Temporal Program Verification and Synthesis as Horn Constraints Solving

Tewodros Awgichew Beyene

Technische Universität München
Lehrstuhl für Theoretische Informatik

Temporal Program Verification and Synthesis as Horn Constraints Solving

Tewodros Awgichew Beyene

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Matthias Althoff

Prüfer der Dissertation:

1. Univ.-Prof. Dr. Andrey Rybalchenko
2. Univ.-Prof. Dr. Helmut Seidl
3. Prof. Dr. Philipp Rümmer, Universität Uppsala, Schweden

Die Dissertation wurde am 31.07.2015 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 24.11.2015 angenommen.

Abstract

We live in a world full of safety-critical software-intensive systems whose safe operation is in dire need. Program verification aims at ensuring a program satisfies a given desirable property. Temporal logics form an important class of specification languages for programs. Program verification applies proof rules to obtain proof subgoals called verification conditions whose validity implies correctness of the program. Verification conditions generally contain auxiliary assertions that must be inferred and the main challenge lies in computing such auxiliary assertions. Temporal verification of universal, i.e., valid for all computation paths, properties of various kinds of programs is a success story. The success in computing the auxiliary assertions for universal properties is due to tremendous advances in the state-of-the-art in (universal) validity checking. In recent years, several verification frameworks that are based on solvers for (universal) Horn clauses are also developed. Existential properties require that there exists a particular computation path on which a given desirable condition is valid. Advances in dealing with existential properties of programs are still not on par with the maturity of verifiers for universal properties. An important challenge is computing auxiliary assertions for existential properties.

In this dissertation, we propose a new method for solving forall-exists quantified Horn constraints extended with well-foundedness conditions. The method is based on a counterexample-guided abstraction refinement 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 Horn constraints are passed to a solver for such constraints.

We present two application domains that our method can not only formalize declaratively and elegantly but also solve efficiently. The first domain is verification of branching-time temporal properties of infinite-state programs. We propose a deductive approach to automated verification of CTL and CTL+FO properties. The practical applicability of the approach is demonstrated by an experimental evaluation on industrial examples. The second domain is computing winning strategies in two-player graph games over the state space of infinite-state programs. The winning conditions are given by safety, reachability, and general Linear Temporal Logic (LTL) properties. Our proposed approach gives a sound and relatively complete proof rule for each property class that deductively describes a winning strategy for a particular player. The practical promise of our approach is demonstrated through several case studies, including a challenging game, as well as examples derived from prior works on program synthesis and repair.

Acknowledgements

I am grateful to my supervisor Andrey Rybalchenko who has been not only a source of advice but also a source of motivation during my entire study. He has always been understanding, patient and supportive. Thank you, Andrey! I would also like to acknowledge the enormous support Corneliu Popeea has been giving me as a colleague and collaborator. All of the results presented in this dissertation are results of joint work with Andrey and Corneliu.

The research effort included in this work was supported by the German Research Council (DFG) through the doctoral program PUMA (Graduiertenkolleg Programm Und Modell-Analyse). I would like to thank DFG, PUMA and Helmut Seidl, who is the coordinator of PUMA and one of the referees for the dissertation. I am also grateful to Philipp Rümmer for his willingness to referee the dissertation as an external examiner. Javier Esparza has always been helpful in providing me with the endless list of documents required for my stay in Germany and travels abroad. Claudia Link is always willing and happy to assist me in various administrative issues. I would like to thank them and all members of the Theoretical Computer Science and Software Reliability Chair.

Julio Marino's and Manuel Carro's lectures on rigorous software development had provided me the first exposure to formal verification tools and techniques. Pedro Barahona had taught me what research in formal verification would look like. I would like to thank them all, once again, for first introducing me with core concepts of formal methods, program logics, and formal program verification.

Edengenet Mashilla, my wife and life partner, gave me unlimited and relentless support during the entire study. When I needed someone Ed was there for me, when I was down she lifted me up, and when I was up she lifted me higher. Ed's confidence in me even when I lacked self-confidence used to push me forward. If not for her, this journey would not be able to come to the end. Thank you and love you forever, *Mierafe*!

The amount of effort my parents, Awgichew Beyene and Almaz Lemma, have invested from day one to get me where I am today is beyond any words. It will not be possible to repay the debt I owe them in many, many lifetimes of full-time work. Thank you, *Gashe ena Eteye*! My brothers, Nahom, Dawit, Demeke and Daniel, and my sisters, Hezab, Roza and Alem have always been very supportive, and I would like to wholeheartedly thank them! Special thanks to Tadelle Oumer for his enormous encouragement over the years. I am indebted to Roza Awgichew, my sister, and Tesfaye Eshetu, a brother and a great friend, for their unwavering support.

The amazing friends and family at the Christ Evangelical Church have brought more joy, purpose and blessing into my stay in Munich. I can not thank them enough! It will not be possible to name them all but a few special names include Pastor Gebeyehu Feleke, Amde Aklilu, Tekeste Teweldemedhin, Saba Berhane, Yoseph Kassa, Daniel Berhanu, Yohanna Yitbarek, Tewodros Abate, Daniel Tamrat, Aman Gashaw, Hanna Abera, Bernardos Tekeste, Michael Mekonnen, Yoseph Tsegaye and Tesfaye Haregu. May God bless you all!

One of the fondly memories I will have about Munich is our Saturday morning football at Westpark. I would like to appreciate all team members for their dedication and also for their trust in making me the team captain, though some would argue that I was self-appointed.

Finally, I would like to extend my regards to everyone who have been encouraging me, supporting me, and more importantly praying for me. Above all, I glorify God for His everlasting love, forgiveness and protection on me. I thank Jesus for providing me with such helpful professors, understanding colleagues, caring family, and supportive friends. Thank You God!

To my parents Awgichew Beyene and Almaz Lemma,
and my wife Edengenet Mashilla.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

We live in a world full of safety-critical software-intensive systems whose safe operation is in dire need. Program verification aims at ensuring a program satisfies a given desirable property. Temporal logics form an important class of specification languages for programs. Solving *forall-exists quantified Horn constraints* extended with well-foundedness conditions provides a new method for the verification and synthesis of programs with respect to existential (and universal) properties.

Program verification applies rules of a program proof system to obtain logical proof subgoals called verification conditions. Their validity implies correctness of the program. The verification conditions generally contain first-order auxiliary assertions such as program invariants that must be inferred. The main challenge is to compute such auxiliary assertions.

Temporal verification of universal (i.e., valid for all computation paths) properties of various kinds of programs, e.g., procedural, multi-threaded, or functional, can be reduced to finding solutions for equations in form of universally quantified Horn clauses extended with well-foundedness conditions. Various techniques, e.g., abstract domains [43], predicate abstraction [58, 71], or interpolation [93], provide a basis for efficient tools for the verification of such properties, e.g., Astree [18], Blast [71], CPAChecker [16], SatAbs [36], Slam [8], Terminator [41], or UFO [1]. To a large extent, the success of checkers of universal properties is determined by tremendous advances in the state-of-the-art in decision procedures for (universal) validity checking, i.e., advent of tools like MathSAT [24] or Z3 [47]. In recent years, several verification frameworks that are based on solvers for Horn clauses are also developed such as Duality [94], HSF [59], SeaHorn [80], and μZ [73].

1.1 Dissertation

Existential properties require that there exists a particular computation path on which a given desirable condition is valid. An example of existential property for a program is existence of a computation path that eventually terminates. Another example is existence of a winning strategy for a particular player in a two-player game. An important challenge is dealing with existential properties, i.e., computing the corresponding auxiliary assertions in the presence of existential quantification. Advances in dealing with existential properties of programs are still not on par with the maturity of verifiers for universal properties. Nevertheless, important first steps were made in proving existence of infinite program computations, see e.g. [52, 63, 99], even in proving existential (as well as universal) CTL properties [39]. Moreover, bounded model checking tools like CBMC [35] or Klee [29] can be very effective in proving existential reachability properties. All these initial achievements inspire further, much needed research on the topic.

In this dissertation, we propose a new method for solving forall-exists quantified Horn constraints extended with well-foundedness conditions. The method is very generalized and has a potential not only to formalize declaratively and elegantly but also to solve efficiently various important application domains in formal methods. These include verification of branching-time temporal properties of infinite-state programs and computing winning strategies in two-player graph games over the state space of infinite-state programs. Our generalized method outperforms the state-of-art specialized solving methods in these application domains.

1.2 Contributions and Outline

We now describe the technical contributions and outline for the dissertation.

Solving algorithm for quantified Horn constraints We begin by describing our method of solving *forall-exists quantified Horn constraints*, which is at the heart of our approach for temporal verification and synthesis of infinite-state programs, in Chapter 3. The method has a template-based counterexample-guided abstraction refinement algorithm 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 Horn constraints are passed to a solver for such constraints.

We show correctness of the algorithm by presenting a condition under which the algorithm is sound, and by discussing how the algorithm ensures progress of refinement.
The work appeared in CAV 2013 [14].

Verification of branching-time temporal properties We describe how our solving method for quantified Horn constraints can be applied for verification of properties given in temporal logics CTL and CTL+FO in Chapter 4 and Chapter 5, respectively. We provide proof rules for generating quantified Horn constraints from a program, given as a transition system, and a temporal property, given in CTL and CTL+FO. The proof rules for temporal quantifiers for both logics are adopted from existing proof rules in [81]. However, we introduce novel yet simple and declarative proof rules for first-order quantifiers of CTL+FO. Our method for solving quantified Horn constraints helps in getting an automatic method for verifying branching-time temporal properties. As far as we know, this is the first automated method particularly for CTL+FO. We demonstrate the practical applicability of the approach for both logics by presenting experimental evaluation using examples from the PostgreSQL database server, the SoftUpdates patch system, the Windows OS kernel.
Part of the result on CTL verification appeared in CAV 2013 [14], and the result on CTL+FO *verification appeared in SPIN 2014 [11].*

Solving two-player graph games We present a Horn constraint-based approach to computing winning strategies in two-player graph games over the state space of infinite-state programs in Chapter 6. The approach handles games with winning conditions given by safety, reachability, and general Linear Temporal Logic (LTL) properties. For each property class, we give a deductive proof rule that describes a winning strategy for a particular player. We show that the proof rules are sound and relatively complete. These proof rules are our main contribution. We offer a prototype implementation of our rules on top of our algorithm for solving quantified Horn constraints, which is given in Chapter 3. The practical promise of the rules is demonstrated through several case studies, including a challenging "Cinderella-Stepmother game" that was forwarded to the community as a challenge [19, 75], as well as examples derived from prior works on program synthesis and repair.
This work appeared in POPL 2014 [12].

Solving two-player recursive graph games An approach for solving *recursive infinite-state games* where games are played on the configuration graphs of programs with recursion and unbounded data is proposed in Chapter 7. 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. We consider two kinds of recursive games: games with safety objectives and games with reachability objectives. Our contributions include sound and relatively complete proof rules for compositional, deductive synthesis using game summaries. Here also we provide a prototype implementation where a sound approximation of the proof rules can be automated using our Horn constraint solver from Chapter 3. An experimental evaluation over a set of systems code benchmarks demonstrates the practical promise of the approach.

This work appeared in VSTTE 2015 [13].

Chapter 2

Preliminaries

In this chapter, we present basic notions that are used throughout the rest of the dissertation. These include formalization of programs as transition systems, syntax and semantics of forall-exists quantified Horn constraints and games.

Programs Given a program, we abstract away from the concrete syntax of its programming language and represent the program by a transition system [81, 89]:

$$P = (\mathit{init}(v), \mathit{next}(v, v'))$$

where

- $v = (u_1, \ldots, u_n)$: A finite tuple of program variables over possibly infinite domains. A state is a valuation of v. We denote by Σ the set of all states.
- $\mathit{init}(v)$: An initial condition. This is an assertion characterizing all the initial states of the transition system. A state is called initial if it satisfies $\mathit{init}(v)$.
- $\mathit{next}(v, v')$: A transition relation. This is an assertion relating a state $s \in \Sigma$ to its successor $s' \in \Sigma$ by referring to both unprimed and primed versions of the state variables. The transition relation $\mathit{next}(v, v')$ identifies state s' as a successor of state s if $(s, s') \models \mathit{next}(v, v')$.

Let P be a transition system. A run of P is a maximal sequence of states $\sigma : s_0, s_1, \ldots$ satisfying the following requirements.

- Initiality: s_0 is initial
- Consecution: For each $j + 1 \in [1..|\sigma|)$, s_{j+1} is a successor of s_j.

The sequence σ is maximal if either σ is infinite, or $\sigma = s_0, \ldots, s_k$ and s_k has no successor with respect to $next(v, v')$. We denote by $runs(P)$ the set of runs of P. An infinite run of P is called a computation. The set of computations of P starting in a state s is denoted by $\Pi_P(s)$.

Program verification and synthesis Program verification is the task of automatically generating proofs for a program's compliance with a given desirable property. If the program does not comply with the desired property, then the program verification task will be to generate a counter-example. There are two important classes of properties: safety properties, which require the absence of "bad" states in each program computation, and reachability properties, which require that some "good" states are reached in every computation. Program synthesis is the task of automatically generating a program that meets a given desirable property. We can view both program verification and program synthesis as search problems where the former is a search for proofs and the later is a search for programs.

Constraints Let $\mathcal{T}$ be a first-order theory in a given signature and $\models_{\mathcal{T}}$ be the entailment relation for $\mathcal{T}$. We write $v, v_0, v_1, \ldots$ and w to denote non-empty tuples of variables. We refer to a formula $c(v)$ over variables v from $\mathcal{T}$ as a constraint. Let *false* and *true* be an unsatisfiable and a valid constraint, respectively.

For example, let x, y, and z be variables. Then, $v = (x, y)$ and $w = (y, z)$ are tuples of variables. $x \leq 2$, $y \leq 1 \wedge x - y \leq 0$, and $f(x) + g(x, y) \leq 3 \vee z \leq 0$ are example constraints in the theory $\mathcal{T}$ of linear inequalities and uninterpreted functions, where f and g are uninterpreted function symbols. $y \leq 1 \wedge x - y \leq 0 \models_{\mathcal{T}} x \leq 2$ is an example of a valid entailment.

Well-founded relations A binary relation $\varphi(v, v')$ is well-founded if it does not admit any infinite (ascending or descending) chains. For example, the relation $x \geq 0 \wedge x' \leq x - 1$ is well-founded.

Disjunctively well-founded relations A binary relation $\varphi(v, v')$ is disjunctively well-founded if it is included in a finite union of well-founded relations [105], i.e., if there exist well-founded $\varphi_1(v, v'), \ldots, \varphi_n(v, v')$ such that $\varphi(v, v') \models_{\mathcal{T}} \varphi_1(v, v') \vee \cdots \vee \varphi_n(v, v')$. For example, the relation $(x \geq 0 \wedge x' \leq x - 1) \vee (y \leq 0 \wedge y' \geq y + 1)$ is disjunctively well-founded.

Queries and dwf-predicates We assume a set of uninterpreted predicate symbols $\mathcal{Q}$ that we refer to as query symbols. The arity of a query symbol is encoded in its name. We write q to denote a query symbol. Given q of a non-zero arity n and a tuple of variables v of length n, we define $q(v)$ to be a query. Furthermore, we introduce an interpreted predicate symbol *dwf* of arity one (*dwf* stands for disjunctive well-foundedness). Given a query $q(v, v')$ over tuples of variables with equal length, we refer to $dwf(q)$ as a *dwf*-predicate. For example, let $\mathcal{Q} = \{r, s\}$ be query symbols of arity one and two, respectively. Then, $r(x)$ and $s(x, y)$ are queries, and $dwf(s)$ is a *dwf*-predicate.

Forall-exists quantified Horn constraints Let $h(v)$ range over queries over v, constraints over v, and existentially quantified conjunctions of queries and constraints with free variables in v. We define a forall-exists quantified Horn constraint to be either an implication $c(v_0) \wedge q_1(v_1) \wedge \cdots \wedge q_n(v_n) \rightarrow h(v)$ or a unit clause $dwf(q)$. The left-hand side of the implication is called the body, written as $body(v)$, and the right-hand side is called the head.

Formally, a forall-exists quantified Horn constraint ($\forall\exists HC$) over a first-order theory $\mathcal{T}$ is constructed as follows:

$$
\begin{aligned}
\forall\exists HC ::= \;& \forall v : conj(v) \rightarrow \exists w : conj(v, w) \mid \\
& conj(v) \rightarrow \phi(v) \mid conj(v) \rightarrow q(v) \mid dwf(q) \\
conj ::= \;& \phi(v) \mid q(v) \mid conj \wedge conj \\
v, w ::= \;& \text{a tuple of distinct variables such that } v \cap w = \emptyset \\
q(v) ::= \;& \text{an uninterpreted predicate symbol applied to a tuple of variables} \\
\phi(v) ::= \;& \text{a formula whose terms and predicates are interpreted over } \mathcal{T}
\end{aligned}
$$

As an example, we give a set of forall-exists quantified Horn constraints below:

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

These clauses represent an assertion over the interpretation of predicate symbols *rank* and *ti*.

In this work, we quantify explicitly only existentially quantified variables and leave universal quantification implicit, i.e., variables that are not quantified in a given constraint are assumed to be universally quantified. This way, we can the example set of Horn

constraints given above as follows:

$$x \geq 0 \to \exists y : x \geq y \wedge \mathit{rank}(x, y),$$
$$\mathit{rank}(x, y) \to \mathit{ti}(x, y),$$
$$\mathit{ti}(x, y) \wedge \mathit{rank}(y, z) \to \mathit{ti}(x, z),$$
$$\mathit{dwf}(\mathit{ti}).$$

A Horn constraint given as $c(v_0) \wedge q_1(v_1) \wedge \cdots \wedge q_n(v_n) \to h(v)$ is said to be an inference clause if its head $h(v)$ contains at least one uninterpreted predicate symbol. For example, the Horn constraint below is an inference clause.

$$\mathit{ti}(x, y) \wedge \mathit{rank}(y, z) \to \mathit{ti}(x, z)$$

Semantics of forall-exists quantified Horn constraints A set of Horn constraints can be seen as an assertion over the queries that occur in the constraints.

We consider a function *ClauseSol* that maps each query $q(v)$ occurring in a given set of Horn constraints into a constraint over v. Such a function is called a solution if the following two conditions hold. First, for each Horn constraints of the form $\mathit{body}(v) \to \mathit{head}(v)$ from the given set we require that replacing each query by the corresponding constraint assigned by *ClauseSol* results in a valid entailment. That is, we require $\mathit{body}(v)\mathit{ClauseSol} \models_{\mathcal{T}} \mathit{head}(v)\mathit{ClauseSol}$, where the juxtaposition represents application of substitution. Second, for each clause of the form $\mathit{dwf}(q)$ we require that the constraint assigned by *ClauseSol* to q is a disjunctively well-founded relation. Let $\models_{\mathcal{Q}}$ be the corresponding satisfaction relation, i.e., $\mathit{ClauseSol} \models_{\mathcal{Q}} \mathit{Clauses}$ if *ClauseSol* is a solution for the given set of clauses.

For example, the previously presented set of clauses, say *Clauses*, has a solution *ClauseSol* such that $\mathit{ClauseSol}(\mathit{rank}(x, y)) = \mathit{ClauseSol}(\mathit{ti}(x, y)) = (x \geq 0 \wedge y \geq x - 1)$. To check $\mathit{ClauseSol} \models_{\mathcal{Q}} \mathit{Clauses}$ we consider the validity of the following implications:

$$x \geq 0 \to \exists y : x \geq y \wedge x \geq 0 \wedge y \leq x - 1,$$
$$x \geq 0 \wedge y \leq x - 1 \to x \geq 0 \wedge y \leq x - 1,$$
$$x \geq 0 \wedge y \leq x - 1 \wedge y \geq 0 \wedge z \leq y - 1 \to x \geq 0 \wedge z \leq x - 1.$$

and the fact that $\mathit{ClauseSol}(\mathit{ti}(x, y)) = (x \geq 0 \wedge y \leq x - 1)$ is a (disjunctively) well-founded relation.

The HSF solver Let us assume we have a set of forall-exists quantified Horn constraints. If none of these Horn constraints has a head with existentially quantified variable, i.e., for each Horn constraint all of the variables are universally quantified, we apply the solver called HSF [59] to find solutions for the set of Horn constraints. On termination, the HSF solver may return a solution when the set of Horn clauses is satisfiable or a counter-example otherwise. But, the solver may not terminate at all.

Games A *(two-player, turn-based, graph) game* is a pair consisting of a transition system and a *winning condition*:

- We consider transition systems that are composed from two players, Adam and Eve. Let v be a tuple of variables. For simplicity, we do not distinguish between variables controlled by Eve and Adam, i.e., both players control all variables.

  We represent the initial states of the transition system by an assertion $init(v)$. The transition relations of Adam and Eve are given by assertions $adam(v, v')$ and $eve(v, v')$, respectively.

- A *winning condition obj* for a game is given by a set of infinite sequences of system states. A game is said to be a safety game, a reachability game, and an LTL game, respectively, when its winning condition is a safety property, a reachability property, and a general LTL property.

Games semantics We present the semantics of games in two steps. First, we define strategies of the individual players. A *strategy* σ for Eve is a set of infinite trees over the states of the system that satisfies the following conditions:

- The roots of trees in σ coincide with the set of initial states, and are considered to be on the first level of the trees. Here, the level of a node is the length of the path to the root plus one.

- The set of successors of each tree node s at an odd level consists of the following set of states.

$$\{s' \mid (s, s') \models adam(v, v')\}$$

- The set of successors of each tree node s at an even level consists of a non-empty subset of the following set of states.

$$\{s' \mid (s, s') \models eve(v, v')\}$$

Thus, a strategy for Eve alternates between universal choices of Adam and existential choices of Eve. We call each infinite sequence of system states that starts at a root of a strategy σ and follows some branch a *play* π determined by σ.

A strategy σ for Eve is *winning* if every play determined by σ is included in the winning condition.

For the given system and a formula φ that describes a winning condition in some temporal logic, we write

$$(init(v), eve(v, v'), adam(v, v')) \models \varphi$$

when Eve has a winning strategy.

We also consider Adam's perspective. A strategy σ for Adam is defined in a similar way. The roots of σ represent a non-empty subset of $init(v)$. σ alternates between existential choices of Adam and universal choices of Eve. If a tree node s is on an odd level, then its successors form a non-empty subset of $\{s' \mid (s, s') \models adam(v, v')\}$. Otherwise, the set of successors is $\{s' \mid (s, s') \models eve(v, v')\}$.

Chapter 3

Solving Forall-Exists Quantified Horn Constraints

3.1 Introduction

Temporal verification of universal, i.e., valid for all computation paths, properties of various kinds of programs is a success story. Various techniques, e.g., abstract domains [43], predicate abstraction [58, 71], or interpolation [93], provide a basis for efficient tools for the verification of such properties, e.g., Astree [18], Blast [71], CPAChecker [16], SatAbs [36], Slam [8], Terminator [41], or UFO [1]. To a large extent, the success of checkers of universal properties is determined by tremendous advances in the state-of-the-art in decision procedures for (universal) validity checking, i.e., advent of tools like MathSAT [24] or Z3 [47].

In contrast, advances in dealing with existential properties of programs, e.g., proving whether there exists a particular computation path, are still not on par with the maturity of verifiers for universal properties. Nevertheless, important first steps were made in proving existence of infinite program computations, see e.g. [52, 63, 99], even in proving existential (as well as universal) CTL properties [39]. Moreover, bounded model checking tools like CBMC [35] or Klee [29] can be very effective in proving existential reachability properties. All these initial achievements inspire further, much needed research on the topic.

In this chapter, we present a method that can serve as a further building block for the verification of temporal existential (and universal) properties of infinite-state systems. Our method solves forall-exists quantified Horn clauses extended with well-foundedness conditions. The conclusion part of such clauses may contain existentially quantified

variables. The main motivation for the development of our method stems from an observation that verification conditions for existential temporal properties, e.g., generated by a deductive proof system for CTL [81], can be expressed by clauses in such form.

Our method, called E-HSF, applies a counterexample-guided refinement 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 for such clauses. At this step, we can benefit from emergent tools in the area of solving Horn clauses over decidable theories, e.g., HSF [59], μZ [73], or Duality [94]. 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. E-HSF turns the counterexample into an additional constraint on the set of witness candidates, and continues with the next iteration of the refinement loop. Notably, our refinement loop conjoins constraints that are obtained for all discovered counterexamples. This way E-HSF guarantees that previously handled counterexamples are not rediscovered and that a wrong choice of witnesses can be mended.

3.2 Solving algorithm

In this section, we present our algorithm E-HSF for solving constraints in form of Horn clauses that contain existential quantification and well-foundedness conditions. We describe the algorithm first and then state its correctness properties. Section 7.2 can be seen for a detailed example of applying E-HSF.

Our solving method proceeds in two steps. First, we rely on Skolemization to reformulate the problem of dealing with existential quantification as a problem of finding witnesses for the existentially quantified variables. Such witnesses are represented by Skolem relations, which is a slight generalisation of Skolem functions that is convenient in our setting. Given a forall-exists quantified Horn constraint $body(v) \rightarrow \exists w : head(v, w)$, the corresponding Skolem relation $rel(v, w)$ determines which value w satisfies $head(v, w)$ for a given value v. Since for each value v such that $body(v)$ holds we need a value w, we require that such v is in the domain of the Skolem relation. We represent the domain of Skolem relation $rel(v, w)$ as the guard $grd(v)$, and will use it later to implement the above requirement.

A function SKOLEMIZE shown in Figure 3.1 implements the Skolemization step. It outputs a set of clauses without existential quantification, yet containing Skolem relations

function SKOLEMIZE
input
Clauses - set of clauses
begin
1: $Skolemized := Parent := Rels := Grds := \emptyset$
2: **for each** $clause \in Clauses$ **do**
3: **match** *clause* **with**
4: | $body(v) \rightarrow \exists w : \bigwedge_{i=1}^{n} conj_i(v, w)$ ->
5: $rel, grd :=$ fresh predicate symbols of arity $\mathbf{v} + \mathbf{w}$ and $\mathbf{v}$, resp.
6: $Parent := \{(grd, clause),\ (rel, clause)\} \cup Parent$
7: $Rels := \{rel\} \cup Rels$
8: $Grds := \{grd\} \cup Grds$
9: $Skolemized := \{body(v) \wedge rel(v, w) \rightarrow conj_i(v, w) \mid i \in 1..n\} \cup$
$\{body(v) \rightarrow grd(v)\} \cup Skolemized$
10: | _ -> $Skolemized := \{clause\} \cup Skolemized$
11: **done**
12: **return** $(Skolemized, Parent, Rels, Grds)$
13: **end**

FIGURE 3.1: The function SKOLEMIZE.

and guards. Furthermore, SKOLEMIZE keeps track of which Skolem relations and guards belong to which clauses.

The second step takes as input a set of Skolemized clauses produced by SKOLEMIZE and either finds a solution, returns that no solution can be found, or diverges. At this step we rely on a set of templates that determine the search space for Skolem relations, their guards, as well as termination arguments used for dealing with well-foundedness. In order to ensure that the guard of a Skolem relation entails its domain, we assume that the guard template implies the projection of the Skolem relation template. Formally, we require that the template functions GRDT and RELT providing guard and Skolem relation templates for the output of SKOLEMIZE satisfy the following condition: for each $grd \in Grds$ and $rel \in Rels$ such that $Parent(grd) = Parent(rel)$ the implication

$$\text{GRDT}(grd)(v) \rightarrow \exists w : \text{RELT}(rel)(v, w) \tag{3.1}$$

is valid (for arbitrary values of template parameters). We establish Equation 3.1 by choosing templates accordingly.

See Figure 3.2. The solving process iteratively determines appropriate candidates for Skolem relations and their guards by using a counterexample driven approach. Each counterexample induces constraints on template parameters and thus rules out failed attempts. Given candidates for Skolem relations and their guards, we record these candidates by introducing appropriate Horn clauses called *Defs*. Then, we apply the solver for (ordinary) Horn clauses, which is called HSF, on the set of Skolemized clauses that is extended with *Defs*. If HSF finds a solution, then we report it as a solution for

```
function E-HSF
 input
  Clauses - set of clauses
 global
  Defs - the set of skolem relations
  Grds - the set of skolem guards
  Constraint - the global constraints
 local
  function SKOLEMIZE
  procedure DEFSREFINE
 begin
1:  Skolemized, Parent, Rels, Grds := SKOLEMIZE(Clauses)
2:  Constraint := true
3:  Defs := {true → rel(v, w) | rel ∈ Rels} ∪ {grd(v) → true | grd ∈ Grds}
4:  match HSF(Skolemized ∪ Defs) with
5:   | solution ClauseSol ->
6:    return "solution ClauseSol"
7:   | error derivation Cex and symbol map SYM ->
8:    CexDefs := {(body → q(...)) ∈ Cex | SYM(q) ∈ Rels ∪ Grds}
9:    if CexDefs = ∅ then
10:     return "error derivation Cex and symbol map SYM"
11:   else
12:     DEFSREFINE(Cex, SYM, CexDefs)
13:     goto 4
 end
```

FIGURE 3.2: The algorithm E-HSF.

the original set of clauses. Otherwise, we inspect a counterexample given by HSF. Such a counterexample is presented by a set of recursion-free Horn clauses which uses a form of Static Single Assignment (SSA) to represent an unfolding of $Skolemized \cup Defs$ that cannot be satisfied.

If the counterexample does not involve any Skolem relations or their guards, then we report that *Skolemized* cannot be satisfied. Otherwise, the unfolding is not satisfiable either because there are no Skolem relations together with guards that make *Skolemized* satisfiable, or because the currently chosen candidates are not correct.

To find out, we call the procedure DEFSREFINE in Figure 3.3, which tries to find new candidates for Skolem relations together with their guards that make *Skolemized* satisfiable by eliminating the current counter-example. The procedure first applies the function RESOLVE that does resolution over the set of clauses *Cex* by excluding the definitions *Defs* for the chosen candidates (in Line 1). The resulting clause may, therefore, contain Skolem relations and guards. Each Skolem relation in the body of the clause is replaced by its corresponding template (in Line 2). Similarly, the head of the clause is replaced by bound and decrease templates if the head is a well-foundedness relation (in

procedure DefsRefine
input
Cex - a set of clauses involved in the counterexample derivation
Sym - symbol map
CexDefs - candidate definitions involved in the counterexample
local
function Resolve
function EncodeValidity
function SmtSolve
begin
1: $(body \wedge \bigwedge_{i=1}^{n} q_i(v_i, w_i) \rightarrow head) \ := \ \textsc{Resolve}(Cex \setminus CexDefs)$
2: $body \ := \ body \wedge \bigwedge_{i=1}^{n} \textsc{RelT}(\textsc{Sym}(q_i))(v_i, w_i)$
3: **match** *head* **with**
4: | $q(v, w)$ **when** $dwf(\textsc{Sym}(q)) \in Clauses$ ->
5: $head \ := \ \textsc{BoundT}(\textsc{Sym}(q))(v) \wedge \textsc{DecreaseT}(\textsc{Sym}(q))(v, w)$
6: | $q(v)$ **when** $\textsc{Sym}(q) \in Grds$ ->
7: $head \ := \ \textsc{GrdT}(\textsc{Sym}(q))(v)$
8: | _ -> **skip**
9: $Constraint \ := \ \textsc{EncodeValidity}(body \rightarrow head) \wedge Constraint$
10: **match** $\textsc{SmtSolve}(Constraint)$ **with**
11: | solution *CexSol* ->
12: $Defs \ := \ \{\textsc{RelT}(rel)(v, w)\, CexSol \rightarrow rel(v, w) \mid rel \in Rels\} \cup$
$\{grd(v) \rightarrow \textsc{GrdT}(grd)(v)\, CexSol \mid grd \in Grds\}$
13: | _ -> **return** "error derivation *Cex* and symbol map Sym"
end

Figure 3.3: The procedure DefsRefine refines Skolem definitions.

