Full LTL Synthesis over Infinite-State Arenas

Full LTL Synthesis over Infinite-State Arenas

Shaun Azzopardi[3(✉)], Luca Di Stefano[2], Nir Piterman[1], and Gerardo Schneider[1]

[1] University of Gothenburg and Chalmers University of Technology, Gothenburg, Sweden
[2] TU Wien, Institute of Computer Engineering, Treitlstraße 3, 1040 Vienna, Austria
[3] Dedaub, San Gwann, Malta
shaun.a@dedaub.com

Abstract. Recently, interest has increased in applying reactive synthesis to richer-than-Boolean domains. A major (undecidable) challenge in this area is to establish when certain repeating behaviour terminates in a desired state when the number of steps is unbounded. Existing approaches struggle with this problem, or can handle at most deterministic games with Büchi goals. This work goes beyond by contributing the first effectual approach to synthesis with full LTL objectives, based on Boolean abstractions that encode both safety and liveness properties of the underlying infinite arena. We take a CEGAR approach: attempting synthesis on the Boolean abstraction, checking spuriousness of abstract counterstrategies through invariant checking, and refining the abstraction based on counterexamples. We reduce the complexity, when restricted to predicates, of abstracting and synthesising by an exponential through an efficient binary encoding. This also allows us to eagerly identify useful fairness properties. Our discrete synthesis tool outperforms the state-of-the-art on linear integer arithmetic (LIA) benchmarks from literature, solving almost double as many syntesis problems as the current state-of-the-art. It also solves slightly more problems than the second-best realisability checker, in one-third of the time. We also introduce benchmarks with richer objectives that other approaches cannot handle, and evaluate our tool on them.

Keywords: Infinite-state synthesis · Liveness refinement · CEGAR

1 Introduction

Reactive synthesis provides a way to synthesise controllers that ensure satisfaction of high-level *Linear Temporal Logic* (LTL) specifications, against uncontrolled environment behaviour. Classically, synthesis was suggested and applied

This work is funded by the ERC consolidator grant D-SynMA (No. 772459) and the Swedish research council project (No. 2020-04963).

R. Piskac and Z. Rakamarić (Eds.): CAV 2025, LNCS 15934, pp. 274–297, 2025.
https://doi.org/10.1007/978-3-031-98685-7_13

in the Boolean (or finite-range) variable setting [29]. Interest in the infinite-range variable setting was soon to follow. Some of the milestones include the adaptation of the theory of CEGAR to infinite-state games [20] and the early adoption of SMT for symbolic representation of infinite-sized sets of game configurations [5]. However, in recent years, success of synthesis in the finite domain as well as maturity of SMT solvers has led to sharply growing interest in synthesis in the context of infinite-range variables, with several tools becoming available that tackle this problem. We highlight the two different (but related) approaches taken by the community: (a) application of infinite-state reactive synthesis from extensions of LTL where atoms include quantifier-free first-order formulas over infinite-range variables [8,14,22,23] and (b) direct applications to the solution of games with an infinite number of configurations [3,18,19,34]. Two notable examples of the two approaches from the last two years include: (a) the identification of a fragment of LTL with first-order atoms that allows for a decidable synthesis framework [30–32] and (b) the introduction of so-called *acceleration lemmas* [18,19,34] targeting the general undecidable infinite-state synthesis problem. The latter directly attacks a core issue of the problem's undecidability: identify whether certain repeated behaviour can eventually force the interaction to a certain state. Thus, solving the (alternating) termination problem.

Infinite-state reactive synthesis aims at producing a system that manipulates variables with infinite domains and reacts to input variables controlled by an adversarial environment. Given an LTL objective, the *realisability problem* is to determine whether a system may exist that enforces the objective. Then, the *synthesis problem* is to construct such a system, or a *counterstrategy* by which the environment may enforce the negation of the objective. While in the finite-state domain realisability and synthesis are tightly connected, this is not the case in the infinite-state domain and many approaches struggle to (practically) scale from realisability to synthesis. In this paper we focus on the more challenging synthesis problem, rather than mere realisability, to be able to construct implementations. Furthermore, our approach is tailored for the general – undecidable – case.

As mentioned, a major challenge is the identification of repeated behaviour that forces reaching a given state. Most approaches rely on one of two basic techniques: either refine an abstraction based on a mismatch in the application of a transition between concrete and abstract representations, or compute a representation of the set of immediate successors/predecessors of a given set of states. Both have limited effectiveness due to the termination challenge. Indeed, in many interesting cases, such approaches attempt at enumerating paths of unbounded length. For example, this is what happens to approaches relying on refinement [14,22], which is sound but often cannot terminate. It follows that reasoning about the effect of repeated behaviour is crucial.

We know of two attempts at such reasoning. temos [8] identifies single-action loops that terminate in a desired state, but cannot generalise to more challenging cases, e.g., where the environment may momentarily interrupt the loop, and moreover it cannot supply unrealisability verdicts. By contrast, rpgsolve [18] summarises terminating sub-games via acceleration lemmas to construct an

argument for realisability, relying on quantifier elimination with uninterpreted functions. However, this approach is limited to at most deterministic Büchi objectives, and is practically more effective for realisability than for synthesis due to the challenges of quantifier elimination. Its extension rpg-STeLA [34] attempts to identify acceleration lemmas that apply to multiple regions and thus solves games compositionally, but only supports realisability.

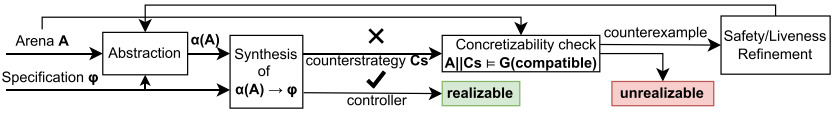


Fig. 1. Workflow of our approach.

In this paper we address the limitations described above, generalising infinite-state reactive synthesis to more expressive objectives. In particular, we consider LTL objectives over infinite-state arenas, without imposing any limit on temporal nesting. Similar to others, our atoms may include quantifier-free first-order formulas. However, we do not restrict the LTL formulas. Furthermore, our approach does not distinguish between realisability and synthesis, and can synthesise both controllers and counterstrategies. As shown in Fig. 1, our approach is based on CEGAR [21], heavily adapted for synthesis. Our main contributions are:

1. An efficient binary encoding of predicates. This reduces complexity, in terms of predicates, of abstraction building/size from exponential to polynomial, and of finite synthesis over abstractions from doubly to singly exponential.
2. A method to check counterstrategy concretisability through invariant checking, that finds minimal counterexamples to concretisability.
3. Two new kinds of liveness refinements: *Structural refinement*, which monitors for terminating concrete loops in the abstract system, and enforces eventual exit; and *Ranking refinement* that relies on the binary encoding, which ensures the well-foundedness of terms relevant to the game in the abstraction.
4. An implementation of the above contributions for LIA problems.
5. The most extensive experimental comparison of infinite-state LIA realisability and synthesis tools in literature. This shows our tool substantially outperforming all others, making it the new state-of-the-art.
6. Separately, we enrich the dataset of existing benchmarks, which currently include at most weak fairness requirements, with a selection of problems incorporating strong fairness.

For the reader's convenience we present the approach informally in Sect. 3, before formalising it in detail (Sects. 4, 5, 6). Then we describe our techniques to improve its efficiency (Sect. 7), present and evaluate our tool (Sect. 8), and conclude while also discussing related and future work (Sects. 9–10). Given space constraints here, more technical details and information about the evaluation can be found in the extended version [2].

2 Background

We use the following notation throughout: for sets S and T such that $S \subseteq T$, we write $\bigwedge_T S$ for $\bigwedge S \wedge \bigwedge_{s \in T \setminus S} \neg s$. We omit set T when clear from the context. $\mathbb{B}(S)$ is the set of Boolean combinations of a set S of Boolean variables.

Linear Temporal Logic, LTL($\mathbb{AP}$), is the language over a set of propositions $\mathbb{AP}$, defined as follows,[1] where $p \in \mathbb{AP}$: $\phi \stackrel{\text{def}}{=} \mathbf{tt} \mid \mathbf{ff} \mid p \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid X\phi \mid \phi U \phi$.

For $w \in (2^{\mathbb{AP}})^\omega$, we write $w \models \phi$ or $w \in L(\phi)$, when w satisfies ϕ. A *Moore machine* is $C = \langle S, s_0, \Sigma_{in}, \Sigma_{out}, \rightarrow, out \rangle$, where S is the set of states, s_0 the initial state, Σ_{in} the set of input events, Σ_{out} the set of output events, $\rightarrow$: $S \times 2^{\Sigma_{in}} \mapsto S$ the complete deterministic transition function, and $out : S \mapsto 2^{\Sigma_{out}}$ the labelling of each state with a set of output events. For $(s, I, s') \in \rightarrow$, where $out(s) = O$ we write $s \xrightarrow{I/O} s'$.

