Kickstarting PRACTICE with a Suite of Analysis Tools for Stochastic Vector Addition Systems

Landon Taylor[1][0000−0002−4071−3625]

Utah State University, Logan, UT 84322, USA
landon.jeffrey.taylor@usu.edu

Abstract. Quantitative analysis of safety properties in *Stochastic Vector Addition Systems* (σVAS) is a challenging problem with real-world impacts. σVAS models often describe highly-concurrent, infinite-state systems, and their associated safety properties are often extremely unlikely to occur (i.e., *rare events*). To that end, this paper presents a series of developments for a σVAS toolset under development, tentatively named *PRobabilistic Research and Analysis of C(RN/TMC) (and VAS) Tool Integration for Checking and Evaluation* (PRACTICE). A custom σVAS input specification format improves usability for real-world users, including synthetic biologists and others. PRACTICE includes an implementation of a dependency graph, which can be used to efficiently gather high-level property reachability information as well as abstract away details of a model. Using *Bounded Model Checking* (BMC), PRACTICE quickly finds ranges for variable bounds and allows SMT-based symbolic model analysis. RL-based witness trace generation enables the rapid discovery of high-probability witness traces, serving as a promising replacement for existing trace generation tools in RAGTIMER. These approaches are tested on a benchmarking suite of real-life σVAS models, and results indicate that PRACTICE has a high potential to benefit users.

1 Introduction

Analysis of quantitative safety properties in a *Stochastic Vector Addition System* (σVAS) is a challenging problem with real-world impacts. σVAS models often describe highly-concurrent, infinite-state systems, and their associated safety properties are often extremely unlikely to occur (i.e., *rare events*).

A *Chemical Reaction Network* (CRN) is a particularly interesting σVAS, as it models the continuous-time kinetics of a biochemical reaction network [5], genetic regulatory network [23], or molecular program [29]. The operating environment for a CRN is often extremely noisy, and irrelevant inputs exponentially increase the size of the reachable state space. Though they are extremely unlikely, rare events are of great interest in CRNs, as they represent undesirable behavior that can lead to severe (i.e., potentially pathological) consequences.

A continuous-time σVAS model (e.g., a CRN) is a *Continuous-Time Markov Chain* (CTMC), which provides a general-purpose framework for quantitative analysis. *Probabilistic Model Checking* (PMC) provides provable guarantees of

the transient reachability probability of a rare event in a σVAS. Unfortunately, transient analysis in PMC often suffers from state explosion due to its need to explicitly enumerate the intractible number of states often present in σVAS models. Several existing tools [15,27,25,12,14,31] construct a *partial* state space to combat the state explosion problem. *Stochastic Simulation Algorithms* (SSA) provide an alternative to PMC, but their results are estimates rather than guarantees, and they often require manual intervention [14].

Analysis of σVAS models remains a challenging problem, and to the author's knowledge, there is no unified toolset specialized to the analysis of σVAS models. To that end, this paper introduces a σVAS toolset under development, tentatively named *PRobabilistic Research and Analysis of C(RN/TMC) (and VAS) Tool Integration for Checking and Evaluation* (PRACTICE) [1]. Written using Rust best practices and designed to provide proofs of its own correctness, PRACTICE aims to attack the following challenges for σVAS analysis.

State Explosion. To combat state explosion, it is desirable that the state space for a σVAS contain only the *most useful* states. Thus, PRACTICE will employ a variety of methods—each suited to a particular type of model—to limit the size of the state space. This paper presents two methods: variable bounding and trace enumeration, along with potential heuristics and optimizations within each. Variable bounding is performed with *Bounded Model Checking* (BMC), and trace enumeration is performed using a simple *Reinforcement Learning* (RL) algorithm. This paper also presents a method that uses a dependency graph to reduce not only the number of states, but the *dimensionality* of the state space.

Model Intuition. It is often desirable to obtain intuition about a model before performing a full analysis. This paper documents a selection of probability-agnostic tools that can be used to provide several insights to a user without the need to perform a full analysis. These tools include a custom σVAS input format specification and parser; transition dependency analysis and abstraction; and BMC-based single (shortest) trace generation, k-step model exporting, and reachability analysis.

Paper Structure The remainder of the paper is outlined as follows:

Section 2 details preliminary topics.
Section 3 describes related work.
Section 4 describes a custom input specification for σVAS models.
Section 5 describes the procedure for creating a dependency graph and reasoning about a model based on its dependency relations.
Section 6 describes a selection of BMC-based approaches for analyzing a σVAS and reducing the size of its state space.
Section 7 describes a prototype RL-based algorithm for trace generation.

[1] PRACTICE is under active development on several distinct branches at `gh/formal-verification-research/practice`
This paper presents algorithms and results implemented on the `bmc` branch.
RL trace generation is at `gh/mossbiscuits/rl_traces` due to compatibility issues.

Section 8 describes preliminary benchmarking results.
Section 9 discusses potential future work and concludes the paper.
Appendix A includes algorithms to improve readability of the main body.
Appendix B includes model files discussed in the paper.

2 Preliminaries

Let $\mathbb{N}$ indicate the set of all non-negative integers (i.e., $0 \in \mathbb{N}$).

2.1 Stochastic Vector Addition Systems

A traditional (i.e., deterministic) *Vector Addition System* (VAS) with m variables is defined as a tuple $\boldsymbol{\Upsilon} \triangleq \langle \mathcal{U}, \mathcal{E}, s_0 \rangle$ as follows: s_0 is a vector describing exactly the model's specified initial state. $\mathcal{U} \in \mathbb{N}^m$ is the set of vectors $\boldsymbol{u}_i \in \mathcal{U}$ such that each $\boldsymbol{u}_i$ precisely describes the effect of transition i when added to the vector representing a state. $\mathcal{E}$ is the set of vectors $\boldsymbol{e}_i \in \mathcal{E}$ such that each $\boldsymbol{e}_i$ precisely describes the condition by which transition i becomes *enabled*. That is, if and only if every transition i is enabled at state s_α, it must also be true that every element of the difference $s_\alpha - \boldsymbol{e}_i$ is non-negative.

A VAS may be extended into the probabilistic domain with a general function that maps each update vector $\boldsymbol{u}_i \in \mathcal{U}$ to a probability. Define a *Stochastic Vector Addition System* (σVAS) $\sigma\boldsymbol{\Upsilon} \triangleq \langle \mathcal{U}, \mathcal{E}, \Theta, s_0 \rangle$ such that $\Theta : \mathcal{U} \times \mathbb{R}^m$ and the semantics of $\mathcal{U}$, $\mathcal{E}$, and s_0 remain the same. Semantics for Θ may be defined in discrete- or continuous-time, and the precise specification of the σVAS probability function is not important for understanding details in this report. Analysis agnostic of Θ enables simple reachability checking for a σVAS.

2.2 Chemical Reaction Networks

A *Chemical Reaction Network* (CRN) is a real-world continuous-time stochastic system that models the (often highly-concurrent) interactions between species (i.e., variables) and reaction (i.e., transitions). Transient CTMC analysis is of particular interest for CRN analysis, as time-bounded reachability probabilities are interesting to the biologists who model these systems. Informally, these properties are of the form "What is the probability that, within X seconds, species Y eventually increases to a count of Z?" CRNs follow the *Stochastic Chemical Kinetic* (SCK) model assumption, which imposes a limit of two reactants per reaction and disallows the consumption or production of more than two molecules of a single species per reaction execution [24].

