

Recursive Games for Compositional Program Synthesis

Tewodros A. Beyene¹(✉), Swarat Chaudhuri²,
Corneliu Popeea¹, and Andrey Rybalchenko³

¹ TU München, Munich, Germany
`beyene@in.tum.de`

² Rice University, Texas, USA

³ Microsoft Research, Cambridge, UK

Abstract. Compositionality, i.e., the use of procedure summarization instead of code inlining, is key to scaling automated verification to large code bases. In this paper, we present a way to exploit compositionality in the context of *program synthesis*.

The goal in our synthesis problem is to instantiate missing expressions in a procedural program so that the resulting program satisfies a safety or termination requirement in spite of an adversarial environment. The problem is modeled as a game between two players — the program and the environment — that take turns changing the program’s state and stack. The objective of the program is to ensure that all executions of this *recursive game* satisfy the requirement. Synthesis involves the modular computation of a strategy under which the program meets this objective. Our solution is based on the notion of *game summaries*, which generalize traditional procedure summaries, and relate program states in a procedural context with sets of states at which the game can return from that context. Our method for compositional reasoning about game summaries is embodied in a set of deductive proof rules. We prove these rules sound and relatively complete. We also show that a sound approximation of these rules can be automated using a Horn constraint solver that utilizes SMT-solving, counterexample-guided abstraction refinement, and interpolation. An experimental evaluation over a set of systems code benchmarks demonstrates the practical promise of the approach.

1 Introduction

The last decade has seen remarkable advances in automated software verification [6, 39]. An essential lesson from these developments is that to be scalable, techniques for reasoning about software need to be *compositional*. In other words, an analysis for a large program needs to be constructed from analyses for modules (commonly, procedures) in the program.

Specifically, successful software analysis tools like SLAM [6] and SATURN [39] use *procedure summarization* [32] to compositionally analyze large systems code bases. The idea here is to compute, for each procedure p in a program, a *summary*: a reachability relation between the input and output states of p . If p calls

a procedure q , then the summary of q is used to compute the summary of p . The approach can handle recursion: if p and q are mutually recursive, then the relationship between the summaries of p and q is given by a system of recursive equations. To compute summaries of p and q , we find a fixpoint of this system.

The use of summaries in automated verification of programs is, by now, well-understood [2, 13]. Less is known about the use of summarization in the emerging setting of *automated program synthesis* [7, 25, 34, 35]. The goal in synthesis is to generate missing expressions in a partial program so that a set of requirements are satisfied. The problem is naturally framed in terms of a *graph game* [15]. This game involves two players — the program and its environment — who take turns changing the state and stack of the program. The program wins the game if all executions of the game satisfy a user-defined requirement, no matter how the environment behaves. Synthesis amounts to the computation of a *strategy* that ensures victory for the program.

There is a large literature, going back to the 1960s, on game-theoretic program synthesis [11, 30, 36]. However, most of these approaches are: (1) restricted to the synthesis of programs over finite data domains; and (2) do not support compositional reasoning about procedural programs. While a recent paper [7] offers a synthesis method that permits programs over unbounded data, it does not support compositional reasoning. An approach for *recursive infinite-state games* — games played on the configuration graphs of programs with recursion and unbounded data — has remained elusive so far.

In this paper, we present such an approach. The key idea here is a generalization of traditional summaries, called *game summaries*, that allow compositional reasoning about strategies in the presence of procedures and recursion. Our contributions include a set of sound and complete rules for compositional, deductive synthesis using game summaries, and a way to automate a sound approximation to these rules on top of an existing automated deduction system.

Concretely, a game summary sum for a program is a relation that relates states of the program to *sets of states*. For a state s and a set of states f , we have $sum(s, f)$ whenever:

1. s is a reachable state.
2. Suppose the game starts from s in a certain procedural context. Then the program has a strategy to ensure that in all executions of game, the *first unmatched return state* — the state to which the game returns from the initial context — is in f .

The generalization to game summaries is called for as the use of traditional summaries leads to incompleteness in the game setting. Game summaries were previously explored in branching-time model checking of pushdown systems [3–5], but their use in synthesis, or for that matter analysis of infinite-state programs, is new.

Our proof rules for compositional inference of game summaries utilize quantifier alternation: an existential quantifier is used to nondeterministically guess moves for the program, and a universal quantifier is used to capture the adversarial environment. The quantifiers are second-order because summaries are

higher-order relations relating states to sets. As in the traditional verification setting, a summary sum is propagated across procedure calls and returns through inductive reasoning. The computation exploits compositionality: to generate the parts of sum involving states of a procedure q , the rule generates the parts of the summary that involve procedures that q calls, and adds these summaries to sum once and for all. Like the corresponding proof rules for verification, the rule is agnostic to whether the input transition relations encode recursion.

To verify that a safety property p is satisfied in all executions of the game, we show that for all s, f such that $sum(s, f)$, s satisfies p . A winning strategy for the program is obtained as an instantiation of the existential quantifiers used in the deduction. Synthesis with respect to termination requirements necessitates the additional use of a disjunctively well-founded transition invariant [31].

We show that our rules are sound, meaning that if they derive a strategy, then the program actually wins under the strategy. They are also relatively complete, meaning that the rules can always derive a winning strategy when one exists, assuming a suitably powerful language of assertions over local and global program variables. Importantly, this completeness proof does not require an encoding of the stack using auxiliary program variables.

We present an implementation RECSYNTH of a sound approximation to our rules on top of an existing automated deduction engine. Specifically, our implementation RECSYNTH feeds our proof rules to the EHSF engine for solving constraints in the form of Horn-like clauses that permit existential quantification in clause heads [8]. Solving the repair problem now amounts to finding an interpretation to unknown sets and relations over program variables. EHSF performs this task with some guidance from user-provided templates, and by using a combination of counterexample-guided abstraction-refinement (CEGAR), interpolation and SMT-solving.

We evaluate RECSYNTH on an array of systems programs, including device driver benchmarks drawn from the SV-COMP software verification competition [9]. Some of our benchmarks contain up to 11 K lines of C code structured into up to 181 procedures. For each of these benchmarks, we set up a synthesis problem by starting with a device driver that satisfies its requirements and eliding certain expressions from the code. Our tool is now used to find values of these expressions so that the resulting code satisfies its specification.

The experimental results are promising: in most cases, RECSYNTH is able to return successfully within a minute, depending on amount of nondeterminism to be resolved. The exploitation of compositionality is essential to these results, as inlining procedures in these examples would lead to programs that are so large as to be beyond the reach of existing program verifiers, let alone known repair/synthesis techniques.

Now we summarize the main contributions of the paper:

- We present an approach to the compositional, deductive synthesis of programs with infinite data domains as well as recursion. The method is based on the use of the new notion of *game summaries*. We give a set of sound and complete proof rules for synthesis using game summaries under safety and termination requirements.

- We offer an implementation (called RECSYNTH) of a sound but incomplete approximation of our inference rules on top of the EHSF deduction engine. We illustrate the promise of the system using an array of challenging benchmarks running into thousands of lines of code.

2 Motivation

Our program synthesis problem can be viewed as a game [15] between two players: a *program player*, whose goal is to satisfy the program's correctness requirements, and an *environment player*, which aims to prevent the program player from doing so. The two players take turns changing the configuration (state and stack) of the program. The transitions of the program come from the user-supplied partial program, with nondeterminism used to capture our lack of knowledge of certain expressions. The environment's transitions model inputs that a hostile outside world feeds to the program. As the game is played on the configuration graph of a recursive program, we call it a *recursive game*. Our goal is to find a winning *strategy* for the program player, i.e., to reduce the nondeterminism in the program's transitions so that the resulting program satisfies the requirements no matter what the environment does.

