

sweap: Reactive Synthesis for Infinite-State Integer Problems

Downloaded from: https://research.chalmers.se, 2026-11-23 02:11 UTC

Citation for the original published paper (version of record):

Azzopardi, S., Di Stefano, L., Piterman, N. (2026). sweap: Reactive Synthesis for Infinite-State Integer Problems. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 16682 LNCS: 340-353. http://dx.doi.org/10.1007/978-3-032-32519-8_18

N.B. When citing this work, cite the original published paper.

research.chalmers.se offers the possibility of retrieving research publications produced at Chalmers University of Technology. It covers all kind of research output: articles, dissertations, conference papers, reports etc. since 2004. research.chalmers.se is administrated and maintained by Chalmers Library

(article starts on next page)



sweap: Reactive Synthesis for Infinite-State Integer Problems

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

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



Abstract. Recent years have seen a significant increase in the interest in reactive synthesis from specifications that relate to infinite state spaces. We present sweap, a tool for synthesis of infinite-state Linear Integer Arithmetic reactive systems. sweap implements a CEGAR approach, relying on state-of-the-art finite-state synthesis tools as black boxes to solve abstract synthesis problems. sweap supports most common input formalisms for infinite-state reactive-synthesis problems: Temporal Stream Logic Modulo Theories, Reactive Program Games, the bespoke input of the lssy tool, and our own bespoke input. We present a mature version of sweap with novel features: a dual abstraction approach that improves its capabilities in proving unrealisability, support for nondeterministic and unbounded updates, more general initialization of variables, and equirealisable reductions for optimisation. Experimental evaluation shows that sweap outperforms its only competitor in this domain.

Keywords: Reactive Synthesis · CEGAR · Infinite-State Systems

1 Introduction

Reactive synthesis aims at generating correct-by-construction programs from high-level temporal specifications. This problem is framed as a two-player game between a *controller* and an adversarial *environment* [18]. A program, then, is a winning strategy for the controller (if any exists). The traditional setting is that of synthesis over propositions. Given the success of state-of-the-art techniques in this finite domain, many in the community have started moving beyond, to infinite-state domains. Several tools and techniques have been developed, showing surprising effectiveness for an undecidable problem.

First attempts to tackle this problem relied on a counter-example guided abstraction refinement (CEGAR) approach—a finite abstraction of the infinite-state problem is computed and fed to an LTL synthesis tool, and the strategy received is used to refine the abstraction or conclude un/realisability. Approaches

© The Author(s) 2026
E. Darulova et al. (Eds.): CAV 2026, LNCS 16682, pp. 340–353, 2026.
https://doi.org/10.1007/978-3-032-32519-8_18

that originally tackled problems defined in Temporal Stream Logic (TSL) [7], LTL modulo theories [14], and others [20], are of this kind. By design, these focus on learning new safety properties of the underlying theory, relative to the predicates and updates in the objective. Since the cost of finite LTL synthesis is high, these approaches start with essentially no abstraction and build it through refinement. In any case, the basic CEGAR setup proves insufficient for infinite-state problems where liveness is important: in this context, one may never learn enough properties to conclude that, e.g., incrementing an integer repeatedly will eventually make it greater than an arbitrary value.

Later approaches targeting Reactive Program Games (RPG) and ISSY improved using symbolic analysis of game graphs, with quantifier elimination computing winning regions of (symbolically represented infinite) graphs; using *acceleration* to bypass infinite refinement that CEGAR-based approaches had [9,10].

This paper concerns sweap, a CEGAR-based tool for synthesising strategies for LTL games over infinite-state arenas. Our approach [3] goes beyond previous CEGAR-based work, by starting with a full predicate abstraction and automatically learning relevant liveness properties of the game. The abstraction and the LTL objective are fed to an LTL synthesis tool in each CEGAR round. We rely on a novel binary encoding of predicates that reduces the size and computation time of predicate abstraction to polynomial rather than exponential. Thanks to this efficient encoding, the size of the LTL formula is no longer exponential in the number of predicates, reducing the cost of synthesis. This binary encoding enables sweap to eagerly add relevant LTL formulas, to the abstraction, that describe the effect of state updates (inc/decrementing) on the predicate state.

In this paper, we present a mature version of sweap supporting higher-level problem that include numeric inputs, uninitialised variables, and unbounded controller updates. Furthermore, we address sweap's previous limitations with solving unrealisable problems: we describe how sweap can dualise the problem by under-approximating control of the arena, rather than the standard over-approximating. We also describe how we implemented automatic translations of other languages into the sweap, allowing better comparison between different approaches, along with several equirealisable reductions. On benchmarks from literature, we show how these new features increase the capacity of sweap and that sweap outperforms the main tool in this domain on literature benchmarks.