To model a CRN as a σVAS, a total order $\leqslant$ is imposed on its species, and each state is modeled as a vector containing species counts ordered by $\leqslant$. Section 4 further describes the process for modeling a CRN or related σVAS in the correct input format for PRACTICE.

2.3 Continuous-Time Markov Chains

CRNs and similar continuous-time σVAS models induce a *Continuous-Time Markov Chain* (CTMC). Formally, a CTMC is a tuple $\mathtt{C} \triangleq \langle \mathcal{S}, s_0, \mathbf{R}, \mathbf{L} \rangle$ defined as follows: $\mathcal{S}$ is the set of reachable states (i.e., the *state space*), and $s_0 \in \mathcal{S}$ is the initial state—equivalent to s_0 in $\sigma\boldsymbol{\Upsilon}$. $\mathbf{R} : \mathcal{S} \times \mathcal{S} \to \mathbb{R}$ is the transition rate matrix, which defines the rate at which each state takes a transition to another state in units of $1/s$. The probability of the transition from s_i to s_j is $p(s_i, s_j) = \frac{\mathbf{R}(s_i, s_j)}{E(s_i)}$, where the *exit rate* $E(s_i)$ is the sum of the rate of all enabled transitions from s_i. A CTMC's time-bounded transient reachability property is given by $\mathsf{P}_{=?}(\Diamond^{[0,T]}\ \Psi)$, where Ψ is a non-nested *satisfying state formula*, specified in *Continuous Stochastic Logic* (CSL) [2,19]. While this report does not provide direct analysis of the time-bounded transient reachability probability, it does utilize the transition rate-to-probabiliy calculation in Section 7 as a heuristic to guide Reinforcement Learning.

2.4 Reinforcement Learning

Reinforcement Learning (RL) is a subfield of machine learning that focuses on how agents can learn to make decisions by interacting with their environment to maximize cumulative rewards. Unlike supervised learning, where models are trained on labeled datasets, RL operates on the principle of trial and error, allowing agents to explore various actions and learn from the consequences of their choices [30]. This paradigm is grounded in the concepts of *Markov Decision Processes* (MDPs), where the agent's objective is to discover an optimal policy that dictates the best action to take in each state to achieve the highest long-term reward [26]. The versatility of RL has led to its application across diverse domains, including robotics, game playing, and autonomous systems. This work applies RL to the problem of trace generation, rewarding traces with high target-reachability probabilities.

2.5 Bounded Model Checking

Checking the reachability of safety property violations is commonly approached with *Bounded Model Checking* (BMC) due to its efficiency at trace generation. BMC uses a *Satisfiability Modulo Theories* (SMT) checker (e.g., Yices [11] or Z3 [9]) to symbolically check the i-step bounded reachability of a given formula. k-induction extends BMC by performing a k-step inductive check following an n-step BMC run [28]. This technique was explored but found to be unhelpful for this paper's contributions.

3 Related Work

3.1 σVAS Reachability Analysis

Reachability in traditional VAS models has been thoroughly addressed [17,21,20], complete with algorithms and theoretical frameworks for determining reachabil-

ity, including polynomial-time algorithms for specific subclasses of VAS and the exploration of decidability issues in more complex scenarios. Furthermore, the integration of reachability analysis with other verification techniques, such as model checking and abstraction refinement, has shown promise in enhancing the efficiency and effectiveness of system verification. Extension to the probabilistic domain for σVAS verification is largely limited to discrete-time models [16,22]. Limited effort appears to exist for continuous-time σVAS models like the ones analyzed in this work.

3.2 BMC Model Analysis

BMC is a well-developed framework to systematically explore the state space of a system up to a specified depth, allowing for the detection of errors within a finite number of steps [8,7]. Researchers have explored the combination of BMC with other verification techniques, such as symbolic model checking [8] and abstraction refinement, to address the limitations of state explosion and to handle larger and more complex systems. Tools like Z3 [9], Yices [11], and NuSMV [6] exemplify practical implementations of BMC, showcasing its applicability in verifying software and hardware systems, including BMC encodings of VASs as detailed in Section 6. Recent work uses BMC to automatically generate variable bounds [1]. This paper extends the work by generating a range of bounds for each variable, as described in Section 6.

3.3 Witness Trace Enumeration

Witness trace (i.e., counterexample) generation is a popular approach for model analysis and partial state space construction. RAGTIMER [31]—to which Section 7 proposes a successor—generates a large number of witness traces, then inserts cycles and concurrent traces to build a partial state space for CTMC analysis. DTMC witness trace generation enables the collection of the most probable shortest paths to refute a probabilistic bounded property [13]. It appears witness trace generation for continuous-time models still requires great effort.

4 σVAS Input Format

A CRN format was specified at `docs.mossbiscuits.com/crn/input_format`. It is formalized and further developed in this paper, including syntax and semantics for a general VAS and additional vocabulary for a σVAS or CRN. At present, the input format accepts only probability-agnostic or continuous-time σVAS models, as these models are of primary interest for the development of PRACTICE.

4.1 Overview

The VAS input format is designed to be minimalistic and simple, reducing the burden of education for non-technical users, including synthetic biologists and

experts in various domains. Each line of the input specifies a unique property of the model, with a flexible dictionary of keywords in plain English.

For example, a continuous-time σVAS input follows the format Listing 1. The first 3 lines declare the names of variables along with their initialization. Line 4 declares a target variable and its desired value. To reach the target, this model must increase `S3` by a count of `40`. Lines 5-8 and 9-12 declare two transitions. Lines 5 and 9 declare the transition names. Lines beginning with `increase` and `decrease` define the update vector and enabled bounds for each transition. For a continuous-time σVAS, Line 8 and 12 declare transition rate constants, adhering to the CRN SCK assumption and probability semantics.

```
1  var S1 init 100
2  var S2 init 200
3  var S3 init 300
4  target S3 = 340
5  transition R1
6    decrease S1 10
7    increase S2 3
8    const 0.4
9  transition R2
10   decrease S1 1
11   increase S3 1
12   const 0.6
```

Fig. 1. Custom σVAS Input Format Example

4.2 Syntax & Semantics

This section formally defines the syntax and semantics of the σVAS input format for the parsing and construction of a generic σVAS $\sigma\boldsymbol{\Upsilon} \triangleq \langle \mathcal{U}, \mathcal{E}, \Theta, s_0 \rangle$. Declarations occupy exactly one line each. Tokens are space- or tab-separated, and newline characters are disallowed within a variable declaration.

Variables and Initial States. For user convenience, an σVAS may include a vector of variable names and a total ordering $\leqslant$ imposed thereon defined by the order in which variables are declared in the input file. The total order $\leqslant$ is preserved through the entire σVAS model. Variable declarations traditionally appear at the beginning of the σVAS input specification, but they may be present on any line. Variables must be declared exactly once per model.

Any line whose first token is in { `var`, `variable`, `species` } is interpreted as a variable declaration. Within a variable declaration, the second token is interpreted as the variable's name (used only for user convenience). If a third token is present, it must be in {`init`, `initial`}, and its purpose is to declare the

initial state for that variable. The fourth token must be present if the third token is present, and it defines the initial state corresponding to the named variable. If only two tokens are present, the variable is assigned an initial value of 0.

