D-SynMA
No.772459

Reactive synthesis of LTL objectives on infinite arenas

Shaun Azzopardi[1, 3] **Luca Di Stefano**[1, 2]
Nir Piterman[1] Gerardo Schneider[1]

[1]Univ. Gothenburg/Chalmers [2]TU Wien [3]Dedaub

VASSAL Workshop, RV, 15 September 2025

Introduction: Reactive synthesis

- Problem instance given as an LTL formula φ
- APs of φ split into inputs and outputs
  - Inputs controlled by adversarial environment
  - Outputs controlled by “us”

Introduction: Reactive synthesis

- Problem instance given as an LTL formula φ
- APs of φ split into inputs and outputs
  - Inputs controlled by adversarial environment
  - Outputs controlled by “us”

Synthesis problem

Find a strategy (i.e. a Mealy machine) to choose outputs such that every play satisfies φ

Realizability problem

Does such a strategy exist? (✔ / ✗)

✔ → φ is realizable   ✗ → φ is unrealizable

Introduction: Reactive synthesis

- Problem instance given as an LTL formula φ
- APs of φ split into inputs and outputs
  - Inputs controlled by adversarial environment
  - Outputs controlled by “us”

Synthesis problem

Find a strategy (i.e. a Mealy machine) to choose outputs such that every play satisfies φ

Realizability problem

Does such a strategy exist? (✔ / ✗)

✔ $\rightarrow \varphi$ is realizable   ✗ $\rightarrow \varphi$ is unrealizable

Infinite-state synthesis

Go beyond just Boolean variables

Our approach

CEGAR-based synthesis, effective for full LTL specifications.

1. Predicate abstraction $\rightarrow$ finite abstract problem
2. Synthesise: $\begin{cases} \text{If successful, we are done ✔} \\ \text{If unrealisable we get a counterstrategy} \end{cases}$
3. Check counterstrategy: $\begin{cases} \text{if genuine, we are done ✗} \\ \text{if spurious, refine abstraction} \end{cases}$
4. Repeat on refined abstraction.

Our approach

CEGAR-based synthesis, effective for full LTL specifications.

1. Predicate abstraction → finite abstract problem
2. Synthesise: $\begin{cases} \text{If successful, we are done ✔} \\ \text{If unrealisable we get a counterstrategy} \end{cases}$
3. Check counterstrategy: $\begin{cases} \text{if genuine, we are done ✗} \\ \text{if spurious, refine abstraction} \end{cases}$
4. Repeat on refined abstraction.

Main novelty:

- Liveness refinements to avoid enumeration
- Exponential reduction w.r.t predicates
- Acceleration (based on above)

Our setting

Arena: $A = \langle V, \mathbb{E}, \mathbb{C}, val_0, \delta \rangle$

- V: state variables (bools, integers, reals)
- $\mathbb{E}$: environment APs (inputs), $\mathbb{C}$: controller APs (outputs)
- val_0: initial valuation of state variables
- $\delta : Val(V) \times 2^{\mathbb{E} \cup \mathbb{C}} \rightarrow Val(V)$: transition function

Our setting

Arena: $A = \langle V, \mathbb{E}, \mathbb{C}, val_0, \delta \rangle$

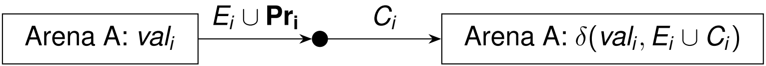
- V: state variables (bools, integers, reals)
- $\mathbb{E}$: environment APs (inputs), $\mathbb{C}$: controller APs (outputs)
- val_0: initial valuation of state variables
- $\delta : Val(V) \times 2^{\mathbb{E} \cup \mathbb{C}} \rightarrow Val(V)$: transition function

Game: $\langle A, \varphi \rangle$

- $\varphi \in LTL(\mathbb{E} \cup \mathbb{C} \cup \mathcal{PR})$
- $\mathcal{PR}$: the set of predicates over V to abstract sets of valuations, e.g., $G\ ((x = 0) \Rightarrow F\ (x = 5))$
- Typically in form *assumptions* $\Rightarrow$ *guarantees*

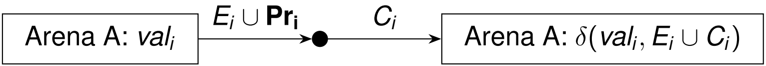
Realisability modulo arena

- **In each move**: environment sets $\mathbb{E}$ and predicates, then controller sets $\mathbb{C}$, and finally the arena's δ updates the variable valuation.



