Quantitative Static Analysis of Communication Protocols Using Abstract Markov Chains

Abdelraouf Ouadjaout, Antoine Miné

Quantitative Static Analysis of Communication Protocols using Abstract Markov Chains[⋆]

Abdelraouf Ouadjaout and Antoine Miné

Sorbonne Universités, UPMC, LIP6, Paris, France
{abdelraouf.ouadjaout,antoine.mine}@lip6.fr

Abstract. In this paper we present a static analysis of communication protocols for inferring parametric bounds of performance metrics. Our analysis is formalized within the theory of abstract interpretation and soundly takes all possible executions into account. We model the concrete executions as Markov chains and we introduce a novel notion of *Abstract Markov Chains* that provides a finite and symbolic representation to over-approximate the (possibly unbounded) set of concrete behaviors. Our analysis operates in two steps. The first step is a classic abstract interpretation of the source code, using stock numerical abstract domains and a specific automata domain, in order to extract the abstract Markov chain of the program. The second step extracts from this chain particular invariants about the stationary distribution and computes its symbolic bounds using a parametric Fourier-Motzkin elimination algorithm. We present a prototype implementation of the analysis and we discuss some preliminary experiments on a number of communication protocols.

1 Introduction

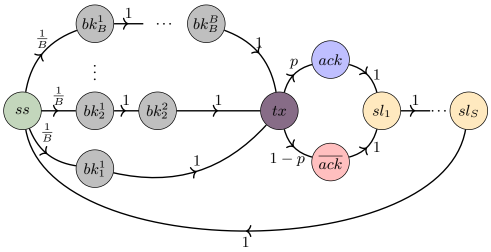
The analysis of probabilistic programs represents a challenging problem. The difficulty comes from the fact that execution traces are characterized by probability distributions that are affected by the behavior of the program, resulting in very complex forms of stochastic processes. In addition, in such particular context, programmers are interested in quantitative properties not supported by conventional semantics analysis, such as the inference of expected values of performance metrics or the probability of reaching bug states. In this work, we focus on the analysis of communication protocols and we aim at assessing their performance formally.

Stationary Distribution. Generally, the quantification of performance metrics for such systems is based on computing the stationary distribution of the associated random process. It gives the proportion of time spent in every reachable state of the system by considering all possible executions. This information is fundamental to compute the expected value of most common performance metrics. For instance, the throughput represents the average number of transmitted packets per time unit. By identifying the program locations where packets are transmitted and by computing the value of the

[⋆] This work is partially supported by the European Research Council under Consolidator Grant Agreement 681393 – MOPSA.

```
int n = 0, a = 0;
while(1) {
  data = sense();
  //Uniform backoff
  sleep(uniform(1, B));
  //Transmission with ack
  if (unicast(data)) a++;
  n++;
  //Save energy
  sleep(S);
}
```

(a)



(b)

Fig. 1: (a) Backoff-based transmission. (b) Associated discrete time Markov chain.

stationary distribution at these locations, we obtain therefore the proportion of packets sent in one time unit. Similarly, many other metrics are based on this distribution, such as the duty cycle (proportion of time where the transceiver is activated) or the goodput (the proportion of successfully transmitted data).

To our knowledge, no existing approach can obtain such information (*i*) *automatically* by analyzing the source code, (*ii*) *soundly* by considering all executions in possibly infinite systems and (*iii*) *symbolically* by expressing the distribution in terms of the protocol parameters. Indeed, while most proposed solutions focus on computing probabilities of program assertions [26,4] or expectation invariants [5,1], only PRISM [16], thanks to its extension PARAM [14], can compute stationary distributions of parametric Markov chains, but is limited to finite state systems with parametric transition probabilities only, while we also support systems where the number of states is a (possibly unbounded) parameter.

Example 1. We illustrate our motivation with a simple wireless protocol shown in Fig. 1(a). This example illustrates a basic embedded application in which a set of sensing devices transmit periodically their readings to a remote central station. To derive the goodput Γ of a sensor, we model the protocol as a discrete time Markov chain as shown in Fig. 1(b). The program begins by acquiring the sensor measurements by calling the function `sense`. This operation corresponds to the state ss in which the chain remains one time tick. To avoid collisions when sending the data, a random backoff is performed using a uniform distribution on the range $[1, B]$, where B is a parameter of the protocol. This is modeled as a fork from state ss to B backoff levels. Each transition is labeled with probability $\frac{1}{B}$ and the chain remains i ticks at level i. An important random aspect of the system is the lossy nature of the wireless links, which is modeled as a Bernoulli distribution with parameter p. This means that at each call of the function `unicast` at state tx, the packet is transmitted and acknowledged with probability p, or lost with probability $1 - p$. Finally, before transmitting the next reading, the program executes the `sleep` statement to save energy for a duration determined by parameter S, which is modeled with the transitions $sl_1 \xrightarrow{1} \ldots \xrightarrow{1} sl_S$.

The goodput Γ of the protocol is the proportion of time spent in state ack, which can be obtained by computing the stationary distribution $\boldsymbol{\pi}$ of the chain. To do so, we first construct the stochastic matrix P where its entries correspond to the probabilities of the chain's transitions. After that, we compute the vector $\boldsymbol{\pi}$ as the eigenvector of the stochastic matrix P associated to the eigenvalue 1. Since the structure and the size of the matrix depend on the parameters B and S, existing solutions can not derive automatically the stationary distribution symbolically in terms of B, S and p. □

Contributions. We propose a solution for this problem based on two main contributions:

1. First, we introduce a novel notion of *Abstract Markov Chains* that approximates a family of discrete time Markov chains. These abstract chains are inferred automatically by analyzing the source code of the program. Thanks to a novel widening algorithm, these chains are guaranteed to have a finite size while covering all possible probabilistic traces of the program.
2. Our second contribution is a result for extracting *distribution invariants* from an abstract Markov chain in the form of a system of parametric linear inequalities for bounding the concrete stationary distribution. Using a parametric-version of the Fourier-Motzkin elimination algorithm, we can infer symbolic and guaranteed bounds of the property of interest.

Example 2. By applying our analysis on the previous example, we can infer that:

$$\frac{B^2(p-1) - B(p-3) + 2(p-1)}{3B^2 + 2BS + B + 4} \leq \Gamma \leq \frac{\left(B^2 - B + 2\right)p}{3B^2 + 2BS + B + 4} \tag{1}$$

System designers can use this invariant to find appropriate parameter values that ensure certain performance constraints. For instance, assume that we know that the deployment zone is characterized by a link quality varying in $[0.7, 0.9]$ and we want to figure out which parameter configuration guarantees that Γ always fit within $[1, 5]$ packets/s (with the assumption that a time tick is 1ms). Using (1), we can show that the instance $\langle B \mapsto 4, S \mapsto 308\rangle$ produces a chain that always verifies these constraints. □

Limitations. Our approach is still in a preliminary development phase and presents some limitations. The analysis supports only discrete probability distributions, such as Bernoulli and discrete uniform distributions. Secondly, we limit the description herein to a simple C-like language and we do not support yet the analysis of real-world implementations. Finally, we do not consider pure non-deterministic statements.

Outline. The remaining of the paper is organized as follows. We present in Section 2 the concrete semantics of the analysis. Section 3 introduces the domain of Abstract Markov Chains and we detail in Section 4 the method to extract the stationary distribution invariants from an abstract chain and how we can infer symbolic bounds of the property of interest. The results of the preliminary experiments are presented in Section 5. We discuss the related work in Section 6 and we conclude the paper in Section 7.

2 Concrete Semantics

We consider communication protocols that can be represented as (possibly infinite) discrete time Markov chains, since it is one of the most widespread stochastic models for performance evaluation used by the networking community. For describing these protocols, we use a simplified probabilistic language having the following C-like syntax:

$$
\begin{array}{rll}
\mathit{Stmt} ::= & x = e; & \lbag x \in \mathcal{V}, e \in \mathit{Exp} \rbag \\
\mid & \texttt{if}(e \bowtie 0)\,\{\,s_1\,\}\,\{\,s_2\,\} & \lbag s_1, s_2 \in \mathit{Stmt}, \bowtie \in \{=, \neq, \leq, <, \geq, >\} \rbag \\
\mid & \texttt{while}(e \bowtie 0)\,\{\,s\,\} & \\
\mid & x = \texttt{uniform}_l(e_1, e_2) & \lbag e_1, e_2 \in \mathit{Exp}, l \in \mathcal{L} \rbag \\
\mid & x = \texttt{bernoulli}_l() & \\
\mid & \texttt{ticks}_l(e) &
\end{array}
$$

where $\mathcal{V}$ is the set of program variables, $\mathcal{L}$ is the set of program locations and Exp is the set of (non-probabilistic) numeric expressions the syntax of which is classic and omitted here. In addition to the common statements of assignments, **`if`** conditionals and **`while`** loops, we consider the following additional markovian statements. The function $\texttt{uniform}_l(e_1, e_2)$ draws a random integer value from a discrete uniform distribution over the interval $[e_1, e_2]$, while the function $\texttt{bernoulli}_l()$ returns a boolean value according to a Bernoulli distribution with parameter p_l. Finally, the function $\texttt{ticks}_l(e)$ models the fact that the program will spend e ticks in the current control location, which results in triggering a transition in the Markov chain of the program. Each of these functions is annotated with the call site location l. Using these primitive functions, we can define any markovian behavior. Since we are interested in communication protocols, we defined a number of auxiliary functions based on these primitives, such as the functions $\texttt{unicast}()$ and $\texttt{sleep}()$ presented previously.

