Strategy Synthesis for Linear Arithmetic Games

AZADEH FARZAN, University of Toronto
ZACHARY KINCAID, Princeton University

Many problems in formal methods can be formalized as two-player games. For several applications—program synthesis, for example—in addition to determining which player wins the game, we are interested in computing a winning strategy for that player. This paper studies the strategy synthesis problem for games defined within the theory of linear rational arithmetic. Two types of games are considered. A *satisfiability game*, described by a quantified formula, is played by two players that take turns instantiating quantifiers. The objective of each player is to prove (or disprove) satisfiability of the formula. A *reachability game*, described by a pair of formulas defining the legal moves of each player, is played by two players that take turns choosing positions—rational vectors of some fixed dimension. The objective of each player is to reach a position where the opposing player has no legal moves (or to play the game forever). We give a complete algorithm for synthesizing winning strategies for satisfiability games and a sound (but necessarily incomplete) algorithm for synthesizing winning strategies for reachability games.

CCS Concepts: • **Theory of computation → Automated reasoning**; • **Software and its engineering → Automatic programming**;

Additional Key Words and Phrases: Logical games, Functional synthesis, Reactive synthesis

ACM Reference format:
Azadeh Farzan and Zachary Kincaid. 2018. Strategy Synthesis for Linear Arithmetic Games. *Proc. ACM Program. Lang.* 2, POPL, Article 61 (January 2018), 31 pages.
https://doi.org/10.1145/3158149

1 INTRODUCTION

Logical games are two-player games in which the rules of the game are defined by logical formulas. Computing winning strategies for logical games has several applications in formal methods, specifically in verification and synthesis of programs. This paper presents algorithms for computing winning strategies for *linear arithmetic games*, in which the moves of the game are defined by linear (rational or integer) arithmetic formulas. This paper considers two classes of linear arithmetic games, namely *satisfiability games* and *reachability games*. We focus on logical games *modulo* the theory of linear rational arithmetic, both due to the pervasive use of this theory in formal methods and the fact that it is relatively well understood.

A quantified formula can be considered as a *satisfiability game* between two players SAT and UNSAT: SAT controls the existential quantifiers and aims to prove the formula is satisfiable, while UNSAT controls the universal quantifiers and aims to prove the formula unsatisfiable [Hintikka 1982]. A winning strategy for SAT is an algorithm for choosing values for existentially quantified variables based on the preceding choices made by the UNSAT player, such that after all choices have been made for both players, the formula evaluates to true. Dually, a winning strategy for the

2475-1421/2018/1-ART61
https://doi.org/10.1145/3158149

UNSAT player is an algorithm for choosing values of universally quantified variables to make the formula evaluate to false. Applications of satisfiability games include:

- *Functional synthesis.* Approaching program synthesis as theorem proving problem has a long history [Manna and Waldinger 1980] and remains a competitive approach today [Kuncak et al. 2010; Reynolds et al. 2015]. A typical synthesis conjecture has the form $\exists f.\forall x.\varphi(f,x)$ where f is the function to be synthesized, x is a parameter to the function, and $\varphi(f,x)$ is a logical specification of the function. A SAT strategy for such a formula corresponds to an implementation of f that satisfies its specification.
- *Adversarial planning.* Consider the problem of finding a plan to accomplish a task within some fixed window of time while interacting with adversarial agents. Such a problem can be encoded as a satisfiability problem where SAT chooses the actions of the plan and attempts to accomplish the goal and UNSAT controls the adversaries and attempts to avoid the goal [Dantam et al. 2016]. A winning SAT strategy corresponds to a plan.

There are a variety of decision procedures for linear rational arithmetic that can be used to determine which player wins a satisfiability game [Bjørner and Janota 2015; Farzan and Kincaid 2016]. However, this paper contributes the first (to our knowledge) complete procedure for strategy synthesis for such games. Our approach is based on our recent decision procedure for linear rational/integer arithmetic [Farzan and Kincaid 2016]. We show that using Craig interpolation, it is possible to extract winning strategies from the certificate produced by this procedure (Section 4). The resulting procedure is complete in the sense that it is guaranteed to synthesize a winning strategy (for the winning player) in finite time.

A *reachability game* is a game between two players (SAFE and REACH) determined by three formulas, which define the initial positions of the game, the moves of SAFE and the moves of REACH. SAFE and REACH alternate moves until one has no available moves and loses. In contrast to satisfiability games which have finite duration, reachability games can be played indefinitely (if SAFE plays with legal moves indefinitely, then SAFE wins the game). Applications of reachability games include:

- *Reactive synthesis*: Reactive synthesis is the problem of a finding a potentially non-terminating program that interacts with an environment [Alur et al. 2016]. For example, consider a robot controller that must avoid crashing into moving obstacles—this problem can be modeled as a reachability game where SAFE controls the robot and attempts to avoid the adversaries, and REACH controls the obstacles and attempts to crash into the robot. A winning strategy for SAFE is a program that can be executed by the robot to avoid crashes.
- *Branching-time verification*: the connection between branching-time verification and games is well known [Cook and Koskinen 2013]. A branching-time safety property can be encoded as a logical game where SAFE controls existential path quantifiers and attempts to avoid some bad state, while REACH controls universal path quantifiers and attempts to reach a bad state.
- *Modular verification*: modular verification proves that a program satisfies a property of interest without having access to the entire code (e.g., the program may make unknown library calls, or call a different module that is being verified independently). This style of modular verification requires synthesizing non-trivial specifications for unknown such that, assuming that specification holds, the property of interest holds [Albarghouthi et al. 2016]. This problem can be encoded as a reachability game where the REACH controls the program and attempts to reach an error, and SAFE controls the unknown code and attempts to avoid the error. The specification for the unknown code is part of the proof that a winning strategy for SAFE is indeed winning.

There has been a great deal of decidability results and algorithms for solving finite state games of infinite duration [Emerson and Jutla 1991; Kupferman and Vardi 1999; Pnueli and Rosner 1989; Thomas 1995]. Linear reachability games are *infinite state* and infinite duration, a class of games that have received relatively less attention [Beyene et al. 2014; De Alfaro et al. 2001]. There is no algorithm that can synthesize a winning strategy for the entire class, so it is important to have a variety of different techniques with different strengths. Our approach to solving reachability games is based on an insight from automated program verification [McMillan 2006]. To verify a property of a program, we unroll it, prove that the property holds on the unrolling, and then attempt to generalize the proof so that it holds for the entire program. Similarly, our strategy synthesis procedure for reachability games unrolls the game, computes a winning strategy for the unrolling, and attempts to generalize the strategy to the infinite-length game. This technique for reachability games crucially makes use of strategy synthesis for satisfiability games, which is a problem also addressed in this paper.

This paper contributes *fully automated* procedures for synthesizing winning strategies for both satisfiability and reachability games. In contrast to many (but not all) approaches to synthesis, our technique does not require user-supplied hints (such as a program grammar as in syntax guided synthesis (SyGuS) [Alur et al. 2013], or solution hints in the form of templates in the style of Beyene et al. [2014]; Srivastava et al. [2013]). While such hints are sometimes natural or even desirable, fully automated strategy synthesis remains a fundamental algorithmic challenge.

The remainder of this paper is structured as follows. In the next section, we give two examples for satisfiability and reachability games to establish intuition. In Section 3, we formalize the class of logical games of interest and establish some foundational results. Sections 4 and 5 present our strategy synthesis algorithms for satisfiability and reachability games, respectively. Section 6 discusses some additional properties of logical games and our algorithms for solving them. Experimental evaluation of both algorithms on a number of case studies appears in Section 7. Section 8 discusses the related work.

2 ILLUSTRATIVE EXAMPLES

We use two examples to explain the high level ideas behind the two main contributions of this paper: (i) strategy synthesis for (bounded) satisfiability games (Section 2.1) and (ii) strategy synthesis for (unbounded) reachability games (Section 2.2).

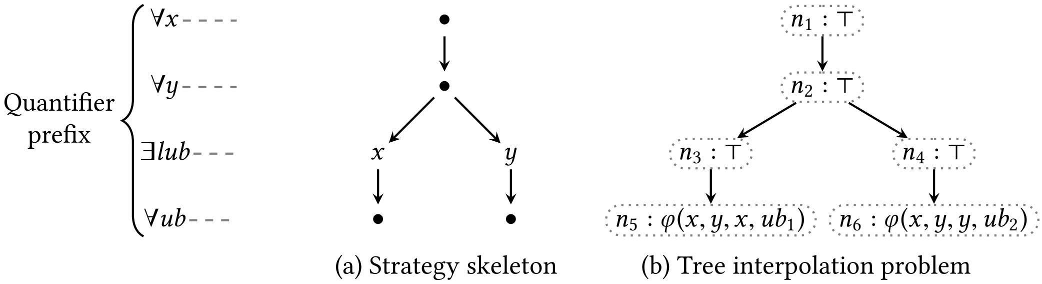
2.1 Linear Arithmetic Satisfiability Games

We use a *functional synthesis* example to illustrate our algorithm for strategy synthesis for satisfiability games. Functional synthesis is the problem of synthesizing a function that meets a given specification. Many functional synthesis queries can be encoded as logical formulas of the form: *for all* inputs, *there exists* an output such that some specification holds. Concretely, suppose that we wish to synthesize a function that computes the *least upper bound* of two rational parameters. This synthesis query can be encoded as the formula $\forall x.\forall y.\exists lub.\forall ub.\varphi(x, y, lub, ub)$, where

$$\varphi(x, y, lub, ub) \triangleq lub \geq x \;\wedge lub \geq y \;\wedge\; [(ub \geq x \;\wedge ub \geq y) \implies ub \geq lub]\,.$$

The above quantified formula can be interpreted as a game played between two players, SAT and UNSAT, whose goals are to prove that the formula is respectively satisfiable and unsatisfiable. A winning strategy for the SAT player (a function that, given the values that the UNSAT player chooses for x and y, selects a value for lub such that φ holds no matter what the UNSAT player chooses for ub) corresponds to an implementation of the least upper bound function.

Farzan and Kincaid [2016] give a decision procedure for linear rational arithmetic (and linear integer arithmetic) that operates by synthesizing a winning *strategy skeleton* for one of the two



(a) Strategy skeleton (b) Tree interpolation problem

Fig. 1. Winning strategy skeleton and its corresponding tree interpolation problem

players. The winning strategy skeleton for the SAT player (of our running example) is illustrated in Figure 1(a). The skeleton is a tree with one level for each quantifier in the formula. Levels that correspond to universally quantified variables (chosen by the UNSAT player) are marked with • placeholders; levels that correspond to existentially quantified variables (chosen by the SAT player) are marked with its possible moves. The skeleton encodes the information that, there is some way for the SAT player to win the game by instantiating *lub* with either x or y (the choice of the UNSAT player). The skeleton is not a strategy, however, because it does not encode when to choose x and when to choose y. For instance, the strategy skeleton in Figure 1(a) would also be winning for the analogous specification of greatest lower bound.

In this paper, we give a method for synthesizing winning strategies from winning strategy skeletons using tree interpolation [Blanc et al. 2013]. Figure 1(b) depicts the tree interpolation problem corresponding to the skeleton in Figure 1(a). The structure of this tree is the same as that of the strategy skeleton. Each leaf is labeled with a formula that represents the constraint that the game proceeds along the corresponding path of the skeleton (i.e., SAT chooses x along the left path and y along the right) and SAT *loses*. Notice the appearance of two distinct copies ub_1 and ub_2 of the upper bound variable: this duplication reflects the fact that UNSAT may respond differently depending on whether SAT chooses x or y. The fact that the strategy skeleton is winning implies that the conjunction of these two formulas is inconsistent; or more intuitively, both moves cannot be losing moves. This inconsistency guarantees the existence of a *tree interpolant*. A tree interpolant is mapping I from the nodes of the tree to formulas that obey certain conditions (given formally in Definition 4.5). For this particular tree interpolation problem, we are interested in the formulas $I(n_3)$ and $I(n_4)$ that label the two nodes n_3 and n_4 at the $\exists lub$ level. The conditions on tree interpolants guarantee that (1) $\neg\varphi(x, y, x, ub_1) \models I(n_3)$, (2) $\neg\varphi(x, y, y, ub_2) \models I(n_4)$, (3) $I(n_3) \land I(n_4)$ is inconsistent, and (4) $I(n_3)$ and $I(n_4)$ are expressed over the symbols x and y that are common to the branches. The *negation* of $I(n_3)$ is a condition under which choosing x is a winning strategy for SAT (condition (1)), while the negation of $I(n_4)$ is a condition under which y wins (condition (2)). Condition (3) ensures that at least one of $\neg I(n_3)$ and $\neg I(n_4)$ must hold. The fact that $I(n_3)$ and $I(n_4)$ are expressed only in terms of x and y (condition (4)) makes them suitable for use in conditionals (SAT's strategy may not branch depending on the value of *ub*, because UNSAT chooses *ub after* SAT chooses *lub*). One solution to this tree interpolation problem yields $I(n_3) = x < y$ and $I(n_4) = y < x$, which leads to the following implementation of *lub*:

$$lub(x, y) = \textbf{if } x \geq y \textbf{ then } x \textbf{ else } y\ .$$

2.2 Linear Arithmetic Reachability Games

We use the Cinderella-Stepmother game [Bodlaender et al. 2012; Hurkens et al. 2011] which is motivated by data applications in wireless sensor networks, and the minimum-backlog problem. It is a turn-based two-player game on an undirected graph, with players *Cinderella* and her *Stepmother*. The graph is a regular pentagon (same configuration as Beyene et al. [2014]; Hurkens et al. [2011]) for the purposes of this example. Each vertex of the pentagon contains a bucket (which represents a buffer) that can store water (which represents data). All buckets have the same capacity c.

In each round, the Stepmother (the reachability player) distributes exactly one litre of water arbitrarily over the buckets. Then, Cinderella chooses a pair of neighbouring buckets and empties them. The Stepmother's goal is to cause an overflow in at least one bucket. Cinderella's goal is to prevent any overflows from happening. When $c < 2$, the Stepmother can eventually force an overflow. When $c \geq 2$, Cinderella can manage to keep the game running forever avoiding an overflow.

It is straightforward to observe that for $c < 1$, the Stepmother can win the game in the first round by pouring her entire litre into a single bucket. It is also easy to observe that Cinderella can win the game for $c \geq 3$, using a *round-robin strategy*, that is, she starts from some bucket, and goes around the pentagon emptying the two buckets adjacent to the two buckets that were emptied in the last round, independent of Stepmother's moves. Since round-robin empties any bucket in at most every three rounds, it is a winning strategy if $c \geq 3$. The Stepmother can beat Cinderella's round-robin strategy for $c < 3$ by putting all of her 1 litre into bucket b_5 in the first three rounds, assuming Cinderella starts her round-robin strategy at bucket b_1.

Let us assume $c = 3$, for the purpose of this example. Note that this is an infinite length game that Cinderella (the safety player) can win. Our algorithm discovers the winning strategy for Cinderella by solving bounded (satisfiability) games of increasing length, and attempting to prove that the strategy discovered for an instance of the bounded game generalizes to a strategy to win the unbounded game.

It is easy to see that for $c = 3$, the Stepmother cannot win in the first three rounds; she can, at best, bring the capacity of one single bucket up to 3 by the end of the third round. Let us quickly discuss how the bounded versions of this game are solved for 1, 2, and 3 rounds, to explain how our algorithm works. Note that each *round* consists of two (game) turns, one for each player.

Round 1: For a game with exactly 1 round, Cinderella can play the trivial strategy of emptying b_1 and b_2. It is clear that always emptying buckets b_1 and b_2 is not a general strategy to win the unbounded game, and therefore, the algorithm proceeds to solve the game for 2 rounds.

Round 2: Cinderella has to decide what move to make in the second round to win the 2-round game, with the move for round 1 already decided in the previous round. Cinderella can repeat the move of emptying buckets b_1 and b_2 and win the 2-round game. Similar to round 1, the strategy is not generalizable to a winning one for the unbounded game, and therefore, the new goal is set to solve a game of 3 rounds.

Round 3: Similar to Round 2, Cinderella can again empty buckets b_1 and b_2 and win the 3-round game. The strategy is still not generalizable to a winning one for the unbounded game, and the new goal is set to solve a game of 4 rounds.

Round 4: With the moves for rounds 1-3 already fixed, Cinderella has to pick a move for round 4 of the game to win. The solver of the satisfiability game of depth 4 determines that no such move exists. That is, once Cinderella has fixed her moves in the first three rounds as discussed above, the Stepmother has a winning strategy for a game of 4 rounds regardless of what move Cinderella makes in round 4. As illustrated in Figure 2, while Cinderella keeps emptying buckets b_1 and b_2, the Stepmother puts all of her 1 litre of water into bucket b_3 at every turn.