Now we show that the standard notion of summaries, ubiquitous in verification of programs with procedures, can be inadequate when solving recursive games.

We consider the source code in Fig. 1 that describes an interaction between an environment player that controls all statement except at line P and the program player that only controls the non-deterministic assignment statement at line P. The goal of the program player is to find a strategy that resolves the non-determinism at line P such that regardless of how the environment player resolves the non-determinism at line E the assertion is always satisfied.

We observe that a standard summary for `foo` can only relate values of the variables in scope `foo` at the start and exit states of its execution. That is, if a triple (x, y, pc) represents a program state then we obtain the following summary for `foo`.

$$sum((x, y, pc), (x', y', pc)) = (pc = P \wedge x' = x \wedge (y' = 0 \vee y' = 1) \wedge pc' = S)$$

<pre> void main(void) { E: if (env_nondet()) { A: foo(); B: x = 0; } else { C: foo(); D: x = 1; } F: assert(x == y); } </pre>	<pre> int x=-1, y=-1; int foo() { P: if (prog_nondet()) { Q: y = 0; } else { R: y = 1; } S: } </pre>
---	--

Fig. 1. Program that exhibits inadequacy of summaries used for verification purposes for solving games.

Hence, when reasoning about the (existence of) winning strategy for the program player we lack crucial information about the calling context in which `foo` is executed. As a result, the summary for `foo` cannot distinguish if the top of the stack stores the value `A` or `C` for the program counter of `main`. That is, when applying $sum((x, y, pc), (x', y', pc))$ in the calling context with $pc = A$, we obtain a state in which $y = 0 \vee y = 1$. Hence the subsequent assignment $x = 1$; leads to an assertion violation.

In contrast, when applying the notion of game summaries we relate each entry state of `foo` with states of `main` at the return sites `A` and `C`. Thus, the game summary can discriminate between the call site on the branch that executes $x = 0$; and the call site on the branch with $x = 1$; . As a result our method is able to identify a winning strategy for the program player.

3 Preliminaries

In this section, we formally define programs, games, and our synthesis problem.

Procedural Programs. A program consists of a finite set of procedures P , where $main \in P$ is a distinguished main procedure. For simplicity we assume that the program has no global variables (yet these can be easily added at the expense of lengthier presentation). Let v be a tuple of local variables that are in scope of each procedure.

We use an assertion $init(v)$ to describe the initial valuation of the local variables of $main$, that is, we assume there is only one such evaluation. We use $step(v, v')$ to represent intra-procedural transitions of all program procedures, i.e., the union of intra-procedural transition relations of all procedures. An assertion $call(v, v')$ represents argument passing transitions of all call sites, i.e., the union of argument passing transition relations at all call sites in the program. The left diagram below shows how the valuation of the program variables in scope changes during a call transition. For simplicity, we assume that the valuation of global variables can be modified during the argument passing.



For return value passing we use the relation $ret(v, v'', v')$ where v represents the callee state at the exit location, v'' represents caller's state at the corresponding call site, and v' is result of passing the return value (while keeping caller's local variables unchanged) and advancing the caller's program counter value beyond the call site. To model the fact that only local states are put on the stack, we assume that only the local variables of v'' occur in the return value passing relation. The right diagram above shows how the valuation of the program variables in scope changes during a return transition. We assume that an assertion $safe(v)$ represents a set of safe valuations, and thus provides the means for specifying temporal safety properties.

Recursive Games. We model the interaction between the program and its environment as a *recursive game*: a game where two players **Prog** and **Env** (standing respectively for the program and the environment) take turns in performing computation steps¹. In this paper, we assume that **Env** executes call and return transitions, as well as some of the intra-procedural steps. We capture two-player games by modifying our definition of programs as follows. We assume that instead of the monolithic intraprocedural transition relation $step(v, v')$, we are given two separate transition relations, $prog(v, v')$ and $env(v, v')$, respectively belonging to **Prog** and **Env**. Among the intra-procedural steps we assume a strict alternation between **Prog** and **Env**. That is, when considering an intra-procedural segment of the computation we assume that the first step executed in the environment, the second step is executed by the program, and so on.

Our partition of computation steps into program and environment steps is chosen to simplify the presentation in the following sections, however it does not restrict the applicability of our results. For example, in a similar way we can model the scenario where the roles of the program and the environment are exchanged, i.e., the program controls calls and returns while the environment controls some of the intra-procedural steps.

Strategies and Plays. Let S be a set of valuations of v . We refer to each $s \in S$ as a state. A stack st is a finite sequence of states, i.e., $st \in S^*$. We use “.” for sequence concatenation. We represent the empty stack by ϵ . A configuration $(s, st) \in S \times S^*$ consists of a state and a stack. A configuration (s, ϵ) such that $init(s)$ is called an initial configuration. A configuration that is in the domain of the program transition relation $prog$ is called a program configuration, otherwise it is an environment configuration. Note that the sets of program and environment configurations are mutually disjoint.

We define a transition relation $next$ on configurations that takes into account both program and environment transitions below.

$$\begin{aligned} next((s, st), (s', st')) = & (prog(s, s') \vee env(s, s')) \wedge st = st' \vee \\ & call(s, s') \wedge st' = (s \cdot st) \vee \\ & \exists s'' : ret(s, s'', s') \wedge st = (s'' \cdot st') \end{aligned}$$

We define a computation tree as a node-labeled tree that satisfies the following conditions. The root is labeled by an initial configuration. Every pair of parent/child nodes (s, st) and (s', st') is related by $next((s, st), (s', st'))$.

A play π is a sequence of configurations that labels a branch of a computation tree. We write π_i , s_i and st_i to refer to the i -th configuration, the i -th state, and the i -th stack of the play, respectively. A play is safe if each of its states s satisfies $safe(s)$.

¹ The name *recursive game* captures the fact that our games are played on the configuration graphs of recursive programs, which can be infinite even when program variables range over finite data domains.

A safe strategy for **Prog** (respectively, **Env**) is a computation tree such that each node that is labeled by an environment (respectively, program) configuration (s, st) contains the entire set $\{(s', st') \mid \text{next}((s, st), (s', st'))\}$ as its children, and each play is safe. A terminating strategy for **Prog** (respectively, **Env**) is defined similarly and requires that each play is finite.

4 Solving Recursive Games

In this section, we present a set of deductive proof rules for synthesizing safe and terminating strategies for compositional program synthesis. These proof rules determine whether **Prog** has a winning strategy by solving implication and well-foundedness constraints on auxiliary assertions over system variables. The rules are based on the notion of *game summaries*, which generalize summaries used in program verification and analysis and permit compositional reasoning in the setting of games. Finally, the rules are sound as well as relatively complete.

4.1 Game Summaries

Given a play π , we define a reachability relation \rightsquigarrow_π that connects positions whose configurations are in the same calling context. Formally, we define

$$i \rightsquigarrow_\pi j = (i \leq j \wedge st_i = st_j \wedge \forall k : i < k < j \rightarrow \exists st' : st' \cdot st_i = st_k).$$

For a configuration π_i we define the set of first unmatched returns (FUR) as the set of configurations that are obtained by following the return transition out the π_i 's calling context. Formally, we obtain the following set.

$$\{\pi'_{j+1} \mid \exists \pi' : \pi_1 = \pi'_1 \wedge \dots \wedge \pi_i = \pi'_i \wedge i \rightsquigarrow_{\pi'} j \wedge \exists s : st_j = s \cdot st_{j+1}\}$$

In the set comprehension above, we ensure that π'_{j+1} is the FUR configuration by asking for a play π' that overlaps with π until the position i , connects π'_{j+1} with π_i within the same calling context, and actually results from a return transition.

