SySCoRe: Synthesis via Stochastic Coupling Relations

Birgit van Huijgevoort
Eindhoven University of Technology
The Netherlands
b.c.v.huijgevoort@tue.nl

Oliver Schön
Newcastle University
United Kingdom
o.schoen2@newcastle.ac.uk

Sadegh Soudjani
Newcastle University
United Kingdom
sadegh.soudjani@newcastle.ac.uk

Sofie Haesaert
Eindhoven University of Technology
The Netherlands
s.haesaert@tue.nl

ABSTRACT

We present SySCoRe, a MATLAB toolbox that synthesizes controllers for stochastic continuous-state systems to satisfy temporal logic specifications. Starting from a system description and a co-safe temporal logic specification, SySCoRe provides all necessary functions for synthesizing a robust controller and quantifying the associated formal robustness guarantees. It distinguishes itself from other available tools by supporting nonlinear dynamics, complex co-safe temporal logic specifications over infinite horizons and model-order reduction. To achieve this, SySCoRe generates a finite-state abstraction of the provided model and performs probabilistic model checking. Then, it establishes a probabilistic coupling to the original stochastic system encoded in an approximate simulation relation, based on which a lower bound on the satisfaction probability is computed. SySCoRe provides non-trivial lower bounds for infinite-horizon properties and unbounded disturbances since its computed error does not grow linearly in the horizon of the specification. It exploits a tensor representation to facilitate the efficient computation of transition probabilities. We showcase these features on several benchmarks and compare the performance of the tool with existing tools.

KEYWORDS

Temporal logic control, stochastic systems, approximate simulation relation, dynamic programming, coupling relations

ACM Reference Format:
Birgit van Huijgevoort, Oliver Schön, Sadegh Soudjani, and Sofie Haesaert. 2023. SySCoRe: Synthesis via Stochastic Coupling Relations. In *Proceedings of the 26th ACM International Conference on Hybrid Systems: Computation and Control (HSCC '23), May 09–12, 2023, San Antonio, TX, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3575870.3587123

This work is supported by the Dutch NWO Veni project CODEC (18244), the UK EPSRC New Investigator Award CodeCPS (EP/V043676/1), and the Horizon Europe EIC project SymAware (101070802).

This work is licensed under a Creative Commons Attribution-NonCommercial International 4.0 License.

HSCC '23, May 09–12, 2023, San Antonio, TX, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0033-0/23/05.
https://doi.org/10.1145/3575870.3587123

1 INTRODUCTION

The design of provably correct controllers is crucial for the development of safety-critical systems, such as autonomous vehicles and smart energy grids [6, 25]. To this end, methods for synthesizing controllers for dynamical systems that are guaranteed to satisfy temporal logic specifications have gained an increasing amount of attention in the control community [8, 9, 24, 37]. Besides establishing the theory underlying these methods, it is equally important to develop tools that facilitate their application. For stochastic systems, a collection of tools that can perform formal controller synthesis is already available. A subset of these tools include in alphabetical order: AMYTISS [23], FAUST [36], hpnmg [17], HYPEG [30], Mascot-SDS [27], the Modest Toolset [15], ProbReach [34], SReachTools [41], and StocHy [11]. A complete list of these tools with their descriptions and capabilities can be found in the ARCH Competition Report (stochastic category) [1]. These tools perform the computations either using analytical methods or employing statistical model checking. The approaches in the analytical methods can further be divided into abstraction-based [11, 23, 27, 36] and abstraction-free techniques [19, 41]. Abstraction-free techniques are generally suffer less from the curse of dimensionality, but are often limited to simple invariance and reachability specifications. In contrast, abstraction-based tools can be applied to a breath of systems and specifications. A survey on formal verification and control synthesis of stochastic systems is given in [24].

SySCoRe contributes to the category of tools that employ analytical abstraction-based methods. It is a MATLAB toolbox applicable to stochastic nonlinear systems with a possibly *unbounded disturbance*. Furthermore, it can perform the controller synthesis to satisfy arbitrary co-safe specifications that can have *unbounded time horizons*. To this end, it uses the (ϵ, δ)-approximate simulation relation provided in [14], that explicitly designs the coupling between the continuous-state model and its (reduced) finite-state abstraction [39]. Hence, SySCoRe extends the capabilities of the current tools by considering properties that are *unbounded in time* and by considering systems with an *unbounded disturbance*.

SySCoRe is a comprehensive toolbox for temporal logic control of stochastic continuous-state systems, implementing all necessary steps in the control synthesis process. Moreover, it supports *model-order reduction* in the abstraction process with formal error quantification quarantees, which makes it applicable to a larger classes of systems. To increase its computational efficiency, SySCoRe performs computations based on tensors and sparse matrices. Furthermore,

computations based on efficient convex optimizations for polytopic sets are implemented where possible. The tool is developed with a focus on ease of use and extensibility, such that it can easily be adapted to suit individual research purposes. The development of SySCoRe is a step towards solving the tooling need for temporal logic control of stochastic systems as it expands both the class of models and the class of specifications for which abstraction-based methods can provide controllers with formal guarantees.

This tool paper is organized as follows. We discuss in Section 2 the temporal logic control problem and the set-up in SySCoRe. We then give an overview of SySCoRe in Section 3 by introducing the associated functions and classes. Section 4 discusses multiple benchmarks that show the capabilities of SySCoRe and how it compares to existing tools. We end the paper with a summary and a discussion of possible extensions. Throughout, we give the core functions of SySCoRe in framed white boxes and example code in gray boxes.

2 TEMPORAL LOGIC CONTROL

The main purpose of SySCoRe is to perform the complete control synthesis procedure in abstraction-based temporal logic control. It is applicable to discrete-time models with a possibly unbounded stochastic disturbance and synthesizes a controller for satisfying co-safe linear temporal logic specifications that may have an unbounded time horizon. The computational approach is based on the theory of approximate simulation relations [14], the coupling between models [14, 39] and robust dynamic programming mappings [13]. In this section, we introduce the class of models and specifications handled by SySCoRe, and show how to set up the problem. Furthermore, we provide a high-level description of the theory underlying the implementations in SySCoRe.

2.1 Problem parameters

Model. Consider discrete-time systems described by stochastic difference equations

$$M: \begin{cases} x_{t+1} = f(x_t, u_t) + B_w w_t \\ y_t = Cx_t, \quad \forall t \in \{0, 1, 2, \dots\}, \end{cases} \tag{1}$$

with state $x_t \in \mathbb{X}$, input $u_t \in \mathbb{U}$, (unbounded) stochastic disturbance $w_t \in \mathbb{W}$, measurable function $f : \mathbb{X} \times \mathbb{U} \to \mathbb{X}$, and matrices B_w and C of appropriate sizes.

To handle nonlinear systems of the form (1) we perform a piecewise-affine (PWA) approximation that yields a system described by

$$\begin{cases} x_{t+1} = A_i x_t + B_i u_t + a_i + B_{w,i} w_t + \kappa_t \text{ for } x_t \in P_i \\ y_t = Cx_t, \end{cases} \tag{2}$$

with P_i a partition of $\mathbb{X}$ and $\kappa_t \in \mathcal{K}_i$ the error introduced by performing the PWA approximation. For ease of notation, we denote the state-dependent error κ_{x_t} as κ_t. Furthermore, $A_i, B_i, B_{w,i}$ and a_i are matrices of appropriate sizes. Details of temporal logic control for nonlinear stochastic systems via piecewise-affine abstractions can be found in [40]. Besides nonlinear systems, we also consider the special case of linear time-invariant (LTI) systems:

$$\begin{cases} x_{t+1} = Ax_t + Bu_t + B_w w_t \\ y_t = Cx_t, \end{cases} \tag{3}$$

with A and B matrices of appropriate sizes.

Remark 1. *This first release of* SySCoRe *assumes the disturbance w_t has unbounded Gaussian distribution $w_t \sim \mathcal{N}(0, I)$. The implementation for other classes of distributions is under way and will be included in the future release of the tool. Note that the assumption of standard Gaussian distribution with zero mean and identity covariance matrix is without loss of generality since any system* (1)-(3) *with disturbance $w \sim \mathcal{N}(\mu, \Sigma)$ can be rewritten to a system in the same class with an additional affine term [5].*

To specify the model, that is a nonlinear system (1), a PWA system (2) or an LTI system (3), we have developed the classes `NonLinModel`, `PWAmodel`, and `LinModel`, respectively. The state space, input space, and the sets needed for defining the specification should be defined in these class descriptions.