Now, the solver has to *backtrack*. The first attempt will be based on the conjecture that the move made in the round before last (round 3) was the mistake that needs to be corrected. The new goal

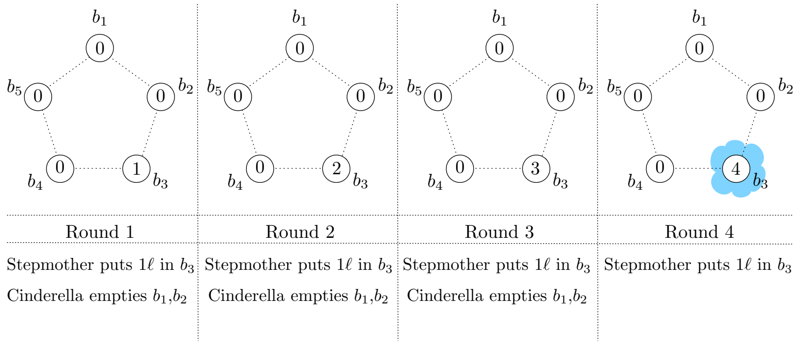

b_1
b_2
b_3
b_4
b_5
Round 1
Round 2
Round 3
Round 4
Stepmother puts 1ℓ in b_3
Cinderella empties b_1,b_2
Stepmother puts 1ℓ in b_3
Cinderella empties b_1,b_2
Stepmother puts 1ℓ in b_3
Cinderella empties b_1,b_2
Stepmother puts 1ℓ in b_3

Fig. 2. Stepmother wins in round 4, if Cinderella's first 3 moves are fixed at emptying b_1 and b_2.

for the algorithm is to *revise* Cinderella's move at round 3 *and* select a new move for round 4 that will win Cinderella the game in 4 rounds. Since the moves at rounds 1 and 2 are fixed, the solver is locally solving a game of 2 rounds starting from the beginning of round 3. The satisfiability game solver can produce a winning strategy for Cinderella for this game. There are several such strategies. Let us assume that the one picked by the solver is:

(R 3) If $b_3 \leq 2$ then empty buckets b_4 and b_5, and if $b_5 \leq 2$ then empty buckets b_3 and b_4.
(R 4) Empty buckets b_1 and b_2.

The algorithm fails to generalize the winning strategy for this 4-round game into a winning strategy for the unbounded game. It is clear that the chosen move at round 4 is minimally good enough not to lose the game for Cinderella in 4 rounds, but is not a good move if the game continues to a 5th round as demonstrated in Figure 3 where there does not exist a good move for Cinderella for round 5.

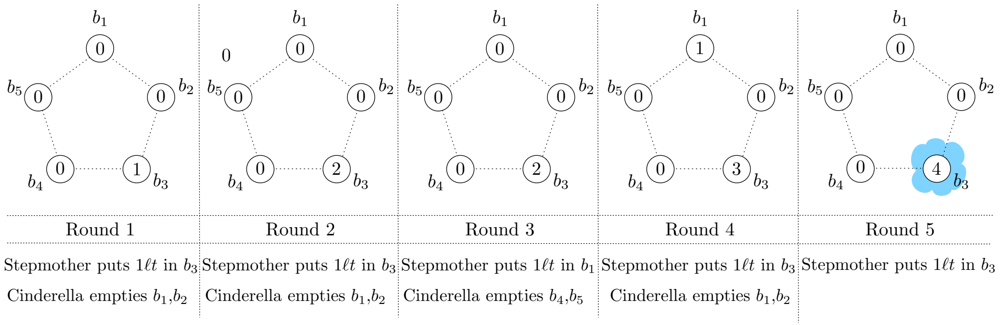

b_1
b_2
b_3
b_4
b_5
0
Round 1
Round 2
Round 3
Round 4
Round 5
Stepmother puts $1\ell t$ in b_3
Cinderella empties b_1,b_2
Stepmother puts $1\ell t$ in b_3
Cinderella empties b_1,b_2
Stepmother puts $1\ell t$ in b_1
Cinderella empties b_4,b_5
Stepmother puts $1\ell t$ in b_3
Cinderella empties b_1,b_2
Stepmother puts $1\ell t$ in b_3

Fig. 3. Stepmother wins in round 5, if Cinderella's first 4 moves are fixed as illustrated.

Round 5: With the moves for rounds 1-4 already fixed, Cinderella has no move that will guarantee a win in the 5th round as discussed before. Again, the algorithm has to *backtrack*. Cinderella's move in round 4 will have to be revised, and the algorithm checks if the game can be won for Cinderella by recomputing moves for rounds 4 and 5 through solving a bounded satisfiability game of depth 2 starting from the 4th round.

Since there is a conditional move in the 3rd round, the algorithm has to compute moves for Cinderella in both eventualities for rounds 4 and 5. Effectively, think of the strategy as having a memory of the choice made in round 3. This satisfiability game of depth 2 can be won by Cinderella, and her winning moves for rounds 4 and 5 are:

- If buckets b_3 and b_4 were emptied in round 3 then:
  - (R 4) empty buckets b_4 and b_5.
  - (R 5) empty buckets b_1 and b_2.
- If buckets b_4 and b_5 were emptied in round 3 then:
  - (R 4) empty buckets b_3 and b_4.
  - (R 5) empty buckets b_1 and b_2.

Now the game is at a crucial moment. The winning strategy for the game of 5 rounds for Cinderella is good enough to generalize to the strategy to win a game of any bound.

Strategy Generalization: How can the algorithm compute a winning strategy for the unbounded game by generalizing the strategy of a 5-round game? This can be done by identifying the state of the game at round 6 with the state of the game at one of the earlier rounds, in the sense that whatever move was a winning move at an earlier state could also be a good move at this state. This type of *cycle discovery* enables the strategy to repeat itself and predict all future winning moves. The main principle behind this discovery is the well-known concept of *invariants* from program verification. Intuitively, we associate every sequence of moves with an *invariant* that holds for every position that the game may be in after that sequence; if the invariant of one sequence s implies that of another (sub-)sequence, then a strategy for continuing the game after s is already known: the game has returned to a position that was previously encountered.

Consider Cinderella's entire strategy for the game of 5 rounds:

Round (1)	Empty buckets b_1 and b_2.	
Round (2)	Empty buckets b_1 and b_2.	
Round (3)	If $b_3 \leq 2$ then empty buckets b_4 and b_5.	If $b_5 \leq 2$ then empty buckets b_3 and b_4.
Round (4)	Empty buckets b_3 and b_4.	Empty buckets b_4 and b_5.
Round (5)	Empty buckets b_1 and b_2.	Empty buckets b_1 and b_2.

The following are invariants that hold (for the game state) before Cinderella is about to make her move at each round:

(R 1) $\varphi_1 = (b1 \leq 3) \wedge (b2 \leq 3) \wedge (b_3 \leq 1) \wedge (b_4 \leq 1) \wedge (b_5 \leq 1) \wedge (b_3 + b_5 \leq 1)$
(R 2) $\varphi_2 = (b1 \leq 3) \wedge (b2 \leq 3) \wedge (b_3 \leq 2) \wedge (b_4 \leq 2) \wedge (b_5 \leq 2) \wedge (b_3 + b_5 \leq 2)$
(R 3) $\varphi_3 = (b1 \leq 1) \wedge (b2 \leq 1) \wedge (b_3 \leq 3) \wedge (b_4 \leq 3) \wedge (b_5 \leq 3) \wedge (b_3 + b_5 \leq 3)$

- Moves under the condition $b_3 \leq 2$:

  (R 4) $\varphi_4 = (b1 \leq 2) \wedge (b2 \leq 2) \wedge (b_3 \leq 1) \wedge (b_4 \leq 3) \wedge (b_5 \leq 3)$
  (R 5) $\varphi_5 = (b1 \leq 3) \wedge (b2 \leq 3) \wedge (b_3 \leq 2) \wedge (b_4 \leq 2) \wedge (b_5 \leq 2) \wedge (b_3 + b_5 \leq 2)$
  (R 6) $\varphi_6 = (b1 \leq 1) \wedge (b2 \leq 1) \wedge (b_3 \leq 3) \wedge (b_4 \leq 3) \wedge (b_5 \leq 3) \wedge (b_3 + b_5 \leq 3)$

- Moves under the condition $b_5 \leq 2$:

  (R 4) $\varphi'_4 = (b1 \leq 2) \wedge (b2 \leq 2) \wedge (b_3 \leq 1) \wedge (b_4 \leq 3) \wedge (b_5 \leq 3)$
  (R 5) $\varphi'_5 = (b1 \leq 3) \wedge (b2 \leq 3) \wedge (b_3 \leq 2) \wedge (b_4 \leq 2) \wedge (b_5 \leq 2) \wedge (b_3 + b_5 \leq 2)$
  (R 6) $\varphi'_6 = (b1 \leq 1) \wedge (b2 \leq 1) \wedge (b_3 \leq 3) \wedge (b_4 \leq 3) \wedge (b_5 \leq 3) \wedge (b_3 + b_5 \leq 3)$

Note that some invariants are intentionally weaker than the strongest condition that can be guaranteed so that we can have $\varphi_6 = \varphi'_6 = \varphi_3$. This equality guarantees that if Cinderella follows the strategy of repeating the moves in rounds 3–5 forever starting from round 3, then she can keep the game state to always remain within the safe region of

$$\varphi = (b1 \leq 3) \wedge (b2 \leq 3) \wedge (b_3 \leq 3) \wedge (b_4 \leq 3) \wedge (b_5 \leq 3).$$

How are these invariants computed? Through techniques that are inspired by invariant generation for program verification [McMillan 2006]. We will expand on this in Section 5.

A noteworthy feature of our technique is that it can solve this game without the need for any hints on the strategy to be introduced by the user. In Beyene et al. [2014], when the same instance is solved, auxiliary variables that keep track of rounds or conditions (and their roles) have to be introduced to the solver through *templates*. Our technique solves the game without the need these hints/templates.

3 LINEAR ARITHMETIC GAMES

We start by defining two classes of logical games. The first variation is *(linear arithmetic) satisfiability games*, which interprets a quantified formula in the theory of linear rational arithmetic as a game between two players, whose objectives are to prove (disprove) satisfiability of the formula. The second variation is *(linear arithmetic) reachability games*, wherein players alternate taking turns forever, or until one of the players is unable to make a move.

3.1 Satisfiability Games

The syntax of linear rational arithmetic (LRA) is as follows. The set of terms is defined by the following grammar

$$s, t \in \mathsf{Term} ::= c \mid x \mid s + t \mid c \cdot t$$

where x is a variable symbol and c is a rational number. Quantifier-free formulas are defined by the grammar

$$F, G \in \mathsf{Formula} ::= t < 0 \mid t = 0 \mid F \wedge G \mid F \vee G$$

Notice that we (without loss of generality) assume that formulas are negation-free. When we use use the negation symbol $\neg F$, we refer to the negation-free formula equivalent to $\neg F$, obtained in the obvious way. A **prenex formula** is a formula of the form

$$\varphi = Q_1 x_1 . Q_2 x_2 . \cdots Q_n x_n . F\ ,$$

where each Q_i is either $\exists$ or $\forall$, F is a quantifier-free formula (the **matrix** of φ), and all variable symbols $\{x_1, ..., x_n\}$ are assumed to be distinct. For a formula φ (or term t), we use $\mathsf{fv}(\varphi)$ ($\mathsf{fv}(t)$) to denote the free variables which appear in φ (t). A prenex formula is a **sentence** if it has no free variables.

A **valuation** is a function $M : V \to \mathbb{Q}$, where V is some finite set of variable symbols. For a term t and a valuation V, we use $[\![t]\!]^M$ to denote the interpretation of t within the valuation M. We use $M \models \varphi$ to denote that M satisfies the formula φ (M is a model of φ), defined in the usual way. For a valuation M, a variable x, and a rational number c, we use $M\{x \mapsto c\}$ to denote the extension of M where x is interpreted as c:

$$M\{x \mapsto c\} \triangleq \lambda y.\text{if } y = x \text{ then } c \text{ else } M(y)$$

A prenex sentence

$$\varphi = Q_1 x_1 . Q_2 x_2 . \cdots Q_n x_n . F$$

defines a **satisfiability game**, which is played as follows. There are two players, SAT and UNSAT, which take turns picking rational numbers. At round i of the game, if Q_i is $\exists$, then SAT chooses a rational number to assign to the variable x_i; if Q_i is $\forall$, then the choice belongs to UNSAT. After playing this game for n rounds, the players' choices define a **play** $\rho \in \mathbb{Q}^n$: a sequence of rational numbers of length n. This play can be identified with a valuation of the variables $\{x_1, ..., x_n\}$ where for each i, $\rho(x_i) \triangleq \rho_i$. The SAT player wins ρ if $\rho \models F$, otherwise UNSAT wins.

Definition 3.1 (Satisfiability Strategy). Let

$$\varphi = Q_1 x_1 . Q_2 x_2 . \cdots Q_n x_n . F$$

be a prenex LRA sentence. A **SAT strategy** for the satisfiability game φ is a function

$$f : \{\rho \in \mathbb{Q}^* : |\rho| < n \land Q_{|\rho|+1} = \exists\} \to \mathbb{Q}$$

Similarly, an **UNSAT strategy** for φ is a function

$$g : \{\rho \in \mathbb{Q}^* : |\rho| < n \land Q_{|\rho|+1} = \forall\} \to \mathbb{Q}$$

We say that a play ρ of φ **conforms** to a SAT strategy f (UNSAT strategy g) if for every $i \in \{1, ..., n\}$ such that Q_i is $\exists$ (Q_i is $\forall$),

$$\rho_i = f(\rho_1 ... \rho_{i-1}) \qquad (\rho_i = g(\rho_1 ... \rho_{i-1}))$$

(i.e., $\rho_i = f(\rho_1 ... \rho_{i-1})$ whenever $f(\rho_1 ... \rho_{i-1})$ is defined).

We say that a SAT strategy f is **winning** if SAT wins every play that conforms to f. Similarly, an UNSAT strategy g is winning if UNSAT wins every play that conforms to g. It is easy to show that φ is satisfiable if and only if the SAT player has a winning strategy in the satisfiability game for φ (and φ is unsatisfiable if and only if the UNSAT player has a winning strategy).

3.2 Reachability Games

Let $d \in \mathbb{N}$ be a natural number. A **(d-dimensional) linear reachability game** $\mathcal{G}(\mathsf{init}, \mathsf{reach}, \mathsf{safe})$ is played as follows: there are two players, REACH (the reachability player) and SAFE (the safety player), which take turns picking positions in $\mathbb{Q}^d$. At each turn i, we denote REACH's choice by $\mathbf{r}_i$, and SAFE's choice by $\mathbf{s}_i$. The *legal moves* of the game are determined by two LRA transition formulas reach and safe, each in $2d$ free variables. REACH may move from position $\mathbf{s}$ to position $\mathbf{r}$ if and only if $\mathsf{reach}(\mathbf{s}, \mathbf{r})$ holds. Similarly, SAFE may move from position $\mathbf{r}$ to position $\mathbf{s}$ if and only if $\mathsf{safe}(\mathbf{r}, \mathbf{s})$ holds. The game starts in a position chosen by REACH, which may be any position $\mathbf{r}$ such that $\mathsf{init}(\mathbf{r})$ holds.

A **play** of the game $\mathcal{G}(\mathsf{init}, \mathsf{reach}, \mathsf{safe})$ is an infinite sequence of positions $\pi = \mathbf{r}_1\mathbf{s}_1\mathbf{r}_2\mathbf{s}_2 \cdots$. We say that REACH **wins** the play π if $\mathsf{init}(\mathbf{r}_1)$ holds and there is some $k \in \mathbb{N}$ such that for all $i < k$, $\mathsf{reach}(\mathbf{s}_i, \mathbf{r}_{i+1})$ holds, and $\mathsf{safe}(\mathbf{r}_k, \mathbf{s}_k)$ does *not* hold; otherwise, SAFE wins π. That is, the first player to make an illegal move loses, and if no player makes an illegal move then the safety player wins.

Definition 3.2 (Strategies). Let $d \in \mathbb{N}$ be a natural number. A **d-dimensional REACH strategy** is a function $f : (\mathbb{Q}^d)^* \to \mathbb{Q}^d$, and a **$d$-dimensional SAFE strategy** is a function $g : (\mathbb{Q}^d)^+ \to \mathbb{Q}^d$.

