GenSys: A Scalable Fixed-Point Engine for Maximal Controller Synthesis over Infinite State Spaces

Stanly Samuel
stanly@iisc.ac.in
Indian Institute of Science,
Bengaluru, India

Deepak D'Souza
deepakd@iisc.ac.in
Indian Institute of Science,
Bengaluru, India

Raghavan Komondoor
raghavan@iisc.ac.in
Indian Institute of Science,
Bengaluru, India

ABSTRACT

The synthesis of maximally-permissive controllers in infinite-state systems has many practical applications. Such controllers directly correspond to maximal winning strategies in logically specified infinite-state two-player games. In this paper, we introduce a tool called GenSys which is a fixed-point engine for computing maximal winning strategies for players in infinite-state safety games. A key feature of GenSys is that it leverages the capabilities of existing off-the-shelf solvers to implement its fixed point engine. GenSys outperforms state-of-the-art tools in this space by a significant margin. Our tool has solved some of the challenging problems in this space, is scalable, and also synthesizes compact controllers. These controllers are comparatively small in size and easier to comprehend. GenSys is freely available for use and is available under an open-source license.

CCS CONCEPTS

• **Theory of computation → Automated reasoning**; **Constraint and logic programming**; *Logic and verification.*

KEYWORDS

reactive synthesis, fixed-points, logic, constraint programming

ACM Reference Format:
Stanly Samuel, Deepak D'Souza, and Raghavan Komondoor. 2021. GenSys: A Scalable Fixed-Point Engine for Maximal Controller Synthesis over Infinite State Spaces. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3468264.3473126

1 INTRODUCTION

Reactive systems are control programs that continuously interact with their environment. Examples range from cyber physical systems, robot motion planning systems, wireless sensor networks to bus arbiters, synchronous and distributed programs, to name a few. Synthesizing such systems automatically from temporal specifications without human intervention has been a challenge in software engineering for decades. This problem is of much practical importance, and there are many approaches in the literature that address it. These approaches can be classified broadly as ones that address finite-state synthesis [6, 12, 16], and ones that address infinite-state synthesis [2, 9, 15, 20, 22].

While modelling a reactive system, we can view it as a game between two non co-operating players, with a given winning condition. The *controller* is the protagonist player for whom we wish to find a strategy, such that it can win against any series of moves by the other player, which is the *environment.* A play of the game is an infinite sequence of *steps*, where each step consists of a move by each player.

The aim of synthesis is to find a "winning region" and a winning strategy for the controller if these exist. A winning region consists of a set of states from which the controller will win if it follows its strategy.

In addition to scalability, speed, and size of the synthesized control program, the quality of "maximal permissiveness," which requires the program to allow as many of its moves as possible while still guaranteeing a win, has also gained importance in recent applications. A *maximal* winning region is one that contains all other winning regions. For instance, a maximally permissive program could be used as a "shield" for a neural network based controller [23], and a maximal control program would serve as the ideal shield. Another practical application of reactive synthesis for software engineering is in the domain of model based fuzz testing and has been explored in [14].

In this paper we introduce our tool GenSys, which performs efficient synthesis of *maximal* control programs, for infinite-state systems. Gensys uses a standard fixpoint computation [21] to compute a maximal controller, and does so by leveraging the tactics provided by off-the-shelf solvers like Z3 [7]. Our approach is guaranteed to find a maximal winning region and a winning strategy for any given game whenever the approach terminates.

GenSys is available on GitHub[1].

2 MOTIVATING EXAMPLE

A classic example of a game with infinite states is that of Cinderella-Stepmother [5, 13]. This has been considered a challenging problem for automated synthesis. The game is practically motivated by the minimum backlog problem [1], which is an online problem in the domain of wireless sensor networks.