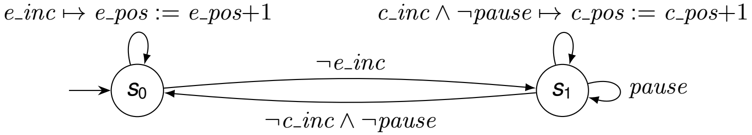
Realisability modulo arena

- **In each move**: environment sets $\mathbb{E}$ and predicates, then controller sets $\mathbb{C}$, and finally the arena's δ updates the variable valuation.



- **Realisability**: There is a Mealy Machine s.t. for each trace: whenever $\forall i.val_i \vDash Pr_i$ the LTL property holds.
- **Unrealisability**: There is a Moore Machine s.t. for each trace: $\forall i.val_i \vDash Pr_i$ and the property does not hold.

Running (Toy) Example - Infinite Race

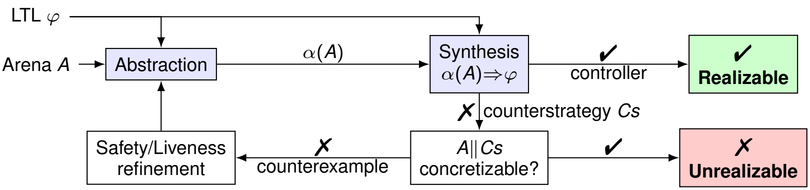


$V = \{e\_pos : \mathbb{N} = 0,$
$\quad c\_pos : \mathbb{N} = 0\}$
$\mathbb{E} = \{e\_inc, pause\}$
$\mathbb{C} = \{c\_inc\}$

Assumptions:
A. $GF(s_1 \wedge \neg pause)$
Guarantees:
G. $GF(s_0 \wedge (c\_pos > e\_pos))$
Goal: A $\implies$ G

- **Assumption**: Environment must ∞ often be in s_1 and not block controller
- **Guarantee**: Controller must ∞ often move back to s_0 with its position (c_pos) larger than the environment's (e_pos).
- **Not encodable as deterministic Büchi game!**

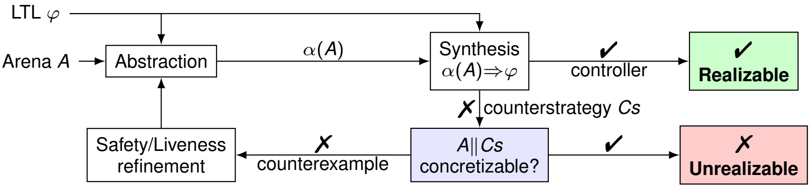
How do we attack this problem? (1/2)



- Abstract each transition t in terms of possible pre- and corresponding post-states: $\alpha(t) \in 2^{Pr} \times 2^{2^{Pr}}$
- Combine into $\alpha(A) \in LTL(\mathbb{E} \cup \mathbb{C} \cup Pr)$
  - Soundly abstracts arena A.
  - Fresh Boolean variable v_p for each predicate p

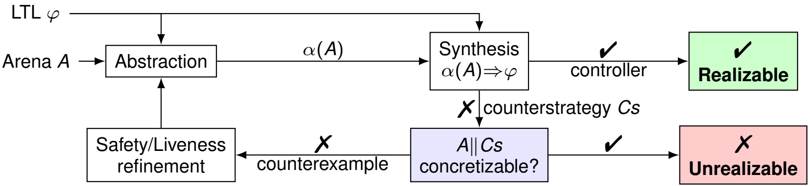
> Controller for abstract problem $\alpha(A) \implies \varphi$ is controller for concrete problem $\implies \langle A, \varphi \rangle$ realizable

How do we attack this problem? (2/2)