Let $\mathcal{G}(\mathsf{init}, \mathsf{reach}, \mathsf{safe})$ be a d-dimensional linear reachability game. A play $\mathbf{r}_1\mathbf{s}_1\mathbf{r}_1\mathbf{s}_1 \cdots$ **conforms** to a REACH strategy f if for every i, $\mathbf{r}_{i+1} = f(\mathbf{s}_1 \cdots \mathbf{s}_i)$. A REACH strategy f is **winning** if REACH wins every play that conforms to f. Conformance and winning strategies for the safety player are defined analogously.

Example 3.3. Consider the Cinderella-Stepmother game from Section 2.2. This is a 5 dimensional reachability game with one dimension b_i corresponding to each bucket. Cinderella is the safety player, and the Stepmother is the reachability player. The initial formula is

$$\mathsf{init} \triangleq \left(\sum_{i=1}^{5} b_i = 1\right) \land \left(\bigwedge_{i=1}^{5} b_i \geq 0\right)$$

indicating that the initial position of the game (chosen by the Stepmother) is any position where the total volume in the 5 buckets is zero. Cinderella's moves are defined by the formula safe $\triangleq$ $\neg overflow \land \bigvee_{i=1}^{5} empty_i$ where

$$overflow \triangleq \left(\bigvee_{i=1}^{5} b_i > 3\right)$$

$$empty_i \triangleq b'_i, b'_{i+1}, b'_{i+2}, b'_{i+3}, b'_{i+4} = 0, 0, b_{i+2}, b_{i+3}, b_{i+4}$$

(subscript arithmetic is mod 5). Thus, the idea that Cinderella loses the game when one of the buckets overflows is encoded by the fact that Cinderella does not have any legal moves when one of the buckets is over capacity. Finally, the Stepmother's moves are defined by the formula

$$\mathsf{reach} \triangleq \sum_{i=1}^{5} b_i' = 1 + \left(\sum_{i=1}^{5} b_i\right) \land \bigwedge_{i=1}^{5} b_i' \geq b_i \ .$$

Let $\mathcal{G}(\mathsf{init}, \mathsf{reach}, \mathsf{safe})$ be a linear reachability game. For any natural number $n \in \mathbb{N}$, the **bounded linear reachability game** $\mathcal{G}_n(\mathsf{init}, \mathsf{reach}, \mathsf{safe})$ is played as $\mathcal{G}(\mathsf{init}, \mathsf{reach}, \mathsf{safe})$ except that each player makes n moves instead of infinitely many. The first player to make an illegal move loses the game; if no player makes an illegal move, the safety player wins. Bounded games are a bridge between reachability games and satisfiability games, in the sense that every bounded reachability game is equivalent to a satisfiability game. More precisely, the reachability player wins the bounded game $\mathcal{G}_n(\mathsf{init}, \mathsf{reach}, \mathsf{safe})$ if and only if the SAT player wins the satisfiability game

$$\exists \mathbf{x}_1 . \forall \mathbf{y}_1 . \cdots \exists \mathbf{x}_n . \forall \mathbf{y}_n . \mathsf{init}(\mathbf{x}_1) \land \mathsf{safe}(\mathbf{x}_1, \mathbf{y}_1) \Rightarrow \mathsf{unroll}(1, n-1)$$

where

$$\mathsf{unroll}(k, 0) \triangleq \mathit{false}$$
$$\mathsf{unroll}(k, d) \triangleq \mathsf{reach}(\mathbf{y}_k, \mathbf{x}_{k+1}) \land \Big(\mathsf{safe}(\mathbf{x}_{k+1}, \mathbf{y}_{k+1}) \Rightarrow \mathsf{unroll}(k+1, d-1)\Big) \ .$$

Proposition 3.4. *If there exists some n such that the reachability player wins the bounded game $\mathcal{G}_n$(init, reach, safe), then the reachability player wins $\mathcal{G}$(init, reach, safe).*

3.2.1 The Strategy Synthesis Problem. In this paper, we are interested in the algorithmic problem of synthesizing winning strategies for reachability games. Any strategy synthesis procedure must inevitably compute a finite description of a strategy within some suitable description language. Our procedure synthesizes strategies that are definable in linear arithmetic, with SAFE strategies additionally restricted to be finite memory, and with REACH strategies additionally restricted to be bounded. We define these terms precisely in Section 6. The remainder of this section establishes some basic results concerning reachability games that are independent of a specific strategy description language, with the aim of setting our expectations of what a strategy synthesis algorithm can be expected to do.

First, we observe that reachability games are **determined**: for any reachability game, either SAFE has a winning strategy or REACH does (this can be shown either by an elementary argument or a consequence of the celebrated Borel determinacy theorem [Martin 1975]). However, there is no algorithm for determining *which*:

Proposition 3.5. *There is no algorithm that, given a linear reachability game $\mathcal{G}$(init, reach, safe) as input, determines which player wins.*

Proof Sketch. The transition relation of a program P can be encoded into linear reachability game by encoding the initial state of the program in init and the transition relation in reach, and taking safe to be the identity relation (that is, $\bigwedge_i x_i = y_i$). The safety player has only one strategy for this game: the function $g(\mathbf{r}_1 ... \mathbf{r}_n) = \mathbf{r}_n$. It is easy to see that g is a winning strategy if and only if the program P is non-terminating. □

As a result of this proposition, any strategy synthesis procedure for reachability games must be either non-terminating or incomplete (the procedure fails to synthesize a winning strategy for some game even though one exists).

The undecidability result also implies that there is no *ideal* language for describing strategies. Suppose that we have a hypothetical strategy description language $\mathcal{L}$ that is decidable in the sense that there is procedure for determining whether any candidate strategy in the language is winning. Then even though reachability games are determined, they are *not* determined by strategies that are definable in $\mathcal{L}$ (i.e., there are games for which no player has an $\mathcal{L}$*-definable* winning strategy). $\mathcal{L}$-definable determinacy would contradict the undecidability result, since one could decide any game by enumerating all candidate strategies and testing each one to see if it wins. Under the reasonable expectation that the description language of a strategy synthesis procedure is decidable, there are games for which the procedure will fail, simply because there is no winning strategy within the description language. We discuss some limitations imposed by our particular choice of description language in Section 6.

4 STRATEGY SYNTHESIS FOR SATISFIABILITY GAMES

This section presents a complete strategy synthesis algorithm for linear satisfiability games. The algorithm is based on SimSat, our recent decision procedure for the theory of linear rational arithmetic [Farzan and Kincaid 2016]. We show how to adapt this algorithm to synthesize strategies by exploiting tree interpolation to extract winning strategies from the satisfiability/unsatisfiability evidence produced by the decision procedure.

SimSat proves that a formula φ is satisfiable or unsatisfiable by synthesizing a winning strategy *skeleton* for one of the two players in the satisfiability game for φ. A strategy skeleton can be thought of as kind of *nondeterministic* strategy. We recall the definition of strategy skeletons:

Definition 4.1 (Strategy Skeleton [Farzan and Kincaid 2016]). Let

$$\varphi = Q_1 x_1.Q_2 x_2.\cdots Q_n x_n.F$$

be a prenex LRA sentence. A SAT *strategy skeleton* for φ is a finite, non-empty set $S \subseteq (\mathsf{Term} \cup \{\bullet\})^n$ of sequences over terms plus a distinguished placeholder $\bullet$, where each sequence $\pi_1 \cdots \pi_n \in S$ has length n and such that for all $i \in \{1, \ldots, n\}$,

- if Q_i is $\exists$, then π_i is a term and $\mathsf{fv}(\pi_i) \subseteq \{x_1, \ldots, x_{i-1}\}$
- if Q_i is $\forall$, then π_i is $\bullet$

An UNSAT strategy skeleton for φ is defined to be a SAT strategy skeleton for the dual game $\neg\varphi$.

A strategy skeleton may be viewed as a forest, if common prefixes of different sequences are unified to form trees. The set of sequences in the skeleton are then exactly the complete paths through such a forest. In the following, we will typically visualize skeletons as such, and use the vocabulary of forests and trees to describe features of skeletons (e.g., a skeleton "branches" at the maximal common prefix of two or more sequences).

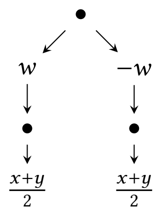
Example 4.2. Consider the following formula:

$$\varphi \triangleq \forall w.\exists x.\forall y.\exists z.w \leq x \wedge 0 \leq x \wedge (y < -w \vee y < w \vee (x < z \wedge z < y))$$

A possible SAT strategy skeleton for φ is:

$$\{\bullet w \bullet \frac{x+y}{2}, \bullet(-w)\bullet\frac{x+y}{2}\} ,$$

which is visualized as the tree to the right. This tree indicates the moves available to a SAT player who plays according to this skeleton: on turn 1, the UNSAT player may choose any value for w (represented by the placeholder $\bullet$). On turn 2, the SAT player may choose between w and $-w$ (where w is the value that the UNSAT player chose on turn 1). Turn 3 again belongs to the UNSAT player. On turn 4, the SAT player must choose $(x + y)/2$, after which the game is finished.



The crucial difference between a strategy skeleton and a strategy is that a skeleton defines only the moves that a player may take, not when to take them (e.g, the skeleton pictured in Example 4.2

indicates that the first move of the SAT player must be either w or $-w$, but does not specify whether w or $-w$ is the appropriate response to a given move of the UNSAT player). A single strategy skeleton corresponds to a set of strategies that *conform* to the skeleton, formalized below.

Definition 4.3. Let $\varphi = Q_1x_1.Q_2x_2.\cdots Q_nx_n.F$ be a prenex LRA sentence, and let S be a strategy skeleton for the SAT player on φ. We say that a play ρ of φ *conforms* to S if there exists some $\pi_1\cdots\pi_n \in S$ such that for all $i \in \{1, \dots, n\}$ such that Q_i is $\exists$, we have $[\![x_i]\!]^\rho = [\![\pi_i]\!]^\rho$. We say that a SAT strategy f for φ conforms to S if every play that conforms to f also conforms to S. Conformance for UNSAT skeletons is defined similarly. It is important to note that when a *play* conforms to a skeleton, it conforms to one of its paths, however, when a *strategy* conforms to a skeleton, it does not necessarily conform to a single path.

Example 4.4. Consider the SAT strategy skeleton from Example:4.2. One strategy that conforms to the skeleton is the function f_1:

$$f_1(w) \triangleq \textbf{if } w < 0 \textbf{ then } w \textbf{ else } -w$$

$$f_1(wxy) \triangleq \frac{x+y}{2}$$

Another conforming strategy is the function f_2:

$$f_2(w) \triangleq w$$

$$f_2(wxy) \triangleq \frac{x+y}{2}$$

Note that f_1 is a winning strategy and f_2 is not.

A critical property of strategy skeletons is that they are naturally ordered: one skeleton is “better” than another if it allows more moves (or equivalently, more strategies conform to it). In contrast, there is no obvious way of saying that one *strategy* is better than another: either a strategy is winning or it is not. SimSat makes essential use of this order. It operates by iteratively improving strategy skeletons (ascending in the order) for both the SAT and UNSAT players until one of the players finds a **winning strategy skeleton** (i.e., a skeleton to which some winning strategy conforms).

In light of this, we may view SimSat as a complete algorithm for synthesizing winning strategy *skeletons* for linear satisfiability games. While SimSat guarantees the existence of a winning strategy that conforms to the synthesized skeleton, it does not need to construct a strategy to provide a yes/no answer to a satisfiability query. The algorithmic problem we must address is thus *how can a winning strategy be extracted from a winning strategy skeleton?* Comparing the (winning) strategy skeleton in Example 4.2 and the (conforming) winning strategy f_1 in Example 4.4, the essential problem is to synthesize branching conditions to replace the nondeterministic branching in the skeleton with deterministic conditionals.

Our strategy extraction procedure is based on *tree interpolation*, a variation of Craig interpolation that has been previously used to synthesize procedure summaries for interprocedural program verification [McMillan and Rybalchenko 2013]. We recall the definition below.

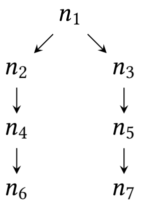
Definition 4.5 (Tree interpolant [McMillan and Rybalchenko 2013]). A labeled tree $T = \langle N, E, r, \Phi\rangle$ consists of a directed tree $\langle N, E, r\rangle$ with root r and an annotation $\Phi : N \to$ Formula. An *interpolant* for T is a function $I : N \to$ Formula such that

(1) $I(r) = \mathit{false}$
(2) For all $n \in N$, $\Phi(n) \land \bigwedge_{(n,c)\in E} I(c) \models I(n)$
(3) For all $n \in N$, $I(n)$ is expressed over the variables common to its descendants *and* non-descendants. That is, for each variable $x \in \text{fv}(I(n))$, there is a descendant d of n and a

non-descendant e such that x appears in both $\Phi(d)$ and $\Phi(e)$.

An interpolant for a labeled tree $T = \langle N, E, r, \Phi \rangle$ always exists as long as the conjunction of all T's annotations ($\bigwedge_{n \in N} \Phi(n)$) is unsatisfiable. The approach we take to strategy synthesis is to construct from a winning skeleton a labeled tree that satisfies this property, and such that conditional guards can be read off an interpolant for the tree. We start with an example, and proceed to the formal definition subsequently.

Example 4.6. Consider again the SAT strategy from Example 4.2. From this skeleton, we construct the structurally isomorphic tree pictured below to the left. Each internal node n_i ($i = 1, 2, 3, 4, 5$) is annotated with assertion *true* ($\Phi(n_i) = \mathit{true}$). Each leaf is annotated with a formula whose models correspond to plays of the game φ, where the SAT player plays according to the path from the root to the leaf and *loses.*



$$\Phi(n_6) = \neg\varphi[z \mapsto \frac{x+y}{2}][y \mapsto \underline{y_1}][x \mapsto w][w \mapsto \underline{w}]$$

$$= \neg\left(\begin{array}{l} \underline{w} \leq \underline{w} \wedge 0 \leq \underline{w} \\ \wedge \left(\begin{array}{l} \underline{y_1} < -\underline{w} \vee \underline{y_1} < \underline{w} \\ \vee \left(\underline{w} < \frac{\underline{w}+\underline{y_1}}{2} \wedge \frac{\underline{w}+\underline{y_1}}{2} < \underline{y_1}\right) \end{array}\right) \end{array}\right)$$

$$\Phi(n_7) = \neg\varphi[z \mapsto \frac{x+y}{2}][y \mapsto \underline{y_2}][x \mapsto -w][w \mapsto \underline{w}]$$

$$= \neg\left(\begin{array}{l} \underline{w} \leq -\underline{w} \wedge 0 \leq -\underline{w} \\ \wedge \left(\begin{array}{l} \underline{y_2} < -\underline{w} \vee \underline{y_2} < \underline{w} \\ \vee \left(-\underline{w} < \frac{\underline{y_2}-\underline{w}}{2} \wedge \frac{\underline{y_2}-\underline{w}}{2} < \underline{y_2}\right) \end{array}\right) \end{array}\right)$$

The formulas $\Phi(n_6)$ and $\Phi(n_7)$ are obtained from the negation of the matrix of the formula φ by substituting the terms appearing on the associated path for the existentially quantified variables and substituting fresh variables for the universally quantified variables (shown underlined). The naming of the fresh variables reflects the structure of the tree: there is one copy of the variable w (corresponding to the one node n_1 at level 1) and two copies of the variable y (corresponding to the two nodes n_4, n_5 at level 3).

Each model of the formula $\Phi(n_6) \wedge \Phi(n_7)$ corresponds to strategy for the UNSAT player to beat any SAT strategy that conforms to the skeleton: given such a model M, the UNSAT player's strategy is as follows: for the first turn, choose $\llbracket \underline{w} \rrbracket^M$ for w. A SAT player that conforms to the given skeleton must respond by choosing $\llbracket \underline{w} \rrbracket^M$ or $-\llbracket \underline{w} \rrbracket^M$ for x. If the SAT player chooses $\llbracket \underline{w} \rrbracket^M$, then the UNSAT player responds with $\llbracket \underline{y_1} \rrbracket^M$; otherwise, with $\llbracket \underline{y_2} \rrbracket^M$. However, the SAT skeleton is winning, so no such UNSAT strategy can exist: $\Phi(n_6) \wedge \Phi(n_7)$ must be unsatisfiable, and therefore there exists an interpolant I for the tree. We construct a winning strategy f for SAT as follows:

$$f(w) \triangleq \textbf{if } \neg(I(n_2))[\underline{w} \mapsto w] \textbf{ then } w \textbf{ else } -w$$

$$f(wxy) \triangleq \frac{x+y}{2} .$$

First we argue that f is a well-defined, in the sense that the variables that appear on the right hand side of an equation are bound on the left. From the property of interpolants we have that $I(n_2)$ must be defined over the symbols common to $\Phi(n_6)$ and $\Phi(n_7)$, which is only $\underline{w}$. Thus, the only variable appearing in $I(n_2)$ is w, which is a bound variable.