2.1 Markovian Traces

We develop a particular stochastic semantics that is isomorphic to a discrete time Markov chain. At the bottom level of this semantics, we have the notion of random events Ξ representing the outcomes of the probability distributions generated during program execution. We can distinguish between two types of random events. The events b_l and $\overline{b}_l$ denote the two outcomes of a statement $\texttt{bernoulli}_l()$. Also, the outcomes of the statement $\texttt{uniform}_l(e_1, e_2)$ are given by the set $\{u_l^{i,a,b} \mid i \in [a, b]\}$, where a and b are the evaluation in the current execution environment of e_1 and e_2 respectively.

Naively, we can consider a Markov chain as a classic automaton over the alphabet Ξ recognizing the probabilistic traces of the program as sequences of random events. However, Markov chains are not just a set of probabilistic traces, but embed a notion of time that is fundamental. Indeed, transitions in a Markov chain occur solely when at least one time tick has elapsed, since a state of the chain can not have a null sojourn time. As we consider that only the $\texttt{ticks}(e)$ statement advances time, some of the program transitions become non-observable at the time scale of the chain. This leads to a *two-level trace semantics* making the distinction between observable and non-observable transitions, which has been introduced by Radhia Cousot in her thesis [9, Section 2.5.4]. We give here a definition of these two types of traces adapted to our settings:

Definition 1 (Scenarios). *A sequence of non-observable transitions is called a scenario and is defined as $\omega \in \Omega \triangleq \Xi^*$ expressing sequences of random events that occur between two observable states. In the sequel, we denote by ε the empty scenario word.*

$$
\begin{aligned}
&\mathbf{S}\llbracket x = e \rrbracket R && = \{(\tau, \rho[x \mapsto v], \omega) \mid (\tau, \rho, \omega) \in R \wedge v \in \mathbf{E}\llbracket e \rrbracket\{\rho\}\} \\
&\mathbf{S}\llbracket \texttt{if}(e \bowtie 0)\,\{\, s_1 \,\}\,\{\, s_2 \,\} \rrbracket R && = (\mathbf{S}\llbracket s_1 \rrbracket \circ \mathbf{S}\llbracket (e \bowtie 0) \rrbracket R) \cup (\mathbf{S}\llbracket s_2 \rrbracket \circ \mathbf{S}\llbracket (e \not\bowtie 0) \rrbracket R) \\
&\mathbf{S}\llbracket \texttt{while}(e \bowtie 0)\,\{\, s \,\} \rrbracket R && = \mathbf{S}\llbracket (e \not\bowtie 0) \rrbracket (\texttt{lfp}\, \lambda X.\, R \cup \mathbf{S}\llbracket s \rrbracket \circ \mathbf{S}\llbracket (e \bowtie 0) \rrbracket X) \\
&\mathbf{S}\llbracket (e \bowtie 0) \rrbracket R && = \{(\tau, \rho, \omega) \in R \mid \exists v \in \mathbf{E}\llbracket e \rrbracket\{\rho\} : v \bowtie 0\} \\
&\mathbf{S}\llbracket \texttt{ticks}_l(e) \rrbracket R && = \{(\tau \overset{\omega}{\to} (l, \rho, \nu), \rho, \varepsilon) \mid (\tau, \rho, \omega) \in R \wedge \nu \in \mathbf{E}\llbracket e \rrbracket\{\rho\}\} \\
&\mathbf{S}\llbracket x = \texttt{bernoulli}_l() \rrbracket R && = \{(\tau, \rho[x \mapsto b], \omega.\xi) \mid (\tau, \rho, \omega) \in R \wedge (b, \xi) \in \{(1, b_l), (0, \overline{b}_l)\}\} \\
&\mathbf{S}\llbracket x = \texttt{uniform}_l(e_1, e_2) \rrbracket R && = \{(\tau, \rho[x \mapsto i], \omega.u_l^{i,a,b}) \mid (\tau, \rho, \omega) \in R \wedge a \in \mathbf{E}\llbracket e_1 \rrbracket\{\rho\} \wedge \\
& && \qquad b \in \mathbf{E}\llbracket e_2 \rrbracket\{\rho\} \wedge i \in [a, b]\}
\end{aligned}
$$

Fig. 2: Concrete transfer functions.

Definition 2 (Markovian traces). *The observable markovian traces are the set* $\mathcal{T}_\Sigma^\Omega \triangleq \{\sigma_0 \overset{\omega_1}{\to} \sigma_1 \overset{\omega_2}{\to} \dots \mid \sigma_i \in \Sigma \wedge \omega_i \in \Omega\}$ *of transitions among observable states labeled with scenarios. An observable state is a tuple* $(l, \rho, \nu) \in \Sigma \triangleq \mathcal{L} \times \mathcal{E} \times \mathbb{N}$ *where (i)* $l \in \mathcal{L}$ *is a program location, (ii)* $\rho \in \mathcal{E} \triangleq \mathcal{V} \to \mathbb{Z}$ *is a program environment and (iii)* $\nu \in \mathbb{N}$ *is a sojourn time representing the number of ticks spent in that state.*

This notion of markovian traces is a set-based representation of Markov chains that fits well within the framework of abstract interpretation. It allows a fluent extension of the classical trace semantics for supporting the particular stochastic and temporal features of discrete time Markov chains. In the following paragraph, we define this semantics domain and we present the most important transfer functions.

2.2 Semantics Domain

The concrete semantics domain of our analysis is defined as $\mathcal{D} \triangleq \wp(\mathcal{T}_\Sigma^\Omega \times \mathcal{E} \times \Omega)$. An element $(\tau, \rho, \omega) \in \mathcal{T}_\Sigma^\Omega \times \mathcal{E} \times \Omega$ encodes the set of traces reaching a given program location and is composed of three parts: (*i*) the observable trace $\tau \in \mathcal{T}_\Sigma^\Omega$ containing the past markovian transitions before the current time tick, (*ii*) the current memory environment $\rho \in \mathcal{E}$, and (*iii*) the partial scenario $\omega \in \Omega$ of non-observable random events that occurred between the last tick and the current execution moment.

To obtain the set of all traces of a program P, we proceed by induction on its abstract syntax tree using a set of concrete transfer functions $\mathbf{S}\llbracket . \rrbracket \in \mathcal{D} \to \mathcal{D}$. We give in Fig. 2 a summary of these functions. We assume given (in a standard way) the function $\mathbf{E}\llbracket e \rrbracket \in \wp(\mathcal{E}) \to \wp(\mathcal{E})$ that provides the possible evaluations of an expression in a set of environments. Non-probabilistic statements have a standard definition. The assignment statement updates the current memory environment by mapping the left-hand variable to the evaluation of the expression. For the `if` assignment, we filter the current environments depending on the evaluation of the condition, and we analyze each branch independently before merging the results. Also, a loop statement is formalized as a fixpoint on the sequences of body evaluation with a filter to extract the iterations violating the loop condition.

The semantics of the statement $x = \texttt{bernoulli}_l()$ is to fork the current partial scenarios ω depending on the result of the function. We append the event b_l in the true case, or the event $\overline{b}_l$ in the false case and we update the variable x with the returned value in the current memory environment. For the statement $x = \texttt{uniform}_l(e_1, e_2)$, we also fork the partial scenarios and update the variable x accordingly, but the difference is that the number of branches depends on the evaluations of e_1 and e_2 in the current memory environment. More precisely, the number of forks corresponds to the number of integer points between the values of e_1 and e_2. Note that, for these two statements, the markovian traces part is not modified since they are tick-less. This is not the case for the $\texttt{ticks}_l(e)$ statement that appends the markovian traces with a new transition to a state where the sojourn time is equal to the evaluation of the expression e. The label of this new transition is simply the computed partial scenario, which is reset to the empty word ε since we keep track of events traces only between two `ticks` statements.

2.3 Stationary Distribution

After collecting the set $T \subseteq \mathcal{T}_{\Sigma}^{\Omega}$ of all possible markovian traces, we want to compute the *stationary distribution* of the associated Markov chain, which reflects the proportion of time spent in every observable state. To do that, we have first to construct a particular transition matrix $\mathbf{P}$, that differs slightly from the classic stochastic matrix of discrete time Markov chains since states in our model embed different values of sojourn time:

$$\mathbf{P}_{(l,\rho,\nu),(l',\rho',\nu')} \triangleq \frac{\nu'}{\nu} \sum_{(l,\rho,\nu) \overset{\omega}{\to} (l',\rho',\nu') \in T} \Pr(\omega) \tag{2}$$

where (l, ρ, ν) and (l', ρ', ν') are two reachable states in the traces T. The function $\Pr \in \Omega \to [0, 1]$ gives the probability of the scenarios and is computed as follows:

$$\begin{cases} \Pr(\varepsilon) \triangleq 1, \Pr(b_l) \triangleq p_l, \Pr(\bar{b}_l) \triangleq 1 - p_l, \Pr(u_l^{i,a,b}) \triangleq \frac{1}{b-a+1} \\ \Pr(\omega\xi) \triangleq \Pr(\omega)\Pr(\xi) \end{cases} \tag{3}$$

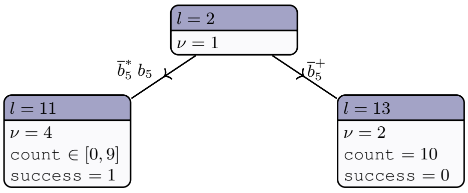
Finally, as for the classic matrix, the stationary distribution of the chain represents the eigenvector $\boldsymbol{\pi}$ of $\mathbf{P}$ associated to the eigenvalue 1, which is obtained by solving the system $\boldsymbol{\pi} = \boldsymbol{\pi}\mathbf{P}$ with the additional normalization constraint $\sum_{(l,\rho,\nu)} \boldsymbol{\pi}_{(l,\rho,\nu)} = 1$. Since the size of $\mathbf{P}$ depends on the size of the reachable states space, $\boldsymbol{\pi}$ can not be computed automatically in general. In the following, we propose a computable abstraction of Markov chains to over-approximate the traces T. Afterwards, we show how we can infer guaranteed bounds of $\boldsymbol{\pi}$ using information provided by our abstract chain.