2 The sweap Approach

Problem Formulation. sweap solves infinite-state reactive synthesis problems given as LTL games. The game arena is a symbolic automaton with a finitely many control states. Transitions relate to input, output, and state variables. The objective is an LTL formula over control states and first-order predicates over all state variables, inputs, and outputs. Inputs and state variables may have infinite domains, while outputs are Boolean. Unlike inputs and outputs, state variables, unless updated explicitly (w.r.t. previous values), preserve their value. Arena

transitions describe conditions on the next value of state variables, with nondeterministic updates, including infinite-branching, resolved by the controller.

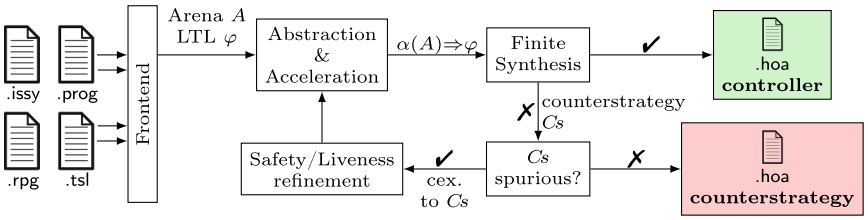


Fig. 1. Overview of sweap's workflow.

CEGAR Approach. sweap takes a standard CEGAR approach [5] (see Fig. 1). Given arena A and objective φ, sweap creates the abstract arena $\alpha(A)$ via predicate abstraction [8], and synthesises for $\alpha(A) \Rightarrow \varphi$ with a finite synthesis engine (Strix [16] or SemML [13,19]). By abstraction soundness, if a controller is found it is also valid for the concrete problem. Spurious counterstrategies (determined through invariant checking), are used to refine the abstraction for a new attempt.

sweap innovates on the CEGAR approach for synthesis [12,14] by using *liveness* refinement, only resorting to safety refinement when the former is unsuccessful. This refinement relies on termination checking of the concrete behaviour induced by spurious lassos in abstract counterstrategies; and then refining the abstract problem with the learned terminating behaviour.

To manage the complexity of predicate abstraction (exponential) and synthesis (doubly exponential), sweap uses a unique and natural binary encoding of predicates. Intuitively, LIA predicates are normalised into the form $t \leq c$, where t is a term over variables, and c a term over constant values. For each term, predicates can be used to define mutually exclusive intervals that cover the number line. For example, given predicates $x \leq 0$, $x \leq 4$, and $x \leq 10$, we can define formulas over these predicates to state that $x \in (-\infty, 0)$, $x \in [0, 4)$, $x \in [4, 10)$, and $x \in [10, \infty)$. As these four statements are mutually exclusive, and their disjunction is a tautology, we can represent them using two Boolean variables. By following these in the abstraction, rather that the single predicates, computing the abstraction becomes polynomial rather than exponential. Other CEGAR approaches create a fresh Boolean variable per predicate [7,14,20], making the Boolean LTL synthesis problem doubly exponential in the number of predicates. In our case, given the binary encoding, the complexity of

Listing 1. sweap example.

```
INPUTS { i : natural }
OUTPUTS {}
STATE VARIABLES { x, y: natural }
CONTROL STATES { q: init }
OBJECTIVE { F y == 0 }
TRANSITIONS
 {q -> q [x > 0 & y > 0
          # x' >= x - 1 &
            y' = y],
  q -> q [x = 0 & y > 0
          # y' >= y - 1 &
            x' = i]    }
```

the abstract synthesis problem is only exponential in the number of predicates. Importantly, sweap further adds acceleration assumptions purely based on these intervals, encoding that incrementing a term infinitely often, without similar decrements, must lead to reaching higher intervals (and, similarly, decrementing leads to lower intervals) [3].

Example 1. Listing 1 is an example of a sweap problem. Transitions are guarded by two conditions, separated by #, respectively over the current state (a guard) and the state change (an update). A transition is triggered when its guard holds, and consequently we require determinism over these guards. A primed variable x' denotes the next value of variable x. The game has an LTL reachability goal stating that variable y eventually has value 0, to be achieved within the constraints of the arena. The arena has two transitions: the first is triggered when both x and y are greater than 0, and lets the controller choose the next value of x within the given constraints. Similarly, the second transition can only be taken when x is 0 and y is greater than 0, and allows some choice in the next value of y, but forces the next value of x to match the current value of input variable i. Transitions describe how the state is allowed to evolve under certain conditions; if no state change is allowed, e.g., when $x = y = 0$, the entire arena just stutters. sweap determines this problem realisable, and returns a controller that simply chooses the smallest allowed value of x and y in each turn.

Other Features. sweap also supports one-shot finite synthesis over its input language, when all the types are finite (Boolean, or bounded intervals). Moreover, it provides arguments to verify returned controllers against the arena and objective through composing these in a nuXmv model. The user can also use this model to simulate the execution of the program against the strategy.