Next we argue that f is winning. From property 2 of Definition 4.5, we have

$$\Phi(n_6) \models I(n_6) \models I(n_4) \models I(n_2) .$$

Taking the converse, we have $\neg I(n_2) \models \neg\Phi(n_6)$. Recalling that $\Phi(n_6)$ is a formula describing the plays of the game φ where the SAT player plays according to the left path and *loses*, its negation describes plays where the SAT player *wins*. So f is a winning strategy as long as the UNSAT player's first move satisfies the condition G. Last, we must show that f wins the plays where the UNSAT player's first move does not satisfy G. From property 2 of Definition 4.5, we have

$$I(n_2) \land I(n_3) \models I(n_1)$$

From property 1, we have $I(n_1) = \mathit{false}$. It follows that $I(n_2) \models \neg I(n_3)$. Again using property 2, we have

$$\Phi(n_7) \models I(n_7) \models I(n_5) \models I(n_3)$$

and taking the converse we have $\neg I(n_3) \models \neg\Phi(n_7)$. Combining the two previous, we have $I(n_2) \models \neg\Phi(n_7)$. Thus, if the UNSAT player's first move does not satisfy G, it must satisfy $\Phi(n_7)$, meaning that the right path is winning.

Let $\varphi = Q_1 x_1 . Q_2 x_2 . \cdots Q_n x_n . F$ be a prenex LRA sentence, and suppose that S is a *winning* strategy skeleton for the SAT player on φ (the case that the φ is unsatisfiable can be handled symmetrically). We define a labeled tree $T(S, \varphi)$ as follows.

With any sequence $\pi \in (\text{Term} \cup \{\bullet\})^*$ we associate a unique variable symbol that we denote by $\underline{x}(\pi)$. For any path $\pi \in S$, we associate a *losing formula* $\text{lose-path}(x_1 \cdots x_n, \pi, \neg F)$ describing the plays where the SAT player conforms to π and loses:

$$\begin{aligned}
\text{lose-path}(\mathrm{y}, \epsilon, G) &\triangleq G \\
\text{lose-path}(\mathrm{y}y, \pi'\bullet, G) &\triangleq \text{lose-path}(\mathrm{y}, \pi', G[y \mapsto \underline{x}(\pi')]) \\
\text{lose-path}(\mathrm{y}y, \pi' t, G) &\triangleq \text{lose-path}(\mathrm{y}, \pi', G[y \mapsto t])
\end{aligned}$$

Finally, the tree $T(S, \varphi) \triangleq \langle N, E, r, \Phi\rangle$ is defined as:

$$\begin{aligned}
N &\triangleq \{\pi \in (\text{Term} \cup \{\bullet\})^* : Q_{|\pi|} = \exists \text{ and } \exists \pi'. \pi\pi' \in S\} \\
E &\triangleq \{\langle \pi, \pi\bullet\rangle : \pi, \pi\bullet \in N\} \cup \{\langle \pi, \pi t\rangle : \pi, \pi t \in N\} \\
r &\triangleq \epsilon \\
\Phi(\pi) &\triangleq \begin{cases} \text{lose-path}(x_1 \cdots x_n, \pi, \neg F) & \text{if } \pi \in S \\ \mathit{true} & \text{otherwise} \end{cases}
\end{aligned}$$

Suppose that I is an interpolant for $T(S, \varphi)$. For any $\pi \in N$, we use $G(\pi)$ to denote $\neg I(\pi)[\sigma]$, where σ is a substitution mapping each $\underline{x}(\pi_1 \cdots \pi_n)$ to x_{n+1}. We define a strategy $\text{extract}(S, I, \varphi)$ to be the function that maps each $\rho \in \mathbb{Q}^*$ to $f(\epsilon, \epsilon, \rho)$, where

$$\begin{aligned}
&f(\pi t, \rho, \epsilon) \triangleq [\![t]\!]^\rho \\
&f(\pi\bullet, \rho, \epsilon) \text{ is undefined} \\
&f(\pi, \rho, a\rho') \triangleq \begin{cases} f(\pi\bullet, \rho a, \rho') & \text{if } Q_{|\pi|+1} = \forall \\ f(\pi t, \rho a, \rho') & \text{if } Q_{|\pi|+1} = \exists, t = \text{select}(\pi, \rho a) \end{cases}
\end{aligned}$$

and where $\text{select}(\pi, \rho a)$ is defined to be the unique t such that $\pi t \in N$ if there is only one such t, and some term t such that $\pi t \in N$ and $\rho a \models G(\pi t)$ if not (choosing some arbitrary syntactic condition to break ties should they occur).

Proposition 4.7. *Let $\varphi = Q_1 x_1 . Q_2 x_2 . \cdots Q_n x_n . F$ be a prenex LRA sentence, let S be a winning SAT strategy skeleton for φ. Let $T(S, \varphi)$ be constructed as above. Let I be any interpolant for $T(S, \varphi)$, and let $f = \text{extract}(S, I, \varphi)$. Then f is a winning strategy for the SAT player in the game φ.*

If the UNSAT player wins φ, then we can compute a winning strategy for UNSAT by following the same procedure using the dual game $\neg\varphi$.

Complexity The decision problem of determining whether a formula is satisfiable is NExptime-hard [Fischer and Rabin 1974]. The SimSat algorithm always returns a strategy skeleton whose size is at most $2^{2^{cn}}$, where n is the size of the input formula and c is a constant, matching the worst-case complexity of quantifier elimination [Ferrante and Rackoff 1975]. The complexity of our procedure for extracting a strategy from a strategy skeleton depends on the complexity of the underlying tree interpolation procedure; supposing that I is a function such that the size of a tree interpolant has size $O(I(t))$ (where t is the size of the input tree), the overall space complexity of our strategy synthesis procedure is $O(I(n2^{2^{cn}}))$.

5 STRATEGY SYNTHESIS FOR REACHABILITY GAMES

This section presents a procedure for synthesizing winning strategies for linear reachability games. The procedure is based on the intuition that a winning strategy for a reachability game can be synthesized from winning strategies for *bounded* variations of the game, which can be obtained through the technique introduced in Section 4.

We begin by defining *safety trees*, a data structure that our strategy synthesis procedure uses to represent both (1) a *partial* strategy for the safety player and (2) a *partial* proof that the strategy is winning. We then describe our strategy synthesis algorithm for reachability games, which operates by iteratively expanding a safety tree until it is *complete* (represents a winning strategy) or cannot be expanded further (the reachability player wins). Last, we describe the two important subroutines of this algorithm: *covering*, which detects situations in which the partial strategy can be made complete; and *expansion*, which unfolds the game one time step staring from a designated node in the tree, re-calculating previous moves if necessary, by finding a winning SAFE strategy for a bounded variation of the game.

A *complete* safety tree represents both a winning strategy for the safety player in a reachability game and an annotation that proves that the strategy is indeed winning. More generally, (not necessarily complete) safety trees represent *partial* strategies, which define moves of the safety player along *some* but not necessarily *all* possible sequences of positions.

Definition 5.1. Let $\mathcal{G}(\text{init}, \text{reach}, \text{safe})$ be a d-dimensional linear reachability game. A **safety tree** for $\mathcal{G}(\text{init}, \text{reach}, \text{safe})$ is a tuple $T = \langle N, E, r, X, \Phi, M, \triangleright \rangle$ where

- (N, E) is a tree with root $r \in E$
- $X \subseteq N$ is a set of ("expanded") nodes
- $\Phi : N \to \text{Formula}$ is an *annotation*, mapping every node to a formula in d free variables $\{x_1, ..., x_d\}$, that describes a set of game positions. Intuitively, the annotation serves the same role as an inductive invariant in program verification—it serves as the basis of a proof that the strategy represented by a safety tree is winning.
- $M : E \to \text{Formula} \times \text{Term}^d$ is a *move assignment*, mapping every node to a pair $\langle G, \mathbf{m} \rangle$ consisting of a *guard* $G \in \text{Formula}$ and a d-tuple of terms $\mathbf{m}$, both in d free variables $\{x_1, ..., x_d\}$. For any $(u, v) \in E$, we define the shorthand $\langle G_{(u,v)}, \mathbf{m}_{(u,v)} \rangle \triangleq M(u, v)$.
- $\triangleright \subseteq (N \setminus X) \times X$ is a *covering relation*. The covering relation is functional: if $u \triangleright v$ and $u \triangleright w$, we must have $v = w$.

We say that T is **well-labeled** if

(1) *Initiation*: $\text{init}(\mathbf{x}) \models \Phi(r)(\mathbf{x})$
(2) *Consecution*: For all $(u, v) \in E$,

$$\Phi(u)(\mathbf{x}) \wedge G(\mathbf{x}) \wedge \mathbf{y} = \mathbf{m}(\mathbf{x}) \wedge \text{reach}(\mathbf{y}, \mathbf{x}') \models \Phi(v)(\mathbf{x}')$$

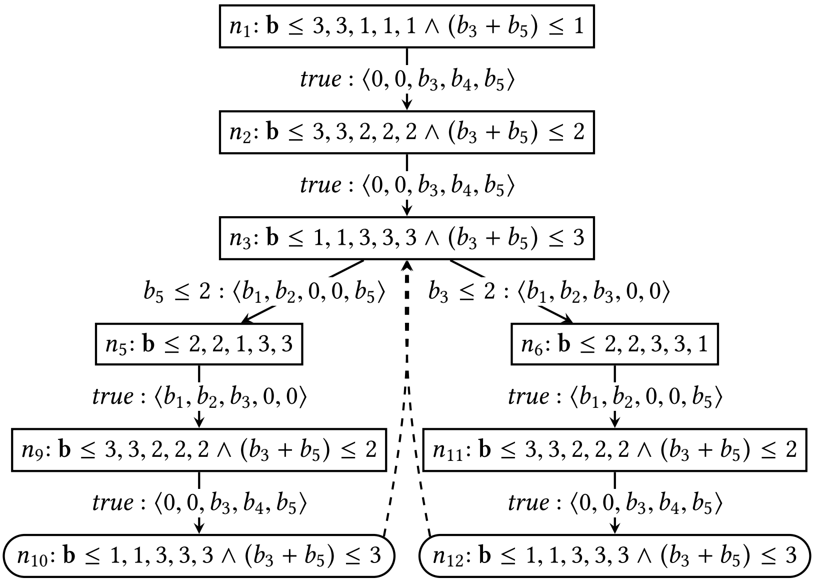

n_1: $\mathbf{b} \leq 3,3,1,1,1 \wedge (b_3 + b_5) \leq 1$
true : $\langle 0, 0, b_3, b_4, b_5 \rangle$
n_2: $\mathbf{b} \leq 3,3,2,2,2 \wedge (b_3 + b_5) \leq 2$
true : $\langle 0, 0, b_3, b_4, b_5 \rangle$
n_3: $\mathbf{b} \leq 1,1,3,3,3 \wedge (b_3 + b_5) \leq 3$
$b_5 \leq 2 : \langle b_1, b_2, 0, 0, b_5 \rangle$
$b_3 \leq 2 : \langle b_1, b_2, b_3, 0, 0 \rangle$
n_5: $\mathbf{b} \leq 2,2,1,3,3$
n_6: $\mathbf{b} \leq 2,2,3,3,1$
true : $\langle b_1, b_2, b_3, 0, 0 \rangle$
true : $\langle b_1, b_2, 0, 0, b_5 \rangle$
n_9: $\mathbf{b} \leq 3,3,2,2,2 \wedge (b_3 + b_5) \leq 2$
n_{11}: $\mathbf{b} \leq 3,3,2,2,2 \wedge (b_3 + b_5) \leq 2$
true : $\langle 0, 0, b_3, b_4, b_5 \rangle$
true : $\langle 0, 0, b_3, b_4, b_5 \rangle$
n_{10}: $\mathbf{b} \leq 1,1,3,3,3 \wedge (b_3 + b_5) \leq 3$
n_{12}: $\mathbf{b} \leq 1,1,3,3,3 \wedge (b_3 + b_5) \leq 3$

Fig. 4. A complete, well-labeled safety tree for the Cinderella-Stepmother game with bucket capacity 3.

where $M(u, v) = \langle G, \mathbf{m} \rangle$.

(3) *Covering*: For all $u \triangleright v$, $\Phi(u)(\mathbf{x}) \models \Phi(v)(\mathbf{x})$.

(4) *Availability*: For all $(u, v) \in E$, $M(u, v) = \langle G, \mathbf{m} \rangle$ is a *legal move* in the sense that

$$G(\mathbf{x}) \wedge \mathbf{y} = \mathbf{m}(\mathbf{x}) \models \mathsf{safe}(\mathbf{x}, \mathbf{y}) \ .$$

(5) *Adequacy*: For all expanded nodes $u \in X$, we have

$$\Phi(u)(\mathbf{x}) \models \bigvee_{(u,v) \in E} G_{(u,v)}(\mathbf{x})$$

We say that T is **complete** if every node is either expanded or covered ($N = X \cup \{u : \exists v. u \triangleright v\}$).

Example 5.2. Consider the Cinderella-Stepmother game from Section 2.2. Figure 4 depicts a complete, well-labeled safety tree for this game. Each node is labeled with an identifier (n_1, n_2, n_3, ...) and its annotation, and each tree edge is labeled by its move assignment. Expanded nodes are drawn as rectangles, and un-expanded nodes as rounded rectangles. The covering relation is shown as a dashed edge.

A well-labeled safety tree defines a partial strategy for the safety player, while a complete and well-labeled safety tree defines a winning (total) strategy for the safety player. Intuitively, to determine the move dictated by safety tree in response to a sequence of positions, we traverse the sequence, matching each position to an edge of the tree, such that the position satisfies the guard of that edge—the move selected to respond to the sequence is the move of the the last matched edge. Safety trees have finite height, but the covering relation allows the safety player to respond to sequences of unbounded length by jumping from u to v when $u \triangleright v$—this effectively creates cycles in the tree where the back-edges belong to the covering relation.

We now formalize the association between partial strategies and well-labeled safety trees. Let $T = \langle N, E, r, X, \Phi, M, \triangleright \rangle$ be a well-labeled safety tree for a d-dimensional linear reachability game. For any expanded node $u \in X$, let $v_1, ..., v_n$ be a list of u's children, and define its *successor function* $succ : \mathbb{Q}^d \to N$ as follows:

$$succ^u_T(\mathbf{r}) \triangleq \begin{cases} v_1 & \text{if } G_{(u,v_1)}(\mathbf{r}) \\ v_2 & \text{if } G_{(u,v_2)}(\mathbf{r}) \\ & \vdots \\ v_n & \text{otherwise} \end{cases}$$

The successor function simply navigates through the tree by choosing the first edge with a satisfied guard. For the expanded node $u \in X$ we define its partial strategy $g^u_T : (\mathbb{Q}^d)^+ \rightharpoonup \mathbb{Q}^d$ as follows:

$$g^u_T(\mathbf{r}) \triangleq \mathbf{m}_{(u,v)} \qquad \text{where } v = succ^u_T(\mathbf{r})$$

$$g^u_T(\mathbf{r}_1 \cdots \mathbf{r}_m) \triangleq g^v_T(\mathbf{r}_2 \cdots \mathbf{r}_m) \qquad \text{where } v = succ^u_T(\mathbf{r}_1)$$

The partial strategy of a covered node $u \triangleright v$, is defined to be the strategy of the node covering it $g^u_T \triangleq g^v_T$. The partial strategy of an un-expanded, un-covered node is undefined on all inputs. Finally, the partial strategy associated with the safety tree is the partial strategy of the root, $g_T \triangleq g^r_T$.

