Compositional Safety Verification with Max-SMT

Marc Brockschmidt [*], Daniel Larraz [†], Albert Oliveras [†], Enric Rodríguez-Carbonell [†] and Albert Rubio [†]
[*]Microsoft Research, Cambridge
[†]Universitat Politècnica de Catalunya

Abstract—**We present an automated compositional program verification technique for safety properties based on *conditional inductive invariants*. For a given program part (e.g., a single loop) and a postcondition φ, we show how to, using a Max-SMT solver, an inductive invariant together with a precondition can be synthesized so that the precondition ensures the validity of the invariant and that the invariant implies φ. From this, we build a bottom-up program verification framework that propagates preconditions of small program parts as postconditions for preceding program parts. The method recovers from failures to prove the validity of a precondition, using the obtained intermediate results to restrict the search space for further proof attempts.**

As only small program parts need to be handled at a time, our method is scalable and distributable. The derived conditions can be viewed as implicit contracts between different parts of the program, and thus enable an incremental program analysis.

I. Introduction

To have impact on everyday software development, a verification engine needs to be able to process the millions of lines of code often encountered in mature software projects. At the same time, the analysis should be repeated every time developers commit a change, and should report feedback in the course of minutes, so that fixes can be applied promptly. Consequently, a central theme in recent research on automated program verification has been *scalability*. As a natural solution to this problem, *compositional* program analyses [1]–[3] have been proposed. They analyze program parts (semi-)independently and then combine the results to obtain a whole-program proof.

For this, a compositional analysis has to predict likely intermediate assertions that allow us to break whole-program reasoning into many instances of local reasoning. This strategy simplifies the individual reasoning steps and allows distributing the analysis [4]. The disadvantage of compositional analyses has traditionally been their precision: local analyses must blindly choose the intermediate assertions. While in some domains (*e.g.* heap) some heuristics have been found [2], effective strategies for guessing and/or refining useful intermediate assertions or summaries in arithmetic domains remains an open problem.

In this paper we introduce a new method for predicting and refining intermediate arithmetic assertions for compositional reasoning about sequential programs. A key component in our approach is Max-SMT solving. Max-SMT solvers can deal with hard and soft constraints, where hard constraints are mandatory, and soft constraints are those that we would like to hold, but are not required to. Hard constraints express what is needed for the soundness of our analysis, while soft ones favor the solutions that are more useful for our technique. More precisely, we use Max-SMT to iteratively infer *conditional inductive invariants*[1], which prove the validity of a property, given that a precondition holds. Hence, if the precondition holds, the program is proved safe. Otherwise, thanks to a novel program transformation technique we call *narrowing*[2], we exploit the failing conditional invariants to focus on what is missing in the safety proof of the program. Then new conditional invariants are sought, and the process is repeated until the safety proof is finally completed. Based on this, we introduce a new bottom-up program analysis procedure that infers conditional invariants in a goal-directed manner, starting from a property that we wish to prove for the program. Our approach makes distributing analysis tasks as simple as in other bottom-up analyses, but also enjoys the precision of CEGAR-based provers.

This work has been supported by Spanish MINECO under the grant TIN2013-45732-C4-3-P (project DAMAS) and the FPI grant BES-2011-044621 (Larraz).

II. Illustration of the Method

In this section, we illustrate the core concepts of our approach by using some small examples. We will give the formal definition of the used methods in Sect. IV.

We handle programs by considering one strongly connected component (SCC) $\mathcal{C}$ of the control-flow graph at a time, together with the sequential parts of the program leading to $\mathcal{C}$, either from initial states or other SCCs.

Instead of program invariants, for each SCC we synthesize *conditional inductive invariants*. These are inductive properties such that they may not always hold whenever the SCC is reached, but once they hold, then they are always satisfied.

a) Conditional Inductive Invariants: As an example, consider the program snippet in Fig. 1, where we do not assume any knowledge about the rest of the program. To prove the assertion, we need an inductive property $\mathcal{Q}$ for the loop such that $\mathcal{Q}$ together with the negation of the loop condition $\mathsf{i} > 0$ implies the assertion. Using our constraint-solving based method CondSafe (cf. Sect. IV-A), we find $\mathcal{Q}_1 = \mathsf{x} + 5 \cdot \mathsf{i} \geq 0$. The property $\mathcal{Q}_1$ can be seen as a *precondition* at the loop entry for the validity of the assertion.

```
while i > 0 do
  x := x + 5;
  i := i − 1;
done
assert(x ≥ 0);
```

Fig. 1.

[1]This concept was previously introduced with the name “quasi-invariant” in [5] in the different context of proving program non-termination.

[2]This narrowing is inspired by the narrowing in term rewrite systems, and is unrelated to the notion with the same name used in abstract interpretation.

b) Combining Conditional Inductive Invariants: Once we have found a conditional inductive invariant for an SCC, we use the generated preconditions as postconditions for its preceding SCCs in the program.

```
while j > 0 do
  j := j − 1;
  i := i + 1;
done
```

Fig. 2.

As an example, assume that the loop from Fig. 1 is directly preceded by the loop in Fig. 2. We now use the precondition $\mathcal{Q}_1$ we obtained earlier as input to our conditional invariant synthesis method, similarly to the assertion in Fig. 1. Thus, we now look for an inductive property $\mathcal{Q}_2$ that, together with $\neg(\mathsf{j} > 0)$, implies $\mathcal{Q}_1$. In this case we obtain the conditional invariant $\mathcal{Q}_2 = \mathsf{j} \geq 0 \wedge \mathsf{x} + 5 \cdot (\mathsf{i} + \mathsf{j}) \geq 0$ for the loop. As with $\mathcal{Q}_1$, now we can see $\mathcal{Q}_2$ as a precondition at the loop entry, and propagate $\mathcal{Q}_2$ up to the preceding SCCs in the program.

c) Recovering from Failures: When we cannot prove that a precondition always holds, we try to recover and find an alternative precondition. In this process, we make use of the results obtained so far, and *narrow* the program using our intermediate results. As an example, consider the loop in Fig. 3.

```
while unknown() do
  assert(x ≠ y);
  x := x + 1;
  y := y + 1;
done
```

Fig. 3.

We again apply our method CondSafe to find a conditional invariant for this loop which, together with the loop condition, implies the assertion in the loop body. As it can only synthesize conjunctions of linear inequalities, it produces the conditional invariant $\mathcal{Q}_3 = x > y$ for the loop. However, assume that the precondition $\mathcal{Q}_3$ could not be proven to always hold in the context of our example. In that case, we use the obtained information to narrow the program and look for another precondition.

```
if ¬(x>y) then
  while ¬(x>y) do
    assert(x ≠ y);
    x := x + 1;
    y := y + 1;
  done
fi
```

Fig. 4.

Intuitively, our program narrowing reflects that states represented by the conditional invariant found earlier are already proven to be safe. Hence, we only need to consider states for which the negation of the conditional invariant holds, i.e., we can add its negation as an assumption to the program. In our example, this yields the modified version of Fig. 3 displayed in Fig. 4. Another call to CondSafe then yields the conditional invariant $\mathcal{Q}'_3 = x < y$ for the loop. This means that we can ensure the validity of the assertion if before the conditional statement we satisfy that $\neg(x > y) \Rightarrow x < y$, or equivalently, $x \neq y$. In general, this narrowing allows us to find (some) disjunctive invariants.

III. Preliminaries

1) SAT, Max-SAT, and Max-SMT: Let $\mathcal{P}$ be a fixed set of *propositional variables*. For $p \in \mathcal{P}$, p and $\neg p$ are *literals*. A *clause* is a disjunction of literals $l_1 \vee \cdots \vee l_n$. A (CNF) *propositional formula* is a conjunction of clauses $C_1 \wedge \cdots \wedge C_m$. The problem of *propositional satisfiability* (*SAT*) is to determine whether a propositional formula is *satisfiable*. An extension of SAT is *satisfiability modulo theories (SMT)* [6], where satisfiability of a formula with literals from a given background theory is checked. We will use the theory of *quantifier-free integer (non-)linear arithmetic*, where literals are inequalities of linear (resp. polynomial) arithmetic expressions.

