state satisfying \( \varphi \).

To handle the possibly infinite state space over \( \text{Var} \), we use a symbolic representation through quantifier-free first-order formulas.

**Definition 3 (Data region):** A data region is represented by a quantifier-free FO formula \( s \) over \( \text{Var} \) and consists of all variable assignments \( \sigma \) satisfying \( s \), i.e., \( \{ \sigma \mid \sigma \models s \} \).

Based on Def. \( I \) we can augment a data region with a control flow location \( l \in L \). This information, sometimes referred to as atomic region \( \{ l \} \) is called region in the following.

**Definition 4 (Region):** We define a region \( r = (l, s) \) as a pair consisting of location \( l \in L \) and data region \( s \). Given such a region \( r \equiv (l, s) \), the corresponding formulas are defined as \( \{ \phi \mid \phi \equiv (pc = l \land s) \} \). Analogously the corresponding formulas for a negated region \( \neg r \equiv (l, s) \) are defined as \( \{ \phi \mid \phi \equiv \neg (pc = l \land s) \} \).

Given two regions \( r_1, r_2 \) and representatives of their corresponding propositional formulas \( \varphi_1, \varphi_2 \), then \( r_1 \) is inductive relative to \( r_2 \) iff \( \varphi_1 \) is inductive relative to \( \varphi_2 \). The analogue can be defined for the special case of edge-relative inductive-ness.

Using their corresponding formula, we can use relative and edge-relative inductiveness for regions very similar to \( I \). Even though it works for two non-negated regions as well, the IC3 algorithm only makes use of the case where we check whether a negated region \( \neg r_2 \) is inductive relative to a non-negated region \( r_1 \). Therefore we will only consider this case in the following and inspect it in detail. We found two different cases whose premise can be statically determined and that simplify the SMT queries that we have to use.

**Lemma 1 (Relative inductive regions):** Assuming two regions \( r_1 = (l_1, s_1) \), \( r_2 = (l_2, s_2) \) with corresponding formulas \( \varphi_1 \) and \( \varphi_2 \), we have:

\[
\varphi_1 \equiv (pc = l_1 \land s_1) \quad \neg \varphi_2 \equiv \neg (pc = l_2 \land s_2)
\]

**Proof 1:** Given two regions \( r_1 = (l_1, s_1) \) and \( r_2 = (l_2, s_2) \), with corresponding formulas \( \varphi_1 \) and \( \varphi_2 \), we have:

\[
\varphi_1 \equiv (pc = l_1 \land s_1) \quad \neg \varphi_2 \equiv \neg (pc = l_2 \land s_2)
\]

Def. \( I \) yields:

\[
(pc = l_1 \land s_1) \land \neg (pc = l_2 \land s_2) \land T_{l_1 \rightarrow l_2} \Rightarrow \neg s_2', \quad \text{if } l_2 \neq l_1 \tag{5}
\]

\[
s_1 \land \neg s_2 \land T_{l_1 \rightarrow l_2} \Rightarrow \neg s_2', \quad \text{if } l_2 = l_1 \tag{6}
\]

If \( l_1 \neq l_2 \), this is equisatisfiable to

\[
(true \land s_1) \land (true \lor \neg s_2) \land T_{l_1 \rightarrow l_2} \Rightarrow (false \lor \neg s_2'), \quad \text{if } l_2 \neq l_1
\]

\[
\equiv s_1 \land T_{l_1 \rightarrow l_2} \Rightarrow \neg s_2'. \tag{7}
\]

Otherwise, we obtain

\[
(true \land s_1) \land (false \lor \neg s_2) \land T_{l_1 \rightarrow l_2} \Rightarrow (false \lor \neg s_2'), \quad \text{if } l_2 = l_1
\]

\[
\equiv s_1 \land \neg s_2 \land T_{l_1 \rightarrow l_2} \Rightarrow \neg s_2'. \tag{8}
\]

\[ \quad \square \]

In Proof \( I \) we can use the presented equisatisfiable transformations because the transition from \( l_1 \) and \( l_2 \) implicitly contains the atoms \((pc = l_1)\) and \((pc' = l_2)\). Therefore in the first case, where \( l_1 \neq l_2 \), the atoms \((pc = l_1)\) and \((pc' = l_2)\) can be rewritten to \( true \), while the atom \((pc' = l_2)\) can be
rewritten to \textit{false}. The analogous holds for the case where \( l_1 = l_2 \).