Running example. Consider a two-dimensional (2D) case study of parking a car with dynamics of the form (3) *with $A = 0.9I_2$, $B = 0.7I_2$, and $B_w = C = I_2$. Furthermore, we have state space $\mathbb{X} = [-10, 10]^2$, input space $\mathbb{U} = [-1, 1]^2$, and disturbance $w \sim \mathcal{N}(0, I_2)$. After specifying matrices A, B, C, B_w, and setting the values for the disturbance w with mean* `mu` *and covariance matrix* `sigma` *equal to zero and identity respectively, we can initialize a model in* SySCoRe *as follows:*

```
% Set up an LTI model
sysLTI = LinModel(A,B,C,[],Bw,mu,sigma);
```

The state and input spaces are defined using `Polyhedron` *from the* `multi-parametric toolbox (MPT3)` *[16] as follows.*

```
% Define bounded state space
sysLTI.X = Polyhedron(combvec([-10,10],[-10,10])');
% Define bounded input space
sysLTI.U = Polyhedron(combvec([-1,1],[-1,1])');
```

Specifications. In SySCoRe, we consider formal specification written using co-safe linear temporal logic (scLTL) [9, 20], which consists of atomic proposition (AP) $AP = \{p_1, p_2, \dots, p_N\}$ that are either true or false. To connect the system and the specification, we label the output space of the system, such that we can relate the trajectories of the system $\boldsymbol{y} = y_0, y_1, y_2, \dots$ to the atomic propositions of the specification ϕ.

Running example cont'd. We consider reach-avoid specification ϕ_{park} with region to reach P_1 and avoid region P_2. First, we define the regions

```
% Specify regions for the specification
P1 = Polyhedron([4, -4; 4, 0; 10, 0; 10 -4]);
P2 = Polyhedron([4, 0; 4, 4; 10, 4; 10 0]);
```

and add them to the system object:

```
% Regions that get specific atomic propositions
sysLTI.regions = [P1;P2];
%  Propositions corresponding to the regions
sysLTI.AP = {'p1', 'p2'};
```

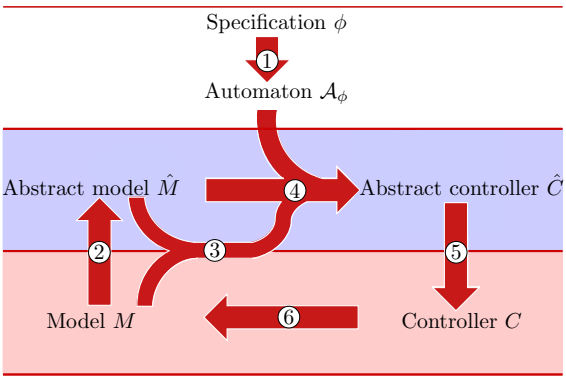


Figure 1: Steps in abstraction-based temporal logic control with 3 main layers: Continuous-state (red), finite-state (blue), and specification (white). The numbers correspond to the following steps: (1) translating the specification to an automaton, (2) (reduced) finite-state abstraction, (3) similarity quantification, (4) synthesizing a controller, (5) control refinement, and (6) deployment.

Implicitly, this means that states inside regions `P1` *and* `P2` *are labeled using the corresponding atomic propositions* `'p1'` *and* `'p2'`*, respectively. Now, we can write the scLTL specification*

$$\phi_{park} = \neg p_2 \, \mathsf{U} \, p_1, \tag{4}$$

using the syntax from [12] as follows

```
% Define the scLTL specification
formula = '(!p2 U p1)';
```

Denote the system M under the controller C by $M \times C$ as in [37]. The goal is to synthesize a controller C, such that the controlled system satisfies an scLTL specification ϕ, denoted as $M \times C \models \phi$. Since we consider stochastic systems, we compute the *satisfaction probability* denoted as $\mathbb{P}(M \times C \models \phi)$. This goal is formulated mathematically next.

Problem statement. Given model M, scLTL specification ϕ, and probability threshold $\rho \in (0, 1)$, design controller C such that

$$\mathbb{P}(M \times C \models \phi) \geq \rho. \tag{5}$$

`SySCoRe` automatically synthesizes a controller by maximizing the right-hand side of (5) on a simplified abstract model and makes the computations robust with respect to the abstraction errors. It provides a robust lower bound on the satisfaction probability, which can be used by the user to compare with probability threshold ρ.

2.2 Stochastic coupling relations for control synthesis

To solve the above problem, we use an abstraction-based approach and the *dynamic programming mappings* from [13], which allows us to consider infinite-horizon properties. More specifically, the abstraction-based temporal logic control implemented in `SySCoRe` has six main steps, namely (1) translating the specification to an automaton, (2) constructing a (reduced) finite-state abstraction, (3) quantifying the similarity, (4) synthesizing a controller, (5) control refinement, and (6) deployment.

Table 1: Main functions of `SySCoRe` for steps (1)-(6), with optional steps (2a) and (2b).

Step	Function
(1) Translate the specification	`TranslateSpec`
(2) Finite-state abstraction	`FSabstraction`
(2a) Piecewise-affine approx.	`PWAapproximation`
(2b) Model-order reduction	`ModelReduction`
(3) Similarity quantification	`QuantifySim`
(4) Synthesize a controller	`SynthesizeRobustController`
(5) Control refinement	`RefineController`
(6) Deployment	`ImplementController`

As visualized in Figure 1, we start from a temporal logic specification that expresses the desired behavior of the controlled system and translate it to an automaton (see top layer). A finite abstract model $\hat{M}$ of the system is also constructed (step 2). For this abstract model $\hat{M}$, its bounded deviation from the original model can be quantified using simulation relations (step 3) [14, 39]. Computing these bounds is based on an efficient invariant set computation formulated as an optimization problem constrained by a set of parameterized matrix inequalities [39]. Based on the automaton, an abstract controller over the abstract model can be synthesized. In step 4, we synthesize an abstract controller $\hat{C}$ and compute the robust satisfaction probability. The robust satisfaction probability takes the deviation bounds computed in step 3 into account and gives a lower bound on the actual satisfaction probability. To compute the robust satisfaction probability and to synthesize an abstract controller $\hat{C}$, `SySCoRe` solves a reachability problem over the abstract system combined with the automaton corresponding to the specification. This reachability problem is then solved as a dynamic programming problem. It is shown in [13] that leveraging the deviation bounds from step 3, the controller for the abstract model can be refined to the original continuous-state model while preserving the guarantees. To construct this controller C, `SySCoRe` refines the abstract controller in step 5. The resulting controller C is a policy that can be represented with finite memory. Finally, `SySCoRe` deploys the controller on the model (step 6). It is important to note that the abstraction step (step 2 in Figure 1) can additionally contain model-order reduction or piecewise-affine approximation, which shows the comprehensiveness of `SySCoRe` enabled by establishing coupled simulation relations.

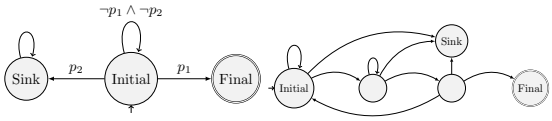
The next section gives a complete overview of the toolbox and specifies how each of the steps from Figure 1 is implemented.

3 TOOLBOX OVERVIEW

After setting-up the problem by specifying the system using the classes `NonLinModel`, `PWAmodel` or `LinModel`, and the specification as an scLTL formula, we continue with the steps illustrated in Figure 1. Each step corresponds to a specific function as in Table 1. Note that the abstraction step may have multiple (formal) approximation stages depending on the type of the model or its dimension.

3.1 Translating the specification

For control synthesis, the scLTL specification is written as a deterministic finite-state automaton (DFA) [9]. Examples of such DFAs



(a) DFA corresponding to specification ϕ_{park} in (4). (b) Atypical DFA

Figure 2: Acyclic DFA in (a) versus cyclic DFA in (b).

are given in Figure 2. We use the tool LTL2BA to translate an scLTL specification, which constructs a non-deterministic Büchi automaton for a general LTL specification [12]. Additionally, we check whether the given formula is written using scLTL (instead of full LTL) and then (if possible) rewrite the non-deterministic Büchi automaton to a DFA. This step is based on powerset conversion [31] that is used to convert a nondeterministic finite-state automaton to a DFA. The complete translation from an scLTL specification to a DFA is implemented in the function TranslateSpec.

```
% Translate an scLTL formula to a DFA
DFA = TranslateSpec(formula, AP);
```

The input formula is given using the syntax of LTL2BA in [12].