3 Abstract Semantics

In order to analyze a program statically, we need a computable abstraction of the concrete semantics domain $\mathcal{D}$. The basic idea is to first partition the set of observable program states $\mathcal{L} \times \mathcal{E} \times \mathbb{N}$ with respect to the program locations, resulting into the intermediate abstraction $\mathcal{L} \times \wp(\mathcal{E} \times \mathbb{N})$. For each location, the set of associated environments is then abstracted with a stock numerical domain $\mathfrak{E}^\sharp$, by considering the sojourn time as a program variable ν. We obtain the abstract states domain $\Sigma^\sharp \triangleq \mathcal{L} \times \mathfrak{E}^\sharp$. As a consequence of this partitioning, observable states at the same program location will be merged. Therefore, we obtain a special structure in which observable abstract states are connected through possibly multiple scenarios coming from the merged concrete states.

```
1  void* sense() {
2    warmup(); //~ticks(1);
3    int count = 0, success = 0;
4    do {
5      if (check()) //~bernoulli()
6        success = 1;
7      else
8        count++;
9    } while (count < 10 && !success);
10   if (success)
11     return doDetection(); //~ticks(4)
12   else
13     return doAbsence(); //~ticks(2)
14 }
```

(a)



(b)

Fig. 3: (a) A simple probabilistic model for the `sense()` function. (b) An abstraction of observable traces represented as a hierarchical automaton.

Example 3. We illustrate this fact in Fig. 3(a) depicting a more complex probabilistic modeling of the previous `sense()` function using a bounded geometric distribution that works as follows. We start by warming up the sensing device during one tick. After that, we check whether the sensor detects some external activity (high temperature, sound noise, etc.) and we perform this check for at most 10 times. We assume that these external activities follow a Bernoulli distribution. At the end, we perform some processing during 4 ticks in case of detection and 2 ticks in case of non-detection.

We can see in Fig. 3(b) that between the observable program locations 2 and 11 many scenarios are possible, which are abstracted with the regular expression $\overline{b}_5^* \, b_5$ that encodes the pattern of having a number of Bernoulli failure outcomes at line 5 before a successful one. However, between lines 2 and 13, we can have only a sequence of failures, which is expressed as $\overline{b}_5^+$. □

The presence of these multi-words transitions leads to a *hierarchical automata* structure organized in two levels. On the one hand, one automata structure is used to encode the transitions between observable abstract states. On the other hand, and for each observable transition, another automata structure is used to encode the regular expressions of scenarios connecting the endpoints of the transition. In other words, we abstract markovian traces with an automaton, the transitions of which have also an automata structure representing a set of scenarios. For modularity reasons, we present however a single generic automata domain to represent regular languages over any abstract alphabet. Afterwards, we instantiate two automata-based domains for abstracting events words and markovian traces.

3.1 Abstract Automata

Le Gall et al. proposed a lattice automata domain [17] to represent words over an abstract alphabet having a lattice structure. We extend this domain to support also abstraction at the state level by merging states into abstract states, which is important to approximate markovian traces. To do so, we define a functor domain $\mathscr{A}(\mathfrak{A}^\sharp, \mathfrak{S}^\sharp)$ parameterized by an abstract alphabet domain $\mathfrak{A}^\sharp$ and an abstract state domain $\mathfrak{S}^\sharp$:

Definition 3 (Abstract automata). *An abstract automaton $A \in \mathscr{A}(\mathfrak{A}^\sharp, \mathfrak{S}^\sharp)$ is a tuple $A = (S, s_0^\sharp, F, \Delta)$, where $S \subseteq \mathfrak{S}^\sharp$ is the set of states, $s_0^\sharp \in S$ is the initial state, $F \subseteq S$ is the set of final states and $\Delta \subseteq S \times \mathfrak{A}^\sharp \times S$ is the transition relation.*

We assume that the parameter domain $\mathfrak{A}^\sharp$ is an abstraction of some concrete alphabet symbols $\mathcal{A}$, having a concretization function $\gamma_\mathfrak{A} \in \mathfrak{A}^\sharp \to \wp(\mathcal{A})$, a partial order $\sqsubseteq_\mathfrak{A}$, a join operator $\sqcup_\mathfrak{A}$, a meet operator $\sqcap_\mathfrak{A}$, a least element $\bot_\mathfrak{A}$ and a widening operator $\triangledown_\mathfrak{A}$. The second parameter domain $\mathfrak{S}^\sharp$ is assumed to be an abstraction of some concrete states $\mathcal{S}$ equipped with a concretization function $\gamma_\mathfrak{S} \in \mathfrak{S}^\sharp \to \wp(\mathcal{S})$, a partial order $\sqsubseteq_\mathfrak{S}$, a join operator $\sqcup_\mathfrak{S}$, a least element $\bot_\mathfrak{S}$ and a widening operator $\triangledown_\mathfrak{S}$.

Let us define some important operators for the $\mathscr{A}$ functor domain. In the following, we denote by $A = (S, s_0^\sharp, F, \Delta)$, $A_1 = (S_1, s_{0_1}^\sharp, F_1, \Delta_1)$ and $A_2 = (S_2, s_{0_2}^\sharp, F_2, \Delta_2)$ three instances of $\mathscr{A}\left(\mathfrak{A}^\sharp, \mathfrak{S}^\sharp\right)$. We also define the auxiliary functions $\mathbf{L} \in \mathscr{A}\left(\mathfrak{A}^\sharp, \mathfrak{S}^\sharp\right) \to \wp\left(\mathfrak{A}^{\sharp\star}\right)$ and $\mathbf{T} \in \mathscr{A}(\mathfrak{A}^\sharp, \mathfrak{S}^\sharp) \to \wp\left(\mathcal{T}_{\mathfrak{S}^\sharp}^{\mathfrak{A}^\sharp}\right)$ giving respectively the set of accepted abstract words and abstract traces.

Definition 4 (Concretization). *The sets of concrete words and traces abstracted by an abstract automaton A are given by:*

$$\begin{cases} \gamma_\mathscr{A}^\mathbf{L}(A) = \{a_1 a_2 \dots \mid \exists a_1^\sharp a_2^\sharp \dots \in \mathbf{L}(A), \forall i : a_i \in \gamma_\mathfrak{A}(a_i^\sharp)\} \\ \gamma_\mathscr{A}^\mathbf{T}(A) = \{s_1 \xrightarrow{a_1} \dots \mid \exists s_1^\sharp \xrightarrow{a_1^\sharp} \dots \in \mathbf{T}(A), \forall i : s_i \in \gamma_\mathfrak{S}(s_i^\sharp) \wedge a_i \in \gamma_\mathfrak{A}(a_i^\sharp)\} \end{cases} \tag{4}$$

Order. To compare two abstract automata, we define the following simulation relation that extends the classical simulation concept found in transition systems by considering the abstraction in the alphabet and states:

Definition 5 (Simulation relation). *A binary relation $\mathcal{R} \subseteq \mathfrak{S}^\sharp \times \mathfrak{S}^\sharp$ is a simulation between A_1 and A_2 iff $\forall (s_1^\sharp, s_2^\sharp) \in \mathcal{R}$ we have $s_1^\sharp \sqsubseteq_\mathfrak{S} s_2^\sharp$ and:*

$$\forall s_1^\sharp \xrightarrow{a_1^\sharp} q_1^\sharp \in \Delta_1, \exists s_2^\sharp \xrightarrow{a_2^\sharp} q_2^\sharp \in \Delta_2 : a_1^\sharp \sqsubseteq_\mathfrak{A} a_2^\sharp \wedge q_1^\sharp \mathcal{R} q_2^\sharp \tag{5}$$

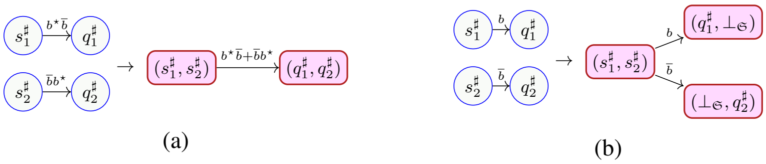
We denote $\preccurlyeq$ the smallest simulation relation between A_1 and A_2 verifying $s_{0_1}^\sharp \preccurlyeq s_{0_2}^\sharp$.

Using this notion we define the partial order relation $\sqsubseteq_\mathscr{A}$ as:

$$A_1 \sqsubseteq_\mathscr{A} A_2 \Leftrightarrow \forall (s_1^\sharp, s_2^\sharp) \in \preccurlyeq : s_1^\sharp \in F_1 \Rightarrow s_2^\sharp \in F_2 \tag{6}$$

which means that A_2 should simulate and accept every accepted trace in A_1.

