A Flexible Toolchain for Symbolic Rabin Games under Fair and Stochastic Uncertainties

Rupak Majumdar[1], Kaushik Mallik[2], Mateusz Rychlicki[3], Anne-Kathrin Schmuck[1], and Sadegh Soudjani[4]

[1] MPI-SWS, Kaiserslautern, Germany, {rupak,akschmuck}@mpi-sws.org
[2] ISTA, Klosterneuburg, Austria, kaushik.mallik@ist.ac.at
[3] School of Computing, University of Leeds, UK, scmkry@leeds.ac.uk
[4] Newcastle University, Newcastle upon Tyne, UK, Sadegh.Soudjani@newcastle.ac.uk

Abstract. We present a flexible and efficient toolchain to *symbolically* solve (standard) Rabin games, fair-adversarial Rabin games, and $2^1/_2$-player Rabin games. To our best knowledge, our tools are the first ones to be able to solve these problems. Furthermore, using the flexible game solvers as back-end, we implement a tool for computing correct-by-construction controllers for stochastic dynamical systems under LTL specifications. Our implementations use the recent theoretical result that all of these games can be solved using the same symbolic fixpoint algorithm but utilizing different, domain specific calculations of the involved predecessor operators. The main feature of our toolchain is the utilization of two programming abstractions: one to separate the symbolic fixpoint computations from the predecessor calculations, and another one to allow the integration of different BDD libraries as back-ends. In particular, we employ a multi-threaded execution of the fixpoint algorithm by using the multi-threaded BDD library Sylvan, which leads to enormous computational savings.

1 Introduction

Piterman and Pnueli [16] derived the currently best known symbolic algorithm for solving two-player Rabin games over finite graphs with a theoretical complexity of $O(n^{k+1}k!)$ in time and space, where n is the number of states and k is the number of pairs in the winning condition. This work did not provide an implementation. In a series of papers [3, 4, 14, 15], Mallik et. al. showed that this symbolic algorithm can be extended to solve different automated design questions for reactive hardware, software, and cyber-physical systems under fair or stochastic uncertainties. The main contribution of their work is to show that these extensions only require a very mild syntactic change of the Piterman-Pnueli fixed-point algorithm (with very little effect on its overall complexity) and domain-specific realizations of two types of predecessor operators used therein.

Using this insight, we present a *toolchain* for *efficient symbolic solution of different extensions of Rabin games*. We have created three inter-connected libraries for solving different parts of the problem from different levels of abstraction. The first library, called `Genie`, offers a set of virtual classes to implement the fixpoint algorithm—abstractly, leaving open (i.e. virtual) the predecessor computation. Alongside, we created two other libraries, called `FairSyn` and `Mascot-SDS`, where `FairSyn` solves fair-adversarial [4] and $2^1/_2$-player Rabin games [3], while `Mascot-SDS` solves abstraction-based control problems [14, 15]. `FairSyn` and `Mascot-SDS` use the optimized fixpoint computation provided by `Genie`, with domain specific implementations of the predecessor operations.

The flexibility of our toolchain comes from two different programming abstractions in `Genie`. Firstly, `Genie` offers multiple high-level optimizations for solving the Rabin fixpoint, such as parallel

execution (requires a thread-safe BDD library like `Sylvan`) and an acceleration technique [13], while abstracting away from the low-level implementations of the predecessor functions. As a result, any synthesis problem using the core Rabin fixpoint of `Genie` can use the optimizations without spending any extra implementation effort. We used these optimizations from `FairSyn` and `Mascot-SDS`, and achieved remarkable computational savings. Secondly, `Genie` offers easy portability of codes from one BDD library to another, which is important as different BDD libraries have different pros and cons, and the choice of the best library depends on the needs. We empirically showed how switching between the two BDD libraries `Sylvan` and `CUDD` impacts the performance of `FairSyn` and `CUDD`: overall, the `Sylvan`-based experiments were significantly faster, whereas the `CUDD`-based experiments consumed considerably lower amount of memory. Using the combined power of multi-threaded BDD operations using `Sylvan` and the optimizations offered by `Genie`, `Mascot-SDS` was between one and three orders of magnitude faster than the state-of-the-art tool in our experiments.

Comparison with Existing Tools: We are not aware of any available tool to directly solve (normal or stochastic) Rabin games *symbolically*. However, it is well-known how to translate *stochastic* Rabin games into (standard) Rabin games [5], and Rabin games into parity games, for which efficient solvers exist, e.g. `oink` [9]. Yet, efficient solutions of stochastic Rabin games via parity games are difficult to obtain, because: (i) the translation from a stochastic Rabin game to a Rabin game involves a quadratic blow-up, and the translation from a Rabin game to a parity game results in an exponential blow-up in the size of the game, (ii) symbolic fixpoint computations become cumbersome very fast for parity games, as the number of vertices and/or colors in the game graph increases, leading to high computation times in practice, and (iii) the only known algorithms capable of handling fair and stochastic uncertainties efficiently are all *symbolic* in nature, while most of the efficient parity game solvers are non-symbolic. Additionally, unlike the Rabin fixpoint, the nesting of the parity fixpoint does not enable parallel execution.