A *game summary* relates states with (over-approximations of) sets of states occurring in their FUR configurations.

4.2 Safe Strategies

We consider a synthesis problem where **Prog** has a winning strategy if only states from *safe*(v) are visited by all plays.

We present the corresponding proof rule in Figure 2. The proof rule relies on a game summary *sum*. We connect the game summary with the reachable states by resorting to reasoning by induction on the number of steps required to reach a state from its entry state. S1 requires that for any initial state s_0 of the program, i.e., $\text{init}(s_0)$, we have $\text{sum}(s_0, \emptyset)$. S2 represent the induction step for intra-procedural steps. Let us assume a state s_1 is given together with a set of states R_1 such that $\text{sum}(s_1, R_1)$. We require that for every state s_2 satisfying

$env(s_1, s_2)$ there exists a set of states R_2 such that $sum(s_2, R_2)$ and $R_1 \supseteq R_2$, and there exist a state s_3 such that $prog(s_2, s_3)$. We also require that there exists a set of states R_3 such that $sum(s_3, R_3)$ and $R_2 \supseteq R_3$. Let us assume a state s_1 is given together with a set of states R_1 such that $sum(s_1, R_1)$. For any state s_2 such that $call(s_1, s_2)$, S3 requires that there exists a set of states R_2 such that $sum(s_2, R_2)$. S4 represent the induction step for a call step. Given states s_1 and s_2 together with sets of states R_1 and R_2 , let us assume $call(s_1, s_2)$, $sum(s_1, R_1)$ and $sum(s_2, R_2)$. For any $s_3 \in R_2$, we require that there exists a set of states R_3 such that $sum(s_3, R_3)$ and $R_1 \supseteq R_3$. S1 and S3 ensure that $sum(v, X)$ is defined at entry states of the program and each procedural level. S2 and S4 ensure that the set of states associated with each reachable state shrinks while traversing on the same procedural level. S5 represent the induction step for a return step. Given the states s_1 and s_2 and a set of states R_1 such that $sum(s_1, R_1)$ and $ret(s_1, s_2)$, we require that s_2 is in R_1 . Since the winning condition requires all states to satisfy $safe(v)$, each state s such that $sum(s, R)$ needs to satisfy $safe(s)$. This condition is enforced by S6.

Example 1. We show how the safety proof rule can be applied using the example in Fig. 1.

Since the initial state of `main` is $(E, -1, -1)$, by S1 we have $sum((E, -1, -1), \emptyset)$. Assuming `Env` decides to move from `E` to `A`, we apply S2 to derive $sum((A, -1, -1), \emptyset)$. To strictly adhere with the alternation of players, we assume that `Prog` does a skip. From $pc = A$, `Env` makes a call to `foo` and the program control will go to the state $(P, -1, -1)$. S3 ensures that there exists a set of states R_1 such that $sum((P, -1, -1), R_1)$. From $sum((P, -1, -1), R_1)$, let `Env` make a skip and let `Prog` move to `R` thereby reaching the state $(R, -1, -1)$. S3 ensures that there exists a set of states R_2 such that $sum((R, -1, -1), R_2)$ and $R_1 \supseteq R_2$. From $sum((R, -1, -1), R_2)$, let once again `Env` make a skip and let `Prog` move to `S` by updating value of y to 1 thereby reaching the state $(S, -1, 1)$. S3 ensures that there exists a set of states R_3 such that $sum((S, -1, 1), R_3)$ and $R_2 \supseteq R_3$. The return step from $(S, -1, 1)$ in `foo` to $(B, -1, 1)$ in `main` together with S5 ensures that $(B, -1, 1)$ is in R_3 and by transitivity in R_2 and R_1 . From $sum((E, -1, -1), \emptyset)$, $call((E, -1, -1), (P, -1, -1))$, and $sum((P, -1, -1), \{(B, -1, 1)\})$, S4 ensures that there exists R_4 such that $sum((B, -1, 1), R_4)$. Continuing in a similar way from $sum((B, -1, 1), R_4)$ by applying S2, we reach a state $(F, 0, 1)$ which violates the assertion.

However, from $sum((P, -1, -1), R_1)$, if `Env` makes a skip and `Prog` moves to `Q` instead `R`, the assertion will be eventually satisfied. If `Env` decides to move from `E` to `B` (instead of `E` to `A`), `Prog` needs to move to `R` instead `Q` for the assertion to be satisfied. Therefore, the safe strategy for `Prog` should use information on the top of the stack to know if it should move to `Q` or `R` from `P`. For example, replacing `prog_nondet()` by $pc = A$ provides a winning strategy for `Prog`.

Theorem 1 (Correctness of Rule RuleSafe). *The proof rule RULESAFE is sound and relatively complete.* ■

Find sum such that:

S1: $init(v)$	$\rightarrow sum(v, \emptyset)$
S2: $sum(v_1, X_1) \wedge env(v_1, v_2)$	$\rightarrow \exists X_2 : sum(v_2, X_2) \wedge X_1 \supseteq X_2 \wedge$ $\exists v_3 : prog(v_2, v_3) \wedge \exists X_3 : sum(v_3, X_3) \wedge X_2 \supseteq X_3$
S3: $sum(v_1, X_1) \wedge call(v_1, v_2)$	$\rightarrow \exists X_2 : sum(v_2, X_2)$
S4: $sum(v_1, X_1) \wedge call(v_1, v_2) \wedge sum(v_2, X_2) \wedge X_2(v_3)$	$\rightarrow \exists X_3 : sum(v_3, X_3) \wedge X_1 \supseteq X_3$
S5: $sum(v_1, X) \wedge ret(v_1, v_2)$	$\rightarrow X(v_2)$
S6: $sum(v, X)$	$\rightarrow safe(v)$

Fig. 2. Proof rule RULESAFE for synthesis with respect to a safety requirement given by assertion $safe(v)$.

A complete proof of the theorem is given below. But first, we define the following auxiliary predicate S that imposes a certain totality and monotonicity condition on game summaries. When considering a pair of configurations in the same calling context, the game summary sum needs to provide corresponding state sets and these state sets need to be non-increasing.

$$S(\pi, i, j) = i \rightsquigarrow_{\pi} j \rightarrow \exists R_i \exists R_j : sum(s_i, R_i) \wedge sum(s_j, R_j) \wedge R_i \supseteq R_j$$

We extend the predicate to range over a prefix of a play as follows.

$$H(\pi, k) = (\forall i \forall j : 0 \leq i \leq j \leq k \rightarrow S(\pi, i, j))$$

The following lemma is crucial for proving the soundness of the proof rule for proving the existence of safe strategies.

Lemma 1. *For each play π and each of its positions k we have $H(\pi, k)$.*

Proof. Let π be a play and k be a position in this play. We prove the lemma by induction over k .

First, we consider the base case $k = 0$. Since $init(s_0)$, from S1 follows $S(\pi, 0, 0)$ via $R_0 = R_0 = \emptyset$.