Given a program \( P \), we can define a finite path \( \pi \) of length \( n \) as a sequence \( l_0, l_1, \ldots, l_n \), s.t. for every \( 0 \leq i < n \) there exists an edge \((l_i, l_i, l_{i+1}) \in G\) in \( P \). A path \( \pi \) is called \textit{feasible} if for every \( l_j \) in \( \pi \) we can construct a region \((l_j, s_j)\) that is non-empty, i.e. \( s_j \neq \textit{false} \), s.t. \( s_i \land T_{l_i \rightarrow l_{i+1}} \Rightarrow s_{i+1} \) for \( 0 \leq i < n \).

### III. Original IC3 Algorithm

Let \( S = (X, I, T) \) be a transition system over a set \( X \) of Boolean variables, and \( I(X) \) and \( T(X, X') \) two propositional formulas respectively describing the initial condition and the transition relation over variables in \( X \) and next-state primed successors \( X' \). Given a propositional property \( P(X) \), we want to verify that every state in \( S \) that is reachable from a state in \( I \) satisfies \( P \). Sometimes also an inverted formulation is used like in [9] where \( \neg P \) states are \textit{bad states} and we want to show that no bad state is reachable from the initial states. The main idea of the IC3 algorithm [1] and the earlier \textit{finite state inductive strengthening (FSIS) [10]} is that if \( P \) is inductive, i.e. \( I \Rightarrow P \) and \( P \land T \Rightarrow P' \), then \( P \) is also an invariant on \( S \). However, even if \( P \) is an invariant on \( S \) it may not be inductive. Therefore the goal of IC3 and FSIS is to produce a so called \textit{inductive strengthening} \( F \) of property \( P \), s.t. \( F \land T \) is inductive. This means that we can restrict the set of states in \( P \) to a smaller set of states in the intersection of \( F \) and \( P \), which still contains all states reachable from \( I \), but excludes unreachable states that will lead to a violation of induction. While FSIS tries to come up with such a strengthening in one step, IC3 proceeds in a more relaxed approach and constructs \( F \) incrementally.

This incremental construction is based on a sequence of frames \( F_0, \ldots, F_k \) for which

\[
I \Rightarrow F_0 \\
F_i \Rightarrow F_{i+1} \quad \text{, for } 0 \leq i < k \\
F_0 \Rightarrow P \quad \text{, for } 0 \leq i \leq k \\
F_i \land T \Rightarrow F'_{i+1} \quad \text{, for } 0 \leq i < k
\]

has to hold in order to produce an inductive invariant. The algorithm starts with two initial checks for 0- and 1-step reachable states in \( \neg P \) and afterwards initializes the first frame \( F_0 \) to \( I \). The rest of the algorithm can be divided into an inner and an outer loop, sometimes also referred to as \textit{blocking} and \textit{propagation} phases, respectively.

The outer loop iterates over the maximal frame index \( k \), looking for states in \( F_k \) that can reach \( \neg P \), so called \textit{counterexamples to induction (CTI)}. If such a CTI exists, it is analyzed in the inner loop, the blocking phase. If no such CTI exists, IC3 tries to propagate clauses learned in frame \( F_i \) forward to \( F_{i+1} \). In the end it checks for termination, which is given if \( F_i = F_{i+1} \) for some \( 0 \leq i < k \).

The objective of the blocking phase is to decide whether a CTI is reachable from \( I \) or not. For this purpose, it maintains a set of pairs of frame indices and states, called \textit{proof obligations}. From this set it picks the pair \((i, s)\) with the smallest frame index \( i \). If there is more than one pair with this frame index, the choice between those is arbitrary. For the chosen state, IC3 checks whether \( \neg s \) is relative inductive to \( F_{i-1} \), using [3]. If it is relative inductive, we can block \( s \) in frames \( F_j \) for \( 0 \leq j \leq i \). But rather than just adding \( \neg s \), IC3 first tries to obtain a clause that is a subset of \( \neg s \) and therefore excludes more states. This clause, called a \textit{generalization} of \( \neg s \), is then added to the frames and afterwards the pair \((i, s)\) in the set of proof obligations is replaced by \((i+1, s)\). If \( s \) is not relative inductive to \( F_{i-1} \) this means that there exists an \( F_{i-1} \) predecessor \( p \) that can reach \( s \). IC3 therefore adds \((i-1, p)\) to the set of proof obligations. The blocking phase terminates if either there exists an \( s \) in the set of proof obligations that is relative inductive to an initial state at index 0, in which case there exists a counterexample path, or for every proof obligation the frame index \( i > k \), i.e. there exists a \( j \geq 0 \), s.t. every predecessor of the original CTI is inductive relative to \( F_j \).