Join. To compute the union of two abstract automata A_1 and A_2, we need to extend the simulation-based traversal in a way to include traces contained in one automaton only, which is formalized with the following concept of *product relation*. The intuition behind it is depicted in Fig. 4 in which we consider transitions decorated with an illustrative regular language over an alphabet $\{b, \bar{b}\}$. In Fig. 4(a), the input transitions $s_1^\sharp \xrightarrow{b^\star \bar{b}} q_1^\sharp$ and $s_2^\sharp \xrightarrow{\bar{b} b^\star} q_2^\sharp$ are combined into a single product transition that accepts the merged alphabet symbol $b^\star \bar{b} + \bar{b} b^\star$. While proceeding similarly for all cases preserves the soundness of



$s_1^\sharp$
$b^\star\bar{b}$
$q_1^\sharp$
$s_2^\sharp$
$\bar{b}b^\star$
$q_2^\sharp$
$\to$
$(s_1^\sharp, s_2^\sharp)$
$b^\star\bar{b}+\bar{b}b^\star$
$(q_1^\sharp, q_2^\sharp)$
$s_1^\sharp$
b
$q_1^\sharp$
$s_2^\sharp$
$\bar{b}$
$q_2^\sharp$
$\to$
$(s_1^\sharp, s_2^\sharp)$
b
$(q_1^\sharp, \bot_{\mathfrak{S}})$
$\bar{b}$
$(\bot_{\mathfrak{S}}, q_2^\sharp)$

(a)

(b)

Fig. 4: Cases of construction of a product transition.

the operator, we can gain in precision by separating singular transitions as shown in Fig. 4(b). In this case, no intersection exists between the transitions $s_1^\sharp \xrightarrow{b} q_1^\sharp$ and $s_2^\sharp \xrightarrow{\bar{b}} q_2^\sharp$. This means that the automata A_1 and A_2 can not perform a simultaneous transition at $s_1^\sharp$ and $s_2^\sharp$, which is expressed as two singular transitions to $(q_1^\sharp, \bot_{\mathfrak{S}})$ and $(\bot_{\mathfrak{S}}, q_2^\sharp)$.

Note that comparing alphabet symbols is not the only means to detect singular transitions. Indeed, in some situations, destination states $q_1^\sharp$ and $q_2^\sharp$ should be kept separated in order for the analysis to preserve some of its precision. To illustrate this point, let us consider the computation of the goodput of a protocol. In order to obtain a precise quantification of this metric, it is necessary to avoid merging states encapsulating different situations of packet transmission status (reception, loss). To do so, we assume that the abstract states domain $\mathfrak{A}^\sharp$ is provided with some equivalence relation $\equiv_{\mathfrak{S}}$ that partitions the states into a finite set of equivalence classes depending on the property of interest. Using this information, we define our product relation as follows:

Definition 6 (Product relation). *A binary relation $\mathcal{R} \subseteq \mathfrak{S} \times \mathfrak{S}$ is a product of A_1 and A_2 iff $\forall (s_1^\sharp, s_2^\sharp) \in \mathcal{R}$ we have $s_1^\sharp \equiv_{\mathfrak{S}} s_2^\sharp$ and:*

$$\begin{cases} q_1^\sharp \mathcal{R} q_2^\sharp \quad \textit{if}\, \exists s_1^\sharp \xrightarrow{a_1^\sharp} q_1^\sharp \in \Delta_1, \exists s_2^\sharp \xrightarrow{a_2^\sharp} q_2^\sharp \in \Delta_2 : a_1^\sharp \sqcap_{\mathfrak{A}} a_2^\sharp \neq \bot_{\mathfrak{A}} \wedge q_1^\sharp \equiv_{\mathfrak{S}} q_2 \\ q_1^\sharp \mathcal{R} \bot_{\mathfrak{S}} \;\textit{if}\, \exists s_1^\sharp \xrightarrow{a_1^\sharp} q_1^\sharp \in \Delta_1, \forall s_2^\sharp \xrightarrow{a_2^\sharp} q_2^\sharp \in \Delta_2 : q_1^\sharp \not\equiv_{\mathfrak{S}} q_2^\sharp \vee a_1^\sharp \sqcap_{\mathfrak{A}} a_2^\sharp = \bot_{\mathfrak{A}} \\ \bot_{\mathfrak{S}} \mathcal{R} q_2^\sharp \;\textit{if}\, \exists s_2^\sharp \xrightarrow{a_2^\sharp} q_2^\sharp \in \Delta_2, \forall s_1^\sharp \xrightarrow{a_2^\sharp} q_1^\sharp \in \Delta_1 : q_1^\sharp \not\equiv_{\mathfrak{S}} q_2^\sharp \vee a_1^\sharp \sqcap_{\mathfrak{A}} a_2^\sharp = \bot_{\mathfrak{A}} \end{cases} \tag{7}$$

with the convention that $s^\sharp \equiv_{\mathfrak{S}} \bot_{\mathfrak{S}}, \forall s^\sharp \in \mathfrak{S}^\sharp$. The smallest product relation containing $(s_{0_1}^\sharp, s_{0_2}^\sharp)$ is denoted $⧖$.

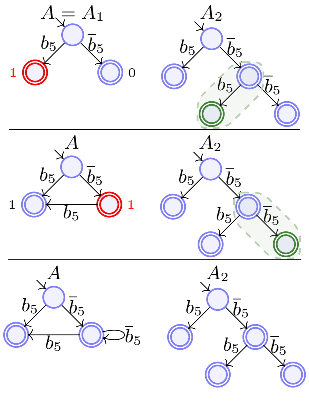
Consequently, to derive the join automaton A, we simply map product state $s_1^\sharp ⧖ s_2^\sharp$ to $s_1^\sharp \sqcup_{\mathfrak{S}} s_2^\sharp$. The final states are the subset of these images where at least $s_1^\sharp$ or $s_2^\sharp$ is final.

Append. We introduce also the append operator $\odot_\phi \in \mathscr{A}(\mathfrak{A}^\sharp, \mathfrak{S}^\sharp) \times \mathfrak{A}^\sharp \to \mathscr{A}(\mathfrak{A}^\sharp, \mathfrak{S}^\sharp)$ that extends an abstract automaton with a set of new leave transitions labeled with a given abstract alphabet symbol. From every final state $s_i^\sharp \in F$, a new edge is created to a new final state, computed as the image of $s_i^\sharp$ through the transfer function $\phi \in \mathfrak{S}^\sharp \to \mathfrak{S}^\sharp$ that annotates the operator $\odot_\phi$. This operator can be formulated as follows:

$$A \odot_\phi a^\sharp \triangleq \texttt{let}\; F' = \{\phi(s^\sharp) \mid s^\sharp \in F\} \;\texttt{in}\; (S \cup F', s_0^\sharp, F', \Delta \cup \{s^\sharp \xrightarrow{a^\sharp} \phi(s^\sharp) \mid s^\sharp \in F\}) \tag{8}$$

```
Input    : Two automata $A_1$ and $A_2$
$A = (S, s_0^\sharp, F, \Delta) \leftarrow A_1$;
≀ Find the increment transitions ∫
$\delta \leftarrow$ increments$(A, A_2)$;
repeat
    $(s_1^\sharp, s_2^\sharp \overset{a_2^\sharp}{\rightarrow} q_2^\sharp) \leftarrow$ head$(\delta)$;
    $q_\equiv^\sharp \leftarrow q_2^\sharp$;
    ≀ Search in $A$ for better candidates ∫
    $Q_\equiv \leftarrow \{s^\sharp \in S \mid s^\sharp \equiv_{\mathfrak{S}} q_2^\sharp\}$;
    if $Q_\equiv \neq \varnothing$ then
        $q_\equiv \leftarrow$ head $\circ$ sort$(\mathcal{I}_{q_2^\sharp}^{A,A_2}, Q_\equiv)$;
    end
    $S' \leftarrow S \cup \{q_\equiv^\sharp\}$;
    $F' \leftarrow F \cup (q_\equiv^\sharp \in F_2)?\, \{q_\equiv^\sharp\} : \varnothing$;
    $\Delta' \leftarrow \Delta \cup \{s_1^\sharp \xrightarrow{a_1^\sharp \nabla_{\mathfrak{A}} a_2^\sharp} q_\equiv^\sharp \mid s_1^\sharp \overset{a_1^\sharp}{\rightarrow} q_\equiv^\sharp \in \Delta \vee a_1^\sharp = a_2^\sharp\}$;
    $A \leftarrow (S', s_0^\sharp, F', \Delta')$;
    $\delta \leftarrow$ increments$(A, A_2)$;
until $\delta \neq \varnothing$;
```

(a)



(b)

Fig. 5: (a) Structural widening algorithm. (b) Result of $(b_5 + \overline{b}_5) \nabla_{\mathscr{A}} (b_5 + \overline{b}_5(b_5 + \overline{b}_5)^?)$.

Widening. Finally, we present a widening operator to avoid growing an automaton indefinitely during loop iterations. The original lattice automata domain [17] proposed a widening operator, inspired from [29,11], that employs a bisimulation-based minimization to merge similar states by comparing their transitions at some given depth. However, it assumes that the abstract alphabet domain is provided with an equivalence relation that partitions the symbols into a finite set of equivalence classes. We believe that it is more meaningful to perform this partitioning on the abstract states as explained earlier for the computation of the product relation. Therefore, we employ a different approach inspired from graph widening [28,18,30]. Basically, we compare the result of successive loop iterations and we try to detect the increment transitions to extrapolate them by creating cycles. However, existing graph widening is limited to finite alphabets and may not ensure the convergence on ascending chains, so we propose an extension to alleviate these shortcomings.