Lines 4-5) or by a guard template if the head is a guard relation (in Lines 6-7). Otherwise, the head is assumed to be a background theory constraint and it will be left as it is (in Line 8). The procedure then applies EncodeValidity to create a formula whose free variables are template parameters of the clause (in Line 9). The formula encodes a constraint over template parameters. We consider a conjunction of such constraints, which is stored as *Constraint*, across all iteration of the solving process, thus ensuring that previously analysed and eliminated counterexamples will not re-appear. Then SmtSolve (in Line 10) returns an assignment of constants to template parameters provided its argument is satisfiable. A solution of *Constraint* determines new candidates, which we formally record using the set of clauses *Defs*. (in Line 12) Now our iteration is ready to go in the next round. However, if *Constraint* is not satisfiable, we return the set of clauses *Cex* as a counter-example and report that *Skolemized* cannot be satisfied (in Line 13).

Correctness The algorithm E-HSF relies on the following propositions. First, the Skolemization step preserves equi-satisfiability under an assumption that each guard needs to be a subset of the corresponding Skolem relation.

Lemma 1 (Skolemization preserves satisfiability)**.** The set of clauses *Clauses* is equi-satisfiable with the set of clauses computed by SKOLEMIZE when domains of Skolem relations contain corresponding guards. Formally, *Clauses* is equi-satisfiable with the set

$$\{grd(v) \rightarrow \exists w : rel(v, w) \mid grd \in Grds \wedge rel \in Rels \wedge Parent(grd) = Parent(rel)\} \cup Skolemized \ .$$

Proof. (Sketch) Let $clause = (body(v) \rightarrow \exists w : q(v, w))$ and $Parent(rel) = Parent(grd) = clause$. We keep pairs (v^*, w^*) such that $body(v^*)$ and $q(v^*, w^*)$ hold in a relation *rel* while storing v^* in *grd*. Then the statement of the lemma follows immediately. □

As previously mentioned, the above relation between Skolem relations and their guards is established by the appropriate choice of RELT and GRDT functions, see Equation 3.1. Then E-HSF inherits its soundness from HSF.

Theorem 1 (Soundness)**.** If HSF is sound, i.e., it returns solutions for given sets of clauses, and if Equation 3.1 holds for each $grd \in Grds$ and $rel \in Rels$ such that $Parent(grd) = Parent(rel)$, then, upon termination, E-HSF returns a solution for *Clauses*.

Proof. Let *ClauseSol* be a result of applying HSF in line 4 of Figure 3.2. The first assumption of the theorem statement guarantees that *ClauseSol* satisfies *Skolemized*. The first assumption ensures that Lemma 1 is applicable, hence, *ClauseSol* satisfies *Clauses*. □

Our method is based on a counterexample guided scheme for discovery of Skolem relations and guards. While this scheme has successful applications in practice, it does not guarantee termination of the refinement process when the set of candidates for Skolem relations and guards is unbounded. Our method necessarily inherits this limitation.

Despite this undecidability imposed limitation, our method strives at achieving termination of refinement process in practice. An important ingredient is provided by the fact that *Constraint* keeps track of the conjunction of constraints used to discover candidates Skolem relations and guards across all iterations.

Theorem 2 (Progress of refinement)**.** E-HSF does not consider any error derivation (counter-example) more than once.

Proof. (Sketch) The progress of refinement property follows directly from the observation that every solution for *Constraint* yields Skolem relations and guards that satisfy each previously discovered error derivation. □

3.3 Solving illustration

We consider the following set *Clauses* that encodes a check whether a program with the variables $v = (x, y)$, an initial condition $init(v) = (y \geq 1)$ and a transition relation $next(v, v') = (x' = x + y)$ satisfies a CTL property $EF\ dst(v)$, where $dst(v) = (x \geq 0)$.

$$
\begin{aligned}
&init(v) \rightarrow inv(v),\\
&inv(v) \wedge \neg dst(v) \rightarrow \exists v' : next(v, v') \wedge inv(v') \wedge rank(v, v'),\\
&rank(v, v') \rightarrow ti(v, v'),\\
&ti(v, v') \wedge rank(v', v'') \rightarrow ti(v, v''),\\
&dwf(ti).
\end{aligned}
$$

Here, $inv(v)$, $rank(v, v')$, and $ti(v, v')$ are unknown predicates that we need to solve for. The predicate $inv(v)$ corresponds to states reachable during program execution, while the last three clauses ensures that $rank(v, v')$ is a well-founded relation [105].

We start the execution of E-HSF from Figure 3.2 by applying Skolemize to eliminate the existential quantification. As a result, the clause that contains existential quantification is replaced by the following four clauses that contain an application of a Skolem relation $rel(v, v')$ introduced by Skolemize as well as an introduction of a lower bound on the guard $grd(v)$ of the Skolem relation:

$$
\begin{aligned}
&inv(v) \wedge \neg dst(v) \wedge rel(v, v') \rightarrow next(v, v'),\\
&inv(v) \wedge \neg dst(v) \wedge rel(v, v') \rightarrow inv(v'),\\
&inv(v) \wedge \neg dst(v) \wedge rel(v, v') \rightarrow rank(v, v'),\\
&inv(v) \wedge \neg dst(v) \rightarrow grd(v).
\end{aligned}
$$

Furthermore, this introduction is recorded as $Rels = \{rel\}$ and $Grds = \{grd\}$. Note that we replaced a conjunction in the head of a clause by a conjunction of corresponding clauses.

First candidate for Skolem relation Next, we proceed with the execution of E-HSF. We initialise *Constraint* with the assertion *true*. Then, we generate a set of Horn clauses *Defs* that provides initial candidates for the Skolem relation and its guard as follows: $Defs = \{true \rightarrow rel(v, v'),\ grd(v) \rightarrow true\}$. Now, we apply the solving algorithm HSF for quantifier free Horn clauses on the set of clauses that contains the result of Skolemization and the initial candidates in *Defs*, i.e., we give to HSF the

following clauses:

$$
\begin{aligned}
&\mathit{init}(v) \to \mathit{inv}(v),\\
&\mathit{inv}(v) \wedge \neg \mathit{dst}(v) \wedge \mathit{rel}(v, v') \to \mathit{next}(v, v'),\\
&\mathit{inv}(v) \wedge \neg \mathit{dst}(v) \wedge \mathit{rel}(v, v') \to \mathit{inv}(v'),\\
&\mathit{inv}(v) \wedge \neg \mathit{dst}(v) \wedge \mathit{rel}(v, v') \to \mathit{rank}(v, v'),\\
&\mathit{inv}(v) \wedge \neg \mathit{dst}(v) \to \mathit{grd}(v),\\
&\mathit{rank}(v, v') \to \mathit{ti}(v, v'),\\
&\mathit{ti}(v, v') \wedge \mathit{rank}(v', v'') \to \mathit{ti}(v, v''),\\
&\mathit{dwf}(\mathit{ti}),\\
&\mathit{true} \to \mathit{rel}(v, v'),\\
&\mathit{grd}(v) \to \mathit{true}.
\end{aligned}
$$

HSF returns an error derivation that witnesses a violation of the given set of clauses. This derivation represents an unfolding of clauses in $\mathit{Skolemized} \cup \mathit{Defs}$ that yields a relation for $\mathit{ti}(v, v')$ that is not disjunctively well-founded. To represent the unfolding, HSF uses a form of static single assignment (SSA) that is applied to predicate symbols, where each unfolding step introduces a fresh predicate symbol that is recorded by the function Sym. We obtain the clauses *Cex* consisting of

$$
\begin{aligned}
&\mathit{init}(v) \to q_1(v),\\
&q_1(v) \wedge \neg \mathit{dst}(v) \wedge q_2(v, v') \to \mathit{next}(v, v'),\\
&\mathit{true} \to q_2(v, v').
\end{aligned}
$$

together with the following bookkeeping of the SSA renaming: $\textsc{Sym}(q_1) = \mathit{inv}$ and $\textsc{Sym}(q_2) = \mathit{rel}$. From *Cex* we extract the clause *CexDefs* that provides the candidate for the Skolem relation. We obtain $\mathit{CexDefs} = \{\mathit{true} \to q_2(v, v')\}$, since $\textsc{Sym}(q_2) = \mathit{rel}$ and hence $\textsc{Sym}(q_2) \in \mathit{Rels}$.

We analyse the counterexample clauses by applying resolution on $\mathit{Cex} \setminus \mathit{CexDefs}$. The corresponding resolution tree is shown below (literals selected for resolution are boxed):

$$
\frac{\mathit{init}(v) \to \boxed{q_1(v)} \qquad \boxed{q_1(v)} \wedge \neg \mathit{dst}(v) \wedge q_2(v, v') \to \mathit{next}(v, v')}{\mathit{init}(v) \wedge \neg \mathit{dst}(v) \wedge q_2(v, v') \to \mathit{next}(v, v')}
$$

Note that $q_2(v, v')$ was not eliminated, since the clause $\mathit{true} \to q_2(v, v')$ was not given to Resolve as input. The result of applying Resolve is the clause $\mathit{init}(v) \wedge \neg \mathit{dst}(v) \wedge q_2(v, v') \to \mathit{next}(v, v')$. We assign the conjunction $\mathit{init}(v) \wedge \neg \mathit{dst}(v)$ to *body* and $\mathit{next}(v, v')$ to *head*, respectively.

Now we iterate i through the singleton set $\{1\}$, which is determined by the fact that the above clause contains only one unknown predicate on the left-hand side. We apply RELT on $\text{SYM}(q_2)$ and set the free variables in the result to (v, v'). This yields a template $v' = Tv + t$ for the Skolem relation $rel(v, v')$. Here, T is a matrix of unknown coefficients $\left(\begin{smallmatrix} t_{xx} & t_{xy} \\ t_{yx} & t_{yy} \end{smallmatrix}\right)$, and t is a vector of unknown free coefficient (t_x, t_y). In other words, our template represents a conjunction of two equality predicates $x' = t_{xx}x + t_{xy}y + t_x$ and $y' = t_{yx}x + t_{yy}y + t_y$. We conjoin this template with *body* and obtain $body = (v' = Tv + t \wedge init(v) \wedge \neg dst(v))$. Since *head* is not required to be disjunctively well-founded, E-HSF proceeds with the generation of constraints over template parameters.

We apply ENCODEVALIDITY on the following implication:

$$x' = t_{xx}x + t_{xy}y + t_x \wedge y' = t_{yx}x + t_{yy}y + t_y \wedge y \geq 1 \wedge \neg x \geq 0 \rightarrow x' = x + y \ .$$

This implication is valid if the following constraint returned by ENCODEVALIDITY is satisfiable.

$$\exists \overbrace{\lambda_1, \lambda_2, \lambda_3, \lambda_4}^{\lambda}, \overbrace{\mu_1, \mu_2, \mu_3, \mu_4}^{\mu} : \lambda_3 \geq 0 \wedge \lambda_4 \geq 0 \wedge \mu_3 \geq 0 \wedge \mu_4 \geq 0 \wedge$$
$$\begin{pmatrix} \lambda \\ \mu \end{pmatrix} \begin{pmatrix} t_{xx} & t_{xy} & -1 & 0 \\ t_{yx} & t_{yy} & 0 & -1 \\ 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} -1 & -1 & 1 & 0 \\ 1 & 1 & -1 & 0 \end{pmatrix} \wedge \begin{pmatrix} \lambda \\ \mu \end{pmatrix} \begin{pmatrix} -t_x \\ -t_y \\ -1 \\ -1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

This constraint requires that the right-hand side on the implication is obtained as a linear combination of the (in)equalities on the left-hand side of the implication. We conjoin the above constraint with *Constraint*.

We apply an SMT solver to compute a satisfying valuation of template parameters occurring in *Constraint* and obtain:

t_{xx}	t_{xy}	t_x	t_{yx}	t_{yy}	t_y
1	1	0	0	0	10

By applying *CexSol* on the template $v' = Tv + t$, which is the result of $\text{RELT}(rel)(v, v')$, we obtain the conjunction $x' = x + y \wedge y' = 10$. In this example, we assume that the template $\text{GRDT}(grd)(v)$ is equal to *true*. Hence, we modify the clauses that record the current candidate for $rel(v, v')$ and $grd(v)$ as follows:

$$Defs = \{x' = x + y \wedge y' = 10 \rightarrow rel(v, v'),\ grd(v) \rightarrow true\}$$

Now we proceed with the next iteration of the main loop in E-HSF.

Second candidate for Skolem relation The second iteration in E-HSF uses *Defs* and *Constraint* as determined during the first iteration. We apply HSF on *Skolemized* $\cup$ *Defs* and obtain an error derivation *Cex* consisting of the clauses

$$\begin{aligned}
&\mathit{init}(v) \to q_1(v),\\
&q_1(v) \wedge \neg \mathit{dst}(v) \wedge q_2(v,v') \to q_3(v,v'),\\
&x' = x + y \wedge y' = 10 \to q_2(v,v'),\\
&q_3(v,v') \to q_4(v,v'),
\end{aligned}$$

together with the function Sym such that $\textsc{Sym}(q_1) = \mathit{inv}$, $\textsc{Sym}(q_2) = \mathit{rel}$, $\textsc{Sym}(q_3) = \mathit{rank}$, and $\textsc{Sym}(q_4) = \mathit{ti}$. From *Cex* we extract $\mathit{CexDefs} = \{x' = x + y \wedge y' = 10 \to q_2(v,v')\}$ since $\textsc{Sym}(q_2) \in \mathit{Rels}$. We apply Resolve on $\mathit{Cex} \setminus \mathit{CexDefs}$ and obtain:

$$\mathit{init}(v) \wedge \neg \mathit{dst}(v) \wedge q_2(v,v') \to q_4(v,v') \ .$$

As seen at the first iteration, we have $\textsc{RelT}(\mathit{rel})(v,v') = (v' = Tv + t)$. Hence we have $\mathit{body} = (\mathit{init}(v) \wedge \neg \mathit{dst}(v) \wedge v' = Tv + t)$.

Since $\textsc{Sym}(q_4) = \mathit{ti}$ and $\mathit{dwf}(\mathit{ti}) \in \mathit{Skolemized}$, the error derivation witnesses a violation of disjunctive well-foundedness. Hence, by applying BoundT and DecreaseT we construct templates $\mathit{bound}(v)$ and $\mathit{decrease}(v,v')$ corresponding to a bound and decrease condition over the program variables, respectively.

$$\begin{aligned}
\mathit{bound}(v) &= (r_x x + r_y y \geq r_0) \ ,\\
\mathit{decrease}(v,v') &= (r_x x' + r_y y' \leq r_x x + r_y y - 1) \ .
\end{aligned}$$

Finally, we set *head* to the conjunction $r_x x + r_y y \geq r_0 \wedge r_x x' + r_y y' \leq r_x x + r_y y - 1$.

By EncodeValidity on the implication $\mathit{body} \to \mathit{head}$ we obtain the constraint

$$\exists \overbrace{\lambda_1, \lambda_2, \lambda_3, \lambda_4}^{\lambda}, \overbrace{\mu_1, \mu_2, \mu_3, \mu_4}^{\mu} : \lambda_3 \geq 0 \wedge \lambda_4 \geq 0 \wedge \mu_3 \geq 0 \wedge \mu_4 \geq 0 \wedge$$
$$\begin{pmatrix}\lambda \\ \mu\end{pmatrix} \begin{pmatrix} t_{xx} & t_{xy} & -1 & 0 \\ t_{yx} & t_{yy} & 0 & -1 \\ 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} -r_x & -r_y & 0 & 0 \\ -r_x & -r_y & r_x & r_y \end{pmatrix} \wedge \begin{pmatrix}\lambda \\ \mu\end{pmatrix} \begin{pmatrix} -t_x \\ -t_y \\ -1 \\ -1 \end{pmatrix} = \begin{pmatrix} -r_0 \\ -1 \end{pmatrix} .$$

We add the above constraint as an additional conjunct to *Constraint*. That is, *Constraint* is strengthened during each iteration.

We apply the SMT solver to compute a valuation template parameters that satisfies *Constraint*. We obtain the following solution *CexSol*:

t_{xx}	t_{xy}	t_x	t_{yx}	t_{yy}	t_y
1	0	1	0	0	1

The corresponding values of r and r_0 are $(-1, 0)$ and -1, which lead to the bound $-x \geq 1$ and the decrease relation $-x' \leq -x - 1$. By applying *CexSol* on the template $v' = Tv + t$ we obtain the conjunction $x' = x + 1 \wedge y' = 1$. Note that the solution for $rel(v, v')$ obtained at this iteration is not compatible with the solution obtained at the first iteration, i.e., the intersection of the respective Skolem relations is empty. Finally, we modify *Defs* according to *CexSol* and obtain:

$$Defs = \{x' = x + 1 \wedge y' = 1 \rightarrow rel(v, v'),\ grd \rightarrow true\}$$

Now we proceed with the next iteration of the main loop in E-HSF. At this iteration the application of HSF returns a solution *ClauseSol* such that

$$\begin{aligned} ClauseSol(inv(v)) &= (y \geq 1)\ , \\ ClauseSol(rel(v)) &= (x' = x + 1 \wedge y' = 1)\ , \\ ClauseSol(rank(v, v')) &= (x \leq -1 \wedge x' \geq x + 1)\ , \\ ClauseSol(ti(v, v')) &= (x \leq -1 \wedge x' \geq x + 1)\ . \end{aligned}$$

Thus, the algorithm E-HSF finds a solution to the original set of forall-exists Horn clauses (and hence proves the program satisfies the CTL property).

3.4 Optimisations

During the process of solving a given set of clauses, E-HSF computes auxiliary assertions that are often related to over-approximations of the reachable states by starting from the initial states. For a given program, the reachable states can be considered as images of its initial states via the transitive closure of the transition relation of the program. E-HSF computes a finite over-approximation of the set of reachable states by applying predicate abstraction at each step of the transition relation. Predicate abstraction can sometimes be too slow and even less advantageous for program variables whose domain is finite and which are not involved in arithmetic computations. This leaves room for an optimisation by giving special treatment, for example by not applying abstraction, for such variables when solving the given set of clauses. Another room for optimisation is in the transition relation itself where some programming language constructs are modeled

in ways that cause inefficient computation when solving clauses. An example is the loop construct of programming languages. In this section, we present two optimisation techniques that can be applied by E-HSF to enable efficient solving in the light of certain kind of constructs.

3.4.1 Explicit evaluation

In a transition system modeling a given program, the control flow of the program is modeled symbolically using a program counter variable. The program counter variable is not involved in any kind of computation except in equality checks and assignments. In addition, the program counter variable ranges over some finite domain, which corresponds to the possible reachable locations for the program control, for a given program. Efficient treatment of the program counter variable along the lines of explicit analysis, e.g., as performed in HSF and CPAchecker frameworks, could lead to high performance when dealing with programs with large control-flow graphs [17].

We consider the following set of clauses that encode a check whether the program given in Figure 3.4 terminates or not.

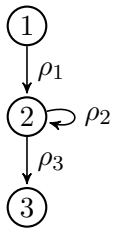
$$
\begin{aligned}
&\mathit{init}(v) \rightarrow \mathit{inv}(v),\\
&\mathit{inv}(v) \wedge \mathit{next}(v, v') \rightarrow \mathit{inv}(v') \wedge \mathit{rank}(v, v'),\\
&\mathit{rank}(v, v') \rightarrow \mathit{ti}(v, v'),\\
&\mathit{ti}(v, v') \wedge \mathit{rank}(v', v'') \rightarrow \mathit{ti}(v, v''),\\
&\mathit{dwf}(\mathit{ti}).
\end{aligned}
$$

Notice that there is no any existentially quantified implication constraint in the set of clauses. This is intentional for the sake of focusing on the explicit evaluation aspect of E-HSF. The unknown predicates that we need to solve for are $\mathit{inv}(v)$, $\mathit{rank}(v, v')$, and $\mathit{ti}(v, v')$. The predicate $\mathit{inv}(v)$ corresponds to states reachable during program execution, while the last three clauses ensures that $\mathit{rank}(v, v')$ is a well-founded relation.

To solve this set of clauses, E-HSF applies resolution over the first four clauses, which are the inference clauses, to infer solutions for the unknown predicates $\mathit{inv}(v)$, $\mathit{rank}(v, v')$,

```
 main(int x) {
1  x = 3;
2  while (x > 0) {
     x--;
3  }
 }
```

(a)



(b)

$$
\begin{aligned}
v =\ & (x, pc),\\
\mathit{init}(v) =\ & (pc = 1),\\
\mathit{next}(v, v') =\ & (pc = 1 \wedge \rho_1 \wedge pc' = 2 \vee\\
& \quad pc = 2 \wedge \rho_2 \wedge pc' = 2 \vee\\
& \quad pc = 2 \wedge \rho_3 \wedge pc' = 3),\\
\rho_1 =\ & (x' = 3),\\
\rho_2 =\ & (x - 1 \geq 0 \wedge x' = x - 1),\\
\rho_3 =\ & (x \leq 0 \wedge x' = x).
\end{aligned}
$$

(c)

FIGURE 3.4: An example program (a), its control-flow graph (b), and the corresponding transition system (c).

and $ti(v, v')$. One set of solutions that can be inferred from the inference clauses is:

$$
\begin{aligned}
\mathit{inv}(v) =\ & (pc = 1 \vee pc = 2 \wedge 0 \leq x \leq 3 \vee pc = 3 \wedge x = 0)\ ,\\
\mathit{rank}(v, v') =\ & (pc = 1 \wedge x' = 3 \wedge pc' = 2 \vee\\
& \quad pc = 2 \wedge x \geq 1 \wedge x' = x - 1 \wedge pc' = 2 \vee\\
& \quad pc = 2 \wedge x \leq 0 \wedge x' = x \wedge pc' = 3)\ ,\\
\mathit{ti}(v, v') =\ & (pc = 1 \wedge x' = 3 \wedge pc' = 2 \vee\\
& \quad pc = 1 \wedge x \geq 1 \wedge x' \leq x - 1 \wedge pc' = 2 \vee\\
& \quad pc = 1 \wedge x \leq 0 \wedge x' = x \wedge pc' = 3 \vee\\
& \quad pc = 2 \wedge x \geq 1 \wedge x' \leq x - 1 \wedge pc' = 2 \vee\\
& \quad pc = 2 \wedge x \leq 0 \wedge x' = x \wedge pc' = 3)\ .
\end{aligned}
$$

Together with the inference, E-HSF also applies the last clause, which is a property clause, to check if each inferred solution for $ti(v, v')$ satisfies the disjunctive well-foundedness requirement. For this check, we first apply BOUNDT and DECREASET to construct templates $\mathit{bound}(v)$ and $\mathit{decrease}(v, v')$ corresponding to a bound and decrease

condition over the program variables, respectively.

$$bound(v) = (r_x x + r_{pc} pc \geq r_0) \ ,$$
$$decrease(v, v') = (r_x x' + r_{pc} pc' \leq r_x x + r_{pc} pc - 1) \ .$$

In general, for each disjunct of the inferred solution of $ti(v, v')$, say $ti_j(v, v')$, we check if $ti_j(v, v') \rightarrow bound(v) \wedge decrease(v, v')$ is satisfiable. However, this can be very inefficient because well-foundedness for many disjuncts $ti_j(v, v')$ can be proven without a need to encode and solve for the implication $ti_j(v, v') \rightarrow bound(v) \wedge decrease(v, v')$. If we consider the first disjunct of $ti(v, v')$ above, which is $(pc = 1 \wedge x' = 3 \wedge pc' = 2)$, the value of the program counter variable pc changes from 1 to 2. This alone is enough to conclude that the disjunct is well-founded. Therefore, by reasoning explicitly on the values of the program counter variable pc, we are able to avoid a much more demanding computation that is needed to encode and solve for the implication:

$$(pc = 1 \wedge x' = 3 \wedge pc' = 2) \rightarrow (r_x x + r_{pc} pc \geq r_0) \wedge (r_x x' + r_{pc} pc' \leq r_x x + r_{pc} pc - 1) \ .$$

It can be seen that explicit evaluation of the program counter variable is enough to prove well-foundedness for the first, second, third and fifth disjuncts (4 of the 5) from the inferred solution of $ti(v, v')$. For the fourth disjunct, which is $(pc = 2 \wedge x \geq 1 \wedge x' \leq x - 1 \wedge pc' = 2)$, the value of pc remains the same. Therefore, we can not say anything on the well-foundedness of the disjunct based on explicit evaluation of the program counter variable. In such cases, the general approach is followed where we apply BoundT and DecreaseT to construct templates $bound(v)$ and $decrease(v, v')$ respectively. However, since the value of the program counter variable stays the same, in these bound and decrease conditions as well as in the disjunct for which we want to prove well-foundedness, expressions over the program counter variable pc (and its primed version pc') can be ignored as shown below.

$$bound(v) = (r_x x \geq r_0),$$
$$decrease(v, v') = (r_x x' \leq r_x x - 1).$$

Then, we solve for:

$$(x \geq 1 \wedge x' = x - 1) \rightarrow (r_x x \geq r_0) \wedge (r_x x' \leq r_x x - 1) \ .$$

One of the procedures of our solving algorithm E-HSF that make use of explicit evaluation based optimisation is the check for well-foundedness, like shown in the example. The perfomance gain due to explicit evaluation is presented in Section 4.5 in the application area of CTL verification.

3.4.2 Loop acceleration

A major challenge in computing the set of reachable states lies in loop constructs of programming languages. Loops are difficult to reason about because the number of iterations can not always be statically determined, and hence, there is a need to reason about loops symbolically independently of the exact number of iterations. Our loop acceleration procedure targets a class of loops called self-loops. A loop at a given progam location is said to be a self loop if

- there is no any other loop at the location,
- it is terminating, and
- each update of the loop is either an increment or decrement by a constant, or an assignment of a constant.

In practice, it is very important to model self-loops of a given program in a way that computing the set of reachable states is efficient. The loop acceleration procedure does pre-processing on a given transition relation by re-writing disjuncts corresponsing to self loops.

We illustrate our loop acceleration based optimisation with an example. Let us consider again the program in Figure 3.4. Note that ρ_2 is a self-loop over the program location ℓ_2. The clauses given below compute a set of reachable states for the program.

$$\begin{aligned} &\mathit{init}(v) \rightarrow \mathit{inv}(v), \\ &\mathit{inv}(v) \wedge \mathit{next}(v, v') \rightarrow \mathit{inv}(v'). \end{aligned}$$

Let us see how $\mathit{inv}(v)$, which corresponds to the set of reachable states, is computed. The first clause ensures that any initial state, which is in $\mathit{init}(v)$, is added to $\mathit{inv}(v)$, i.e., $\mathit{inv}(v) \supseteq \{(pc = 1)\}$. The second clause computes more states and adds to $\mathit{inv}(v)$ by applying the transition relation of the program over $\mathit{inv}(v)$ recursively. The computation first applies ρ_1 over $\mathit{inv}(v) \supseteq \{(pc = 1)\}$ to compute a new state $(pc = 2, x = 6)$ such that $\mathit{inv}(v) \supseteq \{(pc = 1), (pc = 2, x = 6)\}$. Since ρ_1 will not compute any new state, we will apply other relations in the transition relation can compute more new states. The computation first applies ρ_2 over $\mathit{inv}(v) \supseteq \{(pc = 1), (pc = 2, x = 6)\}$ to compute a new state $(pc = 2, x = 4)$ such that $\mathit{inv}(v) \supseteq \{(pc = 1), (pc = 2, x = 6), (pc = 2, x = 4)\}$. Since ρ_2 will still compute a new state by applying on $\mathit{inv}(v)$, we keep on applying ρ_2. Indeed, ρ_2 is applied a total of three times to get $\mathit{inv}(v) \supseteq \{(pc = 1), (pc = 2, x = 6), (pc = 2, x = 4), (pc = 2, x = 2), (pc = 2, x = 0)\}$. The computation finally applies ρ_3 over $\mathit{inv}(v) \supseteq \{(pc = 1), (pc = 2, x = 6), (pc = 2, x = 4), (pc = 2, x = 2), (pc = 2, x = 0)\}$

to compute a new state $(pc = 3, x = 0)$ such that $inv(v) \supseteq \{(pc = 1), (pc = 2, x = 6), (pc = 2, x = 4), (pc = 2, x = 2), (pc = 2, x = 0), (pc = 3, x = 0)\}$. The computation terminates here since no more new states can be computed.

During this computation, ρ_2 is applied three times to decrease the value of x from 6 to 0 in 3 iterations. Once x has the value 0, iteration over the loop terminates since ρ_2 can no more be applied. If x had a large value, say n, at the start of the loop, the process of computing new states by applying ρ_2 takes approximately $n/2$ steps. This can make the process of computing the set of reachable states inefficient since one state is computed at a time. We apply loop acceleration to avoid such inefficient computation.

Let us see how ρ_2 can be modified in such a way that the set of states $\{(pc = 2, x = 4), (pc = 2, x = 2), (pc = 2, x = 0)\}$ can be computed in a single step from the state $(pc = 2, x = 6)$. Since only the value of x change during the application of ρ_2, we focus on how we can compute the set of states $\{(x = 4), (x = 2), (x = 0)\}$ given the state $(x = 6)$. The part of ρ_2 that we are interested in is $x > 0 \wedge x' = x - 2$, i.e, the part excluding constraints over the program counter variables. Let $k \geq 1$ be the loop counter and $x(k)$ be the value of x at iteration k. Since the loop executes for any $k \geq 1$ whose values of $x(k)$ satisfies the loop condition, we can re-write the loop as

$$k \geq 1 \wedge x(k) > 0 \wedge x' = x(k) - 2$$

We can consider $x(k)$ is given by the recurrence

$$x(k) = x(k-1) - 2$$

where $x(0)$ corresponds to the initial value of x before the loop starts to execute. One closed form of the recurrence is given by

$$x(k) = x(0) - 2 * (k - 1)$$

After making the appropriate replacement, the computation of x' at iteration k is given by

$$k \geq 1 \wedge x(k) = x - 2 * (k - 1) \wedge x(k) > 0 \wedge x' = x(k) - 2$$

Going back to our example, when the loop is executed from a state where x has the vale 6, the above equation reduces to $k \geq 1 \wedge k \leq 3 \wedge x' = 6 - k * 2$. We can see that for each of the three values of k, a corresponding value of x' gets computed. We claim that the constraint $k \geq 1 \wedge k \leq 3 \wedge x' = 6 - k * 2$ is the accelerated version for the original constraint $x > 0 \wedge x' = x - 2$ as it computes all values of x' in a single step.

function LoopAccel
input
$T(v, v')$:= $\bigvee_{i=1}^{n} \tau_i(v, v')$ - a transition relation
local
function GetAccelRel
begin
1: **for** i in $1 \ldots n$ **do**
2: $\sigma_i(v, v')$:= $\tau_i(v, v')$
3: **if** $\tau_i(v, v') \models pc = pc'$ **then**
4: let $\tau_i(v, v')$:= $pc = \ell_0 \wedge \rho_i(v, v') \wedge pc' = \ell_0$
5: **if** $\exists k : 1 \leq k \leq n \wedge k \neq i$ and $\tau_k(v, v') \models pc = pc' \wedge pc = \ell_0$ **then**
6: **goto** 2
7: $head(v, v')$:= BoundT(v) $\wedge$ DecreaseT(v, v')
8: $Constraint(v, v')$:= EncodeValidity$(\rho_i(v, v') \rightarrow head(v, v'))$
9: **if** SmtSolve$(Constraint(v, v'))$ **then**
10: let ℓ_{acc} be a fresh program control location value
11: $\sigma_{acc}(v, v')$:= GetAccelRel$(\rho_i(v, v'))$
12: $\sigma_i(v, v')$:= $pc = \ell_0 \wedge \sigma_{acc}(v, v') \wedge pc' = \ell_{acc})$
13: **for** j in $1 \ldots n$ **do**
14: **if** $\tau_j(v, v') \models pc = \ell_0 \wedge pc \neq pc'$ **then**
15: let $\tau_j(v, v')$:= $pc = \ell_0 \wedge \rho_j(v, v') \wedge pc' = \ell_1$
16: $\sigma_i(v, v')$:= $\sigma_i(v, v') \vee pc = \ell_{acc} \wedge \rho_j(v, v') \wedge pc' = \ell_1$
17: **done**
18: **done**
19: **return** $\bigvee_{i=1}^{n} \sigma_i(v, v')$
20: **end**

FIGURE 3.5: The loop acceleration procedure.