### IV. IC3 on Control Flow Automata

In this section we will present our IC3 algorithm for control flow automata as annotated pseudocode, give a short explanation and a proof of partial correctness, followed by an example showing the benefits of our method.

The most straightforward way to lift IC3 to software model checking is to encode the control flow in an additional \textit{pc} variable representing the program location, as presented in [2]. However, this approach introduces some tedious handling of the implicit \textit{pc} variable and is not very competitive. One reason is that the control flow of the input program already gives a very clear structure to the system, which is completely disregarded when encoded inside a global transition formula. Thus our approach tries to exploit as much of the structure given in the input program as possible.

In [1] Bradley draws an analogy between the way IC3 proves properties on a transition system and how a human analyzes a system - by producing a set of lemmas s.t. each holds relative to a previous one and that all together imply the property. It is that stepwise approach that makes IC3 so competitive and which motivated us to apply our version of IC3 directly to a control flow automaton as an explicit representation of a program’s possible execution steps.

With respect to the definition of programs we follow the notion of [9] and reason about error states, rather than property states.

The explicit representation of edges leads to a situation where we can reduce the possible transitions for a region \( r = (l, s) \) to those that are available from \( l \) in the program \( P \), which allows us to formulate significantly smaller solver queries. Explicit initial and error states enable us to statically check 0-step and (potential) 1-step reachability, as well as to avoid initial and error conditions, which in turn reduces the size of the solver queries even further.
In analogy to bit-level IC3, we construct frame sequences \(F_0, \ldots, F_k\), but instead of using global frames, we use location-local frames \(F_{i, l}\) in every \(l \in G\). We interpret those \(F_{i, l}\) as the set of, possibly overapproximating, data regions reachable in at most \(i\) steps at location \(l\).

Algorithm 1 Outer loop

Ensure: ret. value iff \(l_E\) is reachable

function BOOL PROVE
if \(l_0 = l_E\) or \((l_0, t, l_E) \in G\) and sat\((t)\) then
    return false
initialize frames
for \(k = 1\) to \(\ldots\) do
    if not STRENGTHEN\((k)\) then
        return false
    propagate
    if termination then
        return true

Algorithm 2 Strengthening

Require: (a) \(k \geq 1\)

Require: (b) \(\forall i \geq 0, l \in L, F_{i, l} \Rightarrow F_{i+1, l}\)

Require: (c) \(\forall 0 \leq i < k, l, l' \in L, \text{s.t. } (l, t, l') \in G, F_{i, l} \wedge T_{1 \rightarrow l'} \Rightarrow F'_{i+1, l'}\)

Ensure: \(\forall i \geq 0, l \in L, F_{i, l} \Rightarrow F_{i+1, l}\)

Ensure: if ret. value then \(\forall 0 \leq i < k, l, l' \in L, \text{s.t. } (l, t, l') \in G, F_{i, l} \wedge T_{1 \rightarrow l'} \Rightarrow F'_{i+1, l'}\)

Ensure: if ¬ret. value, there exists a counterexample path

function BOOL STRENGTHEN\((k)\): int
while \(\exists l, s, \text{sat}(F_{k, l} \wedge T_{1 \rightarrow l_E})\) do
    assert (b),(c)
    \(s :=\) predecessor data region
    if not BACKWARDBLOCK\((k, l, s)\) then
        return false
    @assert \(s \neq F_{k, l}\)
return true

The function STRENGTHEN in Alg. 2 also works similarly to the original IC3. The main difference here is in the condition of the while loop which checks whether there exists \(l \in L\) s.t. \(e = (l, t, l_E) \in G\) and \(t\) is satisfiable under \(F_{k, l}\). If this is the case, there exists a CTI. Note that in this paper we do not tackle the whole topic of generalization and restrict ourselves to computing weakest preconditions (WP) of the error state w.r.t. \(t\). While this might not offer the best performance possible, it is a safe approximation, as the WP is the smallest overapproximation of CTI states. The so extracted predecessor \(s\) is then analyzed in the function BACKWARDBLOCK, shown in Alg. 3.