3 Beyond Bounded Inputs and Updates (and More)

The theory behind the prototype version of sweap required fixed initial values of state variables, Boolean inputs, and a finite set of assignments for variable updates. This setting can encode specifications equirealisable to any LIA specification in TSL, RPG, and ISSY (up to addition of extra variables), with excellent comparative performance and success rate [3], but required manual translation of specification and the arena, which is error-prone [4]. We have since extended the syntax of sweap's own input language to enable higher-level specifications (see Listing 1). Here we describe the novel features. Throughout we refer to predicates that refer only to state variables as *state predicates* and those that also refer to inputs as *input predicates*.

Uninitialized State Variables. With unintialised state variables, sweap attempts to find a strategy that works for all possible initial values. When constructing the abstraction, sweap determines all possible satisfiable combinations of state predicates. These options become an initial assumption in the abstraction formula, limiting the abstract environment to a legal initial choice of state predicates.

Unbounded Inputs. Numeric inputs are now a first-class feature and the predicate abstraction now also follows input predicates.

Input predicates may relate inputs with state variables: in a naïve approach, where the environment is unconstrained in its choice of input predicates, it would be allowed to choose illegal predicate combinations. To maintain soundness, we construct the set of all legal input states as follows. First, we collect all satisfiable combinations of input predicates; then, for each combination, we perform quantifier elimination (QE), asking for a necessary condition on state variables for its satisfiability. We restrict the environment to this set of combinations and their conditions, enforcing local legality [20].

Given local legality, during spuriousness checking we can assume that inputs are always set in a way compatible with the input predicates chosen by the environment. Thus spuriousness is always due a wrong state predicate guess. However, such a wrong guess may be due to too coarse input predicates. Consider $x' = i$, where x and i are respectively state and input variables. The environment may wish $x = 7$ to be true in the next step, however it only knows $i \geq 0$. It sets this predicate in the current step and $x = 7$ in the next, as the transition is satisfiable. Spuriousness checking will find a mismatch, since not all models of $i \geq 0$ imply $x = 7$. Then we require refining of input predicates.

For safety refinement, given a counterexample, we previously relied solely on sequence interpolation [15]. This constructs a sequence of formulas corresponding to the counterexample: the initial concrete counterexample state, formulas corresponding to the arena transitions and predicate choices in each time step, and a final formula representing the wrong guesses of the counterstrategy. Note that we do not include the concrete values of input variables in the counterexample, and ground all Boolean variables on their concrete values, giving a purely numeric formula. Variables appearing in these formulas are time-step indexed, giving a sequence of formulas $F_0, ..., F_n, M$, where M is the conjunction of the wrong predicate guesses in the last counterexample state. Interpolants are computed for each time-step i, as formulas over the common alphabets of F_i and F_{i+1}. However, input variables are set freshly in each time-step and are never in the common alphabet: thus, interpolation can never discover input predicates.

When interpolation fails, we search for additional input predicates that would allow the abstract environment to reach the predicate state (M) exposed by the counterexample. This resembles how QE is used in other work [20]. We re-use the formulas constructed for interpolation: $f_{ce} = (F_0 \wedge ... \wedge F_n) \implies M$. As a quick first attempt, sweap proceeds by performing QE over $\forall S_0, ..., S_n . f_{ce}$, where S_i is the set of state variables indexed by i. This gives us a formula over pure input predicates. If no fresh predicates are found, we search for predicates that relate inputs with state variables at each time-step, i.e. to relate the inputs and state variables at step i, we apply QE on the formula $\forall S_{\neq i}, I_{\neq i} . f_{ce}$, where $S_{\neq i}$ is the set of state variables not indexed by i, and similarly $I_{\neq i}$ for inputs. This gives us n formulas. From each we extract a set of fresh input predicates, and choose the smallest non-empty such set to use in the next refinement step.

Unbounded Updates. sweap now supports underspecified updates to state variables, with the non-determinism resolved by the controller (see Listing 1). Canonical sweap arenas are still as in the prototype: transitions with guards and a list of assignments for every state variable ($s := f(V)$ is an assignement, where s is a state variable, and $f(V)$ a formula over all arena variables). The new high-level syntax is translated to this canonical form through a realisability-preserving transformation of the arena, that is essentially the same as we have previously described to encode numeric inputs in the arena state [3]. For every transition that requires unbounded controller choice over variables, we redirect it to a fresh state that allows the controller to increment or decrement the variables, allowing it to continue to the original target only when the update condition holds. A modification of the LTL objective is also required, to ensure the evaluation of the formula stutters when the arena is in the introduced states, and a guarantee is added to ensure the controller eventually leaves such states.

4 Flipping Control of the Arena Abstraction

By default the environment controls the state predicates. However, there are unrealisable problems which do not admit finite counterstrategies with this abstraction choice, leading to non-termination for our approach. We introduce a dual approach, that flips control of state variables to the controller, allowing for termination in these cases. We consider a simple illustrative example.