Running example cont'd. For the 2D car park, we consider the reach-avoid specification ϕ_{park} in (4), which we translate to a DFA using TranslateSpec *with* AP *and* formula *given in code lines 13 and 15.*

```
16 % Translate the spec to a DFA
17 DFA = TranslateSpec(formula, sysLTI.AP);
```

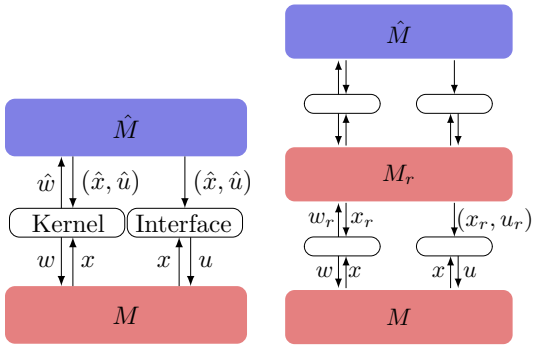
Besides reach-avoid specifications it is also possible to describe many other types of specifications, such as more complex reach-avoid specification, e.g. $\phi_{PD} = \Diamond(p_1 \wedge (\neg p_2 \,\mathsf{U}\, p_3))$, or time-bounded and unbounded safety specifications, e.g. $\phi_{BAS} = \bigwedge_{i=0}^{5} \bigcirc^i p_1$ and $\phi_{vdPol} = p_1 \,\mathsf{U}\, p_2$. These specifications are written in SySCoRe as

```
formula_PD = 'F(p1 & (!p2 U p3))';                    (6a)
formula_BAS = '(p1 & X p1 & X X p1 & X X X p1 ...     (6b)
& X X X X p1 & X X X X X p1)';
formula_vdPol = '(p1 U p2)';                          (6c)
```

Note that it is also possible to directly pass a DFA as an input to SySCoRe instead of giving the specification as an scLTL formula. SySCoRe is able to natively handle both acyclic and cyclic DFAs (see Figure 2), in contrast to many other tools [11, 23, 32, 36] that do not natively support DFAs but often rely on external tools such as PRISM [21] to compute the controller.

3.2 Abstraction

SySCoRe includes two possible abstraction methods, namely finite-state abstraction for continuous-state systems in (1)-(3) and model-order reduction for continuous-state LTI systems (3). However, in order to create a finite-state abstraction of a nonlinear system (1) we require an additional approximation step before constructing a piecewise-affine finite-state abstraction. Note that the piecewise affine approximation itself is considered as an integral part of the finite-state abstraction method.



(a) Coupling between models M and its finite-state abstraction $\hat{M}$ through their inputs and disturbances via an interface function and a coupling kernel. (b) Coupling between continuous-state models M and reduced-order model M_r, and between M_r and its finite-state abstraction $\hat{M}$.

Figure 3: Coupling between different models. Red and blue boxes correspond to respectively continuous-state and finite-state models. In (a) only a finite-state abstraction is performed, while in (b) both model-order reduction and a finite-state abstraction are shown.

Piecewise affine approximation. To approximate a nonlinear system (1) by a PWA system (2), we partition the state space and use a standard first-order Taylor expansion to approximate the nonlinear dynamics in each partition by affine dynamics. Additionally, we compute the error introduced by this approximation. In SySCoRe, this is performed by the function PWAapproximation.

```
% Perform piecewise-affine approximation
sysPWA = PWAapproximation(sysNonLin, Np);
```

Here, the nonlinear system (1) is given by sysNonLin and the number of partitions in each direction is given by Np. The result is a PWA system (2) sysPWA.

Interface function. SySCoRe can construct a reduced-order abstract model M_r and a finite-state abstract model $\hat{M}$ of the original model M. Let us denote the control inputs of these models respectively by u_r and $\hat{u}$. The abstract control inputs u_r and $\hat{u}$ need to be refined to a control input u for M as illustrated in Figure 3. The input refinement is performed by one or multiple interface functions, namely

$$u_{r,t} = \hat{u}_t \qquad \textit{(default)} \qquad (7a)$$

$$u_{r,t} = \hat{u}_t + K(x_{r,t} - \hat{x}_t) \qquad \textit{(option 1)} \qquad (7b)$$

$$u_t = u_{r,t} + Qx_{r,t} + K_{MOR}(x_t - Px_{r,t}). \qquad \textit{(option 1, MOR)} \qquad (7c)$$

To refine the input $\hat{u}$ of a finite-state model to the input u_r of a continuous-state reduced-order model, we implemented two different interface functions in the format of (7a) and (7b). For many cases the default interface function (7a) should work fine, however, the option (7b) gives more influence on the refined controller by including a feedback term. When the interface function (7b) is used, we have to take this into account when constructing the finite-state abstraction to avoid the input bounds being violated, therefore, the interface function must be chosen before constructing the finite-state abstraction. We further use the interface function (7c) to refine

the input u_r of a reduced-order model to the input u of the full-order model. It should be noted that if only a finite-state abstraction is performed without using model-order reduction (MOR), we have $P = I, Q = 0$ and $x_t = x_{r,t}$, hence we obtain interface functions (7a) and (7b) with $u_t = u_{r,t}$ and $x_t = x_{r,t}$.

Running example cont'd. It is required to select an interface function for the input refinement before starting with the temporal logic control steps. For this running example we only use the default interface function (7a) *without model-order reduction, that is* $u_t = \hat{u}_t$. *However, if desired, the user can select the option* (7b) *by setting* `int_f = 1` *and passing this to the functions.*

In the remainder of this section, we discuss how to obtain the reduced-order and finite-state abstract models.

Model-order reduction. It is essential to include model-order reduction for high-dimensional models. For LTI systems (3) this yields a reduced-order model M_r of the form

$$M_r : \begin{cases} x_{r,t+1} = A_r x_{r,t} + B_r u_t + B_{rw} w_{r,t} \\ y_{r,t} = C_r x_{r,t}, \end{cases} \tag{8}$$

with $x_r \in \mathbb{X}_r, u \in \mathbb{U}, y \in \mathbb{Y}, w_r \in \mathbb{W}$, and matrices A_r, B_r, B_{rw} and C_r of appropriate sizes.

In SySCoRe, the function `ModelReduction` constructs a reduced-order model `sysLTIr` of dimension `dimr` based on the original model `sysLTI` by using *balanced truncations* on a closed loop system with a feedback matrix F. This feedback matrix is computed by solving *discrete-time algebraic Riccati equations* that can be tuned using constant `f` [29]. The syntax of `ModelReduction` is

```
% Construct reduced-order model
[sysLTIr, F] = ModelReduction(sysLTI, dimr, f)
```

We couple the inputs u, u_r from M (3) and M_r (8) using the interface function (7c) as illustrated in Figure 3b. This is based on the theoretical results presented in [14, 39]. To compute matrices P and Q for the interface function, we have the function `ComputeProjection` that adds the matrices automatically to the object `sysLTIr`.

```
% Compute matrices P and Q
sysLTIr = ComputeProjection(sysLTI, sysLTIr);
```

Finite-state abstraction. We grid the state space to construct a finite-state abstraction $\hat{M}$ of the continuous-state models (2), (3) or (8). More specifically, we compute the abstract state space $\hat{\mathbb{X}}$ as the set consisting of the centers of the grid cells. Next, the dynamics of the abstract model is defined by using the operator $\Pi : \mathbb{X} \to \hat{\mathbb{X}}$ that maps states x to the center of the grid cell it is in. Details on how to construct a finite-state abstraction of a nonlinear system or an LTI system can be found in [40, Section III], and [13, Section IV] or [39, Section IV] respectively.