The game consists of five buckets with a fixed capacity of C units each, arranged in a circular way. The two players of the game are Cinderella, who is the controller, and the Stepmother, who is

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08…$15.00
https://doi.org/10.1145/3468264.3473126

[1] https://github.com/stanlysamuel/gensys

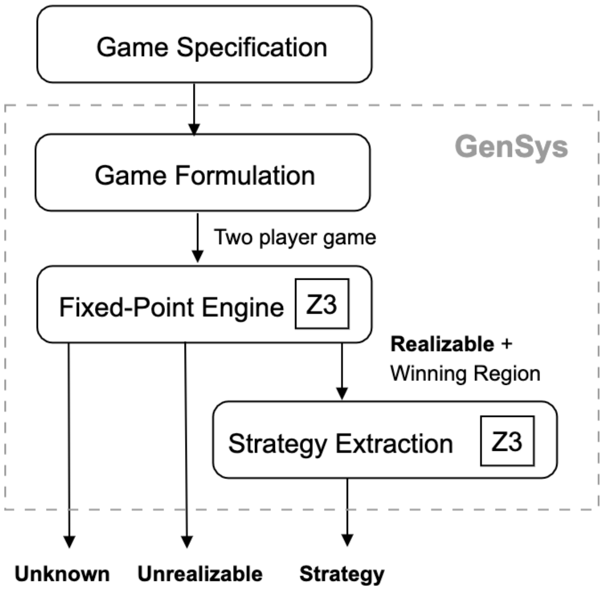

Figure 1: GenSys Tool Architecture

the environment. In each step, Cinderella is allowed to empty any two adjacent buckets, and then the Stepmother tops up the buckets by arbitrarily partitioning one fresh unit of liquid across the five buckets. Cinderella wins if throughout the play none of the buckets overflow; otherwise the Stepmother wins.

The winning region for Cinderella in the Cinderella-StepMother game with bucket capacity three units comprises states where three consecutive buckets have at most two units each, with the sum of the first and third of these buckets being at most 3 (see Table 1).

We will use this game as a running example to illustrate the components of the tool.

3 TOOL DESIGN

GenSys allows users to model a reactive game, to provide a winning condition, and to check automatically if a strategy can be synthesized for the controller to win the game. Figure 1 describes the overall architecture of GenSys. We describe the main components of the tool below.

3.1 Game Specification

The *game specification* is given as input by the user, and consists of four parts: the state space, environment moves, controller moves, and the *winning condition.* A sample game specification is depicted in Figure 2, corresponding to the Cinderella-Stepmother game. The game specification needs to be Python code, and needs to make use of certain API features provided by GenSys. In Figure 2 we have used three buckets for brevity; in our evaluation we use five buckets as that is the standard configuration used in the literature.

State space: Every game consists of a *state space*, where a state consists of a valuation for a set of *variables.* In the example in Figure 2, the variables are named b1, b2, and b3. Intuitively, the values of these variables represent the amount of liquid in each bucket currently. GenSys follows the convention that a variable name of the form "*var_*" represents the "post" value of "*var*" after a move.

Environment move: Lines 6–7 define the state-update permitted to the environment (which would be the StepMother in the example) in each of its moves. In Figure 2, this portion indicates that the StepMother can add a total of one unit of liquid across all three buckets. Semantically, the environment moves can be encoded as a binary relation $Env(s, s')$ on states.

Controller move: This portion defines the state-update permitted to the controller (which would be Cinderella in the example) in each of its moves. Lines 10–19 in the code in Figure 2 indicate that the controller has three alternate options in any of its moves. 'move1' corresponds to emptying buckets b1 and b2, and so on. Semantically, the controller moves can be encoded as a binary relation $Con(s, s')$ on states. In Figure 2, $Con(s, s')$ is a disjunction of each controller move in the Python list *controller_moves.*