A *Mealy machine* is $C = \langle S, s_0, \Sigma_{in}, \Sigma_{out}, \rightarrow \rangle$, where S, s_0, Σ_{in}, and Σ_{out} are as before and $\rightarrow$: $S \times 2^{\Sigma_{in}} \mapsto 2^{\Sigma_{out}} \times S$ the complete deterministic transition function. For $(s, I, O, s') \in \rightarrow$ we write $s \xrightarrow{I/O} s'$.

Unless mentioned explicitly, both Mealy and Moore machines can have an infinite number of states. A *run* of a machine C is $r = s_0, s_1, \ldots$ such that for every $i \geq 0$ we have $s_i \xrightarrow{I_i/O_i} s_{i+1}$ for some I_i and O_i. Run r *produces* the word $w = \sigma_0, \sigma_1, \ldots$, where $\sigma_i = I_i \cup O_i$. A machine C produces the word w if there is a run r producing w. Let $L(C)$ denote the set of all words produced by C.
We cast our synthesis problem into the *LTL reactive synthesis problem*, which calls for finding a Mealy machine that satisfies a given specification over input and output variables $\mathbb{E}$ and $\mathbb{C}$.

Definition 1 (LTL Synthesis). *A specification ϕ over $\mathbb{E} \cup \mathbb{C}$ is said to be* realisable *if and only if there is a Mealy machine C, with input $2^{\mathbb{E}}$ and output $2^{\mathbb{C}}$, such that for every $w \in L(C)$ we have $w \models \phi$. We call C a* controller *for ϕ.*

A specification ϕ is said to be unrealisable *if there is a Moore machine Cs, with input $2^{\mathbb{C}}$ and output $2^{\mathbb{E}}$, such that for every $w \in L(Cs)$ we have that $w \models \neg\phi$. We call Cs a* counterstrategy *for ϕ.*

The problem of synthesis is to construct C or Cs, exactly one of which exists.

Note that the duality between the existence of a strategy and counterstrategy follows from the determinacy of turn-based two-player ω-regular games [24]. We know that finite-state machines suffice for synthesis from LTL specifications [29].

To be able to represent infinite synthesis problems succinctly we consider formulas in a theory. A *theory* consists of a set of terms and predicates over these. Atomic terms are constant values ($\mathcal{C}$) or variables. Terms can be constructed with operators over other terms, with a fixed interpretation. The set $\mathcal{T}(V)$ denotes the terms of the theory, with free variables in V. For $t \in \mathcal{T}(V)$, we write t_{prev} for the term where variables v appearing in t are replaced by fresh variables v_{prev}.

[1] See [28] for the standard semantics.

$\mathbb{V} = \{target : int = 0, floor : int = 0\}$
$\mathbb{E} = \{env\_inc, door\_open\}$
$\mathbb{C} = \{up, down\}$
Assumptions:
A1. $GF\, door\_open$
A2. $GF \neg door\_open$
Guarantees:
G1. $GF floor = target$
G2. $G(door\_open \implies (up \iff down))$
Objective:
$(A1 \wedge A2) \implies (G1 \wedge G2)$

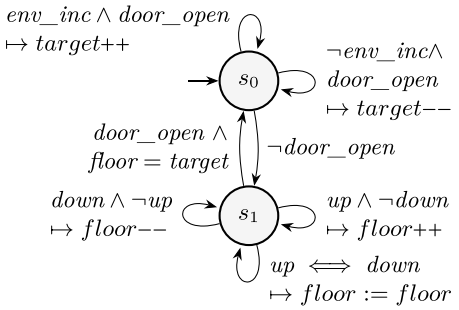


Fig. 2. Elevator example.

We use $\mathcal{ST}(V)$ to denote the set of *state predicates*, i.e., predicates over $\mathcal{T}(V)$, and $\mathcal{TR}(V)$ to denote the set of *transition predicates*, i.e., predicates over $\mathcal{T}(V \cup V_{prev})$, where $v_{prev} \in V_{prev}$ iff $v \in V$. Then, we denote by $\mathcal{P}r(V)$ the set of all predicates $\mathcal{ST}(V) \cup \mathcal{TR}(V)$. We also define the set of updates $\mathcal{U}(V)$ of a variable set V. Each $U \in \mathcal{U}(V)$ is a function $V \mapsto \mathcal{T}(V)$.

We define the set of valuations over a set of variables V as $Val(V) = V \mapsto \mathcal{C}$, using $val \in Val(V)$ for valuations. For a valuation $val \in Val(V)$, we write $val \models s$, for $s \in \mathcal{ST}(V)$ when val is a model of s. We write $t(val)$ for t grounded on the valuation val. Given valuations $val, val' \in Val(V)$, we write $(val, val') \models t$, for $t \in \mathcal{TR}(V)$, when $val_{prev} \cup val'$ is a model of t, where $val_{prev}(v_{prev}) = val(v)$ and $dom(val_{prev}) = V_{prev}$. We say a formula (a Boolean combination of predicates) is satisfiable when there is a valuation that models it. To simplify presentation, we assume $val \not\models t$ for any val that does not give values to all the variables of t.

3 Informal Overview

We give a simple instructive LIA example (Fig. 2) to illustrate our approach. Despite its simplicity, we stress that no other existing approach can solve it (see Sect. 8): since the environment can delay progress by the controller, the resulting objectives are too rich to be expressed by deterministic Büchi automata.

On the right is an automaton representing a partial design for an elevator, our arena (see Sect. 4). A transition labelled $g \mapsto U$ is taken when the guard g holds and it performs the update U. Unmentioned variables maintain their previous value. On the left, we identify input ($\mathbb{E}$) and output ($\mathbb{C}$) Boolean variables. When guards include these variables, the environment and controller's moves can affect which transitions are possible and which one is taken. The updates determine how to change the values of other variables ($\mathbb{V}$), which could range over infinite domains. Thus, the updates of the variables in $\mathbb{V}$ are determined by the interaction between the environment and the controller. The desired controller must have a strategy such that, for every possible choice of inputs, it will set the output variables so that the resulting computation satisfies a given LTL

objective, encoded on the left as $(\bigwedge_i A_i) \implies (\bigwedge_j G_j)$. LTL formulas can include quantifier-free first-order formulas over infinite-domain variables (e.g., $floor = target$). Notice that this objective includes environment fairness, making this synthesis problem impossible to encode as a deterministic Büchi game.

In our elevator, at state s_0 the environment can set a target by controlling variables in $\mathbb{E}$ to increase or decrease *target*. Once a target is set, the environment closes the elevator door (*door_open*), and the arena transitions to s_1. At s_1, the system can force the elevator to go up or down one floor, or remain at the same floor. This is not a useful elevator: it may never reach the target floor, and it may move with the door open. We desire to control it so that the target is reached infinitely often (G1), and the latter never occurs (G2). We also assume aspects of the elevator not in our control to behave as expected, i.e., that the door is not broken, and thus it opens and closes infinitely often (A1–2).

Predicate Abstraction (Definition 5) First, we soundly abstract the arena A in terms of the predicates in the specification $(A_1 \wedge A_2) \implies (G_1 \wedge G_2)$, and the predicates, and Boolean variables of the arena (here, the states in the automaton). That is,[2] $Pr = \{floor \leq target, target \leq floor, s_0, s_1\}$. This abstraction considers all possible combinations of input and output variables and Pr, and gives a set of possible predicates holding in the next state (according to the corresponding updates). For example, consider the propositional state $p = s_1 \wedge up \wedge \neg down \wedge floor < target$. In the automaton, this activates the transition that increments *floor*. Then, satisfiability checking tells us that the successor state is either $p'_1 := s_1 \wedge floor = target$ or $p'_2 := s_1 \wedge floor < target$.

We encode the arena abstraction as an LTL formula $\alpha(A, Pr)$ of the form $init \wedge G(\bigvee_{a \in abtrans} a)$, where *abtrans* is a set of abstract transitions (e.g., $p \wedge Xp'_1$ and $p \wedge Xp'_2$ are in *abtrans*), and *init* is the initial state, i.e., $s_0 \wedge floor{=}target$.

Abstract Synthesis. From this sound abstraction, we create the abstract formula $\alpha(A, Pr) \implies \phi$ and treat predicates as fresh *input* Booleans. If this formula were realisable, a controller for it would also work concretely, but it is not: at the abstract state p, the environment can always force negation of $floor = target$.

Counterstrategy Concretisability (Definition 6). For an unrealisable abstract problem we will find an abstract counterstrategy Cs. To check whether it is spurious, we model-check if A composed with Cs violates the invariant that the predicate guesses of Cs are correct in the arena. Here, Cs admits a finite counterexample ce where the environment initially increments *target*, then moves to s_1, and the controller increments *floor*, but Cs wrongly maintains $floor < target$.

Safety Refinement (Sect. 6.1). By applying interpolation [25] on ce we discover new predicates, e.g., $target - floor \leq 1$, by which we refine the abstraction to exclude ce. If we were to continue using safety refinement, we would be attempting to enumerate the whole space, which causes a state-space explosion, given the exponential complexity of predicate abstraction, and the doubly exponential complexity of synthesis.