Algorithm 3 Inner loop

Require: (b),(c)

Require: \(\text{sat}(F_{i, l'}) \wedge \hat{s} \wedge T_{l' \rightarrow l_E}\)

Ensure: if ret. value, then ¬\(s\) is inductive relative to \(F_{i-1, l}\), \(\forall l, \text{s.t. } (l, t, l') \in G\)

Ensure: if ret. value, then (b),(c)

Ensure: if ¬ret. value, there exists a feasible path \(l_0 \Rightarrow \hat{l}'\)

function BOOL BACKWARDBLOCK\((i): \text{int, } l': \text{location, } \hat{s}: \text{data region}\)

while \(|Q| > 0\) do
    assert \(\forall (i, l', s) \in Q, 0 \leq i \leq k\)
    assert \(\forall (i, l', s) \in Q, \exists \text{path } (l', s) \Rightarrow (l_E, \text{true})\)
    \((i, l', s) = \text{Q.pop}\)
    if \(i = 0\) then
        return false
    else
        @assert \((l', \neg s)\) is inductive relative to \(F_{j, l}\), \(\forall 0 \leq j < i, l \in L \setminus \{l_E\}\)
        for each \(l, \text{s.t. } (l, t, l') \in G\) do
            if \(l = l'\) and \(\text{sat}(F_{i-1, l} \wedge \neg s \wedge T_{l \rightarrow l'})\) then
                generate predecessor \(c\) of \(s\)
                assert \(\forall (i, l', s) \in Q, c \neq s\)
                add \((i-1, l, c)\) and \((i, l', s)\) to \(Q\)
            else
                if \(l \neq l'\) and \(\text{sat}(F_{i-1, l} \wedge T_{l \rightarrow l'} \wedge s')\) then
                    generate predecessor \(c\) of \(s\)
                    assert \(\forall (i, l', s) \in Q, c \neq s\)
                    add \((i-1, l, c)\) and \((i, l', s)\) to \(Q\)
                else
                    block \(s\) in frames \(F_{j, l'}\) for \(0 \leq j < i\)
        return true

While Alg. 1 and 2 are based on [11], we decided to present the inner loop similarly to the representation in [9] as we found it easier to comprehend. The function BACKWARDBLOCK gets as parameter a frame index \(\hat{i}\), a location \(l' \in L\) and a data region \(\hat{s}\). Following [9] we add this initial proof obligation to a priority queue \(Q\), s.t. the obligation with the lowest \(i\) will get popped first. While the queue is non-empty, we start the inner loop by picking the obligation with the smallest \(i\). If \(i = 0\), we can immediately stop with a counterexample because \(l'\) has to be initial. If it were not initial, the previous proof
obligation at level 1 would have included the frame \( F_{(0,l)} \), which is \( \text{false} \) for every non-initial location \( l \). If \( i \neq 0 \) we have to check whether the region \( \neg(l', s) \) is inductive edge-relative to \( F_{(i-1,l)} \) of any predecessor \( l \) by solving the query (5) or (6), depending on whether \( l = l' \) or not. If \( \neg(l', s) \) is inductive edge-relative to \( F_{(i-1,l)} \) for every predecessor \( l \), we can block \( s \) in all \( F_{(j,l'}, 0 \leq j \leq i \). If, on the other hand, \( \neg(l', s) \) is not inductive edge-relative to \( F_{(i-1,l)} \) for some \( i \), then there must exist a predecessor that can reach \( (l', s) \). We therefore take the WP \( c \) of \( s \) w.r.t. the transition formula and add the new proof obligation \( (i-1, l, c) \) to the obligation queue. We also add the old obligation \( (i, l', s) \) to the queue for future re-inspection.

The inner loop terminates with \( \text{true} \) in case that \( Q \) is empty or with \( \text{false} \) in case there exists an obligation at frame 0.