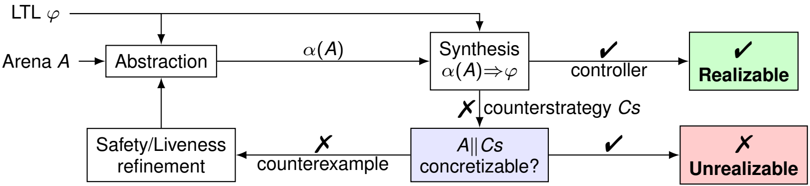
- Invariant checking: $Cs \parallel A \vDash G(\bigwedge_{p \in Pr} v_p \iff p)$
  - Cs chooses the original inputs, driving arena A.
  - $G(\ldots)$ checks correctness of Cs' predicate guesses
- Undecidable (but on benchmarks we never get stuck here).

How do we attack this problem? (2/2)



- Invariant checking: $Cs \parallel A \models G(\bigwedge_{p \in Pr} v_p \iff p)$
    - Cs chooses the original inputs, driving arena A.
    - $G(\ldots)$ checks correctness of Cs' predicate guesses
- Undecidable (but on benchmarks we never get stuck here).

If ✔, then Cs is concrete counterstrategy. $\langle A, \varphi \rangle$ unrealisable

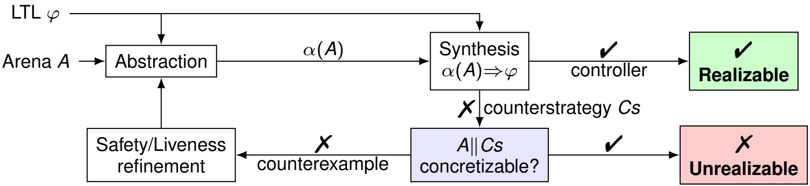


- Invariant checking: $Cs \parallel A \vDash G(\bigwedge_{p \in Pr} v_p \iff p)$
  - *Cs* chooses the original inputs, driving arena *A*.
  - $G(\ldots)$ checks correctness of *Cs*' predicate guesses
- Undecidable (but on benchmarks we never get stuck here).

If ✔, then *Cs* is concrete counterstrategy. $\langle A, \varphi \rangle$ unrealisable

If ✗, then counterexample to concretizability.

How do we attack this problem? (2/2)



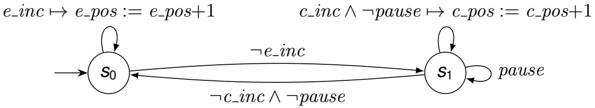
- Invariant checking: $Cs \parallel A \vDash G(\bigwedge_{p \in Pr} v_p \iff p)$
  - *Cs* chooses the original inputs, driving arena *A*.
  - $G(\ldots)$ checks correctness of *Cs*' predicate guesses
- Undecidable (but on benchmarks we never get stuck here).

If ✔, then *Cs* is concrete counterstrategy. $\langle A, \varphi \rangle$ unrealisable

If ✗, then counterexample to concretizability.

If *Cs* not concretisable, this step always terminates and the counterexample is finite.

Safety Refinement

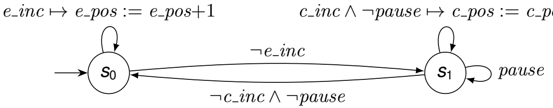


Counterexample ***ce***				Arena Behaviour
CS state	Prog State	Vals	Preds	Triggered Updates
q_0	s_0	$e\_pos = c\_pos = 0$	$\neg(c\_pos > e\_pos)$	
q_1	s_1	$e\_pos = c\_pos = 0$	$\neg(c\_pos > e\_pos)$	$c\_pos := c\_pos + 1$
q_1	s_1	$e\_pos = 0; c\_pos = 1$	$\neg(c\_pos > e\_pos)$	$c\_pos := c\_pos + 1$

- Last state of *ce*, (Pr_j, val_j), will contain at least one $pr \in Pr_j$ s.t. $val_j \not\models pr$.
- From *ce* we get a set of sequence interpolants[1]
- In our case, we initially get $c\_pos - e\_pos = 1$; we add to abstraction to exclude this counterstrategy, and retry.