While it is well known that for normal parity games, computational tractability can be achieved by different non-symbolic algorithms, such as Zielonka's algorithm [21], tangle learning [8] or strategy-improvement [18], implemented in `oink`[9], it is currently unclear if and how these algorithms allow for the efficient handling of fair or stochastic uncertainties. We are therefore unable to compare our toolchain to the translational workflow via parity games in a fair manner.

In the area of temporal logic control of stochastic dynamical systems, `Mascot-SDS` has two powerful features: (a) it can handle synthesis for the rich class of omega-regular (infinite-horizon) specifications, and (b) it provides both over- and under-approximations of the solution, thus enabling a quantitative refinement loop for improving the precision of the approximation. The features of `Mascot-SDS` is compared with other tools in the stochastic category of the recent ARCH competition (see the report [1] for the list of participating tools). As concluded in the report of the competition, other state-of-the-art tools in stochastic category are either limited to a fragment of ω-regular specifications or do not provide any indication of the quality of the involved approximations. The only tool [10] that supports ω-regular specifications uses a different alternate non-symbolic approach, against which `Mascot-SDS` fares significantly well in our experiments (see Sec. 4.2). Even if we leave stochasticity aside, our tool implements a new and orthogonal heuristic for multi-threaded computation of Rabin fixpoints, which is not considered by other controller synthesis tools [11].

2 Theoretical Background

We briefly state the synthesis problems our toolchain is solving. We follow the same (standard) notation for two-player game graphs, winning regions, strategies and μ-calculus formulas, as in [4].

2.1 Solving Rabin Games Symbolically

Given a game graph $G = (V, V_0, V_1, E)$, a Rabin game is specified using a set of Rabin pairs $\mathcal{R} = \{(Q_1, R_1,), \ldots, (Q_k, R_k)\}$, with $Q_i, R_i \subseteq V$ for every $i \in [1; k]$, and $\varphi \coloneqq \bigvee_{i\in[1;k]}(\Diamond\Box\neg R_i \wedge \Box\Diamond Q_i)$ being the Rabin acceptance condition. Piterman and Pnueli [16] showed that the winning region of a Rabin game can be computed using the μ-calculus expression given in (2), where the set transformers $Cpre : 2^V \to 2^V$ and $Apre : 2^V \times 2^V \to 2^V$ are defined for every $S, T \subseteq V$ as:

$$Cpre(S) \coloneqq \{v \in V_0 \mid \exists v' \in S \,.\, (v, v') \in E\} \cup \{v \in V_1 \mid \forall v' \in V \,.\, (v, v') \in E \implies v' \in S\}, \quad (1a)$$

$$Apre(S, T) \coloneqq Cpre(T). \quad (1b)$$

The symbolic fixpoint algorithm for solving Rabin games with $\mathcal{R} = \{(Q_1, R_1), \ldots, (Q_k, R_k)\}$ and $K = [1; k]$:

$$\nu Y_{p_0}.\mu X_{p_0}. \bigcup_{p_1\in K} \nu Y_{p_1}.\mu X_{p_1}. \bigcup_{p_2\in K\setminus\{p_1\}} \nu Y_{p_2}.\mu X_{p_2}. \;\ldots\; \bigcup_{p_k\in K\setminus\{p_1,\ldots,p_{k-1}\}} \nu Y_{p_k}.\mu X_{p_k}. \left[\bigcup_{j=0}^{k} \mathcal{C}_{p_j}\right], \quad (2)$$

where

$$\mathcal{C}_{p_j} \coloneqq \left(\textstyle\bigcap_{i=0}^{j} \overline{R}_{p_i}\right) \cap \left[\left(Q_{p_j} \cap Cpre(Y_{p_j})\right) \cup \left(Apre(Y_{p_j}, X_{p_j})\right)\right],$$

and the definitions of *Cpre* and *Apre* are problem specific.

Fair-Adversarial Rabin Games. A Rabin game is called *fair-adversarial* when there is an additional fairness assumption on a set of edges originating from *Player* 1 vertices in G. Let $E^\ell \subseteq E \cap (V_1 \times V)$ be a given set of edges, called the *live* edges. Given E^ℓ and a Rabin winning condition φ, we say that *Player* 0 wins the *fair-adversarial Rabin game* from a vertex v if *Player* 0 wins the (normal) game for the modified winning condition $\varphi^\ell \coloneqq \left(\bigwedge_{e=(v,v')\in E^\ell}(\Box\Diamond v \implies \Box\Diamond e)\right) \implies \varphi$. Based on the results of Banerjee et al. [4], fair-adversarial Rabin games can be solved via (2), by defining for every $S, T \subseteq V$

$$Cpre(S) \coloneqq \{v \in V_0 \mid \exists v' \in S \,.\, (v, v') \in E\} \cup \{v \in V_1 \mid \forall v' \in V \,.\, (v, v') \in E \implies v' \in S\}, \quad (3a)$$

$$Apre(S, T) \coloneqq Cpre(T) \cup \left\{v \in Cpre(S) \cap V_1 \mid \exists v' \in T \,.\, (v, v') \in E^\ell\right\}. \quad (3b)$$

We see that (3) coincides with (1) if E^ℓ is empty.