Safe Set: We support *safety* winning conditions as of now in GenSys. A safety winning condition is specified by a set of "safe" states in which the controller must forever keep the play in, to win the play. In Lines 24–25, the safe set of states is given by the condition that each bucket's content must be at most the bucket capacity C, which is a command-line parameter to the tool. In other words, there should be no overflows. Semantically, the safe set is a predicate $G(s)$ on states.

```
1 from gensys.helper import *
2 from gensys.fixpoints import *
3 from z3 import *
4
5 #1. Environment moves
6 def environment(b1, b2, b3, b1_, b2_, b3_):
7   return And(b1_ + b2_ + b3_ == b1 + b2 + b3 + 1,
      b1_>=b1, b2_>=b2, b3_>=b3)
8
9 #2. Controller moves
10 def move1(b1, b2, b3, b1_, b2_, b3_):
11   return And(b1_ == 0, b2_ == 0, b3_ == b3)
12
13 def move2(b1, b2, b3, b1_, b2_, b3_):
14     return And( b2_ == 0, b3_ == 0, b1_ == b1)
15
16 def move3(b1, b2, b3, b1_, b2_, b3_):
17     return And( b3_ == 0, b1_ == 0, b2_ == b2)
18
19 controller_moves = [move1, move2, move3]
20
21 #3. Safe set
22 C = sys.argv[1]
23
24 def guarantee(b1, b2, b3):
25     return And(b1 <= C, b2 <= C, b3 <= C, b1 >= 0, b2
       >= 0, b3 >= 0)
26
27 safety_fixedpoint(controller_moves, environment,
      guarantee)
```

Figure 2: Cinderella Game Specification in GenSys

To solve the safety game, the user should call the *safety_fixedpoint* function which implements the fixed-point procedure for this winning condition. This function takes as input moves of both players and the safe set and returns a strategy for the controller, if it exists. More details regarding the procedure is explained in Sections 3.2, 3.3 and 3.4 respectively.

In this prototype version, there is no formal specification language and the game specification needs to be python functions in a specific format, as shown in Fig 2. More details can be found on our tool page[2]. Support for initial variables is not incorporated but is a trivial extension.

3.2 Game Formulation

From the given game specification, this module of our tool formulates one step of the game. This step is represented as the following equation:

$$\begin{aligned} WP(X) \quad \equiv \quad & \exists s'(Con(s,s') \ \wedge G(s') \wedge \\ & \forall s''(Env(s',s'') \implies X(s''))). \end{aligned}$$

A step consists of a move of the controller followed by a move of the environment. The formula above has the state variable s as the free variable. The solution to this formula is the set of states starting from which the controller has a move such that if the environment subsequently makes a move, the controller's move ends in a state that satisfies the given winning condition G, and the environment's move ends in a state that is in a given set of states X. The formula above resembles the weakest pre-condition computation in programming languages. Note that the controller makes the first move [3].

3.3 Fixed-Point Engine

The *winning region* of the game is the greatest solution to the equation in Section 3.2 and can be represented by the greatest fixed-point expression:

$$\nu X. \ (WP(X) \wedge G)$$

It should be noted that for soundness, we require that X be initialized to G as opposed to $True$ in the standard gfp computation.

The winning region represents the set of states starting from which the controller has a way to ensure that only states that satisfy the winning condition G are visited across any infinite series of steps. Our tool computes the solution to the fixed-point equation above using an iterative process (which we describe later in the paper).

Our formulation above resembles similar classical formulations for finite state systems [17, 21]. Those algorithms were guaranteed to terminate due to the finiteness of the state space. This is not true in the case of an infinite state space. Thus, it is possible our approach will not terminate for certain systems. In Figure 1, this possibility is marked with the "Unknown" output. Thus, we are *incomplete* but *sound*. We note that due to the uncomputable nature of the problem [9] there cannot exist a terminating procedure for the problem. However, we have empirically observed that if we bound the variables in $G(s)$, the procedure terminates. For example, for the cinderella specification in Fig 2, if we use the constraint $\bigvee_{i=1}^{3} b_i <= C$ for $G(s)$, the procedure does not terminate.

