# Symbolic Finite- and Infinite-state Synthesis

No Author Given

No Institute Given

**Abstract.** Reactive synthesis can automatically craft programs that satisfy a given specification. However it fails to scale to large domains and becomes undecidable in infinite settings. Scalability is mainly affected by the need to enumerate, which can quickly explode the state space and is not viable in the infinite setting. We believe abstraction-refinement techniques can tackle these problems, but existing approaches mainly exploit safety refinements, which do not significantly aid scalability.
In this paper we introduce a novel iterative abstraction-refinement approach to LTL synthesis, starting from a succinct symbolic representation of problems, and exploiting invariant checking to identify correctness of abstract counterstrategies. We introduce the notion of liveness refinements for synthesis, based on identifying concretely terminating loops in abstract counterstrategies. Other approaches exist that identify and add liveness constraints for infinite-state synthesis, however they either do not ensure progress or only apply them in a localised manner. We also present a proof-of-concept prototype, contribute further benchmarks, and show how our approach goes very significantly beyond the state-of-the-art on our and standard LIA benchmarks. We also contribute a set of finite-state benchmarks, and identify a subset whose solution does not depend on the size of the finite domain, and show that our approach's runtime uniquely does not exhibit dependence on the domain size.

**Keywords:** Reactive synthesis · CEGAR · Infinite-state · Liveness.

## 1 Introduction

Reactive synthesis automatically constructs correct systems from temporal specifications. However, this comes at a heavy price: for Linear Temporal Logic (LTL) specifications, it has doubly exponential time complexity in the number of atomic propositions. On-the-fly approaches that work around this issue have shown some efficacy [29], but large problems remain out of reach. Model checking faces a similar problem that is successfully handled through abstraction-refinement loops, wherein an abstraction of the state space is explored and refined, until a verdict is reached [22]. This approach has also had practical success for infinite-state system verification (although this remains undecidable).

There are several works in this direction for synthesis, for both finite- and infinite-state problems [25, 17]. However, these approaches share a limitation in that abstractions are only refined with safety constraints. In contrast, in our work we propose and explore liveness refinements which are surprisingly missing from

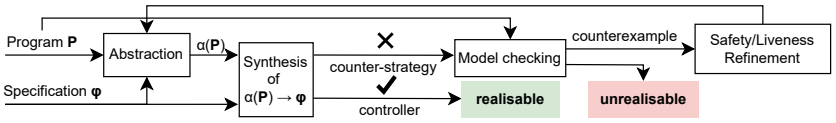


Fig. 1: Workflow of our approach.

previous work in abstraction-refinement approaches to synthesis. The utility of discovering liveness properties of a program for verification is well-known, e.g, this is common in constrained Horn clause solving (e.g., [6]). This has also been proposed to be used in conjunction with predicate abstraction for verification [23, 4], however, to our knowledge, with little adoption in the community.

Our liveness refinements identify looping behaviour that must terminate, and expands the problem classes solvable. Other kinds of non-CEGAR approaches have been independently developed that attempt to exploit this kind of reasoning, e.g. `temos` [10] and `rpgsolve` [20]. However, the former adds such constraints in a way divorced from objectives, and the latter can only deal with deterministic Büchi games and adds constraints in a very localised way.

We are interested in synthesising strategies or counter-strategies for LTL objectives of *programs* that manipulate a finite set of program variables with potentially non-Boolean domains. Programs are machines that update program variable values depending on the program state and Boolean propositions controlled by either the environment or the controller. LTL objectives can include predicates over program variables, and the environment and controller propositions. The closest approach [20] considers only deterministic Büchi games. Other approaches can encode our setting [10, 25, 34] in LTL with non-Boolean domains.

To solve synthesis for this setting we use the workflow depicted in Fig. 1. We maintain a predicate abstraction of the program, initially with predicates extracted from the specification. We attempt to solve the synthesis problem by casting it to an LTL synthesis problem abstracting it. Once a controller is found it is guaranteed to be correct. Novelly, we confirm whether an abstract counterstrategy is not concretisable through invariant checking. If it is spurious we prove this will terminate and return a finite counterexample to the unconcretisability, used as the basis for refinement. Encoding concretisability checking as invariant checking allows to get a complete view of why the counterstrategy fails, giving us short and informative finite counterexamples relevant to the LTL objective.

For safety refinement we use standard sequence interpolation [28]. Our liveness refinement is novel in this context, and is based on adding fairness constraints on the predicate abstraction to ensure certain looping behaviour is terminating. Novelly, we observe that such a constraint can be added when a finite counterexample to concretisability fails in attempting to exercise a lasso in the abstract counterstrategy. From the counterexample we can extract a terminating loop, and analyse this for a corresponding linear ranking function and invariant. To generalise this, we also attempt to generalise the initial condition while maintaining termination. To exclude the counterstrategy, we add a strong fair-

ness constraint to enforce the well-foundedness of the ranking function w.r.t. the invariant, i.e., *the value of the ranking function cannot infinitely often decrease without also infinitely often increasing or violating the invariant.* We term this *ranking refinement.* However, in practice other kinds of termination witnesses may be easier to find, e.g., loop variants. With *structural loop refinement* we can exclude any terminating loop by adding constraints to monitor for execution of the loop, and a fairness constraint that enforces loop termination. Moreover, we show how we can accelerate synthesis by eagerly adding ranking refinements corresponding to well-founded terms in the specification and program.

We consider a set of benchmarks from literature [20], and contribute others with richer goals. Theoretically we show our safety refinements suffice for finite domains, but in practice we show how liveness refinements can uniquely make the problem independent of the finite domain size. For infinite problems our approach solves more double the amount of LIA synthesis problems others can solve. These results are surprising — liveness constraints in general make synthesis harder.

In Section 2 we give the formal background. In Section 3 we illustrate our algorithm on a case study challenging for existing approaches. We introduce our formal setting in Section 4, and reduce the synthesis problem to LTL synthesis in Section 5. In Section 6 we introduce our refinements, and present our evaluation in Section 7. We compare against related work in Section 8, and conclude in Section 9. Proofs can be found in the appendix.

## 2 Background

We use the following notation throughout: for sets $S$ and $T$ such that $S \subseteq T$, we write $\bigwedge_T S$ for $\bigwedge S \wedge \bigwedge_{s \in T \setminus S} \neg s$. Set $T$ is omitted when it is clear.

*Linear Temporal Logic*, LTL($\mathbb{AP}$), is the language over a set of propositions $\mathbb{AP}$, defined as follows, where $p \in \mathbb{AP}$: $\phi \stackrel{\text{def}}{=} \mathbf{tt} \mid \mathbf{ff} \mid p \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid X\phi \mid \phi U \phi$. See [32] for the standard semantics. For $w \in (2^{\mathbb{AP}})^\omega$, we write $w \models \phi$ or $w \in L(\phi)$, when $w$ satisfies $\phi$.

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, $\Sigma_{in}$ the set of input events, $\Sigma_{out}$ the set of output events, $\rightarrow\colon S \times \Sigma_{in} \mapsto S$ the complete deterministic transition function, and $out : S \mapsto \Sigma_{out}$ the labelling of each state with an output event. 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 $S$, $s_0$, $\Sigma_{in}$, and $\Sigma_{out}$ are as before and $\rightarrow\colon S \times \Sigma_{in} \mapsto \Sigma_{out} \times S$ 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 $w$.

We cast our synthesis problem into the *LTL reactive synthesis problem*, which calls for finding a Mealy machine that satisfies a given specification.

**Definition 1 (LTL Synthesis).** *A specification $\phi$ 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 $\phi$.*

*A specification $\phi$ 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 $\phi$.*

Note that the duality between the existence of a strategy and counterstrategy follows from the determinacy of turn-based two-player $\omega$-regular games [27]. We know that finite-state machines suffice for synthesis from LTL specifications [33].

To be able to represent large or infinite synthesis problems succinctly we require the use of variables. We introduce the notion of a theory over a set of variables, which includes a set of predicates and updates over these variables.

A *theory* consists of a set of terms and predicates over these. Atomic terms are constant values ($\mathbb{C}$) 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 s.t. any variable $v$ appearing in $t$ is replaced by a fresh variable $v_{prev}$.

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

Throughout we focus on *Linear Integer Arithmetic* (LIA). We allow variables with integer, natural, or Boolean type. We allow terms formed from addition ($+$) or subtraction ($-$) operators, and predicates comparing terms with $\leq$. We also allow predicates using $<$, $\geq$, $>$, and $=$, as macros for the expected definitions.

## 3 Informal Overview

Fig. 2 illustrates a problem in our setting that challenges existing approaches (see Section 7). On the right, we have an automaton representing a partial design for an elevator, with transition labels $g \mapsto U$ representing that update $U$ is performed when $g$ is true. The value of a variable remains the same when not

$\mathbb{V} = \{target : [0..9] = 0, floor : [0..9] = 0\}$
$\mathbb{E} = \{env\_inc, door\_open\}$
$\mathbb{C} = \{up, down\}$
**Assumptions:**
A1. $GF\, door\_open$
A2. $GF\neg door\_open$
**Guarantees:**
G1. $GF floor = target$
G2. $G(door\_open \implies (up \iff down))$

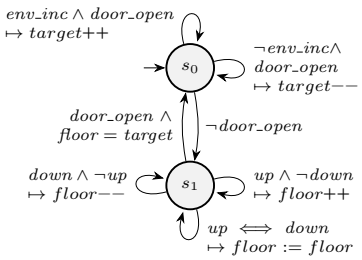


Fig. 2: Elevator example.

mentioned. Technically we represent such programs as functions. On the left, we have an LTL objective we desire to synthesise a controller for.

At state $s_0$ the environment can set a target by controlling its boolean variables ($\mathbb{E}$) to increase or decrease *target*. Once a target has been decided, the environment closes the elevator door (*door_open*), and the program transitions to $s_1$. At $s_1$, the system ($\mathbb{C}$) can force the elevator to go up or down one floor, or remain at the same floor. This is not a good elevator: it may move with the door open and it may never reach the target floor. We desire to control it so that the former never occurs (G2), and the target is reached infinitely often (G1). 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).

We briefly look at how our approach solves this problem. We show how safety refinement suffices, but how liveness refinement accelerates synthesis.

*Predicate Abstraction* We initially soundly abstract the program $P$ in terms of the predicates appearing the specification $\phi = (A_1 \wedge A_2) \implies (G_1 \wedge G_2)$, and the predicates and Boolean variables of the program (i.e., here the states in the automaton). That is, $Pr = \{floor \leq target, target \leq floor, s_0, s_1\}$. Note we always resolve LIA predicates to a form using only $\leq$. This predicate abstraction considers all the possible combinations of environment and controller 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*, and then by satisfiability checking we can identify that two states are possible next: $p'_1 = s_1 \wedge floor = target$ or $p'_2 = s_1 \wedge floor < target$.