[2] LIA predicates are normalised to a form using only $\leq$; other relations are macros.

Efficient Encoding (Sect. 7). We manage state-space explosion through a binary encoding of predicates. Note each predicate on a term corresponds to an interval on the reals. For the term $t = \mathit{floor} - \mathit{target}$, $\mathit{floor} \leq \mathit{target}$ represents $t \in (-\infty, 0]$. $\mathit{target} \leq \mathit{floor}$ represents $t \in [0, \infty)$, and $\mathit{floor} - \mathit{target} \leq 1$ represents $t \in (-\infty, 1]$. These may overlap, but instead we can define formulas whose intervals partition the line $\mathbb{R}$. Here, we get formulas for each interval: $(-\infty, -1], (-1, 0], (0, 1], (1, \infty)$. Binary-encoding these reduces the complexity of abstraction and synthesis by an exponential, w.r.t. arithmetic predicates.

Liveness Refinements (Sect. 6.2). Enumeration is not enough here, given the infinite domain of the variables. Liveness refinements are necessary. Note, once *Cs* guesses that $\mathit{floor} < \mathit{target}$, it remains in states where $\mathit{floor} < \mathit{target}$ is true. Essentially, we discover a *ce* in which *Cs* exercises the loop `while(floor ¡ target) floor := floor + 1`, and the environment believes it is non-terminating. Using known methods to determine the loop is terminating, we construct a monitor for the loop in the abstraction, with extra variables and assumptions. Then a strong fairness constraint that forces the abstraction to eventually exit the loop monitor captures its termination. We term this *structural loop refinement*. Note that this is not tied to a specific region in the arena. This allows us to encode more sophisticated loops, beyond what current tools for LTL objectives can do.

With a new synthesis attempt on the refined abstraction, a fresh terminating loop is learned, `while (target ¡ floor) floor := floor - 1`. Refining accordingly allows us to find a controller and thus solve the problem on the next attempt.

Acceleration (Sect. 7). The described partitions of the values of a term have a natural well-founded ordering which we can exploit to identify that the controller can force the abstraction to move left or right across the intervals. Consider that if the term t is currently in the interval $(1, \infty)$, and the controller can force strict decrements of t, then the value of the t must necessarily eventually move to an interval to the left (unless we have reached the left-most interval). Thus, strict decrements force the value of t to move towards the left of the partition, while strict increments force move towards the right of the partition. Only when the environment can match these increments (decrements) with corresponding decrements (increments) then can this behaviour be prevented.

By adding LTL fairness constraints to represent the described behaviour we can immediately identify a controller, with no further refinements needed.

4 Synthesis Setting

One of our contributions is our special setting that combines arenas and LTL objectives, unlike existing LTL approaches which start immediately from LTL-modulo-theories formulas [8,14,22]. We assume a theory, with an associated set of predicates $\mathcal{P}r(V)$ and updates $\mathcal{U}(V)$ over a set of variables V. We also assume two disjoint sets of Boolean inputs and outputs $\mathbb{E}$ and $\mathbb{C}$, respectively controlled by the environment and the controller. Then our specifications are LTL formulas over these variables, $\phi \in \mathrm{LTL}(\mathbb{E} \cup \mathbb{C} \cup Pr_\phi)$, where $Pr_\phi \subseteq \mathcal{P}r(V)$. LTL formulas talk about an *arena* whose state is captured by the value of V, and which

modifies its state depending on environment and controller behaviour. Arenas are deterministic; we model (demonic) non-determinism with additional environment variables. This allows us to encode concretisability checking as invariant checking, rather than the significantly more complex CTL* model checking.

Definition 2 (Arena). *An arena A over V is a tuple $\langle V, val_0, \delta \rangle$, where V is a finite set of variables, $val_0 \in Val(V)$ is the initial valuation, and $\delta : \mathbb{B}(\mathbb{E} \cup \mathbb{C} \cup \mathcal{P}r(V)) \nrightarrow \mathcal{U}(V)$ is a partial function with finite domain, such that for all $val \in Val(V)$ and for every $E \subseteq \mathbb{E}$ and $C \subseteq \mathbb{C}$ there is always a single $f \in dom(\delta)$ such that $(val, E \cup C) \models f$. An arena is finite when every $v \in V$ is finite.*

Notice that due to the finite domain of δ, an arena A defines a *finite* set of predicates $Pr \subseteq \mathcal{P}r(V)$ and a *finite* set of updates $U \subseteq \mathcal{U}(V)$ that appear in δ. We use the sets Pr and U when clear from the context.

An infinite concrete word $w \in (Val(V) \times 2^{\mathbb{E} \cup \mathbb{C}})^\omega$ is a *model* of A iff $w(0) = (val_0, E \cup C)$ (for some E and C), and for every $i \geq 0$, $w(i) = (val_i, E_i \cup C_i)$, then for the unique $f_i \in dom(\delta)$ such that $(val_i, E_i \cup C_i) \models f_i$ we have $val_{i+1} = (\delta(f_i))(val_i)$. We write $L(A)$ for the set of all models of A.

During our workflow, the words of our abstract synthesis problem may have a different domain than those of the arena. We define these as *abstract words*, and identify when they are concretisable in the arena. Then, we can define the meaning of (un)realisability modulo an arena in terms of concretisability.

Definition 3 (Abstract Words and Concretisability). *For a finite set of predicates $\mathcal{P}r \subseteq \mathcal{P}r(V)$, and a set of Boolean variables $\mathbb{E}'$, such that $\mathbb{E} \subseteq \mathbb{E}'$, an* abstract word *$a$ is a word over $2^{\mathbb{E}' \cup \mathbb{C} \cup \mathcal{P}r}$. Abstract word a* abstracts *concrete word w, with letters from $Val(V) \times 2^{\mathbb{E} \cup \mathbb{C}}$, when for every i, if $a(i) = E_i \cup C_i \cup \mathcal{P}r_i$, then $w(i) = (val_i, (E_i \cap \mathbb{E}) \cup C_i)$ for some $\mathcal{P}r_i \subseteq \mathcal{P}r$, $val_0 \models \bigwedge_{\mathcal{P}r} \mathcal{P}r_0$, and for $i > 0$ then $(val_{i-1}, val_i) \models \bigwedge_{\mathcal{P}r} \mathcal{P}r_i$. We write $\gamma(a)$ for the set of concrete words that a abstracts. We say abstract word a is* concretisable *in an arena A when $L(A) \cap \gamma(a)$ is non-empty.*

Definition 4 (Realisability modulo an Arena). *A formula ϕ in LTL$(\mathbb{E} \cup \mathbb{C} \cup \mathcal{P}r_\phi)$ is said to be* realisable *modulo an arena A, when there is a* controller *as a Mealy Machine MM with input $\Sigma_{in} = 2^{\mathbb{E} \cup \mathcal{P}r_\phi}$ and output $\Sigma_{out} = 2^{\mathbb{C}}$ such that every abstract trace t of MM that is concretisable in A also satisfies ϕ.*

A counterstrategy *to the realisability of ϕ modulo an arena A is a Moore Machine Cs with output $\Sigma_{out} = 2^{\mathbb{E} \cup \mathcal{P}r_\phi}$ and input $\Sigma_{in} = 2^{\mathbb{C}}$ such that every abstract trace t of Cs is concretisable in A and violates ϕ.*

5 Abstract to Concrete Synthesis

We attack the presented synthesis problem through an abstraction-refinement loop. We soundly abstract the arena as an LTL formula that may include fresh predicates and inputs. We fix the set of predicates that appear in the objective ϕ as $\mathcal{P}r_\phi$, and the set of predicates and inputs in the abstraction, respectively, as $\mathcal{P}r$ and $\mathbb{E}'$, always such that $\mathcal{P}r_\phi \subseteq \mathcal{P}r$ and $\mathbb{E} \subseteq \mathbb{E}'$.

Definition 5 (Abstraction). *Formula* $\alpha(A, \mathcal{P}r)$ *in* $LTL(\mathbb{E}' \cup \mathbb{C} \cup \mathcal{P}r)$ *abstracts arena* A *if for every* $w \in L(A)$ *there is* $a \in L(\alpha(A, \mathcal{P}r))$ *such that* $w \in \gamma(a)$.

$\alpha(A, \mathcal{P}r)$ is a standard predicate abstraction [16]. Given the lack of novelty, we refer to Appendix B.1 of [2] for the full details. Note, $\alpha(A, \mathcal{P}r)$ can be non-deterministic, unlike A. Constructing it is essentially an ALLSAT problem: given a transition, we identify sets from $2^{\mathcal{P}r}$ that can be true before the transition and, for each of these, sets of $2^{\mathcal{P}r}$ that can hold after the transition. However, we construct these sets incrementally, adding predicates as we discover them; and improve on the space/time complexity with a binary encoding (Sect. 7).