Table 1: Strategy Synthesized by GenSys for the Cindrella game with bucket size 3

Condition	Move
$0 \le b_1, b_2 \le 3 \ \wedge \ 0 \le b_3, b_4, b_5 \le 2 \ \wedge \ b_3 + b_5 \le 3$	$b_{1\_}, b_{2\_} = 0$
$0 \le b_2, b_3 \le 3 \ \wedge \ 0 \le b_4, b_5, b_1 \le 2 \ \wedge \ b_4 + b_1 \le 3$	$b_{2\_}, b_{3\_} = 0$
$0 \le b_3, b_4 \le 3 \ \wedge \ 0 \le b_5, b_1, b_2 \le 2 \ \wedge \ b_5 + b_2 \le 3$	$b_{3\_}, b_{4\_} = 0$
$0 \le b_4, b_5 \le 3 \ \wedge \ 0 \le b_1, b_2, b_3 \le 2 \ \wedge \ b_1 + b_3 \le 3$	$b_{4\_}, b_{5\_} = 0$
$0 \le b_5, b_1 \le 3 \ \wedge \ 0 \le b_2, b_3, b_4 \le 2 \ \wedge \ b_2 + b_4 \le 3$	$b_{5\_}, b_{1\_} = 0$

Maximality: If the procedure terminates, the winning region is maximal i.e., it contains the exact set of states from where the controller can win. For the proof sketch, assume that the region is not maximal. Then there exists a state which was missed or added to the exact winning region. This is not possible due to the fact that at every step, the formulation in Section 3.2 computes the weakest set of states for the controller to stay in the safe region, against any move of the environment. The detailed proof can be found in Section 8.

3.4 Strategy Extraction

The game is said to be winnable for the controller, or a winning strategy for the controller is said to be *realizable*, if the winning region (computed above) is non-empty.

From the winning region, the strategy can be emitted using a simple logical computation. The strategy is a mapping from subsets of the winning region to specific alternative moves for the controller as given in the game specification, such that every state in the winning region is present in at least one subset, and such that upon taking the suggested move from any state in a subset the successor state is guaranteed to be within the winning region.

In the Cinderella-StepMother game, when there are five buckets and the bucket size C is 3, the strategy that gets synthesized is shown in Table 1.

It is interesting to note that a sound and readable strategy has been synthesized automatically, without any human in the loop.

4 IMPLEMENTATION DETAILS

GenSys is currently in a prototype implementation stage, and serves as a proof of concept for the experimental evaluation that follows. The current version is 0.1.0. Currently GenSys supports safety winning conditions; immediate future work plans include adding support for other types of temporal winning conditions.

GenSys is implemented in Python, and depends on the Z3 theorem prover [7] from Microsoft Research. GenSys has a main loop, in which it iteratively solves for the fixed-point equation in Section 3.3. It first starts with an over-approximation $X = G$, where G is the given safe set, and computes using Z3 a formula that encodes $WP(X)$. It then makes X refer to the formula just computed, re-computes $WP(X)$ again, and so on iteratively, until the formulas denoted by X do not change across iterations. This procedure is described in Section 8.

[2] https://github.com/stanlysamuel/gensys

[3] We also support the scenario where the environment plays first but this is beyond the scope of this paper.

The iterative process above, if carried out naively, can quickly result in very large formulas. To mitigate this issue, we make use of Z3's quantifier elimination tactics. Z3 provides many such tactics; our studies showed that the 'qe2' [4] strategy showed the best results. We believe the quantifer elimination power of Z3 is one of the main reasons for the higher scalability of our approach over other existing approaches.

5 EXPERIMENTAL RESULTS

To evaluate our tool GenSys, we consider the benchmark suite from the paper of Beyene et al. [2], which introduces the Cinderella game as well as some program repair examples. We also consider the robot motion planning examples over an infinite state space introduced by Neider et al. [19].

The primary baseline tool for our comparative evaluation is JSyn-VG [15], whose approach is closely related to ours. Their approach also uses a weakest-precondition like formulation and an iterative approach to compute a fix-point solution. However, their approach uses a "forall-there-exists" formulation of a single step, in contrast to the "there-exists-forall" formulation that we adopt (see the *WP* formulation in Section 3.2). Also, their tool uses a dedicated solver called AE-VAL [10, 11], whereas GenSys uses the standard solver Z3.

We used the latest version of the JSyn-VG, which is available within the JKind model checker (https://github.com/andrewkatis/jkind-1/releases/tag/1.8), for our comparison.