The program abstraction is an LTL formula $init \wedge \alpha(P, Pr)$ of the form $G(\bigvee_{a \in abtrans} a)$, 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.* Given such a sound abstraction, we create the abstract formula $\alpha(P, Pr) \implies \phi$, giving the environment control of the predicates. When this is realisable, the controller also works for the concrete program. In our example this is not realisable. Consider that when $p$, the environment can always choose to avoid making $floor = target$ true in the abstraction.

*Counterstrategy Concretisability.* When a problem is unrealisable we can always find an abstract counterstrategy *Cs*. To check whether it is not spurious, we compose $P$ with *Cs* and use model checking (namely invariant checking) to check the invariant that the predicate guesses of *Cs* are always true of the program. Consider that *Cs* admits a finite counterexample *ce* wherein the environment initially increments *target*, then moves to $s_1$, and the controller increments floor, but the environment maintains $\mathit{floor} < \mathit{target}$, in error.

*Safety Refinement.* With *ce* we can refine the abstraction with safety constraints, through interpolation. This gives us predicates, namely $\mathit{target} - \mathit{floor} = 1$, that when added to the abstraction refine it to exclude variants of *ce*. Repeating this, we find predicates that enumerate the state space, and eventually determine the problem realisable. However, this undesirably adds many propositions; while for integer variables this does not terminate.

*Liveness Refinements.* Consider that once *Cs* guesses that $\mathit{floor} < \mathit{target}$, it remains in states where $\mathit{floor} < \mathit{target}$ is true. Essentially, *Cs* exercises the loop `while(floor < target) floor := floor + 1;`, and the environment believes it is non-terminating. This is disproven by *ce* with a precondition $\mathit{target} = 1$ and $\mathit{floor} = 0$. Preconditions for termination can be expanded through existing approaches (e.g., [13]). In this case *true* suffices.

We can encode termination of this loop in the abstraction by monitoring for it and a strong fairness constraint to force execution of the loop to eventually stop. We term this *structural loop refinement*. Monitoring is done with extra variables for each step of the loop, and transition constraints over predicates corresponding to the updates, precondition, and iteration condition. Note this is not tied to a specific region in the program.

Here however we manage to find a ranking function, i.e., $r = \mathit{floor} - \mathit{target}$. A ranking function is a function to a range where the order relation is well-founded, that is, there are no infinite-descending chains, and decreases as the loop runs. In this case $\mathit{floor} - \mathit{target}$ cannot go below $-9$, with no need for an invariant. In the infinite case, we would need to add the invariant $\mathit{floor} - \mathit{target} \geq 0$. Then, we use $r$ by encoding its well-foundedness as an LTL formula. First, we track decreases and increases of $r$ in the abstraction, with $r_{inc} \stackrel{\text{def}}{=} r_{prev} < r$ and $r_{dec} \stackrel{\text{def}}{=} r_{prev} > r$ where $r_{prev}$ represents the value of $r$ in the previous state. Then, we capture its well-foundedness with the liveness constraint: $GFr_{dec} \implies GFr_{inc}$. Refining the abstraction with this makes *Cs* not viable. After some safety refinements, we discover another ranking function: $\mathit{target} - \mathit{floor}$, allowing us to find a controller.

*Acceleration.* The described process learns predicates that are not necessary to solve the problem until lasso counterexamples are found. To accelerate the process, we can add ranking refinements that restrict the environment in the setting of predicates used in the problem statement. Consider $p \stackrel{\text{def}}{=} \mathit{target} \leq \mathit{floor}$, which we normalise to $0 \leq \mathit{floor} - \mathit{target}$. Since $r \stackrel{\text{def}}{=} \mathit{floor} - \mathit{target}$ is a well-founded function we can add a corresponding ranking refinement. For non-well-founded ones, if they only change with discrete values in the program, we can simply add the predicate as an invariant, writing $GFr_{dec} \implies GF(r_{inc} \vee \neg p)$. Using this acceleration gives us sufficient information to solve the problem immediately.

## 4 Synthesis Setting

Part of our main contribution is our special setting that combines programs and LTL objectives, that is unlike existing approaches. 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 propositions, respectively controlled by the environment ($\mathbb{E}$) and the controller ($\mathbb{C}$). Then our specifications are LTL formulas over these variables, $\phi \in LTL(\mathbb{E} \cup \mathbb{C} \cup Pr_\phi)$, where $Pr_\phi \subseteq \mathcal{P}r(V)$. LTL formulas talk about a *program* whose state is captured by the value of $V$, and modifies its state depending on environment and controller behaviour. Programs are deterministic; we model non-determinism with additional environment variables. This allows to encode concretisability checking as invariant checking, rather than the significantly more complex CTL$^*$ model checking.

**Definition 2 (Program).** *A program $P$ over variables $V$ is a tuple $\langle V, val_0, \delta \rangle$. $val_0 \in Val(V)$ is the initial valuation, and $\delta : Val(V) \times 2^{\mathbb{E}\cup\mathbb{C}} \mapsto Val(V)$ is the deterministic transition function. A program is finite when every $v \in V$ is finite.*

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

*To avoid the inherently large/infinite nature of $\delta$, given a finite set of predicates $Pr \subseteq \mathcal{P}r(V)$, and a finite set of updates $U \subseteq \mathcal{U}(V)$, we consider symbolically represented programs with the symbolic transition function as a partial function $\delta_{sym} : \mathbb{B}(\mathbb{E} \cup \mathbb{C} \cup Pr) \nrightarrow U$, with finite domain, s.t. for all $val \in dom(\delta)$ and for every $E \subseteq \mathbb{E}$ and $C \subseteq \mathbb{C}$ there is always a single $f \in dom(\delta_{sym})$ s.t. $f(val, E \cup C)$ , and for the update $U_f = \delta_{sym}(f)$ we have $U_f(val) = \delta(val, E \cup C)$.*

During our workflow, the words of our abstract synthesis problem may have a different domain that those of the program. We define these as *abstract words*, and identify when they are concretisable in the program.

**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 $\mathbb{E}'$, s.t. $\mathbb{E} \subseteq \mathbb{E}'$, 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 a program $P$ when $L(P) \cap \gamma(a)$ is non-empty.*

With this definition we are ready to define what it means for a specification to be realisable or unrealisable modulo a program, in terms of concretisability of the language of abstract controllers and counterstrategies.

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

**Definition 5 (Unrealisability modulo a Program).** *A counterstrategy to the realisability of a formula $\phi$ in $LTL(\mathbb{E} \cup \mathbb{C} \cup \mathcal{P}r_\phi)$ modulo a program $P$, is a Moore Machine $Cs$ with output $\Sigma_{out} = 2^{\mathbb{E} \cup \mathcal{P}r_\phi}$ and input $\Sigma_{in} = 2^{\mathbb{C}}$ s.t. every abstract trace $t$ of $Cs$ is concretisable with respect to $P$ and violates $\phi$.*

## 5 Abstract to Concrete Synthesis

We attempt to solve the presented synthesis problem through an abstraction-refinement loop. For that purpose, we construct LTL formulas that capture sound abstractions of the program and combine them with the LTL specification. We fix the set of predicates that appear in the LTL formula $\phi$ as $\mathcal{P}r_\phi$. The abstraction may include fresh predicates and environment variables. We fix the set of predicates and environment variables 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 6 (Abstraction).** *Formula $\alpha(P)$ in $LTL(\mathbb{E}' \cup \mathbb{C} \cup \mathcal{P}r)$ is an abstraction of program $P$ if for every $w \in L(P)$ there is $a \in L(\alpha(P))$ s.t. $w \in \gamma(a)$.*

Given an abstraction of the program $\alpha(P)$, we can construct a corresponding sound LTL synthesis problem, $\alpha(P) \implies \phi$, giving the environment control of the predicates in $\alpha(P)$. We get three possible outcomes from attempting synthesis with 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 that allow us to capture when we can determine realisablity (1) and unrealisability (2). For case (3) we refine the abstraction to avoid the counterstrategy.

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

However, an abstract counterstrategy *Cs* may contain traces that are not concretisable, since abstractions are sound but not complete. To analyse *Cs* for concretisability, we define a simulation relation between the concrete program variable state and *Cs*'s state, capturing when each word of *Cs* is concretisable.

**Definition 7 (Counterstrategy Concretisability).** *Consider a counterstrategy as a Moore Machine $Cs = \langle S, s_0, \Sigma_{in}, \Sigma_{out}, \rightarrow, out \rangle$, and a program $P$, showing $\alpha(P) \implies \phi$ is unrealisable, where $\Sigma_{in} = 2^{\mathbb{C}}$ and $\Sigma_{out} = 2^{\mathbb{E}' \cup \mathcal{P}r}$.*
Concretisability *is defined through the simulation relation $\preceq_P \subseteq Val \times S$:*
*For every $val \preceq_P s$, where $out(s) = E \cup ST \cup TR$, it holds that: for every $C \subseteq \mathbb{C}$, let $val_C = \delta(val, (E \cap \mathbb{E}) \cup C)$, $s_C$ be such that $s \xrightarrow{C} s_C$, and $TR_C$ be the transition predicates in $out(s_C)$, then $val \models \bigwedge ST$, $(val, val_C) \models \bigwedge TR_C$, and $val_C \preceq_P s_C$.*
*Cs is* concretisable *w.r.t. $P$ when $val_0 \preceq_P s_0$, for $P$'s initial valuation $val_0$.*

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

**Algorithm 1:** Abstraction-Refinement based Synthesis Algorithm.

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

One of the main novel features of our setting is that it allows us to encode abstract counterstrategy concretisability as a model checking problem. The system for this model checking problem is the composition of the counterstrategy and the program, while the property to check is the simple invariant requiring the predicates chosen by the environment to hold on the program state.

**Proposition 1.** *Counterstrategy concretisability can be encoded as model checking, and terminates for finite problems and non-concretisable counterstrategies.*

This invariant checking gives us the advantage of finding early causes of concretisability failure as finite counterexamples. These counterexamples represent paths in the abstraction that would violate the specification, that are however not possible in the program. We thus use them as the basis for refinement.

**Proposition 2.** *If a counterstrategy is not concretisable, there is a finite counterexample* $a_0, ..., a_k \in (2^{\mathbb{E} \cup \mathbb{C} \cup \mathcal{P}r})^*$*, s.t. concretisability fails locally only on* $a_k$*.*

*Synthesis Algorithm* We present our high-level algorithm in Alg. 1 as a prelude to the next sections. This algorithm takes a program $P$ and a formula $\phi$, and maintains throughout a set of predicates $\mathcal{P}r$ and an LTL formula $\psi$. $\mathcal{P}r$ is used as the basis of our abstraction (line 4). When the abstract problem is realisable a controller is returned (line 5), while if its unrealisable and the counterstrategy is concretisable a counterstrategy is returned (line 7). When it is not concretisable, we refine the abstraction to exclude this counterstrategy (line 8), and extend $\mathcal{P}r$ with the learned predicates, and $\psi$ with the new constraints (line 9). Note the algorithm does not terminate if un/realisability is not determined, which may be the case for infinite programs.