Another extension of SAT is *Max-SAT* [6], which generalizes SAT to finding an assignment such that the number of satisfied clauses in a given formula F is maximized. Finally, *Max-SMT* combines Max-SAT and SMT. A *(weighted partial) Max-SMT* problem is a formula of the form $H_1 \wedge \ldots \wedge H_n \wedge [S_1, \omega_1] \wedge \ldots \wedge [S_m, \omega_m]$, where the hard clauses H_i and the soft clauses S_j (with weight ω_j) are disjunctions of literals over a background theory, and the aim is to find a model of the hard clauses that maximizes the sum of the weights of the satisfied soft clauses.

2) Programs and States: We make heavy use of the program structure and hence represent programs as graphs. For this, we fix a set of (integer) program *variables* $\mathcal{V} = \{v_1, \ldots, v_n\}$ and denote by $\mathcal{F}(\mathcal{V})$ the formulas consisting of conjunctions of linear inequalities over the variables $\mathcal{V}$. Let $\mathcal{L}$ be the set of program *locations*, which contains a *canonical start location* ℓ_0. Program *transitions* $\mathcal{T}$ are tuples (ℓ, τ, ℓ'), where ℓ and $\ell' \in \mathcal{L}$ represent the pre- and post-location respectively, and $\tau \in \mathcal{F}(\mathcal{V} \cup \mathcal{V}')$ describes its transition relation. Here $\mathcal{V}' = \{v'_1, \ldots, v'_n\}$ are the *post-variables*, i.e., the values of the variables after the transition.[3] A transition is *initial* if its source location is ℓ_0. A *program* is a set of transitions. We view a program $\mathcal{P} = (\mathcal{L}, \mathcal{T})$ as a directed graph (the *control-flow graph*, CFG), in which edges are the transitions $\mathcal{T}$ and nodes are the locations $\mathcal{L}$.[4]

A *state* $s = (\ell, \boldsymbol{v})$ consists of a location $\ell \in \mathcal{L}$ and a *valuation* $\boldsymbol{v} : \mathcal{V} \to \mathbb{Z}$. A state $(\ell, \boldsymbol{v})$ is *initial* if $\ell = \ell_0$. We denote an *evaluation step* with transition $t = (\ell, \tau, \ell')$ by $(\ell, \boldsymbol{v}) \to_t (\ell', \boldsymbol{v'})$, where the valuations $\boldsymbol{v}$, $\boldsymbol{v'}$ satisfy the formula τ of t. We use $\to_{\mathcal{P}}$ if we do not care about the executed transition, and $\to^*_{\mathcal{P}}$ to denote the transitive-reflexive closure of $\to_{\mathcal{P}}$. We say that a state s is *reachable* if there exists an initial state s_0 such that $s_0 \to^*_{\mathcal{P}} s$.

3) Safety and Invariants: An *assertion* (t, φ) is a pair of a transition $t \in \mathcal{T}$ and a formula $\varphi \in \mathcal{F}(\mathcal{V})$. A program $\mathcal{P}$ is *safe* for the assertion (t, φ) if for every evaluation $(\ell_0, \boldsymbol{v}_0) \to^*_{\mathcal{P}} \circ \to_t (\ell, \boldsymbol{v})$, we have that $\boldsymbol{v} \models \varphi$ holds.[5] Note that proving that a formula φ always holds at a location ℓ can be handled in this setting by adding an extra location ℓ^* and an extra transition $t^* = (\ell, \mathrm{true}, \ell^*)$ and checking safety for (t^*, φ).

We call a map $\mathcal{I} : \mathcal{L} \to \mathcal{F}(\mathcal{V})$ a *program invariant* (or often just *invariant*) if for all reachable states $(\ell, \boldsymbol{v})$, we have $\boldsymbol{v} \models \mathcal{I}(\ell)$ holds. An important class of program invariants are inductive invariants. An invariant $\mathcal{I}$ is *inductive* if the following conditions hold:

Initiation:		$\top$	$\models \mathcal{I}(\ell_0)$
Consecution:	For $(\ell, \tau, \ell') \in \mathcal{P}$:	$\mathcal{I}(\ell) \wedge \tau$	$\models \mathcal{I}(\ell')'$

[3]For $\varphi \in \mathcal{F}(\mathcal{V})$, $\varphi' \in \mathcal{F}(\mathcal{V}')$ is the version of φ using primed variables.

[4]Since we label transitions only with conjunctions of linear inequalities, disjunctive conditions are represented using several transitions with the same pre- and post-location. Thus, $\mathcal{P}$ is actually a multigraph.

[5]Here, $\to^*_{\mathcal{P}} \circ \to_t$ denotes arbitrary program evaluations that end with an evaluation step using t.

4) Constraint Solving for Verification: Inductive invariants can be generated using a *constraint-based* approach [7], [8]. The idea is to consider templates for candidate invariant properties, such as (conjunctions of) linear inequalities. These templates contain both the program variables $\mathcal{V}$ as well as template variables $\mathcal{V}_T$, whose values have to be determined to ensure the required properties. To this end, the conditions on inductive invariants are expressed by means of *constraints* of the form $\exists\, \mathcal{V}_T\, .\forall\, \mathcal{V} \ldots$.. Any solution to these constraints then yields an invariant. In the case of linear arithmetic, Farkas' Lemma [9] is often used to handle the quantifier alternation in the generated constraints. Intuitively, it allows one to transform $\exists\forall$ problems encountered in invariant synthesis into $\exists$ problems. In the general case, an SMT problem over non-linear arithmetic is obtained, for which effective SMT solvers exist [10], [11]. By assigning weights to the different conditions, invariant generation can be cast as an optimization problem in the Max-SMT framework [5], [12].

IV. Proving Safety

Most automated techniques for proving program safety iteratively construct *inductive program invariants* as over-approximations of the reachable state space. Starting from the known set of initial states, a process to discover more reachable states and refine the approximation is iterated, until it finally reaches a fixed point (i.e., the invariant is inductive) and is strong enough to imply program safety. However, this requires taking the whole program into account, which is sometimes infeasible or undesirable in practice.

In contrast to this, our method starts with the known unsafe states, and iteratively constructs an under-approximation of the set of safe states, with the goal of showing that all initial states are contained in that set. For this, we introduce the notion of *conditional safety*. Intuitively, when proving that a program is $(\tilde{t},\tilde{\varphi})$-*conditionally safe for the assertion* (t,φ) we consider evaluations starting after a $\rightarrow_{\tilde{t}} (\tilde{\ell},\tilde{\boldsymbol{v}})$ step, where $\tilde{\boldsymbol{v}}$ satisfies $\tilde{\varphi}$, instead of evaluations starting at an initial state. In particular, a program that is $(t_0,\top)$-conditionally safe for (t,φ) for all initial transitions t_0 is (unconditionally) safe for (t,φ).

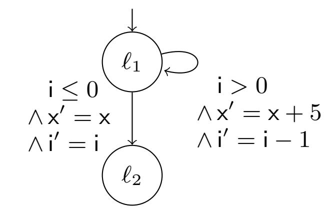
Definition 1 (Conditional safety)**.** *Let $\mathcal{P}$ be a program, $t,\tilde{t}$ transitions and $\varphi,\tilde{\varphi} \in \mathcal{F}(\mathcal{V})$. The program $\mathcal{P}$ is $(\tilde{t},\tilde{\varphi})$-*conditionally safe *for the assertion (t,φ) if for any evaluation that contains $\rightarrow_{\tilde{t}} (\tilde{\ell},\tilde{\boldsymbol{v}}) \rightarrow^*_{\mathcal{P}} (\bar{\ell},\bar{\boldsymbol{v}}) \rightarrow_t (\ell,\boldsymbol{v})$, we have $\tilde{\boldsymbol{v}} \models \tilde{\varphi}$ implies that $\boldsymbol{v} \models \varphi$. In that case we say that the assertion $(\tilde{t},\tilde{\varphi})$ is a* precondition *for the* postcondition (t,φ).