To serve as secondary baselines, we compare our tool with several other tools on the same set of benchmarks as mentioned above. These tools include SimSynth [9] and ConSynth [2], which are based on logic-based synthesis, just like GenSys and JSyn-VG. We also consider the tool DT-Synth [18], which is based on decision tree learning, and the tools SAT-Synth and RPI-Synth, which are based on automata based learning [19]. The numbers we show for SimSynth and ConSynth are reproduced from [9] and [18] respectively, while the numbers for all other tools mentioned above were obtained by us using runs on a machine with an Intel i5-6400 processor and 8 GB RAM. [4] Results for the Cinderella game are not available from the learning-based approaches (i.e., they time out after 900 seconds). SimSynth results are available only for Cinderella among the benchmarks we consider.

Table 2 contains detailed results for the Cinderella game, by considering various values for the bucket size C. It was conjectured by the ConSynth tool authors [2] that the range of bucket sizes between ≥ 1.5 and < 2.0 units is challenging, and that automated synthesis may not terminate for this range. They also mention that this problem was posed by Rajeev Alur as a challenge to the software synthesis community. However, GenSys terminated with a sound result throughout this range. In fact, GenSys was able to scale right upto bucket-size 1.9(20) (i.e., the digit 9 repeated 20 times after the decimal), whereas the state of the art tools time out much earlier. The number of iterations for the fixed-point loop to terminate, i.e., 69, and the time taken to solve, i.e., 31 seconds, affirm that it was indeed challenging to solve for this bucket size. This empirically proves that we can scale to large formula sizes. This is challenging because the formula sizes keep increasing with every iteration of the fixed-point computation.

Table 2: Running times for the Cinderella game for various values of bucket size C. "-" indicates unavailability of data, while ">xm" denotes a timeout after x minutes. R denotes Realizable and U denotes Unrealizable.

C	Out	SimSynth	ConSynth	JSyn-VG	GenSys	
					Time	Iter
3.0	R	2.2s	12m45s	1m26s	0.6s	3
2.5	R	53.8s	**>15m**	1m19s	0.7s	3
2.0	R	68.9s	-	1m6s	0.6s	3
1.9(20)	U	-	-	**>16m**	31.0s	69
1.8	U	**>10m**	-	**>16m**	0.6s	5
1.6	U	1.5s	-	**>16m**	0.4s	4
1.5	U	1.4s	-	14m34s	0.3s	4
1.4	U	0.2s	-	17s	0.2s	3

Table 3: Results on remaining benchmarks. Times are in seconds. >15m denotes a timeout after 15 minutes. Tool name abbreviations: C for ConSynth, J for JSyn-VG, D for DT-Synth, S for SAT-Synth, R for RPI-Synth, G for GenSys.

Benchmark	C	J	D	S	R	G
Repair-Lock	2.5	1.5	0.5	0.6	0.2	0.3
Box	3.7	0.6	0.3	0.3	0.1	0.3
Box Limited	0.4	1.7	0.1	0.4	0.5	0.2
Diagonal	1.9	4.0	2.4	1.34	0.5	0.2
Evasion	1.5	0.5	0.2	81	0.1	0.7
Follow	**>15m**	1.2	0.3	88.9	**>15m**	0.7
Solitary Box	0.4	0.9	0.1	0.3	0.1	0.3
Square 5x5	**>15m**	6.5	2.5	0.6	0.2	0.3

Table 3 shows the results on the other benchmarks. Here also it is clear that GenSys outperforms the other tools in most situations.

SimSynth supports reachability, which is a dual of safety. ConSynth supports safety, reachability and general LTL specifications. The rest of the tools that we consider, including GenSys, natively support safety (and its dual, reachability) winning conditions only.

Regarding maximality, it should be noted that JSyn-VG is the only tool apart from us that synthesizes a maximal controller.

6 FUTURE WORK

The scalability of our approach hints at the potential for addressing more complex winning conditions apart from safety. It would be interesting to address synthesis of maximal controllers for ω-regular specifications, which is a strict superclass of safety, and compare scalability, synthesis time, and controller size for such properties.