The loop acceleration procedure: The procedure re-writes the transition relation of a given program in such a way that disjuncts modeling self-loops are replaced by non-looping relations. The complete procedure is given in Figure 3.5. The procedure takes the transition relation of a given program as input, and manipulates each disjunct of the transition relation if the disjunct corresponds to a self-loop. If the disjunct does not correspond to a self-loop, then the procedure does not make any change to the disjunct. In the procedure, each disjunct is checked if it models a loop (at Line 3), if there is no any other loop on the same location of this loop (at Line 5), and if the loop modeled by the disjunct terminates (at Line 9).

The disjunct that models a self loop is passed to the function GetAccelRel shown in Figure 3.6. This function uses fresh variables k and v_k to represent symbolically the loop counter and the unprimed versions of the program variables v at the k^{th} iteration, respectively. The function takes each of the updates and guards in the given disjunct and encodes new corresponding updates and guards using k and v_k in addition to v. Note that an update expression can only be an increment or decrement by a constant or a constrant assignment. Some valid update expressions are $x' = x + 1, x' = x - 2$ and $pc' = 3$, but expressions like $x' = x + y$ and $x' = 2 * x$ are not valid for self-loops. For

function GetAccelRel
input
$\rho(v, v')$ - a relation corresponding to a self-loop
begin
1: let c be a constant
2: let k be a fresh variable corresponding to a loop counter
3: let v_k be a tuple of fresh variables of length $|v|$
4: $\rho(v, v') := \bigwedge_{i=1}^{m} grd_i(v) \wedge \bigwedge_{j=1}^{n} upd_j(v, v')$
5: $\sigma(v, v') := k \geq 1$
6: **for** j in $1 \ldots n$ **do**
7: **match** $upd_i(v, v')$ **with**
8: | $x' = c$ ->
9: $\sigma(v, v') := \sigma(v, v') \wedge x_k = c \wedge x' = c$
10: | $x' = x + c$ ->
11: $\sigma(v, v') := \sigma(v, v') \wedge x_k = x + (k - 1) * c \wedge x' = x_k + c$
12: **done**
13: **for** i in $1 \ldots m$ **do**
14: $\sigma(v, v') := \sigma(v, v') \wedge grd_i([v_k/v])$
15: **done**
16: **return** $\sigma(v, v')$
17: **end**

FIGURE 3.6: The procedure GetAccelRel.

each update expression in the input disjunct, GetAccelRel encodes two new update expressions.

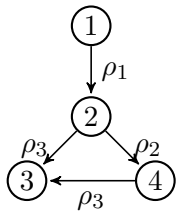
- the first expression defines x_k, which represents a program variable at the start of the k^{th} iteration of the loop, in terms of x and a constant, which represents the corresponding program variable at the start of the loop. This expression simulates the computation of the possible values of the program variable x before the k^{th} iteration of the loop is executed.
- the second expression defines x', which represents the corresponding program variable at the end of the given loop iteration, in terms of x_k. This expression simulates the computation of the possible values of the program variable x after the k^{th} iteration of the loop is executed.

This is done at Line 9 when the update expression is a constant assignment, and at Line 11 when the update expression is an increment or decrement by a constant. For each guard expression of the input disjunct, v_k, which represents the program variables at the start of the k^{th} iteration of the loop, replaces v. This is done at Line 14.

In general, the function GetAccelRel first models the input disjunct as a recurrence and then it computes a closed form for the recurrence. The closed form, which represents values of program variables using the values at the start of the loop and the loop counter, forms a core part of the accelerated relation of the loop.

$$
\begin{aligned}
v &= (x, pc),\\
init(v) &= (pc = 1),\\
next(v, v') &= (pc = 1 \wedge \rho_1 \wedge pc' = 2 \vee\\
&\quad\ pc = 2 \wedge \rho_2 \wedge pc' = 2 \vee\\
&\quad\ pc = 2 \wedge \rho_3 \wedge pc' = 3),\\
\rho_1 &= (x' = 3),\\
\rho_2 &= (k \geq 1 \wedge x_k = x - (k-1) * 2 \wedge x_k > 0 \wedge x' = x_k - 2),\\
\rho_3 &= (x \leq 0 \wedge x' = x).
\end{aligned}
$$

(a)



(b)

FIGURE 3.7: control-flow graph with the accelerated loop (a), and the corresponding modified transition system (b).

Let us get back to the main function LOOPACCEL. The relation returned by GETACCELREL simulates the original self-loop without a need to iteration, and hence, the primed program counter variable is assigned a fresh program location. This fresh location can be considered as a destination location of the self-loop. All these expression together form the accelerated version of the self-loop, which is given at Line 12. Any other disjunct of the transition relation which originates in the same program location as the self-loop must have a version from the fresh program location. This is done at Lines 14-16.

Example revisited: Let us see how LOOPACCEL manipulates the transition relation $next(v, v')$ for the program in Figure 3.4. The disjuncts ρ_1 and ρ_3 stay unchanged since they do not satisfy the condition on Line 3 of LOOPACCEL. For ρ_2, however, the conditions on Line 3 is satisfied as both pc and pc' have the value 2. Since ρ_2 is the only self-loop on the program counter location 2, the goto condition on Line 5 does not hold. In addition, the constraint $(x > 0 \wedge x' = x - 2)$ of ρ_2 satisfies the condition on Line 7 that checks the well-foundedness of the loop. Therefore, ρ_2 models a self-loop that can be accelerated and its relation $(x > 0 \wedge x' = x - 2)$ is passed to GETACCELREL and the corresponding relation for the accelerated loop are generated. The function GETACCELREL adds the constraint $k \geq 1$ over a fresh variable k as the first conjunct of the accelerated loop constraint as shown on Line 3. Then, by taking the only update in the input constraint, i.e, $x' = x - 1$, the expressions $x_k = x - (k-1) * 2$ and $x' = x_k - 2$ are added to the accelerated constraint, at Lines 7-11. Finally, the guard $x > 0$ is taken, and x is substituted by x_k resulting in a new guard $x_k > 0$, at Line 14. The new guard will be the final disjunct of the accelerated constraint. GETACCELREL returns $k \geq 1 \wedge x_k = x - (k-1) * 2 \wedge x' = x_k - 2 \wedge x_k > 0$ back to LOOPACCEL. Assume

ℓ_4 is the new program counter location on Line 10 of LoopAccel. The accelerated version of $(pc = \ell_2 \wedge x > 0 \wedge x' = x - 2 \wedge pc' = \ell_2)$ will be $(pc = \ell_2 \wedge k \geq 1 \wedge x_k = x - (k-1) * 2 \wedge x' = x_k - 2 \wedge x_k > 0 \wedge pc' = \ell_4)$. This is done on Line 12. Now we generate more disjuncts as shown in Lines 13 to 17. The only disjunct of the $next(v, v')$ that satisfies the condition on Line 14 is $(pc = \ell_2 \ \wedge \ x \leq 0 \ \wedge \ x' = x \ \wedge \ pc' = \ell_3)$. The new disjunct $(pc = \ell_4 \ \wedge \ x \leq 0 \ \wedge \ x' = x \ \wedge \ pc' = \ell_3)$ is added to the accelerated transition relation. The complete accelerated version for the transition system of our example program is given in Figure 3.7.

3.5 Implementation

Our implementation of E-HSF relies on HSF [59] to solve universally-quantified Horn clauses over linear inequalities (see line 4 in Figure 3.2) and on the Z3 solver [47] at line 10 in Figure 3.3 to solve for the template parameters in the (possibly non-linear) constraints. The EncodeValidity function at line 11 in Figure 3.3 encodes the Farkas' lemma from linear programming [107].

In general, the templates GrdT and RelT that are applied in the DefsRefine procedure are provided by the user and need to satisfy the condition in Equation 3.1. Our implementation checks this condition for linear templates by using quantifier elimination techniques. For dealing with well-foundedness we use linear ranking functions, and hence corresponding linear templates for DecreaseT and BoundT.

3.6 Related work

Our work is inspired by a recent approach to CTL verification of programs [39]. The main similarity lies in the use of a refinement loop to discover witnesses for resolving non-determinism/existentially quantified variables. The main difference lies in the way candidate witnesses are selected. While [39] refines witnesses, i.e., the non-determinism in witness relations monotonically decreases at each iteration, E-HSF can change witness candidates arbitrarily (yet, subject to the global constraint). Thus, our method can backtrack from wrong choices in cases when [39] needs to give up.

E-HSF generalizes solving methods for universally quantified Horn clauses over decidable theories, e.g. [59, 73, 94]. Our approach relies on the templates for describing the space of candidate witnesses. Computing witnesses using a generalisation approach akin to PDR [73] is an interesting alternative to explore in future work.

Template based synthesis of invariants and ranking functions is a prominent technique for dealing with universal properties, see e.g. [37, 64, 104, 114]. E-HSF implementation of EncodeValidity supporting linear arithmetic inequalities is directly inspired by these techniques, and puts them to work for existential properties.

Decision procedures for quantified propositional formulas on bit as well as word level [76, 121] rely on iteration and refinement for the discovery of witnesses. The possibility of integration of QBF solvers as an implementation of EncodeValidity is an interesting avenue for future research.

Some formulations of proof systems for mu-calculus, e.g., [44] and [95], could be seen as another source of forall-exists clauses (to pass to E-HSF). Compared to the XSB system [44] that focuses on finite state systems, E-HSF aims at infinite state systems and employs a CEGAR-based algorithm. XSB's extensions for infinite state systems are rather specific, e.g., data-independent systems, and do not employ abstraction refinement techniques. Finally, we remark that abstraction-based methods, like ours, can be complemented with program specialization-based methods for verification of CTL properties [55].

3.7 Conclusion

Verification conditions for proving existential temporal properties of programs can be represented using existentially quantified Horn-like clauses. In this chapter, we presented a counterexample guided method for solving such clauses, which can compute witnesses to existentially quantified variables in form of linear arithmetic expressions. By aggregating constraints on witness relations across different counterexamples our method can recover from wrong choices. We leave the evaluation of applicability of our method for problems requiring witness computation, e.g., verification of branching time temporal properties or computing winning strategies in graph games, for the remaining chapters.

Chapter 4

CTL Verification as Horn Constraint Solving

4.1 Introduction

Since Pnueli's pioneering work [102], temporal logics has long been recognised as a fundamental approach to the formal specification and verification of reactive systems [49, 88]. Temporal logics allow precise specification of complex properties. There has been decades of effort on temporal verification of finite state systems [26, 32, 34, 84]. For CTL and other state-based properties, the standard procedure is to adapt "bottom-up" (or "tableaux") techniques for reasoning on finite-state systems. In additional, various classes of temporal logics support model-checking whose success over the last twenty years is allowing large and complex (finite) systems to be verified automatically [26, 33, 69, 92]. In recent decades, however, the research focus has shifted to infinite-state systems in general and on software systems in particular as ensuring correctness for software is in high demand. Most algorithms for verifying CTL properties on infinite-state systems typically involve first abstracting the state space into a finite-state model, and then applying finite reasoning strategies on the abstract model. There is also a lot of effort on algorithms that are focused on a particular fragment of CTL, such as the universal fragment [100] and the existential fragment [66], or some particular classes of infinite-state systems such as pushdown processes [112, 113, 119, 120] or parameterised systems [48, 51].

In this chapter, we take on the problem of automatically verifying CTL properties for a given (possibly infinite-state) program. We propose a method based on solving a set of forall-exists quantified Horn constraints. Our method takes a program P modeled by a transition system $(init(v), next(v, v'))$ and a property given by a CTL formula

$\varphi(v)$, and then it checks if P satisfies $\varphi(v)$, i.e., if $(init(v), next(v, v')) \models_{CTL} \varphi(v)$. The method first generates a set of forall-exists quantified Horn constraints with well-foundedness conditions by exploiting the syntactic structure of the CTL formula $\varphi(v)$. It then solves the generated set of Horn constraints by applying the solving algorithm E-HSF, which has been discussed in Chapter 3. We claim that P satisfies $\varphi(v)$ if and only if the generated set of Horn constraints has a solution. We demonstrate the practical applicability of the method by presenting experimental evaluation using examples from the PostgreSQL database server, the SoftUpdates patch system, the Windows OS kernel.

The rest of the chapter is organised as follows. We start by briefly revising the syntax and semantics of CTL in Section 4.2. In Section 4.3, we present our proof system that generates a set of forall-exists quantified Horn constraints for a given verification problem. We illustrate application of the proof rules on an example in Section 4.4. The experimental evaluation of our method is given in Section 4.5. Finally, we present a bief discussion on related works in Section 4.6 and concluding remarks in Section 4.7.

4.2 CTL basics

In this section, we review the syntax and the semantics of the logic CTL following [81]. Let $\mathcal{T}$ be a first order theory and $\models_{\mathcal{T}}$ denote its satisfaction relation that we use to describe sets and relations over program states. Let c range over assertions in $\mathcal{T}$ and x range over variables. A CTL formula φ is defined by the following grammar using the notion of a path formula ϕ.

$$\varphi ::= c \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid A\,\phi \mid E\,\phi$$
$$\phi ::= X\varphi \mid G\varphi \mid \varphi U \varphi$$

As usual, we define $F\varphi = (trueU\varphi)$. The satisfaction relation $P \models \varphi$ holds if and only if for each s such that $init(s)$ we have $P, s \models \varphi$. We define $P, s \models \varphi$ as follows using an

auxiliary satisfaction relation $P, \pi \models \phi$.

$$
\begin{aligned}
&P, s \models c && \text{iff } s \models_{\mathcal{T}} c \\
&P, s \models \varphi_1 \wedge \varphi_2 && \text{iff } P, s \models \varphi_1 \text{ and } P, s \models \varphi_2 \\
&P, s \models \varphi_1 \vee \varphi_2 && \text{iff } P, s \models \varphi_1 \text{ or } P, s \models \varphi_2 \\
&P, s \models A\,\phi && \text{iff for all } \pi \in \Pi_P(s) \text{ holds } P, \pi \models \phi \\
&P, s \models E\,\phi && \text{iff exists } \pi \in \Pi_P(s) \text{ such that } P, \pi \models \phi \\
&P, \pi \models X\varphi && \text{iff } \pi = s_1, s_2, \ldots \text{ and } P, s_2 \models \varphi \\
&P, \pi \models G\varphi && \text{iff } \pi = s_1, s_2, \ldots \text{for all } i \geq 1 \text{ holds } P, s_i \models \varphi \\
&P, \pi \models \varphi_1 U \varphi_2 && \text{iff } \pi = s_1, s_2, \ldots \text{ and exists } j \geq 1 \text{ such that} \\
& && \qquad P, s_j \models \varphi_2 \text{ and } P, s_i \models \varphi_1 \text{ for } 1 \leq i < j
\end{aligned}
$$

In this chapter, we represent a satisfaction relation $P \models \varphi$ by the relation $P \models_{CTL} \varphi$ to explicitly indicate that φ is a CTL formula. We call such relation a CTL satisfaction, and φ is said to be its formula.

4.3 Proof system

Our CTL verification method encodes the verification problem as a problem of solving forall-exists quantified Horn constraints with well-foundedness conditions. This is done by applying a proof system that consists of various proof rules for handling different kinds of CTL formulas. This proof system is based on a deductive proof system for CTL* from [81] which is adapted in this work to be suitable from the perspective of constraint generation for a CTL satisfaction.

Given a transition system $(\mathit{init}(v), \mathit{next}(v, v'))$ and a CTL formula $\varphi(v)$, the appropriate proof rules are used from the proof system to generate the corresponding set of Horn constraints for the CTL satisfaction $(\mathit{init}(v), \mathit{next}(v, v')) \models_{CTL} \varphi(v)$. There are two sets of proof rules in the proof system.

4.3.1 Proof rules for decomposition

These proof rules are applied recursively to a CTL satisfaction whose formula is not an assertion. The proof rules decompose the given CTL formula into new sub-formulas by following the nesting structure of the formula. Then, the original satisfaction is reduced to new satisfactions over the new sub-formulas and a Horn constraint relating the new satisfactions.

There are different proof rules depending on the outer-most operator of the formula. One case is when the given formula $f(\psi(v))$ nests another formula $\psi(v)$ such that the outer-most operator f is a pair of a temporal path operator and a unary temporal state operator, i.e., $f \in \{AX, AG, AF, EX, EG, EF\}$. The corresponding proof rule RuleCtlDecompUni is given in Figure 4.1 that shows how such satisfactions are decomposed. Another case is when the given formula has a structure $f(\psi_1(v),\ \psi_2(v))$

Given a CTL formula $f(\psi(v))$ where $f \in \{AX, AG, AF, EX, EG, EF\}$, and a transition system $(p(v), next(v, v'))$, find an assertion $q(v)$ such that:

$$\frac{(p(v), next(v, v')) \models_{CTL} f(q(v)) \quad (q(v), next(v, v')) \models_{CTL} \psi(v)}{(p(v), next(v, v')) \models_{CTL} f(\psi(v))}$$

Figure 4.1: Proof rule RuleCtlDecompUni

nesting the formulas $\psi_1(v)$ and $\psi_2(v)$ such that the outer-most operator f is either a pair of a temporal path operator and the state operator until or a disjunction/conjunction, i.e., $f \in \{AU, EU, \wedge, \vee\}$. Note that when f is $\wedge$ (resp. $\vee$), the given formula $f(\psi_1(v),\ \psi_2(v))$ corresponds to $\psi_1(v) \wedge \psi_2(v)$ (resp $\psi_1(v) \vee \psi_2(v)$). The corresponding proof rule RuleCtlDecompBin is given in Figure 4.2 that shows how such satisfactions are decomposed.

Given a CTL formula $f(\psi_1(v),\ \psi_2(v))$ where $f \in \{AU, EU, \wedge, \vee\}$, and a transition system $(p(v), next(v, v'))$, find assertions $q_1(v)$ and $q_2(v)$ such that:

$$\frac{\begin{array}{c} p(v) \rightarrow f(q_1(v), q_2(v)), \\ (q_1(v), next(v, v')) \models_{CTL} \psi_1(v) \quad (q_2(v), next(v, v')) \models_{CTL} \psi_2(v) \end{array}}{(p(v), next(v, v')) \models_{CTL} f(\psi_1(v), \psi_2(v))}$$

Figure 4.2: Proof rule RuleCtlDecompBin

4.3.2 Proof rules for constraints generation

This set of proof rules are applied to a CTL satisfaction whose formula is either an assertion or a basic state formula. Any CTL satisfaction can be decomposed into a set of such simple CTL satisfactions by applying the proof rules from the previous section. The next step will be to generate forall-exists quantified Horn constraints (possibly with well-foundedness condition) that constrain a set of auxiliary assertions over program states.

The simplest of all is the proof rule RuleCtlInit, see Figure 4.3, which is applied when the CTL formula is an assertion.

For a CTL formula given by the assertion $\psi(v)$, and a transition system $(p(v), next(v, v'))$:

$$\frac{p(v) \rightarrow \psi(v)}{(p(v), next(v, v')) \models_{CTL} \psi(v)}$$

FIGURE 4.3: Proof rule RuleCtlInit

The proof rules RuleCtlEX (see Figure 4.4), RuleCtlEG (see Figure 4.5), and RuleCtlEU (see Figure 4.6) are applied for generating Horn constraints when the CTL formula is a basic state formula with existential path operator.

$$\frac{p(v) \rightarrow \exists v' : next(v, v') \wedge q(v')}{(p(v), next(v, v')) \models_{CTL} EX\ q(v)}$$

FIGURE 4.4: Proof rule RuleCtlEX

Find an assertion $inv(v)$ such that:

$$\frac{\begin{array}{c} p(v) \rightarrow inv(v) \\ inv(v) \rightarrow \exists v' : next(v, v') \wedge inv(v') \\ inv(v) \rightarrow q(v) \end{array}}{(p(v), next(v, v')) \models_{CTL} EG\ q(v)}$$

FIGURE 4.5: Proof rule RuleCtlEG

The proof rules RuleCtlAX (see Figure 4.7), RuleCtlAG (see Figure 4.8), and RuleCtlAU (see Figure 4.9) are applied for generating Horn constraints when the CTL formula is a basic state formula with universal path operator.

Our proof system is not exhaustive in terms of having proof rules for all kinds of basic state formula that can be defined in CTL. However, we utilize equivalence between CTL formulas to generate Horn constraints for a basic state formula whose proof rule is not

Find assertions $inv(v)$, $rank(v, v')$ and $ti(v, v')$ such that:

$$p(v) \rightarrow inv(v)$$
$$inv(v) \wedge \neg r(v) \rightarrow q(v) \wedge \exists v' : next(v, v') \wedge inv(v') \wedge rank(v, v')$$
$$rank(v, v') \rightarrow ti(v, v')$$
$$ti(v, v') \wedge rank(v', v'') \rightarrow ti(v, v'')$$
$$dwf(ti)$$

$$(p(v), next(v, v')) \models_{CTL} EU(q(v), r(v))$$

FIGURE 4.6: Proof rule RULECTLEU

$$p(v) \wedge next(v, v') \rightarrow q(v')$$

$$(p(v), next(v, v')) \models_{CTL} AX\ q(v)$$

FIGURE 4.7: Proof rule RULECTLAX

Find an assertion $inv(v)$ such that:

$$p(v) \rightarrow inv(v)$$
$$inv(v) \wedge next(v, v') \rightarrow inv(v')$$
$$inv(v) \rightarrow q(v)$$

$$(p(v), next(v, v')) \models_{CTL} AG\ q(v)$$

FIGURE 4.8: Proof rule RULECTLAG

given in the proof system. The equivalence between the formulas $EU(true, q(v))$ and $EF(q(v))$ is used to define RULECTLEF (see Figure 4.10) from RULECTLEU. In the same way, the equivalence between the formulas $AU(true, q(v))$ and $AF(q(v))$ is used to define RULECTLAF (see Figure 4.11) from RULECTLAU.

Proof rules for finite-state systems: The proof rules discussed so far are devised with the assumption that the transition system $(init(v), next(v, v'))$ is infinite, i.e, for any state s that is reached from the initial state, there always exists a state s' such that $(s, s') \models next(v, v')$. However, for finite transition systems, it may be the case that for some state s that is reached from the initial state, there may not exist a state s' such that $(s, s') \models next(v, v')$. In such case, any computation of the transition system that

Find assertions $inv(v)$, $rank(v, v')$ and $ti(v, v')$ such that:

$$\begin{gathered} p(v) \rightarrow inv(v) \\ inv(v) \wedge \neg r(v) \wedge next(v, v') \rightarrow \ \ q(v) \wedge inv(v') \wedge rank(v, v') \\ rank(v, v') \rightarrow ti(v, v'), \\ ti(v, v') \wedge rank(v', v'') \rightarrow ti(v, v''), \\ dwf(ti). \end{gathered}$$

$$\frac{}{(p(v), next(v, v')) \models_{CTL} AU(q(v), r(v))}$$

FIGURE 4.9: Proof rule RULECTLAU

$$\frac{(p(v), next(v, v')) \models_{CTL} EU(true, q(v))}{(p(v), next(v, v')) \models_{CTL} EF\ q(v)}$$

FIGURE 4.10: Proof rule RULECTLEF

$$\frac{(p(v), next(v, v')) \models_{CTL} AU(true, q(v))}{(p(v), next(v, v')) \models_{CTL} AF\ q(v)}$$

FIGURE 4.11: Proof rule RULECTLAF

reaches the state s will terminate at s since there is no possible further computation. From the point of view of the computation, any property that is not yet satisfied but that is expected to be satisfied eventually is guaranteed not to be satisfied since computation terminates at the given state.

The proof rules that are affected by the finiteness of the transition system are AU and AX. For a property with the operator AU, we must make sure that for the property to eventually be satisfied in the future, there is actually an enabled transition relation from any currently reached state of a computation. This is done bad adding an extra forall-exists quantified Horn constraint $inv(v) \wedge \neg r(v) \rightarrow \exists v' : next(v, v')$ to RULECTLAU to ensure the existence of an enabled transition relation. The resulting proof rule RULECTLAUFINITE is given in Figure 4.12. Similarly, the Horn constraint $p(v) \rightarrow \exists v' : next(v, v')$ is added to RULECTLAX to get the proof rule

RuleCtlAxFinite which is given in Figure 4.13.

Find assertions $inv(v)$, $rank(v, v')$ and $ti(v, v')$ such that:

$$\begin{gathered} p(v) \rightarrow inv(v) \\ inv(v) \wedge \neg r(v) \rightarrow \exists v' next(v, v') \\ inv(v) \wedge \neg r(v) \wedge next(v, v') \rightarrow \ q(v) \wedge inv(v') \wedge rank(v, v') \\ rank(v, v') \rightarrow ti(v, v'), \\ ti(v, v') \wedge rank(v', v'') \rightarrow ti(v, v''), \\ dwf(ti). \end{gathered}$$

$$\overline{(p(v), next(v, v')) \models_{CTL} AU(q(v), r(v))}$$

Figure 4.12: Proof rule RuleCtlAuFinite

$$\begin{gathered} p(v) \rightarrow \exists v' next(v, v') \\ p(v) \wedge next(v, v') \rightarrow q(v') \end{gathered}$$

$$\overline{(p(v), next(v, v')) \models_{CTL} AX\ q(v)}$$

Figure 4.13: Proof rule RuleCtlAxFinite

4.4 Constraint generation

The contraint generation procedure performs a top-down, recursive descent through the syntax tree of the given CTL formula. At each level of recursion, the procedure takes as input an satisfaction $(p(v), next(v, v')) \models_{CTL} \varphi$, where φ is a CTL formula, and assertions $p(v)$ and $next(v, v')$ describe a set of states and a transition relation, respectively. The constraint generation procedure applies proof rules from the proof system presented in the previous section to recursively decompose complex satisfactions and eventually generate forall-exists quantified Horn constraints with well-foundedness conditions. Before starting the actual constraint generation, the procedure recursively re-writes the input satisfaction of a given CTL formula with arbitrary structure into a set of satisfactions of simple CTL formulas where each simple formula is either a basic CTL state formula or an assertion over the background theory. The procedure then takes each satisfaction involving simple formula, introduces auxiliary predicates and generates a sequence of forall-exists quantified Horn constraints and well-foundedness constraints (when needed) over these predicates.

Complexity and Correctness The procedure performs a single top-down descent through the syntax tree of the given CTL formula φ. The run time and the size of the generated constraints is linear in the size of φ. Finding a solution for the generated Horn constraints is undecidable in general. In practice however, our solving algorithm E-HSF often succeeds in finding a solution (see Section 4.5). We formalize the correctness of the constraint generation procedure in the following theorem.

Theorem 3. *For a given program P with $init(v)$ and $next(v, v')$ over v and a CTL formula φ the Horn constraints generated from $(p(v), next(v, v')) \models_{CTL} \varphi$ are satisfiable if and only if $P \models \varphi$.*

The proof can be found in [81].

Example Let us consider the program given in Figure 4.14. It contains the variable *rho* which is assigned a non-deterministic value at Line 2. This assignment results in the program control to move non-deterministically following the evaluation of the condition at Line 4. It is common to verify such programs with respect to various CTL properties as the non-determinism results in different computation paths of the program. Now, we would like to verify the example program with respect to the CTL property $AG(EF\ (WItemsNum \geq 1))$, i.e., from every reachable state of the program, there exists a path to a state where *WItemsNum* has a positive integer value.

```
      int main () {
1:       while(1) {
2:         while(1) {
              rho = nondet();
3:           if (WItemsNum<=5) {
4:             if (rho>0) break; }
5:           WItemsNum++;
6:         }
7:         while(1) {
8:           if (!(WItemsNum>2)) break;
9:           WItemsNum--;
10:        }
11:      }
12:   }
```

FIGURE 4.14: An example program

We can make the following observations about the program. The value of the variable *WItemsNum* is not set initially. Therefore, the property is checked for any arbitrary initial value of *WItemsNum*. The verification problem is more interesting for the case when *WItemsNum* has a non-positive integer value. This is because depending on how the variable *rho* is instantiated at Line 2, we may get a path that will not reach a state where *WItemsNum* gets a positive integer value. For example, if we assume *WItemsNum*

has the value 0 initially and *WItemsNum* is instantiated to the value 1, the program control swings between the two internal loops by keeping the value of *WItemsNum* the same. This resulting path will not reach the state with *WItemsNum* $\geq$ *1*. However, if *rho* is assigned a non-positive value, no matter what the value of *rho* is initially, it will eventually reach a value greater than 5 before exiting the first nested loop. Such path will eventually reach the state with *WItemsNum* $\geq$ *1* and hence the program satisfies the CTL property $AG(EF\ (WItemsNum \geq 1))$.

Our method abstracts away from the concrete syntax of a programming language by modeling a program as a transition system. The transition system for our example program is given below.

$$
\begin{aligned}
v &= (w, pc).\\
init(v) &= (pc = 1).\\
next(v, v') &= (pc = \ell_1 \wedge pc' = \ell_2 \wedge w' = w \vee\\
&\quad pc = \ell_2 \wedge pc' = \ell_3 \wedge w' = w \vee\\
&\quad pc = \ell_3 \wedge w \leq 5 \wedge pc' = \ell_4 \wedge w' = w \vee\\
&\quad pc = \ell_3 \wedge w > 5 \wedge pc' = \ell_5 \wedge w' = w \vee\\
&\quad pc = \ell_4 \wedge pc' = \ell_5 \wedge w' = w \vee\\
&\quad pc = \ell_4 \wedge pc' = \ell_7 \wedge w' = w \vee\\
&\quad pc = \ell_5 \wedge pc' = \ell_6 \wedge w' = w + 1 \vee\\
&\quad pc = \ell_6 \wedge pc' = \ell_3 \wedge w' = w \vee\\
&\quad pc = \ell_7 \wedge pc' = \ell_8 \wedge w' = w \vee\\
&\quad pc = \ell_8 \wedge w \leq 2 \wedge pc' = \ell_{11} \wedge w' = w \vee\\
&\quad pc = \ell_8 \wedge w > 2 \wedge pc' = \ell_9 \wedge w' = w \vee\\
&\quad pc = \ell_9 \wedge pc' = \ell_{10} \wedge w' = w - 1 \vee\\
&\quad pc = \ell_{10} \wedge pc' = \ell_8 \wedge w' = w \vee\\
&\quad pc = \ell_{11} \wedge pc' = \ell_3 \wedge w' = w).
\end{aligned}
$$

In the tuple of program variables v, w corresponds to the program variable *WItemsNum* and *pc* is the program counter variable. The problem of verifying the program with respect to the given property amounts to checking if $(init(v), next(v, v'))$ satisfies $AG(EF(w \geq 1))$, i.e., if the satisfaction $(init(v), next(v, v')) \models_{CTL} AG(EF(w \geq 1))$ holds. Our method first generates a set of Horn constraint corresponding to the verification problem by applying the proof system.