For the induction step we assume $H(\pi, k)$ and prove $H(\pi, k + 1)$. After expanding definitions of H the proof goal is $\forall i \forall j : 0 \leq i \leq j \leq k + 1 \rightarrow S(\pi, i, j)$. For i and j such that $0 \leq i \leq j \leq k$ we obtain $S(\pi, i, j)$ from $H(\pi, k)$ directly. In the rest of this proof we consider the case $0 \leq i \leq j = k + 1$, i.e., our proof goal becomes

$$\exists R_i \exists R_{k+1} : sum(s_i, R_i) \wedge sum(s_{k+1}, R_{k+1}) \wedge R_i \supseteq R_{k+1}$$

for arbitrary $0 \leq i \leq k + 1$ such that $i \rightsquigarrow_{\pi} k + 1$. We proceed by performing a case distinction on how π_k transitions to π_{k+1} .

In case $env(s_k, s_{k+1})$ we rely on the induction hypothesis to obtain R_i and R_k such that $sum(s_i, R_i)$, $sum(s_k, R_k)$, and $R_i \supseteq R_k$. The consequence of S2 yields R_{k+1} such that $sum(s_{k+1}, R_{k+1})$ and $R_k \supseteq R_{k+1}$, which together with $R_i \supseteq R_k$ proves our goal.

For $prog(s_k, s_{k+1})$ we first consider that $env(s_{k-1}, s_k)$ since the program step is always preceded by an environment step, so we have $k - 1 \geq 0$. From the induction hypothesis we obtain corresponding $R_i \supseteq R_{k-1}$. Thus, the premise of S2 holds, as $sum(s_{k-1}, R_{k-1}) \wedge env(s_{k-1}, s_k)$. Hence there exists R_k such that $sum(s_k, R_k)$ and $R_{k-1} \supseteq R_k$, as well as there exists R_{k+1} such that $sum(s_{k+1}, R_{k+1})$ and $R_k \supseteq R_{k+1}$. Hence, we meet our proof goal.

If $call(s_k, s_{k+1})$ then $i = k + 1$ since π_{k+1} is an entry configuration. Hence from $S(\pi, k, k)$ and S3 we directly prove our goal.

With $ret(s_k, s_{k+1})$ we first observe that there is a call configuration π_c and an entry configuration π_e such that $call(\pi_c, \pi_e)$. This call yields the exit configuration π_k and the return configuration π_{k+1} . Since $c \rightsquigarrow_\pi k + 1$ we have $i \rightsquigarrow_\pi c$. From the induction hypothesis we obtain corresponding R_i and R_c such that $R_i \supseteq R_c$. Similarly, from $e \rightsquigarrow_\pi k$ we obtain corresponding $R_e \supseteq R_k$. For S5 we obtain the premise $sum(s_k, R_k) \wedge ret(s_k, s_{k+1})$, and hence $R_k(s_{k+1})$. By transitivity, we have $R_e(s_{k+1})$. We instantiate the premise of S4 as follows.

$$sum(s_c, R_c) \wedge call(s_c, s_e) \wedge sum(s_e, R_e) \wedge R_e(s_{k+1})$$

As a consequence we get R_{k+1} such that $sum(s_{k+1}, R_{k+1})$ and $R_c \supseteq R_{k+1}$. Hence, we have $R_i \supseteq R_{k+1}$, which proves the goal. \square

Coming back to RULESAFE, we split the correctness proof into two parts: soundness and relative completeness.

Soundness. If there exists sum that satisfies premises of RULESAFE then the program has a strategy to win the safety game.

Proof. For a proof by contradiction we assume that sum satisfies the premises of RULESAFE and the program does not have a safe strategy. Hence, there exists a strategy for the environment in which every play eventually violates the safety condition. Let us take one such play π and its position p in which the safety condition is violated. By Lemma 1 for the position p we obtain R_p such that $sum(s_p, R_p)$. Hence from S6 follows $safe(s_p)$, which is a contradiction to our assumption that sum satisfies the premises of RULESAFE. \square

Relative Completeness. If the program has a strategy to win the safety game then there exists sum that satisfies premises of RULESAFE.

Proof. Let us assume that Prog has a safe strategy, i.e., the conclusion of RULESAFE holds. This strategy σ alternates between universal choices of Env and existential choices of Prog. We prove the completeness claim by showing how to construct sum satisfying the premises of the rule. Let $sum(s, R)$ holds for each state s that occurs in a configuration (s, st) of some play where R is the corresponding set of first unmatched return states.

Since the initial state, say s , occurs in the strategy, $sum(s, R)$ is defined such that $R = \emptyset$ and hence S1 is satisfied.

Now we consider an arbitrary pair (s_0, R_0) such that $\text{sum}(s_0, R_0)$. The strategy guarantees that for every successor s_1 of s_0 wrt. **Env** there exists a successor s_2 wrt. **Prog**. For every such s_2 , there exists a set of FURs R_2 such that $\text{sum}(s_2, R_2)$ since s_2 is an **Env** state. In addition, $R_2 \subseteq R_0$ since the set of FURs may only shrink across intra-procedural steps. i.e., sum satisfies S2.

Let us take an arbitrary pair (s_0, R_0) such that $\text{sum}(s_0, R_0)$, and a state s_1 such that $\text{call}(s_0, s_1)$. For the set of FURs R_1 of s_1 , we have $\text{sum}(s_1, R_1)$ since s_1 is an **Env** state. This shows $\text{sum}(v, R)$ satisfies S3.

Next, let us assume that for arbitrary states s_0 and s_1 , we have $\text{sum}(s_0, R_0)$ and $\text{sum}(s_1, R_1)$, and $\text{call}(s_0, s_1)$. It follows that for any $s_2 \in R_1$ and a set of its FURs R_2 , $\text{sum}(s_2, R_2)$ holds since s_2 is an **Env** state. In addition, since s_2 is in the same procedural level as s_0 , $R_2 \subseteq R_0$, i.e. S4 is satisfied.

Now let us assume that for arbitrary states s_0 and s_1 , we have $\text{ret}(s_0, s_1)$ and $\text{sum}(s_0, R_0)$. By definition of FURs, we see that s_1 should be in R_0 , satisfying S5.

Finally, for all pairs (s, R) such that $\text{sum}(s, R)$, we have $\text{safe}(s)$ since we consider a safe strategy. Therefore, sum also satisfies S6. \square

5 Terminating Strategies

Let us now consider a synthesis problem where **Prog** has a winning strategy if a state from which no further move can be made is eventually reached by each play. Reasoning about such eventuality properties demands the use of well-founded orders.

We connect the invariant assertion with the reachable states by resorting to reasoning by induction on the number of steps required to reach a state from its entry state.