[4] We were unable to build SimSynth from source due to the dependency on a very specific version of OCaml. We were unable to get access to ConSynth even after mailing the authors. Thus, we used the numbers for ConSynth from the DT-Synth [18] paper which is the latest paper that evaluates ConSynth. They also describe the difficulty in reproducing the original ConSynth results. We expect the ConSynth results that we have reproduced from the other paper [18] to be accurate, as the numbers for the other tools given in that paper match the numbers we obtained when we ran those tools.

7 CONCLUSION

We have presented the prototype implementation of our tool GenSys. We discussed the design of the tool using a motivating example, and demonstrated scalability of strategy synthesis and the readability of synthesizied strategies. One of the key takeaways is that with the advances in SMT algorithms for quantifier elimination and formula simplification, it is possible to expect scalability for fundamental problems. Tools such as ConSynth, JSyn-VG and SimSynth use external solvers such as E-HSF [3], AE-VAL [10, 11], and SimSat [8] respectively, which appear to slow down the synthesis process. E-HSF requires templates for skolem relations, while AE-VAL restricts the game allowing only the environment to play first. Although SimSynth does not require external templates as a manual input, it follows a two step process where it first synthesizes a template automatically using SimSat, followed by the final strategy synthesis. Our approach does not require an external human in the loop to provide templates, does not pose restrictions on the starting player and is a relatively intuitive approach. Thus, we show an elegant solution that works well in practice. More information about our approach, running the tool and reproducing the results can be found on GitHub[5].

REFERENCES

[1] Michael A. Bender, Sándor P. Fekete, Alexander Kröller, Vincenzo Liberatore, Joseph S.B. Mitchell, Valentin Polishchuk, and Jukka Suomela. 2015. The Minimum Backlog Problem. *Theor. Comput. Sci.* 605, C (Nov. 2015), 51–61. https://doi.org/10.1016/j.tcs.2015.08.027

[2] Tewodros Beyene, Swarat Chaudhuri, Corneliu Popeea, and Andrey Rybalchenko. 2014. A Constraint-Based Approach to Solving Games on Infinite Graphs. *SIGPLAN Not.* 49, 1 (Jan. 2014), 221–233. https://doi.org/10.1145/2578855.2535860

[3] Tewodros A. Beyene, Corneliu Popeea, and Andrey Rybalchenko. 2013. Solving Existentially Quantified Horn Clauses. In *Computer Aided Verification*, Natasha Sharygina and Helmut Veith (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 869–882.

[4] Nikolaj Bjorner and Mikolas Janota. 2015. Playing with Quantified Satisfaction. In *LPAR-20. 20th International Conferences on Logic for Programming, Artificial Intelligence and Reasoning - Short Presentations (EPiC Series in Computing, Vol. 35)*, Ansgar Fehnker, Annabelle McIver, Geoff Sutcliffe, and Andrei Voronkov (Eds.). EasyChair, 15–27. https://doi.org/10.29007/vv21

[5] Marijke Hans L. Bodlaender, Cor A. J. Hurkens, Vincent J. J. Kusters, Frank Staals, Gerhard J. Woeginger, and Hans Zantema. 2012. Cinderella versus the Wicked Stepmother. In *Proceedings of the 7th IFIP TC 1/WG 202 International Conference on Theoretical Computer Science* (Amsterdam, The Netherlands) *(TCS'12)*. Springer-Verlag, Berlin, Heidelberg, 57–71. https://doi.org/10.1007/978-3-642-33475-7_5

[6] Aaron Bohy, Véronique Bruyere, Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. 2012. Acacia+, a tool for LTL synthesis. In *International Conference on Computer Aided Verification*. Springer, 652–657.

[7] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[8] Azadeh Farzan and Zachary Kincaid. 2016. Linear Arithmetic Satisfiability via Strategy Improvement. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence* (New York, New York, USA) *(IJCAI'16)*. AAAI Press, 735–743.