[1] McMillan, 2006

Safety Refinement

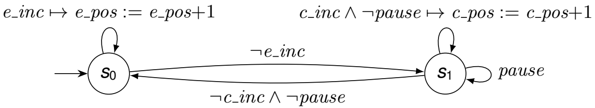


Counterexample *ce*				Arena Behaviour
CS state	Prog State	Vals	Preds	Triggered Updates State
q_0	s_0	$e\_pos = c\_pos = 0$	$\neg(c\_pos > e\_pos)$	
q_1	s_1	$e\_pos = c\_pos = 0$	$\neg(c\_pos > e\_pos)$	$c\_pos := c\_pos + 1$
q_1	s_1	$e\_pos = 0; c\_pos = 1$	$\neg(c\_pos > e\_pos)$	$c\_pos := c\_pos + 1$

- Last state of *ce*, (Pr_j, val_j), will contain at least one $pr \in Pr_j$ s.t. $val_j \not\models pr$.
- From *ce* we get a set of sequence interpolants[1]
- In our case, we initially get $c\_pos - e\_pos = 1$; we add to abstraction to exclude this counterstrategy, and retry.
- More refinements → enumeration → non-termination

[1] McMillan, 2006

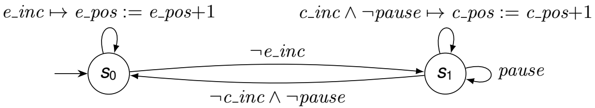
Identifying terminating program loops from *ce*



Counterexample *ce*				Arena Behaviour
CS state	Prog State	Vals	Preds	Triggered Updates
q_0	s_0	$e\_pos = c\_pos = 0$	$\neg(c\_pos > e\_pos)$	
q_1	s_1	$e\_pos = c\_pos = 0$	$\neg(c\_pos > e\_pos)$	$c\_pos := c\_pos + 1$
q_1	s_1	$e\_pos = 0; c\_pos = 1$	$\neg(c\_pos > e\_pos)$	$c\_pos := c\_pos + 1$

- Does *ce* expose failed execution of a lasso in *Cs*?

Identifying terminating program loops from *ce*



Counterexample *ce*				Arena Behaviour
CS state	Prog State	Vals	Preds	Triggered Updates
q_0	s_0	$e\_pos = c\_pos = 0$	$\neg(c\_pos > e\_pos)$	
q_1	s_1	$e\_pos = c\_pos = 0$	$\neg(c\_pos > e\_pos)$	$c\_pos := c\_pos + 1$
q_1	s_1	$e\_pos = 0; c\_pos = 1$	$\neg(c\_pos > e\_pos)$	$c\_pos := c\_pos + 1$

- Does *ce* expose failed execution of a lasso in *Cs*?
- Yes! Self-loop in s_1, triggering $c\_pos := c\_pos + 1$, and expecting $\neg(c\_pos > e\_pos)$ after each iteration.
- I.e., expecting **while**$(\neg(c\_pos > e\_pos))$ $c\_pos := c\_pos + 1$ to not terminate. But it does! (Termination checking)

Liveness refinements - Structural Loop Refinement

- Heuristically generalise precondition (maintaining termination), *true* suffices:

$$\boxed{\textbf{while}(\neg(c\_pos > e\_pos))\ c\_pos := c\_pos + 1}$$

Liveness refinements - Structural Loop Refinement

- Heuristically generalise precondition (maintaining termination), *true* suffices:

$$\textbf{while}(\neg(c\_pos > e\_pos))\ c\_pos := c\_pos + 1$$

- LTL monitor that detects whe we enter/leave this loop ℓ:
  - **Initially not in loop**: $\neg inLoop_\ell$

Liveness refinements - Structural Loop Refinement

- Heuristically generalise precondition (maintaining termination), *true* suffices:

$$\textbf{while}(\neg(c\_pos > e\_pos))\ c\_pos := c\_pos + 1$$

- LTL monitor that detects whe we enter/leave this loop ℓ:
  - **Initially not in loop**: $\neg inLoop_\ell$
  - **In loop iff (loop iteration or (in loop and stutter))**:

$$G\left(\begin{array}{c} whileCond_\ell \wedge loopBody_\ell \\ \vee \\ inLoop_\ell \wedge stutter \end{array} \iff X\ inLoop_\ell\right)$$