Suppose that $u \in X$ is an expanded node. Observe that, due to the *Availability* and *Adequacy* conditions, from any position $\mathbf{r}$ that satisfies $\Phi(u)$, $g^u_T(\mathbf{r})$ is a legal move for the safety player. Due to the *Consecution* condition, if $\mathbf{r}_1$ satisfies $\Phi(u)$ and the reachability player may legally move from $g^u_T(\mathbf{r}_1)$ to $\mathbf{r}_2$, then $r_2 \models \Phi(next^u_T(\mathbf{r}_1))$. Due to the *Covering* condition, the same holds if u is a covered node. Combining these facts, we have the following lemma:

LEMMA 5.3. *Let $T = \langle N, E, r, X, \Phi, M, \triangleright \rangle$ be a well-labeled safety tree, let $u \in N$, and let $\pi = \mathbf{r}_1\mathbf{s}_1 \cdots \mathbf{r}_n\mathbf{s}_n$ be a partial play that starts in a position $\mathbf{r}_1$ that satisfies the annotation $\Phi(u)$ of u, and that conforms to the partial strategy of u (i.e., for all i, $g^u_T(\mathbf{r}_1 \cdots \mathbf{r}_{i-1})$ is defined and equal to $\mathbf{s}_i$). Then SAFE does not lose π, in the sense that either (1) all of SAFE's moves are legal, or (2) REACH makes an illegal move before any illegal move of SAFE.*

Thus, we may view the annotation $\Phi(u)$ of a node u as defining a *safe region*: a set of positions starting from which the partial strategy g^u_T does not lose. From the perspective of program verification, one might view a safety tree as a labeled unwinding of a program [McMillan 2006]. The *Initiation*, *Consecution*, and *Covering* conditions ensure that Φ is an inductive annotation for the graph formed by collapsing every pair of nodes u and v such that u covers v. The fundamental difference is that a safety tree is not an unwinding of some fixed program: the "program" is the safety strategy, which must be synthesized. In fact, checking that an assertion of a program succeeds can be encoded as a reachability game in which the safety player does nothing and the reachability player simulates the program of interest. In this case, the safety strategy is trivial and our strategy synthesis algorithm simulates lazy abstraction with interpolants [McMillan 2006].

The following proposition states the soundness of safety trees as a system for proving that the safety player wins a reachability game. It follows easily from Lemma 5.3.

PROPOSITION 5.4. *Let $\mathcal{G}(init, reach, safe)$ be a linear reachability game. If there exists a well-labeled, complete safety tree for $\mathcal{G}(init, reach, safe)$, then the safety player wins $\mathcal{G}(init, reach, safe)$.*

Our strategy synthesis procedure for reachability games, described in Algorithm 1, aims to find either (1) a *complete, well-labeled* safety tree representing a winning strategy for the safety player or (2) a winning reachability strategy for some bounded version of the game (cf. Proposition 3.4). At a high-level, the algorithm accomplishes this task by maintaining a well-labeled safety tree and repeatedly *covering* or *expanding* nodes until the tree is complete or expansion fails due to a winning strategy for the reachability player being discovered.

The algorithm begins by initializing a trivial safety tree (lines 2-5). At each iteration of the main loop (lines 6-14), we choose a vertex v that is neither expanded nor covered and either cover it (line

```
1  Procedure strategy-synthesis(init, reach, safe, d)
2      $r \leftarrow$ fresh vertex
3      $N \leftarrow \{r\}$
4      $E \leftarrow \emptyset, X \leftarrow \emptyset, \triangleright \leftarrow \emptyset$
5      $\Phi(r) \leftarrow true$
6      while $T$ is not complete do
7          Pick a vertex $v$ that is neither expanded nor covered
8          if force-cover($v$) then
9              continue
10         switch expand($v$, 1) do
11             case Fail: $f$
12                 return Reachability strategy $f$
13             case Success
14                 continue
15     return Safety strategy $T$
```

Algorithm 1: Strategy synthesis for reachability games .

8) or expand it (line 10). The algorithm *covers* v when it detects that the strategy used to reach v forces play into a region that appears elsewhere in the tree. It is the key generalization step that turns a strategy for a bounded game into a strategy for an unbounded game. If no covering exists, the algorithm *expands* v by synthesizing a strategy for the safety player to continue the game starting from the region that the game is in after following the strategy leading to v. Synthesizing such a strategy may be impossible due to a poor move in the strategy leading to v, and so expansion may require the algorithm to back-track and re-synthesize moves earlier in the game. In sections Section 5.1 and Section 5.2, we will describe the covering and expansion sub-procedures in more detail, using the Cinderella-Stepmother game from Section 2.2 as a running example. Note that in the expansion and covering algorithms *the variables* N, r, E, Φ *are treated as globals, as are the parameters init, reach, safe, and d.*

Before we get into algorithmic details, we state two important high-level properties of the algorithm; namely, the algorithm is sound, and it is complete for bounded reachability strategies:

Proposition 5.5 (Soundness). *If Algorithm 1 synthesizes a reachability strategy for a game* $\mathcal{G}(init, reach, safe)$*, then that strategy is winning; if it synthesizes a safety strategy, that strategy is winning.*

Proposition 5.6 (Bounded completeness). *If the reachability player wins* $\mathcal{G}_n(init, reach, safe)$ *for some n, then (assuming a fair expansion policy, in which any vertex that is added to the safety tree is eventually either expanded or covered and remains covered)*[1] *Algorithm 1 terminates with a winning reachability strategy for* $\mathcal{G}(init, reach, safe)$*.*

5.1 Covering

Lemma 5.3 shows that for each vertex u in a safety tree, the annotation $\Phi(u)$ defines a safe region for u: a set of positions for which the partial strategy g_T^u does not lose. If there is a vertex v such that $\Phi(u) \models \Phi(v)$, then the partial strategy of the tree rooted at v is at least as good as one at u. A particular case of interest is when v is an ancestor of u, in which case the path from v to u is a strategy for remaining in a safe region indefinitely. In this sense, covering is the mechanism by which our algorithm detects that a winning SAFE strategy for some bounded variation of a

[1] This policy can be implemented by selecting vertices of the safety tree in breadth-first manner, and allowing `force-cover`(v) to succeed at most once for any given v.

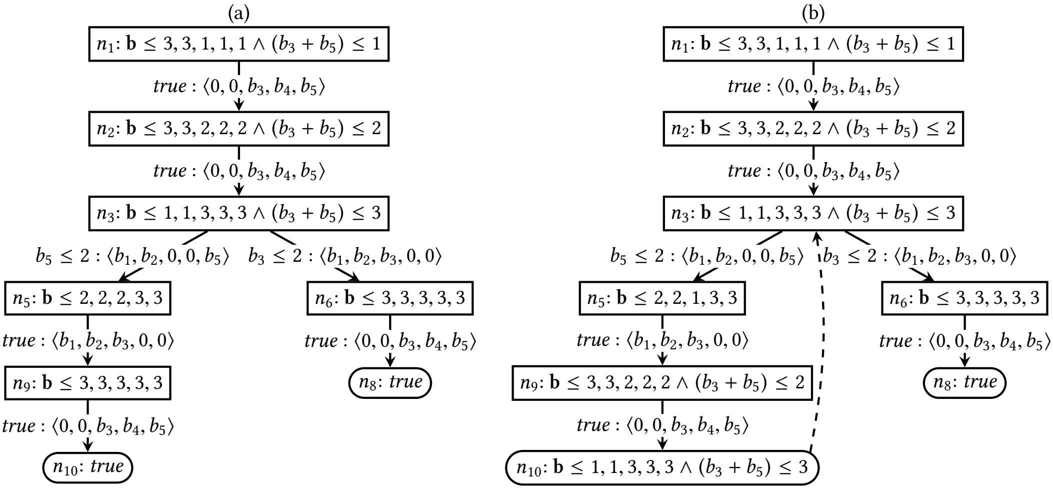

(a)
n_1: $\mathbf{b} \leq 3,3,1,1,1 \wedge (b_3 + b_5) \leq 1$
$true : \langle 0,0,b_3,b_4,b_5 \rangle$
n_2: $\mathbf{b} \leq 3,3,2,2,2 \wedge (b_3 + b_5) \leq 2$
$true : \langle 0,0,b_3,b_4,b_5 \rangle$
n_3: $\mathbf{b} \leq 1,1,3,3,3 \wedge (b_3 + b_5) \leq 3$
$b_5 \leq 2 : \langle b_1,b_2,0,0,b_5 \rangle$
$b_3 \leq 2 : \langle b_1,b_2,b_3,0,0 \rangle$
n_5: $\mathbf{b} \leq 2,2,2,3,3$
n_6: $\mathbf{b} \leq 3,3,3,3,3$
$true : \langle b_1,b_2,b_3,0,0 \rangle$
$true : \langle 0,0,b_3,b_4,b_5 \rangle$
n_9: $\mathbf{b} \leq 3,3,3,3,3$
n_8: $true$
$true : \langle 0,0,b_3,b_4,b_5 \rangle$
n_{10}: $true$
(b)
n_1: $\mathbf{b} \leq 3,3,1,1,1 \wedge (b_3 + b_5) \leq 1$
$true : \langle 0,0,b_3,b_4,b_5 \rangle$
n_2: $\mathbf{b} \leq 3,3,2,2,2 \wedge (b_3 + b_5) \leq 2$
$true : \langle 0,0,b_3,b_4,b_5 \rangle$
n_3: $\mathbf{b} \leq 1,1,3,3,3 \wedge (b_3 + b_5) \leq 3$
$b_5 \leq 2 : \langle b_1,b_2,0,0,b_5 \rangle$
$b_3 \leq 2 : \langle b_1,b_2,b_3,0,0 \rangle$
n_5: $\mathbf{b} \leq 2,2,1,3,3$
n_6: $\mathbf{b} \leq 3,3,3,3,3$
$true : \langle b_1,b_2,b_3,0,0 \rangle$
$true : \langle 0,0,b_3,b_4,b_5 \rangle$
n_9: $\mathbf{b} \leq 3,3,2,2,2 \wedge (b_3 + b_5) \leq 2$
n_8: $true$
$true : \langle 0,0,b_3,b_4,b_5 \rangle$
n_{10}: $\mathbf{b} \leq 1,1,3,3,3 \wedge (b_3 + b_5) \leq 3$

Fig. 5. Forced covering.

reachability game is also a winning strategy for the unbounded game.

The annotation of a vertex rarely entails another by coincidence. Following [McMillan 2006], we use a *forced covering* operation that attempts to *refine* the annotation of a given vertex v so that it entails the annotation of another u, and if successful, adds a covering edge $v \triangleright u$. The forced covering operation appears in Algorithm 2.

We illustrate forced covering by example. Figure 5 depicts the state of a strategy tree before and after a successful forced covering operation of the vertex n_{10} by n_3 (lines 17-25). Notice that in order to strengthen the annotation at n_{10} and maintain the *Consecution* condition, we must strengthen the annotation of every vertex on the path from the root n_1 to n_{10}. The stronger annotation is found by computing a sequence interpolant for the path formula corresponding to the path from n_1 to n_{10} (lines 19-20), which we formalize below.

Definition 5.7. Let $u_1 \cdots u_n$ be a path in a safety tree, and for each each i let $\langle G_i, \mathbf{m}_i \rangle \triangleq M(u_i, u_{i+1})$ be the move assigned to the edge (u_i, u_{i+1}). Define the *path formula* $\mathsf{pf}(u_1 \cdots u_n)$ of the path to be the sequence

$$\mathsf{pf}(u_1 \cdots u_n) \triangleq R_1 \cdot \ldots \cdot R_{n-1}$$

where each $R_i \triangleq G_i(\mathbf{x}_i) \wedge \mathsf{reach}(\mathbf{m}_i(\mathbf{x}_i), \mathbf{x}_{i+1})$ is a formula that represents one round of the game (the move of the safety player corresponding to the edge (u_i, u_{i+1}) plus any legal move of the reachability player).

Definition 5.8. A *sequence interpolant* for a sequence of formulas Γ of length m is a sequence of formulas $\mathbf{I}$ such that (1) each $\mathbf{I}_i$ is expressed over the common variables of $\Gamma_1, \ldots, \Gamma_i$ and $\Gamma_{i+1}, \ldots, \Gamma_m$, (2) $\Gamma_1 \models \mathbf{I}_1$, (3) for all $i > 1$, $\mathbf{I}_{i-1} \wedge \Gamma_i \models \mathbf{I}_i$, and (4) $\mathbf{I}_m = \mathit{false}$.

Let us return to our example of covering n_{10} by n_3. Let

$$\Gamma \triangleq \mathsf{init}(x_1) \wedge \mathsf{pf}(n_1 n_2 n_3 n_5 n_9 n_{10}) \wedge \neg\underbrace{(\mathbf{b}_7 \leq 1,1,3,3,3 \wedge (b_{73} + b_{75}) \leq 3)}_{\Phi(n_3)[\mathbf{x} \mapsto \mathbf{x}_7]}$$

(cf. line 19). Γ constrains the first 7 moves of the reachability player to be legal, assuming that the safety player conforms to the path $n_1 n_2 n_3 n_5 n_9 n_{10}$, and for the final position to belong to the safe region $\Phi(n_3)$ of n_3. Let $\mathbf{I}$ be a sequence interpolant for Γ. By Definition 5.8 property (1), each $\mathbf{I}_i$ is expressed only over the variables $\mathbf{x}_i$, and thus can be interpreted as a formula over a single

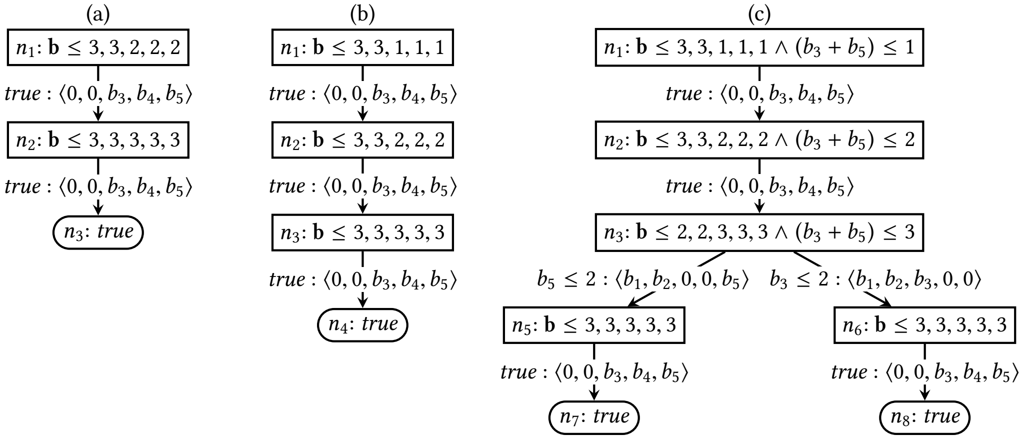

(a)
n_1: $\mathbf{b} \leq 3,3,2,2,2$
$true : \langle 0,0,b_3,b_4,b_5\rangle$
n_2: $\mathbf{b} \leq 3,3,3,3,3$
$true : \langle 0,0,b_3,b_4,b_5\rangle$
n_3: $true$
(b)
n_1: $\mathbf{b} \leq 3,3,1,1,1$
$true : \langle 0,0,b_3,b_4,b_5\rangle$
n_2: $\mathbf{b} \leq 3,3,2,2,2$
$true : \langle 0,0,b_3,b_4,b_5\rangle$
n_3: $\mathbf{b} \leq 3,3,3,3,3$
$true : \langle 0,0,b_3,b_4,b_5\rangle$
n_4: $true$
(c)
n_1: $\mathbf{b} \leq 3,3,1,1,1 \wedge (b_3 + b_5) \leq 1$
$true : \langle 0,0,b_3,b_4,b_5\rangle$
n_2: $\mathbf{b} \leq 3,3,2,2,2 \wedge (b_3 + b_5) \leq 2$
$true : \langle 0,0,b_3,b_4,b_5\rangle$
n_3: $\mathbf{b} \leq 2,2,3,3,3 \wedge (b_3 + b_5) \leq 3$
$b_5 \leq 2 : \langle b_1,b_2,0,0,b_5\rangle$
$b_3 \leq 2 : \langle b_1,b_2,b_3,0,0\rangle$
n_5: $\mathbf{b} \leq 3,3,3,3,3$
n_6: $\mathbf{b} \leq 3,3,3,3,3$
$true : \langle 0,0,b_3,b_4,b_5\rangle$
$true : \langle 0,0,b_3,b_4,b_5\rangle$
n_7: $true$
n_8: $true$

Fig. 6. Expansion of a safety tree for the Cinderella-Stepmother game with bucket capacity 3.

game position. The refine sub-procedure strengthens the annotation along the path by setting $\Phi(n_1) \leftarrow \Phi(n_1) \wedge \mathbf{I}_1[\mathbf{x}_1 \mapsto \mathbf{x}]$ through $\Phi(n_{10}) \leftarrow \Phi(n_{10}) \wedge \mathbf{I}_7[\mathbf{x}_7 \mapsto \mathbf{x}]$. The resulting tree is well-labeled: *Initiation* holds by (2) and *Consecution* by (3). Strengthening the annotation may violate the *Covering* condition, so the refine procedure must also remove violating pairs from the covering relation (lines 31-32); this does not occur in the running example. Last, the force cover procedure checks that the annotation of n_{10} entails the annotation of n_3 and adds the covering edge $n_{10} \triangleright n_3$ (lines 22-24).

```
16 Procedure force-cover($v$)
      /* Given a vertex $v \in N \setminus X$, try find a vertex $u$
         that so that $v$'s annotation can be
         strengthened so that $u$ covers $v$; return true if
         such a vertex is found and false otherwise     */
17    Let $r = u_1 ... u_n = v$ be the path from root to $v$
18    foreach $u \in X$ do
19        $\Gamma \leftarrow \mathsf{init}(\mathbf{x}_1) \cdot \mathsf{pf}(u_1 ... u_n) \cdot \neg(\Phi(u)[\mathbf{x} \mapsto \mathbf{x}_n])$
20        if $\Gamma$ has a sequence interpolant $\mathbf{I}$ then
21            refine($\mathbf{I}, v$)
22            if $\Phi(v) \models \Phi(u)$ then
23                $\triangleright \leftarrow \triangleright \cup (v, u)$
24                return true
25    return false

26 Procedure refine($\mathbf{I}, v$)
      /* Given a sequence of formulas $\mathbf{I}$ and a vertex $v$,
         strengthen the annotation of every node on
         the path to $v$ using $\mathbf{I}$                      */
27    Let $r = u_1 ... u_n = v$ be the path from root to $v$
28    for $i \leftarrow 1$ to $n$ do
29        $\Phi(u_i) \leftarrow \Phi(u_i) \wedge \mathbf{I}_i[\mathbf{x}_i \mapsto \mathbf{x}]$
30        foreach $u$ such that $u \triangleright u_i$ do
31            if $\Phi(u_i) \not\models \Phi(u)$ then
32                Remove $(u, u_i)$ from $\triangleright$
```

Algorithm 2: Forced covering

5.2 Expansion

Expansion is the mechanism by which the algorithm synthesizes new moves for the safety player by computing winning strategies for bounded variations of the reachability game. The expand procedure is given in Algorithm 3.