**Predicate Abstraction** We use an abstraction $\alpha(P, Pr)$ of the program $P$ in terms of a set of predicates $\mathcal{P}r$: this set includes at least the set of predicates in the desired formula $\phi$. For synthesis acceleration, we also add Boolean variables used by $P$, and later also transition predicates acquired from analysing $\phi$. The abstraction focuses on abstracting the symbolic transition relation $\delta_{sym}$ of the program in terms of $\mathcal{P}r$, such that every symbolic transition has corresponding abstract transitions. We rely on SMT checking to compute this abstraction, and construct it incrementally. By memorising the concrete transitions corresponding

to an abstract transition, we refine the latter with the new predicates. This optimisation significantly improves performance. Other optimisations include identifying invariant postconditions of transitions, and reducing the predicate set up to equivalence. Moreover, given that we have an initial valuation, we give an exact abstraction for the initial transition. This standard construction [18], shown in Section B.1, gives a program abstraction:

**Proposition 3.** *The formula $\alpha(P, Pr)$ is an abstraction of $P$.*

## 6 Refinement

We present three refinements, all of which identify new relevant predicates and/or new LTL constraints governing the way these predicates change over time. These arise from analysing the counterexample due to Prop. 2. The first is a safety refinement, which excludes non-viable abstract finite word prefixes. The other two are liveness refinements that exclude non-viable abstract infinite word suffixes. We claim that each refinement to the predicate abstraction, retains soundness, and ensures similar counterexamples will not be re-encountered (progress). We state and show these for each refinement in the appendix, for lack of space.

### 6.1 Safety Refinement

Consider a counterstrategy $Cs$, 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 program state and $Cs$'s desired predicate state. It is standard how to use interpolation to determine sufficient state predicates to make $Cs$ non-viable, we briefly describe this to make the paper self-contained. Let $p_i = \bigwedge\!\!\!\bigwedge_{\mathcal{P}r}(a_i \cap \mathcal{P}r)$, 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, 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 program 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 [28], 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, and 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 $I_i$. Then adding $I(ce)$ to the abstraction maintains soundness, by Prop. 3, and allows for progress by making the counterstrategy unviable in the fresh abstract problem.

Significantly, safety refinement is sufficient for finite programs; this is not clear for other approaches. Baier et al. show it for their approach to reachability games [3]. We cannot guarantee termination for the infinite setting.

**Theorem 3.** *Alg. 1 with safety refinement terminates on finite programs.*

### 6.2 Liveness Refinement

Safety refinement in general may require refinement until a solution is found, which leads to non-termination even for simple infinite problems (see Section 3). To potentially avoid full enumeration we propose *liveness refinements*. Our main insight is that spurious lassos in the counterstrategy can be used to learn liveness properties of the program that exclude infinite looping behaviour.

*Lassos and Loops.* We say a counterexample $ce = a_0, a_1, \ldots, a_k$ induces a lasso in $Cs$ when it corresponds to a path $s_0, \ldots, s_k$ in $Cs$, where $s_k = s_l$ for some $0 \leq l < k$. We focus on the last such $l$. Here, for simplicity, we require that concretisation failed due to a wrong state predicate guess. We split the counterexample into two parts: a stem $a_0, \ldots, a_{l-1}$, and a loop $a_l, \ldots, a_{k-1}$. We further consider the corresponding applications of $\delta_{sym}$, $f_l \mapsto U_l, \ldots, f_{k-1} \mapsto U_{k-1}$, and the program state at step $l$: $val_l$.

```
void main():
    V = *
    assume val_l
    while ⫛(a_l ∩ Pr)
        assume f_l
        V = U_l(V)
        ...
        assume f_{k-1}
        V = U_{k-1}(V)
```

Fig. 3: *ce* loop.

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

**Definition 8 (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 $i$ s.t. $0 \leq i < len(vals)$, where $n = len(body)$, 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.*

We attempt to generalise the loop beyond the specific counterexample.

**Definition 9 (Weakening).** *A loop $l_1 = \langle V_1, pre_1, iter\_cond_1, body_1 \rangle$* is weaker than *loop $l_2 = \langle V_2, pre_2, iter\_cond_2, body_2 \rangle$ when: 1. $V_1 \subseteq V_2$; 2. $body_1$ and $body_2$ have the same length; 3. for $v \in V_1$ and $0 \leq i < len(body_1)$, $U_i^{l_1}(v) = U_i^{l_2}(v)$; and 4. for $w_2 \in L(l_2)$ there is $w_1 \in L(l_1)$ s.t. $w_2$ and $w_1$ agree on $V_1$. A weakening is* proper *if both $L_2$ and $L_1$ terminate.*

There are several weakenings we perform in an attempt to get a proper weakening. 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 [12] of $iter\_cond$. To generalise beyond the stem of *ce*, we further weaken $pre$ to get a weaker sufficient precondition for termination [13]. The technical details of these existing techniques are out scope here, and we refer the reader to the respective references.

As the basis of the refinements we assume a terminating loop $l(ce)$ that is a proper weakening of the found concrete loop. Note in a weakened loop, elements of the body are not necessarily any longer exact copies of transitions

in the symbolic programs. We present an approach that uses ranking functions to refine the abstraction, a weaker loop is not necessary here but helpful. Then we present an approach that monitors for execution of the loop, which requires a strictly weaker loop to be more useful than safety refinement.

*Ranking Refinement.* It is known that each terminating loop has an associated *ranking function*, i.e. a well-founded function that witnesses the termination. We define these as functions from the program variable state to some well founded range (we say that $R$ is well founded when the relation $< \subseteq R \times R$ is well founded).

**Definition 10 (Ranking Function).** *A function $r : Val(V) \to R$, for a well-founded range $R$ is said to be a* ranking function *for a loop $l$ iff for every execution $val_0, \ldots, val_{k-1}, val_k \in L(l)$ then $r(val_{k-1}) > r(val_k)$.*[1]

Relying on the well-foundedness of $< \subseteq R \times R$, it is sound to require that if there are infinitely many decreases in a run then there are also infinitely many increases. For example, given a range of floors that is bounded below, we can only go down (decrease) so far before reaching the bottom and not being able to go lower, without an accompanying infinite number of going upwards (increase). We encode this in LTL, with appropriate new predicates.

**Definition 11 (Ranking Abstraction).** *For a ranking function $r$, let $r^{dec} = r_{prev} > r$ and $r^{inc} = r_{prev} < r$, then we define the ranking abstraction $\alpha_{rk}(P, r) \stackrel{\text{def}}{=} GF(r^{dec}) \implies GF(r^{inc})$.*

Termination checking tools may provide ranking functions *subject to an invariant*. That is, there exists a subset $Val'$ of $Val(V)$ (expressed as a predicate $inv$ over the variables of the loop in the theory of interest) such that: (1) every valuation $val$ reachable in the loop satisfies $val \in Val'$ and (2) the relation $<$ when restricted to $r(Val')$ is well founded (while it is not well founded in general). For example, if the value of an integer $x$ is always non-negative in the loop and only decreases, then $x$ could be supplied as a ranking function subject to the invariant $x \geq 0$. In such a case, we set $\alpha_{rk}(P, r)$ to $GF(r^{dec}) \implies GF(r^{inc} \vee \neg inv)$. This is sound and allows for progress for the same reasons.

This refinement eliminates the counterstrategy, since it eliminates every word with a suffix with strictly decreasing values of $r$, in particular $ce$.

Beyond learning ranking functions from counterexamples, we can be more eager by identifying well-founded terms in the synthesis problem. Consider a satisfiable predicate $p = t_1 \leq t_2$. When $t_2 - t_1$ is well-founded modulo the invariant $p$, which is true for any LIA term $t$, and the program contains updates that decrement this term, we refine with the corresponding abstraction. Experimentally we later show this can significantly accelerate synthesis.

*Structural Loop Refinement.* In practice termination witnesses other than ranking functions, e.g., loop variants, are easier to find. We present a more general refinement that monitors for a loop and enforces its termination.

[1] In practice, we relax this to allow a finite amount of stuttering. We do not formalise this due to lack of space.

For every $(g_i, U_i)$ in the loop body we define the following formulas. We define $cond_0$ as $iter\_cond \wedge g_0$, and for all other $i$, $cond_i \stackrel{\text{def}}{=} g_i$. For $U_i$, of the form $v^0 := t^0, \ldots, v^j := t^j$, we define $p_i$ as $v^0 = t^0_{prev} \wedge \ldots \wedge v^j = t^j_{prev}$. We further define a formula that captures the program 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 12 (Structural Loop Refinement).** *Let $l = \langle pre, iter\_cond, body \rangle$ be a terminating loop, the corresponding condition, transition and stutter formulas respectively be $cond_0, ..., cond_{n-1}$, $p_0, \ldots, p_{n-1}$ and $st$. Assume fresh variables corresponding to each step in the loop $inloop_0, \ldots, inloop_{n-1}$, $inloop_n = inloop_0$, and $inloop = inloop_0 \vee \ldots \vee inloop_{n-1}$.*

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

1. *Initially we are not in the loop, and throughout can only be in one loop step: $\neg inloop \wedge \bigwedge_i G(inloop_i \implies \neg \bigvee_{j \neq i}(inloop_j))$;*
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:*

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

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.

*Refinement Algorithm.* Alg. 2 shows our refinement algorithm. Given a counter-strategy with a counterexample to concretisability (line 2), we check if liveness refinement is applicable (line 3). If it is, we first attempt to weaken the loop (line 5), and try to apply ranking refinement (lines 6-9). If this is not applicable, and the loop was successfully weakened we apply the structural loop refinement (line 10-12). Otherwise, we apply safety refinement (line 13).

## 7 Evaluation

We implemented this approach in a tool, `sweap`. We use existing state-of-the-art tools for sub-routines: Strix [29] for LTL synthesis, nuXmv [9] for model checking, MathSAT [11] for interpolation and SMT checking, and CPAchecker [7] for termination checking. The implementation differs slightly from the algorithm presented for practical reasons. When finding terminating loops we apply structural loop refinement due to CPAchecker returning loop variants rather than

**Algorithm 2:** Refinement Algorithm

```
1  Function refinement(P, Cs):
2      ce := counterexample_to_concretisability(P, Cs)
3      if state_predicate_mismatch(ce, Cs) and matches_lasso(ce, Cs) then
4          pre_ce, post_ce, body_ce := extract_loop(ce)
5          pre, post, body := weaken_loop(P, ce)
6          if found_ranking_function(pre, post, body) then
7              r := ranking_function(pre, post, body)
8              ψ := GF r_dec ⟹ GF r_inc
9              return {r_dec, r_inc}, ψ
10         else if (pre, post, body) strictly weaker than (pre_ce, post_ce, body_ce) then
11             ψ := structural_refinement(pre, post, body)
12             return atomicProps(ψ), ψ
13     return sequence_interpolants(ce), true
```

ranking functions as we define them. We only apply ranking refinements as described to well-founded terms identified in the LTL and program.