$2^1/_2$-Player Rabin Games. A $2^1/_2$-player game is played on a game graph (V, V_0, V_1, V_r, E), and the only difference from a 2-player game graph is the additional set of vertices V_r which are called the *random* vertices. The sets V_1, V_2, and V_r partition V. Based on the results of [3] $2^1/_2$-Player rabin games can be solved via (2) by defining for all $S, T \subseteq V$

$$Cpre(S) \coloneqq \{v \in V_0 \mid \exists v' \in S \,.\, (v, v') \in E\} \cup \{v \in V_1 \cup V_r \mid \forall v' \in V \,.\, (v, v') \in E \Rightarrow v' \in S\}, \quad (4a)$$

$$Apre(S, T) \coloneqq Cpre(T) \cup \{v \in Cpre(S) \cap V_r \mid \exists v' \in T \,.\, (v, v') \in E\}. \quad (4b)$$

2.2 Computing Symbolic Controllers for Stochastic Dynamical Systems

A discrete-time stochastic dynamical system S is represented using a tuple (X, U, W, f), where $X \subseteq \mathbb{R}^n$ is a *continuous* state space, U is a *finite* set of control inputs, $W \subset \mathbb{R}^n$ is a *bounded* set

of disturbances, and $f\colon X \times U \to X$ is the nominal dynamics. If $x^k \in X$ and $u^k \in U$ are the state and control input of S at some time $k \in \mathbb{N}$, then the state at the next time step is given by:

$$x^{k+1} = f(x^k, u^k) + w^k, \tag{5}$$

where w^k is the disturbance at time k which is sampled from W using some (possibly unknown) distribution. Without loss of generality we assume that W is centered around the origin, which can be easily achieved by shifting f if needed. A *path* of S originating at $x^0 \in X$ is an infinite sequence of states $x^0x^1\ldots$ for a given infinite sequence of control inputs $u^0u^1\ldots$, such that (5) is satisfied.

Let φ be a given Rabin specification—called the *control objective*—defined using a finite set of predicates over X. For every controller $C\colon X \to U$, the domain of C, written $Dom(C)$, is the set of states from where the property φ can be satisfied with probability 1. For a fixed φ, a controller $\hat{C}$ is called *optimal* if $Dom(\hat{C})$ contains the domain of every other controller C. The problem of computing such an optimal controller for the system in (5) is in general undecidable. Following [14], we compute an approximate solution instead.

This aproximate solution is optained by a discretization of the state space. For this, we assume that the state space X is a closed and bounded subset of the n-dimensional Euclidean space $\mathbb{R}^n$ for some $n > 0$, and consider a grid-based discretization $\widehat{X}$ of X, where $\widehat{X} = \{[\![a, b)\!) \mid a, b \in \mathbb{R}^n = X\}$. One of the key ingredients of our abstraction process is a function $\widehat{f}$ providing hyper-rectangular over-approximation of the one-step reachable set of the nominal dynamics f of the system S: for every grid element $\widehat{x} \in \widehat{X}$, we have $\widehat{f}(\widehat{x}, u) = [\![a', b')\!) \supseteq \{x' \in X \mid \exists x \in \widehat{x}\,.\, x' = f(x, u)\}$. The function $\widehat{f}$ is known to be available for a wide class of commonly used forms of the function f, and in our implementation we assumed that f is mixed-monotone and $\widehat{f}$ is the so-called decomposition function (see standard literature for details [7]).

Given the over-approximation of the nominal dynamics obtained through $\widehat{f}$, we define, respectively, the over- and the under-approximation of the *perturbed* dynamics as $\overline{g}(\widehat{x}, u) \coloneqq W \oplus \widehat{f}(\widehat{x}, u)$ and $\underline{g}(\widehat{x}, u) \coloneqq W \ominus (-\widehat{f}(\widehat{x}, u))$, where $\oplus$ and $\ominus$ respectively denote the Minkowski sum and the Minkowski difference. Next, we transfer $\overline{g}$ and $\underline{g}$ to the abstract state space $\widehat{X}$ to obtain, respectively, the over- and the under-approximation in terms of the *abstract transition* function[5], i.e., $\overline{h}(\widehat{x}, u) \coloneqq \left\{\widehat{x}' \in \widehat{X} \mid \overline{g}(\widehat{x}, u) \cap \widehat{x}' \neq \emptyset\right\}$ and $\underline{h}(\widehat{x}, u) \coloneqq \left\{\widehat{x}' \in \widehat{X} \mid \underline{g}(\widehat{x}, u) \cap \widehat{x}' \neq \emptyset\right\}$. With $\overline{h}$ and $\underline{h}$ available, it was shown by Majumdar et al. [15] that the over-approximation of the optimal controller can be solved by using the fixpoint algorithm in (2), where the predecessor operators are defined for every $S, T \subseteq \widehat{X}$ as

$$\mathit{Cpre}(S) \coloneqq \left\{\widehat{x} \in \widehat{X} \mid \exists u \in U\,.\, \overline{h}(\widehat{x}, u) \subseteq S\right\} \tag{6a}$$

$$\mathit{Apre}(S, T) \coloneqq \left\{\widehat{x} \in \widehat{X} \mid \exists u \in U\,.\, \overline{h}(\widehat{x}, u) \subseteq S \wedge \underline{h}(\widehat{x}, u) \cap T \neq \emptyset\right\}. \tag{6b}$$

3 Implementation Details

We develop three interconnected tools, `Genie`, `FairSyn`, and `Mascot-SDS`, which work in close harmony to implement efficient solvers for the solution of (2) with pre-operators defined via (3), (4) and (6), respectively. The tools use binary decision diagrams (BDD) to symbolically manipulate

[5] Here we assume that $\widehat{f}(\widehat{x}, u) \subseteq X$; otherwise we need to take some extra steps. Details can be found in the work by Majumdar et al. [15].