The proposed algorithm is executed in two phases. Firstly, we perform a *structural widening* to extrapolate the language recognized by the input automata and we ignore for the moment the abstract states. We show in Fig. 5(a) the main steps of this widening. Assume that A_1 and A_2 are the results of two successive iterations. Without loss of generality, we assume that $A_1 \sqsubseteq_{\mathscr{A}} A_2$. First, we compare A_1 and A_2 in order to extract the increment transitions using the following function:

$$\texttt{increments}(A_1, A_2) = \{(s_1^\sharp, s_2^\sharp \overset{a_2^\sharp}{\rightarrow} q_2^\sharp) \mid s_1^\sharp \not\asymp s_2^\sharp \wedge s_1^\sharp, s_2^\sharp \neq \bot_{\mathfrak{S}} \wedge \\ \exists s_2^\sharp \overset{a_2^\sharp}{\rightarrow} q_2^\sharp \in \Delta_2, \forall s_1^\sharp \overset{a_1^\sharp}{\rightarrow} q_1^\sharp : q_1^\sharp \not\equiv_{\mathfrak{S}} q_2^\sharp \vee a_2^\sharp \not\sqsubseteq_{\mathfrak{A}} a_1^\sharp\}$$

Basically, an increment $(s_1^\sharp, s_2^\sharp \xrightarrow{a^\sharp} q_2^\sharp)$ means that A_1 at state $s_1^\sharp$ can not recognize the symbol $a^\sharp$ while A_2 recognizes it through a move from $s_2^\sharp$ to $q_2^\sharp$. Now, we need to extrapolate A_1 in order to recover this difference, which is done by adding the missing word suffix $a_2^\sharp$ while trying not to grow A_1 in size. The basic idea is to sort states in A_1 depending on how they compare to the missing state $q_2^\sharp$. The comparison is performed with the following *similarity index* expressing the proportion of common partial traces that a state shares with $q_2^\sharp$:

$$\mathcal{I}^{A_1,A_2}_{q_2^\sharp}(q_1^\sharp) = \left|\{a_1^\sharp \dots a_n^\sharp \in \overset{\leftrightarrow}{\mathbf{L}}_{A_2,k}(q_2^\sharp) \mid \exists {a^\sharp}'_1 \dots {a^\sharp}'_n \in \overset{\leftrightarrow}{\mathbf{L}}_{A_1,k}(q_1^\sharp), \forall i : a_i^\sharp \sqsubseteq_{\mathfrak{A}} {a^\sharp}'_i\}\right|$$

where $\overset{\leftrightarrow}{\mathbf{L}}_{A,k}(s^\sharp)$ is the set of words, of length less than k, starting from $s^\sharp$ (reachable words) or ending at $s^\sharp$ (co-reachable words), where k is a parameter of the analysis. After selecting the state $q_{\equiv}^\sharp$ with the highest similarity index, we add the missing transitions after widening the alphabet symbol if a transition already exists in A. By iterating over all increment transitions, we obtain an automata structure that does not grow indefinitely since we add new states only if no existing one is equivalent. By assuming that the number of equivalence classes of $\equiv_{\mathfrak{S}}$ is finite, the widening ensures termination.

After the structural widening, we inspect the states of the resulting automaton to extrapolate them if necessary. We simply compute the simulation relation $\preccurlyeq$ between A_2 and the widened automaton A, and we replace every state $s^\sharp \in S$ with $s^\sharp \nabla_{\mathfrak{S}} (s_1^\sharp \sqcup_{\mathfrak{S}} s_2^\sharp \sqcup_{\mathfrak{S}} \dots)$ where $s_i^\sharp \preccurlyeq s^\sharp, \forall i$.

Example 4. We show in Fig. 5(b) the result of applying this structural widening on the `do-while` loop of the previous `sense()` function of Fig. 3(a). The first two iterations of the loop produce the regular expressions $b_5 + \bar{b}_5$ and $b_5 + \bar{b}_5(b_5 + \bar{b}_5)$ respectively. The widening algorithm starts by detecting the leaf increment transition b_5 and computes the different distances to select an adequate equivalent state. By adding the new transition, we obtain the regular expression $b_5 + \bar{b}_5 b_5$. The next increment transition is labeled with the event $\bar{b}_5$ and its addition to the widened automaton produces a loop which results in the final regular expression $\bar{b}_5^* b_5$. □

3.2 Abstract Scenarios

Using the functor domain $\mathscr{A}$, we instantiate an abstract scenario domain for approximating words of random events. Two considerations are important to take into account. First, the length of the these words may depend on some variables of the program. It is clear that ignoring these relations may lead to imprecise computations of the stationary distribution. Consequently, we enrich the domain with an abstract Parikh vector [25] to count the number of occurrences of random event within accepted words. By using a relational numerical domain, such as octagons [21] or polyhedra [7], we preserve some relationships between the number of events and program variables.

The second consideration is related to the uniform distribution. As shown previously in the concrete transfer function in Fig. 2, the number of outcomes depends on the bounds provided as argument to the function `uniform`. Since these arguments are evaluated in the running environment, we can have an infinite number of outcomes at a given control location when considering all possible executions.

We perform a simplifying abstraction of the random events Ξ in order to obtain a finite size alphabet and avoid the explosion of the uniform distribution outcomes. Assume that we are analyzing the statement $x = \texttt{uniform}_l(e_1, e_2)$ in abstract environment $\rho^\sharp$. Several abstractions are possible. In this work, we choose to partition the outcomes into a fixed number U of abstract outcomes, where U is a parameter of the analysis. The first $U-1$ partitions represent the individual outcomes $\{\min(e_1+i-1, e_2) \mid i \in [1, U-1]\}$, to which we associate the abstract events $\{\mathbf{u}_l^i \mid i \in [1, U-1]\}$. For the remaining outcomes, we merge them into a single abstract event $\mathbf{u}_l^\star$.

Formally, we obtain a simple finite set of abstract events $\Xi^\sharp$ defined as $\Xi^\sharp \triangleq \{\mathbf{b}_l, \overline{\mathbf{b}}_l \mid b_l \in \Xi\} \cup \{\mathbf{u}_l^i, \mathbf{u}_l^\star \mid u_l^- \in \Xi \wedge 1 \leq i \leq U-1\}$. For the Parikh vector, we associate to every abstract event $\xi^\sharp \in \Xi^\sharp$ a counter variable $\kappa_{\xi^\sharp} \in \mathbb{N}$ that will be incremented whenever the event $\xi^\sharp$ occurs.

Therefore, we define the domain of abstract scenarios as $\Omega^\sharp \triangleq \mathscr{A}(\wp(\Xi^\sharp), \Sigma^\sharp)$ where $\Sigma^\sharp$ is our previous mapping $\mathcal{L} \to \mathfrak{E}^\sharp$ from program locations to the stock numeric abstract domain. Let us now describe how probabilistic statements affect an abstract scenario. For the $\texttt{bernoulli}_l()$ statement, we create two new transitions labeled with the abstract events $\mathbf{b}_l$ and $\overline{\mathbf{b}}_l$ respectively and we update the Parikh vector accordingly:

$$\begin{aligned}
&\mathbf{S}[\![x = \texttt{bernoulli}_l()]\!]^\sharp_\Omega \omega^\sharp \triangleq \\
&\quad \texttt{let}\, \phi_0(-, \rho^\sharp) = (l, \mathbf{S}[\![\kappa_{\overline{\mathbf{b}}_l}\texttt{++}]\!]^\sharp_{\mathfrak{E}} \circ \mathbf{S}[\![x = 0]\!]^\sharp_{\mathfrak{E}} \rho^\sharp) \\
&\quad \texttt{and}\, \phi_1(-, \rho^\sharp) = (l, \mathbf{S}[\![\kappa_{\mathbf{b}_l}\texttt{++}]\!]^\sharp_{\mathfrak{E}} \circ \mathbf{S}[\![x = 1]\!]^\sharp_{\mathfrak{E}} \rho^\sharp) \\
&\quad \texttt{in}\ \left(\omega^\sharp \odot_{\phi_0} \{\overline{\mathbf{b}}_l\}\right) \sqcup_{\mathscr{A}} \left(\omega^\sharp \odot_{\phi_1} \{\mathbf{b}_l\}\right)
\end{aligned}$$

Similarly, we give the following abstract transfer function for the $\texttt{uniform}_l(e_1, e_2)$ statement that generates U new transitions with appropriate state updates:

$$\begin{aligned}
&\mathbf{S}[\![x = \texttt{uniform}_l(e_1, e_2)]\!]^\sharp_\Omega \omega^\sharp \triangleq \\
&\quad \texttt{let}\, \phi(i) = \lambda\,(-, \rho^\sharp).\ (l, \mathbf{S}[\![\kappa_{\mathbf{u}_l^i}\texttt{++}]\!]^\sharp_{\mathfrak{E}} \circ \mathbf{S}[\![(x \leq e_2)]\!]^\sharp_{\mathfrak{E}} \circ \mathbf{S}[\![x = e_1 + i - 1]\!]^\sharp_{\mathfrak{E}} \rho^\sharp) \\
&\quad \texttt{and}\, \phi^\star = \lambda\,(-, \rho^\sharp).\ (l, \mathbf{S}[\![\kappa_{\mathbf{u}_l^\star}\texttt{++}]\!]^\sharp_{\mathfrak{E}} \circ \mathbf{S}[\![(e_1 + U \leq x \leq e_2)]\!]^\sharp_{\mathfrak{E}} \circ \mathbf{S}[\![x = \top]\!]^\sharp_{\mathfrak{E}} \rho^\sharp) \\
&\quad \texttt{in}\, (\bigsqcup_{1 \leq i \leq U-1} \omega^\sharp \odot_{\phi(i)} \{\mathbf{u}_l^i\}) \sqcup_{\mathscr{A}} (\omega^\sharp \odot_{\phi^\star} \{\mathbf{u}_l^\star\})
\end{aligned}$$

3.3 Abstract Markov Chains

The product $\mathcal{D}^\sharp \triangleq \mathcal{T}^\sharp \times \Omega^\sharp$ defines the domain of Abstract Markov Chains. It is composed of two parts. The first one is an abstraction of the markovian traces and is defined as the instance $\mathcal{T}^\sharp \triangleq \mathscr{A}(\Omega^\sharp, \Sigma^\sharp)$. This automaton is used to approximate the set of past observable traces reaching a given program location. The second part is an abstraction of the current partial scenarios starting from the last `ticks` statement. Since the states of an abstract scenario automaton already embed an abstraction of the program environments, we also employ this part to encode the current environments.