[9] Azadeh Farzan and Zachary Kincaid. 2017. Strategy Synthesis for Linear Arithmetic Games. *Proc. ACM Program. Lang.* 2, POPL, Article 61 (Dec. 2017), 30 pages. https://doi.org/10.1145/3158149

[10] Grigory Fedyukovich, Arie Gurfinkel, and Natasha Sharygina. 2015. Automated discovery of simulation between programs. In *Logic for Programming, Artificial Intelligence, and Reasoning*. Springer, 606–621.

[11] Grigory Fedyukovich, Arie Gurfinkel, and Natasha Sharygina. 2016. Property directed equivalence via abstract simulation. In *International Conference on Computer Aided Verification*. Springer, 433–453.

[12] Bernd Finkbeiner and Sven Schewe. 2013. Bounded synthesis. *International Journal on Software Tools for Technology Transfer* 15, 5 (2013), 519–539.

[13] Antonius Hurkens, Cor Hurkens, and Gerhard Woeginger. 2011. How Cinderella Won the Bucket Game (and Lived Happily Ever After). *Mathematics Magazine* 84 (10 2011), 278–283. https://doi.org/10.4169/math.mag.84.4.278

[14] Andreas Katis, Grigory Fedyukovich, Jeffrey Chen, David Greve, Sanjai Rayadurgam, and Michael W. Whalen. 2020. Synthesis of Infinite-State Systems with Random Behavior. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 250–261.

[15] Andreas Katis, Grigory Fedyukovich, Huajun Guo, Andrew Gacek, John Backes, Arie Gurfinkel, and Michael W. Whalen. 2018. Validity-Guided Synthesis of Reactive Systems from Assume-Guarantee Contracts. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dirk Beyer and Marieke Huisman (Eds.). Springer International Publishing, Cham, 176–193.

[16] Michael Luttenberger, Philipp J. Meyer, and Salomon Sickert. 2019. Practical synthesis of reactive systems from LTL specifications via parity games. *Acta Informatica* 57, 1-2 (Nov 2019), 3–36. https://doi.org/10.1007/s00236-019-00349-3

[17] Oded Maler, Amir Pnueli, and Joseph Sifakis. 1995. On the synthesis of discrete controllers for timed systems. In *Annual symposium on theoretical aspects of computer science*. Springer, 229–242.

[18] Daniel Neider and Oliver Markgraf. 2019. Learning-Based Synthesis of Safety Controllers. In *2019 Formal Methods in Computer Aided Design (FMCAD)*. 120–128. https://doi.org/10.23919/FMCAD.2019.8894254

[19] Daniel Neider and Ufuk Topcu. 2016. 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*. Springer-Verlag, Berlin, Heidelberg, 204–221. https://doi.org/10.1007/978-3-662-49674-9_12

[20] Matthias Rungger and Majid Zamani. 2016. SCOTS: A tool for the synthesis of symbolic controllers. In *Proceedings of the 19th international conference on hybrid systems: Computation and control*. 99–104.

[21] Wolfgang Thomas. 1995. On the synthesis of strategies in infinite games. In *Annual Symposium on Theoretical Aspects of Computer Science*. Springer, 1–13.

[22] Tichakorn Wongpiromsarn, Ufuk Topcu, Necmiye Ozay, Huan Xu, and Richard M Murray. 2011. TuLiP: a software toolbox for receding horizon temporal logic planning. In *Proceedings of the 14th international conference on Hybrid systems: computation and control*. 313–314.

[23] He Zhu, Zikang Xiong, Stephen Magill, and Suresh Jagannathan. 2019. An inductive synthesis framework for verifiable reinforcement learning. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 686–701. https://doi.org/10.1145/3314221.3314638

[5] https://github.com/stanlysamuel/gensys

8 APPENDIX

8.1 Safety Procedure

Algorithm 1 computes the greatest solution to the equation in Section 3.2.