Note that Alg. 3 slightly differs from the idea of blocking a region \( r \) iff \( r \) is edge-relative inductive to all incoming edges. However, the presented algorithm behaves correctly: Due to the ordering of the obligation queue, the algorithm proceeds in a kind of depth-first search manner. Even if \( r \) has been blocked at level \( i \) via one edge, another obligation \( r' \), that is a predecessor of \( r \), at level \( i - 1 \) will be chosen. Here we can distinguish two cases: Either \( r' \) is the last step on a counterexample path from \( l_0 \) to \( r \) at level \( i \) or it is not, in which case all regions explored will be blocked. In both cases \( r \) at level \( i \) will not be reconsidered before backtracking to an obligation at level \( i + 1 \), in which case \( r \) at level \( i \) has been blocked.

In the following we will show that our algorithm is partially correct, i.e. it is correct given its termination. We construct our proof bottom-up by first proving that Alg. 3 is correct. As we use weakest preconditions, we can actually show full correctness, because termination is achieved for the following reason: Given an initial obligation \( (i, l, c) \) for every \( i \) we have to construct at most \( n^2 \) proof obligations, where \( n \) is the maximal in-degree of locations in \( L \). This upper bound can be established because a WP of \( s \) covers all, possibly infinitely many, predecessor data regions that can reach \( s \) w.r.t. the transition. Therefore we cover all data regions in one step and only have to construct predecessor regions until we reach frame level 0, i.e., after doing this \( i \) times.

**Lemma 2:** Function \( \text{BACKWARDBLOCK} \) returns \( \text{true} \) iff \( \neg \tilde{s} \) is inductive relative to \( F_{(i-1,l)} \) for all locations \( l \) that are a predecessor of \( \tilde{l} \).

**Proof 2:** Function \( \text{BACKWARDBLOCK} \) starts the while loop by examining the proof obligation in the queue that has the lowest frame index \( i \). If the frame index is zero, then \( \tilde{l} \) must be \( l_0 \), because \( F_{(0,l)} \) is the only region with frame index 0 to which any other region is relatively inductive, by construction. This way, there must exist a feasible path from \( l_0 \) to \( \tilde{l} \).

If there exists no feasible path from \( l_0 \) to \( \tilde{l} \) given \( F_0, ..., F_k \), then every path of length \( j \) ending in \( \tilde{l} \) starts in a location \( l' \), s.t. the region \( (l, \neg s) \) is inductive relative to \( F_{(k-j-1,l')} \) for all \( l' \), s.t. \((l', t, l) \in G \). This means that every proof obligation added to \( Q \) is ultimately inductive relative to its predecessors and thus also \( \neg \tilde{s} \) is inductive relative to \( F_{(i-1,l)} \).

We continue by proving correctness of Alg. Here we can guarantee termination for the same reason as for \( \text{BACKWARDBLOCK} \). Because we only search for CTIs, we have at most \( n_E \) of them, where \( n_E \) is the number of predecessor locations of \( l_E \) in \( P \). As we compute exact pre-images by weakest preconditions, every data region has exactly one predecessor data region per edge.

**Lemma 3:** Function \( \text{STRENGTHEN} \) terminates with result \( \text{true} \) iff there exists an inductive strengthening for the frames \( F_{(k,l)} \) in all locations \( l \) in the CFA.

**Proof 3:** Assume a call of function \( \text{STRENGTHEN} \) returns \( \text{false} \), then there must have been a call to \( \text{BACKWARDBLOCK} \) with some \((k,l,s)\), such that \( \text{BACKWARDBLOCK} \) returned \( \text{false} \). From Lemma 2 we know that in this case, there exists a feasible path of length \( k \) from \( l_0 \) to \( l \) that ends up in data region \( s \). Because \( l \) is a predecessor of \( l_E \) and \( s \) is a precondition under \( T_{l_0 \rightarrow l_E} \), there exists a counterexample path of length \( k + 1 \). Otherwise every call of \( \text{BACKWARDBLOCK} \) returned \( \text{true} \), which means that every predecessor location (and data region) of \( l_E \) is unreachable in the current frame sequence. Thus every predecessor of \( l_E \) was excluded from their frames at level \( k \) which yields an inductive strengthening for \( F_k \).

After proving correctness of Alg. 2 and 3 we have to drop termination for Alg. 1. The reason is that there might exist infinite ascending or descending chains that we cannot generalize. This is exemplified in the simple program in Fig. 1.