Example 2. Consider the problem in Listing 2, where $c1$ and $c2$ are controller propositions, i an integer input, and x an integer state variable. In q0 the controller can force any value of x, or move to q1 (if x is non-negative). At q1 the environment chooses an i to increment x with. The controller objective states that eventually the arena moves to q1 and that x is always bounded above by 10.

Listing 2. Dualisation example.

```
...
OBJECTIVE { (F q1) & (G x < 10) }
TRANSITIONS
 { q0 -> q0 [!c1 & !c2 # x' = x + 1]
   q0 -> q0 [!c1 & c2 # x' = x - 1]
   q0 -> q0 [c1 & x < 0 # x' = x]
   q0 -> q1 [c1 & x >= 0 # x' = x]
   q1 -> q1 [true # x' = x + i] }
```

This example is clearly unrealisable: if q1 is reached, there is always an integer i the environment can choose to make x larger than 10 (in fact, $i = 10$ always suffices). However, in our standard flow, where the environment must accurately emit the current predicate state in each turn, the winning counterstrategy is infinite. Consider $x = 0$ initially, and the controller decrements x, leading the environment to respond correctly with $x < 0$. At this point, if the controller chooses to increment x, the predicate abstraction offers two choices for the environment response: $x < 0$ or $x = 0$, i.e. precision is lost. In fact, responding precisely to any possible sequence of inc/decrements requires counting, and thus an infinite amount of state predicates.

Our solution is to provide a setting that switches control of state predicates to the controller. Essentially, in this dual approach, where the controller controls

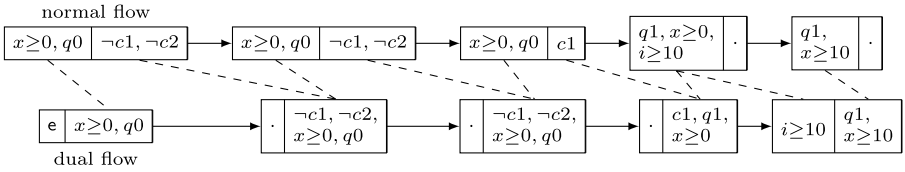


Fig. 2. A trace from the abstraction of Listing 2 and its dualised counterpart.

the state predicates (but the environment remains in control of input predicates) there is a finite counterstrategy: the environment can simply win by setting $i \geq 10$ when `q1` is reached, regardless of the controller choice for state predicates.

To support the dual setting, we change the interaction between the controller and the environment. Normally, initially, the environment chooses initial values of state predicates. Then it chooses inputs followed by the controller choosing outputs. Then, in every step, the environment chooses predicates (based on the previously taken transition) followed by inputs, and the controller replies with outputs. Conceptually, a dual interaction instead includes a choice by the controller for values of state predicates, a choice by the environment of inputs, and another choice by the controller for outputs and predicates (repeatedly in every step). We support this by moving the choice of state predicates to the previous time step. Thus, initially, the controller chooses values for state predicates. Then, in every step, the environment chooses inputs (including input predicates), the controller chooses outputs and updates the state predicates based on the transition. It follows that arena transitions must be executed across time steps.

To enable this in the abstract problem, we change the produced LTL formulas by adding one next operator (X) before any occurrence of input or output variables, and input predicates. Namely, if the original formula is φ and the description of the arena is $\alpha(A)$, the new formula for finite synthesis becomes $\alpha^{dual}(A) \wedge \varphi^{dual}$, with the modifications described. Notice that, in this new problem, it is the responsibility of the controller to obey the constraints of the arena. Furthermore, the initial choices of inputs and outputs (at time 0) do not play a role in the evaluation of such formulas. In the case of undetermined initial values, we include an additional interaction between the environment and the controller at time 0. We check which initial predicate states are feasible and add enough Boolean inputs to the environment so that it can force one such state. We then allow the controller, as explained, to choose the actual predicates. We enforce that the controller copies correctly these values by adding a conjunct φ_{init}. Finally, input predicates, which are determined only by the environment, are also moved to the correct time step (i.e., scoped by X).

We force the environment to choose only legal combinations of input predicates, after the controller acts, by the formula $G(\varphi_{inputs})$, where φ_{inputs} is a disjunction over all legal input predicate combinations, and obtain the final specification $G(\varphi_{inputs}) \implies \varphi_{init} \wedge \alpha^{dual}(A) \wedge \varphi^{dual}$.