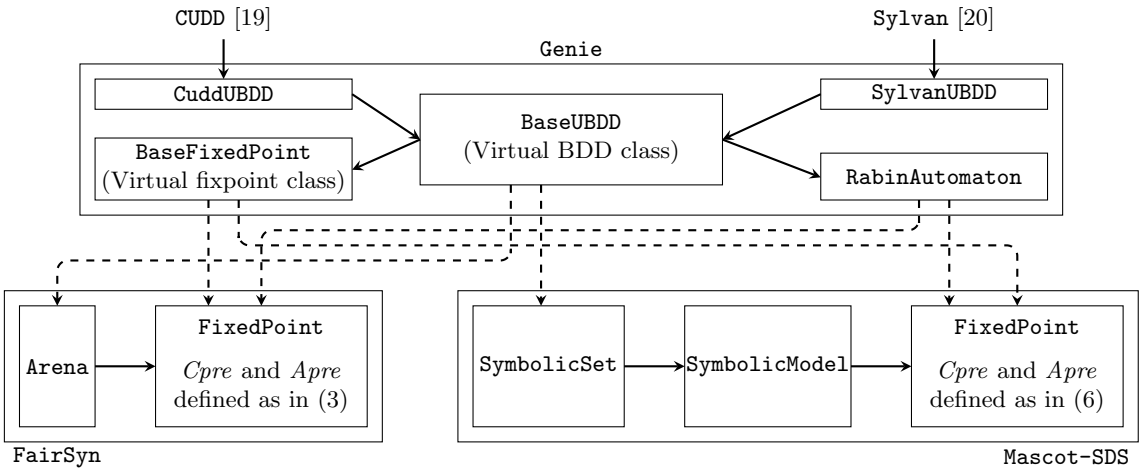

Fig. 1: A schematic diagram of interaction among the three tools. Each block represents one class in the respective tool, and an arrow from class A to class B denotes that B depends on A. The dependency within each tool is shown using solid arrows, while the dependencies of `Mascot-SDS` and `FairSyn` on `Genie` is shown using dashed arrows.

sets of vertices/states of the underlying system, and to manage the BDDs, we offer the flexibility to choose between two of the well-known existing BDD libraries, namely CUDD [19] and Sylvan [20]. The two libraries have their own merits: while CUDD has significantly lower memory footprint, Sylvan offers superior computation speed through multi-threaded BDD operations. Thus, the optimal choice of the library depends on the size of the problem, the computational time limit, and the memory budget, and through our implementation it is possible to choose one or the other by, in some cases, changing only a single single line of code and, in the other cases, changing the value of just one flag. Moreover, we expect that integrating other BDD libraries having the same basic BDD operations in our tools will be easy and seamless—thanks to the programming abstraction offered by `Genie`. Such extensions will possibly bring more diverse set of computational strengths for solving the fundamental synthesis problems that we address.

The tools are primarily written using C++, with some small python scripts implementing parts of visualizations of outputs. The main classes of the three tools and their interactions are depicted in Fig. 1. We briefly describe the core functionalities of the tools in the following.

3.1 `Genie`

`Genie` implements the fixpoint algorithm (2), while abstracting away from the low-level implementation details of the *Cpre* and *Apre* operators. Within `Genie`, there is a second layer of abstraction in the implementation of the fixpoint in the `BaseFixedPoint` class, where we abstract away from the low-level handling of the BDDs. This abstraction is accomplished using the class `BaseUBDD`, a virtual class offering a number of basic BDD operations, whose implementations are in the off-the-shelf BDD library being used. The connection between `BaseUBDD` and the BDD library is achieved through an interface. Currently, we have built interfaces for CUDD and Sylvan, in the classes called `CuddUBDD` and `SylvanUBDD`, respectively. The flexibility to choose between CuddUBDD and SylvanUBDD is illustrated in the small example in Fig. 2. Support for additional BDD libraries can be easily built by creating new interfaces.

```
Cudd mgr;
BDD x = mgr.var();
BDD y = mgr.var();
size_t s = mgr.ReadSize();
cout << "#Nodes = " << s;
```

(a) Using `CUDD`.

```
size_t s = 0;
Bdd x = Bdd::bddVar(0);
s++;
Bdd y = Bdd::bddVar(1);
s++;
cout << "#Nodes = " << s;
```

(b) Using `Sylvan`.

```
// typedef Genie::CuddUBDD UBDD;
typedef Genie::SylvanUBDD UBDD;
UBDD base;
UBDD x = base.var();
UBDD y = base.var();
size_t s = base.nodeSize();
cout << "#Nodes = " << s;
```

(c) Using `Genie` as a wrapper.