For example, Listing 2 shows a simple variable initialization. The order imposed on these variables is $\texttt{S1} \leqslant \texttt{S2} \leqslant \texttt{S3}$. The variable name vector is $[\texttt{S1}, \texttt{S2}, \texttt{S3}]$, and $s_0 = [100, 200, 300]$.

```
var S1 init 100
var S2 init 200
var S3 init 300
```

Fig. 2. Variable Declaration

Transitions For user convenience, transitions are assigned names, and their corresponding vectors use the total ordering $\leqslant$ imposed during variable parsing. Any line whose first token is in {`transition`, `reaction`} is considered a transition declaration. The second token of the line is the transition's name. Initially, each transition t is assumed to have a rate constant of 0, and $\boldsymbol{u}_t = \boldsymbol{e}_t = \mathbf{0}$, such that $\boldsymbol{u}_t \in \mathcal{U}$, and $\boldsymbol{e}_t \in \mathcal{E}$. Transitions must be defined exactly once.

Consider $\boldsymbol{x}[v]$ to indicate a valuation of vector $\boldsymbol{x}$ at the position of the variable named v. Consider a line l such that transition t is declared at the greatest line number not exceeding the line number of l. If the first token of l is in {`decrease`, `decrement`, `consume`, `increase`, `increment`, `produce`}, this defines a deviation from the default values for $\boldsymbol{u}_t$ and $\boldsymbol{e}_t$. Contradictory updates to variables (i.e., multiple increases or decreases for a single variable) are not permitted.

The first token indicates whether the variable should increase or decrease upon execution of t. A token in {`increase`, `increment`, `produce`} indicates an increase, while a token in {`decrease`, `decrement`, `consume`} indicates a decrease. Within l, the second token l_2 indicates the affected variable v, and the third token (i_3 for increasing and d_3 for decreasing), indicates the amount by which transition t decreases the affected variable. If only two tokens are present, the value of i_3 or d_3 is assumed to be 1. Formally, for each line l describing variable v, $\boldsymbol{u}[v] = \sum_{\forall l} i_3 - d_3$ and $\boldsymbol{e}[v] = \sum_{\forall l} d_3$. Intuitively, the update vector for each transition is the sum of increases and decreases for each variable, and the enabled vector is the sum of decreases for each variable.

Similarly, if the first token of l_t is in {`const`, `rate`}, the probability function Θ is updated to reflect a rate constant matching the second token of l_t. If a rate formula is desired instead, the token `formula` should precede the specification of a rate formula using mathematical notation identical to expressions specified by the Rust language.

For example, Listing 3 shows a simple transition description. Imposing the ordering from Listing 2, $\boldsymbol{u}_{\texttt{R1}} = [-10, 3, 0]$ and $\boldsymbol{e}_{\texttt{R1}} = [10, 0, 0]$. Similarly, $\boldsymbol{u}_{\texttt{R2}} =$

$[-1, 0, 1]$ and $\boldsymbol{e}_{\texttt{R2}} = [1, 0, 0]$. The rate constant for `R1` is 0.4, and the rate constant for `R2` is 0.6.

```
transition R1
  decrease S1 10
  increase S2 3
  const 0.4
transition R2
  decrease S1 1
  increase S3 1
  const 0.6
```

Fig. 3. Transition Specification

Property Specification A target property (often a violation of a safety property) is specified on lines whose first token is in {`target`, `goal`, `prop`, `check`}. This token is followed by an expression that uses mathematical functions defined in the Rust specification. At present, exactly one target should be present in a model. For example, Listing 4 describes a target in which the value of `S3` is 340.

```
target S3 = 340
```

Fig. 4. Property Specification

4.3 Limitations

Current tool implementations allow only a subsets of models implementable in the σVAS input format. While the format is non-restrictive for a generic σVAS, properties are limited to equivalence relations until a more generic expression parsing library is developed. Similarly, transition rate formulas are not yet implemented, as they require a fully-features expression parser, which is currently under active development by Practice developers.

5 Dependency Graph

Dependency graphs are used extensively to produce heuristics for witness trace generation and model exploration (ISR citation: paper under review). This paper presents a Rust implementation of dependency graph construction for general σVAS models. This dependency graph will be used in Rust reimplementations of tools like Ragtimer [31] and Wayfarer.

5.1 Dependency Graph Construction

The dependency graph construction algorithm presented in Algorithm 1 and described more in-depth in [14] is a recursive method designed to build a representation of dependencies among various transitions in a σVAS. It first constructs a node representing an artificial transition that models the target specification. It then initializes the state of child nodes based on the current node's properties and calculates the required target values for each child node. By iterating through potential dependencies, the algorithm identifies valid paths that can be taken based on the current state and the defined transitions, while also handling cases of negative targets (i.e., targets that require the decrease of a variable). The algorithm merges duplicate child nodes to optimize the graph structure, ensuring that only relevant nodes are retained. Finally, it recursively processes each child node, updating the overall state of the parent node based on the enabled status of its children.

5.2 Reachability Analysis

The dependency graph can be used for reachability analysis in a CRN, and by extension in a σVAS [14]. Essentially, if a dependency graph is able to be constructed, the target is reachable by at least one witness trace. If a dependency graph is not able to be constructed, then there are no enabled transitions that comprise a witness trace to a target, so the target is unreachable.

5.3 Slashing Abstraction

This paper proposes an abstraction method designed to speed up BMC and other analysis for shortest traces by reducing the *dimensionality* of a σVAS model. Consider a model that requires only a fixed number of transition executions to reach a target state. First, it is likely that these executions do not include *every* transition in the model. Further, it is likely that these transition executions does not change the value of *every* variable. Thus, in many models, shortest trace generation and simple model analysis may benefit from *Slashing Abstraction*, or the elimination (slashing) of entire transitions and variables from the model.

Algorithm 2 presents a simple Slashing Abstraction algorithm. Essentially, the algorithm filters the transitions that are included in the dependency graph and the variables those transitions affect, then builds a new "trimmed" model including only those transitions and variables. It adjusts the size of the vectors in the model, preserving the variables' total order $\leqslant$.

Algorithm 2 only provides a valid abstraction in the context of *shortest* witness traces (i.e., witness traces containing the fewest possible transition executions). Longer traces may include spurious transitions—transitions that are not absolutely required in order to reach a target—so the abstracted σVAS model produced by the algorithm will not allow exploration of witness traces containing those spurious reactions.

5.4 Mining Witness Traces

A dependency graph is an excellent candidate for guiding witness trace generation. At the simplest level, it is possible to manually construct a complete witness trace from a dependency graph by beginning at leaf nodes, executing every transition the requisite number of times, then traversing the tree toward the root node. Both RAGTIMER and Wayfarer use a dependency graph to guide witness trace generation and state space construction, and Section 7 presents an RL-based approach that initially assigns a high reward to transitions that are present in the dependency graph.

6 Bounded Model Checking

Because σVAS models are conducive to probability-agnostic analysis, BMC is an appealing method for several types of model analysis. Traditionally, BMC is used primarily for witness trace generation. However, by systematically encoding a σVAS model into an SMT format and performing BMC unrolling, it is possible to perform many different model analyses.