We start constraint generation by considering the nesting structure of $AG(EF(w \geq 1))$. Since $AG(EF(w \geq 1))$ has AG as the outer-most operator, we

apply RuleCtlDecompUni from Figure 4.1 to split the original satisfaction $(init(v), next(v, v')) \models_{CTL} AG(EF(w \geq 1))$ into a reduced satisfaction $(init(v), next(v, v')) \models_{CTL} AG(p_1(v))$ and a new satisfaction $(p_1(v), next(v, v')) \models_{CTL} EF(w \geq 1)$. We need to solve for the auxiliary assertion $p_1(v)$ satisfying both of the satisfactions.

One one hand, the assertion $p_1(v)$ corresponds to a set of program states that needs to be discovered form the initial state. This is represented by the new satisfaction $(init(v), next(v, v')) \models_{CTL} AG(p_1(v))$ which is reduced directly to a set of Horn constraints by applying RuleCtlAG from Figure 4.8. This set of Horn constraints is over an auxiliary predicate $inv_1(v)$ and given below.

$$
\begin{aligned}
&init(v) \rightarrow inv_1(v),\\
&inv_1(v) \wedge next(v, v') \rightarrow inv_1(v'),\\
&inv_1(v) \rightarrow p_1(v).
\end{aligned}
$$

On the other hand, we require the formula $EF(w \geq 1)$, which was nested in the main formula $AG(EF(w \geq 1))$, must be satisfied from the set of states represented by $p_1(v)$. This is represented by the new satisfaction $(p_1(v), next(v, v')) \models_{CTL} EF(w \geq 1)$. Unlike the reduced satisfaction above, this satisfaction is not always reduced directly to Horn constraints rather it can be reduced into simpler satisfactions if possible. Since $EF(w \geq 1)$ has EF as the outer-most operator, we apply again RuleCtlDecompUni from Figure 4.1 to split the satisfaction $(\mathtt{p}_1(v), next(v, v')) \models_{CTL} EF(w \geq 1)$ into a reduced satisfaction $(\mathtt{p}_1(v), next(v, v')) \models_{CTL} EF(p_2(v))$ and a new satisfaction $(p_2(v), next(v, v')) \models_{CTL} w \geq 1$. Here also, we need to solve for the auxiliary assertion $p_2(v)$ satisfying both of the satisfactions.

The reduced satisfaction $(\mathtt{p}_1(v), next(v, v')) \models_{CTL} EF(p_2(v))$ is reduced directly to a set of Horn constraints by applying RuleCtlEF from Figure 4.10. Due to the existential path quantifier in $(p_1(v), next(v, v')) \models_{CTL} EF(p_2(v))$, we obtain clauses that contain existential quantification. We deal with the eventuality by imposing a well-foundedness condition. This set of Horn constraints is over an auxiliary assertions $inv_2(v)$, $rank(v, v')$, and $ti(v, v')$ and given below.

$$
\begin{aligned}
&p_1(v) \rightarrow inv_2(v),\\
&inv_2(v) \wedge \neg p_2(v) \rightarrow \exists v' : next(v, v') \wedge inv(v') \wedge rank(v, v'),\\
&rank(v, v') \rightarrow ti(v, v'),\\
&ti(v, v') \wedge rank(v, v') \rightarrow ti(v, v''),\\
&dwf(ti).
\end{aligned}
$$

Coming to the new satisfaction $(p_2(v), next(v, v')) \models_{CTL} w \geq 1$, we see that its formula $w \geq 1$ is an assertion with no temporal operators. Since no further decomposition is possible, we apply RuleCtlInit from Figure 4.3 to generate directly the clause:

$$p_2(v) \rightarrow w \geq 1$$

As the original satisfaction $(init(v), next(v, v')) \models_{CTL} AG(EF(w \geq 1))$ is reduced into the satisfactions $(init(v), next(v, v')) \models_{CTL} AG(p_1(v))$, $(p_1(v), next(v, v')) \models_{CTL} EF(p_2(v))$ and $(p_2(v), next(v, v')) \models_{CTL} w \geq 1$, the constraints for the original satisfaction will be the union of the constraints for each of the decomposed satisfactions. The Horn constraints are over the auxiliary assertions $p_1(v)$, $inv_1(v)$, $p_2(v)$, $inv_2(v)$, $rank(v, v')$, and $ti(v, v')$, and they are given below.

$$\begin{aligned}
&init(v) \rightarrow inv_1(v),\\
&inv_1(v) \wedge next(v, v') \rightarrow inv_1(v'),\\
&inv_1(v) \rightarrow p_1(v),\\
&p_1(v) \rightarrow inv_2(v),\\
&inv_2(v) \wedge \neg p_2(v) \rightarrow \exists v' : next(v, v') \wedge inv(v') \wedge rank(v, v'),\\
&rank(v, v') \rightarrow ti(v, v'),\\
&ti(v, v') \wedge rank(v, v') \rightarrow ti(v, v''),\\
&dwf(ti)\\
&p_2(v) \rightarrow w \geq 1
\end{aligned}$$

This will be the final output of our Horn constraint generation procedure.

4.5 Evaluation

We evaluate out method of CTL verification by applying the implementation of the E-HSF solver from Chapter 3 on set of industrial benchmarks from [39, Figure 7]. These benchmarks consists of seven programs: `Windows OS fragment 1`, `Windows OS fragment 2`, `Windows OS fragment 3`, `Windows OS fragment 4`, `Windows OS fragment 5`, `PostgreSQL pgarch` and `Software Updates`. For each of these programs, four slightly different versions are considered for evaluation. In general, the four versions of a given program are the same in terms of the main logic of the program and what the program does, but they may differ on the value assigned to a particular variable or the condition for exiting a loop, etc. This gives us in total a set of 28 programs. Each such program P is given with a CTL property φ, and there are two verification tasks

associated with it: $P \models_{CTL} \varphi$ and $P \models_{CTL} \neg\varphi$. The existence of a proof for a property φ for P implies that $\neg\varphi$ is violated by the same program P, and similarly, a proof for $\neg\varphi$ for P implies that φ is violated by P. However, it may also be the case that both $P \not\models_{CTL} \varphi$ and $P \not\models_{CTL} \neg\varphi$ do not hold.

Templates: As discussed in section 3.5, the templates GRDT and RELT are provided by the user for queries with existentially quantified variables depending on the application. For the application of CTL verification, which is the main topic of interest in this chapter, we claim that the transition relation $next(v, v')$ can be used as a template by adding constraints at each location of non-determinism. There are two kinds of constraints that can be added depending on the two types of possible non-determinism in $next(v, v')$.

- **non-deterministic guards:** this is the case when $next(v, v')$ has a set of more than one disjuncts with the same guard, i.e., there can be more than one enabled moves from a certain state of the program. For each such set, we introduce a fresh case-splitting variable and we strengthen the guard of each disjunct by adding a distinct constraint on the fresh variable. For example, if the set has n disjuncts and B is a fresh variable, we add the constraint $B = i$ for each disjunct i where $1 \leq i \leq n$. To reason about existentially quantified queries, then it will suffices to instantiate B to one of the values in the range $1 \ldots n$. Such reasoning is done by the E-HSF solver.
- **non-deterministic assignments:** this is the case when $next(v, v')$ has a disjunct in which some w', which is a subset of v', is left unconstrained in the disjunct. In such case, we strengthen the disjunct by adding the constraint $x' = T_x * v + t_x$ as conjunct for each variable x' in w'. Solving for T_x and t_x is done by the E-HSF solver.

In our CTL verification examples, both non-deterministic guards and assignments are explicitly marked in the original benchmark programs using names `rho1`, `rho2`, etc. We apply the techniques discussed above to generate templates from the transition relation of each program. In these examples, linear templates are sufficiently expressive. For direct comparison with the results from [39], we used template functions corresponding to the `rho`-variables. The quantifier elimination in $\exists v' : next(v, v')$ can be automated for the theory of linear arithmetic. For dealing with well-foundedness we use linear ranking functions, and hence corresponding linear templates for DECREASET and BOUNDT.

We report the results in Table 4.1. For each program in Column 1, we report the shape of the property in Column 2, the result returned by E-HSF and the time it took to

Program P	Property φ	$P \models_{CTL} \varphi$		$P \models_{CTL} \neg\varphi$	
		Result	Time(s)	Result	Time(s)
`Windows OS fragment 1` (`29 LOC`)	$AG(p \rightarrow AFq)$	✓	0.4	×	0.3
	$EF(p \wedge EGq)$	✓	0.3	×	0.4
	$AG(p \rightarrow EFq)$	✓	0.4	×	0.3
	$EF(p \wedge AGq)$	✓	0.3	×	0.3
`Windows OS fragment 2` (`58 LOC`)	$AG(p \rightarrow AFq)$	✓	0.6	×	0.3
	$EF(p \wedge EGq)$	✓	0.4	×	0.4
	$AG(p \rightarrow EFq)$	✓	0.5	×	0.3
	$EF(p \wedge AGq)$	✓	0.5	×	0.3
`Windows OS fragment 3` (`370 LOC`)	$AG(p \rightarrow AFq)$	✓	12.9	×	1.1
	$EF(p \wedge EGq)$	✓	159.0	×	12.3
	$AG(p \rightarrow EFq)$	✓	13.6	×	0.8
	$EF(p \wedge AGq)$	✓	27.2	×	1.1
`Windows OS fragment 4` (`380 LOC`)	$AFp \vee AFq$	✓	43.3	×	6.3
	$EGp \wedge EGq$	✓	0.4	×	9.5
	$EFp \wedge EFq$	✓	101.7	×	0.6
	$AGp \vee AGq$	✓	0.2	×	32.1
`Windows OS fragment 5` (`43 LOC`)	$AG(AFp)$	✓	0.4	×	0.3
	$EF(EGp)$	✓	0.3	×	0.4
	$AG(EFp)$	✓	0.4	×	0.3
	$EF(AGp)$	✓	0.3	×	0.3
`PostgreSQL pgarch` (`70 LOC`)	$AG(AFp)$	✓	0.5	×	0.3
	$EF(EGp)$	✓	0.4	×	0.6
	$AG(EFp)$	✓	0.7	×	0.3
	$EF(AGp)$	✓	0.4	×	0.5
`Software Updates` (`35 LOC`)	$p \rightarrow EFq$	✓	0.6	×	0.2
	$p \wedge EGq$	×	0.3	×	0.4
	$p \rightarrow AFq$	×	0.2	×	0.2
	$p \wedge AGq$	×	0.3	×	0.2

TABLE 4.1: CTL verification on industrial benchmarks

prove the property in Columns 3 and 4, and the result returned by our tool and the time it took to discover a counterexample for the negated property in Columns 5 and 6. The variables p and q in Column 2 range over the theory of quantifier-free linear integer arithmetic. The symbol ✓ marks the cases where E-HSF was able to find a solution, i.e., a proof that the CTL property φ is valid, and the symbol × marks the cases where E-HSF was able to find a counter-example, i.e., a proof that the negated CTL property $\neg\varphi$ is not valid. The number of LOC of each program is also given in Column 1.

Our method is able to find proofs that the CTL property φ is valid and the negated CTL property $\neg\varphi$ is not valid for all of the programs except the last three programs. For the last three versions of `Software Updates`, not only the negated CTL property $\neg\varphi$ but also the CTL property φ is not valid. This was because φ was satisfied only for some initial states. Our method takes a total of 427 seconds of which 412 seconds

(96.5 % of the total time) is spent on the programs from `Windows OS fragment 3` and `Windows OS fragment 4`. These programs are relatively big as compared to the rest of the programs and their transition relations will have large control flow graphs.

Program P	Property φ	$P \models_{CTL} \varphi$		$P \models_{CTL} \neg\varphi$	
		Result	Time(s)	Result	Time(s)
`Windows OS fragment 1` (29 LOC)	$AG(p \rightarrow AFq)$	✓	0.3	×	0.3
	$EF(p \wedge EGq)$	✓	0.3	×	0.3
	$AG(p \rightarrow EFq)$	✓	0.3	×	0.3
	$EF(p \wedge AGq)$	✓	0.3	×	0.3
`Windows OS fragment 2` (58 LOC)	$AG(p \rightarrow AFq)$	✓	0.4	×	0.3
	$EF(p \wedge EGq)$	✓	0.4	×	0.3
	$AG(p \rightarrow EFq)$	✓	0.4	×	0.3
	$EF(p \wedge AGq)$	✓	0.4	×	0.3
`Windows OS fragment 3` (370 LOC)	$AG(p \rightarrow AFq)$	✓	0.6	×	1.2
	$EF(p \wedge EGq)$	✓	9.4	×	0.5
	$AG(p \rightarrow EFq)$	✓	0.7	×	0.8
	$EF(p \wedge AGq)$	✓	0.9	×	1.1
`Windows OS fragment 4` (380 LOC)	$AFp \vee AFq$	✓	5.7	×	5.2
	$EGp \wedge EGq$	✓	0.3	×	1.0
	$EFp \wedge EFq$	✓	5.0	×	0.3
	$AGp \vee AGq$	✓	0.3	×	6.4
`Windows OS fragment 5` (43 LOC)	$AG(AFp)$	✓	0.3	×	0.3
	$EF(EGp)$	✓	0.3	×	0.3
	$AG(EFp)$	✓	0.3	×	0.3
	$EF(AGp)$	✓	0.3	×	0.3
`PostgreSQL pgarch` (70 LOC)	$AG(AFp)$	✓	0.4	×	0.3
	$EF(EGp)$	✓	0.3	×	0.4
	$AG(EFp)$	✓	0.3	×	0.3
	$EF(AGp)$	✓	0.3	×	0.3
`Software Updates` (35 LOC)	$p \rightarrow EFq$	✓	0.6	×	0.2
	$p \wedge EGq$	×	0.3	×	0.4
	$p \rightarrow AFq$	×	0.2	×	0.2
	$p \wedge AGq$	×	0.3	×	0.3

TABLE 4.2: Optimised CTL verification on industrial benchmarks

The exponentially long times for these programs is due to the fact that transition relations of the programs model the control flow symbolically using a program counter variable. Efficient treatment of control flow leads to significant improvements for dealing with programs with large control flow graphs line the ones from `Windows OS fragment 3` and `Windows OS fragment 4`.

We re-do our experiment by turning on the optimization option that applies explicit evaluation of program control flow variables, which was described in Section 3.4.1, and we report the results in Table 4.2. Our method now takes a much improved time of 52 seconds. In addition, the programs from `Windows OS fragment 3` and `Windows OS`

`fragment 4` takes a total time of 39 seconds compared to 412 seconds perviously. The optimization results in an order-of-magnitude performance improvement.

Our method also compares favourably with state-of-art automated CTL verification methods. We present in Table 4.3 the comparison between the our solving algorithm E-HSF and a CTL verification method from Cook [38]. Here also, we use the programs from Table 4.1, however, for the sake of focusing on the comparison, we exclude programs for which the two methods have different outcomes. For each program in Column 1, we report the shape of the property in Column 2. The time it takes E-HSF to prove the property φ is given in Column 3, and the corresponding time for Cook et al. is given in Column 4. Similarly, the time it takes E-HSF to discover a counterexample for the negated property $\neg\varphi$ is given in Column 5, and the corresponding time for Cook et al. is given in Column 6.

Program P	Property φ	$P \models_{CTL} \varphi$		$P \models_{CTL} \neg\varphi$	
		E-HSF	Cook et al.	E-HSF	Cook et al.
`Windows OS fragment 1` `(29 LOC)`	$AG(p \rightarrow AFq)$	0.3	1.0	0.3	1.4
	$EF(p \wedge EGq)$	0.3	0.1	0.3	0.7
	$AG(p \rightarrow EFq)$	0.3	0.1	0.3	0.1
	$EF(p \wedge AGq)$	0.3	0.1	0.3	0.1
`Windows OS fragment 2` `(58 LOC)`	$EF(p \wedge EGq)$	0.4	1.0	0.3	1.2
	$EF(p \wedge AGq)$	0.4	0.8	0.3	0.2
`Windows OS fragment 3` `(370 LOC)`	$AG(p \rightarrow AFq)$	0.6	5.9	1.2	6.2
	$EF(p \wedge EGq)$	9.4	2.3	0.5	6.0
	$AG(p \rightarrow EFq)$	0.7	6.8	0.8	3.4
	$EF(p \wedge AGq)$	0.9	4.7	1.1	3.1
`Windows OS fragment 4` `(380 LOC)`	$AFp \vee AFq$	5.7	18.5	5.2	13.9
	$EGp \wedge EGq$	0.3	13.5	1.0	14.2
	$EFp \wedge EFq$	5.0	14.7	0.3	4.8
	$AGp \vee AGq$	0.3	8.0	6.4	3.7
`Windows OS fragment 5` `(43 LOC)`	$AG(AFp)$	0.3	1.0	0.3	0.2
	$EF(EGp)$	0.3	0.1	0.3	0.0
	$AG(EFp)$	0.3	1.0	0.3	0.0
	$EF(AGp)$	0.3	0.1	0.3	0.1
`PostgreSQL pgarch` `(70 LOC)`	$AG(AFp)$	0.4	2.0	0.3	1.3
	$EF(EGp)$	0.3	0.1	0.4	0.1
	$AG(EFp)$	0.3	2.0	0.3	0.0
	$EF(AGp)$	0.3	2.0	0.3	2.4

TABLE 4.3: Comparison of our results with Cook [38, Figure 11]

From the result, we can see that while E-HSF takes a total of 48 seconds to finish the task, Cook et al. takes a total of 149 seconds. This amounts to an approximate reduction of 70%. There are a few cases where E-HSF takes longer than Cook et al. We suspect that a more efficient modeling of the original c program as a transition system can help our method a lot. The presence of many temporary program variables in the

transition relation which are not involved in any computation of the program can affect the performance of our method.

In general, although E-HSF is a generic algorithm not specific to CTL verification, our method is able to outperform the state-of-art automated CTL verification method.

4.6 Related work

Verification of properties specified in temporal logics such as CTL has been extensively explored for finite-state systems[26, 32, 34, 84]. There has also been studies on the verification of CTL properties for some restricted types of infinite-state systems. Some examples are pushdown processes [113, 119], pushdown games [120], and parameterised systems [51]. For such restricted systems, the standard procedure is to abstract the infinite-state system model into finite-state model and apply the known methods for finite-state systems. But existing abstraction methods usually do not allow reliable verification of CTL properties where alternation between universal and existential modal operators is common. Many methods of proving CTL properties with only universal path quantifiers are known[30, 40]. There also a few methods mainly focused on proving branching-time properties with only existential path quantifiers. One example is the tool Yasm [66] which implements a proof procedure aimed primarily at the non-nested existential subset of CTL. There are also known techniques for proving program termination (resp. non-termination) [23, 41] which is equivalent with proving the CTL formula $AF\ false$ (resp. $EG\ true$)[63].

Banda et al. [9] proposed a CTL verification approach for infinite state reactive systems based on CLP and abstraction of a CTL semantic function. An automatic proof method that supports both universal and existential branching-time modal operators for (possibly infinite-state) programs is proposed in by Cook et al. [39]. The approach is based on reducing existential reasoning to universal reasoning when an appropriate restriction is placed on the the state-space of the system. While this approach comes close to our approach, the refinement procedure for state-space restrictions may make incorrect choices early during the iterative proof search. These choices may limit the choices available later in the search leading to failed proof attempts in some cases.

4.7 Conclusion

In this chapter, we proposed a method of verifying CTL properties with respect to a (possibly infinite-space) program. The method takes a transition system that models

the input program and a CTL formula specifying the property to prove as inputs. It first applies known proof systems to generate forall-exists quantified Horn constraints with well-foundedness conditions by the taking the transition system and the CTL formula. Then, it applies the solving algorithms E-HSF to solve the set of Horn constraints. The defining feature of this approach is the separation of concerns between the encoding and the solving of the verification problem. We also demonstrate the practical applicability of the approach by presenting an experimental evaluation using examples from the PostgreSQL database server, the SoftUpdates patch system, the Windows OS kernel.

Chapter 5

CTL+FO Verification as Horn Constraint Solving

5.1 Introduction

In specifying the correct behaviour of systems, relating data at various stages of a computation is often crucial. Examples include program termination [41] (where the value of a rank function should be decreasing over time), correctness of reactive systems [74] (where each incoming request should be handled in a certain timeframe), and information flow [72] (where for all possible secret input values, the output should be the same). The logic CTL+FO offers a natural specification mechanism for such properties, allowing to freely mix temporal and first-order quantification. First-order quantification makes it possible to specify variables dependent on the current system state, and temporal quantifiers allow to relate this data to system states reached at a later point.

While CTL+FO and similar logics have been identified as a specification language before, no fully automatic method to check CTL+FO properties on infinite-state systems was developed. Hence, the current state of the art is to either produce verification tools specific to small subclasses of properties, or using error-prone program modifications that explicitly introduce and initialize ghost variables, which are then used in (standard) CTL specifications.

In this chapter, we present a fully automatic procedure to transform a CTL+FO verification problem into a system of forall-exists quantified recursive Horn constraints. Such systems can then be solved by applying the solving algorithm E-HSF, allowing to blend first-order and temporal reasoning. Our method benefits from the simplicity of the

proposed proof rules and the ability to leverage on-going advances in Horn constraint solving.

The rest of the chapter is organised as follows. We start by briefly revising the syntax and semantics of CTL in Section 5.2. In Section 5.3, we present our proof system that generates a set of forall-exists quantified Horn constraints for a given verification problem. We give our constraint generation procedure together with an illustration on an example in Section 5.4. The experimental evaluation of our method is given in Section 5.5. Finally, we present a bief discussion on related works in Section 5.6 and concluding remarks in Section 5.7.

5.2 CTL+FO basics

The following definitions are standard, see e.g. [21, 81]. Let $\mathcal{T}$ be a first order theory and $\models_{\mathcal{T}}$ denote its satisfaction relation that we use to describe sets and relations over program states. Let c range over assertions in $\mathcal{T}$ and x range over variables. A CTL+FO formula φ is defined by the following grammar using the notion of a path formula ϕ.

$$\varphi ::= \forall x : \varphi \mid \exists x : \varphi \mid c \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid A\,\phi \mid E\,\phi$$
$$\phi ::= X\varphi \mid G\varphi \mid \varphi U \varphi$$

As usual, we define $F\varphi = (\mathit{true} U \varphi)$. The satisfaction relation $P \models \varphi$ holds if and only if for each s such that $\mathit{init}(s)$ we have $P, s \models \varphi$. We define $P, s \models \varphi$ as follows using an auxiliary satisfaction relation $P, \pi \models \phi$. Note that d ranges over values from the corresponding domain.

$$\begin{array}{ll}
P, s \models \forall x : \varphi & \text{iff for all } d \text{ holds } P, s \models \varphi[d/x] \\
P, s \models \exists x : \varphi & \text{iff exists } d \text{ such that } P, s \models \varphi[d/x] \\
P, s \models c & \text{iff } s \models_{\mathcal{T}} c \\
P, s \models \varphi_1 \wedge \varphi_2 & \text{iff } P, s \models \varphi_1 \text{ and } P, s \models \varphi_2 \\
P, s \models \varphi_1 \vee \varphi_2 & \text{iff } P, s \models \varphi_1 \text{ or } P, s \models \varphi_2 \\
P, s \models A\,\phi & \text{iff for all } \pi \in \Pi_P(s) \text{ holds } P, \pi \models \phi \\
P, s \models E\,\phi & \text{iff exists } \pi \in \Pi_P(s) \text{ such that } P, \pi \models \phi \\
P, \pi \models X\varphi & \text{iff } \pi = s_1, s_2, \ldots \text{ and } P, s_2 \models \varphi \\
P, \pi \models G\varphi & \text{iff } \pi = s_1, s_2, \ldots \text{for all } i \geq 1 \text{ holds } P, s_i \models \varphi \\
P, \pi \models \varphi_1 U \varphi_2 & \text{iff } \pi = s_1, s_2, \ldots \text{ and exists } j \geq 1 \text{ such that} \\
 & \qquad P, s_j \models \varphi_2 \text{ and } P, s_i \models \varphi_1 \text{ for } 1 \leq i < j
\end{array}$$

For CTL formula $\psi(v)$ and a transition system $(p(v), next(v, v'))$, find an assertion $aux(v, x)$ such that:

$$\frac{p(v) \rightarrow aux(v,x) \quad (aux(v,x), next(v,v') \wedge x' = x) \models_{CTL+FO} \psi(v,x)}{(p(v), next(v,v')) \models_{CTL+FO} \forall x : \psi(v,x)}$$

FIGURE 5.1: Proof rule RULECTLFOUNIV

In this chapter, we represent a satisfaction relation $P \models \varphi$ by the relation $P \models_{CTL+FO} \varphi$ to explicitly indicate that φ is a CTL+FO formula. We call such relation a CTL+FO satisfaction, and φ is said to be its formula.

5.3 Proof system

The proof system for CTL+FO extends the proof system from CTL given in Section 4.3 by adding two proof rules for handling first-order quantification. The additional proof rules handle CTL+FO satisfactions whose formulas have first-order quantifications as their outer-most operators. Example of such satisfactions are $(init(v), next(v, v')) \models_{CTL+FO} \forall x : \phi(v, x)$ and $(init(v), next(v, v')) \models_{CTL+FO} \exists x : \phi(v, x)$, whose formula has universal and existential quantifiers respectively.

Handling First-Order Universal Quantification Let us consider the satisfaction $(p(v), next(v, v')) \models_{CTL+FO} \forall x : \psi(v, x)$. Its formula $\forall x : \psi(v, x)$ is obtained from the formula $\psi(v, x)$ by universally quantifying over the variable x. In our proof system, we have the proof rule RULECTLFOUNIV, which is given in Figure 5.1, for handling first-order universal quantification. A first constraint connects the set of states $p(v)$ on which $\forall x : \psi(v, x)$ needs to hold with the set of states $aux(v, x)$ on which $\psi(v, x)$ needs to hold. The constraint requires that for every state s such that $s \models p(v)$ and for an arbitrary value s_x of x, the extension of s with s_x is in $aux(v, x)$, i.e., $(s, s_x) \models aux(v, x)$. The auxiliary predicate $aux(v, x)$ has its variable x implicitly assigned an arbitrary value. Then, more constraints are generated for the satisfaction $(aux(v, x), next(v, v') \wedge x' = x) \models_{CTL+FO} \psi(v, x)$ that keep track of satisfaction of ψ from the set of states in represented by $aux(v, x)$. Since the value of x is arbitrary but fixed within $\psi(v, x)$, we require that the transition relation does not modify x and thus extend $next$ to $next(v, v') \wedge x' = x$.

For CTL formula $\psi(v)$ and a transition system $(p(v), next(v, v'))$, find an assertion $aux(v, x)$ such that:

$$\frac{p(v) \rightarrow \exists x : aux(v, x) \quad (aux(v, x), next(v, v') \wedge x' = x) \models_{CTL+FO} \psi(v, x)}{(p(v), next(v, v')) \models_{CTL+FO} \exists x : \psi(v, x)}$$

FIGURE 5.2: Proof rule RULECTLFOEXIST

Handling First-Order Existential Quantification Let us consider the satisfaction $(p(v), next(v, v')) \models_{CTL+FO} \exists x : \psi(v, x)$, where the formula $\exists x : \psi(v, x)$ is obtained from the formula $\psi(v, x)$ by existentially quantifying over the variable x. In our proof system, we have the proof rule RULECTLFOEXIST, which is given in Figure 5.2, for handling first-order existential quantification. We use an auxiliary predicate $aux(v, x)$ that implicitly serves as witness for x. A first constraint connects the set of states $p(v)$ on which $\exists x : \psi(v, x)$ needs to hold with the set of states $aux(v, x)$ on which $\psi(v, x)$ needs to hold. We require that for every state s in $p(v)$, a choice of value s_x of x exists such that the extension of s with s_x is in $aux(v, x)$. Then, more constraints are generated for the satisfaction $(aux(v, x), next(v, v') \wedge x' = x) \models_{CTL+FO} \psi(v, x)$ that keep track of satisfaction of ψ on arbitrary x allowed by $aux(v, x)$. As the value of x is restricted but fixed within $\psi(v, x)$, we require here also that the transition relation does not modify x and thus extend $next$ to $next(v, v') \wedge x' = x$. Thus, $aux(v, x)$ serves as a restriction of the choices allowed for x.

5.4 Constraint generation

The contraint generation procedure performs a top-down, recursive descent through the syntax tree of the given CTL+FO formula. At each level of recursion, the procedure takes as input an satisfaction $(p(v), next(v, v')) \models_{CTL+FO} \varphi$, where φ is a CTL+FO formula, and assertions $p(v)$ and $next(v, v')$ describe a set of states and a transition relation, respectively. The tuple v contains variables that are considered to be in scope and define a state. We assume that variables bound by first-order quantifiers in φ do not shadow other variables.

The constraint generation procedure applies proof rules from the proof system presented in the previous section to recursively decompose complex satisfactions and eventually generate Horn constraints. Before starting the actual constraint generation, the procedure recursively re-writes the input satisfaction of a given CTL+FO formula with arbitrary structure into a set of satisfactions of simple CTL+FO formulas where each

simple formula is either a basic state formula or an assertion over the background theory. The procedure then takes each satisfaction involving simple formula, introduces auxiliary predicates and generates a sequence of forall-exists quantified Horn constraints and well-foundedness constraints (when needed) over these predicates.

Complexity and Correctness The procedure performs a single top-down descent through the syntax tree of the given CTL+FO formula φ. The run time and the size of the generated constraints is linear in the size of φ. Finding a solution for the generated Horn constraints is undecidable in general. In practice however, our solving algorithm E-HSF often succeeds in finding a solution (see Section 5.5). We formalize the correctness of the constraint generation procedure in the following theorem.

Theorem 4. *For a given program P with $init(v)$ and $next(v, v')$ over v and a* CTL+FO *formula φ the Horn constraints generated from $(p(v), next(v, v')) \models_{CTL+FO} \varphi$ are satisfiable if and only if $P \models \varphi$.*

Proof. Formally, we prove that the constraints generated for the satisfaction $(init(v), next(v, v')) \models_{CTL+FO} \varphi$ have a solution if and only if the program $P = (init(v), next(v, v'))$ satisfies φ. We proceed by structural induction on the formula φ. The base case, i.e., φ is an assertion c from our background theory $\mathcal{T}$, is trivial. We show here the cases where $\varphi(v) = \exists x : \psi(v, x)$ or $\varphi(v) = \forall x : \psi(v, x)$. The proof for the rest of the cases can be found in [81].