*Benchmarks.* Our tool targets problems in LIA. We thus use LIA benchmarks from Heim and Dimitrova [20], which includes examples from other authors [25, 31]. These benchmarks are defined as deterministic programs, with safety, reachability, or Büchi objectives on the program state labels. `rpgsolve` allows arbitrary integers as input: we model with extra steps that allow the environment to arbitrarily but finitely increase or decrease the value of a variable. We also compile finite-state benchmarks from [25], and finite-variants of our benchmarks.

We contribute other benchmarks, most with richer objectives. `elev.-w-door` is Figure 2. In `batch-arbiter-r`, repeatedly, the environment makes a number of requests which the controller must eventually grant, but grants may be delayed before succeeding. `rep-reach-obst.-*` extends `rep-grid-reach-*` with moving obstacles that delay the controller from progressing. `taxi-service` extends `elev.-w-door` to 2D space and adds obstacles. `rev-lane-*` describes a reversible traffic lane whose entry points can be shut or opened by the controller. Traffic initially flows in one direction. Whenever the environment asks to change the flow, the controller must eventually effect it without risking a car crash. The unrealisable version (`-u`) allows cars to not exit the lane. `xyloop` is a variant of `robot-grid-reach-2d`, allowing varying $y$ only when $x = 0$, but when reached the environment sets a new value for $x$. The goal is to reach $(0, 0)$.

There are other benchmarks we do not include that are trivial to solve [10, 17] (most are realisable in the first abstraction). For another set of reachability finite-state games[3], e.g., the Game of Nim, we do not show results. None of the tools we compare against scale well on these, but reachability-only approaches scale much better on them [16, 3, 15], at least for realisability checking.

*Compared tools.* We consider two different versions of our tool, one employing acceleration ($S_{acc}$), and one that is lazy ($S$). The latter adds initially predicates present in the LTL, the former also adds predicates in the program and ranking refinements for each well-founded term in these predicates. We had two requirements for tools to compare against: (1) ability to express most of our benchmark set; and (2) at least partial ability to synthesise counter/strategies. We thus

compare against Raboniel [25], `temos` [10], and `rpgsolve` [20]. The latter does not handle full LTL, and does not attempt to synthesise counterstrategies. However, it exploits ranking functions but in a more localised manner, making it relevant. We do not expect Raboniel to solve most of the benchmarks, given it mainly performs forms of safety refinement. However, `temos` employs SyGuS to synthesize small programs that the controller can exploit to reach a certain state, which could compete with our tool. `temos` only guarantees correct controllers.

There are other tools we do not compare against because they handle only realisability checking and do not start from LTL formulas, namely GenSys-LTL [34] and MuVal [35]; `rpgsolve`, which we do compare with, reports better performance than either. Termite [36] is a domain-specific tool that takes a similar safety refinement approach for driver synthesis. Another approach for full LTL does not have a publicly available tool [17], but we believe this is superseded by Raboniel. `consynth` [5] requires a user-provided controller template, making it a different problem. There are other tools that handle only reachability or safety goals, some targeting only finite games [24, 30, 8]. Some can handle finite safety/reachability games that require (almost) full enumeration to find a solution much quicker than us [16, 3], but all fail on infinite-state benchmarks where safety refinement is not enough, since these require reasoning about termination. Another approach can potentially handle these, since they reduce a kind of reachability game to a system of constrained Horn clauses, but we are not aware of a publicly available tool [15]. There are other approaches for safety games that seem outperformed by `rpgsolve` [31, 26]. We note many of these other tools can also handle LRA, unlike our current implementation.

The experiments were run on a Linux virtual machine with 24 GiB of memory, hosted on a Windows Server 2022 PC equipped with an Intel i5-6500 CPU.

*Results.*[2] We report the results in Table 1. For the infinite-state experiments, the results are varied. `temos` does not solve any benchmark. It synthesises fairness constraints divorced from the objective, and thus is better suited for problems where there is little interaction between the environment and the controller, but here we consider more sophisticated control problems. Raboniel's performance shows how safety refinements are not enough for many interesting problems, but can be successful for small problems (safety), and unrealisable problems with small counterexamples (`heim-fig7`). When `rpgsolve` succeeds it is quicker than either of our configurations, while it can give realisability results where we timeout. In fact it wins for safety, and ignoring Full LTL problems and restricting attention to realisability, it performs best. However, it often times out while constructing controllers, and is unable to construct counterstrategies. We expect this kind of behaviour for other realisability checking approaches [34, 35]. Our different configurations solve the most problems (even if we ignore Full LTL problems). A portfolio approach would be more effective (solving 24 problems), given there are solved problems not in common between the two.

We observe that when the problem is simple (see the safety examples or `robot-cat-u-2d`) acceleration seems to create new hurdles. Having to construct

[2] In the appendix find data about the amount and kind of refinements performed.

Table 1: Experiments comparing synthesis time (sec.) for **R**aboniel, **T**emos, **RPG**solve, and **S**weap, for benchmarks with Safety, Reachability, Deterministic Büchi, or full LTL goals. – denotes timeout (10 mins), n/a an unsupported goal, unk an inconclusive result, ~~x~~ a correct verdict in x seconds where synthesis timed out or was not supported. Column **W** indicates whether the system (S) or environment (E) wins. The best times are set in bold. * indicates infinite problems solvable in general with only safety refinements.

(a) Infinite-state experiment results.

| G. | Name | W | R | T | RPG | $S_{acc}$ | S |
|---|---|---|---|---|---|---|---|
| Safety | box* | S | 1.1 | unk | **0.5** | 7.7 | 2.9 |
| | box-limited* | S | 2.7 | – | **0.7** | 22.2 | 5.0 |
| | diagonal* | S | 9.8 | unk | **0.5** | 47.7 | 3.9 |
| | evasion* | S | 5.8 | – | **0.8** | – | 15.4 |
| | follow* | S | – | – | **1.1** | – | 229.0 |
| | square* | S | 136.9 | – | **0.7** | 83.3 | 48.9 |
| Reachability | robot-cat-r-1d | S | – | – | ~~79.5~~ | **41.3** | – |
| | robot-cat-u-1d* | E | – | – | ~~75.5~~ | 48.6 | **4.7** |
| | robot-cat-r-2d | S | – | – | – | – | – |
| | robot-cat-u-2d* | E | – | – | – | – | **22.6** |
| | robot-grid-reach-1d | S | – | unk | **1.4** | 2.9 | 8.6 |
| | robot-grid-reach-2d | S | – | unk | ~~0.8~~ | **7.0** | 146.1 |
| | heim-double-x* | S | – | unk | ~~1.1~~ | – | – |
| | xyloop | S | – | unk | – | **3.0** | – |
| Det.Büchi | robot-grid-commute-1d | S | – | unk | ~~2.0~~ | **12.8** | – |
| | robot-grid-commute-2d | S | – | – | ~~12.1~~ | – | – |
| | robot-resource-1d | E | – | unk | ~~25.0~~ | **48.8** | 122.0 |
| | robot-resource-2d | E | – | – | ~~4.8~~ | – | – |
| | heim-buechi | S | unk | – | **4.1** | 437.7 | – |
| | heim-fig7* | E | **1.5** | unk | – | 3.3 | 2.7 |
| | batch-arbiter-u* | E | unk | – | ~~6.7~~ | **3.7** | 4.0 |
| Full LTL | reversible-lane-r | S | – | – | n/a | **9.5** | 48.5 |
| | reversible-lane-u* | E | – | – | n/a | 30.9 | **5.7** |
| | batch-arbiter-r | S | – | – | n/a | **3.2** | 9.5 |
| | elevator-w-door | S | – | – | n/a | **4.2** | 47.8 |
| | rep-reach-obst.-1d | S | unk | unk | n/a | **3.2** | 9.5 |
| | rep-reach-obst.-2d | S | unk | unk | n/a | **16.8** | 74.5 |
| | taxi-service | S | – | – | n/a | – | **228.1** |
| num. solved (out of 28) | | | 6 | 0 | 8 | 20 | 20 |

(b) Finite-state experiment results.

| Name | W | range | R | T | RPG | $S_{acc}$ | S |
|---|---|---|---|---|---|---|---|
| elevator simple | S | 0..5 | 9.0 | – | 7.1 | 4.0 | **3.7** |
| | | 0..10 | 164.2 | – | 32.8 | 5.3 | **4.4** |
| | | 0..50 | – | – | – | – | – |
| elevator signal | S | 0..5 | – | – | **5.4** | 9.9 | – |
| | | 0..10 | – | – | 11.0 | **10.3** | – |
| | | 0..50 | unk | – | – | **9.9** | – |
| rob-grid reach-1d | S | 0..5 | 3.3 | unk | **1.5** | 3.3 | 5.7 |
| | | 0..10 | – | unk | **2.3** | 3.3 | 5.8 |
| | | 0..50 | – | unk | 10.9 | **3.3** | 5.8 |
| rob-grid reach-2d | S | 5×5 | – | – | **3.8** | 6.7 | 14.5 |
| | | 10×10 | – | – | 13.6 | **6.5** | 15.1 |
| | | 50×50 | – | – | – | **7.0** | 14.8 |
| batch-arbiter-u | E | 0..5 | unk | – | ~~2.8~~ | **3.6** | 6.7 |
| | | 0..10 | unk | – | ~~3.9~~ | **3.8** | 6.6 |
| | | 0..50 | unk | – | ~~20.8~~ | **3.8** | 6.7 |
| batch-arbiter-r | S | 0..5 | – | – | n/a | **3.5** | 6.5 |
| | | 0..10 | – | – | n/a | **3.4** | 6.2 |
| | | 0..50 | – | – | n/a | **3.5** | 6.3 |
| reversible -lane-r | S | 0..5 | – | – | n/a | **19.8** | 59.2 |
| | | 0..10 | – | – | n/a | **20.6** | 59.1 |
| | | 0..50 | – | – | n/a | **20.9** | 58.7 |
| reversible -lane-u | E | 0..5 | – | – | n/a | 31.3 | **6.7** |
| | | 0..10 | – | – | n/a | 54.8 | **6.4** |
| | | 0..50 | – | – | n/a | 60.6 | **6.3** |
| elevator w. door | S | 0..5 | – | – | n/a | **7.6** | 111.2 |
| | | 0..10 | – | – | n/a | **7.7** | 148.5 |
| | | 0..50 | – | – | n/a | **7.8** | 138.4 |
| num. solved (out of 27) | | | 3 | 0 | 9 | 26 | 23 |

an explicit abstraction can also take time when the number of predicates is large (`follow`). However, for more complex problems acceleration can significantly reduce the runtime (see `rep-reach-obst.-2d`). Interestingly, for all benchmarks we have manually identified predicates and liveness constraints that allow for the problems to solved by `sweap`, given enough time. This is not true for the other approaches (e.g., they all are not expressive enough to solve `xyloop`). Kesten and Pnueli [23] prove that predicate abstraction with ranking abstractions are complete for model checking. We conjecture this is true also for synthesis, but obviously no automatic approach can be assured to terminate in finding the right refinements. We allow for the user to add refinements manually to exploit this.