Conditional safety is "transitive" in the sense that if a set of transitions $\mathcal{E} = \{\widetilde{t_1},\ldots,\widetilde{t_m}\}$ dominates t,[6] and for all $i = 1,\ldots,m$ we have $\mathcal{P}$ is $(\widetilde{t_i},\widetilde{\varphi_i})$-conditionally safe for (t,φ) and $\mathcal{P}$ is safe for $(\widetilde{t_i},\widetilde{\varphi_i})$, then $\mathcal{P}$ is also safe for (t,φ). In what follows we exploit this observation to prove program safety by means of conditional safety.

A *program component* $\mathcal{C}$ of a program $\mathcal{P}$ is an SCC of the control-flow graph, and its *entry transitions* (or *entries*)

[6]We say a set of transitions $\mathcal{E}$ dominates transition t if every path in the CFG from ℓ_0 that contains t must also contain some $\tilde{t} \in \mathcal{E}$.

```
...
while i > 0 do
  x := x + 5;
  i := i − 1;
done
assert(x ≥ 50)
```



ℓ_1
ℓ_2
$\mathsf{i} \le 0 \wedge \mathsf{x}' = \mathsf{x} \wedge \mathsf{i}' = \mathsf{i}$
$\mathsf{i} > 0 \wedge \mathsf{x}' = \mathsf{x}+5 \wedge \mathsf{i}' = \mathsf{i}-1$

Fig. 5. Source code of program snippet and its CFG.

$\mathcal{E}_\mathcal{C}$ are those transitions $t = (\ell,\tau,\ell')$ such that $t \notin \mathcal{C}$ but ℓ' appears in $\mathcal{C}$. By considering each component as a single node, we can obtain from $\mathcal{P}$ a DAG of SCCs, whose edges are the entry transitions. Our technique analyzes components independently, and communicates the results of these analyses to other components along entry transitions.

Given a component $\mathcal{C}$ and an assertion (t,φ) such that $t \notin \mathcal{C}$ but the source node of t appears in $\mathcal{C}$, we call t an *exit transition* of $\mathcal{C}$. For such exit transitions, we compute a *sufficient* condition $\psi_{\tilde{t}}$ for each entry transition $\tilde{t} \in \mathcal{E}_\mathcal{C}$ such that $\mathcal{C} \cup \{t\}$ is $(\tilde{t},\psi_{\tilde{t}})$-conditionally safe for (t,φ). Then we continue reasoning backwards following the DAG and try to prove that $\mathcal{P}$ is safe for each $(\tilde{t},\psi_{\tilde{t}})$. If we succeed, following the argument above we will have proved $\mathcal{P}$ safe for (t,φ).

In the following, we first discuss how to prove conditional safety of single program components in Sect. IV-A, and then present the algorithm that combines these local analyses to construct a global safety proof in Sect. IV-B.

A. Synthesizing Local Conditions

Here we restrict ourselves to a program component $\mathcal{C}$ and its entry transitions $\mathcal{E}_\mathcal{C}$, and assume we are given an assertion $(t_{\text{exit}},\varphi)$, where $t_{\text{exit}} = (\tilde{\ell}_{\text{exit}},\tau_{\text{exit}},\ell_{\text{exit}})$ is an exit transition of $\mathcal{C}$ (i.e., $t_{\text{exit}} \notin \mathcal{C}$ and $\tilde{\ell}_{\text{exit}}$ appears in $\mathcal{C}$). We show how a precondition (t,ψ) for $(t_{\text{exit}},\varphi)$ can be obtained for each $t \in \mathcal{E}_\mathcal{C}$. Here we only consider the case of φ being a single clause (i.e., a disjunction of literals); if φ is in CNF, each conjunct is handled separately. The preconditions on the entry transitions will be determined by a *conditional inductive invariant*, which like a standard invariant is inductive, but not necessarily initiated in all program runs. Indeed, this initiation condition is what we will extract as precondition and propagate backwards to preceding program components in the DAG.

Definition 2 (Conditional Inductive Invariant)**.** *We say a map $\mathcal{Q} : \mathcal{L} \rightarrow \mathcal{F}(\mathcal{V})$ is a* conditional (inductive) invariant *for a program (component) $\mathcal{P}$ if for all $(\ell,\boldsymbol{v}) \rightarrow_\mathcal{P} (\ell',\boldsymbol{v}')$, we have $\boldsymbol{v} \models \mathcal{Q}(\ell)$ implies $\boldsymbol{v}' \models \mathcal{Q}(\ell')$.*

Conditional invariants are convenient tools to express conditions for safety proving, allowing reasoning in the style of "if the condition for $\mathcal{Q}$ holds, then the assertion (t,φ) holds".

Example 1. *Consider the program snippet in Fig. 5. A conditional inductive invariant supporting safety of this program part is $\mathcal{Q}_5(\ell_1) \equiv \mathsf{x} + 5 \cdot \mathsf{i} \ge 50$, $\mathcal{Q}_5(\ell_2) \equiv \mathsf{x} \ge 50$. In fact, any conditional invariant $\mathcal{Q}_m(\ell_1) \equiv \mathsf{x}+m\cdot\mathsf{i} \ge 50$ with $0 \le m \le 5$*

would be a conditional inductive invariant that, together with the negation of the loop condition $\mathsf{i} \le 0$*, implies* $\mathsf{x} \ge 50$.

We use a Max-SMT-based constraint-solving approach to generate conditional inductive invariants. Unlike in [5], to use information about the initialization of variables before a program component, we take into account the entry transitions $\mathcal{E}_{\mathcal{C}}$. The precondition for each entry transition is the conditional invariant that has been synthesized at its target location.

To find conditional invariants, we construct a constraint system. For each location ℓ in $\mathcal{C}$ we create a template $I_{\ell,k}(\mathcal{V}) \equiv \wedge_{1\le j\le k}\, I_{\ell,j,k}(\mathcal{V})$ which is a conjunction of k linear inequations[7] of the form $I_{\ell,j,k}(\mathcal{V}) \equiv i_{\ell,j} + \sum_{v\in\mathcal{V}} i_{\ell,j,v}\cdot v \le 0$, where the $i_{\ell,j}$, $i_{\ell,j,v}$ are fresh variables from the set of template variables $\mathcal{V}_T$. We then transform the conditions for a conditional invariant proving safety for the assertion $(t_{\text{exit}}, \varphi)$ to the constraints in Fig. 6. Here, e.g., $I'_{\ell',k}$ refers to the variant of $I_{\ell',k}$ using primed versions of the program variables $\mathcal{V}$, but unprimed template variables $\mathcal{V}_T$.