Let us consider the case $\varphi(v) = \exists x : \psi(v, x)$. To prove soundness, we assume that the generated constraints have a solution. For the predicate $aux(v, x)$, this solution is a relation S_{aux} that satisfies all constraints generated for $aux(v, x)$. For each s with $init(s)$, we choose $\overline{x}_s$ such that $(s, \overline{x}_s) \in S_{aux}$. As we require $init(v) \rightarrow \exists x : aux(v, x)$, this element is defined. We now apply the induction hypothesis for $P' = (aux(v, x), next(v, v') \wedge x' = x)$ and $\psi(v, x)$. Then for all s with $init(s)$, we have $P', (s, \overline{x}_s) \models \psi$, and as P' is not changing x by construction, also $P', (s, \overline{x}_s) \models \psi[\overline{x}_s/x]$. From this, $P, s \models \varphi$ directly follows.

For completeness, we proceed analogously. If $P \models \varphi$ holds, then a suitable instantiation $\overline{x}_s$ of x can be chosen for each s with $init(s)$, and thus we can construct a solution for $aux(v, x)$ from $init(v)$.

Similarly, we consider the case $\varphi(v) = \forall x : \psi(v, x)$. To prove soundness, we assume that the generated constraints have a solution. For the predicate $aux(v, x)$, this solution is a relation S_{aux} that satisfies all constraints generated for $aux(v, x)$. For each s with $init(s)$ and any instantiation $\overline{x}_s$ of x, we have $(s, \overline{x}_s) \in S_{aux}$. As we require $init(v) \rightarrow \forall x : aux(v, x)$, this element is well-defined. We now apply the induction hypothesis for

$P' = (aux(v,x), next(v,v') \wedge x' = x)$ and $\psi(v,x)$. Then for all s with $init(s)$, we have $P', (s, \overline{x}_s) \models \psi$, and as P' is not changing x by construction, also $P', (s, \overline{x}_s) \models \psi[\overline{x}_s/x]$. From this, $P, s \models \varphi$ directly follows.

For completeness, we proceed analogously. If $P \models \varphi$ holds, then any instantiation $\overline{x}_s$ of x can be chosen for each s with $init(s)$, and thus we can construct a solution for $aux(v,x)$ from $init(v)$. □

Example We illustrate the constraint generation procedure on a simple example. We consider a property that the value stored in a register v can grow without bound on some computation.

$$\forall x : v = x \rightarrow EF(v > x)$$

This property can be useful for providing evidence that a program is actually vulnerable to a denial of service attack. Let $init(v)$ and $next(v,v')$ describe a program over a single variable v. We apply the constraint generation procedure on the satisfaction $(init(v), next(v,v')) \models_{CTL+FO} \forall x : v = x \rightarrow EF(v > x)$.

The procedure first applies RuleCtlFoUniv to reduce the input satisfaction into:

$$\begin{aligned} &init(v) \rightarrow aux(v,x), \\ &(aux(v,x), next(v,v') \wedge x' = x) \models_{CTL+FO} v = x \rightarrow EF(v > x) \end{aligned}$$

The procedure goes on to decompose the formula $v = x \rightarrow EF(v > x)$ which is in the new entailment. This gives:

$$\begin{aligned} &init(v) \rightarrow aux(v,x), \\ &aux(v,x) \wedge v = x \rightarrow p(v,x), \\ &(p(v,x), next(v,v') \wedge x' = x) \models_{CTL+FO} EF(v > x) \end{aligned}$$

Finally, the constraints for the new satisfaction $(p(v,x), next(v,v') \wedge x' = x) \models_{CTL+FO} EF(v > x)$ are generated. This gives us the final set of Horn constraints generated for

the original input satisfaction.

$$
\begin{aligned}
&\mathit{init}(v) \rightarrow \mathit{aux}(v, x),\\
&\mathit{aux}(v, x) \wedge v = x \rightarrow p(v, x),\\
&p(v, x) \rightarrow \mathit{inv}(v, x),\\
&\mathit{inv}(v, x) \wedge \neg(v > x) \rightarrow \exists v', x' : \mathit{next}(v, x, v', x') \wedge \mathit{inv}(v', x') \wedge \mathit{rank}(v, x, v', x'),\\
&\mathit{rank}(v, x, v', x') \rightarrow \mathit{ti}(v, x, v', x'),\\
&\mathit{ti}(v, x, v', x') \wedge \mathit{rank}(v', x', v'', x'') \rightarrow \mathit{ti}(v, x, v'', x''),\\
&\mathit{dwf}(\mathit{ti}).
\end{aligned}
$$

Note that there exists an interpretation of $\mathit{aux}(w)$, $p(w)$, $\mathit{inv}(w)$, and $\mathit{rank}(w, w')$ and $\mathit{ti}(w, w')$, where $w = (v, x)$, that satisfies these constraints if and only if the program satisfies the property.

5.5 Evaluation

We evaluate our method by applying the implementation of the solving algorithm E-HSF on a subset of the industrial benchmarks from Section 4.5. The benchmark programs considered in this evaluation are `Windows OS fragment 1`, `Windows OS fragment 2`, `Windows OS fragment 3`, and `Windows OS fragment 4`. Remember that for each of these programs, we have considered four slightly different versions for evaluating our CTL verification method. For each such program P which is given with a CTL property φ, we have modified the property to lift the CTL formula to CTL+FO. As an example, consider a CTL property $AG(a = 1 \rightarrow AF(r = 1))$. One modified property to check could be $\exists x : AG(a = x \rightarrow AF(r = 1))$, and another one is $AG(\exists x : (a = x \rightarrow AF(r = 1)))$. Note that both of these properties are in CTL+FO and they are satisfiability-preserving with respect to the original CTL property. By doing similar satisfiability-preserving transformations of the properties for all the example programs, we get a set programs whose properties are specified in CTL+FO as shown in Table 5.1. For each pair of a program and CTL+FO property φ, we generated two verification tasks: $P \models_{CTL+FO} \varphi$ and $P \models_{CTL+FO} \neg\varphi$. While the existence of a proof for a property φ implies that $\neg\phi$ is violated by the same program, we consider both properties to show the correctness of our method.

As in Section 4.5, linear templates are sufficiently expressive for dealing with queries with existentially quantified variables as well as well-foundedness relations.

Program P	Property φ	$P \models_{CTL} \varphi$		$P \models_{CTL} \neg\varphi$	
		Result	Time	Result	Time
`Windows OS fragment 1 (29 LOC)`	$\exists x : AG(p \rightarrow AFq)$	✓	0.2	×	0.3
	$AG(\exists x : p \rightarrow AFq)$	✓	0.2	×	0.3
	$\exists x : EF(p \wedge EGq)$	✓	0.3	×	0.3
	$EF(\exists x : p \wedge EGq)$	✓	0.3	×	0.2
	$\exists x : AG(p \rightarrow EFq)$	✓	0.3	×	0.3
	$AG(\exists x : p \rightarrow EFq)$	✓	0.3	×	0.3
	$\exists x : EF(p \wedge AGq)$	✓	0.4	×	0.3
	$EF(\exists x : p \wedge AGq)$	✓	0.3	×	0.3
`Windows OS fragment 2 (58 LOC)`	$\exists x : AG(p \rightarrow AFq)$	✓	0.3	×	0.3
	$AG(\exists x : p \rightarrow AFq)$	✓	0.3	×	0.3
	$\exists x : EF(p \wedge EGq)$	✓	0.4	×	0.3
	$EF(\exists x : p \wedge EGq)$	✓	0.4	×	0.4
	$\exists x : AG(p \rightarrow EFq)$	✓	0.4	×	0.4
	$AG(\exists x : p \rightarrow EFq)$	✓	0.4	×	0.3
	$\exists x : EF(p \wedge AGq)$	✓	0.5	×	0.3
	$EF(\exists x : p \wedge AGq)$	✓	0.4	×	0.3
`Windows OS fragment 3 (370 LOC)`	$\exists x : AG(p \rightarrow AFq)$	✓	0.2	×	1.7
	$AG(\exists x : p \rightarrow AFq)$	✓	0.2	×	1.0
	$\forall x : EF(p \wedge EGq)$	✓	18.2	×	0.6
	$EF(\forall x : p \wedge EGq)$	✓	18.3	×	0.6
	$\exists x : AG(p \rightarrow EFq)$	✓	0.7	×	1.4
	$AG(\exists x : p \rightarrow EFq)$	✓	0.7	×	0.8
	$\forall x : EF(p \wedge AGq)$	✓	2.5	×	1.8
	$EF(\forall x : p \wedge AGq)$	✓	1.7	×	1.1
`Windows OS fragment 4 (380 LOC)`	$\exists x : AFp \vee AFq$	✓	0.2	×	0.3
	$\exists x : EGp \wedge EGq$	✓	0.3	×	0.9
	$\exists x : EFp \wedge EFq$	✓	6.0	×	0.3
	$\exists x : AGp \vee AGq$	✓	0.2	×	45.2

TABLE 5.1: CTL+FO verification on industrial benchmarks

We run our experiment by turning on the optimization option that applies explicit evaluation of program control flow variables, which was described in Section 3.4.1, and we report the results in Table 5.1. For each program in Column 1, we report the shape of the property in Column 2, the result returned by our tool and the time it took to prove the property in Columns 3 and 4, and the result returned by our tool and the time it took to discover a counterexample for the negated property in Columns 5 and 6. The variables p and q in Column 2 range over the theory of quantifier-free linear integer arithmetic. The symbol ✓ marks the cases where E-HSF was able to find a solution, i.e., a proof that the CTL property φ is valid, and the symbol × marks the cases where E-HSF was able to find a counter-example, i.e., a proof that the negated CTL property $\neg\varphi$ is not valid. The number of LOC of each program is also given in Column 1. Our method is able to find proofs that the CTL+FO property φ is valid and the negated CTL+FO property $\neg\varphi$ is not valid for all of the programs.

5.6 Related work

Verification of CTL+FO and its decidability and complexity have been studied (under various names) in the past. Bohn et al. [21] presented the first model-checking algorithm. Predicates partitioning a possibly infinite state space are deduced syntactically from the checked property, and represented symbolically by propositional variables. This allows to leverage the efficiency of standard BDD-based model checking techniques, but the algorithm fails when the needed partition of the state space is not syntactically derivable from the property.

Working on finite-state systems, Hallé et al. [67], Patthak et al. [98] and Rensink [106] discuss a number of different techniques for quantified CTL formulas. In these works, the finiteness of the data domain is exploited to instantiate quantified variables, thus reducing the model checking problem for quantified CTL to standard CTL model checking.

Hodkinson et al. [74] study the decidability of CTL+FO and some fragments on infinite state systems. They show the general undecidability of the problem, but also identify certain decidable fragments. Most notably, they show that by restricting first order quantifiers to state formulas and only applying temporal quantifiers to formulas with at most one free variable, a decidable fragment can be obtained. Finally, Da Costa et al. [45] study the complexity of checking properties over propositional Kripke structures, also providing an overview of related decidability and complexity results. In temporal epistemic logic, Belardinelli et al. [10] show that checking FO-CTLK on a certain subclass of infinite systems can be reduced to finite systems. In contrast, our method directly deals with quantification over infinite domains.

5.7 Conclusion

This chapter presented an automated method for proving program properties written in the temporal logic CTL+FO, which combines universal and existential quantification over time and data. Our approach relies on a constraint generation algorithm that follows the formula structure to produce constraints in the form of Horn constraints with forall/exists quantifier alternation. The obtained constraints can be solved using an off-the-shelf constraint solver, thus resulting in an automatic verifier.

Chapter 6

Solving Games on Infinite Graphs as Horn Constraint Solving

6.1 Introduction

Many fundamental questions in formal methods reduce to computing winning strategies in *turn-based graph games* [57], i.e., games where two players take turns in moving a token along the edges of a graph, and a player wins if the sequence of nodes visited by the token satisfies a certain ω-regular *winning condition*. For example:

- To synthesize a reactive system from a temporal specification [25, 103, 116], one constructs a graph game where the goal of one player is to satisfy the specification and the goal of the other is to violate it. The desired system is realizable if and only if the first player has a winning strategy in this game.
- The problem of verifying a branching-time property of a system is naturally framed as a graph game [50]. Here, one player models the existential path quantifiers in the property; the other player models the universal quantifiers. The system satisfies the property if and only if the existential player has a winning strategy.
- Graph games are a natural model for "open" systems [83] that explicitly model interactions between a controller (one player) and its environment (the other player). To prove such a system correct, we show that the controller has a strategy to enforce its requirements no matter how the environment behaves.

There is a rich literature on algorithmic approaches to graph games motivated by applications in formal methods [31, 46, 79, 123]. The majority of these approaches focus

on decidable classes of games, such as games on finite graphs. This focus limits the applications of these techniques. For example, an algorithm that requires a finite game graph can only be applied to the verification and synthesis of finite-state systems. To use games in the analysis and synthesis of infinite-state programs, we need symbolic, abstraction-based algorithms for solving games on the state spaces of such programs. While a few such algorithms exist in the literature [46, 70], much more remains to be done on this topic.

This chapter presents a new approach to this problem space. The main contribution is an algorithmic method based on automated deduction for solving (turn-based) games over infinite-state symbolic transition systems.

Specifically, we target three classes of games over infinite graphs: *safety games*, *reachability games*, and *Linear Temporal Logic (LTL) games* [57]. These games differ in the winning condition for the player for whom we are computing a winning strategy (call this player Eve; the other player is called Adam). In a safety game, Eve wins a play (an infinite sequence of nodes visited by the game token) if and only if the play avoids a certain "unsafe" set of nodes. In a reachability game, a play is winning for Eve if and only if it reaches a certain target set of nodes. In LTL games, Eve wins a play if and only if the play satisfies an LTL property. We note that, LTL games subsume parity games, an important class of games where each node of the game graph is labeled with a "color" from the set $\{1, \ldots, N\}$, and a play is winning for Eve if and only if the minimum color seen infinitely often in the play is odd.

The importance of solving the above types of games to formal methods is well-established in the literature. For instance, the problem of solving a parity game over a program's state space is equivalent to that of verifying program properties written in the modal μ-calculus [50, 57] (note that the μ-calculus subsumes popular temporal logics like LTL, CTL, and CTL*). The solution of LTL games is also at the core of reactive synthesis from temporal specifications. Reachability and safety games are important special cases of LTL games that are sufficient for many applications, including program repair [60, 78], program synthesis [111], synthesis of interface specifications [2], and verification of the fragment of the μ-calculus without alternation of fixpoint quantifiers.

For each of the above types of games, we give a deductive proof rule that, given a symbolic representation of the game graph, symbolically represents a winning strategy in the game using forall-exists quantified Horn constraints. The rule is then automated by applying our solving algorithm E-HSF as an engine for automated deduction.

To understand how our rules work, consider a safety game where the objective of Eve is to satisfy the state property p at all points in all plays. To find a winning strategy for

Eve, our rule for safety games computes an invariant *inv* that describes the set of states from which Eve can win the game. This invariant needs to satisfy the following criteria: (a) the initial condition of the game implies *inv*; (b) *inv* implies p; and (c) for all Adam transitions out of *inv* (let us say to a destination state σ), σ satisfies p and there is a Eve transition from σ back to *inv*.

Strategy computation in reachability games relies on well-founded transition invariants [105] to guarantee that a target state is reached after a finite number of rounds of the game. We solve LTL games with temporal objective φ by converting $\neg\varphi$ into a nondeterministic Büchi automaton, then performing a fair termination check on the product of this automaton and the game graph.

All of our rules are sound, meaning that if they derive a strategy for a player, then the player actually wins under the strategy, as well as relatively complete, meaning that they can always derive a winning strategy when one exists, assuming a suitably powerful assertion language.

From a practical point of view, the appeal of our rules is that they leverage the most recent developments in SMT-solving, invariant generation, and termination verification [14, 59]. Specifically, given a symbolic representation of the game graph, our method generates a set of forall-exists quantified Horn constraints (together with well-foundedness conditions when they are required) which are then fed to the E-HSF engine. Solving the game now amounts to resolving these Horn clauses to a bounded depth, proving the unsatisfiability of the resolvent, repeating the process and generalizing from proofs of unsatisfiability to a solution for the original clauses. The E-HSF engine does so using a combination of counterexample-guided abstraction-refinement (CEGAR), interpolation, and SMT solving, and with help from user-provided templates that capture high-level intuitions about the strategy.

We evaluate our method using several challenging case studies, including the "Cinderella-Stepmother game" — an existing challenge problem for infinite-state graph games that allows infinite alternation of discrete and continuous choices by the two players — and games arising out of prior work on program repair [78] and synthesis [118].

Now we summarize the main contributions of the chapter:

- We take on the problem of solving games over state spaces of infinite-state programs using the power of modern automated software analysis technology.
- We present three deductive proof rules for solving such games under the safety, reachability, and LTL winning conditions. Our rules are sound and relatively

complete, and our automata-theoretic rule for LTL games avoids the need to determinize a Büchi automaton.

- We offer a prototype implementation of our rules on top of an existing automated deduction engine. We illustrate the promise of the system through several case studies using examples posed in prior work.

This chapter is organized as follows. In Section 6.2, we describe the Cinderella-Stepmother game as a motivating example. Section 6.3 gives our proof rules for solving games and proves them correct, i.e., sound and relatively complete. Section 6.4 revisits the example from Section 6.2 and applies our rules to variants of it; Section 6.5 presents applications of our rules to repair and synthesis problems from prior work. Section 6.6 presents concrete experimental results. Related work is described in Section 6.7; we conclude with some discussion in Section 6.8.

6.2 The Cinderella-Stepmother game

In this section, we describe a synthesis problem that motivated this work, and that we use in a case study later in the chapter. A version of the problem was previously posed by Rajeev Alur as a challenge problem for the software synthesis community (see Bodlaender et al. [19] and Hurkens et al. [75] for more on the problem).

The problem involves a turn-based game between the mythical Cinderella, and her nemesis, the Stepmother. The game setup involves five buckets arranged in a circle. Each bucket can hold up to c (a constant) units of water; initially, all buckets are empty. In each round of the game, Stepmother brings 1 unit of additional water and splits it among the five buckets. If any of the buckets overflow, Stepmother wins. If not, Cinderella empties two *adjacent* buckets. Cinderella wins if the game goes on forever.

We can model the Cinderella-Stepmother game using the following symbolic transition system. Let v be a set of system variables that represent the amount of water in the five buckets, $v = (b_1, b_2, b_3, b_4, b_5)$. All the buckets are initially empty — this fact is specified as the initial condition

$$\mathit{init}(v) = (b_1 = 0 \wedge b_2 = 0 \wedge b_3 = 0 \wedge b_4 = 0 \wedge b_5 = 0).$$

The transition relation of Stepmother represents a non-deterministic choice of buckets in which 1 unit of additional water is added:

$$\begin{aligned} \mathit{stepmother}(v, v') = (b'_1 + b'_2 + b'_3 + b'_4 + b'_5 = b_1 + b_2 + b_3 + b_4 + b_5 + 1 \\ \wedge\, b'_1 \geq b_1 \wedge \cdots \wedge b'_5 \geq b_5). \end{aligned}$$

The transition relation of the Cinderella player represents a non-deterministic choice of two consecutive buckets that are emptied.

$$\begin{aligned} &\mathit{cinderella}(v, v') = \\ &\bigvee_{i \in \{1 \ldots 5\}} \left(\begin{array}{l} b'_i = 0 \wedge b'_{(i+1)\%5} = 0 \\ \wedge \left(\bigwedge_{j \in \{1..5\}} \left(\begin{array}{c} j \neq i \wedge j \neq (i+1)\%5 \\ \rightarrow b'_j = b_j \end{array} \right) \right) \end{array} \right). \end{aligned}$$

The condition that one of the buckets overflows is described by the assertion

$$\mathit{overflow}(v) = (b_1 > c \vee b_2 > c \vee b_3 > c \vee b_4 > c \vee b_5 > c).$$

Safety game We observe that in the above game, Cinderella wants to enforce a *safety property* — specifically, the property $G(\neg \mathit{overflow}(v))$ — in every play of the game. This property is Cinderella's *winning condition*. Games are classified according to the winning condition of the player for whom we want to compute a strategy. Specifically, suppose we want to compute a strategy for Cinderella. In that case, we are trying to solve a *safety game*.

Reachability game Now suppose we want to compute a strategy for Stepmother instead. We note that the winning condition for Stepmother is the reachability property $F\ \mathit{overflow}(v)$. The game is a *reachability game*.

LTL and parity games It is easy to define generalizations of the game where the winning condition for a player is a general Linear Temporal Logic (LTL) property. Such a game is called an *LTL game*. LTL games are an extremely challenging class of games — the problem of solving such games on finite game graphs is 2EXPTIME-complete [103]. The intuitive reason for this hardness is that it requires a conversion from an LTL formula to a nondeterministic Büchi automaton (an exponential blowup) and then the determinization of this automaton (another exponential blowup).

An important special case of LTL games is *parity games* [57]. Here, each state of the transition system is assigned a color (a number in $\{1, \ldots, N\}$), and the winning condition

for a play is that the minimum color seen infinitely often in the play is odd. (The condition can be stated in LTL in an obvious way.) In Section 6.6, we apply our method on a parity game generalizing our original game.

Discussion From the determinacy of the classes of graph games that we study [91], it follows that for every value of c, either Cinderella or Stepmother has a winning strategy in each of the above games. Now we give some intuitions about what such a winning strategy would look like in the game as originally stated. The discussion of how to automatically solve the problem using our method is postponed until Section 6.4.

First note that if $c < 1.5$ units, then Stepmother wins. Her strategy is as follows: in the first round, she divides 1 unit into two non-adjacent buckets. Then no matter what Cinderella does, there will be a bucket with 0.5 units at the end of the round, and Stepmother can cause a spill in second round by adding 1 unit in that bucket. If $c \geq 3$ units, Cinderella wins: she can just select the buckets in a round-robin order, emptying two buckets in each round, and this strategy is winning no matter what Stepmother does.

The problem becomes more challenging for $1.5 \leq c < 3$. We leave this case as a challenge for the reader — it will soon be apparent that it is highly nontrivial. In such cases, fully automated strategy synthesis seems unrealistic, and computer-*assisted* proofs driven by user-provided hints or templates are more plausible. This is the strategy that our approach takes.

6.3 Proof rules

In this section, we present proof rules for three kind of games: safety, reachability and parity/LTL games. These proof rules are defined with respect to the player Eve and conclude that Eve has a winning strategy by imposing implication and well-foundedness conditions on auxiliary assertions over system variables. For each proof rule we prove its soundness, i.e., a winning strategy exists if the premises are satisfied by some auxiliary assertions, and relative completeness, i.e., if a winning strategy exists then auxiliary assertions satisfying the premises exist under an assumption that the assertion language of our choice is sufficiently expressive. Such correctness criteria are standard for temporal proof rules [87].

Find assertion $inv(v)$ such that:

$$\begin{array}{lll} \text{S1}: & init(v) & \rightarrow inv(v) \\ \text{S2}: & inv(v) \wedge adam(v, v') & \rightarrow safe(v') \wedge \exists v'' : eve(v', v'') \wedge inv(v'') \\ \text{S3}: & inv(v) & \rightarrow safe(v) \end{array}$$

$$(init(v), eve(v, v'), adam(v, v')) \models G\ safe(v)$$

FIGURE 6.1: Proof rule RULESAFE for a safety game.

6.3.1 Safety games

We consider a safety game for which Eve has a winning strategy if only states from $safe(v)$ are visited by all plays, i.e., the winning condition is given by a formula $G\ safe(v)$.

We present the corresponding proof rule in Figure 7.2. The proof rule relies on an invariant assertion $inv(v)$ that represents a set of states reached by Eve in a winning strategy. We connect the invariant assertion with the reachable states by resorting to reasoning by induction on the number of steps to reach a state. The condition S1 requires that the initial state of the game are considered in $inv(v)$. S2 represent the induction step. Here, we require that for every step from $inv(v)$ executed by Adam there exist a step by Eve that leads back to $inv(v)$. Of course, since the winning condition requires that all states of a play need to satisfy $safe(v)$, we require that all states reached after Adam made a step as well as $inv(v)$ satisfy the assertion $safe(v)$. The former condition is enforced by a conjunct $safe(v')$ in the head of S2. The later condition is guaranteed by S3.

Theorem 5 (Correctness of rule RULESAFE)**.** *The proof rule* RULESAFE *is sound and relatively complete.*

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

Soundness We prove the soundness by contradiction. Assume that there exists an assertion $inv(v)$ that satisfies the premises of RULESAFE, yet the conclusion of RULESAFE does not hold. That is, there is no winning strategy for Eve. Hence, there exists a strategy σ for Adam in which each play reaches a state that violates $safe(v)$. This strategy σ alternates between existential choices of Adam and universal choices of Eve. Let $aux(v)$ be a set of states for which σ provides existentially chosen successors wrt. Adam.

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_1 of σ, which also satisfies the initial condition $init(v)$. Note that $s_1 \models inv(v)$, due to S1, and $s_1 \models aux(v)$ due to σ. Each iteration round extends the play obtained so far by two states, say s' and s''. We maintain a condition that each such s'' satisfies $inv(v)$ and $aux(v)$. Let s be the last state of the play π constructed so far. Due to our condition, we have $s \models inv(v) \wedge aux(v)$. Then, σ determines a successor state s' such that $(s, s') \models adam(v, v')$, and S2 guarantees that there exists a state s'' such that $(s', s'') \models eve(v, v')$ and $s'' \models inv(v)$. Furthermore, s'' satisfies $aux(v)$ due to σ. Finally, from S2 and S3 follows that $s' \models safe(v)$ and $s'' \models safe(v)$, respectively.

By iteratively constructing π using the above step we obtain a play that satisfies the strategy σ. Thus, we obtain a contradiction, since according to our construction all states in π satisfies $safe(v)$, however σ guarantees that each play eventually reaches a state that violates $safe(v)$.

Completeness Assume that Eve has a winning strategy, say σ, i.e., the conclusion of RuleSafe holds. We prove the completeness claim by showing how to construct $inv(v)$ that satisfies the premises of RuleSafe.

This strategy σ alternates between universal choices of Adam and existential choices of Eve. Let $inv(v)$ be a set of states for which σ provides universally chosen successors wrt. Adam. Since σ is a winning strategy, all states satisfying $inv(v)$ also satisfy $safe(v)$, i.e., $inv(v)$ satisfies S3. $inv(v)$ satisfies S1, since σ guarantees that Eve wins from every initial state. Now we consider an arbitrary state s that satisfies $inv(v)$. σ guarantees that for every successor s' of s wrt. Adam there exists a successor s'' wrt. Eve such that $s'' \models inv(v)$. Furthermore, since σ is winning, we have $s' \models safe(v)$. Thus we conclude that $inv(v)$ satisfies the condition S2 as well. □

6.3.2 Reachability games

In contrast to safety games, the winning condition of reachability games ensures that a certain set of states called $dst(v)$ is eventually reached by each play, i.e., the winning condition is given by a formula $F\ dst(v)$. Reasoning about such eventuality properties demands the use of well-founded orders.

We present a rule RuleReach for proving that Eve has a winning strategy for a reachability property given by an LTL formula $F dst(v)$ in Figure 6.2. RuleReach requires an invariant assertion $inv(v)$ together with a binary relation $round(v, v')$. Similarly to

Find assertions $inv(v)$ and $round(v, v')$ such that:

$$
\begin{array}{lll}
\text{R1}: & init(v) & \rightarrow inv(v) \\
\text{R2}: & inv(v) \wedge \neg dst(v) \wedge adam(v, v') \wedge \neg dst(v') & \rightarrow \exists v'' : eve(v', v'') \wedge \\
 & & \quad inv(v'') \wedge round(v, v'') \\
\text{R3}: & wf(round(v, v')) &
\end{array}
$$

$$(init(v), eve(v, v'), adam(v, v')) \models F\ dst(v)$$

FIGURE 6.2: Proof rule RULEREACH for a reachability game.

RULESAFE, we use $inv(v)$ to keep track of states that are reached by Eve. This is captured by R1 and a part of R2. To ensure that Adam makes progress when aiming at the set $dst(v)$ we keep track of pairs of states towards reaching it in $round(v, v')$, see the last conjunct in R2. We note that the proof rule only imposes conditions when $dst(v)$ is not yet reached, as encoded by the second conjunct in R2. Finally, to ensure that $dst(v)$ is eventually reached by each play we require that $round(v, v')$ represents a well-founded relation. Thus, it is impossible to return to $inv(v) \wedge \neg dst(v)$ infinitely many times.

Theorem 6 (Correctness of rule RULEREACH)**.** *The proof rule* RULEREACH *is sound and relatively complete.*

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

Soundness We prove the soundness by contradiction. Assume that there exist assertions $inv(v)$ and $round(v, v')$ that satisfy the premises of RULEREACH, yet the conclusion of RULEREACH does not hold. That is, there is no winning strategy for Eve. Hence, there exists a strategy σ for Adam in which each play never reaches a state that satisfies $dst(v)$. This strategy σ alternates between existential choices of Adam and universal choices of Eve. Let $aux(v)$ be a set of states for which σ provides existentially chosen successors wrt. Adam. Note that the implication $aux(v) \rightarrow \neg dst(v)$ is valid, since no play determined by σ visits $dst(v)$.

We derive a contradiction by relying on a certain play π that is determined by σ. The play π is constructed iteratively, in a similar way as done in the proof of Theorem 5. We start from some root state s_1 of σ, which satisfies the initial condition $init(v)$. Note that $s_1 \models inv(v)$, due to R1, and $s_1 \models aux(v)$ due to σ. Each iteration round extends the play obtained so far by two states, say s' and s''. We maintain a condition that each such s'' satisfies $inv(v)$ and $aux(v)$. Let s be the last state of the play π constructed so

far. Due to our condition, we have $s \models \mathit{inv}(v) \wedge \mathit{aux}(v)$. Then, σ determines a successor state s' such that $(s, s') \models \mathit{adam}(v, v')$, and R2 guarantees that there exists a state s'' such that $(s', s'') \models \mathit{eve}(v, v')$ and $s'' \models \mathit{inv}(v)$. Furthermore, s'' satisfies $\mathit{aux}(v)$ due to σ. Finally, from R2 also follows that $s' \models \neg \mathit{dst}(v)$ and $(s, s'') \models \mathit{round}(v, v')$.

By iteratively constructing $\pi = s_1, s_2, \ldots$ using the above step we obtain a play that satisfies the strategy σ. Thus, there is an infinite sequence of states $s_1, s_3, s_5, \ldots$ that takes states occurring at odd positions in π such that each pair of consecutive states s_{2i-1} and s_{2i+1} is connected by $\mathit{round}(v, v')$, for $i \geq 1$. The existence of such an infinite sequence contradicts the well-foundedness condition imposed by R3.

Completeness Assume that Eve has a winning strategy, say σ, i.e., the conclusion of RuleReach holds. We prove the completeness claim by showing how to construct $\mathit{inv}(v)$ and $\mathit{round}(v, v')$ that satisfy the premises of RuleReach.

The strategy σ alternates between universal choices of Adam and existential choices of Eve. Each play $\pi = s_1, s_2, s_3, \ldots$ contributes elements to $\mathit{inv}(v)$ and $\mathit{round}(v, v')$ as follows. Let k be the position of the first occurrence of a state in π that satisfies $\mathit{dst}(v)$, i.e., we have $s_k \models \mathit{dst}(v)$ and $s_i \not\models \mathit{dst}(v)$ for each $i \in 1..k-1$. Such position exists, since the play satisfies $F\mathit{dst}(v)$. Then, for each $i \geq 1$ such that $2i - 1 \leq k$ we add the state s_{2i-1} to $\mathit{inv}(v)$. Furthermore, for each $i \geq 1$ such that $2i + 1 \leq k$ we add the pair of states s_{2i-1} and s_{2i+1} to $\mathit{round}(v, v')$.