Fig. 2: Programs written with `Genie` are easily portable to other BDD libraries: Three example code snippets for creating two BDD variables and then printing the total number of BDD nodes in existence using `CUDD` (left), `Sylvan` (middle) and `Genie` (right). Hard-coded use of specific BDD libraries (left and middle) is not easily portable. Using `Genie` as a wrapper BDD library (left) allows changing BDD libraries by simply commenting/uncommenting the respective "`typedef`" line.

In addition to the flexibility of using different BDD libraries, `Genie` supports two different optimizations for the efficient iterative computation of the Rabin fixpoint in (2)—independently from the actual implementations of the *Apre* and *Cpre* operators. The first optimization is a multi-threaded computation of the Rabin fixpoint, exploiting the fixpoint's inherent parallel structure due to the independence among different sequences of $(p_1, p_2, \ldots)$ used to compute $\bigcup_{j=0}^{k} \mathcal{C}_{p_j}$. The second optimization is an accelerated computation of the Rabin fixpoint, achieved through bookkeeping of intermediate values of the BDD variables. The core of the acceleration procedure for general μ-calculus fixpoints was proposed by Long et al. [13], and the details specific to the fixpoint in (2) can be found in the paper by Banerjee et al. [4].

The Rabin fixpoint is implemented (virtually) in the class `Genie::BaseFixedPoint<UBDD>`, where `UBDD` is a template parameter whose value can be either `CuddUBDD` or `SylvanUBDD` depending on whether we are using `CUDD` or `Sylvan`. To solve the Rabin fixpoint using `Genie`, the user needs to first define a class, call it `FixedPoint<UBDD>`, that is derived from `Genie::BaseFixedPoint<UBDD>` and concretely defines the functions (the predecessors) *Apre* and *Cpre* on an appropriate Rabin game structure of choice. After this, the fixpoint can be solved using the following code:

```
// typedef Genie::CuddUBDD UBDD; // for CUDD
typedef Genie::SylvanUBDD UBDD; // for Sylvan
UBDD base;
FixedPoint<UBDD> fp(...); // construct a FixedPoint object, with appropriate parameters
bool accl = true; // turn acceleration on or off
size_t M = 10; // acceleration parameter (cache size)
UBDD initial_seed = base.one(); // initial over-approximation of winning region (for
    speedup, if unavailable then choose base.one() which is everything)
int verbose = 0; // verbosity
// UBDD result = fp.Rabin(accl, M, initial_seed, verbose); // for sequential solving
UBDD result = fp.Rabin(accl, M, initial_seed, verbose, Genie::ParallelRabinRecurse); //
    for parallel fixpoint solving
```

`Genie` also offers an auxiliary virtual class called `RabinAutomaton` for modeling Rabin automata, which turns out to be helpful when we specify winning conditions and control objectives, respectively, using `FairSyn` and `Mascot-SDS`.

3.2 FairSyn

The core of FairSyn is written as a header-only library, which offers the infrastructure to solve (2) with pre-operators defined via (3) and (4). The main component of FairSyn is the class FixedPoint, which derives from the class BaseFixedPoint from Genie, and implements the concrete definitions of *Cpre* and *Apre* in (3) and (4).

How to use: For computing the winning region and the winning strategy in a fair-adversarial Rabin game (resp. a $2^1/_2$-player Rabin game) using FairSyn, one needs to write a program to create the game as a FixedPoint object. One possible way of constructing a FixedPoint object is through a synchronous product of a game graph (an object of class Arena) and a specification Rabin automaton (an object of class RabinAutomaton) with an input alphabet of sets of nodes of the Arena object. Following is a snippet:

```
// typedef Genie::CuddUBDD UBDD; // use this for CUDD
typedef Genie::SylvanUBDD UBDD; // use this for Sylvan
UBDD base;
...
Arena<UBDD> A(base, vars, nodes, sys_nodes, env_nodes, edges, live_edges); // the game
    graph
RabinAutomaton<UBDD> R(base, vars, inp_alphabet, filename); // the specification automaton
FixedPoint<UBDD> Fp(base, "under", A, R); // the synchronous product
// UBDD strategy = Fp.Rabin(true, 20, Fp.nodes_, 0); // sequential fixpoint solver
UBDD strategy = Fp.Rabin(true, 20, Fp.nodes_, 0, Genie::ParallelRabinRecurse); // parallel
    fixpoint solver
...
```

where vars is a (possibly initially empty) set of integers which will contain the set of newly created BDD variables, nodes, sys_nodes, and env_nodes are, respectively, vectors of indices of various types of vertices, edges and live_edges are, respectively, vectors of the respective types of edges, inp_alphabet is a std::map object that maps input symbols of the Rabin automaton to the respective BDDs representing sets of nodes in the Arena, and filename is the name of the file in which the Rabin automaton is stored (using the standard HOA format [2]). The game is solved by calling Fp.Rabin, a member function of the Genie::BaseFixedPoint class (see Sec. 3.1).

3.3 Mascot-SDS

The core of Mascot-SDS is also written as a header-only library. It is built on top of the well-known tool called SCOTS [17], with several classes of Mascot-SDS still retaining their original identities from SCOTS, owing to the close similarity of the basic uniform grid-based abstraction used in both tools. The main difference between the two tools is that Mascot-SDS synthesizes controllers for *stochastic* systems, while SCOTS synthesizes controllers for only *non-stochastic* systems.