In the overall constraint system, we mark the **Consecution** and **Safety** constraints as hard requirements. Thus, any solution to these constraints is a conditional *inductive* invariant *implying our assertion*. However, as we mark the **Initiation** constraints as soft, the found conditional invariants may depend on preconditions not implied by the direct context of the considered component. On the other hand, the Max-SMT solver prefers solutions that require fewer preconditions. Overall, we create the following Max-SMT formula

$$\mathbb{F}_k \overset{def}{=} \bigwedge_{t\in\mathcal{C}} \mathbb{C}_{t,k} \wedge \bigwedge_{t\in\mathcal{E}_{\mathcal{C}},1\le j\le k} (\mathbb{I}_{t,j,k} \vee \neg p_{\mathbb{I}_{t,j,k}}) \wedge \mathbb{S}_k \wedge \bigwedge_{t\in\mathcal{E}_{\mathcal{C}},1\le j\le k} [p_{\mathbb{I}_{t,j,k}}, \omega_{\mathbb{I}}],$$

where the $p_{\mathbb{I}_{t,j,k}}$ are propositional variables which are true if the **Initiation** condition $\mathbb{I}_{t,j,k}$ is satisfied, and $\omega_{\mathbb{I}}$ is the corresponding weight. [8] We use $\mathbb{F}_k$ in our procedure CondSafe in Algo. 1.

Algorithm 1 Proc. CondSafe computing conditional invariant

```
Input: component $\mathcal{C}$, entry transitions $\mathcal{E}_{\mathcal{C}}$, assertion $(t_{\text{exit}}, \varphi)$
       s.t. $t_{\text{exit}}$ is an exit transition of $\mathcal{C}$ and $\varphi$ is a clause
Output: None | $\mathcal{Q}$, where $\mathcal{Q}$ maps locations in $\mathcal{C}$ to conjunc-
       tions of inequations
1: $k \leftarrow 1$
2: repeat
3:     construct formula $\mathbb{F}_k$ from $\mathcal{C}$, $\mathcal{E}_{\mathcal{C}}$ and $(t_{\text{exit}}, \varphi)$
4:     $\sigma \leftarrow$ Max-SMT-solver($\mathbb{F}_k$)
5:     if $\sigma$ is a model then
6:         $\mathcal{Q} \leftarrow \{\ell \mapsto \sigma(I_{\ell,k}) \mid \ell$ in $\mathcal{C}\}$ return $\mathcal{Q}$
7:     $k \leftarrow k + 1$
8: until $k >$ MAX_CONJUNCTS  return None
```

In CondSafe, we iteratively try "larger" templates of more conjuncts of linear inequations until we either give up (in our implementation, MAX_CONJUNCTS is 3) or finally find a conditional invariant. Note, however, that here we are only trying to prove safety for *one* clause at a time, which reduces the number of required conjuncts as compared to dealing with a whole CNF in a single step. If the Max-SMT solver is able to find a model for $\mathbb{F}_k$, then we instantiate our invariant templates $I_{\ell,k}$ with the values found for the template variables in the model σ, obtaining a conditional invariant $\mathcal{Q}$. When we obtain a result, for every entry transition $t = (\ell, \tau, \ell') \in \mathcal{E}_{\mathcal{C}}$ the conditional invariant $\mathcal{Q}(\ell')$ is a precondition that implies safety for the assertion $(t_{\text{exit}}, \varphi)$. The following theorem states the correctness of this procedure.

Theorem 1. *Let $\mathcal{C}$ be a component, $\mathcal{E}_{\mathcal{C}}$ its entry transitions, and $(t_{\text{exit}}, \varphi)$ an assertion with t_{exit} an exit transition of $\mathcal{C}$ and φ a clause. If the procedure call* CondSafe$(\mathcal{C}, \mathcal{E}_{\mathcal{C}}, (t_{\text{exit}}, \varphi))$ *returns $\mathcal{Q} \ne$ None, then $\mathcal{Q}$ is a conditional inductive invariant for $\mathcal{C}$ and $\mathcal{P}$ is $(t, \mathcal{Q}(\ell'))$-conditionally safe for $(t_{\text{exit}}, \varphi)$ for all $t = (\ell, \tau, \ell') \in \mathcal{E}_{\mathcal{C}}$.*

Proof. All proofs can be found in our technical report [13]. □

B. Propagating Local Conditions

In this section, we explain how to use the local procedure CondSafe to prove safety of a full program. To this end we now consider the full DAG of program components. As outlined above, the idea is to start from the assertion provided by the user, call the procedure CondSafe to obtain preconditions for the entry transitions of the corresponding component, and then use these preconditions as assertions for preceding components, continuing recursively. If eventually for each initial transition the transition relation implies the corresponding preconditions, then safety has been proven. If we fail to prove safety for certain assertions, we backtrack, trying further possible preconditions and conditional invariants.

The key to the precision of our approach is our treatment of failed proof attempts. When the procedure CondSafe finds a conditional invariant $\mathcal{Q}$ for $\mathcal{C}$, but proving $(t, \mathcal{Q}(\ell'))$ as a postcondition of the preceding component fails for some $t = (\ell, \tau, \ell') \in \mathcal{E}_{\mathcal{C}}$, we use $\mathcal{Q}$ to *narrow* our program representation and filter out evaluations that are already known to be safe.

As outlined above, in our proof process we treat each clause of the conjunction $\mathcal{Q}(\ell')$ separately, and pass each one as its own assertion to preceding program components, allowing for a fine-grained *program-narrowing* technique. By construction of $\mathcal{Q}$, evaluations that satisfy all literals of $\mathcal{Q}(\ell')$ after executing $t = (\ell, \tau, \ell') \in \mathcal{E}_{\mathcal{C}}$ are safe. Thus, among the evaluations that use t, we only need to consider those where at least one literal in $\mathcal{Q}(\ell')$ does not hold. Hence, we *narrow* each entry transition by conjoining it with the negation of the conjunction of all literals for which we could not prove safety (see line 13 in Algo. 2). Note that if there is more than one literal in this conjunction, then the negation is a disjunction, which in our program model implies splitting transitions.

We can narrow program components similarly. For a transition $t = (\ell, \tau, \ell') \in \mathcal{C}$, we know that if either $\mathcal{Q}(\ell)$ or $\mathcal{Q}(\ell')'$ holds in an evaluation passing through t, the program

[7]In our overall algorithm, k is initially 1 and increased in case of failures.
[8]Farkas' Lemma is applied *locally* to the subformulas $\mathbb{C}_{t,k}$, $\mathbb{I}_{t,j,k}$ and $\mathbb{S}_k$, and weights are added on the resulting constraints over the template variables.

Initiation:	For $t = (\ell, \tau, \ell') \in \mathcal{E}_\mathcal{C}, 1 \le j \le k$:	$\mathbb{I}_{t,j,k}$	$\stackrel{def}{=}$	τ	$\Rightarrow I'_{\ell',j,k}$
Consecution:	For $t = (\ell, \tau, \ell') \in \mathcal{C}$:	$\mathbb{C}_{t,k}$	$\stackrel{def}{=}$	$I_{\ell,k} \wedge \tau$	$\Rightarrow I'_{\ell',k}$
Safety:	For $t_{\text{exit}} = (\tilde{\ell}_{\text{exit}}, \tau_{\text{exit}}, \ell_{\text{exit}})$:	$\mathbb{S}_k$	$\stackrel{def}{=}$	$I_{\tilde{\ell}_{\text{exit}},k} \wedge \tau_{\text{exit}}$	$\Rightarrow \varphi'$

Fig. 6. Constraints used in $\mathsf{CondSafe}(\mathcal{C}, \mathcal{E}_\mathcal{C}, (t_{\text{exit}}, \varphi))$