T1 requires that for any initial state s_0 of the program, $\text{sum}(s_0, s_0, \emptyset)$. T2 represent the induction step for intra-procedural steps. Let us assume a state s_1 is given together with its entry state s_0 and a set of states R_1 such that $\text{sum}(s_0, s_1, R_1)$. We require that for every state s_2 satisfying $\text{env}(s_1, s_2)$ there exists a set of states R_2 such that $\text{sum}(s_0, s_2, R_2)$ and $R_1 \supseteq R_2$, and there exist a state s_3 such that $\text{prog}(s_2, s_3)$. We also require that there exists a set of states R_3 such that $\text{sum}(s_0, s_3, R_3)$ and $R_2 \supseteq R_3$. We also require that (s_1, s_3) is in *round*. Assume for a state s_1 , $\text{sum}(s_0, s_1, R_1)$ is given. For any state s_2 such that $\text{call}(s_1, s_2)$, T3 requires that there exists a set of states R_2 such that $\text{sum}(s_2, s_2, R_2)$. T4 represent the induction step for a call step. Given states s_1 and s_2 together with sets of states R_1 and R_2 , let us assume $\text{call}(s_1, s_2)$, $\text{sum}(s_0, s_1, R_1)$ and $\text{sum}(s_2, s_2, R_2)$. For any $s_3 \in R_2$, we require that there exists a set of states R_3 such that $\text{sum}(s_0, s_3, R_3)$ and $R_1 \supseteq R_3$. We also require that (s_1, s_3) is in *round*. T1 and T3 ensure that sum is defined at entry states of the program and each procedural level. T2 and T4 ensure that the set of states associated with each reachable state shrinks while traversing on the same procedural level. T5 represent the induction step for a return step. For a return step (s_1, s_2) such that $\text{sum}(s_0, s_1, R_1)$, we require that s_2 is in R_1 . To ensure that the game progresses when aiming at termination, we keep track of pairs

of states across every call site in *descent*. This is done in T3. Finally, to ensure termination by each play we require that both *descent* and *round* represent a well-founded relation. Thus, it is impossible to return to *sum* infinitely many times. This is captured by T6 and T7 (Fig. 3).

Find *sum*, *round*, and *descent* such that:

- T1: $init(v) \rightarrow sum(v, v, \emptyset)$
- T2: $sum(v_1, v_2, R_1) \wedge env(v_2, v_3) \rightarrow \exists R_2 : sum(v_1, v_3, R_2) \wedge R_1 \supseteq R_2$
 $\wedge \exists v_4 : prog(v_3, v_4) \wedge round(v_2, v_4)$
 $\wedge \exists R_3 : sum(v_1, v_4, R_3) \wedge R_3 \supseteq R_2$
- T3: $sum(v_1, v_2, R_1) \wedge call(v_2, v_3) \rightarrow \exists R_2 : sum(v_3, v_3, R_2) \wedge descent(v_1, v_3)$
- T4: $sum(v_1, v_2, R_1) \wedge call(v_2, v_3)$
 $\wedge sum(v_3, v_3, R_2) \wedge R_2(v_4) \rightarrow \exists R_3 : sum(v_1, v_4, R_3) \wedge R_1 \supseteq R_3 \wedge round(v_2, v_4)$
- T5: $sum(v_1, v_2, R) \wedge ret(v_2, v_3) \rightarrow R(v_3)$
- T6: $well\text{-}founded(round)$
- T7: $well\text{-}founded(descent)$

Fig. 3. Proof rule RULETERM for synthesis with respect to the termination requirement.

Theorem 2 (Correctness of Rule RuleTerm). *The proof rule RULETERM is sound and relatively complete.* ■

The complete proof of the theorem can be found in the appendix section.

6 Evaluation

In this section we describe an experimental evaluation of our compositional synthesis approach on infinite-state programs. The evaluation relies on a solver of Horn clauses with alternating quantification, so we first describe this class of clauses.

Implementation. Our prototype implementation RECSYNTH is based on two modules. The first module is a C frontend, derived from the CIL library [28], that transforms C code into verification conditions represented as Horn clauses. This transformation is based on a sound approximation of our proof rules. The second module of RECSYNTH is a solver for Horn clauses that is based on predicate abstraction and counterexample guided abstraction refinement [8]. The Horn

clause solver EHSF is implemented in SICStus Prolog and uses the CLP(Q) solver for handling linear constraints [22] and the Z3 solver [14] for handling non-linear constraints.

We skip the syntax and semantics of the clauses targeted by this system — see [8] for more details. Instead, we illustrate these clauses with the following example:

$$\begin{aligned} x \geq 0 \rightarrow \exists y : x \geq y \wedge \text{rank}(x, y), \quad \text{rank}(x, y) \rightarrow ti(x, y), \\ ti(x, y) \wedge \text{rank}(y, z) \rightarrow ti(x, z), \quad \text{dwf}(ti). \end{aligned}$$

These clauses represent an assertion over the interpretation of “query symbols” *rank* and *ti* (the second order predicate *dwf* represents disjunctive well-foundedness [31], and is not a query symbol). The semantics of these clauses maps each predicate symbol occurring in them into a constraint over v . Specifically, the above set of clauses has a solution that maps both *rank*(x, y) and *ti*(x, y) to the constraint $(x \geq 0 \wedge y \leq x - 1)$.

EHSF resolves clauses like the above using a CEGAR scheme to discover witnesses for existentially quantified variables. The refinement loop collects a global constraint that declaratively determines which witnesses can be chosen. The chosen witnesses are used to replace existential quantification, and then the resulting universally quantified clauses are passed to a solver over decidable theories, e.g., HSF [16] or μZ [21]. Such a solver either finds a solution, i.e., a model for uninterpreted relations constrained by the clauses, or returns a counterexample, which is a resolution tree (or DAG) representing a contradiction. EHSF turns the counterexample into an additional constraint on the set of witness candidates, and continues with the next iteration of the refinement loop.

For the existential clause above, EHSF introduces a witness/skolem relation *sk* over variables x and y , i.e., $x \geq 0 \wedge sk(x, y) \rightarrow x \geq y \wedge \text{rank}(x, y)$. In addition, since for each x such that $x \geq 0$ holds we need a value y , we require that such x is in the domain of the Skolem relation using an additional clause $x \geq 0 \rightarrow \exists y : sk(x, y)$. In the EHSF approach, the search space of a skolem relation *sk*(x, y) is restricted by a template function $\text{TEMPL}(sk)(x, y)$. To conclude this example, we note that one possible solution returned by EHSF is the skolem relation $sk(x, y) = (y = x - 1)$.

Benchmarks. For evaluation, we used benchmarks from the repository of the SV-COMP verification competition [9]. We selected 10 driver files from the directories `ntdrivers` and `ntdrivers-simplified` with sizes ranging between 576 and 11K lines of code. Each benchmark contains assertions that correspond to safety specifications. Due to their complexity and size, these driver benchmarks have been considered a litmus test for verification tools during the last decade [1, 10, 20].

For each benchmark file, our experiments consist of 3 conceptual steps:

- (1) We mark a C expression in the input file where non-determinism is to be resolved. (We call the code region that contains this expression a *hole*.)
- (2) We use the frontend to generate a program representation in Horn clause

form. **(3)** We solve the Horn clauses using EHSF and the solution returned by EHSF corresponds to synthesised-code to fill the hole in the code. If EHSF succeeds in finding a solution for the Horn clauses, our approach guarantees that the device driver code with the hole replaced by the EHSF’s solution satisfies the safety specification present in the original benchmark.

First, we describe in detail the SV-COMP example `kbfiltr_simpl1`, however in an abridged form due to space reasons. Similar to other C benchmark files, `kbfiltr_simpl1` contains code corresponding to the driver and the test harness.

```

419: NTSTATUS IofCallDriver(PDEVICE_OBJECT DvObj, PIRP Irp) {
420:     NTSTATUS returnVal2 ;
421:     ...
456:     if (?) { /* expression to synthesize */
457:         s = IPC;
458:         lowerDriverReturn = returnVal2;
459:     } else {
460:         if (s == MPR1) {
461:             if (returnVal2 == 259L) {
462:                 s = MPR3;
463:                 lowerDriverReturn = returnVal2;
464:             } else {
465:                 s = NP;
466:                 lowerDriverReturn = returnVal2;
467:             }
468:         } else {
469:             if (s == SKIP1) {
470:                 s = SKIP2;
471:                 lowerDriverReturn = returnVal2;
472:             } else { assert(0); }
473:         }
474:     }
475:     return (returnVal2);
476: }
```

Fig. 4. Part of function `IofCallDriver`.