Liveness refinements - Structural Loop Refinement

- Heuristically generalise precondition (maintaining termination), *true* suffices:

$$\textbf{while}(\neg(c\_pos > e\_pos))\ c\_pos := c\_pos + 1$$

- LTL monitor that detects whe we enter/leave this loop ℓ:
  - **Initially not in loop**: $\neg inLoop_\ell$
  - **In loop iff (loop iteration or (in loop and stutter))**:

$$G\left(\begin{array}{c} whileCond_\ell \wedge loopBody_\ell \\ \vee \\ inLoop_\ell \wedge stutter \end{array} \iff X\ inLoop_\ell\right)$$

  - **And enforce its termination, or stable non-progress**: $(GF\neg inLoop_\ell) \vee FG(stutter \wedge inLoop_\ell)$

Liveness refinements - Structural Loop Refinement

- Heuristically generalise precondition (maintaining termination), *true* suffices:

$$\textbf{while}(\neg(c\_pos > e\_pos))\ c\_pos := c\_pos + 1$$

- LTL monitor that detects whe we enter/leave this loop ℓ:
  - **Initially not in loop**: $\neg inLoop_\ell$
  - **In loop iff (loop iteration or (in loop and stutter))**:

$$G\left(\begin{array}{c} whileCond_\ell \wedge loopBody_\ell \\ \vee \\ inLoop_\ell \wedge stutter \end{array} \iff X\ inLoop_\ell\right)$$

  - **And enforce its termination, or stable non-progress**: $(GF\neg inLoop_\ell) \vee FG(stutter \wedge inLoop_\ell)$
- Can also handle when loop body is more than 1 state

Liveness refinements - Structural Loop Refinement

- Heuristically generalise precondition (maintaining termination), *true* suffices:

$$\textbf{while}(\neg(c\_pos > e\_pos))\ c\_pos := c\_pos + 1$$

- LTL monitor that detects whe we enter/leave this loop ℓ:
  - **Initially not in loop**: $\neg inLoop_\ell$
  - **In loop iff (loop iteration or (in loop and stutter))**:

$$G\left(\begin{array}{c} whileCond_\ell \wedge loopBody_\ell \\ \vee \\ inLoop_\ell \wedge stutter \end{array} \iff X\ inLoop_\ell\right)$$

  - **And enforce its termination, or stable non-progress**: $(GF\neg inLoop_\ell) \vee FG(stutter \wedge inLoop_\ell)$
- Can also handle when loop body is more than 1 state

In our example, adding this to abstraction suffices to reach a realizable verdict. (+ controller)

Complexity and Decidability

The Elephant in the Room :)

Abstraction	Exponential in no. of predicates $\|P\|$.
Finite Synthesis	$\rightarrow$ 2EXPTIME-complete in $\|P\|$.
Concretisability checking	$\rightarrow$ undecidable in general.
Liveness refinement	$\rightarrow$ undecidable in general.

Can we optimise?

Complexity and Decidability

The Elephant in the Room :)

Abstraction	Exponential in no. of predicates $\|P\|$.
Finite Synthesis	$\rightarrow$ 2EXPTIME-complete in $\|P\|$.
Concretisability checking	$\rightarrow$ undecidable in general.
Liveness refinement	$\rightarrow$ undecidable in general.

Can we optimise?

Yes, we can reduce the number of Bool variables introduced!

(Recall, each predicate p has a corresponding fresh boolean variable v_p in the finite synthesis problem)

Binary Encoding of Numeric Predicates

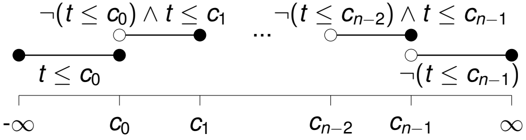
- Massage each predicate into the form $t \leq c$, where t is a term over variables, and c a constant.

Binary Encoding of Numeric Predicates

- Massage each predicate into the form $t \leq c$, where t is a term over variables, and c a constant.
- Collect predicates over term t in P_t, and order them:
  $t \leq c_0, t \leq c_1, ..., t \leq c_{n-1}$, s.t. $c_i < c_{i+1}$
  (for LRA we also need $t < c$).