The concretization function gives the set of concrete markovian traces and partial scenarios encoded by an abstract Markov chain, and employs the previous trace and

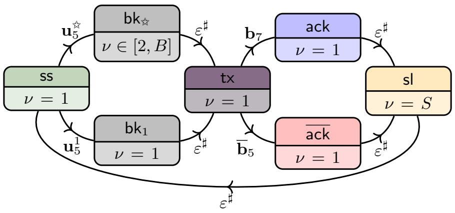


Fig. 6: Abstract Markov chain of the motivating example.

word concretizations (see Definition 4) as follows:

$$\gamma(\tau^\sharp, \omega^\sharp) = \{(\tau, \rho, \omega) \mid \tau \in \gamma_{\mathscr{A}}^{\mathbf{T}}(\tau^\sharp) \wedge \exists (l_1, \rho_1^\sharp) \xrightarrow{\xi_1^\sharp} \dots \xrightarrow{\xi_{n-1}^\sharp} (l_n, \rho_n^\sharp) \in \mathbf{T}(\omega^\sharp) : \\ \rho \in \gamma_{\mathfrak{E}}(\rho_n^\sharp) \wedge \omega \in \gamma_{\mathscr{A}}^{\mathbf{L}}(\xi_1^\sharp \dots \xi_{n-1}^\sharp)\}$$

Let us define now the abstract transfer function for the $\texttt{ticks}_l(e)$ statement since it is the only one that modifies the structure of the abstract Markov chain. It indicates that a new observable state has been encountered and that the pending scenarios are no longer partial and should be used to label the new transition, as shown by the following:

$$\mathbf{S}[\![x = \texttt{ticks}_l(e)]\!]^\sharp(\tau^\sharp, \omega^\sharp) \triangleq \texttt{let}\ \phi(-, \rho^\sharp) = (l, \mathbf{S}[\![\nu = e]\!]_{\mathfrak{E}}^\sharp \rho^\sharp)\ \texttt{in}\ (\tau^\sharp \odot_\phi \omega^\sharp, \varepsilon^\sharp)$$

where $\varepsilon^\sharp$ is the empty scenario word where all Parikh counters are reset to 0. The remaining statements are passed to the underlying $\Omega^\sharp$ and $\mathfrak{E}^\sharp$ domains and affect only the partial scenarios part. We can show that the following soundness condition is preserved:

$$\left(\mathbf{S}[\![s]\!] \circ \gamma\right)(\tau^\sharp, \omega^\sharp) \subseteq \left(\gamma \circ \mathbf{S}[\![s]\!]^\sharp\right)(\tau^\sharp, \omega^\sharp), \forall s \in \mathit{Stmt}, \forall (\tau^\sharp, \omega^\sharp) \in \mathcal{D}^\sharp \tag{9}$$

Example 5. The abstract markovian traces of our motivating example are depicted in Fig. 6 as an abstract automaton corresponding to the result of the analysis with $U = 2$. For the sake of clarity, we give to each abstract state a unique identifier and we show the inferred invariants about the sojourn time variable ν. The program locations and the remaining environment invariants are not represented. □

4 Stationary Distribution

In this section, we present a method for extracting safe bounds of the stationary distribution using information embedded in an abstract Markov chain. We do so by deriving a *distribution invariant* that establishes a set of parametric linear inequalities over the abstract states. Using the Fourier-Motzkin elimination algorithm, we can find guaranteed bounds of time proportion spent in a given abstract state.

We begin with some preliminary definitions. Let $T^\sharp = (S, s_0^\sharp, F, \Delta)$ be the markovian traces part of the program's abstract Markov chain over-approximating a set $T \subseteq \mathcal{T}_\Sigma^\Omega$ of concrete markovian traces. For each statement $\texttt{uniform}_l(e_1, e_2)$, we denote by $\overleftarrow{\texttt{u}e}_l = e_1$ and $\overrightarrow{\texttt{u}e}_l = e_2$ the bounds expressions of the distribution. Also, we define the functions $\hat{\mathbf{max}}[\![e]\!], \hat{\mathbf{min}}[\![e]\!] \in \Sigma^\sharp \to \mathit{Exp} \cup \{\infty\}$ giving respectively the evaluation of the maximal and minimal values of an expression e in a given abstract state, which is generally provided for free by the underlying numerical domain. In the case of relational domains, the returned bounds can be symbolic. For the sake of simplicity, we write $\hat{\mathbf{min}}_\star[\![e]\!]$ and $\hat{\mathbf{max}}_\star[\![e]\!]$ to denote respectively the minimal and maximal evaluations over the set of all reachable abstract states. The following definition gives a means to compute the probability of given abstract scenario.

Definition 7. *Let $\omega^\sharp \in \Omega^\sharp$ be an abstract scenario. Its probability is given by:*

$$\begin{cases} \hat{\Pr}(\varepsilon^\sharp) = 1, \hat{\Pr}(\mathbf{b}_l) = p_l, \hat{\Pr}(\overline{\mathbf{b}}_l) = 1 - p_l, \\ \hat{\Pr}(\mathbf{u}_l^i) = \frac{1}{\hat{\mathbf{min}}_\star[\![\overrightarrow{\mathtt{u}e_l}]\!] - \hat{\mathbf{max}}_\star[\![\overleftarrow{\mathtt{u}e_l}]\!]+1}, \hat{\Pr}(\mathbf{u}_l^\star) = \frac{\hat{\mathbf{max}}_\star[\![\overrightarrow{\mathtt{u}e_l}]\!] - \hat{\mathbf{min}}_\star[\![\overleftarrow{\mathtt{u}e_l}]\!]+2-U}{\hat{\mathbf{min}}_\star[\![\overrightarrow{\mathtt{u}e_l}]\!] - \hat{\mathbf{max}}_\star[\![\overleftarrow{\mathtt{u}e_l}]\!]+1} \\ \hat{\Pr}(\omega^\sharp \xi^\sharp) = \hat{\Pr}(\omega^\sharp)\hat{\Pr}(\xi^\sharp), \hat{\Pr}(\omega_1^\sharp + \omega_2^\sharp) = \hat{\Pr}(\omega_1^\sharp) + \hat{\Pr}(\omega_2^\sharp) \end{cases} \tag{10}$$

By combining the sojourn and probability invariants embedded in the abstract chain, we construct an abstract transition matrix that characterizes completely the stochastic properties of the program inside one finite data structure:

Definition 8 (Abstract transition matrix). *The abstract transition matrix $\hat{\mathbf{P}}$ is a square matrix of size $|S|$ where the entry for every abstract states $\sigma_i^\sharp, \sigma_j^\sharp \in S$ is defined as:*

$$\hat{\mathbf{P}}_{\sigma_i^\sharp, \sigma_j^\sharp} \triangleq \frac{\hat{\mathbf{max}}[\![\nu]\!](\sigma_j^\sharp)}{\hat{\mathbf{min}}[\![\nu]\!](\sigma_i^\sharp)} \sum_{\sigma_i^\sharp \xrightarrow{\omega^\sharp} \sigma_j^\sharp \in \Delta} \hat{\Pr}(\omega^\sharp) \tag{11}$$

Example 6. Consider our previous motivating example and its abstract chain represented in Fig. 6. Let $S = \langle \mathsf{ss}, \mathsf{bk}_1, \mathsf{bk}_\star, \mathsf{tx}, \mathsf{ack}, \overline{\mathsf{ack}}, \mathsf{sl} \rangle$ be the vector of abstract states. To obtain the matrix $\hat{\mathbf{P}}$, we iterate over all the transitions of the abstract chain. Consider for example the case of the transition $\mathsf{ss} \xrightarrow{u_5^\star} \mathsf{bk}_\star$. First, we apply (10) to compute the transitions probabilities $\hat{\Pr}(\mathbf{u}_5^\star) = \frac{B-1}{B}$. Afterwards, we extract the sojourn time bounds $\hat{\mathbf{max}}[\![\nu]\!](\mathsf{bk}_\star) = B$ and $\hat{\mathbf{min}}[\![\nu]\!](\mathsf{ss}) = 1$ from the embedded numeric environments. Finally, we apply (11) to obtain the matrix cell $\hat{\mathbf{P}}_{\mathsf{ss},\mathsf{bk}_\star} = \frac{B(B-1)}{B} = B - 1$. By iterating the same process for all transitions we obtain:

$$\hat{\mathbf{P}} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{S} \\ \frac{1}{B} & 0 & 0 & 0 & 0 & 0 & 0 \\ B-1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & \frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1-p & 0 & 0 & 0 \\ 0 & 0 & 0 & p & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & S & S & 0 \end{pmatrix}$$

□

The vector $\hat{\boldsymbol{\pi}}$ containing the proportion of time spent in every abstract state is called the *abstract stationary distribution*. It is defined as:

$$\hat{\boldsymbol{\pi}}_{\sigma^\sharp} \triangleq \sum_{\sigma \in \gamma_\Sigma(\sigma^\sharp)} \boldsymbol{\pi}_\sigma, \forall \sigma^\sharp \in S \tag{12}$$

where $\boldsymbol{\pi}$ is the concrete stationary distribution described in Section 2.3. It is important to note that since spurious concrete states $\sigma \in \gamma_\Sigma(\sigma^\sharp)$ have a null concrete stationary probability $\boldsymbol{\pi}_\sigma$, the abstract stationary probability $\hat{\boldsymbol{\pi}}_{\sigma^\sharp}$ represents the exact sum of the stationary probabilities of the real concrete states abstracted by $\sigma^\sharp$. Therefore, any lower and/or upper bounds that can be found about $\hat{\boldsymbol{\pi}}_{\sigma^\sharp}$ are also valid for the concrete states abstracted by $\sigma^\sharp$. To compute such bounds, we use $\hat{\mathbf{P}}$ with the following result:

Theorem 1 (Distribution invariant). $\hat{\pi} \le \hat{\pi}\hat{\mathbf{P}}$.