We note that the above construction ensures that for each pair of states s and s'' such that $(s, s'') \models \mathit{round}(v, v'')$ holds: i) we have $s \not\models \mathit{dst}(v)$, and ii) there exists a state s' such that $s' \not\models \mathit{dst}(v)$, $(s, s') \models \mathit{adam}(v, v')$, and $(s', s'') \models \mathit{eve}(v, v')$.

We observe that $\mathit{inv}(v)$ satisfies R1, since σ guarantees that Eve wins from every initial state. Now we consider each pair of states s and s' that satisfies the left hand side of R2. σ guarantees that there exists a successor s'' wrt. Eve. Regardless whether $s'' \models \mathit{dst}(v)$ the above construction guarantees that the right-hand side of R2 is satisfied by assigning s, s', and s'' to v, v', and v'', respectively.

Now we show by contradiction that $\mathit{round}(v, v')$ is well-founded. Assume otherwise, i.e., there exists an infinite sequence of states $s_1, s_2, \ldots$ induced by $\mathit{round}(v, v')$. As noted previously, for each pair of consecutive states s_i and s_{i+1} there exists an intermediate state s'_i such that the sequence $s_1, s'_1, s_2, \ldots, s_i, s'_i, s_{i+1}, \ldots$ is a play. Since this play does not visit any state that satisfies $\mathit{dst}(v)$, we obtain a contradiction to the assumption that Eve has a winning strategy. Hence, we conclude that R3 is satisfied. □

6.3.3 LTL and parity games

Now we show how to solve LTL games and, as a special case, parity games. To state the parity winning condition we assume that the set of all states is partitioned into N subsets that are denoted by the assertions $p_1(v), \ldots, p_N(v)$. Thus, $p_1(v) \vee \cdots \vee p_N(v)$ is valid and for each $1 \leq i < j \leq N$ we have that $p_i(v) \wedge p_j(v)$ is unsatisfiable. Without loss of generality we assume that N is an odd number.

The parity condition states that the system wins the game for a given computation if among the subsets $p_{i_1}(v), \ldots, p_{i_K}(v)$ that are visited infinitely many times by the computation the minimal identifier is odd, i.e., $\min\{i_1, \ldots, i_K\}$ is odd. We can represent the parity condition by the following LTL formula φ.

$$\begin{aligned}\varphi \;=\; & GFp_1(v) \\ & \vee GFp_3(v) \wedge FG\neg(p_1(v) \vee p_2(v)) \\ & \cdots \\ & \vee GFp_N(v) \wedge FG\neg(p_1(v) \vee \cdots \vee p_{N-1}(v))\end{aligned}$$

The first disjunct states that $p_1(v)$ is visited infinitely often, while the second disjunct states that $p_3(v)$ is visited infinitely often and there exists a suffix that neither visits $p_1(v)$ nor $p_2(v)$. The last disjunct states that $p_N(v)$ is visited infinitely often and there is a suffix that visits no other subset.

To solve games where the winning condition is an LTL formula φ, we negate φ and apply a standard technique, e.g., [56], for translating LTL formulas to Büchi automata on the resulting $\neg\varphi$. Let $\mathcal{B}$ be the obtained automaton. We represent $\mathcal{B}$ using assertions over the program counter of the automaton $pc_{\mathcal{B}}$ and the system variables v. Let the initial condition of the automaton be given by $init_{\mathcal{B}}(pc_{\mathcal{B}})$. We represent the transition relation of $\mathcal{B}$ by $next_{\mathcal{B}}(pc_{\mathcal{B}}, v, pc'_{\mathcal{B}})$. This transition relation evolves the value of the program counter of the automaton while taking into consideration the current state of the system given by a valuation of v. Finally, we assume that $acc_{\mathcal{B}}(pc_{\mathcal{B}})$ represents the accepting states of the automaton.

Given a sequence of states $\pi = s_1, s_2, \ldots$ we define a run of $\mathcal{B}$ on π to be an infinite sequence of automaton states $q_0, q_1, q_2, \ldots$ such that $q_0 \models init_{\mathcal{B}}(pc_{\mathcal{B}})$ and $(q_{i-1}, s_i, q_i) \models next_{\mathcal{B}}(pc_{\mathcal{B}}, v, pc'_{\mathcal{B}})$ for each $i \geq 1$. A run is accepting wrt. the Büchi acceptance condition if it contains infinitely many states that satisfy $acc_{\mathcal{B}}(pc_{\mathcal{B}})$. The automaton $\mathcal{B}$ accepts a play π if there exists an accepting run on π. Note that our construction ensures that if $\mathcal{B}$ accepts π then $\pi \not\models \varphi$.

Find assertions $inv(w)$, $aux(w, w', v'')$, $round(w, w', w'')$, and $fair(w, w')$ where $w = (v, pc_{\mathcal{B}})$ such that:

$$
\begin{array}{lll}
\text{B1}: & init(v) \wedge init_{\mathcal{B}}(pc_{\mathcal{B}}) \wedge next_{\mathcal{B}}(pc_{\mathcal{B}}, v, pc'_{\mathcal{B}}) & \rightarrow inv(v, pc'_{\mathcal{B}}) \\
\text{B2}: & inv(w) \wedge adam(v, v') \wedge next_{\mathcal{B}}(pc_{\mathcal{B}}, v', pc'_{\mathcal{B}}) & \rightarrow \exists v'' : eve(v', v'') \wedge aux(w, w', v'') \\
\text{B3}: & aux(w, w', v'') \wedge next_{\mathcal{B}}(pc'_{\mathcal{B}}, v'', pc''_{\mathcal{B}}) & \rightarrow inv(w'') \wedge round(w, w', w'') \\
\text{B4}: & round(w, w', w'') \wedge (acc_{\mathcal{B}}(pc_{\mathcal{B}}) \vee acc_{\mathcal{B}}(pc'_{\mathcal{B}})) & \rightarrow fair(w, w'') \\
\text{B5}: & fair(w, w') \wedge round(w', w'', w''') & \rightarrow fair(w, w''') \\
\text{B6}: & wf(fair(w, w')) &
\end{array}
$$

$$(init(v), eve(v, v'), adam(v, v')) \models \varphi$$

FIGURE 6.3: Proof rule BÜCHITERM for an LTL game.

A proof rule BÜCHITERM for LTL games based on the above automata-theoretic approach [117] is presented in Figure 6.3. The winning condition is given by an LTL formula φ. BÜCHITERM requires that the negation of the winning condition is translated into a Büchi automaton $\mathcal{B}$, which together with the system description appears in the proof rule. An interesting property of this proof rule is that it relies on a non-deterministic Büchi automaton representation of the negated winning condition, and does not require any determinization via Rabin, Muller, or Streett acceptance conditions.

We consider a synchronous parallel product of the transition relations of the players and the transition relation of the Büchi automaton, which is expressed in the proof rule by appropriate conjunctions. We use $w = (v, pc_{\mathcal{B}})$ to refer to the vector of the system variables and the program counter of the automaton.

The existence of a winning strategy for Eve depends on the identification of auxiliary assertions $inv(w)$, $aux(w, w', v'')$, $round(w, w', w'')$, and $fair(w, w')$ as follows. $inv(w)$ keeps track of the system states reached by Adam, similarly to RULESAFE and RULEREACH. To deal with the non-determinism in the transition relation of the automaton, we introduce an intermediate book-keeping assertion $aux(w, w', v'')$, which allows us to decouple the treatment of the automaton state q'' from the selection of s''. $round(w, w', w'')$ contains all triples of adjacent program states occurring in plays. Here, it is more fine-grained than the counterpart in RULEREACH, as we keep track of intermediate states visited by Eve instead of only considering the combined steps (visited by Adam). For keeping track of acceptance $fair(w, w')$ contains all pairs of program states that describe play segments visiting Büchi accepting states at least once. We derive $fair(w, w')$ from $round(w, w', w'')$ using transitive closure-like conditions B4 and

B5. Finally, the well-foundedness condition B6 shows that accepting states cannot be visited infinitely many times.

Theorem 7 (Correctness of rule BüchiTerm)**.** *The proof rule* BüchiTerm *is sound and relatively complete.*

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

Soundness We prove the soundness by contradiction. Assume that there exist assertions $inv(w)$, $aux(w, w', v'')$, $round(w, w', w'')$, and $fair(w, w')$ that satisfy the premises of BüchiTerm, yet the conclusion of BüchiTerm does not hold. That is, there is no winning strategy for Eve. Hence, there exists a strategy σ for Adam in which each play violates φ. This strategy σ alternates between existential choices of Adam and universal choices of Eve. We derive a contradiction by relying on a certain set of trees whose branches are sequences $(s_1, q_1), (s_2, q_2), \ldots$ that are jointly determined by σ and the assumed assertions (via BüchiTerm).

The requisite branches are constructed iteratively, in a similar way as the play construction is done in the proof of Theorem 5. We start from some root state s_1 of σ, which satisfies the initial condition $init(v)$. Then the play is extended from a state s by considering an existential choice s' offered by σ that is followed by an existential choice s'' offered by B2. We obtain appropriate runs $q_0, q_1, \ldots$ by applying B1, B2, and B3 for values of v, v', and v'' determined by currently considered s, s', and s'', respectively. Since the automaton $\mathcal{B}$ is non-deterministic, for each s, s', and s'' there is a set of appropriate automaton states. Considering each choice leads to a tree construction, as described below.

First, we consider s_1 and B1, and for each q_1 such that there exists q_0 with $init(s_1) \wedge init_{\mathcal{B}}(q_0) \wedge next_{\mathcal{B}}(q_0, s_1, q_1)$ we add a (s_1, q_1) as a root to our tree. We remember the state q_0 that was used to create each (s_1, q_1). Then, for each tree leaf (s, q) we perform the following tree expansion. First, we consider the state s' that σ provides as a successor of s. Then, we rely on B2, and for each q' such that $next_{\mathcal{B}}(q, s', q')$ holds we add (s', q') as a child node of (s, q). Furthermore, for given s and q, and each s' and q' we take s'' such that $eve(s', s'') \wedge aux(s, q, s', q', s'')$. Now we rely on B3, and for each q'' such that $next_{\mathcal{B}}(q', s'', q'')$ holds we add (s'', q'') as a child node of the corresponding (s', q').

By applying the above tree expansion steps we construct a set of trees where every branch is a sequence $(s_1, q_1), (s_2, q_2), \ldots$ that comes with the corresponding initial automaton state q_0. Note that $s_1, s_2, \ldots$ is a play determined by σ, hence it violates φ. Thus, there

exists a branch for which the sequence $q_0, q_1, \ldots$ is an accepting run of $\mathcal{B}$ for the corresponding play determined by the branch. Let $(s_{i_1}, q_{i_1}), (s_{i_2}, q_{i_2}), \ldots$ be a subsequence such that $q_{i_j} \models \mathit{acc}_{\mathcal{B}}(\mathit{pc}_{\mathcal{B}})$ for each $j \geq 1$. Then, each pair (s_{i_j}, q_{i_j}) contributes

$$((s_{i_j}, q_{i_j}), (s_{i_{j+1}}, q_{i_{j+1}}))$$

(or its closest neighbour visited by Adam) to $\mathit{fair}(w, w')$. Thus, the condition B6 is violated.

Completeness Assume that Eve has a winning strategy, say σ, i.e., the conclusion of BüchiTerm holds. We prove the completeness claim by showing how to construct $\mathit{inv}(w)$, $\mathit{aux}(v, w', v'')$, $\mathit{round}(v, w', w'')$, and $\mathit{fair}(w, w')$ that satisfy the premises of BüchiTerm.

The strategy σ alternates between universal choices of Adam and existential choices of Eve. Each play $\pi = s_1, s_2, s_3, \ldots$ contributes elements to $\mathit{inv}(w)$, $\mathit{aux}(v, w', v'')$, and $\mathit{round}(w, w', w'')$ in the following way through an appropriate sequence of automaton states $q_0, q_1, q_2, \ldots$. Since $\pi \models \varphi$, we note that π is not accepted by $\mathcal{B}$. Hence, either there is an infinite run $q_0, q_1, q_2, \ldots$ that is not accepting, or the exists a finite run $q_0, \ldots, q_n$ that cannot be extended (i.e., there is no automaton state q_{n+1} such that $\mathit{next}_{\mathcal{B}}(q_n, s_n, s_{n+1})$). In either case, for each $i \geq 0$ (and $2i + 1 \leq n$ if the run is finite) we let (s_{2i-1}, q_{2i-1}) be an element of $\mathit{inv}(w)$, $(s_{2i-1}, q_{2i-1}, s_{2i}, q_{2i}, s_{2i+1})$ be an element of $\mathit{aux}(w, w', v'')$, and $(s_{2i-1}, q_{2i-1}, s_{2i}, q_{2i}, s_{2i+1}, q_{2i+1})$ be an element of $\mathit{round}(w, w', w'')$. Then we define $\mathit{fair}(w, w')$ for the obtained $\mathit{round}(w, w', w'')$ as the least solution of B4 and B5.

Since the run is not accepting, it visits accepting states only finitely many times. Hence, $\mathit{fair}(w, w')$ is well-founded. □

6.4 Case study: Cinderella-Stepmother games

In this section we illustrate our constraint-based approach to solving games applying it to the Cinderella-Stepmother game introduced in Section 6.2. We consider five variants of this game corresponding to different winning conditions. In Section 6.6 we report on running times required for solving these games by applying our method.

6.4.1 Games with Cinderella's safety objective

In these games, we attempt to obtain winning strategies for Cinderella in her attempt to keep the buckets from overflowing. The winning condition for Cinderella is $G\ \neg overflow(v)$. As mentioned in Section 6.2, the Cinderella player has simple winning strategies for bucket capacity $c \geq 3$. For values $2 \leq c \leq 3$, the strategies are more involved. (For values $c \leq 2$ there are no strategies for the player Cinderella to win the game).

Round strategy We define the first game using the value $c = 3$ for the bucket capacity. A winning strategy might follow an alternation of consecutive buckets that are emptied. Accordingly, we use an auxiliary variable r for a pair of buckets to be emptied, to remember the previous choice made by the Cinderella player. The tuple of game variables contains the five bucket variables from v and is extended with the round variable r as follows: $w = (b_1, b_2, b_3, b_4, b_5, r)$. The initial states assertion sets the round variable to $r = 1$. We let Adam play the role of Stepmother and therefore the transition relation of Adam is based on the assertion $stepmother(v, v')$, while the transition relation of Eve is given by $cinderella(v, v')$. (Both constraints are given in Section 6.2.)

$$\begin{aligned} init(w) &= (b_1 = 0 \wedge \cdots \wedge b_5 = 0 \wedge r = 1) \\ eve(w, w') &= cinderella(v, v') \\ adam(w, w') &= (stepmother(v, v') \wedge r' = r) \end{aligned}$$

Considering the safety condition $obj(w) = G(\neg overflow(v))$, we instantiate the proof rule from Figure 7.2 as follows.

$$(init(w), adam(w, w'), eve(w, w')) \models obj(w)$$

There exists a strategy for Eve provided that the premises of the proof rule are satisfied. These premises are Horn clauses over the auxiliary assertion $inv(w)$. We apply a solver, e.g., E-HSF, to find a solution for the auxiliary assertion. The clauses S1 and S3 are universally quantified over the game variables, while the existentially quantified clause S2 is skolemized in the E-HSF approach. We use the skolem relation $rel(w, w', w'')$ to denote the witness constraint corresponding to the existentially quantified variables w''.

$$\begin{aligned} inv(w) \wedge stepmother(v, v') \wedge r' = r \wedge rel(w, w', w'') \rightarrow \\ cinderella(v', v'') \wedge inv(w'') \end{aligned}$$

E-HSF requires a template for the skolem relation and we present below the intuition behind this constraint. For each of the five disjuncts from $cinderella(v, v')$ transition relation, we add guards (one guard exclusive to the others) and update the value of the round variable. We use $c_1(v, v')$ to $c_5(v, v')$ to denote the five disjuncts from the transition relation of the Cinderella player introduced in Section 6.2. We obtain the following template constraint.

$$\begin{aligned}\textsc{Templ}(rel)(w, w', w'') = (&r' = 1 \wedge r'' = ?_1 \wedge c_1(v', v'') \vee \\ &r' = 2 \wedge r'' = ?_2 \wedge c_2(v', v'') \vee \\ &r' = 3 \wedge r'' = ?_3 \wedge c_3(v', v'') \vee \\ &r' = 4 \wedge r'' = ?_4 \wedge c_4(v', v'') \vee \\ &r' = 5 \wedge r'' = ?_5 \wedge c_5(v', v''))\end{aligned}$$

The template parameters are denoted by "?"-variables and different subscripts indicate distinct template parameters.

Our approach is able to synthesize automatically the values used to update the round and implicitly the order in which the Cinderella player should alternate emptying the buckets. E-HSF returns the solution $?_1 = 4, ?_2 = 1, ?_3 = 1, ?_4 = 3, ?_5 = 1$. Corresponding to this solution, the strategy for the Cinderella player consists of a repeating sequence of three player moves:

1. Since initially $r = 1$ and the first disjunct is enabled, decide to empty buckets 1 and 2 and update the round variable $r'' = 4$.
2. Since the disjunct $r' = 4 \wedge r'' = ?_4$ is enabled, decide to empty buckets 4 and 5 and update the round variable to $r'' = 3$.
3. Since the disjunct $r' = 3 \wedge r'' = ?_3$ is enabled, decide to empty buckets 3 and 4 and update the round variable to $r'' = 1$.

After these three moves, r has value 1, the first disjunct is again enabled and the strategy will continue with the first move/decision above. This strategy ensures that the Cinderella player empties often enough all the buckets and therefore the Stepmother player cannot enforce an overflow. This game is won by the Cinderella player based on the round strategy described above.

Second strategy We show how our approach can be used to obtain a strategy for the case of the game that is more difficult for Cinderella to win, i.e., $c = 2$.

We fix the roles of the two players similar to the previous paragraph: $eve(v, v') = cinderella(v, v')$ and $adam(v, v') = stepmother(v, v')$. To explain the rationale behind the Cinderella's decisions for this case, we refer to the proof rule from Figure 7.2. We repeat the second clause S2 instantiated for the two players of the C-S game:

$$inv(v) \wedge stepmother(v, v') \rightarrow safe(v') \ \wedge \exists v'' : cinderella(v', v'') \\ \wedge \ inv(v'')$$

To change the state of the system from v' to v'', the strategy for the Cinderella player takes into consideration her previous move (reflected in variables v) and the reply by Stepmother (reflected in variables v'). Therefore the template for the strategy considers five cases depending on which buckets the Cinderella player may have emptied in the previous turn:

$$\begin{aligned} \textsc{Templ}(rel)(v, v', v'') = (&b_1 = 0 \wedge b_2 = 0 \wedge T_{12}(v', v'') \vee \\ &b_2 = 0 \wedge b_3 = 0 \wedge T_{23}(v', v'') \vee \\ &b_3 = 0 \wedge b_4 = 0 \wedge T_{34}(v', v'') \vee \\ &b_4 = 0 \wedge b_5 = 0 \wedge T_{45}(v', v'') \vee \\ &b_5 = 0 \wedge b_1 = 0 \wedge T_{51}(v', v'')). \end{aligned}$$

The T_{ij} conjuncts refer to non-obvious knowledge and relate to an invariant stating that each pair of non-adjacent buckets should have total contents at most 1 [75]. The first part of the template, i.e., T_{12}, is based on the intuition that if in the previous round Cinderella emptied buckets 1 and 2 ($b_1 = 0 \wedge b_2 = 0$), then during the next round she will decide to empty another pair of buckets. That is, either the pair of buckets 3 and 4 ($b_3'' = 0 \wedge b_4'' = 0$) or the pair of buckets 4 and 5 ($b_4'' = 0 \wedge b_5'' = 0$) will be emptied. However, the condition on which to decide if to empty buckets 3 and 4 or buckets 4 and 5 is not straightforward. We use template parameters and leave the decision to be automated by our game solving approach. The formula T_{12} is provided as follows.

$$\begin{aligned} T_{12}(v', v'') = (&?_5 * b_5' + ?_2 * b_2' \leq 1 \wedge b_3'' = 0 \wedge b_4'' = 0 \vee \\ &?_1 * b_1' + ?_3 * b_3' \leq 1 \wedge b_4'' = 0 \wedge b_5'' = 0) \end{aligned}$$

Following a similar argument, we obtain the formulas that complete the definition of the template $\textsc{Templ}(rel)(v, v', v'')$:

$$\begin{aligned}
T_{23}(v', v'') = (?_1 * b'_1 + ?_3 * b'_3 \leq 1 \wedge b''_4 = 0 \wedge b''_5 = 0 \vee \\
?_2 * b'_2 + ?_4 * b'_4 \leq 1 \wedge b''_5 = 0 \wedge b''_1 = 0) \\
T_{34}(v', v'') = (?_2 * b'_2 + ?_4 * b'_4 \leq 1 \wedge b''_5 = 0 \wedge b''_1 = 0 \vee \\
?_3 * b'_3 + ?_5 * b'_5 \leq 1 \wedge b''_1 = 0 \wedge b''_2 = 0) \\
T_{45}(v', v'') = (?_3 * b'_3 + ?_5 * b'_5 \leq 1 \wedge b''_1 = 0 \wedge b''_2 = 0 \vee \\
?_4 * b'_4 + ?_1 * b'_1 \leq 1 \wedge b''_2 = 0 \wedge b''_3 = 0) \\
T_{51}(v', v'') = (?_4 * b'_4 + ?_1 * b'_1 \leq 1 \wedge b''_2 = 0 \wedge b''_3 = 0 \vee \\
?_5 * b'_5 + ?_2 * b'_2 \leq 1 \wedge b''_3 = 0 \wedge b''_4 = 0).
\end{aligned}$$

The template parameters are marked as before by "?"-variables and we aim to obtain solutions for the five template parameters $?_1, ?_2, ?_3, ?_4, ?_5$. Our approach is indeed able to synthesize automatically values for these parameters. The E-HSF engine returns the solutions $?_1 = 1, ?_2 = 1, ?_3 = 1, ?_4 = 1, ?_5 = 1$. The resulting strategy for the Cinderella player guarantees that no state with overflow can be reached. For a different perspective, we refer the interested reader to an article on (non-automated) reasoning and invariants needed to establish strategies for the Cinderella-Stepmother game similar to the ones we synthesize [75].

6.4.2 Game with Stepmother's reachability objective

We continue illustrating our approach with the Cinderella-Stepmother game, this time based on a reachability objective: the winning condition for the Stepmother player requires that a state with overflow is reached, $obj(v) = F\ overflow(v)$. For this game, we use the bucket capacity $c = 1.4$, a value for which the Stepmother has indeed a winning strategy. To derive this strategy, we instantiate the proof rule for the reachability game as follows.

$$\begin{aligned}
init(v) &= (b_1 = 0 \wedge \cdots \wedge b_5 = 0) \\
eve(v, v') &= stepmother(v, v') \\
adam(v, v') &= cinderella(v, v')
\end{aligned}$$

Next we provide a template corresponding to the existentially quantified clause. The insight behind the template is that the quantity of water from each bucket increases during the turn of Stepmother, but without specifying the amount, i.e., $(b''_i = b'_i + ?_i \wedge$

$?_i \geq 0$).

$$\textsc{Templ}(rel)(v, v', v'') = (?_1 + \cdots + ?_5 = 1 \wedge \bigwedge_{i \in \{1..5\}} (b_i'' = b_i' + ?_i \wedge ?_i \geq 0)$$

Our approach computes the auxiliary assertions that are required by the reachability proof rule and a witness for the existential quantifier. The witness instantiates the template parameters and represents the Stepmother's strategy to ensure that the buckets eventually overflow no matter what moves are made by the Cinderella player.

$$rel(v, v') = (b_1' = b_1 + 0.8 \wedge b_2' = b_2 \wedge b_3' = b_3 + 0.1 \wedge b_4' = b_4 \wedge b_5' = b_5 + 0.1)$$

In this case, since the addition of water is done in non-adjacent buckets, e.g., b_1 and b_3, eventually the game reaches an overflow state, and the Stepmother is the player to win this game.

6.4.3 Games with Cinderella's LTL objectives

Apart from games with safety and reachability objectives, our approach is able to handle games with more general LTL objectives. For this game, we use the following player roles.

$$\begin{aligned} init(v) &= (b_1 = 0 \wedge \cdots \wedge b_5 = 0) \\ eve(v, v') &= cinderella(v, v') \\ adam(v, v') &= stepmother(v, v') \end{aligned}$$

We use the value $c = 1.4$ for the bucket capacity, similar to Section 6.4.2. As already explained, with this value Stepmother has a strategy to win the game with the objective $\varphi(v) = F\, overflow(v)$. Consequently, Cinderella does not have a strategy to win the game with the objective set to the complement formula, i.e., $\neg\varphi(v) = G\, \neg overflow(v)$. For this section, we formalize a winning condition that is a weaker logical formula than $\neg\varphi(v)$ for which Cinderella has a winning strategy. The objective constraint $GF\, \neg overflow(v)$ states that an overflow state does not occur infinitely often in the plays of the game.

More generally, we use *color* to indicate the most significant bucket for which an overflow occurs.

- A state without overflow: $(color = 0)$.
- A state with overflow such that i is the smallest index of those that correspond to buckets that have overflown: $(color = i)$.

We group the states of the system based on the truth value of the predicates $color = i$ as follows.

$$p_0(v) = (color = 0) = (b_1 \leq 1.4 \wedge \cdots \wedge b_5 \leq 1.4)$$
$$p_1(v) = (color = 1) = (b_1 > 1.4)$$
$$p_2(v) = (color = 2) = (b_1 \leq 1.4 \wedge b_2 > 1.4)$$
$$p_3(v) = (color = 3 \vee color = 4 \vee color = 5) = \ldots$$

A winning condition corresponding to a value i ensures that states from $p_i(v)$ occur infinitely often in the plays of the system, and that i is the smallest value for which states occur infinitely often.

$$win(v, i) = (GF\ p_i(v) \wedge \bigwedge_{j \in \{0,..,i-1\}} FG\ \neg p_j(v))$$

Our approach for solving games with LTL objectives proceeds in three steps: 1) complement the LTL formula φ representing the winning condition; 2) construct a Büchi automaton corresponding to the complemented formula $\neg\varphi$; 3) instantiate the proof rule from Figure 6.3 using the Büchi automaton representation.

LTL game 1 For the first LTL game, we define the objective for the Cinderella player $obj(v) = win(v, 0) = GF\ p_0(v)$. We complement the objective formula to obtain $FG\ \neg p_0(v) = FG\ overflow(v)$, then construct the Büchi automaton corresponding to the complemented formula as follows.

$$\begin{aligned} init_{\mathcal{B}}(pc_{\mathcal{B}}) &= (pc_{\mathcal{B}} = 0) \\ next_{\mathcal{B}}(pc_{\mathcal{B}}, v, pc'_{\mathcal{B}}) &= (pc_{\mathcal{B}} = 0 \wedge pc'_{\mathcal{B}} = 0 \vee \\ &\qquad pc_{\mathcal{B}} = 0 \wedge pc'_{\mathcal{B}} = 1 \wedge overflow(v) \vee \\ &\qquad pc_{\mathcal{B}} = 1 \wedge pc'_{\mathcal{B}} = 1 \wedge overflow(v)) \\ acc_{\mathcal{B}}(pc_{\mathcal{B}}) &= (pc_{\mathcal{B}} = 1) \end{aligned}$$

We instantiate the proof rule from Figure 6.3 as follows.

$$(init(v), adam(v, v'), eve(v, v')) \models obj(v)$$

There exists a strategy for Eve provided that the premises of the proof rule are satisfied. These premises are Horn clauses over the auxiliary assertion $inv(w), aux(w, w', v''), round(w, w', w'')$, and $fair(w, w')$. We apply E-HSF to find a solution for the auxiliary assertions. The clause B2 is an existentially quantified clause.

By skolemization of the existential clause B2 we obtain the following.

$$\begin{aligned} inv(w) \ &\wedge stepmother(v, v') \wedge next_{\mathcal{B}}(pc_{\mathcal{B}}, v, pc'_{\mathcal{B}}) \\ &\wedge rel(w, w', v'') \rightarrow cinderella(v', v'') \wedge aux(w, w', v'') \end{aligned}$$

Using the template described in the paragraph "Second strategy" from Section 5.1, our approach is able to derive solutions for the auxiliary assertions and the following template parameters $?_1 = ?_2 = ?_3 = ?_4 = ?_5 = 1$. We conclude that Cinderella is the player to win this game, and that her strategy ensures that states without overflow occur infinitely often in the plays of the game.

LTL game 2 For the second LTL game, we define the objective for the Cinderella player $win(v, 0) \vee win(v, 2)$. The objective for the Stepmother player is $win(v, 1) \vee win(v, 3)$. The formula corresponding to the Cinderella's objective:

$$\varphi = (GF\ p_0(v) \vee (GF\ p_2(v) \wedge FG\ \neg p_1(v) \wedge FG\ \neg p_0(v))).$$

The complemented formula is

$$\neg\varphi = (FG\ \neg p_0(v) \wedge (FG\ \neg p_2(v) \vee GF\ p_1(v) \vee GF\ p_0(v))).$$

The Büchi automaton corresponding to the complemented formula contains 10 distinct control states, from which two are accepting states. Using our proof rule, we are able to compute automatically auxiliary assertions and obtain that the same second strategy is winning for the Cinderella player.

Note that for the player Cinderella, the LTL game 2 (with objective $win(v, 0) \vee win(v, 2)$) is easier to win than the LTL game 1 (with objective $win(v, 0)$). However, the relation between the two objectives is not immediately usable in a deductive approach like ours. We presented both LTL games 1 and 2, since our approach based on the proof rule from Figure 6.3 constructs different Buechi automata and different auxiliary assertions for the two objectives.

6.5 Case study: program repair/synthesis games

In this section we illustrate how our constraint-based approach to solving games applies to the synthesis of reactive programs from temporal specifications. We consider synthesis problems obtained from program repair questions, see Section 6.5.1 and Section 6.5.2,

as well as inference of thread synchronization, see Section 6.5.3. In Section 6.6 we report on running times required for solving these games by applying E-HSF.

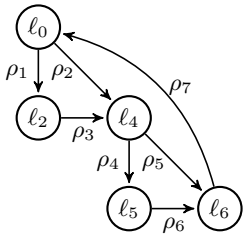
6.5.1 Program repair game with safety objective

We model program repair as a game following [78]. That is, given a set of suspect statements, we look for a modification of those program statements such that the modified program satisfies its specification. For the first repair game, we assume that a program is given by a tuple $(init(v), next(v, v'), error(v))$ that represents initial states, a transition relation, and error states, respectively.

As an example we consider the program shown in Figure 3 in [78]. The program has three program variables $v = (l, gl, pc)$. The variable l models a lock, the variable gl is used to keep track of the status of the lock, while the variable pc is the program counter variable. The initial states of the program are

$$init(v) = (gl = 0 \wedge l = 0 \wedge pc = \ell_0).$$

We show the control-flow graph of the program below.