In SySCoRe, the construction of the finite-state abstraction is implemented in the functions `GridInputSpace` and `FSabstraction`. The function `GridInputSpace` constructs the abstract input space `uhat` by selecting a finite number of inputs from the input space `sysLTI.U`.

```
% Construct abstract input space
[uhat, InputSpace] = GridInputSpace(lu, sys.U, ...
     options);
```

Here, `lu` is the number of abstract inputs in each direction and `options` are used to select an interface function from (7). If interface function (7b) or (7c) is chosen, `GridInputSpace` also divides the continuous input space into a part for actuation and for feedback, and returns these spaces as output `InputSpace`. This is done to make sure that the input bounds $u \in \mathbb{U}$ of the original model are satisfied. Next, we use `FSabstraction` to compute a *probability matrix* that contains the transition probabilities between states for all possible inputs in `uhat`.

```
% Construct abstract model
sysAbs = FSabstraction(sys, uhat, l, tol, DFA, ...
     options);
```

Here, `sys` is the continuous-state system, `uhat` is the abstract input space $\hat{\mathbb{U}}$, `l` is the number of grid cells in each direction and `tol` is the tolerance for truncating to zero. This means that if a probability is smaller than the value set by `tol`, then we set it to zero to increase sparsity and hence decrease computation time. Via efficient tensor computations, we split the computation of the probability matrix into two parts: one for the deterministic part of the transitions computed as a sparse matrix, and one for the stochastic part of the transitions. This reduces the required memory allocation and computation time drastically. For development purposes `options` can be used to select whether or not to use this efficient *tensor* computation. The complete probability matrix can then be obtained by using a tensor multiplication, however, we do not store the complete probability matrix and compute it when necessary in order to save memory.

Running example cont'd. To construct a finite-state abstraction of the car park model `sysLTI` *(defined in code lines 1-13), we compute the abstract input space* `uhat`*:*

```
% Construct abstract input space uhat
lu = 3;  % number of abstract inputs
uhat = GridInputSpace(lu, sysLTI.U);
```

and construct the abstract model `sysAbs` *using the* `DFA` *constructed in code line 17 as follows:*

```
% Construct finite-state abstraction
l = [200, 200];  % number of grid cells
tol = 10^-6;
sysAbs = FSabstraction(sysLTI, uhat, l, tol, ...
       DFA, 'TensorComputation', true);
```

3.3 Similarity quantification

To quantify the similarity between the model and its abstraction (either reduced order or finite state), we compute ϵ and δ such that they satisfy the (ϵ, δ)-stochastic simulation relation as defined in [39, Definition 4]. Here, ϵ and δ represent bounds on the output and probability deviations, respectively. This simulation relation allows us to consider scLTL specifications with unbounded time properties [13].

When using model-order reduction, we construct two simulation relations, one relation $\mathcal{R}_{MOR}$ between the original model M (3) and reduced-order model M_r (8), and one relation $\mathcal{R}$ between M_r and the finite-state model $\hat{M}$. The simulation relations are of the form

$$\mathcal{R}_{MOR} := \left\{(x_r, x) \in \mathbb{X}_r \times \mathbb{X} \mid ||x - Px_r||_{D_r} \leq \epsilon_r\right\} \tag{9a}$$

$$\mathcal{R} := \left\{(\hat{x}, x_r) \in \hat{\mathbb{X}} \times \mathbb{X}_r \mid ||x_r - \hat{x}||_D \leq \epsilon\right\}, \tag{9b}$$

with $||x||_D = \sqrt{x^\top Dx}$ the weighted two-norm, where $D = D^\top \geq 0$ is positive semi-definite. Following [39], these simulation relations can be combined into one total simulation relation between M and $\hat{M}$. Following [13, Section IV.A], we can now compute the initial state of the reduced-order model as the state $x_{r,0}$ that minimizes $||x_0 - Px_{r,0}||_{D_r}$, that is $x_{r,0} := (P^\top D_r P)^{-1} P^\top D_r x_0$.

The computation of the simulation relation relies heavily on the coupling of the inputs $u, \hat{u}$ and disturbances $w, \hat{w}$ of the two models. The inputs are coupled through an interface function and the disturbances via a coupling kernel. This is illustrated in Figure 3 and is based on the method developed in [39]. More specifically, the underlying computation is based on finding an invariant set for the error dynamics $x_{r,t+1} - \hat{x}_{t+1}$. To this end, an optimization problem constrained by parameterized linear matrix inequalities is used to find a value for δ that corresponds with the given value of ϵ [39]. To solve this optimization problem, we use the `multi-parametric toolbox (MPT3)` [16] with `YALMIP` [26] and with either solver `SeDuMi` [22] or `MOSEK` [7].

In `SySCoRe`, similarity quantification is implemented in the function `QuantifySim`.

```
% Quantify similarity
[simRel, interface] = QuantifySim(sys, sysAbs, ...
    epsilon, options)
```

This function quantifies the similarity between the models `sys` and `sysAbs`, with `sysAbs` either a reduced-order or a finite-state approximation of `sys`, hence in terms of behavior `sysAbs` $\preceq$ `sys`. `QuantifySim` yields a simulation relation `simRel` of the form (9) that is stored in the object `SimRel`. This object includes a method to check whether two states belong to the simulation relation and a method to combine the two simulation relations from (9) if necessary. Besides that, the function `QuantifySim` also returns the feedback-matrix of the `interface` function, when interface (7b) or (7c) is chosen through the `options`.

Running example cont'd. Next, we quantify the similarity between the model of the car stored in `sysLTI` *and its finite-state abstraction* `sysAbs` *constructed in code line 24 by choosing a suitable value for* ϵ *and using the function* `QuantifySim`.

```
% Choose a value for epsilon
epsilon = 1.005;
% Quantify similarity
simRel = QuantifySim(sysLTI, sysAbs, epsilon);
```

Piecewise affine systems. The function `QuantifySim` can handle both PWA (2) and LTI models (3). However, for PWA systems the probability deviation is a PWA function $\boldsymbol{\delta}(\hat{x})$ that depends on the partition of the abstract state [40].

3.4 Synthesizing a robust controller

We synthesize a robust (finite-state) controller based on the dynamic programming approach described in [13], which is robust in the sense that it takes the deviation bounds ϵ and δ into account to compute a lower bound on the actual satisfaction probability. Furthermore, it is proven in [13, Theorem 4] that the resulting control policy synthesized for the abstract model can always be refined to a control policy for the actual model.

More specifically, we implicitly construct a product composition of the finite-state model $\hat{M}$ with the DFA such that computing the *satisfaction probability* becomes a reachability problem over this product composition. This can in turn be solved using *dynamic programming* by associating a robust dynamics programming operator that allows for an iterative computation of the lower bound on the satisfaction probability. Denote the state of the DFA by q, then the probability that a trajectory starting at $(\hat{x}, q)$ reaches the set of accepting states by applying policy $\boldsymbol{\mu}$ within horizon $[1, 2, \dots N]$ is denoted as $V_N^{\boldsymbol{\mu}}(\hat{x}, q)$. This is equivalent to the probability of satisfying the specification ϕ over this time horizon. The probability V is computed iteratively by defining the operator

$$\mathbf{T}^{\hat{u}}(V)(\hat{x}, q) := \boldsymbol{L}\left(\mathbb{E}_{\hat{u}}\left(\min_{q^+ \in Q^+} \max\{1_{Q_f}(q^+), V(\hat{x}^+, q^+)\}\right) - \delta\right), \tag{10}$$

where $\hat{x}^+$and q^+ are resp. the next state of the abstract model and of the DFA, $\mathbb{E}$ is expectation with respect to the probabilistic transitions in the abstract model, $1_{Q_f}(q)$ is an indicator function that is equal to 1 if q is inside the set of accepting states Q_f of the DFA and is 0 otherwise, $\boldsymbol{L} : \mathbb{R} \to [0, 1]$ is a truncation function, and with

$$Q^+(q, \hat{y}^+) := \left\{\tau_{\mathcal{A}_\phi}(q, L(y^+)) \mid ||y^+ - \hat{y}^+|| \leq \epsilon\right\}, \tag{11}$$

where $\tau_{\mathcal{A}_\phi}$ is the transition function of the DFA and $L(y^+)$ is the label of the next output. This operator is robust in the sense that the probability gets reduced by δ at every time step and the worst case transition of the DFA is considered with respect to ϵ. The derivation of this operator for Markov decision processes can be found in [13].

Synthesis of an abstract control strategy `pol` and the computation of the robust *satisfaction probability* `satProb` is performed by the function `SynthesizeRobustController` and it is based on the abstract model `sysAbs`, the specification as a `DFA` and the simulation relation `simRel`.

```
% Compute satisfaction probabilty and policy
[satProb, pol] = SynthesizeRobustController(...
sysAbs, DFA, simRel, thold, options)
```

We include the possibility to set the threshold `thold` that stops the value iteration when the difference between two iterations is smaller than this threshold. The default value is set to $1 \cdot 10^{-12}$. This choice is justified by the fact that the operator in (10) is contractive and will always converge monotonically to a fixed-point. Additionally, we include the `options` to compute the value function only for the initial DFA state and to compute an upper bound on the satisfaction probability. Internally, the dynamic programming algorithm computes the product between large-scale matrices (one of which is the probability matrix as mentioned in Section 3.2 on finite-state abstractions). By performing these computations using a tensor product [28], we gain superior computational efficiency.