The two main classes of Mascot-SDS are called SymbolicSet and SymbolicModel, which respectively model the abstract spaces obtained through uniform grid-based discretizations (like $\widehat{X}$ in Sec. 2.2) and the abstract transition relations ($\overline{h}$ and $\underline{h}$ in Sec. 2.2). The abstract transition relations are computed using an auxiliary class called SymbolicModelMonotonic (not shown in Fig. 1). Notice that we offer the flexibility to use both CUDD and Sylvan while creating objects from SymbolicSet and SymbolicModel. A FixedPoint object is a child of the class BaseFixedPoint from Genie, which is created by taking a synchronous product between a SymbolicModel object and a RabinAutomaton

object specifying the control objective given as user input. The class `FixedPoint` implements the concrete definitions of the *Cpre* and *Apre* operator according to (6).

How to use: To make the basic usage easier, we have written a pair of tools called `Synthesize` and `Simulate` using the library of `Mascot-SDS`. `Synthesize` synthesizes controllers for stochastic dynamical systems whose nominal dynamics is mixed-monotone, and `Simulate` visualizes simulated closed-loop trajectories using the synthesized controller. The inputs to `Synthesize` include the dynamic model of the system and the control objective; the latter can be specified either in LTL or using a Rabin automaton. To use `Synthesize`, simply use the following syntax:

```
<path-to-Synthesize binary>/Synthesize <path-to-input-file>/<input.cfg> <sylvan/cudd flag>
```

where the `<input.cfg>` is an input configuration file containing all the inputs, and the `<sylvan/cudd flag>` is either `1` or `0` depending on whether the parallel version using `Sylvan` is to be run or the sequential version using `CUDD`.

Some of the main ingredients in the `input.cfg` file are: (a) the description of the dynamical system's variable spaces (like state space, input space, etc.) including their discretization parameters, (b) the file where the decomposition function of the nominal dynamics of the system is stored, (c) the absolute value of maximum disturbance, and (d) the specification either as an LTL formula or as the filename where a Rabin automaton is stored (in HOA format [2]). The decomposition function is required to be given as a C-compatible header file so that `Synthesize` can link to (use) this function at runtime (see the `mascot-sds/examples/` directory for examples). When the specification is given as a Rabin automaton (over a labeling alphabet of the system states), the automaton needs to be stored in a file in the HOA format. Alternatively, an LTL specification can be given, along with a mapping between the atomic predicates and the states of the system. In that case `Synthesize` uses `Owl` [12] to convert the LTL specification to a Rabin automaton.

The output of `Synthesize` is a folder called `data` that contains pieces of the controller encoded in BDDs and stored in binary files as well as various metadata information stored in text files. These files can be processed by `Simulate` to visualize simulated closed-loop trajectories of the system. The usage of `Simulate` is similar to `Synthesize`:

```
<path-to-Simulate binary>/Simulate <path-to-input-file>/<input.cfg> <sylvan/cudd flag>
```

where the `input.cfg` file should, in this case, contain information that are required to simulate the closed-loop, like for how many time steps the simulation should run, the python script that will plot the state space predicates (see the examples), etc.

4 Examples

We present experimental results, showcasing practical usability of our tools as well as comparing performances with the state-of-the-art. All the experiments were run on a computer equipped with Intel Xeon E7-8857 v2 48 core processor and 1.5 TB RAM.

4.1 Synthesizing Code-Aware Resource Mangers of OS using `FairSyn`

We consider a case study introduced by Chatterjee et al. [6]. It considers the problem of synthesizing a code-aware resource manager for a network protocol, i.e., multi-threaded program running on a single CPU. The task of the resource manager is to grant different threads accesses to different

shared synchronization resources (mutexes and counting semaphores). The specification is deadlock freedom across all threads at all time while assuming a fair scheduler (scheduling every thread always eventually) and fair progress in every thread (i.e., taking every existing execution branch always eventually). By making the resource-manager code aware, we can avoid deadlocks by utilizing its knowledge about the require and release characteristics of all threads for different resources. Chatterjee et al. [6] showed that this problem can be reduced to the problem of computing the winning strategy in a certain $2^1/_2$-player game. We refer to [4] for more details. In Table 1, we summarize the computational times for both `CUDD` and `Sylvan`-based implementations of `FairSyn`.

Broadcast Queue Capacity	Output Queue Capacity	Number of Vertices	Number of Transitions	Number of Live edges	Number of BDD variables	Computation Time (seconds)	
						`CUDD`	`Sylvan`
1	1	5,307,840	10,135,300	5,124,100	25	255.33	11.40
2	1	21,231,400	40,541,200	20,496,400	27	957.99	29.20
3	1	21,414,100	42,080,300	21,265,900	27	903.01	31.13
1	2	21,340,800	40,879,100	20,834,300	27	1308.09	39.57
1	3	21,559,400	42,756,100	21,772,800	27	1249.37	41.76
2	2	85,363,200	163,516,000	83,337,200	29	5127.93	111.62
3	2	86,061,400	169,673,000	86,415,400	29	5104.20	114.30
2	3	86,237,400	171,024,000	87,091,200	29	5644.09	118.12
3	3	86,870,100	177,181,000	90,169,300	29	6156.57	137.56

Table 1: Performance of `FairSyn` on the code-aware resource management benchmark experiment.