This theorem allows us to establish a system of parametric linear inequalities where the unknowns are the entries of the vector $\hat{\boldsymbol{\pi}}$. By adding the normalization condition $\sum_{\sigma^\sharp \in S} \hat{\boldsymbol{\pi}}_{\sigma^\sharp} = 1$, we can use this system to find safe bounds of the property of interest. Without loss of generality, assume that the time proportion of this property is associated to the stationary probability of some state $s^\sharp$. To compute a safe range of $\hat{\boldsymbol{\pi}}_{s^\sharp}$, we just have to perform a projection of the linear system $\hat{\boldsymbol{\pi}} \le \hat{\boldsymbol{\pi}}\hat{\mathbf{P}}$ that keeps only $\hat{\boldsymbol{\pi}}_{s^\sharp}$ and removes the other unknowns while preserving all constraints.

To do so, we have implemented a parametric Fourier-Motzkin projection algorithm [13,27] that returns parametric solutions to such problems. It eliminates the unnecessary unknowns sequentially and builds a decision tree that gives the system solutions depending on adequate parameters conditions. The general idea of the algorithm is the following. Assume that we are at the step of eliminating the unknown $\hat{\boldsymbol{\pi}}_{s_i^\sharp}$. We iterate over all leaves of the current decision tree $\{\langle C, I\rangle\}$, where I is a set of linear inequalities on the remaining unknowns and C is the condition on the parameters for obtaining the solution I. We examine the coefficients $\{a_{i,j}\}$ of $\hat{\boldsymbol{\pi}}_{s_i^\sharp}$ in I and we partition the inequalities depending on the sign of these coefficients. When the sign can not be determined, we create new branches within the decision tree to eliminate this ambiguity and we append the appropriate sign condition ($a_{i,j} > 0$, $a_{i,j} < 0$ and $a_{i,j} = 0$) to the branch condition C. At the end, we obtain a set of new leaves where all coefficients of $\hat{\boldsymbol{\pi}}_{s_i^\sharp}$ have known signs. At this point, we can transform I into a new system of inequalities by combining every couple of inequalities having opposite coefficient signs in a way to eliminate $\hat{\boldsymbol{\pi}}_{s_i^\sharp}$, and we keep the inequalities where the coefficient is null. After eliminating all untargeted unknowns, we obtain a set of bounding inequalities of $\hat{\boldsymbol{\pi}}_{s^\sharp}$ annotated by some parameters conditions.

Two important points should be noted. Firstly, this algorithm may not scale well for complex problems because the size of the decision tree can grow considerably in the presence of too many parametric coefficients. [1]. To improve the efficiency of the algorithm, we can reduce the precision of these linear parametric inequalities by using more abstract representations such as the domain of *interval linear inequalities* [6]. The second point is related to the soundness of the result. In our current implementation, we rely on an underlying symbolic environment to determine the sign of coefficients, which prevents us from ensuring the soundness of floating points operations during these computations. Nevertheless, we believe that we can inspire from *guaranteed linear programming* [24] to strengthen the resolution process and overcome this problem.

[1] That being said, this algorithm has shown to be more effective than built-in functions of many off-the-shelf symbolic environments, such as Sage and Mathematica, that did not return solutions for most benchmarks.

Protocol	PRISM	MARCHAL		
		Box	Octagon	Polyhedra
Single backoff				
$B = 2, S = 100$	2.96	1.92	2.57	1.94
$B = 4, S = 100$	2.94	2.19	4.54	2.65
Unbounded backoffs				
$B = 2, S = 100$	5.06	3.38	10.69	4.75
$B = 4, S = 100$	5.44	7.98	42.98	15.89
Bounded backoffs				
$B = 2, S = 100, N = 2$	4.12	8.96	28.59	14.52
$B = 4, S = 100, N = 2$	6.37	22.64	100.6	45.70

(a)

Protocol	PRISM	MARCHAL					
		Box		Octagon		Polyhedra	
		$U = 2$	$U = 4$	$U = 2$	$U = 4$	$U = 2$	$U = 4$
Single backoff							
$B = 20, S = 1000$	13.75	1.87	2.16	2.64	4.73	2.03	2.78
$B \in [2, 20], S \in [100, 1000]$	674.80	1.77	2.31	2.40	4.39	2.30	3.07
$B \geq 2, S \geq 100$	∞	1.7	1.84	2.57	4.10	2.11	2.69
Unbounded backoffs							
$B = 20, S = 1000$	45.11	5.12	10.28	12.81	44.10	6.87	17.72
$B \in [2, 20], S \in [100, 1000]$	∞	6.91	33.99	10.43	105.75	33.88	86.30
$B \geq 2, S \geq 100$	∞	2.87	4.95	39.09	102.44	35.0	83.20
Bounded backoffs							
$B = 20, S = 1000, N = 7$	50.24	7.01	17.58	43.33	173.10	17.64	61.84
$B \in [2, 20], S \in [100, 1000], N \in [1, 7]$	∞	16.82	57.77	120.34	338.45	110.31	252.79
$B \geq 2, S \geq 100, N \geq 1$	∞	6.49	16.51	75.79	251.78	55.32	150.66

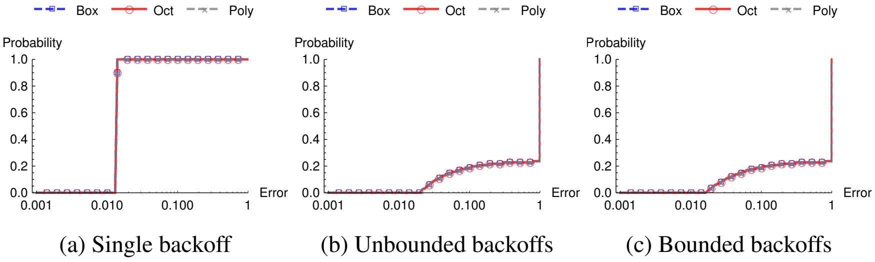
(b)

Table 1: Analysis time in seconds with (a) complete and (b) approximate partitioning.

5 Experiments

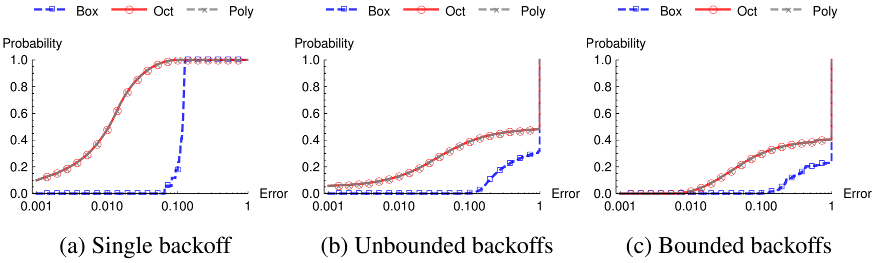
The proposed approach has been implemented in a prototype analyzer called MARCHAL (*MARkov CHains AnaLyzer*) using the OCaml language, the CIL frontend [23] and the Apron library [15]. Also, we implemented the parametric Fourier-Motzkin elimination algorithm in Mathematica. For our benchmarks, we compare MARCHAL to PRISM on three commonly used backoff mechanisms and we compute for each case the expected value of the throughput. The first backoff mechanism is the motivating example shown in Fig. 1 in which a single backoff is performed before transmitting every packet. In the second backoff mechanism, the sender tries to enhance the transmission reliability by performing an unbounded number of backoffs until receiving an acknowledgment from the destination. Finally, the third case study employs a bounded number of backoffs in which the number of successive attempts is limited by a parameter N. For all these cases, the backoff window is chosen uniformly from $[1, B]$ and the sleep period after the transmission transaction is determined by a parameter S.

The benchmarks consist in two categories of experiments in order to highlight the differences between MARCHAL and PRISM. For the first category, we fix the parameters to some small values and we configure MARCHAL to perform a complete partitioning



(a) Single backoff (b) Unbounded backoffs (c) Bounded backoffs

Fig. 7: Error distribution for the case $B = 20, S = 10^3$ with $U = 4$.



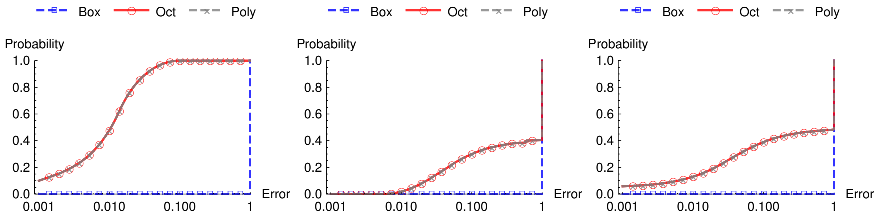
(a) Single backoff (b) Unbounded backoffs (c) Bounded backoffs

Fig. 8: Error distribution for the case $B \in [2, 20], S \in [10^2, 10^3]$ with $U = 4$.

of the uniform distribution. In this case, both tools are able to obtain the exact stationary distributions within a small delay, as summarized in Table 1(a), with an advantage of PRISM in many cases. In the second category, we extend the parameters space by considering three sub-cases: (i) the parameters have fixed large value, (ii) the parameters are not fixed but are bounded in some intervals, and finally (iii) the parameters are unbounded. To cover these cases in finite time, MARCHAL applies an approximate partitioning (into $U = 2$ and $U = 4$ partitions) and therefore can infer approximate and safe bounds of the throughput. However, PRISM can obtain only precise results and therefore can not provide an answer in most cases within a timeout of 30mn. The analysis times of this category of experiments are summarized in Table 1(b).