6.1 SMT-Lib Model Formula

PRACTICE includes a BMC encoder for σVAS models. This encoder was built with the Z3 Rust bindings[2] due to the high compatibility of Z3. Originally, the author attempted to use the Rust Yices bindings[3], but it appears these bindings are deprecated.

Algorithm 3 builds a Z3 encoding for a generic σVAS model. This encoding reflects best practices in SMT encoding for transition systems: it disjoints the update constraints for each transition, then conjoins that transition encoding with the initial state and violation of a safety property (i.e., target specification). This encoding can be exported to a file in a standard SMT format, including SMT-Lib [3] for checking with any desired SMT solver. However, it is often more practical to allow PRACTICE to handle SMT solving with Z3 internally.

6.2 Bounded SMT Unrolling

It is possible to check the validity of s_0 with the formula output from Algorithm 3 alone, but an unroller is required to perform BMC. Algorithm 4 shows the data structure and a selection of methods for generating an unrolling, which can also be exported to a general SMT format.

[2] See `https://docs.rs/z3/latest/z3`
[3] See `https://docs.rs/yices2/latest/yices2`

6.3 Witness Trace Generation

The unrolling procedure enables traditional BMC witness trace generation for a σVAS model. A trace is simply a satisfying assignment to the formula generated by the BMC encoding in conjunction with the unroller up to a given step. For generating a single trace, this can provide some intuition, but trace generation is prohibitively slow for generating large quantities of traces. Section 7 more adequately solves the issue of witness trace generation by using RL and heuristics based on a σVAS model's dependency graph.

6.4 Variable Bound Deduction

A critical challenge for state space reduction is finding adequate variable bounds. BMC provides provable guarantees for reachability at a given length k. By encoding properties about variable bounds, it is possible to efficiently discover the minimal and maximal values within a witness trace for each variable.

This paper discusses four key bounds—loose and tight upper and lower bounds—defined as follows:

Loose Upper Bounds describe the *maximal* value for a variable during the execution of *all* witness traces of length k.
Tight Upper Bounds describe the *maximal* value for a variable during the execution of *at least one* witness trace of length k.
Loose Lower Bounds describe the *minimal* value for a variable during the execution of *all* witness traces of length k.
Tight Lower Bounds describe the *minimal* value for a variable during the execution of *at least one* witness trace of length k.

Intuitively, by imposing the loose bounds on a model, it is possible to generate *all* witness traces while remaining within the bounds. In other words, *it is not possible to generate a witness trace of length k with a variable outside the loosest bounds*. This statement is treated as a safety property after generating a BMC trace of length k, enabling the discovery of loosest bounds. Similarly, imposing a tight bound on a model means *it is not possible to generate at least one trace with a tighter bound*, which is similarly possible to encode in a BMC formula.

This procedure is described at a high level in Algorithm 5. Essentially, it performs a binary search starting at the extremes of the possible range of variable values. It iteratively searches for a bound where a trace is not found, then adjusts the possible range of values to narrow the search space for the bound.

7 Witness Trace Generation & Reinforcement Learning

Generating adequate witness traces to accumulate the reachability probability for a σVAS model is an extremely challenging problem. It is challenging to determine when to explore new transitions for witness traces versus following known transitions. RAGTIMER attempts to address this challenge by randomly

simulating traces, giving preference to transitions in a dependency graph. This problem can be represented as a stochastic game: it is desirable to maximize the probability of traces without knowing probabilities in advance.

Reinforcement Learning (RL) is particularly well-suited to these kinds of games. In a prototype development, an RL alternative to the resource-intensive trace generation from RAGTIMER appears promising. At a high level, the RL approach builds a dependency graph for a σVAS model, then assigns transitions present in the dependency graph a high reward (10.0 in the example), assigning a low reward (1.0 in the example) to unnecessary transitions. Thus, the algorithm favors transitions that lead to a target without excluding transitions that may boost the reachability probability. In this way, witness trace generation is not restricted to shortest traces, but it does favor short traces to long ones.

Algorithm 6 provides a high-level description of the RL procedure. The algorithm simply generates a trace, favoring transitions with a higher reward and attempting to maximize the trace probability. After each trace is generated, the reward function is updated for the next trace generation cycle. Rewards are normalized periodically to prevent any reward from growing to an unreasonable size.

8 Benchmarks & Practical Applications

A set of interesting and challenging case studies was used to validate the effectiveness and practicality of each approach presented in this paper. For some techniques, this section presents detailed benchmarks, while for others, approaches were simply required to pass a feasibility test. All benchmarks were completed on a machine with an Intel Core i7 14700F processor using 32 GB of RAM and running Ubuntu 24.

8.1 Case Studies

Several challenging studies were used to benchmark each method, described as follows. A high-level model description is provided in this paper; see the source material for each for an in-depth description, and see Appendix B for equivalent models in the custom σVAS format presented in Section 4.

Single Species Production-Degradation The *Single Species Production-Degradation* Model (SSPD) consists of two species in a simple two-reaction production-degradation interaction [18]. s_0 for $\mathsf{S}_1 \leqslant \mathsf{S}_2$ is $[1, 40]$. The property of interest is $\mathsf{P}_{=?}(\Diamond^{[0,100]}\, \mathsf{S}_2 = 80)$. The transitions are specified as follows:

$$\mathcal{R}_1 : \mathsf{S}_1 \xrightarrow{1.0} \mathsf{S}_1 + \mathsf{S}_2$$
$$\mathcal{R}_2 : \mathsf{S}_2 \xrightarrow{0.025} \emptyset$$

Modified Yeast Polarization The *Modified Yeast Polarization* Model (MYP) consists of seven species interacting through eight reactions [18]. Given the ordering $\mathrm{R} \leqslant \mathrm{L} \leqslant \mathrm{RL} \leqslant \mathrm{G} \leqslant \mathrm{G_a} \leqslant \mathrm{G_{bg}} \leqslant \mathrm{G_d}$, the initial state $s_0 = [50, 2, 0, 50, 0, 0, 0]$. This model is an adaptation of the pheremone-induced G-protein cycle in Saccharomyces cerevisia [**?**]. The property of interest is $\mathsf{P}_{=?}(\Diamond^{[0,20]}\, G_{bg} = 50)$. The transitions are specified as follows:

$$\mathcal{R}_1 :\ \emptyset \xrightarrow{0.0038} \mathrm{R}, \quad \mathcal{R}_2 :\ \mathrm{R} \xrightarrow{4.00\times10^{-4}} \emptyset, \quad \mathcal{R}_3 :\ \mathrm{L}+\mathrm{R} \xrightarrow{0.042} \mathrm{RL}+\mathrm{L},$$
$$\mathcal{R}_4 :\ \mathrm{RL} \xrightarrow{0.010} \mathrm{R}, \quad \mathcal{R}_5 :\ \mathrm{RL}+\mathrm{G} \xrightarrow{0.011} \mathrm{G_a}+\mathrm{G_{bg}}, \quad \mathcal{R}_6 :\ \mathrm{G_a} \xrightarrow{0.100} \mathrm{G_d},$$
$$\mathcal{R}_7 :\ \mathrm{G_d}+\mathrm{G_{bg}} \xrightarrow{1.05\times10^{3}} \mathrm{G}, \quad \mathcal{R}_8 :\ \emptyset \xrightarrow{3.21} \mathrm{RL}.$$