The transition relation is defined as follows.

$$\begin{aligned}
\rho_1(v, v') &= (pc = \ell_0 \wedge pc' = \ell_2 \wedge l' = l \wedge gl' = gl) \\
\rho_2(v, v') &= (pc = \ell_0 \wedge pc' = \ell_4 \wedge l' = l \wedge gl' = gl) \\
\rho_3(v, v') &= (pc = \ell_2 \wedge pc' = \ell_4 \wedge l \leq 0 \wedge l' = 1) \\
\rho_4(v, v') &= (pc = \ell_4 \wedge pc' = \ell_5 \wedge gl \neq 0 \wedge l' = l \wedge gl' = gl) \\
\rho_5(v, v') &= (pc = \ell_4 \wedge pc' = \ell_6 \wedge gl = 0 \wedge l' = l \wedge gl' = gl) \\
\rho_6(v, v') &= (pc = \ell_5 \wedge pc' = \ell_6 \wedge l \geq 1 \wedge l' = 0 \wedge gl' = gl) \\
\rho_7(v, v') &= (pc = \ell_6 \wedge pc' = \ell_0 \wedge l' = l) \\
next(v, v') &= (\rho_1(v, v') \vee \cdots \vee \rho_7(v, v'))
\end{aligned}$$

Note that in $\rho_3(v, v')$ and $\rho_7(v, v')$ the variable gl is assigned a non-deterministic value, since it is not constrained by the corresponding assertions. The execution of the program

enters an error state at location ℓ_2 if the lock variable l is held, and at location ℓ_5 if the lock variable l is not held, i.e., we have

$$error(v) = (pc = \ell_2 \wedge l = 1 \vee pc = \ell_5 \wedge l = 0).$$

We instantiate the safety game proof rule such that the system role is played by the program transition relation and the environment role is to provide inputs to the program (in this case, the program does not expect any inputs).

$$\begin{aligned} eve(v, v') &= next(v, v') \\ adam(v, v') &= skip(v, v') \\ obj(v) &= G\ \neg error(v) \end{aligned}$$

A repair of the program restricts the transition relation of the program such that $G\ \neg error(v)$ holds. To this end, we provide a template corresponding to the existentially quantified clause of the proof rule:

$$\begin{aligned} \textsc{Templ}(rel)(v, v', v'') = (pc' = \ell_2 \wedge pc'' = \ell_4 \wedge gl'' = ?_1 \\ \vee\ pc' = \ell_6 \wedge pc'' = \ell_0 \wedge gl'' = ?_2) \end{aligned}$$

Our algorithm returns the witness for the existential quantifier clause that instantiates the template parameters $?_1 = 1$ and $?_2 = 0$. This corresponds to a repaired program that assigns the value 1 to gl at location ℓ_2, and assigns the value 0 to gl at location ℓ_6. We obtain the same program repair as the solution originally presented in [78].

6.5.2 Concurrent program repair games with safety and response objectives

We illustrate how our approach can be applied to concurrent program repair problems, and in particular to repair problems under fairness assumptions. We use the Critical Section example from Figure 5 in [78] for this purpose. In this example, the assignment $turn1B = false$ at location ℓ_2 is faulty. The goal is to repair this assignment, and hence, the entire program by checking if there exists an assignment to the variable $turn1B$ from its domain $\{true,\ false\}$ such that the resulting program satisfies certain temporal properties. These properties are used in directing the repair process towards the correct version of the program.

Let $(f1a, f1b, t1b, f2a, f2b, t2b)$ be abbreviations of original variable names ($flag1A$, $flag1B$, $turn1B$, $flag2A$, $flag2B$, $turn2B$). We encode the original program over variables $v = (pc_1, pc_2, x, y, f1a, f1b, t1b, f2a, f2b, t2b)$ using an initial condition $init(v)$ such that

$$\begin{aligned} init(v) = (&f1a = 0 \wedge f1b = 0 \wedge t1b = 0 \wedge f2a = 0 \\ &\wedge f2b = 0 \wedge t2b = 0 \wedge pc_1 = \ell_1 \wedge pc_2 = \ell_1), \end{aligned}$$

and a transition relation $next(v, v')$. Since the program is multi-threaded with two threads, we give $next(v, v')$ as a disjunction of transition relations of individual threads

$$next_1(v, v') \wedge pc_2' = pc_2 \vee next_2(v, v') \wedge pc_1' = pc_1.$$

For the first thread we define (note that we explicate assignments of a non-deterministic value to a variable z by $z' = ND$ and we omit equalities for variables that do not change, hence, each variable z that does not appear in $z' = \ldots$ is constrained by $z' = z$):

$$\begin{aligned} next_1(v, v') = (&pc_1 = \ell_1 \wedge pc_1' = \ell_2 \wedge f1a' = 1 \vee \\ &pc_1 = \ell_2 \wedge pc_1' = \ell_3 \wedge t1b' = ND \vee \\ &pc_1 = \ell_3 \wedge f1b = 1 \wedge t1b = 1 \wedge pc_1' = \ell_3 \vee \\ &pc_1 = \ell_3 \wedge (f1b = 0 \vee t1b = 0) \wedge pc_1' = \ell_4 \vee \\ &pc_1 = \ell_4 \wedge pc_1' = \ell_5 \wedge f1a' = 0 \vee \\ &pc_1 = \ell_5 \wedge t1b = 1 \wedge pc_1' = \ell_6 \wedge f2a' = 1 \vee \\ &pc_1 = \ell_5 \wedge t1b = 1 \wedge pc_1' = \ell_9 \vee \\ &pc_1 = \ell_6 \wedge pc_1' = \ell_7 \wedge t2b' = 1 \vee \\ &pc_1 = \ell_7 \wedge f2b = 1 \wedge t2b = 1 \wedge pc_1' = \ell_7 \vee \\ &pc_1 = \ell_7 \wedge (f2b = 0 \vee t2b = 0) \wedge pc_1' = \ell_8 \vee \\ &pc_1 = \ell_8 \wedge pc_1' = \ell_9 \wedge f2a' = 0 \vee \\ &pc_1 = \ell_9 \wedge pc_1' = \ell_1). \end{aligned}$$

Note that the second disjunct above leaves the value of $t1b$ unrestricted, as denoted by $t1b' = ND$. For the second thread we define:

$$\begin{aligned} next_2(v, v') = (&pc_2 = \ell_1 \wedge pc_2' = \ell_2 \wedge f1b' = 1 \vee \\ &pc_2 = \ell_2 \wedge pc_2' = \ell_3 \wedge t1b' = 0 \vee \\ &pc_2 = \ell_3 \wedge f1a = 1 \wedge t1b = 0 \wedge pc_2' = \ell_3 \vee \\ &pc_2 = \ell_3 \wedge (f1a = 0 \vee t1b = 1) \wedge pc_2' = \ell_4 \vee \\ &pc_2 = \ell_4 \wedge pc_2' = \ell_5 \wedge f2b' = 1 \vee \\ &pc_2 = \ell_5 \wedge pc_2' = \ell_6 \wedge t2b' = 0 \vee \\ &pc_2 = \ell_6 \wedge f2a = 1 \wedge t2b = 0 \wedge pc_2' = \ell_6 \vee \\ &pc_2 = \ell_6 \wedge (f2a = 0 \vee t2b = 1) \wedge pc_2' = \ell_7 \vee \\ &pc_2 = \ell_7 \wedge pc_2' = \ell_8 \wedge f2b' = 0 \vee \\ &pc_2 = \ell_8 \wedge pc_2' = \ell_9 \wedge f1b' = 0 \vee \\ &pc_2 = \ell_9 \wedge pc_2' = \ell_1). \end{aligned}$$

In the original problem there are two properties directed by which the program should be repaired. The first property requires that the two threads do not enter their critical sections at the same time. This property is specified with the following LTL formula.

$$\begin{aligned} \varphi_1(v) = G\ ((&pc_1 = \ell_4 \rightarrow pc_2 \neq \ell_4) \wedge \\ &(pc_1 = \ell_8 \rightarrow pc_2 \neq \ell_7)) \end{aligned}$$

The second property requires that neither of the threads can be in a deadlock state. This property is specified as follows.

$$\begin{aligned} \varphi_2(v) = G\ ((&pc_1 = \ell_3 \rightarrow F\ (pc_1 = \ell_4)) \wedge \\ &(pc_1 = \ell_7 \rightarrow F\ (pc_1 = \ell_8)) \wedge \\ &(pc_2 = \ell_3 \rightarrow F\ (pc_2 = \ell_4)) \wedge \\ &(pc_2 = \ell_6 \rightarrow F\ (pc_2 = \ell_7))) \end{aligned}$$

Safety game Doing the program repair using the first property amounts to applying the safety proof rule from Figure 7.2 to find $inv(v)$. Since there is no interaction with the environment, $adam(v, v')$ will simply be equivalent with $skip(v, v')$. To apply our proof rule, we use $next(v, v')$ for $eve(v, v')$ and $\varphi_1(v)$ as the winning condition obj. We use the following template for the existential clause:

$$\textsc{Templ}(rel)(v, v', v'') = (t1b'' =?_1).$$

E-HSF computes the solution $?_1 = 1$, and correspondingly we obtain a modified version of $next_1(v, v')$ where the non-deterministic assignment $t1b' = ND$ from the second disjunct is replaced by $t1b' = 1$.

Fair LTL game The second property relies on fairness assumptions. To deal with the fairness, we apply a transformation technique from [97] that reduces fair parallelism semantics to the usual parallelism semantics. The idea is to use the equivalence $P \models_{fair} \Phi$ if and only $T_{fair}(P) \models \Phi$, where P is the original program, T_{fair} is the fair transformation function, $T_{fair}(P)$ is the transformed program with embedded tracking of fairness, and Φ is the property to check. The transformation does not change the initial states of the program, but it significantly modifies the semantics of the transition relation of the program. A counter variable is introduced for each thread from the program, and the first statement of each loop is strengthened by adding a guard and an update involving the counter variables. See [97] for details. For our example program, the transformation:

- introduces the counters k_1 and k_2,
- adds the guard $k_1 \leq k_2$ and the update constraint $(k_1' = ND \wedge k_2' = k_2 - 1)$ to the first, third and ninth disjunct from $next_1(v, v')$,
- adds the guard $k_2 \leq k_1$ and the update constraint $(k_1' = k_1 - 1 \wedge k_2' = ND)$ to the first, third and sixth disjunct in $next_2(v, v')$.

Let $initT(v, k_1, k_2) = init(v)$ be the initial condition of the transformed program. We refer to the transformed transition relations as $nextT_1(v, k_1, k_2, v', k_1', k_2')$ and $nextT_2(v, k_1, k_2, v', k_1', k_2')$, and present them below.

$$
\begin{aligned}
&nextT_1(v, k_1, k_2, v', k_1', k_2') = \\
&\quad (pc_1 = \ell_1 \wedge pc_1' = \ell_2 \wedge f1a' = 1 \\
&\quad \wedge k_1 \leq k_2 \wedge k_1' = ND \wedge k_2' = k_2 - 1 \vee \\
&\quad \ldots \vee \\
&\quad pc_1 = \ell_3 \wedge f1b = 1 \wedge t1b = 1 \wedge pc_1' = \ell_3 \\
&\quad \wedge k_1 \leq k_2 \wedge k_1' = ND \wedge k_2' = k_2 - 1 \vee \\
&\quad \ldots \vee \\
&\quad pc_1 = \ell_7 \wedge f2b = 1 \wedge t2b = 1 \wedge pc_1' = \ell_7 \\
&\quad \wedge k_1 \leq k_2 \wedge k_1' = ND \wedge k_2' = k_2 - 1 \vee \\
&\quad \ldots)
\end{aligned}
$$

$$
\begin{aligned}
&nextT_2(v, k_1, k_2, v', k_1', k_2') = \\
&\quad (pc_2 = \ell_1 \wedge pc_2' = \ell_2 \wedge f1b' = 1 \\
&\quad \wedge\, k_2 \leq k_1 \wedge k_1' = k_1 - 1 \wedge k_2' = ND \vee \\
&\quad \ldots \vee \\
&\quad pc_2 = \ell_3 \wedge f1a = 1 \wedge t1b = 0 \wedge pc_2' = \ell_3 \\
&\quad \wedge\, k_2 \leq k_1 \wedge k_1' = k_1 - 1 \wedge k_2' = ND \vee \\
&\quad \ldots \vee \\
&\quad pc_2 = \ell_6 \wedge f2a = 1 \wedge t2b = 0 \wedge pc_2' = \ell_6 \\
&\quad \wedge\, k_2 \leq k_1 \wedge k_1' = k_1 - 1 \wedge k_2' = ND \vee \\
&\quad \ldots)
\end{aligned}
$$

The second property is more complicated than the first property since it involves nesting of temporal operators. Like the case for the first property, we assume $adam(v, v')$ to be equivalent with $skip(v, v')$. We make $nextT(v, v')$ to play the role of $eve(v, v')$, and $G\ ((pc_1 = \ell_3 \rightarrow F\ pc_1 = \ell_4) \wedge (pc_1 = \ell_7 \rightarrow F\ pc_1 = \ell_8) \wedge (pc_2 = \ell_3 \rightarrow F\ pc_2 = \ell_4) \wedge (pc_2 = \ell_6 \rightarrow F\ pc_2 = \ell_7))$ is now a winning condition obj.

We reuse the template used for the previous game and we get exactly the same solution. That is, we determine $t1b' = ND$ to $t1b' = 1$ in the second disjunct of $nextT_1(v, k_1, k_2, v', k_1', k_2')$.

6.5.3 Synthesis of synchronization game with safety objective

Synthesis of synchronization in multi-threaded programs [118] can be automated using our approach. For illustration, we use the example program from Figure 1 in [118] and represent it using a tuple $(init(v), next(v, v'), error(v))$ for the case when three threads are involved in computation. The program variables are $v = (x, y_1, y_2, z, pc_1, pc_2, pc_3)$, the initial states are described by $init(v) = (x = 0 \wedge z = 0 \wedge pc_1 = \ell_1 \wedge pc_2 = \ell_1 \wedge pc_3 = \ell_1)$. The transition relation of the program is

$$
\begin{aligned}
next(v, v') = (&next_1(v, v') \wedge pc_2' = pc_2 \wedge pc_3' = pc_3 \vee \\
&next_2(v, v') \wedge pc_1' = pc_1 \wedge pc_3' = pc_3 \vee \\
&next_3(v, v') \wedge pc_1' = pc_1 \wedge pc_2' = pc_2)
\end{aligned}
$$

such that

$$
\begin{aligned}
&next_1(v, v') = \\
&\quad (pc_1 = \ell_1 \wedge pc_1' = \ell_2 \wedge x' = x + z \wedge skip(y_1, y_2, z) \vee \\
&\quad\ pc_1 = \ell_2 \wedge pc_1' = \ell_3 \wedge x' = x + z \wedge skip(y_1, y_2, z)) \\
&next_2(v, v') = \\
&\quad (pc_2 = \ell_1 \wedge pc_2' = \ell_2 \wedge z' = z + 1 \wedge skip(x, y_1, y_2) \vee \\
&\quad\ pc_2 = \ell_2 \wedge pc_2' = \ell_3 \wedge z' = z + 1 \wedge skip(x, y_1, y_2))
\end{aligned}
$$

$$
\begin{aligned}
&next_3(v, v') = \\
&\quad (pc_3 = \ell_1 \wedge pc_3' = \ell_2 \wedge x = 1 \wedge y_1' = 3 \wedge skip(x, y_2, z) \vee \\
&\quad\ pc_3 = \ell_1 \wedge pc_3' = \ell_2 \wedge x = 2 \wedge y_1' = 6 \wedge skip(x, y_2, z) \vee \\
&\quad\ pc_3 = \ell_1 \wedge pc_3' = \ell_2 \wedge (x \leq 0 \vee x \geq 3) \wedge y_1' = 5 \wedge \\
&\quad\ skip(x, y_2, z) \vee \\
&\quad\ pc_3 = \ell_2 \wedge pc_3' = \ell_3 \wedge y_2' = x \wedge skip(x, y_1, z) \vee \\
&\quad\ pc_3 = \ell_3 \wedge pc_3' = \ell_4 \wedge y_1 \neq y_2 \wedge skip(x, y_1, y_2, z)).
\end{aligned}
$$

Different interleavings of the three threads lead to different values of y_1 and y_2. An assertion in the third thread at location ℓ_3 requires that the values given to y_1 and y_2 are not equal, i.e., we have

$$error(v) = (pc_3 = \ell_3 \wedge y_1 = y_2).$$

For the given program, some interleavings lead to the values of y_1 and y_2 being equal, while other interleavings lead to distinct values for the two variables. The goal of [118] is to add synchronization to the program such that the assertion holds on all executions. To apply our proof rule to this problem, we encode the choice between executing a single step and executing an atomic section using auxiliary variables. The transition relation of the program is augmented with guards deciding a single step or an atomic section based on the values of the auxiliary variables. For our example, we use four auxiliary variables $(c_{11}, c_{21}, c_{31}, c_{32})$. We obtain an extended tuple of variables $w = (v, c_{11}, c_{21}, c_{31}, c_{32})$. A constraint $c_{ij} = \ell$ is used in thread i to decide that the control flows from location ℓ_j to location ℓ. Correspondingly, the transition relation of the first thread is augmented to:

$$
\begin{aligned}
&next_1(w, w') = \\
&(pc_1 = \ell_1 \wedge c_{11} = \ell_2 \wedge pc_1' = \ell_2 \wedge x' = x + z \wedge skip(\dots) \vee \\
&\ pc_1 = \ell_1 \wedge c_{11} = \ell_3 \wedge pc_1' = \ell_3 \wedge x' = x + 2 * z \wedge skip(\dots) \vee \\
&\ pc_1 = \ell_2 \wedge pc_1' = \ell_3 \wedge x' = x + z \wedge skip(\dots))
\end{aligned}
$$

The transition relations of the second and third thread are instrumented similarly. We instantiate the safety game proof rule such that the system role is played by the instrumented program transition relation and the environment role is to provide inputs to the program (similar to the previous case, this example program does not expect any inputs).

$$\begin{aligned} init(w) &= init(v) \\ eve(w,w') &= (next_1(w,w') \vee next_2(w,w') \vee next_3(w,w')) \\ adam(w,w') &= skip(w,w') \end{aligned}$$

Furthermore, we represent the search of initial parameter values using a strengthening of the original initial condition with an assertion $mid(w,t)$ such that:

$$\begin{aligned} &init(w) \rightarrow \exists t : mid(w,t) \\ &(mid(w,t), eve(w,w'), adam(w,w')) \models G\ \neg error(w) \end{aligned}$$

We use E-HSF and provide the following template for the existential clause involving the initial states.

$$\begin{aligned} \textsc{Templ}(rel)(w,t) = (c_{11} =?_{11} \wedge c_{21} =?_{21} \\ \wedge\ c_{31} =?_{31} \wedge c_{32} =?_{32}). \end{aligned}$$

A solution to $mid(w,t)$ sets the auxiliary variables to target program locations so that the objective of the game is satisfied, i.e., the error states are not reachable. E-HSF returns the following witness for the existential quantifier clause:

$$rel(w,t) = (c_{11} = 3 \wedge c_{21} = 3 \wedge c_{31} = 4 \wedge c_{32} = 3).$$

We note that our proof rule does not represent an optimization problem. The solutions we obtain correspond to a synthesized program that is not necessarily the most efficient one, i.e., the longest atomic sections may be picked instead of smaller steps. Dealing with optimality is a subject for future work.

6.6 Evaluation

In this section we describe how we used E-HSF as a proof-of-concept engine of our proposed quantified Horn constraints based approach to solving infinite-state games. E-HSF uses two SMT solvers for handling non-linear constraints: the Z3 solver [47] and the Barcelogic solver [20, 22]. For our experiments we used a computer with an Intel Core 2 Duo 2.53 GHz CPU and 4 GB of RAM. Table 7.1 shows the results corresponding to the case studies described in the previous sections: Cinderella-Stepmother games with

safety objectives (G1 and G2), with reachability objectives (G3) and with more general LTL objectives (G4 and G5). Lastly, we show results on the program repair/synthesis games (G6, G7, G8 and G9). A T/O-mark stands for time out after 15 minutes.

Name	Game	Player	Objective	Result	z3	Barcelogic
G1	Cinderella ($c = 3$)	Cinderella	$G\ \neg overflow$	✓	3.2s	**1.2s**
G2	Cinderella ($c = 2$)	Cinderella	$G\ \neg overflow$	✓	**1m52s**	**1m52s**
G3	Cinderella ($c = 1.4$)	Stepmother	$F\ overflow$	✓	**18s**	1m14s
G4	Cinderella ($c = 1.4$)	Cinderella	$win(0)$	✓	**7m16s**	SysError
G5	Cinderella ($c = 1.4$)	Cinderella	$win(0) \vee win(2)$	✓	**4.7s**	**4.7s**
G6	Repair-Lock	Program	$G\ \neg error$	✓	**0.3s**	**0.3s**
G7	Repair-Critical	Program	$G\ \neg error$	✓	17.7s	**16.9s**
G8	Repair-Critical	Program	$G\ (at\_p \rightarrow F\ \neg at\_p)$	✓	**53.3s**	3m6s
G9	Synch-Synth	Program	$G\ \neg error$	✓	T/O	**1s**

TABLE 6.1: Statistics for the case studies

For each game we report the player and the objective for which we synthesize strategies (see Columns 3 and 4). Column 5 shows the result obtained from our tool: an ✓-mark stands for a strategy successfully synthesized by our tool using either Z3 or Barcelogic as solving back-ends. In one case (G5), due to the general LTL objective we obtain a large Büchi automaton. Our normal encoding times-out for both Z3 and Barcelogic. However, our method is able to synthesize a winning strategy quickly when we exploit the explicit evaluation optimization(from Section 3.4.1) where we treat infinite datatypes symbolically using a decision procedure, and finite domain datatypes, like the program counter variable of the Büchi automaton, explicitly without abstraction. Because the control locations of the Büchi automaton $pc_{\mathcal{B}}$ range over a finite domain, this optimization allows the tool to track the states of $pc_{\mathcal{B}}$ explicitly, and this simplifies the proof obligations.

We believe our approach will benefit from future improvements in constraint solving. The cases when either Z3 or Barcelogic times out are challenging SMT problems and of potential interest to the SMT-solving community.

6.7 Related work

There is a rich literature on decision procedures for graph games with application to formal methods [50, 83, 103, 116]. In particular, many techniques, both explicit-state [116] and symbolic [68, 101], are known for games on finite graphs. Decidability results are also known for games on certain restricted classes of infinite graphs, such as pushdown graphs [27, 120] and prefix-recognizable graphs [28].

Known approaches to games on graphs that represent state spaces of general infinite-state programs can be divided into two categories: those based on symbolic execution and those based on abstraction-refinement. De Alfaro et al [46] offer an example of the first kind of approach. In this work, a symbolic semi-algorithm is used to explore the state space of the game directly. In contrast, we reduce the problem of solving a game to Horn constraint solving, leaving the constraints to be solved by a dedicated solver relying on CEGAR and interpolation.

The second category of methods [7, 53, 54, 61, 62, 65] lift predicate abstraction and CEGAR, originally proposed for safety verification, to games. The core idea here is the use of abstract transition systems where overapproximate ("may") and underapproximate ("must") transitions are permitted, and which are model-checked against properties with 3-valued semantics. In contrast to existing approaches of this sort, we do not directly construct a program abstraction with must-transitions or 3-valued semantics. Instead, we use a Skolem relation that is iteratively refined. Moreover, our backend solver uses disjunctively well-founded transition invariants [105] to resolve liveness goals for players, which (so far as we are aware) existing approaches do not do. This algorithmic difference has a significant impact in practice.

Our approach is perhaps more closely related to a recent paper by Cook and Koskinen [39], which uses a combination of CEGAR with a form of Skolemization for verification of branching-time properties of programs. However, this method only studies verification of CTL — the class of properties that we handle is significantly larger (e.g., it includes the full μ-calculus).

Games have a particularly close connection to program synthesis and repair, areas that have seen a flurry of activity in the last few years. However, in recent as well as classical algorithms in these areas, the focus is tends to be either on finite-state systems [78, 103, 111], or on functional, rather than reactive, programs [82, 115, 118]. In contrast, the natural application of our approach is in the repair and synthesis of infinite-state reactive programs.

Finally, our work here was inspired by the rich tradition of on deductive program synthesis [90, 103, 110]. The main difference between this line of work and ours lies in our focus on automation. For example, Slanina [110] also offers a deductive rule for games with response objectives. However, the proof rule demands global ranking functions and justice and compassion assumptions, which are known to be difficult to discharge automatically.

6.8 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 [61, 65] and automata-theoretic approaches to games [116] remains a fascinating open question.

Chapter 7

Program synthesis via Solving Recursive Games as Horn Constraint Solving

7.1 Introduction

The last decade has seen remarkable advances in automated software verification [8, 122]. 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 [8] and SATURN [122] use *procedure summarization* [108] to compositionally analyse 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 [3, 42]. Less is known about the use of summarization in the emerging setting of *automated program synthesis* [12, 82, 111, 115]. 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* [57]. 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 [25, 103, 116]. 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 Chapter 6 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 chapter, we present such an approach. The key idea here is a generalisation 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 generalisation 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 [4–6], 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 Eve 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 [105].

We show that our rules are sound, meaning that if they derive a strategy, then Eve 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 of a sound approximation to our rules on top of the E-HSF automated deduction engine from Chapter 3. Specifically, our implementation feeds our proof rules to the E-HSF engine for solving constraints in the form of forall-exists quantified Horn clauses that permit existential quantification in clause heads. Solving the repair problem now amounts to finding an interpretation to unknown sets and relations over program variables. E-HSF 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 our method on an array of systems programs, including device driver benchmarks drawn from the SV-COMP software verification competition [15]. Some of our benchmarks contain up to 11K 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 all cases, our method is able to return successfully within a few seconds, 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 summarise the main contributions of the chapter:

- 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 of a sound but incomplete approximation of our inference rules on top of the E-HSF deduction engine. We illustrate the promise of the system using an array of challenging benchmarks running into thousands of lines of code.

This chapter is organised as follows. In Section 7.2, we present a motivating example that illustrates our approach. Section 7.3 formally defines our synthesis problem. Section 7.4 introduces game summaries. Section 7.5 gives our sound and complete proof rules for compositional synthesis. Section 7.6 presents our implementation of the rules and experimental results. Related work is presented in Section 7.7; we conclude with some discussion in Section 7.8.

7.2 Motivation

Our program synthesis problem can be viewed as a game [57] 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 Figure 7.1 that describes an interaction between an environment player that controls all statement except at line `PL` and the program player that only controls the non-deterministic assignment statement at line `PL`. The goal of

```
   void main(void) {
E:   if (env_nondet()) {
A:     foo();
B:     x = 0;
     } else {
C:     foo();
D:     x = 1;
     }
F:   assert( x == y );
   }
```

```
  int x=-1, y=-1;
   int foo() {
P:   if (prog_nondet()) {
Q:     y = 0;
     } else {
R:     y = 1;
     }
S: }
```

FIGURE 7.1: Program that exhibits inadequacy of procedure summaries.

the program player is to find a strategy that resolves the non-determinism at line `PL` such that regardless of how the environment player resolves the non-determinism at line `EL` 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`.

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

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 `B` for the program counter of `main`. That is, When applying $sum((x, y, pc), (x', y', pc))$ in the calling context with $pc = \texttt{A}$ we obtain a state in which $y = 0 \vee y = 1$. Hence the subsequent assignment `x = 1;` leads to an assertion violation.

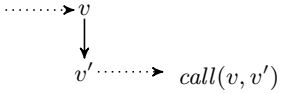
In contrast, when applying the notion of game summaries we relate each entry state of `foo` with states of `main` at the return sites `A` and `B`. Thus, the game summary can discriminate between the call site on the branch that executes `x = 0;` and the call site on the branch with `x = 1;`. As a result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

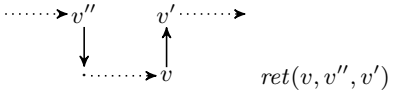
7.3 Preliminaries

In this section, we formally define procedural programs, recursive games, and the synthesis problem we are interested in.

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 evaluating. 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 unaffected caller's local 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, i.e., g'' does not occur in the assertion $ret(v, v'', v')$. 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 Eve and Adam (standing respectively for the program and the environment) take turns in performing computation steps[1]. In this paper, we assume that Adam 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, $eve(v, v')$ and $adam(v, v')$, respectively belonging to Eve and Adam. Among the intra-procedural steps we assume a strict alternation between Eve and Adam. That is, when considering

[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.

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 states. A stack st is a finite sequence of states, i.e., $st \in S^*$. We use "$\cdot$" 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 $\mathit{init}(s)$ is called an initial configuration. A configuration that is in the domain of the program transition relation eve 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.

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

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 $\mathit{next}((s, st), (s', st'))$.

A play π is a sequence of configurations that labels a branch of a computation tree. We write π_i and s_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 $\mathit{safe}(s)$.

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

7.4 Game summaries

Our proof 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. Given a play π, we define a reachability relation $\rightsquigarrow_\pi$ 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.

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

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.

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 2. *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 $adam(s_k, s_{k+1})$ 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 $eve(s_k, s_{k+1})$ we first consider that $adam(s_{k-1}, s_k)$ 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 $sum(s_{k-1}, R_{k-1}) \wedge adam(s_{k-1}, s_k)$. 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. □

7.5 Proof rules

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 Eve has a winning strategy by solving implication and well-foundedness constraints on auxiliary assertions over system variables. The rules are sound as well as relatively complete.

7.5.1 Safe strategies

We consider a synthesis problem where Eve has a winning strategy if only states from $safe(v)$ are visited by all plays. We present the corresponding proof rule in Figure 7.2.

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, $sum(s_0, \emptyset)$. 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 $sum(s_1, R_1)$. We require that for every state s_2 satisfying $adam(s_1, s_2)$ 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 $eve(s_2, s_3)$. We also require that there exists a set of states R_3 such that $sum(s_3, R_3)$ and $R_2 \supseteq R_3$.

Assume a state s_1 and its corresponding set of states R_1 such that $sum(s_1, R_1)$ is given. 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 $sum(v, \mathcal{R})$ 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. For a return step (s_1, s_2) such that $sum(s_1, R_1)$, we require that s_2 is in R_1.

Since the winning condition requires all states to satisfy $safe(v)$, each states s such that $sum(s, R)$ need 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 Figure 7.1.

Find *sum* such that:

$$
\begin{array}{lll}
\text{S1:} & \mathit{init}(v) & \rightarrow \mathit{sum}(v, \emptyset) \\
\text{S2:} & \mathit{sum}(v_1, R_1) \wedge \mathit{adam}(v_1, v_2) & \rightarrow \exists R_2 : \mathit{sum}(v_2, R_2) \wedge R_1 \supseteq R_2 \wedge \\
 & & \quad \exists v_3 : \mathit{eve}(v_2, v_3) \wedge \\
 & & \quad\quad \exists R_3 : \mathit{sum}(v_3, R_3) \wedge R_2 \supseteq R_3 \\
\text{S3:} & \mathit{sum}(v_1, R_1) \wedge \mathit{call}(v_1, v_2) & \rightarrow \exists R_2 : \mathit{sum}(v_2, R_2) \\
\text{S4:} & \mathit{sum}(v_1, R_1) \wedge \mathit{call}(v_1, v_2) \wedge & \\
 & \quad \mathit{sum}(v_2, R_2) \wedge R_2(v_3) & \rightarrow \exists R_3 : \mathit{sum}(v_3, R_3) \wedge R_1 \supseteq R_3 \\
\text{S5:} & \mathit{sum}(v_1, R) \wedge \mathit{ret}(v_1, v_2) & \rightarrow R(v_2) \\
\text{S6:} & \mathit{sum}(v, R) & \rightarrow \mathit{safe}(v)
\end{array}
$$

$$(\mathit{init}(v), \mathit{eve}(v, v'), \mathit{adam}(v, v'), \mathit{call}(v, v'), \mathit{ret}(v, v', v'')) \models G\ \mathit{safe}(v)$$

FIGURE 7.2: Proof rule RULESAFE for synthesis with respect to a safety requirement given by assertion $\mathit{safe}(v)$.