Given abstraction $\alpha(A, \mathcal{P}r)$, we construct a corresponding sound LTL synthesis problem, $\alpha(A, \mathcal{P}r) \implies \phi$, giving the environment control of the predicates in $\alpha(A, \mathcal{P}r)$. We get three possible outcomes from attempting synthesis of this: (1) it is realisable, and thus the concrete problem is realisable; (2) it is unrealisable and the counterstrategy is concretisable; or (3) the counterstrategy is not concretisable. We prove theorems and technical machinery essential to allow us to determine realisability (1) and unrealisability (2). In case (3) we refine the abstraction to make the counterstrategy unviable in the new abstract problem.

Theorem 1 (Reduction to LTL Realisability). *For* ϕ *in* $LTL(\mathbb{E} \cup \mathbb{C} \cup \mathcal{P}r_\phi)$ *and an abstraction* $\alpha(A, \mathcal{P}r)$ *of* A *in* $LTL(\mathbb{E}' \cup \mathbb{C} \cup \mathcal{P}r)$, *if* $\alpha(A, \mathcal{P}r) \implies \phi$ *is realisable over inputs* $\mathbb{E}' \cup \mathcal{P}r$ *and outputs* $\mathbb{C}$, *then* ϕ *is realisable modulo* A.

However, an abstract counterstrategy Cs may contain unconcretisable traces, since abstractions are sound but not complete. To analyse Cs for concretisability, we define a simulation relation between states of the concrete arena and states of Cs, capturing whether each word of Cs is concretisable. Recall, a set of predicates $\mathcal{P}r$ is the union of a set of state predicates, ST (describing one state), and transition predicates, TR (relating two states), which require different treatment.

Definition 6 (Counterstrategy Concretisability). *Consider a counterstrategy as a Moore Machine* $Cs = \langle S, s_0, \Sigma_{in}, \Sigma_{out}, \rightarrow, out \rangle$, *and an arena* A, *where* $\Sigma_{in} = 2^{\mathbb{C}}$ *and* $\Sigma_{out} = 2^{\mathbb{E}' \cup \mathcal{P}r}$.
Concretisability *is defined through the simulation relation* $\preceq_A \subseteq Val \times S$:
For every valuation val *that is simulated by a state* s, $val \preceq_A s$, *where* $out(s) = E \cup ST \cup TR$, *it holds that:*

1. *the valuation satisfies the state predicates of* s: $val \models \bigwedge ST$, *and*
2. *for every possible controller output* $C \subseteq \mathbb{C}$: *let* $val_C = \delta(val, (E \cap \mathbb{E}) \cup C)$, s_C *be s.t.* $s \xrightarrow{C} s_C$, *and* TR_C *be the transition predicates in* $out(s_C)$, *then*
   (a) *the transition predicates of* s_C *are satisfied by the transition* $(val, val_C) \models \bigwedge TR_C$, *and*
   (b) *the valuation after the transition simulates the Cs state after the transition:* $val_C \preceq_A s_C$.

Cs is concretisable *w.r.t. A when* $val_0 \preceq_A s_0$, *for A's initial valuation* val_0.

With concretisability defined, we then have a method to verify whether an abstract counterstrategy is also a concrete counterstrategy.

Theorem 2 (Reduction to LTL Unrealisability). *Given arena abstraction* $\alpha(A, \mathcal{P}r)$*, if* $\alpha(A, \mathcal{P}r) \implies \phi$ *is unrealisable with a counterstrategy Cs and Cs is concretisable w.r.t. A, then* ϕ *is unrealisable modulo A.*

In practice, we encode counterstrategy concretisability as a model checking problem on the composition of the counterstrategy and the arena, with the required invariant that predicate values chosen by the counterstrategy hold on the arena. Conveniently, this also gives witnesses of unconcretisability as finite counterexamples (rather than infinite traces), which we use as the basis for refinement. Crucially, this depends on the choices of the environment/controller being finite, which also gives us semi-decidability of finding non-concretisability.

Proposition 1. *Counterstrategy concretisability is encodable as invariant checking, and terminates for finite problems and non-concretisable counterstrategies.*

Proposition 2. *A non concretisable counterstrategy induces a finite counterexample* $a_0, \ldots, a_k \in (2^{\mathbb{E} \cup \mathbb{C} \cup \mathcal{P}r})^*$ *and concretisability fails locally only on* a_k*.*

Synthesis Semi-Algorithm. Alg. 1 shows our high-level approach. Taking an arena A and an LTL formula ϕ, it maintains a set of predicates $\mathcal{P}r$ and an LTL formula ψ. When the abstract problem (in terms of $\mathcal{P}r$) is realisable, a controller is returned (line 5); otherwise, if the counterstrategy is concretisable, it is returned (line 7). If the counterstrategy is not concretisable, we refine the abstraction to exclude it (line 8), and extend $\mathcal{P}r$ with the learned predicates, and ψ with the new LTL constraints (line 9). Alg. 1 diverges unless it finds a (counter)strategy.

Algorithm 1: Synthesis algorithm based on abstraction refinement.

```
1 Function synthesise(A, φ):
2     Pr, ψ := Pr_φ, true
3     while true do
4         φ^A_α := (α(A, Pr) ∧ ψ) ⟹ φ
5         if realisable(φ^A_α, 𝔼 ∪ Pr, ℂ) then return (true, strategy(φ^A_α, 𝔼 ∪ Pr, ℂ))
6         Cs := counter_strategy(φ^A_α, 𝔼 ∪ Pr, ℂ)
7         if concretisable(φ, A, Cs) then return (false, Cs)
8         Pr′, ψ′ := refinement(A, Cs)
9         Pr, ψ := Pr ∪ Pr′, ψ ∧ ψ′
```

6 Refinement

We now present the two refinements on which our iterative approach relies, based on an analysis of a discovered counterstrategy. These refinements soundly refine the abstraction with predicates and/or new LTL constraints such that similar counterexamples will not be re-encountered in the next iteration.[3]

6.1 Safety Refinement

Consider a counterstrategy Cs and a counterexample $ce = a_0, a_1, \ldots, a_k$. The transition from a_{k-1} to a_k induces a mismatch between the concrete arena state and Cs's desired predicate state. It is well known that interpolation can determine sufficient state predicates to make Cs non-viable in the fresh abstract problem; we give a brief description for the reader's convenience. Let $p_i = \bigwedge_{\mathcal{P}r}(a_i \cap \mathcal{P}r)$, with each variable v replaced by a fresh variable v_i, and each variable v_{prev} by v_{i-1}. Similarly, let g_i and u_i be respectively the corresponding symbolic transition guard and update (i.e., $\delta(g_i) = u_i$), such that all updates $v := t$ are rewritten as $v_{i+1} = t_i$, where term t_i corresponds to t with every variable v replaced by v_i.

In order to characterize the mismatch between the arena and its abstraction, we construct the following formulas. Let $f_0 = val_0 \wedge p_0 \wedge g_0 \wedge u_0$, where we abuse notation and refer to val_0 as a Boolean formula. For $1 \leq i < k$, let $f_i = p_i \wedge g_i \wedge u_i$, while $f_k = p_k$. Then $\bigwedge_{i=0}^{k} f_i$ is unsatisfiable. Following McMillan [25], we construct the corresponding set of *sequence interpolants* $I_0, ..., I_{k-1}$, where $f_0 \implies I_1$, $\forall 1 \leq i < k. I_i \wedge f_i \implies I_{i+1}$, $I_{k-1} \wedge f_k$ is unsatisfiable, as all the variables of I_i are shared by both f_{i-1} and f_i. From these we obtain a set of state predicates $I(ce)$ by removing the introduced indices in each I_i. Adding $I(ce)$ to the abstraction refines it to make the counterstrategy unviable.

6.2 Liveness Refinement

Relying solely on safety refinement results in non-termination for interesting problems (e.g., Fig. 2). To overcome this limitation, we propose *liveness refinement*. Our main insight is that if the counterexample exposes a spurious lasso in the counterstrategy, then we can encode its termination as a liveness property.

Lassos and Loops. A counterexample $ce = a_0, \ldots, a_k$ *induces* a lasso in Cs when it corresponds to a path $s_0, \ldots, s_k$ in Cs, where $s_k = s_j$ for some $0 \leq j < k$. We focus on the last such j. Here, for simplicity, we require that concretisation failed due to a wrong state predicate guess. We split the counterexample into two parts: a stem $a_0, \ldots, a_{j-1}$, and a loop $a_j, \ldots, a_{k-1}$. Let $g_j \mapsto U_j, \ldots, g_{k-1} \mapsto U_{k-1}$ be the corresponding applications of δ and let val_j be the arena state at step j.

```
V = *
assume val_j
while ⋀(a_j ∩ Pr)
  assume g_j
  V = U_j(V)
  ...
  assume g_{k-1}
  V = U_{k-1}(V)
```

Fig. 3. ce loop.