4.2 Synthesizing Controllers for Stochastic Dynamical Systems using `Mascot-SDS`

We use `Mascot-SDS` to synthesize controllers for two different applications.

A Bistable Switch. First, we compare our tool's performance against the state-of-the-art tool called StochasticSynthesis (abbr. SS) [10] on a benchmark example that was proposed by the authors of SS. In this example, there is a 2-dimensional nonlinear bistable switch that is perturbed with bounded stochastic noise. There are two synthesis problems with two different control objectives: one, a safety objective, and, two, a Rabin objective with two Rabin pairs. The model of the system and the control objectives can be found in the original paper [10].

The tool SS uses graph theoretic techniques to solve the controller synthesis problem, which is an alternative approach that is substantially different from our symbolic fixpoint based technique. In Table 2, we summarize the performance of `Mascot-SDS` powered by `CUDD` and `Sylvan`, alongside the performance of SS. Both `Mascot-SDS` and SS compute controllers whose domains under-approximate the optimal controller domains. The second column of Table 2 shows a measure of the approximation error. For every comparable approximation error bound, both versions of `Mascot-SDS` significantly outperformed SS, both time and memory-wise. In fact, `Mascot-SDS` with `Sylvan` was at least an order of magnitude faster in all instances. This is particularly astonishing, since SS uses a sophisticated *lazy* abstraction refinement technique, whereas `Mascot-SDS` uses a plain *uniform* abstraction which is typically computationally expensive. This shows the immense potential of our toolchain; we plan to extend `Mascot-SDS` with lazy gridding, an orthogonal

optimization, in a future releas to make further computational savings. For `Mascot-SDS` itself, as expected, `Sylvan` was significantly faster than `CUDD`. On the other hand, though `Sylvan` used less memory than `CUDD` in the simpler setups (the ones with more error), the memory requirement of `Sylvan` quickly grew and surpassed that of `CUDD` for the more complicated setup.

Spec.	upper bound on approx. error	Total running time			Peak memory footprint		
		`Mascot-SDS`		SS [10]	`Mascot-SDS`		SS [10]
		`CUDD`	`Sylvan`		`CUDD`	`Sylvan`	
φ_1 (1 Rabin pair)	20 %–30 %	11 s	<2 s	27 s	351 MiB	79 MiB	223 MiB
	10 %–20 %	9 s	2 s	43 s	351 MiB	105 MiB	290 MiB
	5 %–10 %	14 s	4 s	1 h 49 min	405 MiB	251 MiB	25 GiB
	0 %–5 %	48 s	10 s	TO	553 MiB	759 MiB	TO
φ_2 (2 Rabin pairs)	20 %–30 %	21 s	<2 s	21 s	324 MiB	40 MiB	202 MiB
	10 %–20 %	26 s	2 s	25 s	371 MiB	80 MiB	203 MiB
	5 %–10 %	37 s	4 s	1 min 17 s	436 MiB	242 MiB	490 MiB
	0 %–5 %	2 min 24 s	13 s	TO	573 MiB	761 MiB	TO

Table 2: Performance comparison between `Mascot-SDS` and StochasticSynthesis (abbreviated as SS) [10] on the bistable switch. Col. 1 shows the specifications considered and the respective numbers of Rabin pairs, Col. 2 shows the approximation error ranges (smaller error means more intense computation), Col. 3, 4, and 5 compare the total running times and Col. 6, 7, and 8 compare the peak memory footprint (as measured using the "time" command) for `Mascot-SDS` with `CUDD`, `Mascot-SDS` with `Sylvan`, and SS respectively. "TO" stands for timeout (5 h of cutoff time).

Table-Serving Robot. We consider the controller synthesis problem for a table-serving robot that needs to satisfy the following specification: $\square\lozenge \mathit{kitchen} \wedge \square\neg \mathit{obtsacle} \wedge (\square\lozenge \mathit{request} \leftrightarrow \square\lozenge \mathit{table})$, where *table*, *kitchen*, *obstacle*, and *request* are predicates over the state space. The robot itself is modeled as the discrete-time abstraction of the standard 3-dimensional Dubins vehicle [14] with an additional (i.e., 4th) dimension that records if a *request*, which is controlled by the environment, is pending. In Table 3, we summarize the computational resources, and, in Fig. 3, we show a simulated closed-loop trajectory that was plotted using our tool `Simulate`. We observe that `Sylvan` was much faster, but `CUDD` consumed much less memory.

	`CUDD`	`Sylvan`
Comp. time	1 h 3 min	2 min 55 s
Peak memory	673 MiB	1.1 GiB

Table 3: Performance of `Mascot-SDS` with `CUDD` and `Sylvan` for the table-serving robot experiment.

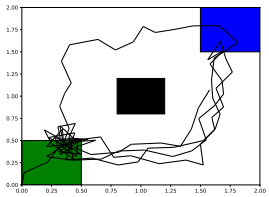

Fig. 3: Simulated closed-loop trajectory of the robot for 100 time steps with *kitchen* (green), *table* (blue), and *obstacle* (black).