See Fig. 4 for the function `IofCallDriver`, a function that is invoked repeatedly on many execution paths of the driver. This function has two arguments and some of the variables accessed in its body have global scope, i.e., the variables `s`, `IPC`, `lowerDriverReturn`, `MPR1`, `MPR3`, `NP`, `SKIP1` and `SKIP2`. The safety requirement is instrumented in the code using a finite-state automaton representation, where the variable `s` corresponds to the current state of the automaton. The variable `s` is assigned integer values corresponding to different states of the automaton, i.e., `UNLOADED` = 0, `NP` = 1, `DC` = 2, `SKIP1` = 3, `SKIP2` = 4,

MPR1 = 5, MPR3 = 6 and IPC = 7. The file contains 10 assertions, including the assertion shown on line 472. For our experiment, we marked the code region from line 456 as non-deterministic. (The original SV-COMP benchmark file contained the conditional test $s == NP$ on line 456.)

For applying RECSYNTH we provide a template corresponding to the hole expression that reflects the choice of automaton states

$$\begin{aligned} \text{TEMPL}(sk)(v) = \\ (\text{?}_{\text{UNLOADED}} * \text{UNLOADED} + \text{?}_{\text{NP}} * \text{NP} + \text{?}_{\text{DC}} * \text{DC} + \text{?}_{\text{SKIP1}} * \text{SKIP1} \\ + \text{?}_{\text{SKIP2}} * \text{SKIP2} + \text{?}_{\text{MPR1}} * \text{MPR1} + \text{?}_{\text{MPR3}} * \text{MPR3} + \text{?}_{\text{IPC}} * \text{IPC} = s) \end{aligned}$$

together with a template constraint

$$\begin{aligned} 0 \leq \text{?}_{\text{UNLOADED}} \leq 1 \wedge 0 \leq \text{?}_{\text{NP}} \leq 1 \wedge 0 \leq \text{?}_{\text{DC}} \leq 1 \wedge 0 \leq \text{?}_{\text{SKIP1}} \leq 1 \\ \wedge 0 \leq \text{?}_{\text{SKIP2}} \leq 1 \wedge 0 \leq \text{?}_{\text{MPR1}} \leq 1 \wedge 0 \leq \text{?}_{\text{MPR3}} \leq 1 \wedge 0 \leq \text{?}_{\text{IPC}} \leq 1 \\ \wedge \text{?}_{\text{UNLOADED}} + \text{?}_{\text{NP}} + \text{?}_{\text{DC}} + \text{?}_{\text{SKIP1}} + \text{?}_{\text{SKIP2}} + \text{?}_{\text{MPR1}} + \text{?}_{\text{MPR3}} + \text{?}_{\text{IPC}} = 1 \end{aligned}$$

that reflects a comparison with an automaton state and excludes arithmetic operations on them.

The task of EHSF is to find suitable values for the template parameters, i.e., the unknown coefficients $\text{?}_{\text{UNLOADED}}$, ?_{NP} , ?_{DC} , ?_{SKIP1} , ?_{SKIP2} , ?_{MPR1} , ?_{MPR3} , and ?_{IPC} , and thus determine the *hole* expression. RECSYNTH returns in 1s with the solution $NP = s$.

Results. For our experiments we used a computer with an Intel Core i7 2.3 GHz CPU and 16 GB of RAM. See Table 1 for our experimental results. For each of the 10 SV-COMP benchmark files, we list the benchmark name and three synthesis scenarios named after a function where the synthesis region is located (Column 1). We also report the size of the file (Column 2) and results of running RECSYNTH (Column 4,5,6). For each file we also report verification results using the complete driver code. For example, the result from the first row of the benchmark **parport** indicates that verifying the driver code succeeds after 19s. (The benchmark indeed satisfies its safety specification.) For the three code regions, **IoofCalldriver**, **PptDispatch**, and **KeSetEvent**, our tool synthesises a solutions after 26s, 27s, and 33s, respectively.

In all cases, RECSYNTH is able to succeed within 2-3 times overhead compared to the verification time. We inspected the synthesized expressions and observed that in most cases we obtain the original expressions that was erased when constructing the benchmark. In the remaining cases the synthesized expressions were logically equivalent to the original expressions.

Overall, our results indicate the feasibility of our synthesis approach across a range of different drivers and code regions to synthesize.

Table 1. Application of RECSYNTH on 10 drivers.

Benchmark	LOC	Time (sec)		Steps	Benchmark	LOC	Time (sec)		Steps
		Total	SMT				Total	SMT	
kbfiltr_simpl1	576	0.9			cdaudio_simpl1	2124	8.1		
IofCallDriver		1.1	0.5	3	IofCallDriver		11.7	3.7	13
StubDriverInit		1.4	0.6	12	HPCdrDevice		11.9	3.9	12
KbFilterPnP		1.3	0.5	4	KeSetEvent		12.9	4.1	16
kbfiltr_simpl2	1001	1.2			diskperf	4462	3.2		
IofCallDriver		1.9	0.9	3	IofCallDriver		3.6	0.9	10
StubDriverInit		2.1	0.6	12	FwdIrpSync		4.4	1.2	14
KbFilterPnP		1.9	0.5	4	KeSetEvent		4.2	1.1	14
diskperf_simpl1	1095	2.7			floppy	8285	6.3		
IofCallDriver		3.6	1.5	8	IofCallDriver		7.8	2.8	11
FwdIrpSync		3.9	1.2	12	FloppyPnp		7.7	2.9	11
KeSetEvent		3.6	1.1	11	KeSetEvent		9.2	3.1	16
floppy_simpl3	1123	3.8			cdaudio	8827	11.3		
IofCallDriver		3.3	0.9	1	IofCallDriver		14.2	4.1	22
FloppyPnp		3.4	1.0	1	HPCdrDevice		10.9	3.4	19
KeSetEvent		4.1	1.4	6	KeSetEvent		11.6	4.3	24
floppy_simpl4	1598	5.6			parport	10934	13.5		
IofCallDriver		6.7	1.2	1	IofCallDriver		17.7	4.2	14
FloppyPnp		6.8	1.3	1	PptDispatch		18.7	3.8	13
KeSetEvent		7.5	1.9	6	KeSetEvent		22.1	4.4	18

7 Related Work

The last few years have seen much work on constraint-based software synthesis [25, 34, 35, 37]. Like our paper, these approaches advocate synthesis from partial programs, and leverage modern SMT-solving and invariant generation techniques. However, most of these approaches are not compositional. Exceptions include work on *component-based* synthesis, where programs are synthesized by composing routines from a software library in an example-driven way [23], and modular sketching [33]. The former work is restricted to the synthesis of loop-free programs. The latter work allows the use of summaries for library functions called from a procedure with missing expressions, but requires that the library procedures do not contain unknown expressions themselves. In contrast, our approach synthesizes programs with procedures that call each other in arbitrary ways.

There is a rich literature on synthesis and repair of finite-state reactive systems based on game-theoretic techniques [11, 24, 27, 30, 36], using both explicit-state [36] and symbolic [29] approaches. Also well-known are algorithms for *pushdown games* [12, 38], which can be expanded into synthesis algorithms for reactive programs with procedures and finite-domain variables [17]. Synthesis of finite-state reactive systems from components has also been studied [26].

The elemental distinction between these approaches and ours is that our programs can handle data from infinite domains.