The resulting control policy `pol` is a mapping $\mu : \hat{\mathbb{X}} \times Q \to \hat{\mathbb{U}}$ from the pair of abstract and DFA states to the abstract input space. The abstract controller can now be written as $\hat{C} : \hat{u} = \mu(\hat{x}, q)$.

Running example cont'd. After specifying the desired threshold for convergence `thold`*, we synthesize a robust control policy* `pol` *based*

on the finite-state abstract model `sysAbs`*, the specification as a* `DFA` *and the simulation relation* `simRel` *constructed in code line 28. In this case, we are only interested in the satisfaction probability* `satProb` *of the initial DFA state, hence we set the* `options` *to* `true`.

```
% Specify threshold for convergence error
thold = 1e-6;
% Synthesize an abstract robust controller
[satProb, pol] = SynthesizeRobustController(...
sysAbs, DFA, simRel, thold, true);
```

The robust satisfaction probability is computed for all $x_0 \in \mathbb{X}$*. For initial states* $x_0 = [-4,-5]^\top$*,* $x_0 = [-8,2]^\top$*, and* $x_0 = [4,8]^\top$*, it equals respectively* 0.60, 0.52*, and* 0.42*.*

3.5 Control refinement

To refine an abstract finite-state controller to a controller C that can be implemented on the original continuous-state system (see step 5 in Figure 1) we use one or multiple interface functions from (7a) as illustrated in Figure 3. In `SySCoRe`, control refinement is included in the class `RefineController`, where it is possible to select an interface function using the `options`.

```
% Refine abstract controller
Controller = RefineController(satProb, pol, ...
    sysAbs, simRel, sys, DFA, options);
```

This class not only refines the finite-state input to the actual input, but also determines the state of the finite-state model based on the state of the original model.

Running example cont'd. To construct a controller C *that can be implemented on the original model* M *based on the abstract control policy* `pol` *computed in code line 32, we use the following.*

```
% Refine abstract controller
Controller = RefineController(satProb, pol, ...
      sysAbs, simRel, sysLTI, DFA);
```

3.6 Deployment

The final step is to deploy the controller on the model and perform simulations using `ImplementController`.

```
% Implement the controller on the model
xsim = ImplementController(x0, N, Controller, ...
    option);
```

Here, `N` is the desired time horizon for the simulation and `option` is used to supply the number of trajectories and/or additional model-order reduction inputs.

Running example cont'd. To simulate the controlled system with the `Controller` *constructed in code line 34, we use* `ImplementController` *to obtain the state trajectory starting at* x_0*. Trajectories of the controlled system with three initial states are illustrated in Figure 4.*

```
x0 = [-4; -5]; % initial state
N = 40; % time horizon
% Simulate controlled system
xsim = ImplementController(x0, N, Controller);
```

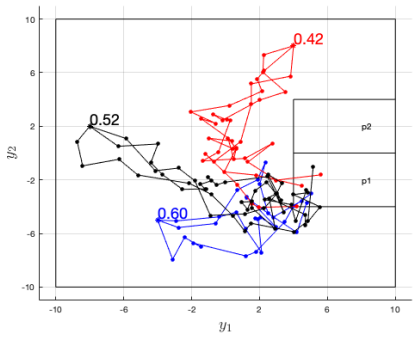


Figure 4: Trajectories for the running example. Three trajectories are obtained for each initial state: $x_0 = [-4,-5]^\top$ **(blue),** $x_0 = [-8,2]^\top$ **(black), and** $x_0 = [4,8]^\top$ **(red). The corresponding robust satisfaction probability is given at the inital state.**

4 BENCHMARKS

To show the capabilities of `SySCoRe`, we included multiple benchmarks, of which some are discussed here. The package delivery has a complex specification with a cyclic DFA, the building automation system includes model-order reduction and the Van der Pol oscillator is nonlinear. We evaluate the run time and memory usage of the benchmarks, and compare `SySCoRe` to some existing tools.

4.1 Package delivery

With the package delivery benchmark [4], we show the capability of `SySCoRe` to handle complex scLTL specifications beyond basic reach-avoid scenarios, i.e., cyclic DFAs. Consider an agent traversing in a 2D space, whose dynamics can be described by an LTI system (3) with $A := 0.9I_2$, $B := I_2$, $B_w := \sqrt{0.2}I_2$, $C := I_2$, and disturbance $w_k \sim \mathcal{N}(0, I_2)$. We initialize the system using `LinModel`.

Define the state space $\mathbb{X} = [-6,6]^2$, input space $\mathbb{U} = [-1,1]$, output space $\mathbb{Y} = \mathbb{X}$, and regions p_1, p_2 and p_3 as follows: $p_1 := [5,6] \times [-1,1]$, $p_2 := [0,1] \times [-5,1]$ and $p_3 := [-4,-2] \times [-4,-3]$. The agent can pick up a package at p_1 and must deliver it to p_3. If the agent visits p_2 while carrying a package, it loses the package and has to pick up a new package at p_1. This corresponds to the scLTL specification $\Diamond(p_1 \wedge (\neg p_2 \,\mathcal{U}\, p_3))$ implemented as in (6a). We generate the corresponding DFA using `TranslateSpec`.

Next, we construct a finite-state abstraction using `GridInputSpace` and `FSabstraction`. We choose state abstraction $l = [400, 400]$, which allows us to generate a simulation relation using `QuantifySim` with an epsilon of just 0.075. Note that the partition size l is a tuning parameter which is determined empirically. We synthesize a robust controller for the discrete abstraction using

```
[satProb, pol] = SynthesizeRobustController( ...
    sysAbs, DFA, rel, thold, false);
```

Since the resulting control policy is conditional on both the current system state and the DFA state, we set the 5th argument to `false`. By doing so, we synthesize a controller for all DFA states instead of only the initial one. The obtained robust satisfaction probability `satProb` over different initial states x_0 is displayed

in Figure 5a and has a peak satisfaction probability is 0.663. We included function `plotSatProb` to plot the satisfaction probability.

Finally, we refine the controller using `RefineController`. To demonstrate the performance of the obtained controller, we simulate the controlled system using `ImplementController` for $N = 60$ time steps and an initial state of $x_0 = [-5, -5]^T$. Note that N is an empirical parameter and should be set high enough for the DFA to terminate. As expected, the agent moves to region p_1 to pick up a package, and delivers it to p_3 whilst avoiding p_2. To plot trajectories we included the function `plotTrajectories`.

4.2 Van der Pol oscillator

In this benchmark, we show how SySCoRe can be applied to nonlinear stochastic systems. For this, consider the discrete-time dynamics of the Van der Pol oscillator [4], given by

$$\begin{aligned} x_{1,t+1} &= x_{1,t} + x_{2,t}\tau + w_{1,t} \\ x_{2,t+1} &= x_{2,t} + (-x_{1,t} + (1 - x_{1,t}^2)x_{2,t})\tau + u_t + w_{2,t}, \end{aligned} \tag{12}$$

where the sampling time τ is set to $0.1s$, $w_t \sim \mathcal{N}(0, 0.2I_2)$, and $y_t = x_t$. We define the state space $\mathbb{X} = [-3, 3]^2$, input space $\mathbb{U} = [-1, 1]$, and output space $\mathbb{Y} = \mathbb{X}$. For the Van der Pol oscillator, we are looking at an unbounded safety specification (cf. (6c)), where the objective is to synthesize a controller such that the system remains in the region $p_1 := \mathbb{X}$ until reaching region $p_2 := [-1.4, -0.7] \times [-2.9, -2]$, corresponding to the scLTL specification $p_1 \cup p_2$. First, we construct a DFA for the formula (6c) using `TranslateSpec`.