Algorithm 2 Procedure CheckSafe for proving a program safe for an assertion

```
Input: Program $\mathcal{P}$, a component $\mathcal{C}$, entries $\mathcal{E}_\mathcal{C}$, assertion
    $(t_{\text{exit}}, \varphi)$ s.t. $t_{\text{exit}}$ is an exit transition of $\mathcal{C}$ and $\varphi$ a clause
Output: Safe | Maybe
 1: let $(\ell_{\text{exit}}, \tau_{\text{exit}}, \ell'_{\text{exit}}) = t_{\text{exit}}$
 2: if $(\tau_{\text{exit}} \Rightarrow \varphi')$ then return Safe
 3: else if $\ell_{\text{exit}} = \ell_0$ then return Maybe
 4: $\mathcal{Q} \leftarrow \mathsf{CondSafe}(\mathcal{C}, \mathcal{E}_\mathcal{C}, (t_{\text{exit}}, \varphi))$
 5: if $\mathcal{Q} = $ None then return Maybe
 6: for all $t = (\ell, \tau, \ell') \in \mathcal{E}_\mathcal{C}$, $L \in \mathcal{Q}(\ell')$ do
 7:     $\tilde{\mathcal{C}} \leftarrow \mathsf{component}(\ell, \mathcal{P})$
 8:     $\mathcal{E}_{\tilde{\mathcal{C}}} \leftarrow \mathsf{entries}(\tilde{\mathcal{C}}, \mathcal{P})$
 9:     $\mathsf{res}[t, L] \leftarrow \mathsf{CheckSafe}(\mathcal{P}, \tilde{\mathcal{C}}, \mathcal{E}_{\tilde{\mathcal{C}}}, (t, L))$
10: if $\forall t = (\ell, \tau, \ell') \in \mathcal{E}_\mathcal{C}, L \in \mathcal{Q}(\ell')$ . $\mathsf{res}[t, L] = $ Safe then
11: return Safe
12: else
13:     $\hat{\mathcal{E}}_\mathcal{C} \leftarrow \{(\ell, \tau \wedge \neg(\bigwedge_{\substack{L \in \mathcal{Q}(\ell') \\ \mathsf{res}[t,L] = \mathsf{Maybe}}} L'), \ell') \mid t = (\ell, \tau, \ell') \in \mathcal{E}_\mathcal{C}\}$
14:     $\hat{\mathcal{C}} \leftarrow \{(\ell, \tau \wedge \neg\mathcal{Q}(\ell')' \wedge \neg\mathcal{Q}(\ell), \ell') \mid (\ell, \tau, \ell') \in \mathcal{C}\}$
15: return $\mathsf{CheckSafe}(\mathcal{P}, \hat{\mathcal{C}}, \hat{\mathcal{E}}_\mathcal{C}, (t_{\text{exit}}, \varphi))$
```

is safe. Thus, we narrow the program by replacing τ by $\tau \wedge \neg\, \mathcal{Q}(\ell) \wedge \neg\, \mathcal{Q}(\ell')'$ (see line 14 in Algo. 2).

This narrowing allows us to generate disjunctive conditional invariants, where each result of CondSafe is one disjunct. Note that not *all* disjunctive invariants can be discovered like this, as each intermediate result needs to be inductive using the disjuncts found so far. However, this is the pattern observed in *phase-change* algorithms [14].

Our overall safety proving procedure CheckSafe is shown in Algo. 2. The helper procedures component and entries are used to find the program component for a given location and the entry transitions for a component. The result of CheckSafe is either Maybe when the proof failed, or Safe if it succeeded. In the latter case, we have managed to create a chain of conditional invariants that imply that $(t_{\text{exit}}, \varphi)$ always holds.

Theorem 2. *Let $\mathcal{P}$ be a program, $\mathcal{C}$ a component and $\mathcal{E}_\mathcal{C}$ its entries. Given an assertion $(t_{\text{exit}}, \varphi)$ such that t_{exit} is an exit transition of $\mathcal{C}$ and φ is a clause, if* $\mathsf{CheckSafe}(\mathcal{P}, \mathcal{C}, \mathcal{E}_\mathcal{C}, (t_{\text{exit}}, \varphi)) =$ *Safe, then $\mathcal{P}$ is safe for $(t_{\text{exit}}, \varphi)$.*

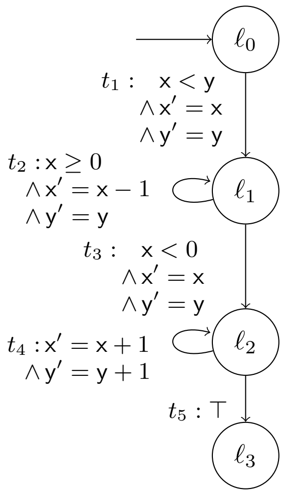
Example 2. *We demonstrate* CheckSafe *on the program displayed on Fig. 7, called $\mathcal{P}$ in the following, which is an extended version of the example from Fig. 3.*

We want to prove the assertion $(t_5, \mathsf{x} \neq \mathsf{y})$. Hence we make a first call $\mathsf{CheckSafe}(\mathcal{P}, \{t_4\}, \{t_3\}, (t_5, \mathsf{x} {\neq} \mathsf{y}))$*: the non-trivial SCC containing ℓ_2 is $\{t_4\}$ and its entry transitions are $\{t_3\}$. Hence, we call* $\mathsf{CondSafe}(\{t_4\}, \{t_3\}, (t_5, \mathsf{x} \neq \mathsf{y}))$ *and the resulting conditional invariant for ℓ_2 is either $\mathsf{x} < \mathsf{y}$ or $\mathsf{y} < \mathsf{x}$. Let us assume it is $\mathsf{y} < \mathsf{x}$. In the next step, we propagate this to the predecessor SCC $\{t_2\}$, and call* $\mathsf{CheckSafe}(\mathcal{P}, \{t_2\}, \{t_1\}, (t_3, \mathsf{y} < \mathsf{x}))$.

In turn, this leads to calling $\mathsf{CondSafe}(\{t_2\}, \{t_1\}, (t_3, \mathsf{y} < \mathsf{x}))$ *to our synthesis subprocedure. No conditional invariant supporting this assertion can be found, and hence None is returned by* CondSafe*, and consequently Maybe is returned by* CheckSafe*. Hence, we return to the original SCC $\{t_4\}$ and its entry $\{t_3\}$, and then by narrowing we obtain two new transitions:*

$$t'_4 = (\ell_2, \mathsf{x}' = \mathsf{x} + 1 \wedge \mathsf{y}' = \mathsf{y} + 1 \wedge \neg(\mathsf{y} < \mathsf{x}), \ell_2),$$
$$t'_3 = (\ell_1, \mathsf{x} < 0 \wedge \mathsf{x}' = \mathsf{x} \wedge \mathsf{y}' = \mathsf{y} \wedge \neg(\mathsf{y} < \mathsf{x}), \ell_2).$$

Using these, we call $\mathsf{CheckSafe}(\mathcal{P}, \{t'_4\}, \{t'_3\}, (t_5, \mathsf{x} {\neq} \mathsf{y}))$*. The next call to* CondSafe *then yields the conditional invariant $\mathsf{x} < \mathsf{y}$ at ℓ_2, which is in turn propagated backwards with the call* $\mathsf{CheckSafe}(\mathcal{P}, \{t_2\}, \{t_1\}, (t'_3, \mathsf{x} < \mathsf{y}))$*. This then yields a conditional invariant $\mathsf{x} < \mathsf{y}$ at ℓ_1, which is finally propagated back in the call* $\mathsf{CheckSafe}(\mathcal{P}, \{\}, \{\}, (t_1, \mathsf{x} < \mathsf{y}))$*, which directly returns Safe.*



ℓ_0
$t_1 : \mathsf{x} < \mathsf{y} \wedge \mathsf{x}' = \mathsf{x} \wedge \mathsf{y}' = \mathsf{y}$
$t_2 : \mathsf{x} \ge 0 \wedge \mathsf{x}' = \mathsf{x} - 1 \wedge \mathsf{y}' = \mathsf{y}$
ℓ_1
$t_3 : \mathsf{x} < 0 \wedge \mathsf{x}' = \mathsf{x} \wedge \mathsf{y}' = \mathsf{y}$
$t_4 : \mathsf{x}' = \mathsf{x} + 1 \wedge \mathsf{y}' = \mathsf{y} + 1$
ℓ_2
$t_5 : \top$
ℓ_3

Fig. 7.