Game summaries have previously been explored in the context of branching-time model checking of pushdown systems [3–5]. Pushdown systems can be viewed as recursive programs over finite data domains. Branching-time model checking of pushdown systems is a computationally hard problem — EXPTIME-complete in the size of the pushdown system. This is why the traditional definition of summaries, which gives an algorithm that is polynomial in the system size, does not suffice here. [3, 5] give an algorithm for this problem based on game summaries. However, this algorithm relies on the fact that pushdown systems have a finite number of control states and stack symbols, and assumes an explicit, rather than symbolic, representation of summaries. Two keys contribution of our work are an extension of the idea of game summaries to a setting with infinite data domains, and its application in synthesis.

8 Conclusion

We have presented a constraint based approach to computing winning strategies in infinite-state games. The approach consists of: (1) a set of sound and relatively complete proof rules for solving such games, and (2) automation of the rules on top of an existing automated deduction engine. We demonstrate the practical promise of our approach through several case studies using examples derived from prior work on program repair and synthesis.

Many avenues for future work remain open. The system we have presented is a prototype. Much more remains to be done on engineering it for greater scalability. In particular, we are especially interested in applying the system to reactive synthesis questions arising out of embedded systems and robotics. On the theoretical end, exploring opportunities of synergy between our approach and abstraction-based [18, 19] and automata-theoretic approaches to games [36] remains a fascinating open question.

A Correctness Proofs for RuleTerm

Proof. We split the proof into two parts: soundness and relative completeness.

Soundness. We prove the soundness by contradiction.

Assume that there exist an assertions $sum(v_1, v_2, R)$, $round(v_1, v_2)$ and $descent(v_1, v_2)$ that satisfy the premises of the rule, yet the conclusion of the rule does not hold. That is, there is no winning strategy for **Prog**.

Hence, there exists a strategy σ for **Env** in which each play does not terminates. This strategy σ alternates between existential choices of **Env** and universal choices of **Prog**. Let $aux(v)$ be a set of states for which σ provides existentially chosen successors wrt. **Prog**. Note that no play terminates from any $s \in aux(v)$ since no play determined by σ terminates.

We derive a contradiction by relying on a certain play π that is determined by σ . The play π is constructed iteratively. We start from some root state s_0 of σ , which satisfies the initial condition $init(v)$. Note that $sum(s_0, s_0, R_0)$, due to T1, and $aux(s_0)$ due to σ .

Each iteration round extends the matched play $s_0..s$ obtained so far in three ways:

- by two states, say s_1 and s_2 where $env(s, s_1)$ and $prog(s_1, s_2)$,
- by a state, say s_1 where $call(s, s_1)$, or
- by a sequence of states $s_1..s_2$ where we have $call(s, s_1)$, $sum(s_1, R_1)$, and $s_1..s_2$ is a play from s_1 to one of its FURs $s_2 \in R_1$.

We maintain a condition that for the last state s of each such play, $sum(s_0, s, R)$ and $aux(s)$ where $s_0 = entry(s)$, i.e., s_0 is the entry state of the calling context of s .

Let s be the last state of the play π constructed so far, and $s_0 = entry(s)$. Due to our condition, we have $sum(s_0, s, R)$ and $aux(s)$. We iteratively construct a play π taking one of the following steps at a time:

- σ determines a successor state s_1 such that $env(s, s_1)$, and T2 guarantees that there exists a state s_2 such that $prog(s_1, s_2)$, $round(s, s_2)$, and $sum(s_0, s_2, R_2)$ such that $R_2 \subseteq R$. The play is extended by s_1, s_2 . Furthermore, $aux(s_2)$ due to G.
- σ determines a successor state s_1 such that $call(s, s_1)$, and T3 guarantees that there exists a set of FURs R_1 of s_1 such that $sum(s_1, s_1, R_1)$, and also $descent(s_0, s_1)$. The play is extended by s_1 . Furthermore, $aux(s_1)$ due to σ .
- σ determines a sequence of successor state s_1 such that $call(s, s_1)$, where $sum(s_1, s_1, R_1)$ is given together with some $s_2 \in R_1$. Here, T4 guarantees that there exists a set of FURs R_2 for s_2 such that $sum(s_0, s_2, R_2)$ where $R_2 \subseteq R$, and also $round(s, s_2)$. The play is extended by $s_1..s_2$. Furthermore, $aux(s_2)$ due to σ .

By iteratively constructing π following the above steps, we obtain a play that satisfies the strategy σ . Hence, one of the following follows:

- there exists an infinite sequence of **Env** states at some procedural level if the infinite play is due to infinite intra-procedural steps by **Env** which contradicts with T6.
- there exists an infinite sequence of entry states if the infinite play is due to infinite call steps by **Env** which contradicts with T7

Relative Completeness. Let us assume that **Prog** has a winning strategy, say σ . We show how to construct $sum(v_1, v_2, R)$, $round(v_1, v_2)$ and $descent(v_1, v_2)$ satisfying the premises of the rule by taking an arbitrary play π determined by σ .

Let $sum(v_1, v_2, R)$ be the set of all triplets (s_0, s, R) such that s is a state in π for which σ provides a universally chosen successor w.r.t. **Env**, $s_0 = entry(s)$, and R is the set of FURs in σ starting at s . Let $round(v_1, v_2)$ be the set of all pairs of states (s_1, s_2) such that s_1 and s_2 are consecutive **Env** states on the same

procedural level. Let $\text{descent}(v_1, v_2)$ be the set of all pairs of states (s_1, s_2) such that s_1 and s_2 are entry states of two consecutive procedural levels.

Since an initial state is an **Env** state, $\text{sum}(v_1, v_2, R)$ is defined for any initial state, satisfying T1.

Let us take an arbitrary summary $\text{sum}(s_0, s_1, R_1)$. σ guarantees that for every successor s_2 of s_1 wrt. **Env** there exists a successor s_3 wrt. **Prog**. For every such s_3 , $\text{sum}(s_0, s_3, R_3)$. Since the set of FURs may only shrink across intra-procedural steps, $R_3 \subseteq R_1$. In addition, we have $\text{round}(s_1, s_3)$ since s_1 and s_3 are consecutive **Env** states on the same procedural level, i.e. $\text{sum}(v_1, v_2, R)$ and $\text{round}(v_1, v_2)$ satisfy T2.

For an arbitrary **Env** state s_1 with $\text{sum}(s_0, s_1, R_1)$ and a state s_2 such that $\text{call}(s_1, s_2)$, we get $\text{sum}(s_2, s_2, R_2)$ since s_2 is an **Env** state. Since s_0 and s_2 are entry states to the caller and callee context respectively, we have $\text{descent}(s_0, s_2)$, i.e. T3 is satisfied.

Let us consider a pair of states s_1 and s_2 such that $\text{sum}(s_0, s_1, R_1)$, $\text{sum}(s_2, s_2, R_2)$, and $\text{call}(s_1, s_2)$. For any $s_3 \in R_2$, we have $\text{sum}(s_0, s_3, R_3)$ since s_3 is an **Env** state by definition of FURs, and s_0 is in the same procedural level with all states in R_1 including s_2 . It follows that any FUR of s_2 is also FUR of s_0 implying $R_2 \subseteq R_0$. In addition, we have $\text{round}(s_1, s_3)$ since s_1 and s_3 are consecutive **Env** states on the same procedural level, i.e. T4 is satisfied.

Now let us consider a state s_1 such that $\text{sum}(s_0, s_1, R_1)$ for $s_0 = \text{entry}(s_1)$ and $\text{ret}(s_1, s_2)$ for some state s_2 . By definition of FURs, we see that s_2 is in R_1 , satisfying T5.