The finite-state results show that restricted domains do not help Raboniel and `temos` much. For benchmarks in Table 1a but not in Table 1b the results are similar for each finite range and tool, thus for lack of space we do not show them here. We note that for a subset (second to fifth row) `rpgsolve`'s runtime tends to increase based on the size of the finite domain, whereas the difference in running times is negligible for `sweap` on varying domains. `sweap` shows largely the same behaviour on the LTL benchmarks. These benchmarks share one thing

in common: they have essentially the same solution regardless of the domain. This suggests that for this problem class `sweap` can find a solution independent of the domain size, whereas `rpgsolve` runtime may vary based on the domain.

Two benchmarks stand out. `elevator-simple` is unlike the others, for this we vary the number of some Boolean variables (and size of the LTL formula). For each domain then it becomes a different problem, requiring more predicates. For the rest we vary the range of numeric variables. Without acceleration, `sweap` fails on `elevator-signal`, failing to find any liveness refinements. This suggests we need to apply liveness refinements in a wider context.

## 8 Related Work

There are many approaches that tackle symbolic synthesis, many listed in the previous section. Given the space restrictions, here we choose to mainly discuss methods related to our notion of liveness refinements, we refer to other work [16, 20] for recent good general overviews of existing methods. We find three different classes of approaches that include methods using learned liveness constraints:

*Fixpoint solving:* There are methods that extend standard fixpoint approaches to symbolic game solving. de Alfaro et. al [1] identify certain games decidable with such an approach. GenSys-LTL [34] uses quantifier elimination to compute the controllable predecessor of a given set, terminating if a finite number of steps is sufficient. Heim and Dimitrova in `rpgsolve` [20] take this further by synthesising what they term *acceleration lemmas*. They attempt to synthesise linear ranking functions with invariants to prove that a loop in the game is terminating, allowing finding fixpoints GenSys-LTL cannot find. This information is however only used in a local manner and specific to a particular game region. For example, for the reachability problem `xyloop` this results in the need for an infinite number of accelerations, leading to divergence. Another limitation is the reliance on identifying one location in a game where a ranking function decreases. This is problematic both in cases when the choice of where to exit a region is part of the game-playing and when the ranking needs to decrease differently based on the history of the play. The latter would be required in order to scale their approach to winning conditions beyond the inexpressive Büchi and co-Büchi.

*Abstraction:* Other methods attempt synthesis on an explicit abstraction of the problem. If this fails, the resulting counterexample can be used to refine the abstraction and the approach is repeated. Our method falls under this approach. We find methods taking this approach that target games directly [21, 2, 36] and others that work at the level of the specification [17, 25, 10]. Many of these focus on refining states in the abstraction, a kind of safety refinement. As far as we know, only `temos` [10] adds some form of liveness information of the underlying infinite domain. `Temos` is not a classic approach, it only attempts synthesis once. Initially 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, they can also identify liveness constraints that abstract the effects in the limit of repeating an update, adding constraints of the form $G(pre \wedge (uWpost) \implies Fpost)$. However this can only deal with one update of one variable at a time, and fails when the environment can delay $u$.

*Constraint Solving:* Several approaches tackle the synthesis problem by encoding it as a constrained Horn clause solving problem. A benefit is that this allows for the synthesis of ranking functions to prove termination of parts of a program. Beyene et. al. [5] solve general LTL and $\omega$-regular infinite-state games with constraint solving, implemented in `consynth`. The user must however give a controller template, essentially giving a partial solution to the problem. Beyene et. al. attempt to fill this template which may require synthesising ranking functions. An unrealisability verdict here is however not generalisable, unlike in `sweap`, given it is limited to this specified template. MuVal [35] can encode realisability checking of LTL games as validity checking in a fixpoint logic that extends constrained Horn clauses. They also require encoding the automaton corresponding to the LTL formula directly in the input formula. MuVal discovers ranking functions based on templates to enforce bounded unfolding of recursive calls. Contrastingly, we do not need to rely on templates but can handle any argument for termination.

## 9 Conclusions

We have presented an abstraction-refinement approach to the synthesis of LTL specifications beyond Boolean domains that relies on deducing liveness refinements that can forego the need for a large or infinite number of safety refinements. We design our framework carefully to be able to encode spuriousness checking of abstract counterstrategies as simple invariant checking, and use loops in counterexamples to find liveness refinements. Our evaluation shows that with liveness refinements we can surprisingly automatically solve problems with large finite or infinite state spaces beyond what existing approaches can solve, even though we maintain an explicit predicate abstraction.

*Future work.* We believe that symbolic approaches for LTL synthesis could significantly accelerate our approach [14, 19], however, the tool support for these approaches is not yet mature. Experimenting with such a tool [14] we sometimes observed considerable decrease in runtime for our synthesis checks, however this tool does not supply strategies. We are considering expanding the conditions for when to apply liveness refinement, e.g., by identifying loops at different abstraction levels of the counterstrategy, rather than just at the state label level. We are also considering a dual approach, where the controller is given control of the abstract program, making model checking semi-decidable for realisability checking in the infinite case, and identifying loops the controller wishes to settle in. This would also enable to compute parts of the winning regions of both players (as in the approach of `rpgsolve`), and accelerate synthesis. Other directions include extending the tool beyond LIA, and applying standard methods to manage the size of predicate abstractions (e.g., [22]) in this context.

## References


1. de Alfaro, L., Henzinger, T.A., Majumdar, R.: Symbolic algorithms for infinite-state games. In: CONCUR 2001 — Concurrency Theory. pp. 536–550. Springer Berlin Heidelberg, Berlin, Heidelberg (2001)
2. de Alfaro, L., Roy, P.: Solving games via three-valued abstraction refinement. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007 – Concurrency Theory. pp. 74–89. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
3. Baier, C., Coenen, N., Finkbeiner, B., Funke, F., Jantsch, S., Siber, J.: Causality-based game solving. In: Computer Aided Verification. pp. 894–917. Springer International Publishing, Cham (2021)
4. Balaban, I., Pnueli, A., Zuck, L.D.: Ranking abstraction as companion to predicate abstraction. In: Formal Techniques for Networked and Distributed Systems - FORTE 2005. pp. 1–12. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
5. Beyene, T.A., Chaudhuri, S., Popeea, C., Rybalchenko, A.: A constraint-based approach to solving games on infinite graphs. In: The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 221–234. ACM (2014)
6. Beyene, T.A., Popeea, C., Rybalchenko, A.: Solving existentially quantified horn clauses. In: Computer Aided Verification. pp. 869–882. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
7. Beyer, D., Keremoglu, M.E.: Cpachecker: A tool for configurable software verification. In: Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6806, pp. 184–190. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16
8. Bloem, R., Könighofer, R., Seidl, M.: Sat-based synthesis methods for safety specs. In: McMillan, K.L., Rival, X. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 1–20. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
9. 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, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8559, pp. 334–342. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_22
10. 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. p. 229–243. PLDI 2022, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3519939.3523429, https://doi.org/10.1145/3519939.3523429
11. 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, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7795, pp. 93–107. Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_7
12. Clarke, E.M., Grumberg, O., Peled, D.A.: Model checking. MIT Press, London, Cambridge (1999)

13. Cook, B., Gulwani, S., Lev-Ami, T., Rybalchenko, A., Sagiv, M.: Proving conditional termination. In: Computer Aided Verification. pp. 328–340. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
14. 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. Lecture Notes in Computer Science, vol. 14570, pp. 83–102. Springer (2024). https://doi.org/10.1007/978-3-031-57246-3_6
15. Faella, M., Parlato, G.: Reachability games modulo theories with a bounded safety player. In: Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence and Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence and Thirteenth Symposium on Educational Advances in Artificial Intelligence. AAAI'23/IAAI'23/EAAI'23, AAAI Press (2023). https://doi.org/10.1609/aaai.v37i5.25779
16. Farzan, A., Kincaid, Z.: Strategy synthesis for linear arithmetic games. Proc. ACM Program. Lang. **2**(POPL) (dec 2017). https://doi.org/10.1145/3158149, https://doi.org/10.1145/3158149
17. Finkbeiner, B., Klein, F., Piskac, R., Santolucito, M.: Temporal stream logic: Synthesis beyond the bools. In: Computer Aided Verification. pp. 609–629. Springer International Publishing, Cham (2019)
18. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: CAV'97. LNCS, vol. 1254, pp. 72–83. Springer (1997). https://doi.org/10.1007/3-540-63166-6_10
19. 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. Lecture Notes in Computer Science, vol. 14574, pp. 55–78. Springer (2024). https://doi.org/10.1007/978-3-031-57228-9_4
20. Heim, P., Dimitrova, R.: Solving infinite-state games via acceleration. Proc. ACM Program. Lang. **8**(POPL) (jan 2024). https://doi.org/10.1145/3632899
21. Henzinger, T.A., Jhala, R., Majumdar, R.: Counterexample-guided control. In: 30th International Colloquium on Automata, Languages and Programming. Lecture Notes in Computer Science, vol. 2719, pp. 886–902. Springer (2003)
22. 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, January 16-18, 2002. pp. 58–70. ACM (2002). https://doi.org/10.1145/503272.503279
23. Kesten, Y., Pnueli, A.: Verification by augmented finitary abstraction. Information and Computation **163**(1), 203–243 (2000). https://doi.org/https://doi.org/10.1006/inco.2000.3000
24. Legg, A., Narodytska, N., Ryzhyk, L.: A sat-based counterexample guided method for unbounded synthesis. In: Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification. pp. 364–382. Springer International Publishing, Cham (2016)
25. Maderbacher, B., Bloem, R.: Reactive synthesis modulo theories using abstraction refinement. In: 22nd Conference on Formal Methods in Computer-Aided Design, FMCAD 2022. p. 315–324. TU Wien Academic Press (2022). https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_38
26. Markgraf, O., Hong, C.D., Lin, A.W., Najib, M., Neider, D.: Parameterized synthesis with safety properties. In: Programming Languages and Systems. pp. 273–292. Springer International Publishing, Cham (2020)

27. Martin, D.A.: Borel determinacy. Annals of Mathematics **102**(2), 363–371 (1975), http://www.jstor.org/stable/1971035
28. McMillan, K.L.: Lazy abstraction with interpolants. In: Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4144, pp. 123–136. Springer (2006). https://doi.org/10.1007/11817963_14
29. Meyer, P.J., Sickert, S., Luttenberger, M.: Strix: Explicit reactive synthesis strikes back! In: Computer Aided Verification - 30th International Conference, CAV 2018, July 14-17, 2018, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10981, pp. 578–586. Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_31
30. Narodytska, N., Legg, A., Bacchus, F., Ryzhyk, L., Walker, A.: Solving games without controllable predecessor. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification. pp. 533–540. Springer International Publishing, Cham (2014)
31. Neider, D., Topcu, U.: An automaton learning approach to solving safety games over infinite graphs. In: Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9636. p. 204–221. Springer-Verlag, Berlin, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_12
32. Piterman, N., Pnueli, A.: Temporal logic and fair discrete systems. In: Handbook of Model Checking, pp. 27–73. Springer (2018)
33. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL. pp. 179–190. ACM Press (1989)
34. 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
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

# A Supplementary Material for Section 3

Here we give some more detail of the structural refinement for the example in the informal overview.