Enzymatic Futile Cycle The *Enzymatic Futile Cycle* Model (EFC) consists of six species interacting over six reactions [18]. The initial state under ordering $[\mathsf{S}_1 \leqslant \mathsf{S}_2 \leqslant \mathsf{S}_3 \leqslant \mathsf{S}_4 \leqslant \mathsf{S}_5 \leqslant \mathsf{S}_6]$ is $s_0 = [1, 50, 0, 1, 50, 0]$. The property of interest is $\mathsf{P}_{=?}(\Diamond^{[0,100]}\, \mathsf{S}_5 = 25)$. The transitions are defined as follows:

$$\mathcal{R}_1 :\ \mathsf{S}_1+\mathsf{S}_2 \xrightarrow{1.0} \mathsf{S}_3, \ \mathcal{R}_2 :\ \mathsf{S}_3 \xrightarrow{1.0} \mathsf{S}_1+\mathsf{S}_2, \quad \mathcal{R}_3 :\ \mathsf{S}_3 \xrightarrow{0.1} \mathsf{S}_1+\mathsf{S}_5,$$
$$\mathcal{R}_4 :\ \mathsf{S}_4+\mathsf{S}_5 \xrightarrow{1.0} \mathsf{S}_6, \ \mathcal{R}_5 :\ \mathsf{S}_6 \xrightarrow{1.0} \mathsf{S}_4+\mathsf{S}_5, \quad \mathcal{R}_6 :\ \mathsf{S}_6 \xrightarrow{0.1} \mathsf{S}_4+\mathsf{S}_2.$$

Simplified Motility Regulation The *Simplified Motility Regulation* Model (SMR) consists of nine species interacting over twelve reactions [**?**]. Under ordering $\mathrm{codY} \leqslant \mathrm{flache} \leqslant \mathrm{SigD\_hag} \leqslant \mathrm{CodY} \leqslant \mathrm{CodY\_flache} \leqslant \mathrm{hag} \leqslant \mathrm{CodY\_hag} \leqslant \mathrm{SigD} \leqslant \mathrm{Hag}$, the initial state is $s_0 = [1, 1, 1, 10, 1, 1, 1, 10, 10]$. The property of interest is $\mathsf{P}_{=?}(\Diamond^{[0,10]}\, \mathrm{CodY} = 20)$. The transitions are specified as follows:

$$\mathcal{R}_1 :\ \mathrm{codY} \xrightarrow{0.1} \mathrm{codY}+\mathrm{CodY}, \quad \mathcal{R}_2 :\ \mathrm{CodY} \xrightarrow{0.0002} \emptyset,$$
$$\mathcal{R}_3 :\ \mathrm{flache} \xrightarrow{1.0} \mathrm{flache}+\mathrm{SigD}, \quad \mathcal{R}_4 :\ \mathrm{SigD} \xrightarrow{0.0002} \emptyset,$$
$$\mathcal{R}_5 :\ \mathrm{SigD\_hag} \xrightarrow{1.0} \mathrm{SigD}+\mathrm{hag}+\mathrm{Hag}, \quad \mathcal{R}_6 :\ \mathrm{Hag} \xrightarrow{0.0002} \emptyset,$$
$$\mathcal{R}_7 :\ \mathrm{SigD}+\mathrm{hag} \xrightarrow{0.01} \mathrm{SigD\_hag}, \quad \mathcal{R}_8 :\ \mathrm{SigD\_hag} \xrightarrow{0.1} \mathrm{SigD}+\mathrm{hag},$$
$$\mathcal{R}_9 :\ \mathrm{CodY}+\mathrm{flache} \xrightarrow{0.02} \mathrm{CodY\_flache}, \quad \mathcal{R}_{10} : \mathrm{CodY\_flache} \xrightarrow{0.1} \mathrm{CodY}+\mathrm{flache},$$
$$\mathcal{R}_{11} : \mathrm{CodY}+\mathrm{hag} \xrightarrow{0.01} \mathrm{CodY\_hag}, \quad \mathcal{R}_{12} : \mathrm{CodY\_hag} \xrightarrow{0.1} \mathrm{CodY}+\mathrm{hag}.$$

8.2 σVAS Input Format

Each of the above case studies has been specified successfully in the custom σVAS input format specified in Section 4. These models are found in Appendix B. Conversion to the custom σVAS input format was simple, and can potentially be automated for very large models. This leads to the conclusion that the custom σVAS input format is expressive enough to encapsulate complex CRN behavior.

8.3 Dependency Graph Construction

For all provided case studies, PRACTICE generates a dependency graph that matches the author's manual analysis of the σVAS model. For each case study,

dependency graph generation required a negligible amount of time (less than one second) and memory (less than 1 MB). PRACTICE will eventually contain a dependency graph visualization tool, but for now, it prints a nicely formatted plain-text version of the graph.

8.4 SMT Modeling for BMC

For all case studies, PRACTICE efficiently constructs a BMC encoding as well as an unrolling up to a specified number of steps. If the specified number of steps is greater than the number of steps required for target reachability, unrolling terminates upon target reachability.

8.5 Slashing Abstraction

Each case study's dependency graph enables Slashing Abstraction. That is, it is possible to remove variables and transitions from each model. Results from this are described in Table 1. The most complex models see a great benefit from this method; at least half of the transitions from each model are removed without loss of information for shortest-trace generation. As the following section notes, this makes analysis possible for some models which were previously not feasible.

Model	Variables Removed	Transitions Removed
SSPD	$\emptyset$	$\mathcal{R}_1$
MYP	G_D	$\mathcal{R}_1$, $\mathcal{R}_2$, $\mathcal{R}_4$, $\mathcal{R}_6$, $\mathcal{R}_7$
EFC	S0, S2	$\mathcal{R}_1$, $\mathcal{R}_2$, $\mathcal{R}_3$
SMR	S3, S4, S6	$\mathcal{R}_1$, $\mathcal{R}_2$, $\mathcal{R}_3$, $\mathcal{R}_4$, $\mathcal{R}_5$ $\mathcal{R}_6$, $\mathcal{R}_7$

Table 1. Results from Slashing Abstraction.

8.6 Variable Bounding

While BMC variable bounding can be time-intensive (requiring several minutes on large models), it is memory-efficient and may enable the analysis of models that are otherwise impossible to check due to memory constraints. Because the size of a state space is largely based on combinatorial variable values, reducing the ranges of values can significantly reduce the size of a state space.

Table 2 shows results from a test of variable bound generation in PRACTICE. For each of the three most complex case studies (MYP, EFC, and SMR), BMC analysis of the entire model was not able to complete within the time limit of 10 minutes. However, trimming the model enabled the analysis to complete within 10 minutes for two models (MYP and EFC). This leads to the conclusion that—while it is not necessarily ideal for trace generation of non-shortest traces—Slashing Abstraction is a useful way to obtain *some* analysis on models where analysis was not otherwise feasible.

Model	Time (seconds)	Memory
SSPD	2.786	< 1 MB
MYP	Timeout	< 1 MB
EFC	Timeout	< 1 MB
SMR	Timeout	< 1 MB
SSPD (Abstracted)	28.391	< 1 MB
MYP (Abstracted)	436.229	< 1 MB
EFC (Abstracted)	414.998	< 1 MB
SMR (Abstracted)	Timeout	< 1 MB

Table 2. Results from variable bounding benchmarks.