Binary Encoding of Numeric Predicates

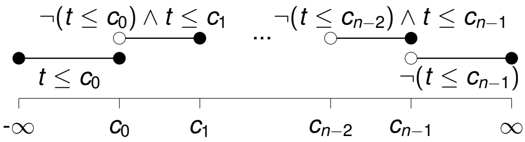
- Massage each predicate into the form $t \leq c$, where t is a term over variables, and c a constant.
- Collect predicates over term t in P_t, and order them: $t \leq c_0, t \leq c_1, ..., t \leq c_{n-1}$, s.t. $c_i < c_{i+1}$ (for LRA we also need $t < c$).
- These predicates partition the number line:



- Can thus encode with $log_2(n+1)$ vars instead of n vars
- e.g., given $x \leq 0, x \leq 1, x \leq 2$, we just need 2 bits:

Partition	Binary Encoding
$x \leq 0$	00
$\neg(x \leq 0) \wedge x \leq 1$	01
$\neg(x \leq 1) \wedge x \leq 2$	10
$\neg(x \leq 2)$	11

Binary Encoding – Complexity



Let $|P_t|$ the number of predicates over term t.

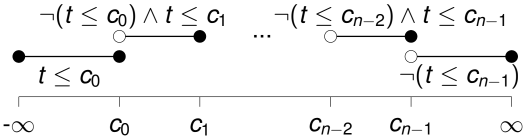
Abstraction

From $2^{2^{\sum_{t \in terms}|P_t|}}$ to $(\prod_{t \in terms}(|P_t| + 1))^2$ SMT calls per transition.

Synthesis

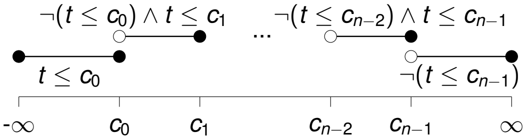
From $2^{2^{\sum_{t \in terms}|P_t|}}$ to $2^{\prod_{t \in terms}|P_t|+1}$.

Acceleration



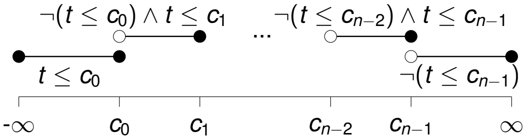
- From $t \leq c_0$, strictly monotonically increasing the value of t means $\neg t \leq c_{n-1}$ becomes true eventually.
- Dually for decreasing.

Acceleration



- From $t \leq c_0$, strictly monotonically increasing the value of t means $\neg t \leq c_{n-1}$ becomes true eventually.
- Dually for decreasing.
- For LRA, need to check that every change is bounded by some value ϵ (for LIA, $\epsilon = 1$).

Acceleration



- From $t \leq c_0$, strictly monotonically increasing the value of t means $\neg t \leq c_{n-1}$ becomes true eventually.
- Dually for decreasing.
- For LRA, need to check that every change is bounded by some value ϵ (for LIA, $\epsilon = 1$).

Define $t_{inc} \stackrel{\text{def}}{=} t_{prev} < t$ and $t_{dec} \stackrel{\text{def}}{=} t < t_{prev}$, then:

- $GF\ t_{inc} \Rightarrow GF(t_{dec} \vee \neg(t \leq c_{n-1}))$
- $GF\ t_{dec} \Rightarrow GF(t_{inc} \vee (t \leq c_0))$

Experimental Design

Benchmarks (only LIA):

- **Safe/Reach/Det. Büchi**: 80 from literature + 1 new
  - Hand-translation into equirealizable problems for our tool.
  - LIA: Equivalent to ours $\rightarrow$ for numeric inputs, we have to add extra states allowing arbitrary increment/decrement.
- **Full LTL benchmark set**: 14 new benchmarks

> To be fair, we only compare with other tools on deterministic Büchi objectives, (although the tools may accept other objectives they will not reach verdict on Full LTL).

Comparison against `raboniel`, `temos`, `rpgSolve`, `rpg-STeLA`, and `tslmt2rpg+rpgSolve`.