Let us now discuss the precision of the proposed approach. To evaluate it, we first compute the distance between the maximal and minimal bounds of the throughput over a large sample of parameters values, which results in a discrete set of observations of the maximal error of the analysis. From the resulting set of values, we compute the empirical distribution that gives the fraction of observations having a given maximal error. After that, we compute the cumulative distribution function for a better visualization of the variation of the error for the parameters sample.

We depict in Figs. 7, 8 and 9 the obtained results when setting $U = 4$. We can notice that the analysis with the octagon and polyhedra domains returned always the same precision level. Also, all domains give the same precision for the case of fixed parameters values. This is justified by the fact that the choice of the numerical domain affects the form of the sojourn time invariants used to compute the abstract transition



(a) Single backoff (b) Unbounded backoffs (c) Bounded backoffs

Fig. 9: Error distribution for the case $B \geq 2, S \geq 10^2$ with $U = 4$.

matrix. In the studied programs, all invariants have an octagonal form, so there is no need to infer more precise invariants. For the particular case of fixed parameters values, these invariants are just numeric intervals, which justifies that all domains offered the same precision level.

Additionally, we notice that the precision of the analysis for the unbounded and bounded backoff mechanisms is lower than the single backoff case, which is principally due to the partitioning of the uniform distribution that was too coarse in these cases. In practice, we were able to improve the precision of the inferred bounds by increasing the partition parameter U, but at the cost of analysis time. It is clear that more adequate partitioning techniques are necessary to obtain more precise results. Another source of precision loss is related to the current construction of the abstract transition matrix. We can see in (11) that the entries of the matrix reflect numeric constraints of the sojourn time over individual states only. However, some programs may constrain also the sojourn time over a sequence of abstract states, which is the case for example of the bounded backoff protocol that imposes a limit on the number of retransmissions. Since a retransmission involves a succession of many states, some invarants are ignored in the construction of $\hat{\mathbf{P}}$ which affects the precision of the analysis.

6 Related Work

The analysis of probabilistic programs has gained great interest over the last years. Many techniques have been proposed for extracting automatically quantiative properties from programs with varying precision/scalability tradeoffs.

PRISM [16] is a famous model checker that has been successfully applied for analyzing many probabilistic systems. It supports several interesting stochastic models, but is limited to finite state systems. Probabilistic symbolic execution [12,26] is another approach that annotates classical symbolic execution states with information about the past random events to be used in recovering the path probability. However, in most solutions, volume counting techniques are required, which limit their scalability.

Monniaux [22] and Di Pierro et al. [10] were the first propositions to extend abstract interpretation to probabilistic programs. Later, several works were proposed in the same direction [19,2,3], but they lack the ability to analyze some classical program constructs such as loops. In [8], Cousot et al. proposed a more general framework for probabilistic abstract interpretation that introduces the concept of *law abstraction* as a means to approximate probability distributions on program states. This formalism provides general theoretic guidelines to build sound probabilistic abstract interpretations, but does not provide practical solutions for widening loop iterations.

Another family of approaches is based on a weakest pre-expectation calculus introduced by McIver et al. [20] in order to infer *quantitative invariants* expressed as expectations of some program expressions. Chakarov et al. [4] extended this work in order to infer bounds of the probability of program assertions using the theory of Martingales. In [5], Chakarov et al. proposed another pre-expectation based analysis using abstract interpretation for discovering expectation invariants through the abstract domain of polyhedra with an appropriate widening operator. More recently, Barthe et al. [1] described a symbolic execution method that uses Doob's decomposition in order to infer Martingale expressions that help in deriving post-loop expectation of program variables.

7 Conclusion

We have presented a novel approach for obtaining guaranteed bounds of performance metrics of communication protocols. The method is based on the framework of abstract interpretation and proposes an Abstract Markov Chains domain for approximating the probabilistic semantics of programs. We have also explained how to exploit the information encapsulated within this domain in order to infer a sound approximation of the stationary distribution of the protocol, which is the key ingredient for computing a large range of performance metrics such as the throughput and the energy consumption. A prototype of the analysis have been presented along with some preliminary results. Many problems are still open to enhance the proposed approach. To enhance precision, we believe that is important to consider (*i*) developing more adequate partitioning of the `uniform` distribution and (*ii*) inferring multi-state sojourn time invariants. Finally, we have presented the analysis of a single process and we are interested in extending it to networked concurrent programs.

References

1. Barthe, G., Espitau, T., Ferrer Fioriti, L., Hsu, J.: Synthesizing probabilistic invariants via Doob's decomposition. In: CAV '16. LNCS, vol. 9779, pp. 43–61. Springer (2016)
2. Bouissou, O., Goubault, E., Goubault-Larrecq, J., Putot, S.: A generalization of p-boxes to affine arithmetic. Computing 94(2), 189–201 (2012)
3. Bouissou, O., Goubault, E., Putot, S., Chakarov, A., Sankaranarayanan, S.: Uncertainty propagation using probabilistic affine forms and concentration of measure inequalities. In: TACAS '16. LNCS, vol. 9636, pp. 225–243. Springer (2016)
4. Chakarov, A., Sankaranarayanan, S.: Probabilistic program analysis with martingales. In: CAV '13. LNCS, vol. 8044, pp. 511–526. Springer (2013)
5. Chakarov, A., Sankaranarayanan, S.: Expectation invariants for probabilistic program loops as fixed points. In: SAS '14. LNCS, vol. 8723, pp. 85–100. Springer (2014)
6. Chen, L., Miné, A., Wang, J., Cousot, P.: An abstract domain to discover interval linear equalities. In: VMCAI'10. LNCS, vol. 5944, pp. 112–128. Springer (2010)

7. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL '78. pp. 84–97. ACM (1978)
8. Cousot, P., Monerau, M.: Probabilistic abstract interpretation. In: ESOP '12. LNCS, vol. 7211, pp. 169–193. Springer (2012)
9. Cousot, R.: Fondements des méthodes de preuve d'invariance et de fatalité de programmes parallèles. Thèse d'État ès sciences mathématiques, Institut National Polytechnique de Lorraine, Nancy, France (1985)
10. Di Pierro, A., Wiklicky, H.: Concurrent constraint programming: Towards probabilistic abstract interpretation. In: PPDP '00. pp. 127–138. ACM (2000)
11. Feret, J.: Abstract interpretation-based static analysis of mobile ambients. In: SAS '01. LNCS, vol. 2126, pp. 412–430. Springer (2001)
12. Geldenhuys, J., Dwyer, M.B., Visser, W.: Probabilistic symbolic execution. In: ISSTA '12. pp. 166–176. ACM (2012)
13. Grïßlinger, A.: Extending the Polyhedron Model to Inequality Systems with Non-linear Parameters using Quantifier Elimination. Master thesis, University of Passau (2003)
14. Hahn, E., Hermanns, H., Zhang, L.: Probabilistic reachability for parametric markov models. International Journal on Software Tools for Technology Transfer 13(1), 3–19 (2011)
15. Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: CAV '09. LNCS, vol. 5643, pp. 661–667. Springer (2009)
16. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: CAV '11. LNCS, vol. 6806, pp. 585–591. Springer (2011)
17. Le Gall, T., Jeannet, B.: Lattice automata: A representation for languages on infinite alphabets, and some applications to verification. In: SAS '07. LNCS, vol. 4634, pp. 52–68. Springer (2007)
18. Lesens, D., Halbwachs, N., Raymond, P.: Automatic verification of parameterized networks of processes. Theoretical Computer Science 256(1-2), 113–144 (2001)
19. Mardziel, P., Magill, S., Hicks, M., Srivatsa, M.: Dynamic enforcement of knowledge-based security policies. In: CSF '11. pp. 114–128 (2011)
20. McIver, A., Morgan, C.: Abstraction, Refinement And Proof For Probabilistic Systems. Monographs in Computer Science, Springer (2004)
21. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation (HOSC) 19(1), 31–100 (2006)
22. Monniaux, D.: Abstract interpretation of probabilistic semantics. In: SAS '00. LNCS, vol. 1824, pp. 322–339. Springer (2000)
23. Necula, G., McPeak, S., Rahul, S., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: CC '02. pp. 213–228 (2002)
24. Neumaier, A., Shcherbina, O.: Safe bounds in linear and mixed-integer linear programming. Mathematical Programming 99(2), 283–296 (2004)
25. Parikh, R.: On context-free languages. Journal of ACM 13(4), 570–581 (1966)
26. Sankaranarayanan, S., Chakarov, A., Gulwani, S.: Static analysis for probabilistic programs: Inferring whole program properties from finitely many paths. In: PLDI '13. pp. 447–458. ACM (2013)
27. Suriana, P.: Fourier-Motzkin with Non-Linear Symbolic Constant Coefficients. Master thesis, Massachusetts Institute of Technology (2016)
28. Van Hentenryck, P., Cortesi, A., Le Charlier, B.: Type analysis of Prolog using type graphs. The Journal of Logic Programming 22(3), 179–209 (1995)
29. Venet, A.: Automatic analysis of pointer aliasing for untyped programs. Science of Computer Programming 35(2), 223–248 (1999)
30. Villemot, S.: Automates finis et intérpretation abstraite: Application à l'analyse statique de protocoles de communication. Rapport de DEA, École normale supérieure (2002)