8.7 RL-Based Witness Trace Generation

Because of the nature of RL, testing often produces noisy results until fine tuning the learning model. Testing on the MYP model regularly produces traces with an execution probablility around 10^{-50}. This probability gives an estimate of (but is not exactly) the transient reachability probability desired. To obtain the transient reachability probability, a partial state space may be explicitly constructed, as in Ragtimer. Trace generation is significantly faster than that of Ragtimer. For example, Ragtimer often requires 1 to 5 minutes to set up and compile a model; RL-based witness trace generation can generate over $100,000$ traces in that time. Further, RL-based witness trace generation does not require complex dependencies like the existing implementation of Ragtimer does. Figure 5 shows the trace execution probability during the generation of traces. In general, the average probability of traces increases over time as traces are generated, which indicates that RL is a promising approach for trace generation.

9 Discussion

Many σVAS models are intractibly complex to analyze, and the presented methods enhance the analysis of these models. Evaluation on a real-life benchmark suite indicate that Practice will be a useful toolset in practical settings. A custom input format allows for simple modeling of σVAS systems by laypeople. A dependency graph enables efficient analysis of a σVAS model on a symbolic level, including the ability to make significant cuts to a state space with Slashing Abstraction. BMC-based approaches enable efficient bounding of variables and model exploration with many SMT tools. An RL-based witness trace generation algorithm enables the rapid collection of witness traces into a state space, enabling state space construciton in the style of Ragtimer.

9.1 Future Work

While results from this work are promising, Practice requires a significant amount of effort to develop a fully-operational, verified toolset. The primary

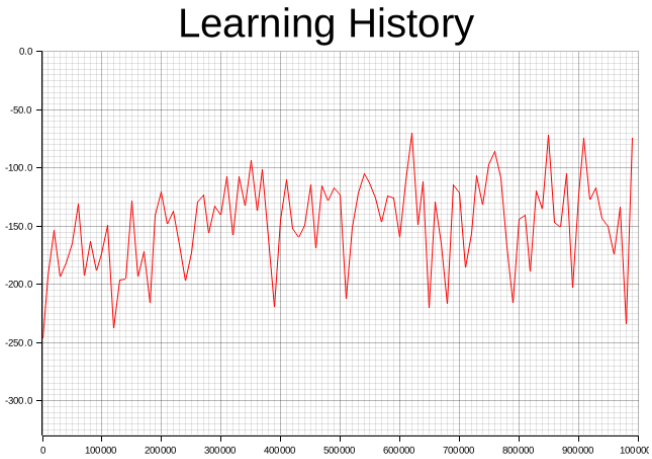

Fig. 5. Probability History for RL Model. The x-axis is the index of the trace (with traces randomly sampled), and the y-axis is the log-scale probability.

concern in future work is compatibility with Rust verification tools. Creusot [10] appears most promising, but a bug involving Rust `traits` has been discovered during PRACTICE development and has not yet been resolved.

Further development of methods presented in this paper can include improved symbolic σVAS analysis using more advanced methods, including *Property-Directed Reachability* (PDR) [4] or bi-directional state space exploration. Additionally, RL-based witness trace generation appears promising, and it requires significant enhancements, generalization, and integration with PRACTICE.

References

1. Ahmadi, M., Buecherl, L., Myers, C.J., Zhang, Z., Winstead, C., Zheng, H.: Rare-Event Guided Analysis of Infinite-State Chemical Reaction Networks. In: Hillston, J., Soudjani, S., Waga, M. (eds.) Quantitative Evaluation of Systems and Formal Modeling and Analysis of Timed Systems. pp. 196–212. Springer Nature Switzerland, Cham (2024). https://doi.org/10.1007/978-3-031-68416-6_12
2. Aziz, A., Sanwal, K., Singhal, V., Brayton, R.: Model-checking continuous-time markov chains. ACM Trans. Comput. Logic **1**(1), 162–170 (Jul 2000)
3. Barrett, C., Fontaine, P., Tinelli, C.: The satisfiability modulo theories library (SMT-LIB). <code>www.SMT-LIB.org</code> (2016)
4. Bradley, A.R.: SAT-Based Model Checking without Unrolling. In: Jhala, R., Schmidt, D. (eds.) Verification, Model Checking, and Abstract Interpretation, vol. 6538, pp. 70–87. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_7
5. Chellaboina, V., Bhat, S.P., Haddad, W.M., Bernstein, D.S.: Modeling and analysis of mass-action kinetics. IEEE Control Systems Magazine **29**(4), 60–78 (2009)
6. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In: Brinksma, E., Larsen, K.G. (eds.) Computer Aided Verification. pp. 359–364. Springer, Berlin, Heidelberg (2002). https://doi.org/10.1007/3-540-45657-0_29
7. Clarke, E.M., Grumberg, O., Kroening, D., Peled, D., Veith, H.: Model Checking. The Cyber-Physical Systems Series, The MIT Press, Cambridge, Massachusetts London, England, second edition edn. (2018)
8. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.): Handbook of Model Checking. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8
9. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
10. Denis, X., Jourdan, J.H., Marché, C.: Creusot: A Foundry for the Deductive Verification of Rust Programs. In: ICFEM 2022 - 23th International Conference on Formal Engineering Methods. Springer Verlag (Oct 2022)
11. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) Computer-Aided Verification (CAV'2014). Lecture Notes in Computer Science, vol. 8559, pp. 737–744. Springer (Jul 2014)
12. Hahn, E.M., Hermanns, H., Wachter, B., Zhang, L.: Infamy: An infinite-state markov model checker. In: Proceedings of the 21st International Conference on Computer Aided Verification. pp. 641–647. CAV '09, Springer-Verlag, Berlin, Heidelberg (2009)
13. Han, T., Katoen, J.P.: Counterexamples in probabilistic model checking. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 72–86. Springer (2007)
14. Israelsen, B., Taylor, L., Zhang, Z.: Efficient Trace Generation for Rare-Event Analysis in Chemical Reaction Networks. In: Caltais, G., Schilling, C. (eds.) Model Checking Software. vol. 13872, pp. 83–102. Springer Nature Switzerland, Cham (May 2023). https://doi.org/10.1007/978-3-031-32157-3_5