While there exist an inductive strengthening, e.g., \((x \geq 10)\), for every maximal frame index \( k \) our algorithm will block a region \((1, 9 - (k - 1))\) of which there can be infinitely many for unbounded integers, such that there will never exist an \( i \) for which \( F_{(i,1)} = F_{(i+1,1)} \). Note that in the following, as all statements over \( F_{(i,l)} \) always concern all non-error locations, we will slightly abuse notation and use \( F_i \) in place of \( F_{(i,l)} \) where \( l \in L \setminus \{l_E\} \).

**Lemma 4:** In case function \( \text{PROVE} \) terminates, it returns \( \text{true} \) iff there exists an inductive strengthening \( F \) for \( P \), s.t. \( F \land P \) is inductive.

**Proof 4:** Assume \( \text{PROVE} \) terminates with \( \text{true} \), then every call of \( \text{STRENGTHEN} \) for every \( j < k \) must have returned \( \text{true} \) and there must exist a frame with index \( i < k \), s.t. \( F_i = F_i + 1 \), i.e. the frame \( F_i \) is inductive, because \( F_i \land T = F_i' \). Therefore there cannot exist a counterexample path of length \( k \) (more precise of length \( i \)) or less and there cannot exist one of length greater than \( k \), because \( F_i \) is inductive.
The function call STRENGTHEN(2) enters BACKWARD-BLOCK with the initial obligation $(2, 1, x \neq y)$. For predecessor $l_0$, we check $\text{sat}(F_{(1,l_0)} \land y' = x' \land x' \neq y')$, which is again unsatisfiable, as well as $\text{sat}(F_{(1,1)} \land x = y \land x' = x + 1 \land y' = y + 1 \land x' \neq y')$ for location 1, which is not satisfiable either. We can therefore block $x \neq y$ in frame $F_{(2,1)}$, too.

Again we have an empty $Q$ and no more CTI at level 2. We check for termination and find $F_{(1,l_0)} = F_{(2,l_0)}$ as well as $F_{(1)} = F_{(2,1)}$, which means that we found an inductive strengthening $F = (1, x = y)$ s.t. $l_E$ is not reachable in the CFA.

V. Benchmarks

In this section we start with details of our implementation, followed by an evaluation and end with a discussion of the presented results.

Implementation

We implemented our IC3CFA algorithm on top of an existing proprietary model checking framework. A flow chart of the framework is shown in Fig. 3. The framework makes use of the LLVM project enabling parsing a wide range of input languages and translates them into the LLVM intermediate representation (IR). To close the gap between compiler oriented semantics of C as assumed by LLVM and verification semantics that was encountered in the development of the UFO model checker we use the approach of initializing variables with a call to an external function as presented in [12]. We translate this IR into our own intermediate verification language (IVL) that is more suitable for verification, e.g. no SSA form, no three-address code. On the IVL code we execute some optimization stages that are also widely used in other model checkers, e.g. the Kratos software model checker [13], like program slicing, expression propagation and bisimulation minimization, as well as Steensgaard’s pointer analysis [14] to rewrite simple pointer expressions. The bit-precise memory model (similar to the one of CBMC [15]) supports limited pointer operations, including array-element and record-field addressing. After reaching a fixpoint of these optimizations, we construct the program’s control flow graph, labeled with instructions of a guarded command language similar to that of [16]. This allows for efficient construction of weakest preconditions [16, 17]. The results of our preprocessing are in line with the CFAs produced by Kratos and only differ by one or two locations. While the presented approach is fully theory unaware and can be used for infinite-domain theories such as linear real arithmetic (LRA) we use the finite-domain theory of bit vectors. Our tool supports the Z3 and MathSAT SMT solvers; all following benchmarks were executed using the Z3 solver. To minimize overhead and reduce unnecessary pushing attempts we implemented the efficient pushing strategy of [18]. For storing frames efficiently we implemented the delta encoding approach of [9] which, in our experiments, reduced memory consumption as well as runtime significantly. For some benchmarks the reduction

<table>
<thead>
<tr>
<th>i</th>
<th>l:</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>l_0</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>false</td>
<td>$x = y$</td>
<td>$x = y$</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 2. Example program and resulting frames

Now assume that PROVE returns false; Then there must exist a $k$, s.t. for no $i < k$, $F_i$ is inductive and STRENGTHEN for $k$ returns false, i.e. there exists a path of length $k$ from the initial to the error state.