[3] We prove a progress theorem for each refinement in Appendix C of [2].

The counterexample proves that the while-program in Fig. 3 terminates (in one iteration). To strengthen the refinement, we try to weaken the loop (e.g., expand the precondition) such that it still accepts the loop part of *ce* while terminating. We formalise loops to be able to formalise this weakening.

Definition 7 (Loops). *A* loop *is a tuple* $l = \langle V, pre, iter\_cond, body\rangle$*, where pre and iter_cond are Boolean combinations of predicates over variables* V*, and body is a finite sequence of pairs* (g_i, U_i)*, where* $g_i \in \mathcal{P}r(V)$ *and* $U_i \in \mathcal{U}(V)$*.*

A finite/infinite sequence of valuations $vals = val_0, val_1, \ldots$ *is an execution of* l*,* $vals \in L(l)$*, iff* $val_0 \models pre$*, for all* i *such that* $0 \leq i < |vals|$*, where* $n = |body|$*, then* $val_i \models g_{i \bmod n}$*,* $val_{i+1} = U_{i \bmod n}(val_i)$ *and if* $i \bmod n = 0$ *then* $val_i \models iter\_cond$*. We say a loop is* terminating *if all of its executions are finite.*

Definition 8 (Weakening). *Loop* $l_1 = \langle V_1, pre_1, ic_1, body_1\rangle$ is weaker than $l_2 = \langle V_2, pre_2, ic_2, body_2\rangle$ *when: 1.* $V_1 \subseteq V_2$*; 2.* $pre_2 \implies pre_1$ *and* $ic_2 \implies ic_1$*; 3.* $|body_1| = |body_2|$*; 4. for* $w_2 \in L(l_2)$ *there is* $w_1 \in L(l_1)$ *such that* w_2 *and* w_1 *agree on* V_1*. A weakening is* proper *if both* l_1 *and* l_2 *terminate.*

Heuristics. We attempt to find loop weakenings heuristically. In all cases we reduce *iter_cond* to focus on predicates in a_k that affect concretisability. We also remove variables from the domain of the loop that are not within the cone-of-influence [10] of *iter_cond*. We then attempt two weaker pre-conditions: (1) *true*; and (2) the predicate state before the loop is entered in the *ce*. We check these two loops, in the order above, successively for termination (using an external tool). The first loop proved terminating ($l(ce)$) is used as the basis of the refinements.

Structural Loop Refinement. We present a refinement that monitors for execution of the loop and enforces its termination.

We define some predicates useful to our definition. For each transition in the loop we define a formula that captures when it is triggered: $cond_0 \stackrel{\text{def}}{=} iter\_cond \wedge g_0$ and $cond_i \stackrel{\text{def}}{=} g_i$ for all other i. For each update U_i, we define a conjunction of transition predicates that captures when it occurs: recall U_i is of the form $v^0 := t^0, \ldots, v^j := t^j$, then we define p_i as $v^0 = t^0_{prev} \wedge \ldots \wedge v^j = t^j_{prev}$. This sets the value of variable v^k to the value of term t^k in the previous state. We further define a formula that captures the arena stuttering modulo the loop, $st \stackrel{\text{def}}{=} \bigwedge_{v \in V_l} v = v_{prev}$, where V_l is the set of variables of the loop. A technical detail is that we require updates in the loop $l(ce)$ to not stutter, i.e., $U(val) \neq val$ for all *val*. Any loop with stuttering can be reduced to one without, for the kinds of loops we consider. Thus, here $p_i \wedge st$ is contradictory, for all i.

Definition 9 (Structural Loop Refinement). *Let* l *be a terminating loop, and* $cond_i$*,* p_i*, and* st *(for* $0 \leq i < n$*) be as defined above. Assume fresh variables corresponding to each step in the loop* $inloop_0, \ldots, inloop_{n-1}$*, and* $inloop = inloop_0 \vee \ldots \vee inloop_{n-1}$*.*

The structural loop abstraction $\alpha_{loop}(A, l)$ *is the conjunction of the following:*

1. *Initially we are not in the loop, and we can never be in multiple loop steps at the same time:* $\neg inloop \wedge \bigwedge_i G(inloop_i \implies \neg \bigvee_{j \neq i}(inloop_j))$*;*

2. *The loop is entered when pre holds and the first transition is executed:* $G(\neg inloop \implies ((pre \wedge cond_0 \wedge X(p_0)) \iff X(inloop_1)))$;
3. *At each step, while the step condition holds, the correct update causes the loop to step forward, stuttering leaves it in place, otherwise we exit:*

$$\bigwedge_{0 \leq i < n} G\left((inloop_i \wedge cond_i) \implies X\left(\begin{array}{l}(p_i \implies inloop_{i+1\%n}) \wedge \\ (st \implies inloop_i) \wedge \\ (\neg(st \vee p_i) \iff \neg inloop)\end{array}\right)\right);$$

4. *At each step, if the expected step condition does not hold, we exit:* $\bigwedge_{0 \leq i < n} G((inloop_i \wedge \neg cond_i) \implies X \neg inloop)$; *and*
5. *The loop always terminates, or stutters:* $GF(\neg inloop) \vee \bigvee_i FG(st_i \wedge inloop_i)$.

Note the fresh propositions ($inloop_i$) are controlled by the environment. The LTL formulas 1–4 monitor for the loop, exiting if a transition not in the loop occurs, and progressing or stuttering in the loop otherwise. LTL formula 5 enforces that the loop is exited infinitely often, or that the execution stutters in the loop forever. This ensures that the abstract counterstrategy is no longer viable.

7 Efficient Encoding and Acceleration

The problem we tackle is undecidable, but we rely on decidable sub-routines of varying complexity: predicate abstraction (exponential in the number of predicates) and finite synthesis (doubly exponential in the number of propositions, of which predicates are a subset). Here we present an efficient binary encoding of predicates of similar forms that (1) reduces the size of and the satisfiability checks needed to compute the abstraction from exponential to polynomial, and (2) reduces complexity of abstract synthesis from doubly to singly exponential, when restricted to predicates. Moreover, this encoding allows us to identify fairness assumptions refining the abstraction, which significantly accelerate synthesis. Computing this encoding only involves simple arithmetic, but we have not encountered previous uses of it in literature.

We collect all the known predicates over the same term, giving a finite set of predicates $P_t = \{t \bowtie c_0, ..., t \bowtie c_n\}$, where t is a term only over variables, $\bowtie \in \{<, \leq\}$ and each c_i is a value. W.l.g. we assume $t \bowtie c_i \implies t \bowtie c_{i+1}$ for all i. Thus, $t < c$ appears before any other predicate $t \bowtie c + \alpha$ for $\alpha \geq 0$. For simplicity, let us assume that t is a single variable. To enable a binary representation we find disjoint intervals representing the same constraints on variable values. Namely, replace the predicates in P_t with (1) $t \bowtie c_0$, (2) for $0 < i \leq n$ the predicate $\neg(t \bowtie c_{i-1}) \wedge t \bowtie c_i$, and finally, (3) $\neg(t \bowtie c_n)$. Effectively, forming a partition of the real line $\mathbb{R}$.

Let $part(P_t) = \{t \bowtie c_0, \neg(t \bowtie c_{i-1}) \wedge t \bowtie c_i, \neg(t \bowtie c_n) \mid 0 < i \leq n\}$. We call the left- and right-most partitions the *border* partitions since they capture the left and right intervals to infinity. The other formulas define non-intersecting bounded intervals/partitions along $\mathbb{R}$. Figure 4 illustrates these partitions: this

set of formulas covers the whole line, i.e. for each point $t = c$, there is a formula f in $part(P_t)$ such that $(t = c) \models f$. Further, note how each two distinct formulas $f_1, f_2 \in part(P_t)$ are mutually exclusive. Namely, $f_1 \wedge f_2 \equiv \bot$. Given this mutual exclusivity, it is easy to construct a representation to reduce the number of binary variables in the predicate abstraction. The complexity of computing these partitions is only the complexity of sorting P_t in ascending order based on values.

In a standard predicate abstraction approach, the number of predicates is $\sum_{t \in terms} |P_t|$. With this encoding, they shrink to $\sum_{t \in terms} \lceil |log_2(|P_t| + 1)| \rceil$. Moreover, this enables a more efficient predicate abstraction computation: given we know each formula in $part(P_t)$ is mutually exclusive, we can consider each formula separately. Then, for each t instead of performing $2^{2 \times |P_t|}$ satisfiability checks we just need $(|P_t| + 1)^2$, giving a polynomial time complexity in terms of predicates, $(\prod_{t \in terms}(|P_t| + 1))^2$, instead of the exponential $2^{2 \times \sum_{t \in terms} |P_t|}$. The complexity of synthesis improves very significantly in terms of predicates, to $2^{\prod_{t \in terms} |P_t| + 1}$, instead of $2^{2^{\sum_{t \in terms} |P_t|}}$.