15. Jeppson, J., Volk, M., Israelsen, B., Roberts, R., Williams, A., Buecherl, L., Myers, C.J., Zheng, H., Winstead, C., Zhang, Z.: STAMINA in C++: modernizing an infinite-state probabilistic model checker. In: Jansen, N., Tribastone, M. (eds.) Quantitative Evaluation of Systems - 20th International Conference, QEST 2023, Antwerp, Belgium, September 20-22, 2023, Proceedings. Lecture Notes in Computer Science, vol. 14287, pp. 101–109. Springer (2023). https://doi.org/10.1007/978-3-031-43835-6_7, https://doi.org/10.1007/978-3-031-43835-6_7
16. Kučera, A.: On the Existence and Computability of Long-Run Average Properties in Probabilistic VASS. In: Kosowski, A., Walukiewicz, I. (eds.) Fundamentals of Computation Theory. pp. 12–24. Lecture Notes in Computer Science, Springer International Publishing, Cham (2015). https://doi.org/10.1007/978-3-319-22177-9_2
17. Kučera, A., Leroux, J., Velan, D.: Efficient Analysis of VASS Termination Complexity. In: Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science. pp. 676–688. LICS '20, Association for Computing Machinery, New York, NY, USA (Jul 2020). https://doi.org/10.1145/3373718.3394751
18. Kuwahara, H., Mura, I.: An efficient and exact stochastic simulation method to analyze rare events in biochemical systems. The Journal of Chemical Physics **129**(16), 165101 (Oct 2008). https://doi.org/10.1063/1.2987701
19. Kwiatkowska, M., Norman, G., Parker, D.: Stochastic Model Checking, pp. 220–270. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
20. Leroux, J.: Polynomial Vector Addition Systems With States. In: Chatzigiannakis, I., Kaklamanis, C., Marx, D., Sannella, D. (eds.) 45th International Colloquium on Automata, Languages, and Programming, {ICALP} 2018, July 9-13, 2018, Prague, Czech Republic. LIPIcs, vol. 107, pp. 134:1–134:13. Schloss Dagstuhl - Leibniz-Zentrum f{\"{u}}r Informatik, Prague, Czech Republic (Jul 2018)
21. Lipton, R.J.: The reachability problem requires exponential space. Department of Computer Science, Yale University **Research report**(62) (1976)
22. Marsan, M.A., Balbo, G., Conte, G., Donatelli, S., Franceschinis, G.: Modelling with Generalized Stochastic Petri Nets. ACM SIGMETRICS Performance Evaluation Review **26**(2), 2 (Aug 1998). https://doi.org/10.1145/288197.581193
23. Myers, C.J.: Engineering Genetic Circuits. Chapman & Hall/CRC Mathematical and Computational Biology, Chapman & Hall/CRC, 1 edn. (July 2009)
24. Myers, C.J.: Engineering Genetic Circuits. Chapman & Hall / CRC Mathematical and Computational Biology Series, CRC Press, 1 edn. (2010)
25. Neupane, T., Zhang, Z., Madsen, C., Zheng, H., Myers, C.J.: Approximation techniques for stochastic analysis of biological systems. In: Liò, P., Zuliani, P. (eds.) Automated Reasoning for Systems Biology and Medicine, vol. 30, chap. 12, pp. 327–348. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-17297-8_12
26. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley Series in Probability and Statistics, John Wiley & Sons, Inc. (Apr 1994)
27. Roberts, R., Neupane, T., Buecherl, L., Myers, C.J., Zhang, Z.: STAMINA 2.0: Improving scalability of infinite-state stochastic model checking. In: Finkbeiner, B., Wies, T. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 319–331. Springer International Publishing, Cham (2022)
28. Sheeran, M., Singh, S., Stålmarck, G.: Checking Safety Properties Using Induction and a SAT-Solver. In: Hunt, W.A., Johnson, S.D. (eds.) Formal Methods in Computer-Aided Design. pp. 127–144. Springer, Berlin, Heidelberg (2000). https://doi.org/10.1007/3-540-40922-X_8

29. Soloveichik, D., Seelig, G., Winfree, E.: Dna as a universal substrate for chemical kinetics. Proceedings of the National Academy of Sciences **107**(12), 5393–5398 (2010). https://doi.org/10.1073/pnas.0909380107, https://www.pnas.org/doi/abs/10.1073/pnas.0909380107
30. Sutton, R., Barto, A.: Reinforcement Learning: An Introduction. IEEE Transactions on Neural Networks **9**(5), 1054–1054 (Sep 1998). https://doi.org/10.1109/TNN.1998.712192
31. Taylor, L., Israelsen, B., Zhang, Z.: Cycle and Commute: Rare-Event Probability Verification for Chemical Reaction Networks. In: Formal Methods in Computer-Aided Design. pp. 284–293. TU Wien Academic Press (Oct 2023). https://doi.org/10.34727/2023/isbn.978-3-85448-060-0_37

A Algorithms

These algorithms are omitted from the main body to improve readability. This page is intentionally blank; algorithms begin on the following page.

Algorithm 1 Recursive Dependency Graph Construction

```
Function: node.rec_build_graph
Input: vas, depth
if self.enabled then
    return Success
end if
child_init ← VasState.new(calculate child init vector)
child_targets ← calculate child targets
negative_targets ← calculate negative targets
all_targets ← combine child_targets and negative_targets
for target in all_targets do
    for trans in vas.transitions do
        if self.parents does not contain trans then
            this_child_targets ← initialize empty list
            executions ← 0
            if sign match for target and trans then
                this_child_targets.push(target)
                executions ← calculate executions
            else
                continue
            end if
            if executions > 0 then
                child ← create new GraphNode
                child.parents.push(self.transition)
                self.children.push(child)
            end if
        end if
    end for
end for
for child in self.children do
    child.rec_build_graph(vas, depth + 1)
    if !child.enabled then
        self.enabled ← false
    end if
end for
return Success
```

Algorithm 2 Slashing Abstraction

Input: $model$, $dg : DependencyGraph$
Output: $trimmed\_model : AbstractVas$
$initial\_state \leftarrow$ empty list
$transitions \leftarrow$ empty list
$dg\_transitions \leftarrow dg.all\_transitions()$
for $i = 0$ **to** $num_variables$ **do**
  $is\_used \leftarrow false$
  for each $t \in dg\_transitions$ **do**
    if $t.update\_vector[i] \neq 0 \vee t.enabled\_bounds[i] \neq 0$ **then**
      $is\_used \leftarrow true$
      break
    end if
  end for
  if is_used **then**
    $initial\_state.push(model.initial\_state[i])$
  end if
end for
for each $t \in dg\_transitions$ **do**
  $transitions.push(t)$ {Update t with new indices}
end for
$target \leftarrow model.target$ {Update $target$ with new indices}
$trimmed\_model \leftarrow \{initial\_state, transitions, target\}$
return $trimmed\_model$

B σVAS Models

B.1 Single Species Production-Degradation

```
species S0 init 1
species S1 init 40
target S1 = 80
reaction R0
    consume S0
    produce S0
    produce S1
    const 1.0
reaction R1
    consume S1
    const 0.025
```

B.2 Modified Yeast Polarization

```
species R init 50
species L init 2
species RL init 0
```