Since the dynamics of the oscillator (`sysNonLin`) in (12) are nonlinear, the abstraction process is split into two parts as outlined in Section 3.2. First, we construct a PWA approximation as follows.

```
% Number of grid points in each direction
N = [41 41];
% Perform PWA approximation
sysPWA = PWAapproximation(sysNonLin, N);
```

In the second part of the abstraction step, a finite-state abstraction (`sysAbs`) of the PWA approximation (`sysPWA`) is constructed using `GridInputSpace` and `FSAbstraction` with `l=[600,600]` grid cells. To generate a simulation relation between this abstraction and the original model, we set $\epsilon = 0.1$ and compute a suitable weighting matrix D for the simulation relation on $(\hat{x}, x)$, as described in Section 3.3. To reduce computation time, we only use a finite number of states to compute this weighting matrix. Details on why we need this global weighting matrix can be found in [40].

```
% Compute weighting matrix D for the simulation ...
     relation based on the following states
States = [1/8*x1l, 6/10*x2u; 5/7*x1u, 5/17*x2u; ...
     2/13*x1u, 5/9*x2l; 3/4*x1l, 1/7*x2l; 0, 0]';
[D, ~] = ComputeD(epsilon, sysPWA, sysAbs, ...
     'interface', int_f, 'states', States);
% Quantify similarity
[rel, sysPWA] = QuantifySim(sysPWA, sysAbs, ...
     epsilon, 'interface', int_f, 'weighting', D);
```

Note that `QuantifySim` returns `sysPWA` instead of the usual `interface`, because each piecewise-affine system gets its own interface function and we store this directly in `sysPWA`.

Next, we use `SynthesizeRobustController` to synthesize a robust controller for `sysAbs` and show the satisfaction probability (displayed in Figure 5b) using `plotSatProb`. Finally, we refine the controller as follows:

```
Controller = RefineController(satProb, pol, ...
     sysAbs, rel, sysPWA, DFA, int_f);
```

As before, `ImplementController` is used to simulate the system.

4.3 Building automation system

In the last benchmark, we address a large-scale system showcasing the model-order reduction capabilities of SySCoRe. We consider a 7D *affine* stochastic system of a building automation system, regulating the temperature in two zones influenced by a 6D disturbance. A detailed description including the system dynamics can be found in [3, 10]. The goal is to synthesize a controller maintaining the temperature in zone one at $20°C$ with a maximum permissible deviation of $\pm 0.5°C$ for 6 consecutive time steps. We translate the specification (6b) to a DFA using `TranslateSpec`.

The dynamics of this building automation system are not of the form (3), since it is influenced by a Gaussian disturbance with mean $\mu \neq 0$ and variance $\Sigma \neq I$. Furthermore, it is not an LTI system, but affine, which cannot be handled by our current implementation of model-order reduction. To deal with the disturbance, we first transform the system to a system with Gaussian disturbance $w \sim \mathcal{N}(0, I)$ using the following:

```
% Transform the model
[sysLTI, a] = NormalizeDisturbance(sysLTI,a);
```

To deal with the affine dynamics, we perform a steady-state shift and simulate the steady-state system that has LTI dynamics. After performing the control synthesis steps, we compensate for this steady-state shift again to obtain the dynamics of the actual system.

Now, we can start with the synthesis steps. First, we reduce the 7D model to a 2D reduced-order model (see Eq. (8)) using function `ModelReduction` with `f = 0.098` and dimr=2.

```
% Perform model-order reduction
[sysLTIr, ~] = ModelReduction(sysLTI, dimr, f);
```

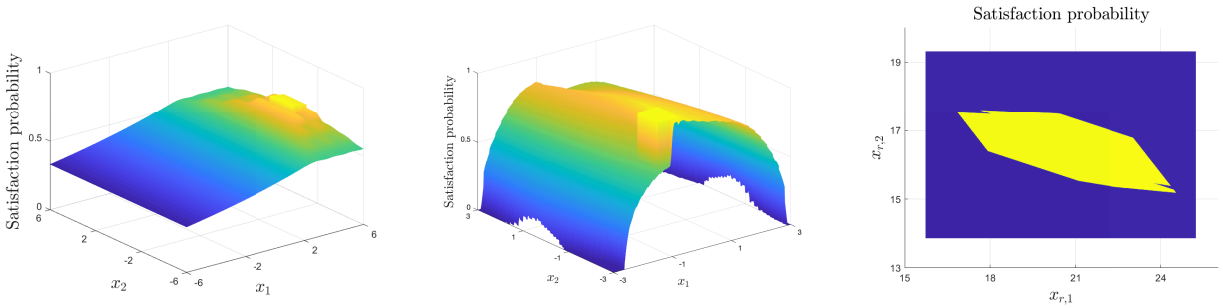
As mentioned in Section 3.2, we use an interface function of the form (7c), which is selected using `int_f = 1` and compute the matrices P and Q using `ComputeProjection`. Next, we define the state and input spaces, and the output regions and APs for the reduced-order model as before.

To construct the finite-state abstraction of the reduced-order model, we first grid the input space with `lu = 3`.

```
% Construct abstract input space
[uhat,sysLTIr.U] = GridInputSpace(lu, sysLTIr.U, ...
     'interface', int_f, 0.6, 0.175);
```

Here, we have chosen to use 60% of the input space for actuation and 17.5% for feedback. This leaves 22.5% for the $Qx_{r,t}$ part of the interface function, which is currently not guaranteed to be satisfied.

Before constructing a finite-state abstraction of the reduced-order model, we reduce the state space to increase the computational speed. This step is currently only available for invariance



(a) Robust satisfaction probability of the package delivery benchmark.

(b) Robust satisfaction probability of the Van der Pol oscillator benchmark.

(c) Robust satisfaction probability of the reduced-order model of the building automation system benchmark. Yellow and blue correspond to a probability of 0.9035 and 0 resp.

Figure 5: Robust satisfaction probability of the initial DFA state as a function of the initial state for the different benchmarks. In (a) the package delivery benchmark, in (b) the van der Pol benchmark, and in (c) the building automation system.

Table 2: An overview of the different benchmarks and their total computation time in seconds (s) and memory usage in megabyte (MB). The details of the computation times for each step are reported in Table 3. *Dim.* and *Comp.* are abbreviations for *Dimension* and *Computation*, respectively. The size of the specification refers to the number of states of the DFA.

Benchmark	System		MOR	Specification			Comp. time (s)	Memory (MB)
	Dynamics	Dim.		Type	Time horizon	Size		
Running example	Linear	2	No	Reach-avoid	Unbounded	3	7.94	27.53
Package delivery	Linear	2	No	Reach-avoid	Unbounded	3	11.02	133.4
Van der Pol oscillator	Nonlinear	2	No	Safety, reachability	Unbounded	3	3191.6	178.83
Building automation	Linear	7	Yes	Safety	Bounded	8	122.05	5365.6

Table 3: Computation times for steps (1)-(6) in seconds and as percentage of the total runtime. Steps (1)-(6) correspond to (1) translating the specification, (2) finite-state abstraction, (3) similarity quantification, (4) synthesizing a controller, (5) control refinement, and (6) deployment. Step (5) is almost instantaneous (≈ 0.001 s), therefore, we take steps (5) and (6) together.

	Step (1)	Step (2)	Step (3)	Step (4)	Step (5) and (6)	Total
Running example	0.259s (3.26%)	1.316s (16.57%)	5.590s (70.38%)	0.507s (6.39%)	0.204s (2.57%)	7.944s (100%)
Package delivery	0.284s (2.58%)	1.657s (15.04%)	6.193s (56.2%)	1.708s (15.5%)	0.702s (6.37 %)	11.02s (100%)
Van der Pol oscillator	0.590s (0.02%)	1440.1s (45.1%)	1748.6s (54.8%)	2.854s (0.09%)	1.417s (0.04%)	3191.6s (100%)
Building automation	0.361s (0.30%)	4.80s (3.94%)	67.92s (55.7%)	37.33s (30.6%)	9.19s (7.53%)	122.05s (100%)

specifications and is performed by `ReduceX`, which performs a number of backwards iterations on the safety region P_1 to determine a good guess of the invariant set. This set is then used as the reduced state space. The construction of the finite-state abstraction of the reduced-order model is as before, except that we give the total number of grid cells as input `l`.

```
% Reduce the state space to speed up computations
[sysLTIr, ~] = ReduceX(sysLTIr, sysLTIr.U{2}, ...
    P1, 'invariance', 5);
% Construct finite-state abstraction
l = [3000*3000];  % Total number of grid cells
tol=10^-6;
sysAbs = FSabstraction(sysLTIr, uhat, l, tol, ...
    DFA, 'TensorComputation', true);
```

To relate the reduced-order finite-state model `sysAbs` to the original model `sysLTI`, we construct two simulation relations: relation `rel_1` with $\epsilon_1 = 0.2413$ between `sysLTI` and `sysLTIr`, and relation `rel_2` with $\epsilon_2 = 0.1087$ between `sysLTIr` and `sysAbs`.

```
% Compute MOR simulation relation
[rel_1, K, kernel] = QuantifySim(sysLTI, ...
    sysLTIr, epsilon_1, 'MOR', sysAbs);
% Compute finite-state simulation relation
[rel_2] = QuantifySim(sysLTIr, sysAbs, epsilon_2);
% Combine simulation relations
rel = CombineSimRel(rel_1, rel_2, sysLTIr, sysAbs);
```

For model-order reduction we have to explicitly define the coupling `kernel` matrix F, that is later used to compute the disturbance of the reduced-order model as $w_r = w + F(x - Px_r)$. For details see [39].