*Structural Refinement.* We can exclude terminating loops in general by adding logic that monitors for the loop, and a constraint that forces it to terminate. To identify when the loop is entered we add a formula of the form: $G(iter\_cond \wedge (Xact) \iff Xinloop)$. Here, *iter_cond* corresponds to the iteration condition of the loop ($floor < target$), and *act* to $floor = floor_{prev} + 1 \wedge target = target_{prev}$. Note the stutter update for target is implicit in the while loop we considered, but we make it explicit here to remain sound. Then a loop is exited when the iteration condition does not hold after an iteration, $G((inloop \wedge \neg iter\_cond) \implies \neg inloop)$. We also allow arbitrary stuttering while in the loop, i.e. $G(inloop \implies (X(st \implies inloop)))$, where $st = (floor = floor_{prev} \wedge target = target_{prev})$. Initially we are not in the loop, $\neg inloop$, and importantly, we have to be infinitely often outside the loop or stutter inside the loop eventually forever, $GF(\neg inloop) \vee FG(st \wedge inloop)$. Adding these constraints for both loops allows us to also solve the problem. Note however this requires adding more variables than with ranking refinement, therefore we prefer ranking refinement when possible.

# B Supplementary Material for Section 5

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

*Proof.* This follows immediately from the soundness of the abstraction.

**Lemma 1.** *An abstract counterstrategy Cs is concretisable w.r.t. $P$ iff every trace $t \in L(Cs)$ is concretisable w.r.t. $P$.*

*Proof.* This follows from Defn. 3.

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

*Proof.* This follows immediately from Defns. 3, 5, and 7.

**Proposition 1.** *Counterstrategy concretisability can be encoded as model checking, and terminates for finite problems and non-concretisable counterstrategies.*

*Proof.* Given a program $P$, a formula $\phi$, and a counterstrategy $CS$ with predicates $\mathcal{P}r$, we compose $P$ with $CS$, giving the program $P \times CS$, in the following manner:

The variables of $P \times CS$ are the variables of $P$, the set of states of $CS$, and fresh Boolean variables for each predicate in $\mathcal{P}r$, i.e. $v_p$ for each predicate

$p \in \mathcal{P}r$, we denote this set by $V_{\mathcal{P}r}$. Initially all the $CS$ state variables are false except the initial state $s_0$, and all predicate variables expected to be true by $s_0$ set to true, and all others to false.

For each transition $g \mapsto U$ in the program and for each transition $s_i \xrightarrow{C} s_{i+1}$ in $CS$, such that $out(s_i) = (E_i, Pr_i)$, and $out(s_i) = (E_{i+1}, Pr_{i+1})$, there is a transition $g \wedge \bigwedge\!\!\bigwedge_S \{s\} \wedge \bigwedge\!\!\bigwedge E_{i+1} \wedge \bigwedge\!\!\bigwedge_{V_{Pr}} V_{Pr_i} \mapsto U'$ in $P \times CS$. $U'$ consists of $U$, extended with the following updates: $\{s_{i+1} := true;\}$, $\{s_j := false \mid j \neq i+1\}$, $\{v_p := true \mid p \in Pr_{i+1}\}$, $\{v_p := true \mid p \notin Pr_{i+1}\}$, and $\{v_{prev} := v \mid v \in V\}$. The transitions of $P \times CS$ are exactly these transitions.

It should be easy to that this program satisfies the invariant $G \bigwedge(v_p \iff p)$ iff $CS$ is concretisable on $P$. Moreover, for finite programs this program is finite, for which model checking is decidable.

Assume the counterstrategy is not concretisable, then by Defn. 7 there must be a finite counterexample. Moreover, the program has one initial state, and only allows for finite branching in each time step. Thus if there is a counterexample it will be found in finite time. □

**Proposition 2.** *If a counterstrategy is not concretisable, there is a finite counterexample $a_0, ..., a_k \in (2^{\mathbb{E} \cup \mathbb{C} \cup \mathcal{P}r})^*$, s.t. concretisability fails locally only on $a_k$.*

*Proof.* This follows easily from the fact that concretisability checking can be encoded as invariant checking.

**Proposition 3.** *The formula $\alpha(P, Pr)$ is an abstraction of $P$.*

*Proof.* Recall that a formula over $\mathbb{E} \cup \mathbb{C} \cup \mathcal{P}r$ is an abstraction of a program $P$ if for all concrete models of the program there is an abstract word in the abstraction that abstracts the concrete word.

Consider a word $w_a$ in $L_{\mathcal{P}r}(P)$. We claim there is a word $a$ s.t. $w_a \in \gamma(a)$ and $a$ is in $\alpha(P, Pr)$. For the initial state it should be clear that the initial condition $\bigwedge\!\!\bigwedge Pr_\iota$ ensures the initial program state is properly abstracted. The rest of the abstraction abstracts transitions, thus we proof its correctness by induction on pairs of successive letters. Throughout, for the concrete word $w_a$, we set $w_a(i) = (val_i, E_i \cup C_i)$.

For the base case, we consider the first two letters of $a$, $a(0)$ and $a(1)$. Since $a$ is concretisable it follows easily that $Pr_\iota \subseteq a(0)$, and that there is a transition $f \mapsto U$ s.t. $f(w_a(0))$, $val_1 = U(val_0)$, and $(val_0, val_1) \models a(1)$. From this it follows that this initial transition is captured by Defn. 13.

For the inductive case, consider $a(i)$ and $a(i+1)$. By Defn. 3, for $j \in \{i, i+1\}$ we have that $w_a(j) = (val_j, E_j \cup C_j)$, then $a(j) = E_j \cup C_j \cup \mathcal{P}r_j$ for some predicate set $\mathcal{P}r_j \subseteq \mathcal{P}r$, and $(val_i, val_{i+1}) \models \bigwedge\!\!\bigwedge_{\mathcal{P}r} \mathcal{P}r_{i+1}$. Concretisability of the word, ensures there is a transition $f \mapsto U$ s.t. $f$ and $a(i)$ is satisfiable ($w_a(i)$ is a model for this). Consider that $\bigwedge Pr_i \wedge f$ is satisfiable, implying that $\bigwedge(Pr_i)_{prev} \wedge f_{prev}$. Moreover, consider that $val_{i+1} = U(val_i)$, implying that $\bigwedge(Pr_i)_{prev} \wedge f_{prev} \bigwedge_{v:=t \in U} v = t_{prev} \wedge \bigwedge\!\!\bigwedge Pr_{i+1}$ is also satisfiable, as required by Defn. 14.

### B.1 Predicate Abstraction

We define an abstraction of the program in terms of a set of predicates $\mathcal{P}r$. Initially, $\mathcal{P}r$ is exactly the set of predicates appearing in the desired formula $\phi$.[3]

The program abstraction then focuses on abstracting the symbolic transition relation $\delta_{sym}$ of the program in terms of $\mathcal{P}r$, such that every symbolic transition has corresponding abstract transitions. We rely on satisfiability checking to compute this abstraction. Moreover, given that we have an initial variable valuation, we give a sound and complete abstraction for the initial transition. This will be crucial later to ensure progress of safety refinement.

**Definition 13 (Abstracting the Initial Transition).** *Given a set of predicates $\mathcal{P}r$ and a program $P$, the* initial transition abstraction *of $P$ w.r.t. $\mathcal{P}r$ is the relation $\iota_{\mathcal{P}r} \subseteq 2^{\mathbb{E}\cup\mathbb{C}} \times 2^{\mathcal{P}r}$, such that $(E \cup C, Pr_{E,C}) \in \iota_{\mathcal{P}r}$ iff there exists $f \in dom(\delta_{sym})$ such that $\bigwedge\!\!\!\bigwedge(E \cup C) \wedge val_0 \wedge f$ is true and if $U = \delta_{sym}(f)$ then $(val_0, U(val_0)) \models \bigwedge\!\!\!\bigwedge_{\mathcal{P}r} Pr_{E,C}$.*

Notice that $(E \cup C, Pr_{E,C}) \in \iota_{\mathcal{P}r}$ iff $\delta(val_0, E \cup C) \models \bigwedge\!\!\!\bigwedge_{\mathcal{P}r} Pr_{E,C}$. Furthermore, due to determinism of $P$, for every $E \cup C$ there is a unique $Pr_{E,C}$ such that $(E \cup C, Pr_{E,C}) \in \iota_{\mathcal{P}r}$.

**Definition 14 (Abstracting Transitions).** *Given a set of predicates $\mathcal{P}r$ and a program $P$, the* abstract transition *of $P$ w.r.t. $\mathcal{P}r$ is a relation $\delta_{\mathcal{P}r} \subseteq 2^{\mathbb{E}\cup\mathbb{C}\cup\mathcal{P}r} \times 2^{\mathcal{P}r}$, such that $(E \cup C \cup Pr^0, Pr^1) \in \delta_{\mathcal{P}r}$ iff there exists $f \in dom(\delta_{sym})$ such that $\bigwedge\!\!\!\bigwedge(E \cup C \cup Pr^0) \wedge f$ is satisfiable and if $U = \delta_{sym}(f)$ then $\bigwedge\!\!\!\bigwedge Pr^0_{prev} \wedge f_{prev} \wedge \bigwedge_{v:=t \in U} v = t_{prev} \wedge \bigwedge\!\!\!\bigwedge Pr^1)$ is satisfiable as well.*

*We further assume this is reduced up to reachability, s.t. $(E \cup C \cup Pr^0) \in dom(\delta_{\mathcal{P}r})$ iff $Pr^0 \in ran(\delta_{\mathcal{P}r})$ or $Pr^0 \in ran(\iota_{\mathcal{P}r})$.*

Note that $(E \cup C \cup Pr, Pr') \in \delta_{\mathcal{P}r}$ if and only if there exist valuations $val$ and $val$ such that $val \models \bigwedge\!\!\!\bigwedge_{\mathcal{P}r}(Pr)$, $(val, val') \models \bigwedge\!\!\!\bigwedge_{\mathcal{P}r} Pr'$, and $\delta(val, E \cup C) = val'$.

Based on these we define a formula in $LTL(\mathbb{E} \cup \mathbb{C} \cup \mathcal{P}r)$ that abstracts the program. Let $Pr_\iota$ be the set of predicates true for $val_0$: $val_0 \models \bigwedge\!\!\!\bigwedge_{\mathcal{P}r} Pr_\iota$.

**Definition 15 (Safety Abstraction).** *The abstract characteristic safety formula $\alpha(P, \mathcal{P}r)$ w.r.t. a set of predicates $\mathcal{P}r$ is the conjunction of: $\bigwedge\!\!\!\bigwedge Pr_\iota$ characterising the predicates holding initially, $\bigvee_{(E\cup C, Pr)\in\iota_{\mathcal{P}r}}(\bigwedge\!\!\!\bigwedge E \cup C) \wedge X \bigwedge\!\!\!\bigwedge Pr$ characterising the initial transition, and $G\left(\bigvee_{(S,Pr')\in\delta_{\mathcal{P}r}}(\bigwedge\!\!\!\bigwedge S \wedge X \bigwedge\!\!\!\bigwedge Pr')\right)$ characterising all the other transitions.*

Note that the initial transitions captured by the second conjunct are a special case of the full transition abstraction in the third conjunct.