**Theorem 1:** If the algorithm terminates, it returns true iff $P$ is an invariant on $S$.

**Proof 5:** By Lem. [24] the theorem holds. □
in runtime was almost 20 times. Generalization of computed preimages is not implemented, yet.

**Evaluation**

For evaluation of our algorithm and to compare it to others we used a subset\(^1\) of 28 programs from the set of benchmarks used in [2], originating from different domains such as device drivers, communication protocols, SystemC designs and textbook algorithms, some of which are contained in the set of benchmarks of the software verification competition [http://sv-comp.sosy-lab.org/2015/]. One out of four programs contains a bug.

All presented results can be reproduced by our tool via the web-interface at [http://www-i2.informatik.rwth-aachen.de/mctools/vplc/fmcad15/](http://www-i2.informatik.rwth-aachen.de/mctools/vplc/fmcad15/).

All experiments have been executed on a cluster using a single core per instance, running at 2.1 GHz with a memory limit of 4GB per file and a timeout of 1200 seconds.

We briefly compare our implementation to the IC3SMT, Tree-IC3 and Tree-IC3-ITP algorithms of [2]. Scatter plots in Fig. 4 give a graphical idea of how IC3CFA compares time-wise against each of these algorithms while the appended table gives statistics of the overall performance of the four algorithms. The second column highlights the number of solved instances of the 28 benchmark programs. The third column shows the time in seconds for solved instances only, while the last column also takes timeouts into account.

IC3CFA, the implementation of the straight-forward lifting of IC3 to SMT as described in the beginning of Sec. [LV] is clearly the one that performs worst from all four. This comparison is in line with the observations from [2] and shows that IC3CFA benefits from using the explicit representation of control flow over the implicit representation using a dedicated pc variable. IC3CFA is able to solve nine more instances than IC3SMT and does so in almost 9% of the time.

The comparison of IC3CFA with the Tree-IC3 implementation shows the effect of constructing frames for control flow locations, rather than unrolling the ART, as both algorithms are based on a very similar representation of control flow automata and both use weakest preconditions to construct the preimage of a state. IC3CFA is able to solve one more instance than Tree-IC3 in a third of the time.

Besides these two direct comparisons we also evaluated our implementation against the Tree-IC3-ITP implementation that almost resembles Tree-IC3 with the only difference in the computation of preimages, where Tree-IC3-ITP uses interpolants. This comparison shows the advantage of using interpolants over weakest preconditions, as Tree-IC3-ITP is able to solve six more instances than IC3CFA.

\(^1\)Currently we do not support assignment of nondeterministic values inside a loop, due to limitations in our current memory model.
evaluating the performance of IC3 based software model checking algorithms. Another important factor that has to be considered when using IC3SMT approach, while easy to lift, is very inefficient and is not competitive to control flow based techniques. Second, while the TreeIC3 approach is, in very rare cases, slightly more efficient on very small examples, the absence of overhead in ART unrolling makes our approach much faster on medium to large scale examples. Third, the effect of how predecessors are constructed, i.e. weakest preconditions against interpolants, is another important factor that has to be considered when evaluating the performance of IC3 based software model checking algorithms.

VI. CONCLUSION

In this paper we have presented an approach to lift the application domain of the inductive, incremental verification algorithm IC3 from hardware to software model checking. In resemblance to other adaptations of IC3 to software verification, it supports the handling of infinite-state systems by generalising SAT to SMT solving. Its distinguishing feature, however, is the explicit consideration of a program’s control flow by maintaining location-specific frames for storing approximate reachability information. In comparison to more implicit approaches such as Tree-IC3, which is based on an unrolling of the ART, this allows to apply a stronger form of relative inductiveness than obtained in comparable algorithms. The gain in efficiency is demonstrated by experimental evaluations against different implementations of existing, control flow based IC3 algorithms. Due to the novelty of IC3 there is a wide range of future research on this field. One particularly interesting perspective that we intend to investigate are the potentialities of using generalization as a way to overapproximate the exact preimages that we use up to now.

REFERENCES