Figure 2 shows how an abstract trace for Example 2 changes between the normal and the dual flow, when following the predicates: $x \geq 0$, $x \geq 10$, and $i \geq 10$. For simplicity, we write $x \geq 0$ sfor $x \geq 0 \wedge \neg(x \geq 10)$, and ignore $i \geq 10$ until q1 is reached. Each trace step contains two boxes for the environment and controller's choices in each timestep, respectively. When certain choices do not affect the objective or abstraction, we simply write $\cdot$ in their place. For the dual flow, we introduce fresh propositions to let the environment choose the initial arena state: here we introduce the environment proposition e which when set forces the controller to choose $x \geq 0$ and q0. Note that state predicates (e.g., $x \geq 0$) stay at the same time step, but in the dual flow are chosen by controller. Controller outputs $(c1, c2)$ and original environment inputs (i) are instead pushed to the next state. Hence, if the LTL objective had an assumption $G(q1 \rightarrow (i \leq 0))$, then for the dual problem this would be transformed into $G(q1 \rightarrow X(i \geq 0))$, to maintain the semantics and preserve (un)realisablity.

5 Equirealisable Translations and Reductions

sweap now handles other input languages used by the community, specifically Temporal Stream Logic (TSL), Reactive Program Games (RPGs), and the Issy input format. Problems in each format can be translated to equirealisable problems in the other formats, but they differ in style. In this section, we detail how sweap transforms these into its format, while maintaining equirealisability. We also present equirealisable reductions sweap performs to reduce game size.

Reactive Program Games (RPG). A problem described in RPG is a deterministic symbolic arena with an explicit objective over arena control states (safety, reachability, Büchi, or parity). It includes numeric inputs with a finite set of assignments. The translation of the arena defined to sweap's arena format is thus simply a parsing and compilation problem. The RPG objectives are transformed into equivalent LTL objectives over arena states in the expected manner.

Temporal Stream Logic (TSL). TSL problems are LTL formulas, without any arena, over predicates and a finite set of assignments. To support assignments, we create an arena that allows the controller to choose among them. To keep the predicate abstraction small, rather than considering all the possible assignment combinations, we partition assignments based on a cone-of-influence analysis. Assignments depending on inputs are placed in the first partition. Each arena control state corresponds to a partition, and transitions from it to choosing a combination of assignments in the partition. The controller's non-deterministic choice is resolved with minimal fresh outputs. The TSL formula is modified to serve as the objective, by replacing explicit references to each assignment $u =$ v := expr, with where v is updated at state s by a statement $\neg s\, U\, (s \wedge f_u)$, where f_u is a formula over the fresh outputs that trigger u in the state s (s corresponds to the partition u is in). Predicates are handled similarly, so that they are only evaluated in the initial state of the arena, preserving equi-realisability.

Issy Input Format. An ISSY problem is a list of formula objectives and games, with transitions guarded by update formulas. Formula objectives cannot talk about game arena states, but range over input and state variables, and also over primed state variables, representing their next-state value. These formulas and games each express their own objectives, and are interpreted conjunctively.

We apply some optimisations: (i) Boolean state variables appearing only primed are transformed into (unprimed) controller propositions; and (ii) numeric state variables that are never related with other variables have their predicates given a binary encoding, and the fresh controller propositions are used instead.

ISSY problems do not require an explicit game arena. We check formula-only problems with Spot [6], to identify whether an arena can be extracted directly from an objective, or whether an arena, such as that described for TSL, can be constructed. When these are not possible, we construct a dummy arena that allows the controller to choose any next value of any state variable.

When arenas are explicitly defined, we transform these into one sweap arena. For each arena, we normalise transition conditions into DNF form. Unsatisfiable disjuncts are removed, while the update conditions and guards are separated. Then, for each state, we get a set of guard and update-condition pairs. By Issy semantics, the disjunction of the guards here must be a tautology. Given our optimisations (see (i) above), this may no longer be the case: we resolve this by a transition to a fresh losing state for the uncovered part of the domain. In the presence of multiple games, we compute their cross-product. Then, we resolve any non-determinism in the transition guards with fresh controller propositions, at which point we get a sweap arena. Game arena objectives are expressed equivalently in LTL and conjuncted to the formula objectives. Objectives need to be massaged, replacing any predicate that references primed variables into their next-state representation (e.g., $x > x'$ becomes $X(x_{prev} > x)$).

Equirealisable Reductions. sweap performs several reductions to simplify games into equirealisabile ones, resulting in smaller synthesis problems.

When there is a non-deterministic choice of next variable value, sweap analyses whether the choice is relevant to the game. It is relevant if the objective predicates over the variable, or if a data-flow analysis from the relevant state determines that it is accessed before being assigned a fresh value. Irrelevant variables from a state are stuttered, or, initially, given a default value.

sweap also employs a reachability analysis, removing any unreachable arena control states. Furthermore, it computes the arena's strongly-connected components (SCCs), using these to optimise acceleration formulas. SCCs are also used during the translation of ISSY and RPG problems to detect further losing states (e.g., states from which a Büchi objective is not reachable). Once a losing state is reached, the game stops and the arena state stutters.