Synthesizing and refining the controller are done as before and the satisfaction probability of the reduced-order model is shown in Figure 5c (obtained through `plotSatProb`). We simulate the controlled system 6 times, making sure the output is shifted with respect to the steady-state solution. The resulting trajectories can be evaluated using `plotTrajectories`.

4.4 Performance evaluation

The performance of `SySCoRe` is evaluated on the benchmarks mentioned above. The details of the benchmarks and their total run time and memory usage are reported in Table 2. The computation times

Table 4: Results of the benchmarks for different tools. Here, *n.a.* means that a tool is *not applicable* and *n.s.* means that the current version of the tool does *not natively support* the computations on the benchmark, but that we do not see fundamental limitations hindering such an extension. To compare the tools we exclude the deployment of the controller (step (6)), since this step is not performed by the other tools.

(a) Package delivery benchmark.

Tool	Run time (s)
AMYTISS	n.s.
FAUST	n.s.
SReachTools	n.a.
StocHy	n.s.
SySCoRe	10.319

(b) Van der Pol benchmark.

Tool	Run time (s)
AMYTISS	n.s.
FAUST	n.a.
SReachTools	n.a.
StocHy	n.a.
SySCoRe	3190.2

(c) Building automation benchmark.

Tool	Run time (s)	Max. reach probability
AMYTISS	312.14	≈ 0.8
FAUST	n.s.	n.s.
SReachTools	4.59	≥ 0.99
StocHy	≥ 335.876	$\geq 0.8 \pm 0.23$
SySCoRe	112.86	≥ 0.9035

per step are reported in Table 3. The data has been obtained on a computer with a 2,3 GHz Quad-Core Intel Core i5 processor and 16 GB 2133 MHz memory by taking the average over 5 computations. Here, we observed a maximum 6% standard deviation.

Table 2 can be used to compare the different benchmarks with respect to the computations performed by SySCoRe. The main difference between the running example and the package delivery benchmark is the DFA. The DFA of the package delivery benchmark requires more memory, however, the increase in computation time is small. Due to the simple DFA of the running example, we only compute the satisfaction probability for the initial DFA state. This will not suffice for the package delivery benchmark, which is the reason that more computation time is spent on steps (4)-(6) compared to the running example (see Table 3). The computation time for the nonlinear benchmark is large, however, the memory usage remains reasonable. The increase in computation time is mainly due to the fine gridding. We can also see in Table 3 that the similarity quantification takes a considerable amount of time. This is because we perform this step for each partition separately (1600 times in this case). For higher-dimensional systems that require model-order reduction (building automation system benchmark), the computation time and memory usage increase substantially, mainly due to the the fact that the similarity quantification has to be performed multiple times. However, we also see from Table 3 a large increase in the computation time for the controller synthesis.

Table 3 shows that the similarity quantification of step (3) requires the most computation time, followed by the finite-state abstraction of step (2). The large computation time of the similarity quantification is due to solving an optimization problem constrained by parameterized matrix inequalities that could be non-convex. For most abstraction-based approaches in the literature, the main bottleneck is the finite-state abstraction. This shows the efficiency of our tensor-based implementations. It should be noted that the tensor computations is also exploited in the control synthesis step.

4.5 Comparison to existing tools

A comparison of the results on the benchmarks obtained by SySCoRe and current tools is given in Table 4. The package delivery benchmark has a complex DFA and cannot be handled natively by tools AMYTISS, FAUST, and StocHy (see Table 4). SReachTools can only handle safety specifications and is not applicable to this benchmark. The Van der Pol oscillator benchmark poses significant challenges for the tools due to its nonlinear dynamics, as reported in Table 4. Only AMYTISS can solve a benchmark that resembles this one as considered in [1] with multiplicative noise instead of additive noise. AMYTISS can only handle systems with a bounded disturbance, hence it cannot directly solve the benchmark as presented here.

The benchmark on the building automation system can be solved by AMYTISS, SReachTools, and StocHy without being able to use model-order reduction. This benchmark considers a stochastic safety problem and the performance of multiple tools is compared in [1, 2]. Table 4 reports the results of SySCoRe together with the results from running the repeatability packages of [1, 2] on a computer with a 2,3 GHz Quad-Core Intel Core i5 processor and 16 GB 2133 MHz memory. For StocHy, there was no repeatability package available, however, since the computational power of the CPU used for the results in [1] was more than our computer, we included the results of [1] as a lower bound on the computation time required by StocHy. Note that this benchmark belongs to the class of partially degenerate systems [35]. The formulation of the abstraction error for this class is available but the current version of FAUST does not natively support partially degenerate systems. With respect to the computation time, SReachTools performs best, and AMYTISS and StocHy require a longer computation time. Though from the results in [1], we see that AMYTISS could be faster than the current implementation of SySCoRe when parallel execution within CPUs is available (this parallel computation will be exploited in future versions of SySCoRe). With respect to accuracy, both AMYTISS and StocHy obtain a maximum reachability probability smaller than SySCoRe, while SReachTools still outperforms SySCoRe. This shows that SReachTools is the best option for this benchmark, which is expected since it is developed exactly for linear systems and stochastic reach-avoid problems with small disturbances.

5 SUMMARY AND EXTENSIONS

This paper described the first release of SySCoRe, a tool that excels at control synthesis problems for systems with a large (unbounded) stochastic disturbances and temporal specifications with possibly unbounded time horizon. It combines reduced-order models and finite abstractions with formal guarantees obtained by coupled stochastic simulation relations. SySCoRe substantially extends the class of models and specifications that current tools can handle for control synthesis. Furthermore, the modular development of SySCoRe allows ease of use and facilitates future extensions. The efficient implementation of tensor computations in SySCoRe allows for fast computations, which can be exploited further by including more parallel computations as done in AMYTISS.

An important direction for future releases is the implementation of model-order reduction to piecewise-affine systems, such that it

can also be applied to nonlinear systems. Currently, only Gaussian disturbances are implemented in SySCoRe, however, extensions to other distributions are under way and require deriving new inequality constraints for the optimization problem solved in the similarity quantification. The computation time of the similarity quantification is large due to solving optimization problems constrained by parameterized matrix inequalities that could be non-convex. We are working on improving the efficiency of solving this optimization.

The modular implementation of SySCoRe can be utilized to integrate model-order reduction with discretization-free approaches such as SReachTools [38, 41] and the barrier certificates [18], or to perform synthesis for stochastic systems with parametric uncertainty [33]. To get non-trivial lower bounds, SySCoRe currently requires fine-tuning the hyper parameters (e.g., the grid size and the output deviation). It is of interest to automatically design these parameters or to provide guidelines to the user on the appropriate values.

REFERENCES

[1] Alessandro Abate, Henk Blom, Marc Bouissou, Nathalie Cauchi, Hassane Chraibi, Joanna Delicaris, Sofie Haesaert, Arnd Hartmanns, Mahmoud Khaled, Abolfazl Lavaei, Hao Ma, Kaushik Mallik, Mathis Niehage, Anne Remke, Stefan Schupp, Fedor Shmarov, Sadegh Soudjani, Adam Thorpe, Vlad Turcuman, and Paolo Zuliani. 2021. ARCH-COMP21 Category Report: Stochastic Models. In *8th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH21) (EPiC Series in Computing, Vol. 80)*. EasyChair, 55–89.

[2] Alessandro Abate, Henk Blom, Nathalie Cauchi, Joanna Delicaris, Arnd Hartmanns, Mahmoud Khaled, Abolfazl Lavaei, Carina Pilch, Anne Remke, Stefan Schupp, et al. 2020. ARCH-COMP20 Category Report: Stochastic Models.. In *ARCH*. 76–106.

[3] Alessandro Abate, Henk Blom, Nathalie Cauchi, Sofie Haesaert, Arnd Hartmanns, Kendra Lesser, Meeko Oishi, Vignesh Sivaramakrishnan, Sadegh Soudjani, Cristian-Ioan Vasile, and Abraham P. Vinod. 2018. ARCH-COMP18 Category Report: Stochastic Modelling. *EPiC Series in Computing* 54 (2018), 71 – 103.