The complexity of this construction, ignoring satisfiability checking, is at worst exponential in the size of $\mathbb{E}\cup\mathbb{C}\cup\mathcal{P}r$. Depending on the theory, satisfiability checking may increase this complexity. For LIA, this is exponential, leaving the complexity lower than the complexity of LTL reactive synthesis.

[3] A technical detail is that Boolean variables in $V$ are also in $\mathcal{P}r$.

## C Supplementary Material for Section 6

**Theorem 4 (Existence of Sequence Interpolants [McM06]).** *For a sequence of formulas $f_0, \ldots, f_k$, s.t. $\bigwedge_{i=0}^{k} f_i$ is unsatisfiable and for every $i, j$ either $|i-j| \leq 1$ or $f_i$ and $f_j$ do not share variables, then there is a set of sequence interpolants $I_0, \ldots, I_{k-1}$, where $f_0 \implies I_0$, $\forall 1 \leq i < k. f_i \wedge I_i \implies I_{i+1}$, and $I_{k-1} \wedge f_k$ is unsatisfiable. Furthermore, the variables of each $I_i$ appear in both $f_i$ and $f_{i+1}$.*

**Proposition 4 (Safety Refinement Progress).** *For an abstraction $\alpha(P, Pr)$ allowing a counterstrategy Cs with a finite counterexample ce, then $\alpha(P, Pr \cup I(ce))$ does not allow counterexamples that induce the same refinement.*

*Proof.* Suppose the abstraction $\alpha(P, Pr \cup I(ce))$ contains an unconcretisable w.r.t. $P$ word $a$ with a finite prefix $a_0, \ldots, a_k$, such that concretisability fails only on $a_k$. Suppose further that interpolants $I^a = I(a_0, \ldots, a_k)$ corresponding to this trace are equal (or a subset) of $I(ce)$.

Note how the initial transition abstraction ensures $I_0^a$ is always true in the first step, thus $a_0$ must guess $I_0^a$ to be true. Similarly, by Thm. 4 and Defn. 14 the correct guesses of interpolants must be maintained throughout. Then, $a_k$ must guess $I_k^a$ to be true, however this creates a contradiction, since Defn. 14 requires the predicate guesses to be satisfiable, but Thm. 4 ensures $I_k^a$ is not satisfiable with $a_k$. □

**Theorem 3.** *Alg. 1 with safety refinement terminates on finite programs.*

*Proof.* Note that a finite program $P$ has a finite number of possible variable valuations, and thus $\delta_P$ is finite. Then model checking is decidable. Finding sequence interpolants is also decidable.

Moreover, recall that given a counterexample, the interpolants $I$ learned through safety refinement always strictly refine the abstraction, Prop. 4. Consider a predicate set $\mathcal{P}r$ and $\mathcal{P}r' = \mathcal{P}r \cup I$, where $I$ is a set of interpolants discovered through analysing a counterexample. Consider also that $\delta_{\mathcal{P}r}$ is an abstraction for $\delta$, s.t. each element $(E \cup C \cup Pr, Pr') \in \delta_{\mathcal{P}r}$ has a corresponding concrete finite set of transitions in $\delta$. Upon adding $I$ to the abstraction, each original abstract transition is replicated for each subset of $I$. Each of these new abstract transitions partition a subset of the original set of concrete transitions of the corresponding abstract transition between them.

Then, $\delta_{Pr}$ accepts strictly more valuation pairs $((E \cup C, val), val')$ than $\delta_{Pr'}$. Given there is a finite set of such valuation pairs, and refinement always makes progress, then refinement can only be repeated for a finite amount of steps. This ensures there cannot be an infinite chain of discovered spurious counterstrategies, and thus a concretisable counterstrategy or controller are eventually found. □

The following is a standard result, which we use to show that our ranking refinement eliminates the counterexample.

**Theorem 5.** *If a loop $l$ has a ranking function, then it is terminating.*

*Proof (sketch).* Recall the well-foundedness ensures there can not be infinite decreasing chains of values in $R$. The value of $r$ decreases in each step of the loop, by definition, and thus the infinite language of the loop must be empty, and the loop terminates.

**Proposition 5 (Ranking Abstraction Correctness).** *Given a ranking function $r$, then $\alpha(P, Pr \cup \{r^{dec}, r^{inc}\}) \wedge \alpha_{rk}(P, r)$ is an abstraction of $P$.*

*Proof.* $\alpha(P, Pr)$ is an abstraction of $P$. What is left to show is that $\alpha_{rk}(P, r)$ also is an abstraction of $P$. Note, every abstract trace in $L(\alpha_{rk}(P, r))$ satisfies that either eventually always $r$ does not decrease, or infinitely often it is both decreases and increases. Consider a concrete word, by the property of $r$ having a well-founded range, every concrete word will exhibit exactly this behaviour, and thus the corresponding abstract word will be in $L(\alpha_{rk}(P, r))$. □

**Proposition 6 (Ranking Refinement Progress).** *For abstraction $\alpha(P, Pr)$ allowing a counterstrategy Cs with a finite counterexample ce that induces a lasso in Cs and a ranking function $r$, then abstraction $\alpha(P, Pr \cup \{r_{dec}, r_{inc}\}) \wedge \alpha_{rk}(P, r)$ does not allow counterexamples that induce the same refinement.*

*Proof.* Suppose that the abstraction $\alpha(P, Pr \cup \{r_{dec}, r_{inc}\}) \wedge \alpha_{rk}(P, r)$ contains an unconcretisable w.r.t. $P$ word $a$, with a prefix $a_0, \ldots, a_l, \ldots, a_k$, such that concretisability fails only on $a_k$, $a_k = a_l$, and the suffix of the word is of the form $(a_l, \ldots, a_k - 1)^\omega$ (note this is an equivalent condition to finding a lasso in a counterstrategy).

If concretisability fails because of a transition predicate mismatch then a ranking refinement is not applicable, thus the result holds.

Consider that concretisability fails because of a state predicate mismatch, and thus ranking refinement is applicable. Furthermore consider that $a$ induces the same ranking function $r$. Thus $r$ is strictly decreasing in the program transitions corresponding to $(a_l, \ldots, a_k - 1)^\omega$. Given $\alpha_{rk}(P, r)$, $r_{inc}$ must not be guessed as true in any of $a_l, \ldots, a_k - 1$, otherwise given $r$ is a ranking function for this loop, the concretisability would have failed earlier than $a_k$ on the wrong guess of $r_{inc}$. Similarly if $r_{dec}$ is guessed as false throughout. Then only $r_{dec}$ is guessed as true in $a_l, \ldots, a_k - 1$, giving a contradiction, since then the word is not in $\alpha_{rk}(P, r)$. □

**Proposition 7 (Structural Loop Refinement Correctness).** *For a terminating loop $l$, and a set of predicates $\mathcal{P}r_l$ that consists of exactly all the atomic predicates over program variables in pre and $\alpha_{loop}(P, l)$, then $\alpha(P, Pr \cup \mathcal{P}r_l) \wedge \alpha_{loop}(P, l)$ is an abstraction of $P$.*

*Proof.* Consider a word $a \in AL_{\mathcal{P}}r(P)$. We know this is in $\alpha(P, Pr \cup \mathcal{P}r_l)$ by Prop. 3, what is left to show is that it is in $\alpha_{loop}(P, l)$, modulo some additions of $inloop_i$ variables. It should be easy to see that conditions 1-6 do not put any restrictions on the variable state space. Only condition 7 has the potential to eliminate program words unsoundly.

Consider $a$ is a word in $AL_{\mathcal{P}r}(P)$ but that it has no counterpart in $\alpha_{loop}(P, l)$. Then, it must satisfy the negation of condition 7, i.e. $FG(inloop) \wedge \bigwedge_i GF(\neg(st_i \wedge inloop_i))$, so that the word eventually remains in the loop without stuttering. Then $a$ must have a maximal $k$ s.t. the suffix $a_k$ has that $pre \wedge iter\_cond$ is satisfiable with $a(k)$, and $a(k+1)$ satisfies $p_0$, since this is the only way for *inloop* to start being true (formula 3). At each point in time then either 4 or 5 hold and leave *inloop* true. Moreover, at each loop step there must be a finite amount of stuttering (as required by negation of 7). Thus $a_k$ corresponds, up to stuttering, to the loop with precondition $pre$, iteration condition $iter\_cond$, and body $(g_0, U_0), \ldots, (g_n, U_n)$ (given the correspondence of the predicates $p_i$ to these guarded updates). Note then that any concretisation of $a_k$ must not exit from this loop. However, by assumption this loop is terminating, creating a contradiction. □

**Proposition 8 (Structural Loop Refinement Progress).** *For an abstraction $\alpha(P, Pr)$ allowing a counterstrategy Cs with a finite counterexample ce that induces a lasso in Cs and a corresponding loop $l$, the abstraction $\alpha(P, Pr \cup \mathcal{P}r_l) \wedge \alpha_{loop}(P, l)$ does not allow counterexamples that induce the same refinement.*

*Proof.* Suppose that the abstraction $\alpha(P, Pr \cup \mathcal{P}rl) \wedge \alpha_{loop}(P, l)$ contains an unconcretisable w.r.t. $P$ word $a$, with a prefix $a_0, \ldots, a_l, \ldots, a_k$, (s.t. $l < k$) such that concretisability fails due to a state predicate mismatch on $a_k$, and this exercises a lasso in the counterstrategy $s_0, \ldots, s_l, \ldots, s_k$, s.t. $s_k = s_l$, and the suffix of the word is thus of the form $(a_l, \ldots, a_k - 1)^\omega$. Suppose further that $a$ also has the corresponding loop $l$, and requires the same refinement.