References

1. Abate, A., Blom, H., Bouissou, M., Cauchi, N., Chraibi, H., Delicaris, J., Haesaert, S., Hartmanns, A., Khaled, M., Lavaei, A., Ma, H., Mallik, K., Niehage, M., Remke, A., Schupp, S., Shmarov, F., Soudjani, S., Thorpe, A., Turcuman, V., Zuliani, P.: ARCH-COMP21 category report: Stochastic models. In: 8th International Workshop on Applied Verification of Continuous and Hybrid Systems. pp. 55–89 (2021)
2. Babiak, T., Blahoudek, F., Duret-Lutz, A., Klein, J., Křetínskỳ, J., Müller, D., Parker, D., Strejček, J.: The hanoi omega-automata format. In: International Conference on Computer Aided Verification. pp. 479–486. Springer (2015)
3. Banerjee, T., Majumdar, R., Mallik, K., Schmuck, A.K., Soudjani, S.: A direct symbolic algorithm for solving stochastic Rabin games. In: Tools and Algorithms for the Construction and Analysis of Systems: 28th International Conference, Proceedings, Part II. pp. 81–98. Springer (2022)
4. Banerjee, T., Majumdar, R., Mallik, K., Schmuck, A.K., Soudjani, S.: Fast symbolic algorithms for omega-regular games under strong transition fairness. TheoretiCS (to appear) (2023), arXiv preprint arXiv:2202.07480
5. Chatterjee, K., de Alfaro, L., Henzinger, T.A.: The complexity of stochastic Rabin and Streett games. In: Proceedings of the 32nd International Colloquium on Automata, Languages and Programming (ICALP). Lecture Notes in Computer Science, vol. 3580, pp. 878–890. Springer (2005)
6. Chatterjee, K., De Alfaro, L., Faella, M., Majumdar, R., Raman, V.: Code aware resource management. Formal Methods in System Design **42**(2), 146–174 (2013)
7. Coogan, S., Arcak, M.: Efficient finite abstraction of mixed monotone systems. In: Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control. pp. 58–67 (2015)
8. van Dijk, T.: Attracting tangles to solve parity games. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification. pp. 198–215. Springer International Publishing, Cham (2018)
9. van Dijk, T.: Oink: An implementation and evaluation of modern parity game solvers. In: Beyer, D., Huisman, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 291–308. Springer International Publishing, Cham (2018)
10. Dutreix, M., Huh, J., Coogan, S.: Abstraction-based synthesis for stochastic systems with omega-regular objectives. Nonlinear Analysis: Hybrid Systems **45**, 101204 (2022)
11. Geretti, L., Alexandre Dit Sandretto, J., Althoff, M., Benet, L., Chapoutot, A., Chen, X., Collins, P., Forets, M., Freire, D., Immler, F., et al.: ARCH-COMP20 category report: Continuous and hybrid systems with nonlinear dynamics. In: Proc. of the 7th International Workshop on Applied Verification of Continuous and Hybrid Systems. pp. 49–75 (2020)
12. Křetínskỳ, J., Meggendorfer, T., Sickert, S.: Owl: a library for omega-words, automata, and ltl. In: International Symposium on Automated Technology for Verification and Analysis. pp. 543–550. Springer (2018)
13. Long, D.E., Browne, A., Clarke, E.M., Jha, S., Marrero, W.R.: An improved algorithm for the evaluation of fixpoint expressions. In: International Conference on Computer Aided Verification. pp. 338–350. Springer (1994)
14. Majumdar, R., Mallik, K., Schmuck, A.K., Soudjani, S.: Symbolic qualitative control for stochastic systems via finite parity games. IFAC-PapersOnLine **54**(5), 127–132 (2021)
15. Majumdar, R., Mallik, K., Soudjani, S.: Symbolic controller synthesis for büchi specifications on stochastic systems. In: Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Control. pp. 1–11 (2020)
16. Piterman, N., Pnueli, A.: Faster solutions of Rabin and Streett games. In: 21st Annual IEEE Symposium on Logic in Computer Science (LICS'06). pp. 275–284. IEEE (2006)
17. Rungger, M., Zamani, M.: Scots: A tool for the synthesis of symbolic controllers. In: Proceedings of the 19th international conference on hybrid systems: Computation and control. pp. 99–104 (2016)
18. Schewe, S.: An optimal strategy improvement algorithm for solving parity and payoff games. In: Kaminski, M., Martini, S. (eds.) Computer Science Logic. pp. 369–384. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)

19. Somenzi, F.: Cudd: Cu decision diagram package release 3.0.0 (2015). URL: https://github.com/ivmai/cudd
20. Van Dijk, T., Van De Pol, J.: Sylvan: Multi-core decision diagrams. In: Tools and Algorithms for the Construction and Analysis of Systems: 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings 21. pp. 677–691. Springer (2015)
21. Zielonka, W.: Infinite games on finitely coloured graphs with applications to automata on infinite trees. Theoretical Computer Science **200**(1), 135–183 (1998). https://doi.org/https://doi.org/10.1016/S0304-3975(98)00009-7, https://www.sciencedirect.com/science/article/pii/S0304397598000097