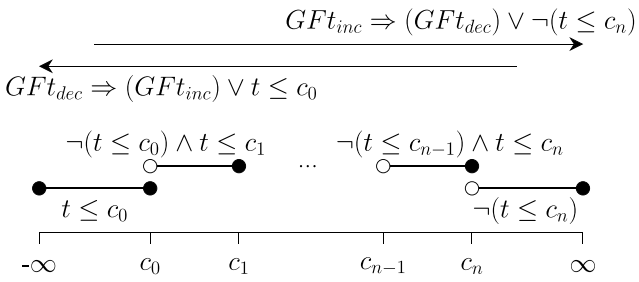


Fig. 4. Partitions for binary encoding.

Note that, to get the full view of time complexity for both abstraction and synthesis, the complexity described must be respectively multiplied by $|dom(\delta)| \times 2^{|B|}$ and $2^{2^{|B|}}$, where B is the set of Boolean propositions in the concrete problem.

As an optimisation, if both terms t and $-t$ are part of the abstraction, we transform predicates over $-t$ to predicates over t: $-t \leq c$ becomes $t \geq -c$, which becomes $\neg(t < -c)$. We note the approach described applies to both LIA and LRA, and might have applications beyond our approach.

Acceleration. The partitioning optimises the encoding of predicates extracted from the problem and learned from safety refinements. Moreover, it allows to identify liveness properties relevant to the infinite-state arena.

Consider that an abstract execution is within the leftmost partition, e.g., within $t \leq 0$. An increment in t in the arena leads to an environment choice in the abstraction of whether to stay within $t \leq 0$ or move to the next partition. Suppose the controller can repeatedly increment t with a value bounded from 0.

In the abstraction, the environment can still force an abstract execution satisfying $t \leq 0$ forever. The same is true for every partition, unless its size is

smaller than the increment, e.g., a partition with one element. This abstract behaviour is not concretisable. That is, for every concrete value of t and every c, after a finite number of increments bounded from 0, the predicate $t \bowtie c$ becomes false. Similarly for any other partition. The dual is true for decrements. We note that in LIA, every increment or decrement is bounded from 0.

We encode this fact using fairness assumptions that rely on detecting increases and decreases of a term's value with transition predicates. If for a term t we identify that all changes of t in A are at least ϵ, we define the transition predicates $t_{inc} := t_{prev} \leq t - \epsilon$ and $t_{dec} := t \leq t_{prev} - \epsilon$, refining the abstraction by a memory of when transitions increase or decrease the value of t. Notice that as changes to t are at least ϵ, when both t_{dec} and t_{inc} are false t does not change. We then add the fairness assumptions: $(GFt_{dec}) \implies GF(t_{inc} \vee f_l)$ and $(GFt_{inc}) \implies GF(t_{dec} \vee f_r)$, where f_l (f_r) is t's left-(right-)most partitions.

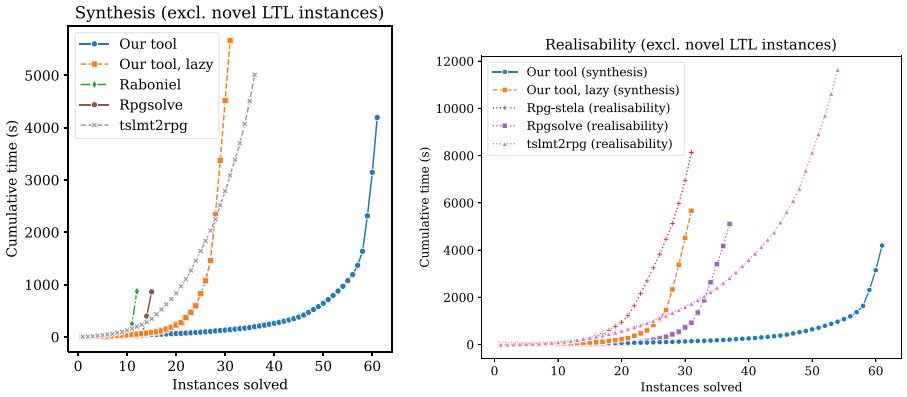
The first (second) assumption enforces every abstract execution where t strictly decreases (increases) and does not increase (decrease), to make progress towards the left-(right-)most partition. Thus, the environment cannot block the controller from exiting a partition, if they can repeatedly force a bounded from 0 decrease (increase) without increases (decreases). For each term, we can then add these two corresponding fairness LTL assumptions to the abstraction. If the left- and right-most partitions are updated during safety refinement, we update the predicates inside these fairness assumptions with the new border partitions, ensuring we only ever have at most two such assumptions per term. In our implementation for LIA $\epsilon = 1$, and to optimise we leave out these assumptions if we cannot identify increases or decreases bounded from 0 in the arena.

8 Evaluation

We implemented this approach in a tool[4] targeting discrete synthesis problems. State-of-the-art tools are used as sub-routines: Strix [26] (LTL synthesis), nuXmv [7] (invariant checking), MathSAT [9] (interpolation and SMT checking), and CPAchecker [6] (termination checking). As a further optimisation, the tool performs also a binary encoding of the states variables of the arena, given they are mutually exclusive.

We compare our tool against 5 tools from literature raboniel [22], temos [8], rpgsolve [18], rpg-STeLA [34], and tslmt2rpg (+rpgsolve) [19]. We consider also a purely lazy version of our tool, with acceleration turned off to evaluate its utility. We do not compare against other tools fully outperformed by the rpg tools [33,35], limited to safety/reachability [3,13,27], and another we could not acquire [23]. All experiments ran on a Linux workstation equipped with 32 GiB of memory and an Intel i7-5820K CPU, under a time limit of 20 min and a memory limit of 16 GiB. We show cumulative synthesis times in Fig. 5a for tools that support synthesis, and cumulative realisability times for other tools compared with our tools' cumulative synthesis times in Fig. 5b.

[4] https://github.com/shaunazzopardi/sweap. An artifact for this paper is available [11].



(a) Synthesis. (b) Realisability.

Fig. 5. Time comparison.

Benchmarks. We collect 80 LIA benchmarks from the literature. Most encode practical problems, such as robotic mission control, job scheduling, sorting, or data buffering. They are defined in TSL [14] or as deterministic games, and may include arbitrary integers as input, which we equivalently encode with extra steps that let the environment set variables to any finite value (see Sect. 9). All these benchmarks consist of problems encodable as deterministic Büchi games. Some benchmarks [34] compose multiple such games together, for added difficulty. Following others, we ignore problems [8,14] that are trivial. We only introduce one novel reachability game to these benchmarks, robot-tasks,[5] that we crafted to highlight the limitations of previous approaches compared to our own. Some of the problems from [34] are not available in TSL format. We test those on neither raboniel nor temos but we expect they would both fail, as their techniques are insufficient for Büchi goals (see Sect. 9), and for tslmt2rpg we simply consider the time taken by rpgsolve on the corresponding RPG problem.

Results (comparative evaluation).[6] It is clear from Fig. 5a that the eager version of our tool solves almost double more synthesis problems than the best competitor, and faster. The lazy version is comparable to the best competitor. For realisability, Fig. 5b shows our tool with acceleration scaling and performing much better on synthesis than the other tools do on realisability. However, the lazy version is outperformed by the rpg tools. Table 1a summarises the evaluation; for each tool we report the number of solved problems (out of 81), the ones it solved in the shortest time, and those no other tool was able to solve. Our tool is the clear winner in each category. If we consider synthesis, even without acceleration we are comparable to the state of the art: our tools solve 61 (eager) and 31 (lazy) problems, while the best competitor tslmt2rpg solves 36. When

[5] Appendix D.1 of [2] has more details about this new benchmark.
[6] Appendix D.2 of [2] has additional experimental data, and an extended discussion.

looking closely at the behaviour on the easiest instances (see Fig. 6 in [2]), we see that our tool has an initialization overhead of a few seconds while other tools can solve simple problems in under 1 s. However, our tool scales better. We also ran our lazy tool without the binary encoding, and measured noticeably worse performances: it times out on two more problems, and takes on average 10% more time (see Fig. 7 in [2]).

Table 1. Experimental results.

(a) Comparative evaluation of **Rab**oniel, **Tem**os, **RPG**solve, **T**slmt**2R**pg, **R**pg-**St**eLa, and our **S**ynthesis tool, with and without *acc*eleration.

Synthesis	Rab	Tem	RPG	T2R	S_{acc}	S
solved	12	0	15	36	**61**	31
best	5	0	11	13	**43**	4
unique	0	0	1	11	**27**	0

Realisability	RPG	RSt	T2R	S_{acc}	S
solved	37	31	54	**61**	31
best	21	0	13	**37**	7
unique	0	0	**11**	9	0

(b) LTL benchmarks.