Algorithm 1: Safety Procedure

```
Input   : Game formulation $WP$, Safe region $G$
Output  : Winning region $X$, if algorithm terminates
$X := G$ ;
$W := WP(X) \wedge G$ ;
while $X \nRightarrow W$ do
    $X := W$;
    $W := WP(X) \wedge G$
end
return $X$;
```

Algorithm 1 takes the game formulation as input and returns the winning region for the controller, if it terminates. The winning region is a quantifier free formula in the base theory. At every iteration, the formula $WP(X) \wedge G$ is projected to eliminate quantifiers to return an equivalent quantifier free formula W. The projection operation is intrinsic to the Z3 solver.

8.2 Proof:

We prove the correctness of the Algorithm 1 by reasoning over X.

Lemma 8.1. *At the i'th step of Algorithm 1, X_i is the exact set of states from where the controller has a strategy to keep the game in G for at least i steps.*

Proof: We prove this by induction over the valuations of predicate X at every step in Algorithm 1.

Base case: $i = 0$ and $X_0 = G$. Trivially, the game stays in G and hence it is the set of states from where the controller has a strategy to keep the game in G for at least 0 steps. This is also the weakest (and hence exact) set of states as there are no other states from where the controller can from without making a move.

Inductive step: Assume that the IH holds i.e., X_{i-1} is the exact set of states from where the controller has a strategy to keep the game in G for at least $i - 1$ steps.

X_i is computed as $X_i := WP(X_{i-1}) \wedge G$. From any state $s \in X_i$, the controller can stay in the safe region and ensure reaching X_{i-1} in one step ensuring the fact that it can keep the game in G for at least i steps. Hence, X_i is sound.

Claim: X_i is the weakest.

Proof: Assume a state $s \notin X_i$ and from where the controller can ensure a win. This is not possible because s must be a solution to $WP \wedge G$.

Theorem 8.2 (Soundness). *The predicate X returned by Algorithm 1 is a winning region for the controller.*

Proof: Let $X_{k+1} = X_k$ for some step k in Algorithm 1. Let $s \in X_{k+1}$. From Lemma 8.1, X_k is the exact set of states from where the controller has a strategy to keep the game in G for at least k steps. Similarly, the lemma holds for X_{k+1}. Since $X_{k+1} = WP(X_k)$, from s, the controller can ensure a move to reach X_k in one step. Since $X_{k+1} = X_k$, the controller can ensure a move to reach X_{k+1} in one step as well. As this process can be repeated forever, X_k (and hence, X) is a winning region.

Theorem 8.3 (Maximality). *X returned by Algorithm 1 is the weakest region i.e., no state from where controller can win, is missed.*

Proof: Assume not. Then there exists a state $s \notin X$ from which the controller can keep the game in the safe region for infinite steps. Let the algorithm terminate at some step k. By Lemma 8.1, X_k is the exact set of states from where the controller has a strategy to keep the game in G for at least k steps. Infinite steps also include the k'th step of the algorithm, since k is arbitrary. Hence $s \in X_k$. Contradiction.

From the above two theorems, X is sound and the weakest set of states from where the controller can ensure a move.

8.3 Strategy Extraction:

Once the winning region X has been computed, the strategy for the controller can be extracted in one step. In this paper, we assume that the controller is a disjunction of finite number of moves. Thus, for n moves:

$$Con(s, s') = \bigvee_{i=1}^{n} Move_i(s, s')$$

Let

$$\begin{aligned} WP_i(X) \quad \equiv \quad & \exists s'(Move_i(s, s') \ \wedge G(s') \ \wedge \\ & \forall s''(Env(s', s'') \implies X(s''))). \end{aligned}$$

Given the winning region X, the strategy extraction step computes the condition under which each move of the controller should be played, as follows:

$$Condition_i = WP_i(X) \wedge G$$

For n moves, the strategy returned is a map from conditions to moves as follows:

$$Condition_i \mapsto Move_i, \qquad i \in \{1...n\}$$

Soundness and Maximality of the synthesized strategy: This follows from from the soundness and maximality of the winning region X. The nuance is that the argument now depends on each move $Move_i(s, s')$ of the controller instead of $Con(s, s')$.