[4] Alessandro Abate, Henk Blom, Joanna Delicaris, Sofie Haesaert, Arnd Hartmanns, Birgit Van Huijgevoort, Abolfazl Lavaei, Hao Ma, Mathis Niehage, Anne Remke, Oliver Schön, Stefan Schupp, Sadegh Soudjani, and Lisa Willemsen. 2022. ARCH-COMP22 Category Report: Stochastic Models. (2022).

[5] Edward J Allen, Linda JS Allen, Armando Arciniega, and Priscilla E Greenwood. 2008. Construction of equivalent stochastic differential equation models. *Stochastic analysis and applications* 26, 2 (2008), 274–297.

[6] Rajeev Alur. 2015. *Principles of cyber-physical systems*. MIT press.

[7] Mosek ApS. 2019. Mosek optimization toolbox for matlab. *User's Guide and Reference Manual, Version* 4 (2019).

[8] C. Baier and J.-P. Katoen. 2008. *Principles of Model Checking*. MIT Press.

[9] Calin Belta, Boyan Yordanov, and Ebru Aydin Gol. 2017. *Formal methods for discrete-time dynamical systems*. Vol. 15. Springer.

[10] N. Cauchi and A. Abate. 2018. Benchmarks for cyber-physical systems: A modular model library for building automation systems. *IFAC-PapersOnLine* 51, 16 (2018), pp. 49–54.

[11] N. Cauchi and A. Abate. 2019. StocHy-automated verification and synthesis of stochastic processes. *Proc. of the 22nd ACM International Conference on Hybrid Systems: Computation and Control* (2019), 258–259.

[12] Paul Gastin and Denis Oddoux. 2001. Fast LTL to Büchi automata translation. In *International Conference on Computer Aided Verification*. Springer, 53–65.

[13] S. Haesaert and S. Soudjani. 2020. Robust dynamic programming for temporal logic control of stochastic systems. *IEEE TAC* (2020).

[14] S. Haesaert, S. Soudjani, and A. Abate. 2017. Verification of general Markov decision processes by approximate similarity relations and policy refinement. *SIAM Journal on Control and Optimization* 55, 4 (2017), 2333–2367.

[15] Arnd Hartmanns and Holger Hermanns. 2014. The Modest Toolset: An Integrated Environment for Quantitative Modelling and Verification. In *20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (Lecture Notes in Computer Science, Vol. 8413)*. Springer, 593–598.

[16] Martin Herceg, Michal Kvasnica, Colin N Jones, and Manfred Morari. 2013. Multi-parametric toolbox 3.0. In *2013 European control conference (ECC)*. IEEE, 502–510. http://control.ee.ethz.ch/~mpt.

[17] Jannik Hüls, Henner Niehaus, and Anne Remke. 2020. Hpnmg: A C++ Tool for Model Checking Hybrid Petri Nets with General Transitions. In *12th International NASA Formal Methods Symposium, NFM 2020*. Springer.

[18] Pushpak Jagtap, Sadegh Soudjani, and Majid Zamani. 2020. Formal synthesis of stochastic systems via control barrier certificates. *IEEE Trans. Automat. Control* 66, 7 (2020), 3097–3110.

[19] Niklas Kochdumper, Felix Gruber, Bastian Schürmann, Victor Gaßmann, Moritz Klischat, and Matthias Althoff. 2021. AROC: A toolbox for automated reachset optimal controller synthesis. In *Proceedings of the 24th International Conference on Hybrid Systems: Computation and Control*. 1–6.

[20] Orna Kupferman and Moshe Y Vardi. 2001. Model checking of safety properties. *Formal methods in system design* 19, 3 (2001), 291–314.

[21] Marta Kwiatkowska, Gethin Norman, and David Parker. 2002. PRISM: Probabilistic symbolic model checker. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*. Springer, 200–204.

[22] Yann Labit, Dimitri Peaucelle, and Didier Henrion. 2002. SeDuMi interface 1.02: a tool for solving LMI problems with SeDuMi. In *Proceedings. IEEE International Symposium on Computer Aided Control System Design*. IEEE, 272–277.

[23] Abolfazl Lavaei, Mahmoud Khaled, Sadegh Soudjani, and Majid Zamani. 2020. AMYTISS: Parallelized automated controller synthesis for large-scale stochastic systems. In *International Conference on Computer Aided Verification*. Springer, 461–474.

[24] Abolfazl Lavaei, Sadegh Soudjani, Alessandro Abate, and Majid Zamani. 2022. Automated verification and synthesis of stochastic hybrid systems: A survey. *Automatica* 146 (2022), 110617.

[25] Edward Ashford Lee and Sanjit Arunkumar Seshia. 2016. *Introduction to embedded systems: A cyber-physical systems approach*. MIT Press.

[26] Johan Lofberg. 2004. YALMIP: A toolbox for modeling and optimization in MATLAB. In *2004 IEEE international conference on robotics and automation (IEEE Cat. No. 04CH37508)*. IEEE, 284–289.

[27] Rupak Majumdar, Kaushik Mallik, and Sadegh Soudjani. 2020. Symbolic Controller Synthesis for Büchi Specifications on Stochastic Systems. In *Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Control* (Sydney, New South Wales, Australia) *(HSCC '20)*. Association for Computing Machinery, New York, NY, USA, Article 14, 11 pages.

[28] Petter Nilsson, Sofie Haesaert, Rohan Thakker, Kyohei Otsu, Cristian-Ioan Vasile, Ali-Akbar Agha-Mohammadi, Richard M Murray, and Aaron D Ames. 2018. Toward specification-guided active mars exploration for cooperative robot teams. (2018).

[29] Thrasyvoulos Pappas, Alan Laub, and Nils Sandell. 1980. On the numerical solution of the discrete-time algebraic Riccati equation. *IEEE Trans. Automat. Control* 25, 4 (1980), 631–641.

[30] Carina Pilch and Anne Remke. 2017. HYPEG: Statistical Model Checking for hybrid Petri nets: Tool Paper. In *Proceedings of the 11th EAI International Conference on Performance Evaluation Methodologies and Tools* (Venice, Italy) *(VALUETOOLS 2017)*. ACM, 186–191.

[31] Michael O Rabin and Dana Scott. 1959. Finite automata and their decision problems. *IBM journal of research and development* 3, 2 (1959), 114–125.

[32] 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.

[33] Oliver Schön, Birgit van Huijgevoort, Sofie Haesaert, and Sadegh Soudjani. 2022. Correct-by-Design Control of Parametric Stochastic Systems. In *2022 IEEE 61st Conference on Decision and Control (CDC)*. IEEE, 5580–5587.

[34] Fedor Shmarov and Paolo Zuliani. 2015. ProbReach: Verified Probabilistic δ-Reachability for Stochastic Hybrid Systems. In *HSCC*. ACM, 134–139.

[35] Sadegh Soudjani and Alessandro Abate. 2013. Probabilistic reach-avoid computation for partially degenerate stochastic processes. *IEEE Trans. Automat. Control* 59, 2 (2013), 528–534.

[36] Sadegh Soudjani, Caspar Gevaerts, and Alessandro Abate. 2015. FAUST^2 Formal Abstractions of Uncountable-STate STochastic Processes. In *International conference on tools and algorithms for the construction and analysis of systems*. Springer, 272–286.

[37] Paulo Tabuada. 2009. *Verification and control of hybrid systems: a symbolic approach*. Springer Science & Business Media.

[38] Adam J Thorpe, Kendric R Ortiz, and Meeko MK Oishi. 2021. SReachTools Kernel Module: Data-Driven Stochastic Reachability Using Hilbert Space Embeddings of Distributions. In *2021 60th IEEE Conference on Decision and Control (CDC)*. IEEE, 5073–5079.

[39] Birgit C van Huijgevoort and Sofie Haesaert. 2022. Similarity quantification for linear stochastic systems: A coupling compensator approach. *Automatica* 144 (2022), 110476.

[40] Birgit C van Huijgevoort, Siep Weiland, and Sofie Haesaert. 2022. Temporal logic control of nonlinear stochastic systems using a piecewise-affine abstraction. *IEEE Control Systems Letters* (2022).

[41] Abraham P Vinod, Joseph D Gleason, and Meeko MK Oishi. 2019. SReachTools: a MATLAB stochastic reachability toolbox. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*. 33–38.