Name	U	Time (s)	
		S_{acc}	S
arbiter		**2.77**	4.90
arbiter-failure		2.04	**1.98**
elevator		**2.53**	15.92
infinite-race		**1.98**	4.38
infinite-race-u	•	–	–
infinite-race-unequal-1		**6.50**	–
infinite-race-unequal-2		–	–
reversible-lane-r		**7.39**	17.53
reversible-lane-u	•	18.70	**4.54**
rep-reach-obst-1d		**2.47**	9.04
rep-reach-obst-2d		**3.85**	38.51
rep-reach-obst-6d		–	–
robot-collect-v4		**16.51**	–
taxi-service		**39.26**	68.02
taxi-service-u	•	4.14	**3.50**

Evaluation on Novel LTL Benchmarks. We contribute 15 benchmarks with LTL objectives unencodable as deterministic Büchi objectives, i.e., they are theoretically out of scope for other tools. For sanity checking we attempted them on the other tools and validated their inability to decide these problems. We do not include them with the previous benchmarks to ensure a fairer evaluation. Three of these benchmarks could be solved by other tools if infinite-range inputs are used (arbiter, infinite-race, and infinite-race-u), but they fail since incrementing and decrementing requires environment fairness constraints.

These benchmarks involve control of cyber-physical systems such as the elevator from Fig. 2, variations thereof, a reversible traffic lane, and robotic missions, some of which are extensions of literature benchmarks. They also include strong fairness and/or let the environment delay progress for the controller.[7] Table 1b

[7] These benchmarks are also described in detail in Appendix D.1 of [2].

reports how both configurations of our tool handle our novel benchmarks. Column U marks unrealisable problems. The lazy approach outperforms the eager one on just 3 benchmarks out of 15. On 11 problems, acceleration enriches the first abstraction enough to lead immediately to a verdict. We note that solving infinite-race-unequal-1 requires structural refinement, as it allows infinite amount of increments and decrements, but of unequal value, while for literature benchmarks acceleration is enough.

Failure Analysis. Lastly, we discuss four limitations in our approach exposed by our experiments. Section 9 contains more detail on when and why the other tools fail. The first is inherent to synthesis: the Boolean synthesis problem may become big enough to exceed machine resources. A bespoke finite-state synthesis procedure could mitigate this, by relying on the underlying parity game rather than creating fresh problems.

The second is that some unrealisable problems admit no finite counterstrategies in our setting. robot-repair, which no tool solves, is the only such example from literature (we also designed infinite-race-u to be of this kind). Briefly, this involves two stages: a losing loop for which the controller controls exit and (after the loop) a state wherein the goal is unreachable. The environment cannot universally quantify over all predicates (since it controls them), hence no finite counterstrategy exists. But if we construct the dual problem, by swapping objectives between the environment and controller, we do find a strategy for the original environment goal. We are working on automating this dualisation.

The third is that our requirements for when to apply structural refinement may be too strong, and thus some loops go undiscovered. Instead of looking for loops solely in the counterexample prefix, one may instead consider the strongly connected components of the counterstrategy.

Lastly, there are pathological counterexamples, irrelevant to the problem, that involve the controller causing an incompatibility by going to a partition and the environment not being able to determine exactly when dec/increments should force an exit from this partition. This is the main cause of failure for our lazy approach. Modifications to concretisability checking might avoid this issue.

9 Related Work

Before discussing related synthesis approaches, we note that Balaban, Pnueli, and Zuck describe a similar CEGAR approach for infinite-state model checking [4]. From counterexamples they discover ranking functions for terminating loops, and encode their well-foundedness in the underlying fair discrete system, similar to how we encode well-foundedness during acceleration. Our structural refinement is instead more localised to specific loops. We may benefit from the more general ranking abstraction, but it is often easier to prove termination of loops through loop variants rather than ranking functions, which do not admit the same encoding. Interestingly, their approach is relatively complete, i.e. given the right ranking functions and state predicates the LTL property can be ver-

ified. We cannot say the same about our approach, given, as mentioned in the previous section, there are some unrealisable problems we cannot terminate on.

We discuss the exact differences between our setting and that of TSL synthesis [14] and RPG [18]. We then discuss infinite-state synthesis more generally.

TSL and RPG Compared to our Approach. We start by noting that, in the context of linear integer arithmetic, for every possible synthesis problem in TSL or RPG, we can effectively construct an equi-realisable problem in our setting (see Appendix E.1 of [2] for the full details). In both TSL and RPG, variables are partitioned between inputs and outputs. At each step of the game, the environment sets values for all inputs (so, choosing among potentially infinitely-many or continuously-many candidate values in one step) and the controller responds by choosing among a finite set of deterministic updates to its own variables. The environment also initialises *all* variables. Dually, in our setting, players only own Boolean variables and have only a finite set of choices. Then, infinite-range variables are updated based on the joint choice. For all three, repeating single interactions ad-infinitum leads to traces that are either checked to satisfy an LTL formula (TSL and our setting) or to satisfy safety, reachability, or repeated reachability w.r.t. certain locations in the arena/program (RPG). The restriction to finite-range updates hinders the applicability of our approach to linear real arithmetic, given the necessity of repeated uncountable choices there. However, we expect the more novel parts of our approach (liveness refinements and acceleration) to still be applicable in this richer theory. Indeed, we define acceleration in a way that it is also applicable for LRA in Sect. 7.

Infinite-state Arenas. Due to space restrictions, we refer to other work [13,18] for a general overview of existing symbolic synthesis methods, and leave out infinite-state methods restricted to decidable settings, such as pushdown games [37], Petri-net games [15], or restrictions of FO-LTL such as those mentioned in the introduction [30–32]. Such approaches tend to apply very different techniques. We instead discuss methods that take on the undecidable setting, and how they acquire/encode liveness information. We find three classes of such approaches:

Fixpoint Solving. These extend standard fixpoint approaches to symbolic game solving. GenSys-LTL [33] uses quantifier elimination to compute the controllable predecessor of a given set, terminating only if a finite number of steps is sufficient. A similar approach limits itself to the GR(1) setting [23], showing its efficiency also in the infinite setting. rpgsolve [18] takes this further by finding so-called *acceleration lemmas*. It attempts to find linear ranking functions with invariants to prove that loops in the game terminate, and thus it may find fixpoints that GenSys-LTL cannot. This information is however only used in a particular game region. In problems such as robot-tasks, this requires an infinite number of accelerations, leading to divergence. The reliance on identifying one location in a game where a ranking function decreases is also problematic when the choice of where to exit a region is part of the game-playing, or when the ranking needs to decrease differently based on the play's history. The latter would be required in order to scale their approach to objectives beyond Büchi and co-

Büchi. The realisability solver rpg-STeLA tries to bypass the locality limitation by using game templates to identify lemmas that can be used in multiple regions. It does well on benchmarks that were designed for it in a compositional way, but in many other cases, the extra work required to identify templates adds significant overhead. For example, it causes divergence in robot-tasks. As a bridge between program specifications in TSL and the rpg tools, tslmt2rpg [19] translates TSL specifications to RPG while adding semantic information about infinite-range variables that allows it to simplify regions in games. As for rpg-STeLA the analysis of the semantic information often causes a time overhead. Crucial here is the underlying solver, which often times out on quantifier elimination.

Abstraction. Other methods, including ours, attempt synthesis on an explicit abstraction of the problem. A failure witness may be used to refine the abstraction and make another attempt. Some of these methods target games directly [1,20,36]; others work at the level of the specification [8,14,22]. Many of these focus on refining states in the abstraction, a kind of safety refinement, as in the case of the tool raboniel [22]. As far as we know, only temos [8] adds some form of liveness information of the underlying infinite domain. It attempts to construct an abstraction of an LTL (over theories) specification by adding consistency invariants, and transitions. It also uses syntax-guided synthesis to generate sequences of updates that force a certain state change. Interestingly, it can also identify liveness constraints that abstract the effects in the limit of repeating an update u, adding constraints of the form $G(pre \wedge (u W post) \implies F post)$. However, it can only deal with one update of one variable at a time, and fails when the environment can delay u. Moreover, it does not engage in a CEGAR-loop, giving up if the first such abstraction is not realisable.

Constraint Solving. One may encode the synthesis problem into constrained Horn clauses (CHC), and synthesise ranking functions to prove termination of parts of a program. Consynth [5] solves general LTL and ω-regular infinite-state games with constraint solving. However, it needs a controller template: essentially a partial solution to the problem. This may require synthesising ranking functions, and (unlike our approach) makes unrealisability verdicts limited to the given template and thus not generalisable. MuVal [35] can encode realisability checking of LTL games as validity checking in a fixpoint logic that extends CHC. It also requires encoding the automaton corresponding to the LTL formula directly in the input formula, and discovers ranking functions based on templates to enforce bounded unfolding of recursive calls. Contrastingly, we do not rely on templates but can handle any argument for termination.