C. Improving Performance

The basic method CheckSafe can be extended in several ways to improve performance. We now present a number of techniques that are useful to reduce the runtime of the algorithm and distribute the required work. Note that none of these techniques influences the precision of the overall framework.

a) Using conditional invariants to disable transitions: When proving an assertion, it is often necessary to find invariants that show the unfeasibility of some transition, which allows disabling it. In our framework, the required invariants can be conditional as well. Therefore, CheckSafe must be called recursively to prove that the conditional invariant is indeed invariant. In our implementation, we generate constraints such that every solution provides conditional invariants either implying the postcondition or disabling some transition. By imposing different weights, we make the Max-SMT solver prefer solutions that imply the postcondition.

b) Handling unsuccessful proof attempts: One important aspect is that the presented algorithm does not *learn* facts about the reachable state space, and so duplicates work when assertions appear several times. To alleviate this for *unsuccessful* recursive invocations of CheckSafe, we introduce a simple memoization technique to avoid repeating such calls. So when $\mathsf{CheckSafe}(\mathcal{P}, \mathcal{C}, \mathcal{E}_{\mathcal{C}}, (t, \varphi)) = \mathsf{Maybe}$, we store this result, and use it for all later calls of $\mathsf{CheckSafe}(\mathcal{P}, \mathcal{C}, \mathcal{E}_{\mathcal{C}}, (t, \varphi))$. This strategy is valid as the return value Maybe indicates that our method cannot prove the assertion (t, φ) at all, meaning that later proof attempts will fail as well. In our implementation, this memoization of unsuccessful attempts is local to the initial call to CheckSafe. The rationale is that, when proving unrelated properties, it is likely that few calls are shared and that the book-keeping does not pay off.

c) Handling successful proof attempts: When a recursive call yields a *successful* result, we can *strengthen* the program with the proven invariant. Remember that $\mathsf{CheckSafe}(\mathcal{P}, \mathcal{C}, \mathcal{E}_{\mathcal{C}}, (t, \varphi)) = \mathsf{Safe}$ means that whenever the transition t is used in *any* evaluation, φ holds in the succeeding state. Thus, we can add this knowledge explicitly and change the transition in the original program. In practice, this strengthening is applied only if the first call to CheckSafe was successful, i.e, no narrowing was applied. The reason is that, if the transition relation of t was obtained through repeated narrowing, in general one needs to split transitions, and it is not correct to just add φ' to t. Namely, assume that $t_o = (\ell, \tau_o, \ell')$ is the original (unnarrowed) version of a transition $t = (\ell, \tau, \ell') \in \mathcal{E}_{\mathcal{C}}$. As t is an entry transition of $\mathcal{C}$, we have $\tau = \tau_o \wedge \neg\psi'_1 \wedge \ldots \wedge \neg\psi'_m$ by construction, where ψ_i is the additional constraint we added in the i-th narrowing of component entries. Thus what we proved is that $\psi'_1 \vee \ldots \vee \psi'_m \vee \varphi'$ always holds after using transition t_o. So we should replace t_o in the program with a transition labeled with $\tau_o \wedge (\psi'_1 \vee \ldots \vee \psi'_m \vee \varphi')$. As we cannot handle disjunctions natively, this implies replacing t_o by $m + 1$ new transitions.

Note that this program modification approach, unlike memoization, makes the gained information available to the Max-SMT solver when searching for a conditional invariant. A similar strategy can be used to strengthen the transitions in the considered component $\mathcal{C}$.

d) Parallelizing & distributing the analysis: Our analysis can easily be parallelized. We have implemented this at two stages. First, at the level of the procedure CondSafe, we try at the same time different numbers of template conjuncts (lines 3-6 in Algo. 1), which requires calling several instances of the solver simultaneously. Secondly, at a higher level, the recursive calls of CheckSafe (line 9 in Algo. 2) are parallelized. Note that, since narrowing and the "learning" optimizations described above are considered only locally, they can be handled as asynchronous updates to the program kept in each worker, and do not require synchronization operations. Hence, distributing the analysis onto several worker processes, in the style of Bolt [4], would be possible as well.

Other directions for parallelization, which have not been implemented yet, are to return different conditional invariants in parallel when the Max-SMT problem in procedure CondSafe has several solutions. Moreover, based on experimental observations that successful safety proofs have a short successful path in the tree of proof attempts, we are also interested in exploring a look-ahead strategy: after calling CondSafe in CheckSafe, we could make recursive calls of CheckSafe on some processes while others are already applying narrowing.

e) Iterative proving: Finally, one could store the conditional invariants generated during a successful proof, which are hence invariants, so that they can be re-used in later runs. E.g., if a single component is modified, one can reprocess it and compute a new precondition that ensures its postcondition. If this precondition is implied by the previously computed invariant, the program is safe and nothing else needs to be done. Otherwise, one can proceed with the preceding components, and produce respective new preconditions in a recursive way. Only when proving safety with the previously computed invariants in this way fails, the whole program needs to be reprocessed again. This technique has not been implemented yet, as our prototype is still in a preliminary state.

V. Related Work

Safety proving is an active area of research. In the recent past, techniques based on variations of counterexample guided abstraction refinement have dominated [15]–[24]. These methods prove safety by repeatedly unfolding the program relation using a symbolic representation of program states, starting in the initial states. This process generates an over-approximation of the set of reachable states, where the coarseness of the approximation is a consequence of the used symbolic representation. Whenever a state in the over-approximation violates the safety condition, either a true counterexample was found and is reported, or the approximation is refined (using techniques such as predicate abstraction [25] or Craig interpolation [26]). When further unwinding does not change the symbolic representation, all reachable states have been found and the procedure terminates. This can be understood as a "top-down" ("forward") approach (starting from the initial states), whereas our method is "bottom-up" ("backwards"), i.e. starting from the assertions.

Techniques based on Abstract Interpretation [27] have had substantial success in the industrial setting. There, an abstract interpreter is instantiated by an abstract domain whose elements are used to over-approximate sets of program states. The interpreter then evaluates the program on the chosen abstract domain, discovering reachable states. A widening operator, combining two given over-approximations to a more general one representing both, is employed to guarantee termination of the analysis when handling loops.

"Bottom-up" safety proving with preconditions found by abduction has been investigated in [28]. This work is closest to ours in its overall approach, but uses fundamentally different techniques to find preconditions. Instead of applying Max-SMT, the approach uses an abduction engine based on maximal universal subsets and quantifier elimination in Presburger arithmetic. Moreover, it does not have an equivalent to our narrowing to exploit failed proof attempts. In a similar vein, [29] uses straight-line weakest precondition computation and backwards-reasoning to infer loop invariants supporting validity of an assertion. To enforce a generalization towards inductive invariants, a heuristic syntax-based method is used.

Automatically constructing program proofs from independently obtained subproofs has been an active area of research in the recent past. Splitting proofs along syntactic boundaries (e.g., handling procedures separately) has been explored in [1], [2], [4], [30]. For each such unit, a summary of its behavior is computed, i.e., an expression that connects certain (classes of) inputs to outputs. Depending on the employed analyzers, these summaries encode under- and over-approximations of reachable states [1] or changes to the heap using separation logic's frame rule [2]. Finally, [4] discusses how such compositional analyses can leverage cloud computing environments to parallelize and scale up program proofs.

VI. Implementation and Evaluation

We have implemented the algorithms from Sect. IV-A and Sect. IV-B in our early prototype VeryMax, using the Max-SMT solver for non-linear arithmetic [31] in the Barcelogic [32] system. We evaluated a sequential (VeryMax-Seq) and a parallel (VeryMax-Par) variant on two benchmark sets.