Now we show by contradiction that $\text{round}(v_1, v_2)$ is well-founded. Assume otherwise, i.e., there exists an infinite sequence of states s_1, s_2, \dots induced by $\text{round}(v_1, v_2)$ and **Prog** still terminates. As noted previously, for each pair of consecutive **Env** states s_i and s_{i+1} there exists an intermediate sequence of state $s'_i \dots s''_i$ such that the sequence $s_1, s'_1, \dots, s''_1, s_2, \dots, s_i, s'_i, \dots, s''_i, s_{i+1}, \dots$ is a play. Since this play does not terminate, we obtain a contradiction to the assumption. Hence, we conclude that $\text{round}(v_1, v_2)$ is well-founded, satisfying T6.

Similarly, we show by contradiction that $\text{descent}(u, v)$ is well-founded, satisfying T7. \square

References

1. Albarghouthi, A., Li, Y., Gurfinkel, A., Chechik, M.: UFO: a framework for abstraction- and interpolation-based software verification. In: CAV (2012)
2. Alur, R., Chaudhuri, S.: Temporal reasoning for procedural programs. In: Barthe, G., Hermenegildo, M. (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 45–60. Springer, Heidelberg (2010)
3. Alur, R., Chaudhuri, S., Madhusudan, P.: A fixpoint calculus for local and global program flows. In: POPL, pp. 153–165 (2006)
4. Alur, R., Chaudhuri, S., Madhusudan, P.: Languages of nested trees. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 329–342. Springer, Heidelberg (2006)

5. Alur, R., Chaudhuri, S., Madhusudan, P.: Software model checking using languages of nested trees. *ACM Trans. Program. Lang. Syst.* **33**(5), 15 (2011)
6. Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. In: *POPL* (2002)
7. Beyene, T.A., Chaudhuri, S., Popeea, C., Rybalchenko, A.: A constraint-based approach to solving games on infinite graphs. In: *POPL* (2014)
8. Beyene, T.A., Popeea, C., Rybalchenko, A.: Solving existentially quantified horn clauses. In: Sharygina, N., Veith, H. (eds.) *CAV 2013*. LNCS, vol. 8044, pp. 869–882. Springer, Heidelberg (2013)
9. Beyer, D.: Second competition on software verification. In: Piterman, N., Smolka, S.A. (eds.) *TACAS 2013 (ETAPS 2013)*. LNCS, vol. 7795, pp. 594–609. Springer, Heidelberg (2013)
10. Beyer, D., Keremoglu, M.E.: CPACHECKER: a tool for configurable software verification. In: *CAV* (2011)
11. Büchi, J.R., Landweber, L.: Solving sequential conditions by finite-state strategies. *Trans. Amer. Math. Soc.* **138**, 295–311 (1969)
12. Cachat, T.: Symbolic strategy synthesis for games on pushdown graphs. In: Widmayer, P., Triguero, F., Morales, R., Hennessy, M., Eidenbenz, S., Conejo, R. (eds.) *ICALP 2002*. LNCS, vol. 2380, pp. 704–715. Springer, Heidelberg (2002)
13. Cook, B., Podolski, A., Rybalchenko, A.: Summarization for termination: no return!. *Formal Methods Syst. Design* **35**(3), 369–387 (2009)
14. de Moura, L., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) *TACAS 2008*. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
15. Grädel, E., Thomas, W., Wilke, T. (eds.): *Automata, Logics, and Infinite Games*. LNCS, vol. 2500. Springer, Heidelberg (2002)
16. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: *PLDI* (2012)
17. Griesmayer, A., Bloem, R., Cook, B.: Repair of boolean programs with an application to C. In: Ball, T., Jones, R.B. (eds.) *CAV 2006*. LNCS, vol. 4144, pp. 358–371. Springer, Heidelberg (2006)
18. Grumberg, O., Lange, M., Leucker, M., Shoham, S.: *Don't Know* in the μ -Calculus. In: Cousot, R. (ed.) *VMCAI 2005*. LNCS, vol. 3385, pp. 233–249. Springer, Heidelberg (2005)
19. Gurfinkel, A., Chechik, M.: Why waste a perfectly good abstraction? In: Hermanns, H., Palsberg, J. (eds.) *TACAS 2006*. LNCS, vol. 3920, pp. 212–226. Springer, Heidelberg (2006)
20. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: *POPL* (2004)
21. Hoder, K., Bjørner, N., de Moura, L.: μZ – an efficient engine for fixed points with constraints. In: Gopalakrishnan, G., Qadeer, S. (eds.) *CAV 2011*. LNCS, vol. 6806, pp. 457–462. Springer, Heidelberg (2011)
22. Holzbaaur, C.: *OFAI clp(q, r) Manual, 1.3.3(edn.)*. Austrian Research Institute for Artificial Intelligence, Vienna, TR-95-09 (1995)
23. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: *ICSE* (2010)
24. Jobstmann, B., Griesmayer, A., Bloem, R.: Program repair as a game. In: Etesami, K., Rajamani, S.K. (eds.) *CAV 2005*. LNCS, vol. 3576, pp. 226–238. Springer, Heidelberg (2005)
25. Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Complete functional synthesis. In: *PLDI* (2010)

26. Lustig, Y., Vardi, M.Y.: Synthesis from component libraries. *STTT* **15**(5–6), 603–618 (2013)
27. Madhusudan, P.: Synthesizing reactive programs. In: *CSL*, pp. 428–442 (2011)
28. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: *CC* (2002)
29. Piterman, N., Pnueli, A., Sa’ar, Y.: Synthesis of reactive(1) designs. In: Emerson, E.A., Namjoshi, K.S. (eds.) *VMCAI 2006*. LNCS, vol. 3855, pp. 364–380. Springer, Heidelberg (2006)
30. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: *POPL*, pp. 179–190. ACM (1989)
31. Podelski, A., Rybalchenko, A.: Transition invariants. In: *LICS* (2004)
32. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In: *Program Flow Analysis: Theory and Applications*, pp. 189–234 (1981)
33. Singh, R., Singh, R., Xu, Z., Krosnick, R., Solar-Lezama, A.: Modular synthesis of sketches using models. In: McMillan, K.L., Rival, X. (eds.) *VMCAI 2014*. LNCS, vol. 8318, pp. 395–414. Springer, Heidelberg (2014)
34. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: *ASPLOS*, pp. 404–415 (2006)
35. Srivastava, S., Gulwani, S., Foster, J.S.: From program verification to program synthesis. In: *POPL*, pp. 313–326 (2010)
36. Thomas, W.: On the synthesis of strategies in infinite games. In: *STACS*, pp. 1–13 (1995)
37. Vechev, M.T., Yahav, E., Yorsh, G.: Abstraction-guided synthesis of synchronization. In: *POPL* (2010)
38. Walukiewicz, I.: Pushdown processes: games and model-checking. *Inf. Comput.* **164**(2), 234–263 (2001)
39. Xie, Y., Aiken, A.: SATURN: A scalable framework for error detection using boolean satisfiability. *ACM TOPLAS*, **29**(3) (2007)

Verified Software: Theories, Tools, and Experiments
7th International Conference, VSTTE 2015, San
Francisco, CA, USA, July 18-19, 2015. Revised Selected
Papers
Gurfinkel, A.; Seshia, S.A. (Eds.)
2016, IX, 223 p. 41 illus. in color., Softcover
ISBN: 978-3-319-29612-8