10 Conclusions

We have presented a specialised CEGAR approach for LTL synthesis beyond the Boolean domain. In our evaluation our implementation significantly outperforms other available synthesis tools, often synthesising a (counter-)strategy before other tools finish checking for realisability. Key to this approach are liveness refinements, which forgo the need for a large or infinite number of safety

refinements. We carefully designed our framework so it can encode spuriousness checking of abstract counterstrategies as simple invariant checking, using loops in counterexamples to find liveness refinements. Another main contribution is the reduction of the complexity of predicate abstraction and synthesis by an exponential, through a binary encoding of related predicates. This also allows to identify well-foundedness constraints of the arena, which we encode in the abstraction through LTL fairness requirements.

Future Work. We believe that symbolic approaches for LTL synthesis and synthesis for LTL over structured arenas [12,17], could significantly benefit our technique. In these, determinisation for LTL properties would have to be applied only to the objective, and not to the arena abstraction. Tool support for these is not yet mature or available. For one such tool [12], we sometimes observed considerable speedup for realisability; however, it does not supply strategies.

Other directions include dealing with identified limitations (see Sect. 8), extending the tool beyond LIA, dealing with infinite inputs automatedly, and applying other methods to manage the size of predicate abstractions, e.g., [21], data-flow analysis, and implicit abstraction, and to make it more informative.

References

1. de Alfaro, L., Roy, P.: Solving games via three-valued abstraction refinement. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 74–89. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74407-8_6
2. Azzopardi, S., Piterman, N., Stefano, L.D., Schneider, G.: Full LTL synthesis over infinite-state arenas (2025). https://arxiv.org/abs/2307.09776
3. Baier, C., Coenen, N., Finkbeiner, B., Funke, F., Jantsch, S., Siber, J.: Causality-based game solving. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12759, pp. 894–917. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81685-8_42
4. Balaban, I., Pnueli, A., Zuck, L.D.: Ranking abstraction as companion to predicate abstraction. In: Wang, F. (ed.) FORTE 2005. LNCS, vol. 3731, pp. 1–12. Springer, Heidelberg (2005). https://doi.org/10.1007/11562436_1
5. Beyene, T.A., Chaudhuri, S., Popeea, C., Rybalchenko, A.: A constraint-based approach to solving games on infinite graphs. In: 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 221–234. ACM (2014)
6. Beyer, D., Keremoglu, M.E.: Cpachecker: A tool for configurable software verification. In: Computer Aided Verification - 23rd International Conference, CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16
7. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuxmv symbolic model checker. In: Computer Aided Verification - 26th International Conference, CAV 2014. LNCS, vol. 8559, pp. 334–342. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_22

8. Choi, W., Finkbeiner, B., Piskac, R., Santolucito, M.: Can reactive synthesis and syntax-guided synthesis be friends? In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, pp. 229–243. PLDI 2022, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3519939.3523429
9. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The mathsat5 SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013. LNCS, vol. 7795, pp. 93–107. Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_7
10. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, London, Cambridge (1999)
11. Di Stefano, L., Azzopardi, S., Piterman, N., Schneider, G.: Software artifact for "full LTL synthesis over infinite-state arenas" (2025). https://doi.org/10.5281/zenodo.15189175
12. Ehlers, R., Khalimov, A.: Fully generalized reactivity(1) synthesis. In: Finkbeiner, B., Kovács, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024. LNCS, vol. 14570, pp. 83–102. Springer (2024). https://doi.org/10.1007/978-3-031-57246-3_6
13. Farzan, A., Kincaid, Z.: Strategy synthesis for linear arithmetic games. Proc. ACM Program. Lang. **2**(POPL) (2017). https://doi.org/10.1145/3158149
14. Finkbeiner, B., Klein, F., Piskac, R., Santolucito, M.: Temporal stream logic: synthesis beyond the bools. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 609–629. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_35
15. Finkbeiner, B., Olderog, E.: Ten years of petri games. In: Jansen, N., Junges, S., Kaminski, B.L., Matheja, C., Noll, T., Quatmann, T., Stoelinga, M., Volk, M. (eds.) Principles of Verification: Cycling the Probabilistic Landscape - Essays Dedicated to Joost-Pieter Katoen on the Occasion of His 60th Birthday, Part III. LNCS, vol. 15262, pp. 399–422. Springer (2025). https://doi.org/10.1007/978-3-031-75778-5_19
16. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer (1997). https://doi.org/10.1007/3-540-63166-6_10
17. Hausmann, D., Lehaut, M., Piterman, N.: Symbolic solution of Emerson-Lei games for reactive synthesis. In: Foundations of Software Science and Computation Structures - 27th International Conference, FoSSaCS 2024. LNCS, vol. 14574, pp. 55–78. Springer (2024). https://doi.org/10.1007/978-3-031-57228-9_4
18. Heim, P., Dimitrova, R.: Solving infinite-state games via acceleration. Proc. ACM Program. Lang. **8**(POPL) (2024). https://doi.org/10.1145/3632899
19. Heim, P., Dimitrova, R.: Translation of temporal logic for efficient infinite-state reactive synthesis. Proc. ACM Program. Lang. **9**(POPL) (2025)
20. Henzinger, T.A., Jhala, R., Majumdar, R.: Counterexample-guided control. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 886–902. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-45061-0_69
21. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, 16-18 January 2002, pp. 58–70. ACM (2002). https://doi.org/10.1145/503272.503279

22. Maderbacher, B., Bloem, R.: Reactive synthesis modulo theories using abstraction refinement. In: 22nd Conference on Formal Methods in Computer-Aided Design, FMCAD 2022, pp. 315–324. TU Wien Academic Press (2022). https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_38
23. Maderbacher, B., Windisch, F., Bloem, R.: Synthesis from infinite-state generalized reactivity(1) specifications. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Software Engineering Methodologies - 12th International Symposium, ISoLA 2024, Crete, Greece, 27-31 October 2024, Proceedings, Part IV. LNCS, vol. 15222, pp. 281–301. Springer (2024). https://doi.org/10.1007/978-3-031-75387-9_17
24. Martin, D.A.: Borel determinacy. Ann. Math. **102**(2), 363–371 (1975). http://www.jstor.org/stable/1971035
25. McMillan, K.L.: Lazy abstraction with interpolants. In: Computer Aided Verification, 18th International Conference, CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer (2006). https://doi.org/10.1007/11817963_14
26. Meyer, P.J., Sickert, S., Luttenberger, M.: Strix: explicit reactive synthesis strikes back! In: Computer Aided Verification - 30th International Conference, CAV 2018. LNCS, vol. 10981, pp. 578–586. Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_31
27. Neider, D., Markgraf, O.: Learning-based synthesis of safety controllers. In: 2019 Formal Methods in Computer Aided Design (FMCAD), pp. 120–128. IEEE (2019). https://doi.org/10.23919/FMCAD.2019.8894254
28. Piterman, N., Pnueli, A.: Temporal logic and fair discrete systems. In: Handbook of Model Checking, pp. 27–73. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_2
29. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL, pp. 179–190. ACM Press (1989)
30. Rodríguez, A., Sánchez, C.: Boolean abstractions for realizability modulo theories. In: Enea, C., Lal, A. (eds.) Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, 17-22 July 2023, Proceedings, Part III. LNCS, vol. 13966, pp. 305–328. Springer (2023). https://doi.org/10.1007/978-3-031-37709-9_15
31. Rodríguez, A., Sánchez, C.: Adaptive reactive synthesis for LTL and LTLF modulo theories. In: Wooldridge, M.J., Dy, J.G., Natarajan, S. (eds.) Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2014, 20-27 February 2024, Vancouver, Canada, pp. 10679–10686. AAAI Press (2024). https://doi.org/10.1609/AAAI.V38I9.28939
32. Rodríguez, A., Sánchez, C.: Realizability modulo theories. J. Log. Algebraic Methods Program. **140**, 100971 (2024). https://doi.org/10.1016/J.JLAMP.2024.100971
33. Samuel, S., D'Souza, D., Komondoor, R.: Symbolic fixpoint algorithms for logical LTL games. In: 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE) pp. 698–709 (2023). https://doi.org/10.1109/ASE56229.2023.00212
34. Schmuck, A.K., Heim, P., Dimitrova, R., Nayak, S.P.: Localized attractor computations for infinite-state games. In: Gurfinkel, A., Ganesh, V. (eds.) 36th International Conference on Computer Aided Verification (CAV). LNCS, vol. 14683, pp. 135–158. Springer, Montreal, QC, Canada (2024). https://doi.org/10.1007/978-3-031-65633-0_7

35. Unno, H., Satake, Y., Terauchi, T., Koskinen, E.: Program verification via predicate constraint satisfiability modulo theories. CoRR abs/2007.03656 (2020). https://arxiv.org/abs/2007.03656
36. Walker, A., Ryzhyk, L.: Predicate abstraction for reactive synthesis. In: 2014 Formal Methods in Computer-Aided Design (FMCAD), pp. 219–226 (2014). https://doi.org/10.1109/FMCAD.2014.6987617
37. Walukiewicz, I.: Pushdown processes: games and model-checking. Inf. Comput. **164**(2), 234–263 (2001). https://doi.org/10.1006/INCO.2000.2894