The first set (which we will call HOLA-BENCHS) are the 46 programs from the evaluation of safety provers in [28] (which were collected from a variety of sources, among others, [14], [33]–[43], the NECLA Static Analysis Benchmarks, etc.). The programs are relatively small (they have between 17 and 71 lines of code, and between 1 and 4 nested or consecutive loops), but expose a number of "hard" problems for analyzers. All of them are safe.

On this first benchmark set we compare with three systems. The first two were leading tools in the Software Verification Competition 2015 [44]: CPAchecker[9] [45], which was the overall winner and in particular won the gold medal in the *"Control Flow and Integer Variables"* category, and SeaHorn [46], which got the silver medal, and also won the *"Simple"* category. We also compare with HOLA [28], an abduction-based backwards reasoning tool. Unfortunately, we were not able to obtain an executable for HOLA. For this reason we have taken the experimental data for this tool directly from [28], where it is reported that the experiments were performed on an Intel i5 2.6 GHz CPU with 8 Gb of memory. For the sake of a fair comparison, we have run the other tools on a 4-core machine with the same specification, using the same timeout of 200 seconds. Tab. I summarizes the results, reporting the number of successful proofs, failed proofs, and timeouts (TO), together with the respective total runtimes. Both versions of VeryMax are competitive, and our parallel version was two times faster than our sequential one on four cores. As a reference, on these examples VeryMax-Seq needed 2.8 overall calls (recursive or after narrowings) on average, with a maximum of 16. The number of narrowings was approximately 1, with a maximum of 13. Our memoization technique making use of already failed proof attempts was employed in about one third of the cases.

In our second benchmark set (which we will refer to as NR-BENCHS) we have used integer abstractions of 217 numerical algorithms from [47]. For each procedure and for each array access in it, we have created two safety problems with one assertion each, expressing that the index is within bounds. In some few cases the soundness of array accesses in the original program depends on properties of floating-point variables, which are abstracted away. So in the corresponding abstraction some assertions may not hold. Altogether, the resulting benchmark suite consists of 6452 problems, of up to 284 lines of C code. Due to the size of this set, and to give more room to exploit parallelism (both tools with which we compare on these benchmarks, CPAchecker and SeaHorn, make use of several cores), we performed the experiments with a more powerful machine, namely, an 8-core Intel i7 3.4 GHz CPU with 16 GB of memory. The time limit is 300 seconds.

The results can be seen in Tab. II. On these instances, VeryMax is able to prove more assertions than any of the other tools, while being about as fast as SeaHorn, and significantly faster than CPAchecker. Note that many examples are solved very quickly in the sequential solver already, and thus do not profit from our parallelization. VeryMax is at an early stage of development, and is hence not yet fully tuned. For example, a number of program slicing techniques have not been implemented yet, which would be very useful for handling larger programs. Thus, we expect that further development will improve the tool performance significantly. The benchmarks and our tool can be found at http://www.cs.upc.edu/~albert/VeryMax.html.

VII. Conclusion

We have presented a novel approach to compositional safety verification. Our main contribution is a proof framework that refines intermediate results produced by a Max-SMT-based precondition synthesis procedure. In contrast to most earlier work, we proceed *bottom-up* to compute summaries of code that are guaranteed to be relevant for the proof.

We plan to further extend VeryMax to cover more program features and include standard optimizations (e.g., slicing and constraint propagation with simple abstract domains). It currently handles procedure calls by inlining, and does not support recursive functions yet. However, they can be handled by introducing templates for function pre/postconditions.

In the future, we are interested in experimenting with alternative precondition synthesis methods (e.g., abduction-based ones). We also want to combine our method with a Max-SMT-based termination proving method [12] and extend it to *existential* properties such as reachability and non-termination [5]. We expect to combine all of these techniques in an *alternating* procedure [1] that tries to prove properties at

[9]We ran CPAchecker with two different configurations, predicateAnalysis and sv-comp15.

Tool	Safe	Σ s	Fail	Σ s	TO	Total s
CPAchecker sv-comp15	33	2424.41	3	61.28	10	4489.73
CPAchecker predicateAnalysis	25	503.05	11	19.72	10	2271.12
SeaHorn	32	7.95	13	3.477	1	211.56
HOLA	43	23.53	0	0	3	623.53
VeryMax-Seq	44	293.14	2	50.69	0	343.83
VeryMax-Par	45	138.40	1	12.81	0	151.21

TABLE I
EXPERIMENTAL RESULTS ON HOLA-BENCHS BENCHMARK SET.

Tool	Safe	Σ s	Unsafe	Σ s	Fail	Σ s	TO	Total s
CPAchecker sv-comp15	5570	614803.98	251	6188.30	326	28749.78	305	735336.82
CPAchecker predicateAnalysis	5928	23417.15	170	495.13	234	9105.69	120	64652.29
SeaHorn	6077	4276.21	233	135.25	80	529.09	62	24167.11
VeryMax-Seq	6105	5940.88	0	0	326	26739.30	21	38980.80
VeryMax-Par	6106	4789.73	0	0	346	18878.42	0	23668.15

TABLE II
EXPERIMENTAL RESULTS ON NR-BENCHS BENCHMARK SET.

the same time as their duals, and which uses partial proofs to narrow the state space that remains to be considered. Eventually, these methods could be combined to verify arbitrary temporal properties. In another direction, we want to consider more expressive theories to model program features such as arrays or the heap.

REFERENCES

[1] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. Tetali, “Compositional may-must program analysis: unleashing the power of alternation,” in *POPL*, 2009.

[2] C. Calcagno, D. Distefano, P. O’Hearn, and H. Yang, “Compositional shape analysis by means of bi-abduction,” *JACM*, vol. 58, no. 6, 2011.

[3] B. Li, I. Dillig, T. Dillig, K. L. McMillan, and M. Sagiv, “Synthesis of circular compositional program proofs via abduction,” in *TACAS*, 2013.

[4] A. Albarghouthi, R. Kumar, A. V. Nori, and S. K. Rajamani, “Parallelizing top-down interprocedural analyses,” in *PLDI*, 2012.

[5] D. Larraz, K. Nimkar, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio, “Proving non-termination using Max-SMT,” in *CAV*, 2014.

[6] A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, Eds., *Handbook of Satisfiability*. IOS Press, 2009.

[7] M. Colón, S. Sankaranarayanan, and H. Sipma, “Linear Invariant Generation Using Non-linear Constraint Solving,” in *CAV*, 2003.

[8] A. R. Bradley, Z. Manna, and H. B. Sipma, “Linear ranking with reachability,” in *CAV*, 2005.

[9] A. Schrijver, *Theory of Linear and Integer Programming*. Wiley, 1998.

[10] C. Borralleras, S. Lucas, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio, “SAT Modulo Linear Arithmetic for Solving Polynomial Constraints,” *JAR*, vol. 48, no. 1, 2012.

[11] D. Jovanovic and L. M. de Moura, “Solving non-linear arithmetic,” in *IJCAR*, 2012.

[12] D. Larraz, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio, “Proving termination of imperative programs using Max-SMT,” in *FMCAD*, 2013.

[13] M. Brockschmidt, D. Larraz, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio, “Compositional Safety Verification with Max-SMT,” 2015, http://arxiv.org/abs/1507.03851.

[14] R. Sharma, I. Dillig, T. Dillig, and A. Aiken, “Simplifying loop invariant generation using splitter predicates,” in *CAV*, 2011.

[15] T. Ball and S. K. Rajamani, “The SLAM toolkit,” in *CAV*, 2001.

[16] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, “Software verification with BLAST,” in *SPIN*, 2003.

[17] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav, “SATABS: SAT-based predicate abstraction for ANSI-C,” in *TACAS*, 2005.

[18] K. L. McMillan, “Lazy abstraction with interpolants,” in *CAV*, 2006.

[19] A. Podelski and A. Rybalchenko, “ARMC: The logical choice for software model checking with abstraction refinement,” in *PADL*, 2007.