Figure 6 depicts two expansion steps, going from (a) to (b) by expanding n_3 and from (b) to (c) by expanding n_4. The node n_3 is expanded by finding a single safe move for Cinderella to make in the third round of the game, assuming that her first two moves are fixed to emptying the buckets b_1 and b_2. As explained in Section 2.2, attempting to expand n_4 similarly fails: Stepmother can overflow a bucket pouring 1L into bucket 3 on each round. As a result, the algorithm *backtracks* by deleting the node n_4 and expanding the node n_3 again *but with an increased depth of 2* (lines 37-43). We will explain the expand procedure by illustrating the operation of expand(n_3, 2) in more detail.

```
33 Procedure expand($(v, k)$)
       /* Given a vertex $v$ and a number $k$, expand $v$ by
          $k$ rounds (or some $\ell^{th}$ ancestor of $v$ at least
          $k + \ell$ rounds); if this is impossible return a
          winning reachability strategy. */
34     Let $r = u_0 \dots u_n = v$ be the path from $r$ to $v$
35     $F \leftarrow \left( \begin{array}{l} \exists \mathbf{x}_1 \dots \mathbf{x}_n . \\ \forall \mathbf{y}_n . \exists \mathbf{x}_{n+1} \forall \mathbf{y}_{n+1} \cdots \exists \mathbf{x}_{n+k} \forall \mathbf{y}_{n+k} \\ \left( \begin{array}{l} \mathsf{init}(\mathbf{x}_1) \\ \land \mathsf{pf}(u_1 \cdots u_n) \\ \land \mathsf{safe}(\mathbf{x}_n, \mathbf{y}_n) \end{array} \right) \Rightarrow \mathsf{unroll}(n, k-1) \end{array} \right)$
36     switch SimSat($\varphi$) do
37         case Winning SAT skeleton $S$
38             if $\exists (p, v) \in E$ then
                   /* Delete $v$ from $T$ */
39                 For all $u \triangleright v$, remove $(u, v)$ from $\triangleright$
40                 $E \leftarrow E \setminus \{(p, v)\}$
41                 $N \leftarrow N \setminus \{v\}$
                   /* Increase depth, expand parent */
42                 $d \leftarrow \max(k + 1, \mathsf{height}(v))$
43                 return expand(p,d)
44             else
                   /* $v = r$: REACH wins */
45                 $f \leftarrow$ REACH strategy extracted
                   from $S$
46                 return Fail: $f$
47         case Winning UNSAT skeleton $U$
48             $U \leftarrow \{\pi : \bullet^{dn} \pi \in U\}$
49             $\Gamma \leftarrow \mathsf{init}(\mathbf{x}_1) \cdot \mathsf{pf}(u_1 \dots u_n) \cdot \mathsf{lose}(U, n)$
50             $\mathbf{I} \leftarrow$ sequence interpolant for $\Gamma$
51             refine($\mathbf{I}, v$)
52             paste($U$,$v$,$n$)
53             return Success

54 Procedure paste($(U, v, n)$)
       /* Given an UNSAT strategy $U$, a vertex $v$, and
       number $n$ s.t. $\Phi(v)[\mathbf{x} \mapsto \mathbf{x}_n] \models \neg \mathsf{lose}(U, n)$, add
       $U$ as a subtree of $v$ */
55     $\{\mathbf{t}_1, \dots, \mathbf{t}_m\} \leftarrow \{\mathbf{t} \in \mathsf{Term}^d : \exists \pi . \mathbf{t} \pi \in U\}$
56     For each $i$, let $U_i = \{\mathbf{t}_i \pi : \mathbf{t}_i \pi \in U\}$
57     $I \leftarrow$ tree interpolant for
                   $0 : \Phi(v)[\mathbf{x} \mapsto \mathbf{x}_n]$
                      ↙        ...        ↘
           $1 : \mathsf{lose}(U_1, n)$          $m : \mathsf{lose}(U_m, n)$
58     for $i \leftarrow 1$ to $m$ do
59         $c_i \leftarrow$ fresh vertex
60         $N \leftarrow N \cup \{c_i\}, E \leftarrow E \cup \{(v, c_i)\}$
61         $G \leftarrow \neg \mathbf{I}(i)[\mathbf{x}_n \mapsto \mathbf{x}], \mathbf{m} \leftarrow \mathbf{t}_i[\mathbf{x}_n \mapsto \mathbf{x}]$
62         $M(v, c_i) \leftarrow \langle G, \mathbf{m} \rangle$
63         $U' \leftarrow \{\pi : \mathbf{t}_i \bullet^d \pi \in U\}$
64         if $U' \neq \emptyset$ then
65             csc $\leftarrow \Phi(v) \land G \land \mathsf{reach}(\mathbf{m}, \mathbf{x}_{n+1})$
66             annot $\leftarrow$ Interpolant for csc,
               $\mathsf{lose}(U', n + 1)$)
67             $\Phi(c_i) \leftarrow \mathsf{annot}[\mathbf{x}_{n+1} \mapsto \mathbf{x}]$
68             paste($U', c_i, n + 1$)
69         else
70             $\Phi(c_i) \leftarrow$ true
```

Algorithm 3: Node expansion

We expand n_3 (going from (b) to (c)) using an UNSAT strategy for a satisfiability game φ (line 35) defined as

$$\varphi : \begin{array}{l} \exists \mathbf{b}_1\mathbf{b}_2\mathbf{b}_3\forall \mathbf{b}'_3\exists \mathbf{b}_4\forall \mathbf{b}'_4. \\ \left(\begin{array}{l} \mathsf{init}(\mathbf{b}_1) \land \mathsf{pf}(n_1n_2n_3) \\ \land \left(\mathsf{safe}(\mathbf{b}_3, \mathbf{b}'_3) \Rightarrow \left(\begin{array}{l} \mathsf{reach}(\mathbf{b}'_3, \mathbf{b}_4) \\ \land \left(\mathsf{safe}(\mathbf{b}_4, \mathbf{b}'_4) \Rightarrow \mathit{false}\right) \end{array} \right)\right) \end{array} \right) \end{array}$$

The formula represents the bounded game where each player plays four moves, but Cinderella's first two moves are fixed (she empties b_1 and b_2 on both rounds). Using Farzan and Kincaid [2016]'s SimSat procedure, we get the following UNSAT strategy, which indicates that Cinderella can win the bounded game by emptying either b_3 and b_4 or b_4 and b_5 on round 3, and b_1 and b_2 on round 4:

$$U = \left\{ \begin{array}{l} \bullet^{15} b_{3,1}b_{3,2}00b_{3,5} \bullet^5\ 00b_{4,3}b_{4,4}b_{4,5}, \\ \bullet^{15} b_{3,1}b_{3,2}b_{3,3}00 \bullet^5\ 00b_{4,3}b_{4,4}b_{4,5} \end{array} \right\}$$

(In the above, $\bullet^n$ refers to a sequence of n $\bullet$s. In particular, the leading $\bullet^{15}$ corresponds to the leading quantifier prefix $\exists \mathbf{b}_1\mathbf{b}_2\mathbf{b}_3$). Similarly to the forced covering procedure, we use sequence interpolation to strengthen the annotation at n_1, n_2, and n_3 to ensure that U defines a winning strategy from any position that satisfies the annotation at n_3 (lines 49-51). The leading prefixes $\bullet^{15}$ (corresponding to the existential quantifier prefix $\exists \mathbf{b}_1\mathbf{b}_2\mathbf{b}_3$) are removed from U, and then paste is called to add U as a subtree to n_3.

Given a node v and an UNSAT strategy skeleton U for some bounded unrolling of a game, the paste procedure adds U as subtree to v while simultaneously synthesizing conditionals and annotations. Conditional synthesis is accomplished via tree interpolation, as in Section 4. On the running example, the paste procedure begins by computing the set of first moves permitted by U and partitions U by the first move:

$$\begin{array}{ll} \mathbf{t}_1 = b_{3,1}b_{3,2}00b_{3,5} & U_1 = \{b_{3,1}b_{3,2}00b_{3,5} \bullet^5\ 00b_{4,3}b_{4,4}b_{4,5}\} \\ \mathbf{t}_2 = b_{3,1}b_{3,2}b_{3,3}00 & U_2 = \{b_{3,1}b_{3,2}b_{3,3}00 \bullet^5\ 00b_{4,3}b_{4,4}b_{4,5}\} \end{array}$$

We find guards for each move by computing a tree interpolant for a tree with one branch for each move:

$$\begin{tikzcd} & 0 : \mathbf{b}_3 \leq 1,1,3,3,3 \land (b_3 + b_5) \leq 2 \arrow[dl] \arrow[dr] & \\ 1 : \mathsf{lose}(U_1, 3) & & 2 : \mathsf{lose}(U_2, 3) \end{tikzcd}$$

where

$$\mathsf{lose}(U_1,3) : \left(\begin{array}{l} \mathsf{safe}(\mathbf{b}_3, b_{3,1}b_{3,2}00b_{3,5}) \Rightarrow \\ \left(\begin{array}{l} \mathsf{reach}(b_{3,1}b_{3,2}00b_{3,5}, \underline{\mathbf{b}}_4^1) \\ \land \left(\mathsf{safe}(\underline{\mathbf{b}}_4^1, 00b_{4,3}b_{4,4}b_{4,5}) \Rightarrow \mathit{false}\right) \end{array} \right) \end{array} \right) \qquad \mathsf{lose}(U_2,3) : \left(\begin{array}{l} \mathsf{safe}(\mathbf{b}_3, b_{3,1}b_{3,2}b_{3,3}00) \Rightarrow \\ \left(\begin{array}{l} \mathsf{reach}(b_{3,1}b_{3,2}00b_{3,5}, \underline{\mathbf{b}}_4^2) \\ \land \left(\mathsf{safe}(\underline{\mathbf{b}}_4^2, 00b_{4,3}b_{4,4}b_{4,5}) \Rightarrow \mathit{false}\right) \end{array} \right) \end{array} \right)$$

The tree is unsatisfiable because starting from any position that satisfies the annotation at n_3, we know that either $\mathbf{t}_1$ or $\mathbf{t}_2$ is a winning move. Notice that the interpolants for the subtrees 1 and 2 may only be over the common variables $\mathbf{b}_3$: we interpret $I(1)$ and $I(2)$ as guards for their respective branches by negating and substituting $\mathbf{b}_3 \mapsto \mathbf{b}$. We then iterate over each move $\mathbf{t}_1, \mathbf{t}_2$ to create new children and recursively paste the appropriate sub-strategy of U. E.g., for the move $\mathbf{t}_1$, we create the child n_5 and paste the sub-strategy consisting of the single move in U_1 after $\mathbf{t}_1$: $U' = \{00b_{4,3}b_{4,4}b_{4,5}\}$. The annotation at n_5 is computed using binary interpolation between the consecution formula $\mathsf{csc} \triangleq \Phi(n_3) \land G_{(n_3,n_5)} \land \mathsf{reach}(\mathbf{m}_{(n_3,n_5)}, \mathbf{x}_{n+1})$ and the losing formula $\mathsf{lose}(U', n+1)$, ensuring that (1) the consecution condition holds and (2) U' is a winning strategy starting from any position satisfying $\Phi(n_5)$. Finally, we recursively paste U' to n_5, which has the effect of adding n_7 as a child of n_5. The other branch for $\mathbf{t}_2$ proceeds similarly, adding the nodes n_6 and n_8.

6 STRATEGY DESCRIPTION LANGUAGES AND THEIR IMPLICATIONS

A strategy synthesis procedure must inevitably compute a finite description of a strategy within some suitable description language. For satisfiability games, the strategies we consider are those definable in linear rational arithmetic (in a sense we will define precisely below). For reachability games, a safety strategy is described by a safety tree, while a reachability strategy is described by a definable function in linear rational arithmetic. The description language suggests a refinement of the question of determinacy: rather than ask if, for every game, one of the players has a winning strategy, we can ask if, for every game, one of the players has a winning strategy that can be defined within the description language. This section addresses this refined determinacy question and discusses the resulting limitations imposed on our strategy synthesis algorithms.

For linear arithmetic games, a natural requirement is that strategies should be definable within linear arithmetic. We recall the notion of definability (specialized to linear rational arithmetic) below.

Definition 6.1 (Definability, [Marker 2000]). Let $S \subseteq \mathbb{Q}^n$ be a set of n-tuples of rationals. We say that S is *definable* (in linear rational arithmetic) if there exists a LRA formula $\varphi_S(x_1, ..., x_n)$ such that for all $a_1, ..., a_n \in \mathbb{Q}$, $\langle a_1, ..., a_n \rangle \in S$ if and only if $\varphi_S(a_1, ..., a_n)$ holds.

A function $f : \mathbb{Q}^n \to \mathbb{Q}$ is definable if its graph

$$\{(a_1, ..., a_n, f(a_1, ..., a_n)) : a_1, ..., a_n \in \mathbb{Q}\}$$

is definable. Equivalently, f is definable if it is a piece-wise linear function, with each cell in the partition of $\mathbb{Q}^n$ defined by a linear arithmetic formula.

We extend the notion of definability to strategies for satisfiability games as follows. Let $\varphi = Q_1 x_1.Q_2 x_2.\cdots Q_n x_n.F$ be a formula, and $k_1, ..., k_m$ be the sequence of existential positions in φ (i.e., $\{k_1, ..., k_m\} = \{i : Q_i = \exists\}$). A SAT strategy

$$f : \{\rho \in \mathbb{Q}^* : |\rho| < n \land Q_{|\rho|+1} = \exists\} \to \mathbb{Q}$$

for φ can be identified with m functions $f_1, ...f_m$ such that each f_i is a total function $f_i : \mathbb{Q}^{k_i - 1} \to \mathbb{Q}$ and $f_i(a_1, ..., a_{k_i-1}) = f(a_1 \cdots a_{k_i-1})$ for all $a_1, ..., a_{k_i-1} \in \mathbb{Q}$. We say that f is definable if each f_i is definable. Definable UNSAT strategies are defined similarly.

The algorithm presented in Section 4 is a constructive proof that satisfiability games are definably determined: for any game, either SAT has a definable winning strategy or UNSAT does. However, our strategy synthesis algorithm for reachability games does not enjoy this property, as we will demonstrate in the following.