Algorithm 3 Z3 Encoding for VAS Model

Input: Model *model*, Bit-width *bits*, Context *ctx*
Output: Unroller, Initial Formula, Transition Formula, Target Formula
Initialize $init\_constraints \leftarrow []$
Initialize $state\_vars \leftarrow \{\}$
Initialize $next\_vars \leftarrow \{\}$
$init \leftarrow model.initial\_states$
$vars \leftarrow model.variable\_names$
for each variable v_i in $vars$ **do**
    Create state variable $state\_var \leftarrow BV(v_i, bits)$
    Create next variable $next\_var \leftarrow BV(v_i\_next, bits)$
    Add $state\_var$ to $state\_vars$
    Add $next\_var$ to $next\_vars$
    Add constraint $state\_var = BV(init[0].vector[i], bits)$ to $init\_constraints$
end for
$init\_formula \leftarrow$ Z3_AND($init\_constraints$)
Initialize $transition\_constraints \leftarrow []$
for each transition in $model.transitions$ **do**
    Initialize $current\_transition \leftarrow []$
    for each update u_i in transition.update_vector **do**
        $state\_var \leftarrow state\_vars[v_i]$
        $next\_var \leftarrow next\_vars[v_i]$
        if transition.enabled_bounds[i] > 0 **then**
            Add constraint $state\_var \geq BV(transition.enabled\_bounds[i], bits)$
        end if
        if $u_i > 0$ **then**
            Add constraint $next\_var = state\_var + BV(u_i, bits)$
        else if $u_i < 0$ **then**
            Add constraint $next\_var = state\_var - BV(-u_i, bits)$
        else
            Add constraint $next\_var = state\_var$
        end if
    end for
    $current\_transition\_formula \leftarrow$ Z3_AND($current\_transition$)
    Add $current\_transition\_formula$ to $transition\_constraints$
end for
$transition\_formula \leftarrow$ Z3_OR($transition\_constraints$)
$target \leftarrow model.target$
$target\_formula \leftarrow state\_vars[target.variable] = BV(target.value, bits)$
$unroller \leftarrow$ Unroller($state\_vars, next\_vars$)
return $unroller, init\_formula, transition\_formula, target\_formula$

Algorithm 4 Unroller for VAS Model

Struct: Unroller
Fields:
  state_vars: HashMap of state variables
  next_vars: HashMap of next state variables
  var_cache: HashMap for variable caching
  time_cache: Vector of time caches
Method: at_time(term, k)
  Input: Term *term*, Time k
  Get cache at time k using get_cache_at_time(k)
  Substitute *term* with cache values
  Return: Substituted term
Method: at_all_times_or(term, k)
  Input: Boolean term *term*, Time k
  Initialize empty list *terms*
for $i = 0$ **to** k **do**
    Add at_time(term, i) to *terms*
end for
  Return: Z3_OR of *terms*
Method: at_all_times_and(term, k)
  Input: Boolean term *term*, Time k
  Initialize empty list *terms*
for $i = 0$ **to** k **do**
    Add at_time(term, i) to *terms*
end for
  Return: Z3_AND of *terms*
Method: get_var(v, k)
  Input: Variable v, Time k
  Create key from v and k
if key exists in *var_cache* **then**
    Return: Cached variable
end if
  Create new variable v_k with name format "$v@k$"
  Store v_k in *var_cache*
  Return: v_k
Method: get_cache_at_time(k)
  Input: Time k
while length of *time_cache* is less than or equal to k **do**
    Initialize empty cache
    Set t to current length of *time_cache*
  for each (s, state_var) in *state_vars* **do**
      Get s_t and n_t using get_var
      Insert s_t and n_t into cache
  end for
    Add cache to *time_cache*
end while
  Return: Reference to cache at time k

Algorithm 5 Simplified Variable Bound Calculator

```
Input: BMC formula at step k
  Create hashmap bounds
for each variable state_var do
    Set min_bound to initial state value
    Set max_bound to maximum possible value
    Set bound to 0
  while true do
      Reset solver
      Create bound_formula using state_var and bound
      Combine bound_formula with existing formula
      Assert combined formula in solver
    if status is SAT then
      if bound outside max/min bounds then
          Break
      end if
        Update max/min bounds and calculate new bound
    else
      if bound ≥ max_bound then
          Decrease bound
      end if
        Update max/min bounds and calculate new bound
    end if
  end while
    Store bound in bounds[s]
end for
return bounds
```

Algorithm 6 Trace Generation in Reinforcement Learning

Function: make_traces(transitions, traces)
  Input: List of transitions, Number of traces to generate
  Initialize learning with transitions
for $i = 0$ **to** $traces - 1$ **do**
    Generate a trace using learning
    adjust_rewards(*learning*, *trace*, *trace.probability*)
end for
Function: generate_trace(*learning*)
  Input: Learning object
  Initialize a simulator to s_0
while true **do**
  Randomly pick an enabled transition, favoring higher rewards
  Update the trace probability
end while
Function: adjust_rewards(*learning*, *trace*, *probability*)
  Input: Learning object, Trace, Probability of the trace
  Calculate weighted *average* favoring recent history
  Calculate *improvement* comparing *probability* and *average*
  Calculate *length_penalty* based on historical trace lengths
  Update *improvement* by subtracting *length_penalty*
for each (*transition*, *reward*) in *learning.reward* **do**
    Count occurrences of *transition* in *trace*
  if count > 0 **then**
    if *improvement* > 0 **then**
      Enhance *reward* based on *improvement* and *count*
    else
      Penalize *reward* based on *improvement* and *count*
    end if
  end if
  if *transition* is in dependency graph **then**
    Increase *reward* slightly {Keep incentivizing necessary transitions}
  end if
end for
  Add trace probability to learning history
  Normalize rewards if total reward exceeds maximum threshold
  Calculate mean and standard deviation of rewards
  Adjust rewards based on their deviation from the mean

```
species G init 50
species GA init 0
species GBG init 0
species GD init 0
target GBG = 50
reaction R1
    produce R 1
    const 0.0038
reaction R2
    consume R 1
    const 0.0004
reaction R3
    consume R 1
    consume L 1
    produce RL 1
    produce L 1
    const 0.042
reaction R4
    consume RL 1
    produce R 1
    const 0.010
reaction R5
    consume RL 1
    consume G 1
    produce GA 1
    produce GBG 1
    const 0.011
reaction R6
    consume GA 1
    produce GD 1
    const 0.100
reaction R7
    consume GBG 1
    consume GD 1
    produce G 1
    const 1050
reaction R8
    produce RL 1
    const 3.210
```

B.3 Enzymatic Futile Cycle

```
species S0 init 1
species S1 init 50
species S2 init 0
species S3 init 1
species S4 init 50
species S5 init 0
```

```
target S4 = 25
reaction R0
    consume S0
    consume S1
    produce S2
    const 1.0
reaction R1
    consume S2
    produce S1
    produce S0
    const 1.0
reaction R2
    consume S2
    produce S0
    produce S4
    const 0.1
reaction R3
    consume S3
    consume S4
    produce S5
    const 1.0
reaction R4
    consume S5
    produce S3
    produce S4
    const 1.0
reaction R5
    consume S5
    produce S1
    produce S3
    const 0.1
```

B.4 Simplified Motility Regulation

```
species S0 init 1
species S1 init 10
species S2 init 1
species S3 init 10
species S4 init 1
species S5 init 1
species S6 init 10
species S7 init 1
species S8 init 1
target S1 = 20
reaction R0
    consume S0
    produce S0
    consume S1
```

```
    const 0.1
reaction R1
    consume S1
    const 0.0002
reaction R2
    consume S2
    produce S2
    produce S3
    const 1.0
reaction R3
    consume S3
    const 0.002
reaction R4
    consume S4
    produce S3
    produce S5
    produce S6
    const 1.0
reaction R5
    consume S6
    const 0.0002
reaction R6
    consume S3
    consume S5
    produce S4
    const 0.01
reaction R7
    consume S4
    produce S4
    produce S5
    const 0.1
reaction R8
    consume S1
    consume S2
    produce S7
    const 0.02
reaction R9
    consume S7
    produce S1
    produce S0
    const 0.1
reaction R10
    consume S1
    consume S5
    produce S8
    const 0.01
reaction R11
    consume S8
    produce S1
    produce S5
```

```
65      const 0.1
```