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

However, from $\mathit{sum}((\mathtt{P}, -1, -1), R_1)$, if *Adam* makes a skip and *Eve* moves to `Q` instead `R`, the assertion will be eventually satisfied. If *Adam* decides to move from `E` to `B` (instead of `E` to `A`), *Eve* needs to move to `R` instead `Q` for the assertion to be satisfied. Therefore, the safe strategy for *Eve* 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 $pc = \mathtt{A}$ provides a winning strategy for *Eve*.

Theorem 8 (Correctness of rule RuleSafe)**.** *The proof rule* RuleSafe *is sound and relatively complete.* ■

Proof. 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.

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 safe strategy for the environment in which there exists a play π that violates the safety condition at position p. By Lemma 2 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.

Relative completeness If the program has a strategy to win the safety game, then there exists *sum* that satisfies premises of RuleSafe.

Let us assume that Eve has a safe strategy, i.e., the conclusion of RuleSafe holds. This strategy σ alternates between universal choices of Adam and existential choices of Eve. 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 $sum(s_0, R_0)$. The strategy guarantees that for every successor s_1 of s_0 wrt. Adam there exists a successor s_2 wrt. Eve. For every such s_2, there exists a set of FURs R_2 such that $sum(s_2, R_2)$ since s_2 is an Adam 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 $sum(s_0, R_0)$, and a state s_1 such that $call(s_0, s_1)$. For the set of FURs R_1 of s_1, we have $sum(s_1, R_1)$ since s_1 is an Env state. This shows $sum(v, R)$ satisfies S3.

Next, let us assume that for arbitrary states s_0 and s_1, we have $sum(s_0, R_0)$ and $sum(s_1, R_1)$, and $call(s_0, s_1)$. It follows that for any $s_2 \in R_1$ and a set of its FURs

R_2, $sum(s_2, R_2)$ 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 $ret(s_0, s_1)$ and $sum(s_0, R_0)$. By definition of FURs, we see that s_1 should be in R_0, satisfying S5.

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

□

7.5.2 Terminating strategies

Let us now consider a synthesis problem where Eve 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, $sum(s_0, s_0, \emptyset)$. 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 $sum(s_0, s_1, R_1)$. We require that for every state s_2 satisfying $adam(s_1, s_2)$ there exists a set of states R_2 such that $sum(s_0, s_2, R_2)$ and $R_1 \supseteq R_2$, and there exist a state s_3 such that $eve(s_2, s_3)$. We also require that there exists a set of states R_3 such that $sum(s_0, s_3, R_3)$ and $R_2 \supseteq R_3$. We also require that (s_1, s_3) is in *round*. Assume for a state s_1, $sum(s_0, s_1, R_1)$ is given. For any state s_2 such that $call(s_1, s_2)$, T3 requires that there exists a set of states R_2 such that $sum(s_2, s_2, R_2)$. 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 $call(s_1, s_2)$, $sum(s_0, s_1, R_1)$ and $sum(s_2, s_2, R_2)$. For any $s_3 \in R_2$, we require that there exists a set of states R_3 such that $sum(s_0, s_3, R_3)$ 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 $sum(s_0, s_1, R_1)$, 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.

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

$$
\begin{array}{lll}
\text{T1:} & \mathit{init}(v) & \rightarrow \mathit{sum}(v, v, \emptyset) \\
\text{T2:} & \mathit{sum}(v_1, v_2, R_1) \wedge \mathit{adam}(v_2, v_3) & \rightarrow \exists R_2 : \mathit{sum}(v_1, v_3, R_2) \wedge R_1 \supseteq R_2 \\
 & & \quad \wedge \exists v_4 : \mathit{eve}(v_3, v_4) \wedge \mathit{round}(v_2, v_4) \\
 & & \quad \wedge \exists R_3 : \mathit{sum}(v_1, v_4, R_3) \wedge R_3 \supseteq R_2 \\
\text{T3:} & \mathit{sum}(v_1, v_2, R_1) \wedge \mathit{call}(v_2, v_3) & \rightarrow \exists R_2 : \mathit{sum}(v_3, v_3, R_2) \wedge \mathit{descent}(v_1, v_3) \\
\text{T4:} & \mathit{sum}(v_1, v_2, R_1) \wedge \mathit{call}(v_2, v_3) & \\
 & \wedge \mathit{sum}(v_3, v_3, R_2) \wedge R_2(v_4) & \rightarrow \exists R_3 : \mathit{sum}(v_1, v_4, R_3) \wedge R_1 \supseteq R_3 \wedge \mathit{round}(v_2, v_4) \\
\text{T5:} & \mathit{sum}(v_1, v_2, R) \wedge \mathit{ret}(v_2, v_3) & \rightarrow R(v_3) \\
\text{T6:} & \mathit{wf}(\mathit{round}) & \\
\text{T7:} & \mathit{wf}(\mathit{descent}) &
\end{array}
$$

$$(\mathit{init}(v), \mathit{eve}(v, v'), \mathit{adam}(v, v'), \mathit{call}(v, v'), \mathit{ret}(v, v', v'')) \models F\ \mathit{dst}(v)$$

FIGURE 7.3: Proof rule RULEREACH for synthesis with respect to the termination requirement.

Theorem 9 (Correctness of rule RULEREACH)**.** *The proof rule* RULEREACH *is sound and relatively complete.* ■

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 $\mathit{sum}(v_1, v_2, R)$, $\mathit{round}(v_1, v_2)$ and $\mathit{descent}(v_1, v_2)$ that satisfy the premises of the rule, yet the conclusion of the rule does not hold. That is, there is no winning strategy for Eve.

Hence, there exists a strategy σ for Adam in which each play does not terminates. This strategy σ alternates between existential choices of Adam and universal choices of Eve. Let $\mathit{aux}(v)$ be a set of states for which σ provides existentially chosen successors wrt. Eve. Note that no play terminates from any $s \in \mathit{aux}(v)$ 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 $adam(s, s_1)$ and $eve(s_1, s_2)$,
- 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 $adam(s, s_1)$, and T2 guarantees that there exists a state s_2 such that $eve(s_1, s_2)$, $round(s, s_2)$, 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 Adam states at some procedural level if the infinite play is due to infinite intra-procedural steps by Adam which contradicts with T6.
- there exists an infinite sequence of entry states if the infinite play is due to infinite call steps by Adam which contradicts with T7

Relative completeness Let us assume that Eve 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 inπ for which σ provides a universally chosen successor w.r.t. Adam, $s_0 = entry(s)$, 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 Adam states on the same procedural level. Let $descent(v_1, v_2)$ 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 Adam state, $sum(v_1, v_2, R)$ is defined for any initial state, satisfying T1.

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

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

Let us consider a pair of states s_1 and s_2 such that $sum(s_0, s_1, R_1)$, $sum(s_2, s_2, R_2)$, and $call(s_1, s_2)$. For any $s_3 \in R_2$, we have $sum(s_0, s_3, R_3)$ since s_3 is an Adam 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 $round(s_1, s_3)$ since s_1 and s_3 are consecutive Adam states on the same procedural level, i.e. T4 is satisfied.

Now let us consider a state s_1 such that $sum(s_0, s_1, R_1)$ for $s_0 = entry(s_1)$ and $ret(s_1, s_2)$ 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 $round(v_1, v_2)$ is well-founded. Assume otherwise, i.e., there exists an infinite sequence of states $s1, s2, ...$ induced by $round(v_1, v_2)$ and Eve still terminates. As noted previously, for each pair of consecutive Adam states s_i and s_{i+1} there exists an intermediate sequence of state $s'_i ... s''_i$ such that the sequence $s_1, s'_1, .., s''_1, s_2, ..., s_i, s'_i, .., s''_i, s_{i+1}, ...$ is a play. Since this play does not terminate, we obtain a contradiction to the assumption. Hence, we conclude that $round(v_1, v_2)$ is well-founded, satisfying T6.

Similarly, we show by contradiction that descent(u, v) is well-founded, satisfying T7. □

7.6 Evaluation

In this section we describe an experimental evaluation of our compositional synthesis approach on infinite-state programs.

Implementation Our prototype implementation is based on two modules. The first module is a C frontend, derived from the CIL library [96], that transforms C code into verification conditions represented as a set of forall-exists quantified Horn constraints. This transformation is based on a sound approximation of our proof rules. The second module is the E-HSF solver which is used as deduction engine for solving the set of Horn constraints generated for each program.

Benchmarks For evaluation, we used benchmarks from the repository of the SV-COMP verification competition [15]. 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, 16, 71].

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 *h*ole.) **(2)** We use the frontend to generate a program representation in Horn clause form. **(3)** We solve the Horn clauses using E-HSF and the solution returned by E-HSF corresponds to synthesised-code to fill the hole in the code. If E-HSF succeeds in finding a solution for the Horn clauses, our approach guarantees that the device driver code with the hole replaced by the E-HSF'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.

See Figure 7.4 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` $= 1$,

```
419:  NTSTATUS IofCallDriver(PDEVICE_OBJECT DvObj, PIRP Irp) {
420:    NTSTATUS returnVal2 ;
        ...
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:  }
```

FIGURE 7.4: Part of function `IofCallDriver`.

$\texttt{DC} = 2$, $\texttt{SKIP1} = 3$, $\texttt{SKIP2} = 4$, $\texttt{MPR1} = 5$, $\texttt{MPR3} = 6$ and $\texttt{IPC} = 7$. 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 our method, we provide a template corresponding to the hole expression that reflects the choice of automaton states

$$\begin{aligned}&\textsc{Templ}(\mathit{rel})(v) = \\ &\quad (?_{\texttt{UNLOADED}} * \texttt{UNLOADED} + ?_{\texttt{NP}} * \texttt{NP} + ?_{\texttt{DC}} * \texttt{DC} + ?_{\texttt{SKIP1}} * \texttt{SKIP1} \\ &\quad +?_{\texttt{SKIP2}} * \texttt{SKIP2} + ?_{\texttt{MPR1}} * \texttt{MPR1} + ?_{\texttt{MPR3}} * \texttt{MPR3} + ?_{\texttt{IPC}} * \texttt{IPC} = \texttt{s})\end{aligned}$$

together with a template constraint

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

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

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

Results For our experiments we used a computer with an Intel Core i7 2.3 GHz CPU and 16 GB of RAM. See Table 7.1 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 E-HSF (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 13.5s. (The benchmark indeed satisfies its safety specification.) For the three code regions, `IofCalldriver`, `PptDispatch`, and `KeSetEvent`, our tool synthesises a solutions after 17.7s, 18.7s, and 22.1s, respectively.

In all case, our method is able to succeed within 1.5-2 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.

7.7 Related work

The last few years have seen much work on constraint-based software synthesis [82, 111, 115, 118]. 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 [77], and modular sketching [109]. 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 [25, 78, 86, 103, 116], using both explicit-state [116] and symbolic [101] approaches. Also well-known are algorithms for *pushdown games* [27, 120], which can be expanded into synthesis algorithms for reactive programs with procedures

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

TABLE 7.1: Evaluation of our method on device driver programs from SV-COMP

and finite-domain variables [60]. Synthesis of finite-state reactive systems from components has also been studied [85]. 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 [4–6]. 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. [4, 6] 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.

7.8 Conclusion

We have presented a constraint based approach to computing winning strategies in infinite-state recursive games. The approach consists of: (1) a notion of game summaries which generalizes the traditional procedure summaries, (2) a set of sound and relatively complete proof rules for solving such games, and (3) automation of the rules on top of an existing automated deduction engine. We demonstrate the practical promise of our approach by using a set of benchmarks, which consists of 10 device driver programs, derived from the SV-COMP repository.

Chapter 8

Future Work

We have proposed a new method for solving forall-exists quantified Horn constraints extended with well-foundedness conditions. We then show the practical applicability of our method by presenting some important problems in formal methods that our method has a potential not only to formalize declaratively and elegantly but also to solve efficiently. These include verification of branching-time temporal properties of infinite-state programs and computing winning strategies in two-player graph games over the state space of infinite-state programs.

However, more work remains before claiming an automated technique that takes a program (or a partial program) written in a given programming language and performs verification (or synthesis) with respect to a given temporal property. We describe a few directions for future work below.

More temporal logics The most commonly used temporal logic in the industry is LTL. An interesting work will be extending our method of solving games with general LTL objectives to a full Horn constraints based LTL verification method. Then, combining the resulting LTL verification method with the existing CTL verification method can be a further line of research for the verification of CTL*, which is more expressive than both LTL and CTL.

More games applications There are various application domains in formal methods whose analyses are amenable to game-theoretic approaches. One is verification of autonomous systems as autonomous systems by definition comprises agent(s) and environment players. We are also especially interested in applying our method to various reactive synthesis questions such as controller synthesis arising out of embedded systems and robotics.

Chapter 9

Summary and Conclusion

We started by introducing the notions of programs, games, well-foundedness and forall-exists quantified Horn constraints.

We then described the algorithm E-HSF for solving forall-exists quantified Horn constraints with well-foundedness conditions. The algorithm makes use of a set of user provided templates to discover witnesses for existentially quantified variables by applying a counterexample-guided abstraction refinement(CEGAR). The refinement loop of E-HSF collects a global constraint that declaratively determines which witnesses can be chosen from the space of witnesses provided by a template. The chosen witnesses are used to replace existential quantification, and then the resulting universally quantified Horn constraints are passed to a solver for such constraints. We claimed the correctness of E-HSF by presenting a condition under which the algorithm is sound, and by discussing how the global constraint collected in each refinement loop of the CEGAR algorithm ensures progress of refinement.

Next, we described how our solving algorithm can be applied for verification of properties given in temporal logics CTL and CTL+FO. We provided proof rules for generating forall-exists quantified Horn constraints from an infinite-state program, given as a transition system, and a temporal property, given in CTL and CTL+FO. The proof rules for temporal quantifiers for both logics are adopted from existing proof rules. However, we introduced novel yet simple and declarative proof rules for first-order quantifiers of CTL+FO. This gives an automated method for verifying branching-time temporal properties. We demonstrated the practical applicability of the method for both logics by applying a prototype implementation of the method on a set of industrial benchmarks from the PostgreSQL database server, the SoftUpdates patch system, the Windows OS kernel.

Then, we presented a Horn constraint-based approach to computing winning strategies in two-player graph games over the state space of infinite-state programs. The approach handles games with winning conditions given by safety, reachability, and general Linear Temporal Logic (LTL) properties. For each property class, we gave a deductive proof rule that describes a winning strategy for a particular player. We showed that the proof rules are sound and relatively complete. While these proof rules are our main contribution, we also provided a prototype implementation of our rules on top of the E-HSF deduction engine. We demonstrated the practical applicability of the proof rules through case studies, including a challenging "Cinderella-Stepmother game", and examples derived from prior works on program synthesis and repair.

Finally, we proposed an extension to our method of computing winning strategies for infinite-state graph games to recursive games where games are played on the configuration graphs of programs with recursion and unbounded data. We introduced a generalization of traditional summaries, called game summaries, that allow compositional reasoning about strategies in the presence of procedures and recursion. We provided sound and relatively complete proof rules for solving recursive games with safety and reachability objectives. A sound approximation of the proof rules is automated using E-HSF. We demonstrated the practical promises of our method by applying it over a set of benchmarks from SV-COMP verification competition.

This dissertation demonstrates that solving forall-exists quantified Horn constraints can provide a framework not only to formalize declaratively and elegantly but also to solve efficiently various problems in the area of temporal program verification and synthesis. In addition, we hope that our work might lead to advent of successful tools for solving existential properties, analogous to the successful tools for solving universal properties.

Bibliography

[1] A. Albarghouthi, Y. Li, A. Gurfinkel, and M. Chechik. Ufo: A framework for abstraction- and interpolation-based software verification. In *CAV*, 2012.

[2] R. Alur, P. Cerný, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. In *POPL*, pages 98–109, 2005.

[3] R. Alur and S. Chaudhuri. Temporal reasoning for procedural programs. In *VMCAI*, pages 45–60, 2010.

[4] R. Alur, S. Chaudhuri, and P. Madhusudan. A fixpoint calculus for local and global program flows. In *POPL*, pages 153–165, 2006.

[5] R. Alur, S. Chaudhuri, and P. Madhusudan. Languages of nested trees. In *CAV*, pages 329–342, 2006.

[6] R. Alur, S. Chaudhuri, and P. Madhusudan. Software model checking using languages of nested trees. *ACM Trans. Program. Lang. Syst.*, 2011.

[7] T. Ball and O. Kupferman. An abstraction-refinement framework for multi-agent systems. In *LICS*, pages 379–388. IEEE, 2006.

[8] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, 2002.

[9] G. Banda and J. P. Gallagher. Constraint-based abstract semantics for temporal logic: A direct approach to design and implementation. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, 2010.

[10] F. Belardinelli, A. Lomuscio, and F. Patrizi. An abstraction technique for the verification of artifact-centric systems. In *KR*, 2012.

[11] T. A. Beyene, M. Brockschmidt, and A. Rybalchenko. Ctl+fo verification as constraint solving. In *SPIN*, 2014.

[12] T. A. Beyene, S. Chaudhuri, C. Popeea, and A. Rybalchenko. A constraint-based approach to solving games on infinite graphs. In *POPL*, 2014.

[13] T. A. Beyene, S. Chaudhuri, C. Popeea, and A. Rybalchenko. Recursive games for compositional program synthesis. In *VSTTE*, 2015.

[14] T. A. Beyene, C. Popeea, and A. Rybalchenko. Solving existentially quantified Horn clauses. In *CAV*, 2013.

[15] D. Beyer. Second competition on software verification - (Summary of SV-COMP 2013). In *TACAS*, pages 594–609, 2013.

[16] D. Beyer and M. E. Keremoglu. CPAchecker: A tool for configurable software verification. In *CAV*, 2011.

[17] D. Beyer and S. Löwe. Explicit-state software model checking based on CEGAR and interpolation. In *FASE*, 2013.

[18] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI*, 2003.

[19] M. Bodlaender, C. Hurkens, V. Kusters, F. Staals, G. Woeginger, and H. Zantema. Cinderella versus the Wicked Stepmother. In *IFIP TCS*, pages 57–71, 2012.

[20] M. Bofill, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. The Barcelogic SMT solver. In *CAV*, pages 294–298, 2008.

[21] J. Bohn, W. Damm, O. Grumberg, H. Hungar, and K. Laster. First-order-CTL model checking. In *FSTTCS*, 1998.

[22] C. Borralleras, S. Lucas, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. SAT modulo linear arithmetic for solving polynomial constraints. *J. Autom. Reasoning*, 48(1):107–131, 2012.

[23] A. R. Bradley, Z. Manna, and H. B. Sipma. Polyranking for polynomial loops. *Automata, Languages and Programming*, pages 1349–1361, 2005.

[24] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani. The MathSAT 4SMT solver. In *CAV*, 2008.

[25] J. R. Büchi and L. Landweber. Solving sequential conditions by finite-state strategies. *Trans. Amer. Math. Soc.*, 138:295–311, 1969.

[26] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 1020 states and beyond. In *Logic in Computer Science, 1990. LICS '90, Proceedings., Fifth Annual IEEE Symposium on e*, pages 428–439, Jun 1990.

[27] T. Cachat. Symbolic strategy synthesis for games on pushdown graphs. In *ICALP*, pages 704–715. 2002.

[28] T. Cachat. Uniform solution of parity games on prefix-recognizable graphs. *Electronic Notes in Theoretical Computer Science*, 68(6):71–84, 2003.

[29] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[30] S. Chaki, E. M. Clarke, O. Grumberg, J. Ouaknine, N. Sharygina, T. Touili, and H. Veith. State/event software verification for branching-time specifications. In *IFM*, volume 3771, pages 53–69. Springer, 2005.

[31] K. Chatterjee and L. Doyen. Energy parity games. *TCS*, 2012.

[32] E. Clarke, Y. Lu, B. Com, H. Veith, and S. Jha. Tree-like counterexamples in model checking. In *In Proceedings of the 17 th Annual IEEE Symposium on Logic in Computer Science (LICS'02)*. IEEE Computer Society, 2002.

[33] E. M. Clarke. Temporal logic model checking: Two techniques for avoiding the state explosion problem. In *Proceedings of the 2Nd International Workshop on Computer Aided Verification*, CAV '90, 1991.

[34] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[35] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, 2004.

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

[37] M. Colón, S. Sankaranarayanan, and H. Sipma. Linear invariant generation using non-linear constraint solving. In *CAV*, 2003.

[38] B. Cook, H. Khlaaf, and N. Piterman. Faster temporal reasoning for infinite-state programs. 2014.

[39] B. Cook and E. Koskinen. Reasoning about nondeterminism in programs. In *PLDI*, 2013.

[40] B. Cook, E. Koskinen, and M. Vardi. Temporal property verification as a program analysis task. *Form. Methods Syst. Des.*, 41(1):66–82, Aug. 2012.

[41] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI*, 2006.

[42] B. Cook, A. Podelski, and A. Rybalchenko. Summarization for termination: no return! *Formal Methods in System Design*, 35(3), 2009.

[43] 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.

[44] B. Cui, Y. Dong, X. Du, K. N. Kumar, C. R. Ramakrishnan, I. V. Ramakrishnan, A. Roychoudhury, S. A. Smolka, and D. S. Warren. Logic programming and model checking. In *PLILP/ALP*, 1998.

[45] A. Da Costa, F. Laroussinie, and N. Markey. Quantified CTL: expressiveness and model checking. In *CONCUR*, 2012.

[46] L. De Alfaro, T. Henzinger, and R. Majumdar. Symbolic algorithms for infinite-state games. In *CONCUR*, pages 536–550. Springer, 2001.

[47] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[48] S. Demri, A. Finkel, V. G. Govert, and V. Drimmelen. Model checking ctl* over flat presburger counter systems. *JANCL*, 2010.

[49] E. A. Emerson. Handbook of theoretical computer science (vol. b). chapter Temporal and Modal Logic. 1990.

[50] E. A. Emerson and C. Jutla. Tree automata, mu-calculus and determinacy. In *FOCS*, pages 368–377. IEEE, 1991.

[51] E. A. Emerson and K. S. Namjoshi. Automatic verification of parameterized synchronous systems (extended abstract). In *Proceedings of the 8th International Conference on Computer Aided Verification*, CAV '96, pages 87–98. Springer-Verlag, 1996.

[52] F. Emmes, T. Enger, and J. Giesl. Proving non-looping non-termination automatically. In *IJCAR*, 2012.

[53] H. Fecher and M. Huth. Ranked predicate abstraction for branching time: Complete, incremental, and precise. In *ATVA*, pages 322–336. Springer, 2006.

[54] H. Fecher and S. Shoham. Local abstraction–refinement for the μ-calculus. *STTT*, 13(4):289–306, 2011.

[55] F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Generalization strategies for the verification of infinite state systems. *Theory and Practice of Logic Programming*, 13:175–199, 2 2013.

[56] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In *CAV*, pages 53–65, 2001.

[57] E. Grädel, W. Thomas, and T. Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research*, 2002.

[58] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*, 1997.

[59] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, 2012.

[60] A. Griesmayer, R. Bloem, and B. Cook. Repair of boolean programs with an application to C. In *CAV*, pages 358–371. Springer, 2006.

[61] O. Grumberg, M. Lange, M. Leucker, and S. Shoham. Don't know in the μ-calculus. In *VMCAI*, pages 233–249, 2005.

[62] O. Grumberg, M. Lange, M. Leucker, and S. Shoham. When not losing is better than winning: Abstraction and refinement for the full μ-calculus. *Information and Computation*, 205(8):1130–1148, 2007.

[63] A. Gupta, T. A. Henzinger, R. Majumdar, A. Rybalchenko, and R.-G. Xu. Proving non-termination. In *POPL*, 2008.

[64] A. Gupta, R. Majumdar, and A. Rybalchenko. From tests to proofs. In *TACAS*, 2009.

[65] A. Gurfinkel and M. Chechik. Why waste a perfectly good abstraction? In *TACAS*, pages 212–226. 2006.

[66] A. Gurfinkel, O. Wei, and M. Chechik. Yasm: A software model-checker for verification and refutation. In T. Ball and R. B. Jones, editors, *CAV*, Lecture Notes in Computer Science, pages 170–174. Springer, 2006.

[67] S. Hallé, R. Villemaire, O. Cherkaoui, and B. Ghandour. Model checking data-aware workflow properties with CTL-FO$^+$. In *EDOC*, 2007.

[68] A. Harding, M. Ryan, and P.-Y. Schobbens. A new algorithm for strategy synthesis in LTL games. In *TACASs*, pages 477–492. Springer, 2005.

[69] Z. Hassan, A. R. Bradley, and F. Somenzi. Incremental, inductive ctl model checking. In *Proceedings of the 24th International Conference on Computer Aided Verification*, 2012.

[70] T. A. Henzinger, R. Jhala, and R. Majumdar. Counterexample-guided control. In *ICALP*, pages 886–902, 2003.

[71] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL*, 2004.

[72] J. Heusser and P. Malacaria. Quantifying information leaks in software. In *ASAC*, 2010.

[73] K. Hoder, N. Bjørner, and L. de Moura. μZ- an efficient engine for fixed points with constraints. In *CAV*, 2011.

[74] I. Hodkinson, F. Wolter, and M. Zakharyaschev. Decidable and undecidable fragments of first-order branching temporal logics. In *LICS*, 2002.

[75] A. J. C. Hurkens, C. A. J. Hurkens, and G. J. Woeginger. How Cinderella won the bucket game (and lived happily ever after). *Mathematics Magazine*, 84(4):pp. 278–283, 2011.

[76] M. Janota, W. Klieber, J. Marques-Silva, and E. M. Clarke. Solving QBF with counterexample guided refinement. In *SAT*, 2012.

[77] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *ICSE*, 2010.

[78] B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *CAV*, pages 226–238, 2005.

[79] M. Jurdziński. Small progress measures for solving parity games. In *STACS*, pages 290–301, 2000.

[80] T. Kahsai, J. A. Navas, A. Gurfinkel, and A. Komuravelli. The seahorn verification framework. In *CAV*, 2015.

[81] Y. Kesten and A. Pnueli. A compositional approach to CTL* verification. *Theor. Comput. Sci.*, 331(2-3):397–428, 2005.

[82] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete functional synthesis. In *PLDI*, 2010.

[83] O. Kupferman and M. Y. Vardi. Robust satisfaction. In *CONCUR*, pages 383–398, 1999.

[84] O. Kupferman, M. Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *J. ACM*, 47(2):312–360, 2000.

[85] Y. Lustig and M. Y. Vardi. Synthesis from component libraries. *STTT*, 15(5-6):603–618, 2013.

[86] P. Madhusudan. Synthesizing reactive programs. In *CSL*, pages 428–442, 2011.

[87] Z. Manna and A. Pnueli. Completing the temporal picture. *Theor. Comput. Sci.*, 83(1):91–130, 1991.

[88] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[89] Z. Manna and A. Pnueli. *Temporal verification of reactive systems: safety.* 1995.

[90] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *TOPLAS*, 2(1):90–121, 1980.

[91] D. Martin. Borel determinacy. *The Annals of Mathematics*, 102(2):363–371, 1975.

[92] K. L. McMillan. *Symbolic Model Checking.* 1993.

[93] K. L. McMillan. Lazy abstraction with interpolants. In *CAV*, 2006.

[94] K. L. McMillan and A. Rybalchenko. Computing relational fixed points using interpolation. Technical report, 2012. available from authors.

[95] K. S. Namjoshi. Certifying model checkers. In *CAV*, 2001.

[96] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC*, 2002.

[97] E.-R. Olderog and K. R. Apt. Fairness in parallel programs: The transformational approach. *ACM Trans. Program. Lang. Syst.*, 10(3), 1988.

[98] A. C. Patthak, I. Bhattacharya, A. Dasgupta, P. Dasgupta, and P. Chakrabarti. Quantified computation tree logic. *Information processing letters*, 82(3):123–129, 2002.

[99] É. Payet and F. Spoto. Experiments with non-termination analysis for Java Bytecode. *Electr. Notes Theor. Comput. Sci.*, 253(5), 2009.

[100] W. Penczek, B. Wozna, and A. Zbrzezny. Bounded model checking for the universal fragment of ctl. *Fundam. Inf.*, 2002.

[101] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. In *VMCAI*, pages 364–380, 2006.

[102] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, 1977.

[103] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL*, pages 179–190. ACM, 1989.

[104] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, 2004.

[105] A. Podelski and A. Rybalchenko. Transition invariants. In *LICS*, 2004.

[106] A. Rensink. Model checking quantified computation tree logic. In *CONCUR*, 2006.

[107] A. Schrijver. *Theory of linear and integer programming.* Wiley-Interscience series in discrete mathematics and optimization. Wiley, 1999.

[108] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. *Program Flow Analysis: Theory and Applications*, 1981.

[109] R. Singh, R. Singh, Z. Xu, R. Krosnick, and A. Solar-Lezama. Modular synthesis of sketches using models. In *VMCAI*, 2014.

[110] M. Slanina. Control rules for reactive system games. In *AAAI Spring Symposium on Logic-Based Program Synthesis*, 2002.

[111] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415, 2006.

[112] F. Song and T. Touili. Efficient ctl model-checking for pushdown systems. In *In CONCUR*, 2011.

[113] F. Song and T. Touili. Pommade: Pushdown model-checking for malware detection. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 607–610. ACM, 2013.

[114] S. Srivastava and S. Gulwani. Program verification using templates over predicate abstraction. In *PLDI*, 2009.

[115] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *POPL*, pages 313–326, 2010.

[116] W. Thomas. On the synthesis of strategies in infinite games. In *STACS*, pages 1–13, 1995.

[117] M. Y. Vardi. Verification of concurrent programs: The automata-theoretic framework. *Ann. Pure Appl. Logic*, 51(1-2):79–98, 1991.

[118] M. T. Vechev, E. Yahav, and G. Yorsh. Abstraction-guided synthesis of synchronization. In *POPL*, 2010.

[119] I. Walukiewicz. Model checking ctl properties of pushdown systems. In S. Kapoor and S. Prasad, editors, *FSTTCS*, Lecture Notes in Computer Science, pages 127–138. Springer, 2000.

[120] I. Walukiewicz. Pushdown processes: Games and model-checking. *Information and computation*, 164(2):234–263, 2001.

[121] C. M. Wintersteiger, Y. Hamadi, and L. M. de Moura. Efficiently solving quantified bit-vector formulas. *Formal Methods in System Design*, 2013.

[122] Y. Xie and A. Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM TOPLAS*, 29(3), 2007.

[123] W. Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theoretical Computer Science*, 200(1):135–183, 1998.