First, we give a characterization of the SAFE and REACH strategies that can be synthesized by our algorithm. Safety trees describe *definable finite memory* strategies, where the finite memory comes from the control structure provided by the tree. More precisely, we say that f is a **definable finite memory strategy** if there is a directed graph $G = \langle V, E \rangle$, a vertex $v \in R$, a function $\Phi : E \to \text{Formula}$, and a function $\mathbf{m} : V \to \text{Term}^d$ such that: (1) for any $\mathbf{r}_1 \cdots \mathbf{r}_n \in (\mathbb{Q}^d)^*$, there is a unique path $r = v_1 \cdots v_n$ emanating from r such that $\mathbf{r}_i \models \Phi(v_i, v_{i+1})$ for all i, and (2) $f(\mathbf{r}_1 ... \mathbf{r}_n) = \mathbf{m}(v_n)(\mathbf{r}_n)$. Note that not every definable finite memory strategy can be represented by a safety tree because a safety tree also incorporates a linear arithmetic proof that it is winning (its annotation). There exist definable finite memory strategies with no such proofs (analogous to the situation in program verification: there are programs in linear arithmetic with no safe inductive invariant in linear arithmetic). The class of REACH strategies that can be synthesized by our algorithm corresponds to the class of definable strategies for some fixed bounded variation of the reachability game.

In Section 3.2.1, we argued that every strategy description language must have limitations. We will now comment on some of the specific limitations of Algorithm 1 that result from our choice of strategy description language.

Definable Bounded Reachability Strategies A consequence of limiting the reachability player to bounded strategies is that our algorithm is not capable of synthesizing termination arguments for the reachability player, as illustrated in Example 6.2.

Example 6.2. Consider a two dimensional reachability game $\mathcal{G}(\text{init}, \text{reach}, \text{safe})$ where

$$\begin{aligned}\text{init}(x_0, x_1) &\triangleq x_0 = 0 \land x_1 = 0\\ \text{safe}(x_0, x_1, y_0, y_1) &\triangleq (x_0 = 0 \land y_0 = 1) \lor (x_0 = y_0 \land x_1 > 0 \land y_1 = x_1)\\ \text{reach}(y_0, y_1, x_0, x_1) &\triangleq x_0 = y_0 \land x_1 = y_1 - 1\end{aligned}$$

The game starts at the position $(0, 0)$. In the first move of the game, the safety player moves to $(1, p)$ for some $p \in \mathbb{Q}$, after which the players take turns decrementing p and checking that p is positive. The reachability player has a winning strategy for this game, but not for $\mathcal{G}_n(\text{init}, \text{reach}, \text{safe})$ for any n.

Termination arguments are typically necessary for proving liveness properties. It would be an interesting direction of future research to complement our strategy synthesis procedure (which is primarily aimed at synthesizing safety strategies) with a more powerful method for synthesizing reachability strategies.

Definable Finite-Memory Safety Strategies Reachability games are a special case of Gale-Stewart games [Gale and Stewart 1953], and so are determined by *memoryless* strategies. Thus at first glance, restricting (as we do) the safety player to a finite memory strategy may not seem like a limitation. However, this result does not take *definability* into account: there are reachability games where the safety player has a definable infinite-memory strategy but no *definable* finite-memory strategy. The game in Example 6.3 is one such example.

Example 6.3. Consider a four dimensional reachability game $\mathcal{G}(\text{init}, \text{reach}, \text{safe})$ where

$$\text{init}(x_0, x_1, x_2, x_3) \triangleq x_0 = 1 \land x_1 > 0 \land x_2 = 0 \land x_3 = 0$$

$$\text{safe}\left(\begin{array}{c} x_0, x_1, x_2, x_3, \\ y_0, y_1, y_2, y_3 \end{array}\right) \triangleq y_1 = x_1 \land y_2 = x_2 \land \left(\begin{array}{r} (x_0 \neq 0 \rightarrow y_0 = x_0 \land y_3 = x_3) \\ \land (x_0 = 0 \Rightarrow y_0 = -1) \end{array}\right)$$

$$\text{reach}\left(\begin{array}{c} y_0, y_1, y_2, y_3, \\ x_0, x_1, x_2, x_3 \end{array}\right) \triangleq x_1 = y_1 \land \left(\begin{array}{r} (y_0 = 1 \Rightarrow ((y_0 = 1 \lor y_0 = 0) \land x_2 = y_2 + y_1 \land x_3 = y_3) \\ \land (y_0 = -1 \Rightarrow \left(\begin{array}{c} \neg(y_2 = 0 \land y_3 = 0) \\ \land x_0 = y_0 \land x_2 = y_2 - y_1 \land x_3 = y_3 - 1) \end{array}\right) \end{array}\right)$$

The game starts at a position $(1, k, 0, 0)$, with $k > 0$ chosen by the reachability player. The game consists of three phases 1,0,−1, with the phase stored in the first coordinate. In the first phase of the game (addition phase), the safety player is passive and the reachability player either moves from $(1, k, n, z)$ to $(1, k, n + k, z)$ or transitions to phase 0 by moving to $(0, k, n + k, z)$. In phase 0 (guess phase), the safety player transitions to phase -1 and selects some value for the fourth coordinate j, moving from $(0, k, n, z)$ to $(-1, k, n, j)$. In phase -1 (subtraction phase), the safety player is passive and the reachability player moves from $(1, k, n, j)$ to $(1, k, n - k, j - 1)$ unless n and j are simultaneously zero, in which case the reachability player loses.

The safety player has an obvious *nonlinear* (i.e., not definable) strategy of choosing j to be n/k in phase 0. The safety player also has an *infinite memory* strategy, in which, it remembers the number of rounds played with the game being in phase 1. There is no definable finite memory strategy.

7 CASE STUDIES

We illustrate the effectiveness of our strategy synthesis approach by applying it to a variety of games and program synthesis examples. We implemented our approach in a tool we call SimSynth, which is written in OCaml on top of the implementation of SimSat [Farzan and Kincaid 2016].

Name	Alchemist-CSDT	CVC4-1.5.1	SimSynth
max15	Timeout	3.3s	Timeout
array_search15	Timeout	0.1s	3.0s
array_sum8_15	Timeout	0.0s	0.3s
tenfunc2	0.0s	0.1s	0.1s
polynomial4	0.0s	21.0s	0.0s
hms	Timeout	Timeout	0.0s
scaleweights	Timeout	0.1s	0.3s
lub10	Timeout	38.1s	4.0s
inverse10	Timeout	Timeout	2.4s
round10	Error	Timeout	8.8s
puzzle35	Timeout	Timeout	0.1s
puzzle35_opt	Timeout	Unknown	0.2s

Fig. 7. Functional synthesis benchmarks

SimSynth uses the Craig interpolation facilities of Z3 for sequence, tree, and binary interpolation [McMillan and Rybalchenko 2013]. Our experiments were conducted on a machine running Ubuntu 16.04 equipped with a 4-core Intel(R) Core(TM) i7 2.70GHz processor and 8GB memory.

7.1 Linear Satisfiability Games Instances

Functional Synthesis Benchmarks The procedure presented in Section 4 is a decision procedure for the *single invocation functional synthesis* problem [Reynolds et al. 2015]. Instances of single invocation synthesis can be translated into $\forall^*\exists^*$ formulas. In Figure 7, we compare this procedure with Alchemist-CSDT and CVC4-1.5.1, the two top competitors in the conditional linear integer arithmetic category of the 2016 Syntax-Guided Synthesis Competition (SyGuS-COMP). The benchmarks are drawn from the most difficult single-invocation benchmarks from SyGuS-COMP,[2] from [Kuncak et al. 2010], and some new benchmarks. The `lub10`, `round10`, and `puzzle35_opt` benchmarks are particularly interesting since they are *optimal* synthesis problems, which correspond to $\forall^*\exists^*\forall^*$ formulas.

7.2 Linear Reachability Games

This section includes a collection of instances for the reachability game solver.

```
assume(21 <= temp <= 24);
while(*) {
  assert(20 <= temp <= 25);
  // keep the temperature between 20 and 25
  if (isOn == 1) {
    temp = temp + 1 - (1/10 * (temp - 19));
  } else {
    temp = temp - (1/10 * (temp - 19));
  isOn = ??;
  }
}
```

Program Synthesis The code snippet on the right illustrates an *incomplete program* (i.e. a program with unknown parts) that is a controller for a thermostat (taken from [Beyene et al. 2014]). The task, which is the standard task of program synthesis, is to discover the appropriate code for the unknown part so that the program satisfies its specification. This specification is captured by an `assert` statement in the code. SimSynth finds a winning strategy for the safety player, (who wants to maintain the safety of the assertion), in 0.3 seconds. It suggests for the unknown ?? to be filled in with the value 1 once every six iterations of the loop and zero for all others. Note that this problem is not a functional synthesis problem. It

[2] SyGuS-COMP contains several families of parameterized benchmarks. We selected the benchmark in each family with the largest parameter (e.g., we selected 15-way maximum, but omitted 14-way maximum, 13-way maximum, ...).

is unclear what the specification for a functional synthesis problem should be for the purpose of synthesizing the missing code. The task of the synthesizer in an instance like this is to discover both the missing code and the loop invariant that would guarantee the desired property. The strength of our approach is exactly this: synthesizing the code and the invariant that proves its correctness together.

Cinderella-Stepmother Game We used several instances of the game introduced in Section 2.2, which have different winners and different winning strategies for each winner to test our reachability game solver.

The table on the right summarizes the results, for varying bucket capacities c (timeout is set at 10 minutes). In Section 2.2 we argued that the Stepmother can trivially win the game when $c < 1$, and Cinderella can win the game for $c \geq 3$ by adapting a *round-robin strategy*. We used the specific case of $c = 3$ as the example to illustrate our algorithm. The game becomes more challenging to determine for $1 < c < 3$. In this range, Cinderella has a winning strategy for $2 \leq c$ and the Stepmother has a winning strategy for $c < 2$. We refer the reader to [Hurkens et al. 2011] for descriptions of winning strategies for bucket capacities in this range.

Capacity	Winner	Time
c=3	Cinderella	2.2s
c=2.5	Cinderella	53.8s
c=2	Cinderella	68.9s
c=1.8	—	to
c=1.7	Stepmother	2.5s
c=1.6	Stepmother	1.5s
c=1.5	Stepmother	1.4s
c=1.4	Stepmother	0.2s

In [Beyene et al. 2014], it is conjectured that “the problem becomes more challenging for $1.5 \leq c < 3$ … in such cases fully automated strategy synthesis seems unrealistic, and computer-assisted proofs driven by user-provided hints or templates are more plausible.” SimSynth can solve a large section of this interval without *any* user-provided hints.

Game of Nim Nim is played with a number of heaps of pebbles. Two players take turns removing pebbles from distinct heaps. On each turn, a player must remove at least one pebble, and may remove any number of pebbles provided that they all come from the same heap. The goal of the game is to be the player to remove the last object. We experimented with 3 variations of the game, with 2, 3, and 4 heaps. The table on the right reports the results. The tuple parameter indicates the initial number of pebbles in each heap. The number of rounds is bounded by the values of initial heap sizes across all plays in a game, but different plays can take different number of rounds (in contrast to games corresponding to quantified formulas that are always played at a fixed number of rounds). As a consequence of the boundedness of the game, our strategy synthesis algorithm is complete for this family of games, making it an interesting stress test for SimSynth. The timeout was set at 10 minutes for the purpose of this experiment.

Heap Sizes	Winner	Time
(4,4)	Player 2	9.2s
(4,5)	Player 1	13.0s
(5,5)	Player 2	68.6s
(5,6)	Player 1	66.2s
(6,6)	—	to
(2,2,2)	Player 1	0.4s
(1,2,3)	Player 1	2.9s
(2,3,3)	Player 1	2.1s
(3,3,3)	Player 1	2.0s
(4,4,4)	Player 1	11.7s
(5,5,5)	Player 1	106s
(5,5,6)	—	to
(2,2,2,2)	Player 2	111s
(2,2,2,3)	—	to

Game of Tag This pursuit-evasion-style game is a version of the playground game of *Tag*, where a *tagged* player is chasing other players to *tag* them. For this example, we have two players on the 2-dimensional plane, represented as two points, which move at different speeds v_1 (for the reachability player or the tagged player) and v_2 (for the safety player). We considered two variations:

- $v_1 > v_2$, where the reachability strategy was synthesized in 1.9 seconds, and
- $v_2 > v_1$, where the safety strategy was synthesized in 0.5 seconds.

8 RELATED WORK

Games on Finite Graphs and Restricted Infinite Graphs. There is a rich literature on decision procedures for games on graphs with application to formal methods [Emerson and Jutla 1991; Kupferman and Vardi 1999; Pnueli and Rosner 1989; Thomas 1995]. There are many known algorithms for solving games on *finite graphs*. These include both explicit-state [Thomas 1995] and symbolic [Harding et al. 2005; Piterman et al. 2006] techniques. There are known decidability results for games on certain restricted classes of infinite graphs, such as pushdown graphs [Cachat 2002; Walukiewicz 2001] and prefix-recognizable graphs [Cachat 2003].

Both types of games, unlike linear reachability games, are decidable and admit efficient algorithms, and so we will forgo a direct comparison of our approach with these.

Infinite Games on Infinite Graphs. For graphs that represent the state space of infinite-state programs, the known approaches for solving games falls in one of the the two categories: (1) those based on symbolic execution, and (ii) those based on abstraction-refinement. The approach proposed by De Alfaro et al. [2001] is an example of the first category, where a symbolic semi-algorithm explores the state space of the game directly. The exploration is based on the *controllable precondition* operator, which keeps track of the set of states from which a player can force the game into a given region in a single round by choosing the appropriate move. The state-space of the game is partitioned into equivalence classes (e.g. through the two-player versions of trace equivalence of bisimilarity), with a classification of the termination of their semi-algorithm coinciding with the corresponding equivalence relation having a finite index on the games state space. These termination criteria, however, are not sufficient to guarantee the possibility of synthesizing a winning strategy, the main focus of this paper. An extra condition is required for that, namely the availability of only finitely many possible moves between two regions related by the controllable precondition operator.

The second category of methods [Ball and Kupferman 2006; Fecher and Huth 2006; Fecher and Shoham 2011; Grumberg et al. 2005, 2007; Gurfinkel and Chechik 2006] adapt an approach to solving games inspired by predicate abstraction and CEGAR, originally proposed for safety verification. These methods make use of an abstract transition system with overapproximate ("may") and underapproximate ("must") transitions, and properties are interpreted over a 3-valued semantic domain. These approaches, which are referred to as *reductionist* methods in De Alfaro et al. [2001], an explicit abstraction is constructed so that verification/game solving is reduced to a well-understood finite state problem. Our approach (like modern methods for program verification [McMillan 2006]) does not explicitly construct such an abstraction. In contrast, our approach exploits the intuition deals with infinite-state games of increasing lengths to solve the general problem.

The closest work to this paper is the technique proposed in Beyene et al. [2014]. Sound and complete proof rules for verifying the existence of winning strategies for safety, reachability, and linear temporal logic games are devised in Beyene et al. [2014]. The proof rules are expressed as constrained Horn clauses with existential quantifiers, allowing solvers for such clauses to be brought to bear on solving games, in particular EHSF [Beyene et al. 2013]. To cope with existential quantifiers, EHSF makes use of Skolem relations with templates. An advantage of the template-based approach is that it can be used to model human intuition about the problem at hand. The disadvantage is that it places an additional burden on the user to provide the templates.

Pnueli and Kesten [2002] present a sound and relatively complete proof system for proving CTL* properties of (possibly infinite state) reactive systems. This system can be used to prove that a given player wins a linear reachability games. Pnueli and Kesten focus on the design of the proof system, and do not address automation. Pnueli and Kesten's proof system is based on *statification*: iteratively replacing temporal subformulae with sufficient (non-temporal) state formulae. Their proof system accomplishes a similar task to our safety trees, for a broader set of properties but without a clear road towards practical automation. The disjunction of annotations of all vertices of a safety tree corresponds to a *winning region* for the safety player, which corresponds to a statification of the temporal game formula.

Program Synthesis and Repair. There is a close relation between games and program synthesis/repair. There is a long tradition of using game-solving at the service of *reactive* program synthesis, however the majority of the effort has been for reactive finite-state systems [Jobstmann et al. 2005; Pnueli and Rosner 1989; Solar-Lezama et al. 2006], or on functional, rather than reactive, programs [Kuncak et al. 2010; Reynolds et al. 2015; Srivastava et al. 2010; Vechev et al. 2010].