[20] A. R. Bradley, “SAT-based model checking without unrolling,” in *VMCAI*, 2011.

[21] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko, “Synthesizing software verifiers from proof rules,” in *PLDI*, 2012.

[22] A. Albarghouthi, A. Gurfinkel, and M. Chechik, “Whale: an interpolation-based algorithm for inter-procedural verification,” in *VMCAI*, 2012.

[23] A. Cimatti and A. Griggio, “Software model checking via IC3,” in *CAV*, 2012.

[24] K. Hoder and N. Bjørner, “Generalized property directed reachability,” in *SAT*, 2014.

[25] C. Flanagan and S. Qadeer, “Predicate abstraction for software verification,” in *POPL*, 2002.

[26] K. L. McMillan, “Craig interpolation and reachability analysis,” in *SAS*, 2003.

[27] P. Cousot and R. Cousot, “Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints,” in *POPL*, 1977.

[28] I. Dillig, T. Dillig, B. Li, and K. L. McMillan, “Inductive invariant generation via abductive inference,” in *OOPSLA*, 2013.

[29] C. S. Pasareanu and W. Visser, “Verification of Java programs using symbolic execution and invariant generation,” in *SPIN*, 2004.

[30] G. Yorsh, E. Yahav, and S. Chandra, “Generating precise and concise procedure summaries,” in *POPL*, 2008.

[31] D. Larraz, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio, “Minimal-model-guided approaches to solving polynomial constraints and extensions,” in *SAT*, 2014.

[32] M. Bofill, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio, “The barcelogic SMT solver,” in *CAV*, 2008.

[33] A. Gupta and A. Rybalchenko, “InvGen: An efficient invariant generator,” in *CAV*, 2009.

[34] S. Gulwani, S. Srivastava, and R. Venkatesan, “Program analysis as constraint solving,” in *PLDI*, 2008.

[35] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko, “Path invariants,” in *PLDI*, 2007.

[36] B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani, “Automatically refining abstract interpretations,” in *TACAS*, 2008.

[37] A. R. Bradley and Z. Manna, “Property-directed incremental invariant generation,” *Formal Asp. Comput.*, vol. 20, no. 4-5, 2008.

[38] A. Miné, “The octagon abstract domain,” *Higher-Order and Symbolic Computation*, vol. 19, no. 1, 2006.

[39] R. Jhala and K. L. McMillan, “A practical and complete approach to predicate refinement,” in *TACAS*, 2006.

[40] R. Sharma, A. V. Nori, and A. Aiken, “Interpolants as classifiers,” in *CAV*, 2012.

[41] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani, “SYNERGY: a new algorithm for property checking,” in *FSE*, 2006.

[42] B. S. Gulavani and S. K. Rajamani, “Counterexample driven refinement for abstract interpretation,” in *TACAS*, 2006.

[43] I. Dillig, T. Dillig, and A. Aiken, “Automated error diagnosis using abductive inference,” in *PLDI*, 2012.

[44] D. Beyer, “Software verification and verifiable witnesses - (report on SV-COMP 2015),” in *TACAS*, 2015.

[45] D. Beyer and M. E. Keremoglu, “CPAchecker: A tool for configurable software verification,” in *CAV*, 2011.

[46] T. Kahsai, J. A. Navas, A. Gurfinkel, and A. Komuravelli, “The SeaHorn verification framework,” in *CAV*, 2015.

[47] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C++ (2Nd Ed.): The Art of Scientific Computing*. New York, NY, USA: Cambridge University Press, 2002.

Formal Verification of Automatic Circuit Transformations for Fault-Tolerance

Dmitry Burlyaev
Univ. Grenoble Alpes; INRIA
dmitry.burlyaev@inria.fr

Pascal Fradet
INRIA; Univ. Grenoble Alpes
pascal.fradet@inria.fr

***Abstract*—We present a language-based approach to certify fault-tolerance techniques for digital circuits. Circuits are expressed in a gate-level Hardware Description Language (HDL), fault-tolerance techniques are described as automatic circuit transformations in that language, and fault-models are specified as particular semantics of the HDL. These elements are formalized in the Coq proof assistant and the properties, ensuring that for all circuits their transformed version masks all faults of the considered fault-model, can be expressed and proved. In this article, we consider Single-Event Transients (SETs) and fault-models of the form "at most 1 SET within k clock cycles". The primary motivation of this work was to certify the Double-Time Redundant Transformation (DTR), a new technique proposed recently [1]. The DTR transformation combines double-time redundancy, micro-checkpointing, rollback, several execution modes and input/output buffers. That intricacy requested a formal proof to make sure that no single-point of failure existed. The correctness of DTR as well as two other transformations for fault-tolerance (TMR & TTR) have been proved in Coq.**

I. Introduction

Circuit tolerance towards soft (non-destructive, non-permanent) errors has become a design characteristic as important as performance and power consumption [2]. The increased risk of soft errors results from the continuous shrinking of transistor size that makes components more sensitive to radiation [3].

The most widely-used methods to make circuits fault-tolerant rely on hardware redundancy. Triple-Modular Redundancy (TMR) [4] remains the most popular technique along with Finite State Machine (FSM) encoding (one hot, hamming, *etc.*). Some more complex ones are based on time-redundancy (re-execution) [1], [5], [6]. All these techniques can be realized through automatic circuit transformations and some of them are already supported by CAD tools. Since fault-tolerance is typically used in critical domains (aerospace, nuclear power, etc.), the correctness of such transformations is essential. If there is little doubt about the correctness of simple transformations such as TMR, this is not the case for more intricate ones.

The overall correctness of an automatic circuit transformation for fault tolerance consists not only in its functional correctness when no soft errors occur but also in its proper behavior under error occurrences. Widely-used post-synthesis verification tools (*e.g.,* model checking) are simply inappropriate to prove that a transformation ensures some property for all possible circuits; only proof-based approaches are suitable.

We propose an approach using the Coq proof assistant [7] to formally verify the functional and fault-tolerance properties of circuit transformations. We define the syntax and semantics of a simple gate-level functional HDL to describe circuits. Fault models, that specify the kind and occurrences of faults to be masked, are formalized in the language semantics. In this paper, we focus on SETs and fault-models of the form "at most 1 SET every k cycles". Fault-tolerance transformations are defined as recursive functions on the syntax of the language. Proofs rely mainly on relating the execution of the source circuit without faults to the execution of the transformed circuit *w.r.t.* the considered fault-model. They make use of several techniques (case analysis, induction on the type or the structure of circuits, co-induction on input streams).

While our approach is general, it has been originally developed to prove DTR, an involved transformation combining double-time redundancy, micro-checkpointing, rollback, several execution modes and input/output buffers [1]. If manual checks were quite useful to develop that transformation, they were error-prone and not convincing enough. This transformation served as an advanced case study. The correctness of DTR as well as two other transformations (among which TMR) have all been proved in Coq.

Section II introduces the syntax and semantics of our gate-level HDL. In section III, we present the specification of fault-models in the language formal semantics. Section IV explains the proof methodology adopted to show the correctness of circuit transformations. It is illustrated by examples taken from the simplest transformation: TMR. Section V introduces the DTR circuit transformation [1] and sketches the associated proofs. Section VI presents related work, summarizes our contributions and suggests a few extensions.

Throughout this article, we use standard mathematical and semantic notations. The corresponding Coq specifications and proofs are available online [8].

II. Circuit Description Language

We describe circuits at the gate level using a purely functional language inspired from Sheeran's combinator-based languages such as μFP [9] or Ruby [10]. We equip our language with dependent types which, along with the language syntax, ensure that circuits are well-formed by construction (gates correctly plugged, no dangling wires, no combinational loops, …).

ISBN: 978-0-9835678-5-1. Copyright owned jointly by the authors and FMCAD, Inc.