6 Experimental Evaluation

In this section, we evaluate the presented version of sweap[1], and how it compares to the state-of-the-art tool lssy [10]. lssy is the only other available infinite-state synthesis tool that handles problems unsolvable by safety refinements alone, and supports the same formalisms as sweap.

sweap refers to a virtual portfolio approach: we attempt to solve the same problem as-is and dualised, always using SemML as the finite synthesis engine, and then take the fastest time among the two. When both configurations reach a verdict, these are consistent across all experiments. All experiments described in this section were run on a GNU/Linux computing cluster equipped with AMD EPYC 9355 processors. Each task was given 4 CPU cores, a time limit of 10 min, and a memory limit of 64 GiB. We run lssy by using a container image obtained from its official Github repository.[2]

Table 1. Results of comparative evaluations of sweap against sweap-strix and lssy. Columns report the number of **S**olved problems, **F**astest, and **U**nique solutions.

(a) All problems in sweap's input format.

Tool	S	F	U
sweap-strix	65	41	0
sweap	**84**	**42**	**19**

Breakdown of problems solved by sweap:

	PROG	ISSY	RPG	TSL
total:	84	91	77	55
only **base**:	52	61	51	36
only **dual**:	12	5	8	11
base faster than **dual**:	8	8	4	2
dual faster than **base**:	12	17	14	6

(b) Problems in ISSY, TSL, and RPG.

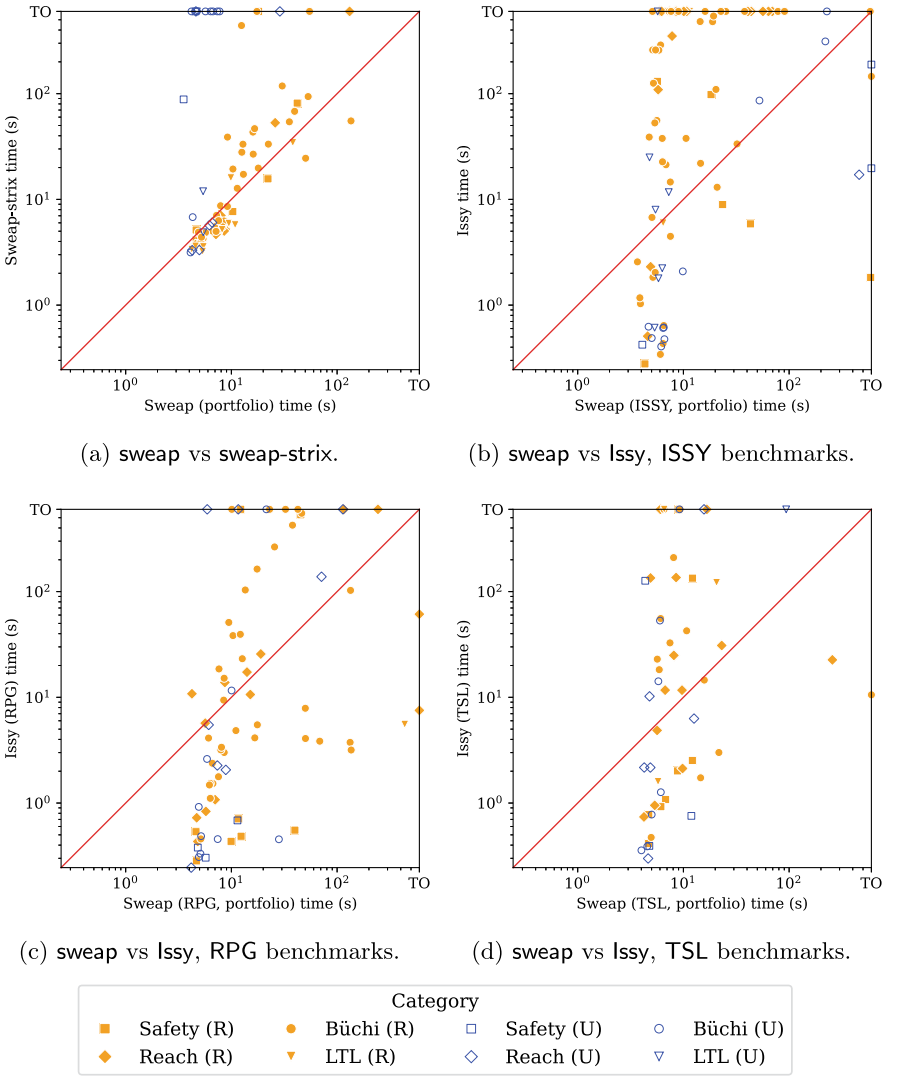
		Realisable			Unrealisable		
Format	Tool	S	F	U	S	F	U
ISSY	lssy	42	18	2	20	**14**	3
	sweap	**70**	**53**	**40**	**21**	5	**4**
RPG	lssy	52	**34**	2	16	**14**	0
	sweap	**57**	25	**7**	**20**	6	**4**
TSL	lssy	32	17	2	14	**10**	0
	sweap	**38**	**21**	**8**	**17**	7	**3**