16Gb memory, 20 minute timeout, Intel i7-5820K CPU

Our prototype implementation `sweap`[2]

- Handles LIA problems
- Relies on Strix for LTL synthesis, nuXmv for model/invariant checking, CPAChecker for termination checking, MathSat for SMT solving.
- Tool features:
  - Outputs HOA controller/counterstrategy;
  - Results verified against original arena (to protect against possible bugs); and
  - Finite-state model checking (either through described approach, or immediate enumeration+binary encoding)

[2] https://github.com/shaunazzopardi/sweap/

Our prototype implementation `sweap`[2]

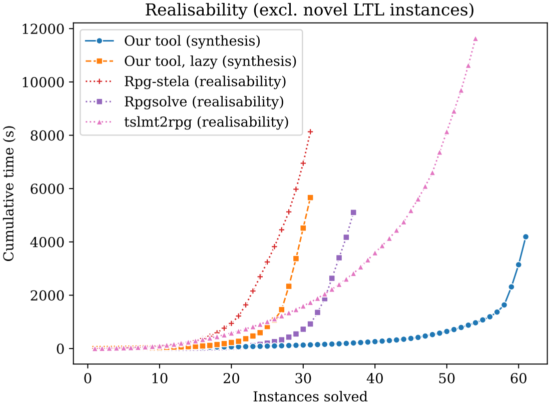
- Handles LIA problems
- Relies on Strix for LTL synthesis, nuXmv for model/invariant checking, CPAChecker for termination checking, MathSat for SMT solving.
- Tool features:
  - Outputs HOA controller/counterstrategy;
  - Results verified against original arena (to protect against possible bugs); and
  - Finite-state model checking (either through described approach, or immediate enumeration+binary encoding)

Configurations for experiments

- `sweap` $\rightarrow$ acceleration enabled, and
- $\texttt{sweap}_{\mathit{lazy}}$ $\rightarrow$ acceleration disabled.

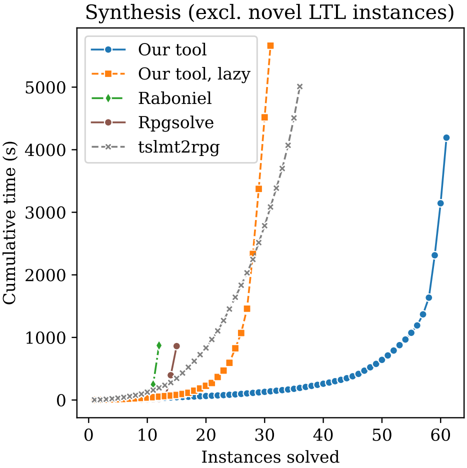
[2] https://github.com/shaunazzopardi/sweap/

Curve lower and more to the right is better.



Comparative Results - Synthesis

Curve lower and more to the right is better.



Evaluation - Full LTL benchmarks

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

Evaluation - Failure Analysis

Some abstractions get too big for synthesis (OOM, timeout)

- With SemML we can solve more, but need more memory.

Evaluation - Failure Analysis

Some abstractions get too big for synthesis (OOM, timeout)

- With SemML we can solve more, but need more memory.

Unrealisable problems with no counterstrategies

- Can also happen w/ det. Büchi (1 problem no tool can solve)
- Solve dualized problem

Evaluation - Failure Analysis

Some abstractions get too big for synthesis (OOM, timeout)

- With SemML we can solve more, but need more memory.

Unrealisable problems with no counterstrategies

- Can also happen w/ det. Büchi (1 problem no tool can solve)
- Solve dualized problem

Lazy approach often misses liveness refinements we can infer from acceleration

Future Work

- Similar approaches to model checking rely on safety refinements + discovering ranking functions:[3]
  - Relatively complete; a similar result here if we can encode ranking functions in LTL?
- Ideally: a finite synthesis tool that allows direct inputting of arena, à la GR[1].
- Direct manipulation of game graph, instead of rebuilding it every iteration. (SemML?)
- Tool "interface" improvements:
  - Support for LRA
  - Native support for numeric inputs and outputs
  - Automatic translation from RPG and TSL, and back (WIP)
- Plan common benchmark format with other teams (WIP)

[3] Balaban, Pnueli, and Zuck, 2005