Branching-Time Verification. There are several techniques for verifying various classes of branching time program properties[Ball and Kupferman 2006; Cook and Koskinen 2013; Fecher and Huth 2006; Fecher and Shoham 2011; Grumberg et al. 2005, 2007; Gurfinkel and Chechik 2006]. The work of Cook and Koskinen [2013] stands out as the most recent and most complete (handling the entire class of CTL, CTL*, or modal μ-calculus properties) among the automated techniques. Cook and Koskinen [2013] proposed an automatic proof method for verifying branching-time properties of programs, through a combination of a form of Skolemization with a refinement loop. They reduce existential reasoning to universal reasoning by placing restrictions on the program state space and then ensuring the original existential property holds using non-termination [Gupta et al. 2008] proofs to prove that the chosen restriction is *recurrent*. The iteratively refine the candidate restrictions until an appropriate one is found. Their proof rule is sound and complete, but their algorithm is incomplete because it relies on (i) an incomplete universal CTL reasoning, (ii) an incomplete algorithm to synthesize restrictions, which may too greedy in its proof search and fail to produce a proof. In principle, the class of games handled by Cook and Koskinen [2013] is more general than reachability games; however our approach for solving reachability games rests on substantially different foundations.

Satisfiability Games. Game semantics for quantifiers is a classical topic in logic [Hintikka 1982] that has been recently been exploited in the design of decision procedures [Bjørner and Janota 2015; Farzan and Kincaid 2016]. Such procedures determine the winner of satisfiability games. This paper contributes the first complete method for synthesizing the strategy of the winner.

ACKNOWLEDGMENTS

We thank Aws Albarghouthi, Pavol Cerny, and the anonymous reviewers for their valuable feedback on this paper.

REFERENCES

Aws Albarghouthi, Isil Dillig, and Arie Gurfinkel. 2016. Maximal specification synthesis. In *POPL*. 789–801.

Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design (FMCAD), 2013*. IEEE, 1–8.

Rajeev Alur, Salar Moarref, and Ufuk Topcu. 2016. Compositional Synthesis of Reactive Controllers for Multi-agent Systems. In *CAV*. 251–269.

Thomas Ball and Orna Kupferman. 2006. An abstraction-refinement framework for multi-agent systems. In *LICS*. IEEE, 379–388.

Tewodros Beyene, Swarat Chaudhuri, Corneliu Popeea, and Andrey Rybalchenko. 2014. A Constraint-based Approach to Solving Games on Infinite Graphs. In *POPL*. 221–233.

Tewodros A. Beyene, Corneliu Popeea, and Andrey Rybalchenko. 2013. Solving Existentially Quantified Horn Clauses. In *CAV*. 869–882.

Nikolaj Bjørner and Mikolás Janota. 2015. Playing with Quantified Satisfaction. In *LPAR - short presentations*. 15–27.

Régis Blanc, Ashutosh Gupta, Laura Kovács, and Bernhard Kragl. 2013. Tree Interpolation in Vampire. In *LPAR-19*. 173–181.

Marijke Bodlaender, Cor Hurkens, Vincent Kusters, Frank Staals, Gerhard Woeginger, and Hans Zantema. 2012. Cinderella versus the Wicked Stepmother. In *IFIP TCS*. 57–71.

Thierry Cachat. 2002. Symbolic strategy synthesis for games on pushdown graphs. In *ICALP*. 704–715.

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

Byron Cook and Eric Koskinen. 2013. Reasoning about Nondeterminism in Programs. In *PLDI*. 219–230.

Neil T. Dantam, Zachary K. Kingston, Swarat Chaudhuri, and Lydia E. Kavraki. 2016. Incremental Task and Motion Planning: A Constraint-Based Approach. In *Robotics: Science and Systems XII, University of Michigan, Ann Arbor, Michigan, USA, June 18 - June 22, 2016.*

Luca De Alfaro, Thomas Henzinger, and Rupak Majumdar. 2001. Symbolic algorithms for infinite-state games. In *CONCUR*. Springer, 536–550.

E. Allen Emerson and Charanjit Jutla. 1991. Tree automata, mu-calculus and determinacy. In *FOCS*. IEEE, 368–377.

Azadeh Farzan and Zachary Kincaid. 2016. Linear Arithmetic Satisfiability via Strategy Improvement. In *IJCAI*. 735–743.

Harald Fecher and Michael Huth. 2006. Ranked predicate abstraction for branching time: Complete, incremental, and precise. In *ATVA*. Springer, 322–336.

Harald Fecher and Sharon Shoham. 2011. Local abstraction–refinement for the μ-calculus. *STTT* 13, 4 (2011), 289–306.

Jeanne Ferrante and Charles Rackoff. 1975. A decision procedure for the first order theory of real addition with order. *SIAM J. Comput.* 4, 1 (1975), 69–76.

Michael J Fischer and Michael O Rabin. 1974. *Super-Exponential Complexity of Presburger Arithmetic.* Technical Report. Project MAC Mass. Inst. Of Tech.

David Gale and Frank M Stewart. 1953. Infinite games with perfect information. *Contributions to the Theory of Games* 2 (1953), 245–266.

Orna Grumberg, Martin Lange, Martin Leucker, and Sharon Shoham. 2005. Don't know in the μ-calculus. In *VMCAI*. 233–249.

Orna Grumberg, Martin Lange, Martin Leucker, and Sharon Shoham. 2007. When not losing is better than winning: Abstraction and refinement for the full μ-calculus. *Information and Computation* 205, 8 (2007), 1130–1148.

Ashutosh Gupta, Thomas A. Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. 2008. Proving non-termination. In *POPL*. 147–158.

Arie Gurfinkel and Marsha Chechik. 2006. Why waste a perfectly good abstraction? In *TACAS*. 212–226.

Aidan Harding, Mark Ryan, and Pierre-Yves Schobbens. 2005. A new algorithm for strategy synthesis in LTL games. In *TACASs*. Springer, 477–492.

Jaakko Hintikka. 1982. Game-theoretical semantics: insights and prospects. *Notre Dame Journal of Formal Logic Notre-Dame, Ind.* 23, 2 (1982), 219–241.

Antonius J. C. Hurkens, Cor A. J. Hurkens, and Gerhard J. Woeginger. 2011. How Cinderella Won the Bucket Game (and Lived Happily Ever After). *Mathematics Magazine* 84, 4 (2011), pp. 278–283.

Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. 2005. Program Repair as a Game. In *CAV*. 226–238.

Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. 2010. Complete functional synthesis. In *PLDI*. 316–329.

Orna Kupferman and Moshe Y. Vardi. 1999. Robust satisfaction. In *CONCUR*. 383–398.

Zohar Manna and Richard Waldinger. 1980. A Deductive Approach to Program Synthesis. *ACM Trans. Program. Lang. Syst.* 2, 1 (Jan. 1980), 90–121.

David Marker. 2000. Introduction to model theory. *Model theory, algebra, and geometry* 39 (2000), 15–35.

Donald A. Martin. 1975. Borel Determinacy. *Annals of Mathematics* 102, 2 (1975), 363–371.

Kenneth McMillan. 2006. Lazy abstraction with interpolants. In *CAV*. 123–136.

Kenneth McMillan and Andrey Rybalchenko. 2013. *Solving Constrained Horn Clauses using Interpolation.* Technical Report. MSR.

Nir Piterman, Amir Pnueli, and Yaniv SaâĂŹar. 2006. Synthesis of reactive(1) designs. In *VMCAI*. 364–380.

Amir Pnueli and Yonit Kesten. 2002. A Deductive Proof System for CTL*. In *CONCUR*. 24–40.

Amir Pnueli and Roni Rosner. 1989. On the synthesis of a reactive module. In *POPL*. ACM, 179–190.

Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark W. Barrett. 2015. Counterexample-Guided Quantifier Instantiation for Synthesis in SMT. In *CAV*. 198–216.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. 2006. Combinatorial sketching for finite programs. In *ASPLOS*. 404–415.

Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. 2010. From program verification to program synthesis. In *POPL*. 313–326.

Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. 2013. Template-based program verification and program synthesis. *International Journal on Software Tools for Technology Transfer* 15, 5 (01 Oct 2013), 497–518.

Wolfgang Thomas. 1995. On the synthesis of strategies in infinite games. In *STACS*. 1–13.

Martin T. Vechev, Eran Yahav, and Greta Yorsh. 2010. Abstraction-guided synthesis of synchronization. In *POPL*. 327–338.
Igor Walukiewicz. 2001. Pushdown processes: Games and model-checking. *Information and computation* 164, 2 (2001), 234–263.

A PROOFS

Lemma 5.3. *Let $T = \langle N, E, r, X, \Phi, M, \triangleright \rangle$ be a well-labeled safety tree, let $u \in N$, and let $\pi = \mathbf{r}_1\mathbf{s}_1 \cdots \mathbf{r}_n\mathbf{s}_n$ be a partial play that starts in a position $\mathbf{r}_1$ that satisfies the annotation $\Phi(u)$ of u, and that conforms to the partial strategy of u (i.e., for all i, $g_T^u(\mathbf{r}_1 \cdots \mathbf{r}_{i-1})$ is defined and equal to $\mathbf{s}_i$). Then SAFE does not lose π, in the sense that either (1) all of SAFE's moves are legal, or (2) REACH makes an illegal move before any illegal move of SAFE.*

Proof. Let $v = succ_T^u(\mathbf{r}_1)$. We will prove that SAFE may legally move from $\mathbf{r}_1$ to $\mathbf{s}_1$, and that either REACH's move from $\mathbf{s}_1$ to $\mathbf{r}_2$ is *illegal* or $\mathbf{r}_2$ satisfies $\Phi(v)$. The main result follows by induction.

If u is an expanded node, then by the *Adequacy* condition and the fact that $\mathbf{r}_1$ satisfies $\Phi(u)$, we have that $G_{(u,w)}(\mathbf{r}_1)$ holds for some $(u, w) \in E$; by the definition of $succ_T^u$ we have that $G_{(u,v)}(\mathbf{r}_1)$ holds (i.e., $\mathbf{r}_1$ must satisfy the guard of the transition from u to v). By the *Availability* condition and the fact that $G_{(u,v)}(\mathbf{r}_1)$ holds, we can conclude that safe$(\mathbf{r}_1, \mathbf{s}_1)$—i.e., the move from $\mathbf{r}_1$ to $\mathbf{s}_1$ is a legal for the SAFE player. By the *Consecution* condition and the fact that $\mathbf{r}_1$ satisfies $\Phi(u)$ and $G_{(u,v)}$, we conclude that either reach$(\mathbf{s}_1, \mathbf{r}_2)$ *fails* to hold (i.e., moving from $\mathbf{s}_1$ to $\mathbf{r}_2$ is illegal for REACH), or $\Phi(v)(\mathbf{r}_2)$ holds.

If u is covered by some node u', then by the *Covering* condition we have that $\mathbf{r}_1$ that satisfies the annotation $\Phi(u')$ and proceed as above. Note that since u' may not be covered, there is no circularity in this argument. □

Proposition 5.5. *Let $\mathcal{G}(init, reach, safe)$ be a linear reachability game. If there exists a well-labeled, complete safety tree for $\mathcal{G}(init, reach, safe)$, then the safety player wins $\mathcal{G}(init, reach, safe)$.*

Proof. Let $T = \langle N, E, r, X, \Phi, M, \triangleright \rangle$ be a well-labeled, complete safety tree for $\mathcal{G}(\text{init}, \text{reach}, \text{safe})$ Let $\pi = \mathbf{r}_1\mathbf{s}_1 \cdots$ be a play conforming to T. We must show that SAFE wins π. If $\mathbf{r}_1$ is a legal move for the REACH player, we have that $\mathbf{r}_1$ satisfies $\Phi(r)$ by the *Initiation* condition (if not, then we are done—REACH has made an illegal move and SAFE wins). By Lemma 5.3, we have that SAFE does not lose any finite prefix of π. It follows that SAFE wins π. □

Proposition 5.6. *If the reachability player wins $\mathcal{G}_n(init, reach, safe)$ for some n, then (assuming a fair expansion policy, in which any vertex that is added to the safety tree is eventually either expanded or covered and remains covered) Algorithm 1 terminates with a winning reachability strategy for $\mathcal{G}(init, reach, safe)$.*

Proof. We prove that, if the algorithm does not terminate, then SAFE wins $\mathcal{G}_n(\text{init}, \text{reach}, \text{safe})$ for all n.

For any i, let $T_i = \langle N_i, E_i, r, X_i, \Phi_i, M_i, \triangleright_i \rangle$ denote the safety tree that results after i iterations of the main loop in Algorithm 1. Let $U_{i,\ell}$ denote the set of vertices of T_i at level $\leq \ell$ that are not expanded. We define n-reachability in T_i as follows:

- The root vertex is n-reachable for all n.
- If u is n-reachable and $(u, v) \in E_i$, then v is $(n + 1)$-reachable.
- If u is n-reachable and $u \triangleright_i v$, then v is n-reachable.

We begin with a lemma: for any ℓ, there is some i such that $U_{i,\ell} = U_{i',\ell}$ for all $i' > i$. We proceed by induction on ℓ. The base case is trivial: the root of the tree is unchanging, and after expanding it remains expanded ($i = 1$ suffices). For the induction step, let ℓ be a natural number and suppose that i is such that $U_{i,\ell} = U_{i',\ell}$ for all $i' > i$. Observe that, since vertices at level $\leq \ell$ are never expanded after iteration i (otherwise the set of unexpanded vertices at level $\leq \ell$ would change, contradicting the induction hypothesis), we have $U_{j,\ell+1} \supseteq U_{j+1,\ell+1}$ for all $j \geq i$ (expansion of a vertex at level $\geq \ell + 1$ may result in removing unexpanded nodes at level $\ell + 1$, but can only add *expanded* nodes to level $\ell + 1$). Since a descending chain of finite sets cannot strictly decrease infinitely often, there is some k such that $U_{k,\ell+1} = U_{k',\ell+1}$ for all $k' > k$.

Now we prove that for every n, there exists ℓ_n and i_n such that for all $i' > i_n$ (1) every n-reachable vertex in $T_{i'}$ is at level $\leq \ell_n$ (2) every $(n-1)$-reachable vertex in $T_{i'}$ is either expanded or covered. We prove this by induction on n. The base case is trivial: there is only one 0-reachable vertex in any tree (the root) and it is always at level 0. For the induction step, let n be arbitrary and suppose that i_n and ℓ_n are as above. By the lemma above, there is some j such that $U_{j,\ell+1} = U_{j',\ell+1}$ for all $j' > j$. By the assumption of fairness, for each $u \in U_{j,\ell+1}$, there is some natural $k(u)$ and some vertex $c(u)$ such that $u \triangleright_{k'} c(u)$ for all $k' > k(u)$ (i.e., a vertex that is never expanded must eventually become covered and stay covered). Define i_{n+1} to be the maximum over j and all $k(u)$ ($u \in U_{j,\ell+1}$) and ℓ_{n+1} to be the maximum over $\ell_n + 1$ and the level of $c(u)$ (for $u \in U_{j,\ell+1}$). Let $i' \geq i_{n+1}$ be arbitrary.

(1) Let v be $(n+1)$-reachable in $T_{i'}$. We must show that v has level $\leq \ell_{n+1}$.
   - Case: there is some $(u, v) \in E_{i'}$ such that u is n-reachable. Since u is n-reachable, it must have level $\leq \ell_n$ (by the induction hypothesis), and thus v must have level $\leq \ell_n + 1 \leq \ell_{n+1}$.
   - Case: there is some $u \triangleright_{i'} v$ such that u is $(n+1)$-reachable. Since u is covered by v, u is not expanded, and thus may not cover any vertex. Thus, there is some $(u', u) \in E_{i'}$ such that u' is n-reachable. As above, we have that u has level $\leq \ell_n + 1$. By the construction of i_{n+1} and ℓ_{n+1}, we have that $v = c(u)$, and thus v must have level $\leq \ell_{n+1}$.

(2) Let v be n-reachable in $T_{i'}$. By the induction hypothesis, v has level $\leq \ell_n$, and thus by construction of i_{n+1}, v is either expanded or covered by $k(v)$.

Finally, observe that for any n, the tree T_{i_n} defines a partial SAFE strategy such that $g^r_{T_{i_n}}(\mathbf{r}_1 \cdots \mathbf{r}_m)$ is defined for all sequences of length $m \leq n$ such that $\mathbf{r}_1$ satisfies init. By Lemma 5.3, SAFE does not lose any play of $\leq n$ rounds that conforms to this strategy. We conclude that the partial strategy defined by T_{i_n}, when restricted to sequences of length $\leq n$, is a winning SAFE strategy for $\mathcal{G}_n(\mathsf{init}, \mathsf{reach}, \mathsf{safe})$. □