Evaluation of Novel Features. First, we consider the new version of sweap alongside a configuration sweap-strix that uses Strix as the finite synthesis engine and does not use dualisation. This configuration roughly corresponds to the capabilities that the tool had in its prototype stage, but includes optimisations mentioned in this paper. We perform our comparison over a collection of 95 benchmarks written in sweap's input language (PROG) [3]. 72 of these are realisable. The results are summarised in Table 1a. The new version of sweap solves 84 problems, 19 more than sweap-strix, and is faster on 42 of them. sweap-strix manages to be faster on 41 problems, but there are no problems that it can solve and sweap cannot. We note that the prototype version of sweap required at least 20 min to solve a similar amount of these benchmarks as sweap-strix [3]. Specifically, we are able to solve 7 more problems than sweap-strix just by using SemML; the other 12 extra problems are solved through dualisation, which is also faster on 8 of the 20 problems solved by both workflows. Note that this represents 52% of all 23 unrealisable benchmarks in this set.

[1] The development repository for sweap is: https://github.com/shaunazzopardi/sweap.

[2] https://github.com/phheim/issy/, revision `423bd1a`, as instructed by its developers.

The scatter plot in Fig. 3a compares the runtimes of sweap-strix and sweap. It shows that sweap's superiority over sweap-strix comes from solving unrealisable problems, which are hard without the novel dualisation feature, and from finding faster solutions to mildly complex problems, most of them with Büchi objectives. sweap-strix remains faster on small problems, where the overhead of SemML may be overkill, possibly a sign that sweap may benefit from optimising startup time.



(a) sweap vs sweap-strix. (b) sweap vs Issy, ISSY benchmarks.

(c) sweap vs Issy, RPG benchmarks. (d) sweap vs Issy, TSL benchmarks.

Fig. 3. Each marker corresponds to a synthesis task that both tools carried out without runtime errors. Execution times for our new tool are always mapped to the hoizontal axis. Filled-out and outlined markers correspond to **R**ealisable and **U**nrealisable problems, respectively. Timeout (TO) is 10 min. Further details are provided in [2].

Comparative Evaluation with Issy. We collect a total of 266 reactive synthesis problems over Linear Integer Arithmetic from the literature [3,9–11,17,21] and try to synthesise for them using sweap and Issy. We wish to compare how they perform on the *synthesis problem*, rather than simply returning a verdict without a strategy. However, Issy does not produce counterstrategies, and thus we can only compare on realisable problems. For completeness, we also compare performance on unrealisable problems separately, but we stress that in this case Issy is solving a simpler problem, by only returning a verdict.

Out of 266 problems, 115 are expressed in ISSY (89 realisable), 86 in RPG (66 realisable), and 65 in TSL (45 realisable). Table 1b summarise the results for the two experiments. For realisability, sweap is able to solve more problems than Issy across all formats and find more unique solutions. Issy performs comparatively well, and is faster on more realisable RPG problems. Dualisation does not contribute significantly on realisable problems. For unrealisability, sweap performs relatively well, even though it is solving a harder problem. Dualisation is helpful here, uniquely solving 5 ISSY, 8 RPG, and 11 TSL problems, and is faster on average than the default configuration. Note that whenever both tools manage to solve a problem, their verdicts are always consistent.

Figure 3b–3d compare execution times for the two tools. Though describing conceptually different comparisons, we show the results for both realisable and unrealisable problems together, due to space constraints. In general, sweap incurs a certain overhead at startup, always requiring at least a few seconds to reach a verdict, but does better on larger problems.

We note that the Issy approach is less memory intensive than ours, due to our reliance on finite-synthesis tools. This is shown in Heim and Dimitrova's evaluation of Issy against the prototype version of sweap [10,11], where only 4 GiB/6 GiB of memory were allocated. sweap's default reliance on SemML further increases sweap memory requirements to at least 40 GiB of memory.

7 Conclusion

We presented a mature version of our reactive synthesis tool sweap. The tool supports all the major input formats that recently emerged from infinite-state synthesis research, and its bespoke format has been extended to support high-level specifications. With its novel features and optimisations, the new version of sweap significantly outperforms its earlier prototype [3] and compares positively against the state-of-the-art tool Issy [10].

We believe we can optimise our CEGAR loop by manipulating the abstract game structures directly, instead of formulating them through fresh LTL formulas at every iteration. In [3] we argued that our technique may generalise to linear real arithmetic (LRA), and we are investigating a practical implementation.

Acknowledgments. LDS is supported by the European Union (under grant agreement ID 101212818) and NP is supported by Swedish research council (VR) project (No. 2020-04963) and the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

Data Availibility Statement. Code and benchmark data are archived on Zenodo [1].