If $a$ guesses $pre$ wrongly (false) at $a_l$ then concretisability will fail at $a_l$ rather than at $a_k$. Thus we assume $pre$ is guessed correctly, and similarly for the iteration condition at $a_l$. Moreover, all of the transition predicates ($p_i$ and $st_i$) must be guessed correctly, otherwise the mismatch is not a state predicate mismatch. However, then if all these are guessed correctly $a$ is a witness that the abstraction allows words that go through the loop (as captured by conditions 1-6), and remains in the loop, violating condition 7, violating $\alpha_{loop}(P, l)$. □

## D Supplementary Material for Section 7

Here we show details about the number of state predicates, transition predicates, and the number of refinements performed by each configuration of `sweap` for each benchmark. Respectively, *init st.* and *init tr.* indicate the number of initial state and transition predicates. Note, $x$ initial transition predicates indicates $x/2$ accelerations performed, i.e. $x/2$ strong fairness constraints added as assumptions to the abstract LTL problem. Respectively, *num. sf. rf.* and *num. sl. rf.* indicate the number of safety and strucural loop refinements performed, and *added st.* and *added tr.* indicate the number of state and transition predicates added by such refinements.

| Name | config | init st. | init tr. | num. sf. rf | num. sl. rf. | added st. | added tr. |
|---|---|---|---|---|---|---|---|
| box | $S_{acc}$ | 7 | 4 | 0 | 0 | 0 | 0 |
| | S | 4 | 0 | 0 | 0 | 0 | 0 |
| box-limited | $S_{acc}$ | 6 | 4 | 0 | 0 | 0 | 0 |
| | S | 2 | 0 | 0 | 0 | 0 | 0 |
| diagonal | $S_{acc}$ | 8 | 8 | 0 | 0 | 0 | 0 |
| | S | 4 | 0 | 0 | 0 | 0 | 0 |
| evasion | $S_{acc}$ | 12 | 16 | 0 | 0 | 0 | 0 |
| | S | 4 | 0 | 1 | 0 | 1 | 0 |
| follow | $S_{acc}$ | 20 | 24 | 0 | 0 | 0 | 0 |
| | S | 12 | 0 | 0 | 0 | 0 | 0 |
| square | $S_{acc}$ | 10 | 4 | 0 | 0 | 0 | 0 |
| | S | 8 | 0 | 0 | 0 | 0 | 0 |
| robot-cat-r-1d | $S_{acc}$ | 7 | 8 | 0 | 0 | 0 | 0 |
| | S | 0 | 0 | 4 | 0 | 11 | 0 |
| robot-cat-u-1d | $S_{acc}$ | 7 | 8 | 0 | 0 | 0 | 0 |
| | S | 0 | 0 | 1 | 0 | 2 | 0 |
| robot-cat-r-2d | $S_{acc}$ | 18 | 16 | 0 | 0 | 0 | 0 |
| | S | 0 | 0 | 3 | 0 | 9 | 0 |
| robot-cat-u-2d | $S_{acc}$ | 18 | 16 | 0 | 0 | 0 | 0 |
| | S | 0 | 0 | 1 | 0 | 4 | 0 |
| robot-grid-reach-1d | $S_{acc}$ | 3 | 2 | 0 | 0 | 0 | 0 |
| | S | 0 | 0 | 1 | 2 | 2 | 3 |
| robot-grid-reach-2d | $S_{acc}$ | 6 | 4 | 0 | 0 | 0 | 0 |
| | S | 0 | 0 | 1 | 4 | 4 | 6 |
| heim-double-x | $S_{acc}$ | 5 | 6 | 6 | 0 | 12 | 0 |
| | S | 0 | 0 | 7 | 0 | 24 | 0 |
| xyloop | $S_{acc}$ | 3 | 8 | 0 | 0 | 0 | 0 |
| | S | 2 | 0 | 10 | 1 | 13 | 3 |
| robot-grid-commute-1d | $S_{acc}$ | 7 | 8 | 0 | 0 | 0 | 0 |
| | S | 0 | 0 | 4 | 2 | 8 | 4 |
| robot-grid-commute-2d | $S_{acc}$ | 14 | 16 | 0 | 0 | 0 | 0 |
| | S | 0 | 0 | 3 | 0 | 10 | 0 |
| robot-resource-1d | $S_{acc}$ | 5 | 4 | 2 | 0 | 4 | 0 |
| | S | 0 | 0 | 6 | 1 | 8 | 2 |
| robot-resource-2d | $S_{acc}$ | 7 | 6 | 1 | 1 | 3 | 3 |
| | S | 0 | 0 | 2 | 2 | 6 | 3 |
| heim-buechi | $S_{acc}$ | 9 | 6 | 0 | 0 | 0 | 0 |
| | S | 0 | 0 | 9 | 0 | 10 | 0 |
| heim-fig7 | $S_{acc}$ | 3 | 2 | 0 | 0 | 0 | 0 |
| | S | 0 | 0 | 0 | 0 | 0 | 0 |
| batch-arbiter-u | $S_{acc}$ | 4 | 2 | 0 | 0 | 0 | 0 |
| | S | 1 | 0 | 1 | 0 | 2 | 0 |
| reversible-lane-r | $S_{acc}$ | 4 | 8 | 0 | 0 | 0 | 0 |
| | S | 4 | 0 | 1 | 2 | 1 | 4 |
| reversible-lane-u | $S_{acc}$ | 4 | 8 | 2 | 0 | 2 | 0 |
| | S | 4 | 0 | 0 | 0 | 0 | 0 |
| batch-arbiter-r | $S_{acc}$ | 2 | 4 | 0 | 0 | 0 | 0 |
| | S | 1 | 0 | 1 | 1 | 1 | 2 |
| elevator-w-door | $S_{acc}$ | 4 | 8 | 0 | 0 | 0 | 0 |
| | S | 2 | 0 | 4 | 2 | 4 | 4 |
| rep-reach-obst.-1d | $S_{acc}$ | 3 | 2 | 0 | 0 | 0 | 0 |
| | S | 0 | 0 | 1 | 2 | 2 | 3 |
| rep-reach-obst.-2d | $S_{acc}$ | 8 | 4 | 0 | 0 | 0 | 0 |
| | S | 0 | 0 | 1 | 4 | 4 | 6 |
| taxi-service | $S_{acc}$ | 14 | 16 | 0 | 0 | 0 | 0 |
| | S | 4 | 0 | 0 | 4 | 0 | 8 |

Table 2: Infinite-state experiments details for $\texttt{sweap}_{acc}$ and $\texttt{sweap}$.

| Name | range | config | init st. | init tr. | num. sf. rf | num. sl. rf. | added st. | added tr |
|---|---|---|---|---|---|---|---|---|
| elevator-simple | 0..5 | $S_{acc}$ | 10 | 2 | 0 | 0 | 0 | 0 |
| | | S | 10 | 0 | 0 | 0 | 0 | 0 |
| | 0..10 | $S_{acc}$ | 11 | 2 | 0 | 0 | 0 | 0 |
| | | S | 10 | 0 | 0 | 0 | 0 | 0 |
| | 0..50 | $S_{acc}$ | 100 | 2 | 0 | 0 | 0 | 0 |
| | | S | 100 | 0 | 0 | 0 | 0 | 0 |
| elevator-signal | 0..5 | $S_{acc}$ | 7 | 10 | 0 | 0 | 0 | 0 |
| | | S | 0 | 0 | 6 | 0 | 15 | 0 |
| | 0..10 | $S_{acc}$ | 7 | 10 | 0 | 0 | 0 | 0 |
| | | S | 0 | 0 | 5 | 0 | 14 | 0 |
| | 0..50 | $S_{acc}$ | 7 | 10 | 0 | 0 | 0 | 0 |
| | | S | 0 | 0 | 6 | 0 | 15 | 0 |
| elevator-w.-door | 0..5 | $S_{acc}$ | 6 | 8 | 0 | 0 | 0 | 0 |
| | | S | 2 | 0 | 4 | 3 | 4 | 5 |
| | 0..10 | $S_{acc}$ | 6 | 8 | 0 | 0 | 0 | 0 |
| | | S | 2 | 0 | 4 | 3 | 5 | 5 |
| | 0..50 | $S_{acc}$ | 6 | 8 | 0 | 0 | 0 | 0 |
| | | S | 2 | 0 | 4 | 3 | 5 | 5 |
| rob-grid-reach-1d | 0..5 | $S_{acc}$ | 2 | 2 | 0 | 0 | 0 | 0 |
| | | S | 0 | 0 | 1 | 1 | 1 | 2 |
| | 0..10 | $S_{acc}$ | 2 | 2 | 0 | 0 | 0 | 0 |
| | | S | 0 | 0 | 1 | 1 | 1 | 2 |
| | 0..50 | $S_{acc}$ | 2 | 2 | 0 | 0 | 0 | 0 |
| | | S | 0 | 0 | 1 | 1 | 1 | 2 |
| rob-grid-reach-2d | 0..5 | $S_{acc}$ | 4 | 4 | 0 | 0 | 0 | 0 |
| | | S | 0 | 0 | 1 | 2 | 2 | 4 |
| | 0..10 | $S_{acc}$ | 4 | 4 | 0 | 0 | 0 | 0 |
| | | S | 0 | 0 | 1 | 2 | 2 | 4 |
| | 0..50 | $S_{acc}$ | 4 | 4 | 0 | 0 | 0 | 0 |
| | | S | 0 | 0 | 1 | 2 | 2 | 4 |
| batch-arbiter-r | 0..5 | $S_{acc}$ | 3 | 2 | 0 | 0 | 0 | 0 |
| | | S | 1 | 0 | 1 | 1 | 1 | 2 |
| | 0..10 | $S_{acc}$ | 3 | 2 | 0 | 0 | 0 | 0 |
| | | S | 1 | 0 | 1 | 1 | 1 | 2 |
| | 0..50 | $S_{acc}$ | 3 | 2 | 0 | 0 | 0 | 0 |
| | | S | 1 | 0 | 1 | 1 | 1 | 2 |
| batch-arbiter-u | 0..5 | $S_{acc}$ | 3 | 2 | 0 | 0 | 0 | 0 |
| | | S | 1 | 0 | 1 | 1 | 1 | 2 |
| | 0..10 | $S_{acc}$ | 3 | 2 | 0 | 0 | 0 | 0 |
| | | S | 1 | 0 | 1 | 1 | 1 | 2 |
| | 0..50 | $S_{acc}$ | 3 | 2 | 0 | 0 | 0 | 0 |
| | | S | 1 | 0 | 1 | 1 | 1 | 2 |
| reversible-lane-r | 0..5 | $S_{acc}$ | 6 | 4 | 0 | 0 | 0 | 0 |
| | | S | 4 | 0 | 1 | 2 | 1 | 4 |
| | 0..10 | $S_{acc}$ | 6 | 4 | 0 | 0 | 0 | 0 |
| | | S | 4 | 0 | 1 | 2 | 1 | 4 |
| | 0..50 | $S_{acc}$ | 6 | 4 | 0 | 0 | 0 | 0 |
| | | S | 4 | 0 | 1 | 2 | 1 | 4 |
| reversible-lane-u | 0..5 | $S_{acc}$ | 6 | 4 | 1 | 0 | 1 | 0 |
| | | S | 4 | 0 | 0 | 0 | 0 | 0 |
| | 0..10 | $S_{acc}$ | 6 | 4 | 2 | 0 | 2 | 0 |
| | | S | 4 | 0 | 0 | 0 | 0 | 0 |
| | 0..50 | $S_{acc}$ | 6 | 4 | 2 | 0 | 2 | 0 |
| | | S | 4 | 0 | 0 | 0 | 0 | 0 |

Table 3: Finite-state experiments details for $\texttt{sweap}_{acc}$ and `sweap`.

## References for the Appendix

[McM06] Kenneth L. McMillan. Lazy abstraction with interpolants. In *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4144 of *Lecture Notes in Computer Science*, pages 123–136. Springer, 2006.