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Azzopardi, S., Di Stefano, L., Piterman, N.: Code and benchmark data for "sweap: reactive synthesis for infinite-state integer problems" (2026). https://doi.org/10.5281/zenodo.19684663
2. Azzopardi, S., Di Stefano, L., Piterman, N.: Sweap: reactive synthesis for infinite-state integer problems (extended version) (2026). https://doi.org/10.48550/arXiv.2605.11992
3. Azzopardi, S., Di Stefano, L., Piterman, N., Schneider, G.: Full LTL synthesis over infinite-state arenas. In: Piskac, R., Rakamaric, Z. (eds.) 37th International Conference on Computer Aided Verification (CAV). LNCS, vol. 15934, pp. 274–297. Springer, Cham (2025). https://doi.org/10.1007/978-3-031-98685-7_13
4. Azzopardi, S., Piterman, N., Schneider, G., Stefano, L.D.: LTL synthesis on infinite-state arenas defined by programs (2023). https://doi.org/10.48550/ARXIV.2307.09776, https://doi.org/10.48550/arXiv.2307.09776
5. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5), 752–794 (2003). https://doi.org/10.1145/876638.876643
6. Duret-Lutz, A., et al.: From spot 2.0 to spot 2.10: what's new? CoRR **abs/2206.11366** (2022). https://doi.org/10.48550/ARXIV.2206.11366, https://doi.org/10.48550/arXiv.2206.11366
7. Finkbeiner, B., Klein, F., Piskac, R., Santolucito, M.: Temporal stream logic: synthesis beyond the bools. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 609–629. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_35
8. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-63166-6_10
9. Heim, P., Dimitrova, R.: Solving infinite-state games via acceleration. Proc. ACM Program. Lang. **8**(POPL) (2024). https://doi.org/10.1145/3632899
10. Heim, P., Dimitrova, R.: Issy: a comprehensive tool for specification and synthesis of infinite-state reactive systems. In: Piskac, R., Rakamaric, Z. (eds.) 37th International Conference on Computer Aided Verification (CAV). LNCS, vol. 15934, pp. 298–312. Springer, Cham (2025). https://doi.org/10.1007/978-3-031-98685-7_14
11. Heim, P., Dimitrova, R.: Modular attractor acceleration in infinite-state games. In: Junges, S., Katz, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 32nd International Conference, TACAS. LNCS, vol. 16505, pp. 398–418. Springer, Cham (2026). https://doi.org/10.1007/978-3-032-22752-2_21
12. Henzinger, T.A., Jhala, R., Majumdar, R.: Counterexample-guided control. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) Automata, Languages and Programming, 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30 - July 4, 2003. Proceedings, pp. 886–902. LNCS, Springer, Cham (2003). https://doi.org/10.1007/3-540-45061-0_69

13. Kretínský, J., Meggendorfer, T., Prokop, M., Zarkhah, A.: SemML: enhancing automata-theoretic LTL synthesis with machine learning. In: Gurfinkel, A., Heule, M. (eds.) 31st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 15696, pp. 233–253. Springer, Cham (2025). https://doi.org/10.1007/978-3-031-90643-5_12
14. Maderbacher, B., Bloem, R.: Reactive synthesis modulo theories using abstraction refinement. In: 22nd Conference on Formal Methods in Computer-Aided Design, FMCAD 2022, pp. 315–324. TU Wien Academic Press (2022). https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_38
15. McMillan, K.L.: Lazy abstraction with interpolants. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006). https://doi.org/10.1007/11817963_14
16. Meyer, P.J., Sickert, S., Luttenberger, M.: Strix: explicit reactive synthesis strikes back! In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 578–586. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_31
17. Neider, D., Topcu, U.: An automaton learning approach to solving safety games over infinite graphs. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 204–221. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_12
18. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, 11–13 January 1989, pp. 179–190. ACM Press (1989). https://doi.org/10.1145/75277.75293
19. Prokop, M., Meggendorfer, T., Kretínský, J.: Semml 2.0: synthesizing controllers from LTL. In: 38th International Conference on Computer Aided Verification, CAV (2026)
20. Rodríguez, A., Gorostiaga, F., Sánchez, C.: Counter example guided reactive synthesis for LTL modulo theories. In: Proc. of the 37th Int'l Conference on Computer Aided Verification (CAV'25), Part IV. LNCS, vol. 15934, pp. 224–248. Springer, Cham (2025). https://doi.org/10.1007/978-3-031-98685-7_11, https://link.springer.com/chapter/10.1007/978-3-031-98685-7_11
21. Schmuck, A.K., Heim, P., Dimitrova, R., Nayak, S.P.: Localized attractor computations for infinite-state games. In: Gurfinkel, A., Ganesh, V. (eds.) 36th International Conference on Computer Aided Verification (CAV). LNCS, vol. 14683, pp. 135–158. Springer, Cham (2024). https://doi.org/10.1007/978-3-031-65633-0_7

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

