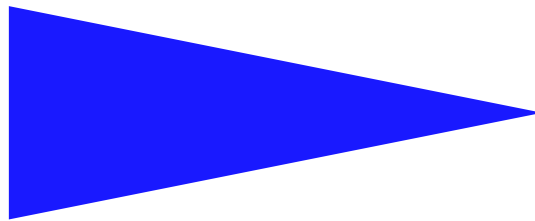




IRISA
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTEMES ALÉATOIRES

PUBLICATION
INTERNE
N° 1839



ANALYSIS OF COMMUNICATING INFINITE STATE MACHINES USING
LATTICE AUTOMATA

TRISTAN LE GALL & BERTRAND JEANNET



Analysis of Communicating Infinite State Machines using Lattice Automata

Tristan Le Gall & Bertrand Jeannot

Systemes communicants
Projet VerTeCs

Publication interne n° 1839 — March 2007 — 36 pages

Abstract: Communication protocols can be formally described by the Communicating Finite-State Machines (CFSM) model. This model is expressive, but not expressive enough to deal with complex protocols that involve structured messages encapsulating integers or lists of integers. This is the reason why we propose an extension of this model : the Symbolic Communicating Machines (SCM). We also propose an approximate reachability analysis method, based on lattice automata. Lattice automata are finite automata, the transitions of which are labeled with elements of an atomic lattice. We tackle the problem of the determinization as well as the definition of a widening operator for these automata. We also show that lattice automata are useful for the interprocedural analysis.

Key-words: Asynchronous systems, Abstract interpretation, Verification of infinite systems, FIFO channels, Lattice automata, Interprocedural analysis

(Résumé : *tsvp*)



Analyse des automates communicants étendus en utilisant des automates de treillis

Résumé : Les protocoles de communication utilisés actuellement peuvent être formellement décrits en terme d'automates communicants. Ce modèle, très expressif, n'est cependant pas suffisant pour modéliser facilement des protocoles complexes qui font intervenir des messages structurés encapsulant des entiers ou des listes d'entiers. C'est pourquoi nous proposons, dans ce document, une extension de ce modèle : les automates communicants symboliques. Nous proposons également une méthode d'analyse d'accessibilité approchée, basées sur des automates de treillis. Les automates de treillis sont des automates finis dont les transitions sont étiquetées par des éléments d'un treillis atomique. Nous aborderons le problème de la détermination de ces automates, ainsi que la définition d'un opérateur d'élargissement approprié. Nous montrerons aussi que ces automates sont utiles pour l'analyse interprocédurale.

Mots clés : Systèmes asynchrones, Interprétation abstraite, Vérification de systèmes infinis, canaux FIFO, Automates de treillis, Analyse interprocédurale

1 Introduction

Communication protocols play a very important role in the context of distributed computing and computer networks. These protocols, which aim at ensuring the data transmission and/or detect communication failures, may be wrong due to logical errors of the protocol designer. Such errors may be difficult to detect and to understand, due to the complexity and the non-determinism of the behavior of asynchronous systems. Moreover, even if the protocol is free of logical error, some properties remains of interest, like the maximum size that the buffers may have during the execution of the system.

For these reasons, the verification of communication protocols is both an important issue from the point of view of the protocol design, and a challenging problem for analysis techniques.

Communicating Finite-State Machines. Two fundamental approaches have been followed for the analysis of communicating systems in general. One consists in eliminating the need for analyzing FIFO queues contents by adopting a partial order semantics or a so-called *true concurrency* model: when one process sends a message to another process, one just records the information that the output precedes the input. The seminal work about event structures [40] leads later to scenario-based models like (High-level) Message Sequence Charts [27, 42] incorporated in UML.

The second approach, on which this paper focuses, consists in considering a model with explicit FIFO queues, and in analyzing their possible contents. A well-studied model is the model of *Communicating Finite-State Machines* (CFSM), in which finite-state machines communicate by sending messages belonging to a finite alphabet to unbounded FIFO queues. Although it is simple, this model is already quite expressive, since reachability is undecidable for this class of systems [11].

In [23], we proposed a method for the approximated analysis of *Communicating Finite-State Machines* (CFSM) based on Abstract Interpretation. The principles of our method were to represent the queue contents by regular languages and to define a *widening operator* that ensures the termination of the computation, but introduces some approximations.

Beyond CFSMs. In this paper, we extend this approach to more expressive communicating systems, where processes manipulate variables of unbounded types and where FIFO queues also transmit values belonging to infinite domains, like integers. The CFSM model is indeed not expressive enough for protocols that explicitly use variables, counters and infinite alphabets of messages. Typical examples of such protocols are sliding window protocols like the Service Specific Connection Oriented Protocol (SSCOP) [10].

We will sketch our approach with the following example :

Example. Fig. 1 depicts a very simplified version of a sliding window protocol. The *sender* process tries to send data (identified by an integer) to the *receiver* process. The receiver process sends an acknowledgment message identifying the data received. The sender has two variables : s is the index of the next data to send, and a is the index of the last acknowledgment message received. The protocol ensures that the sender waits for acknowledgment if $s = a + 10$. If the sender gets a message $\text{ack}(p)$ with $p > a + 1$, it means that at least one message has been lost and the protocol terminates with an error. There are two queues: one from the sender to the receiver containing data messages, the other containing acknowledgment messages. Notice that we do not model here possible loss of messages. \square

As in [23], we want to abstract the contents of the queues by regular languages, represented by finite automata. However, the messages contained in FIFO queues do not belong any more to a finite alphabet, as shown by the example.

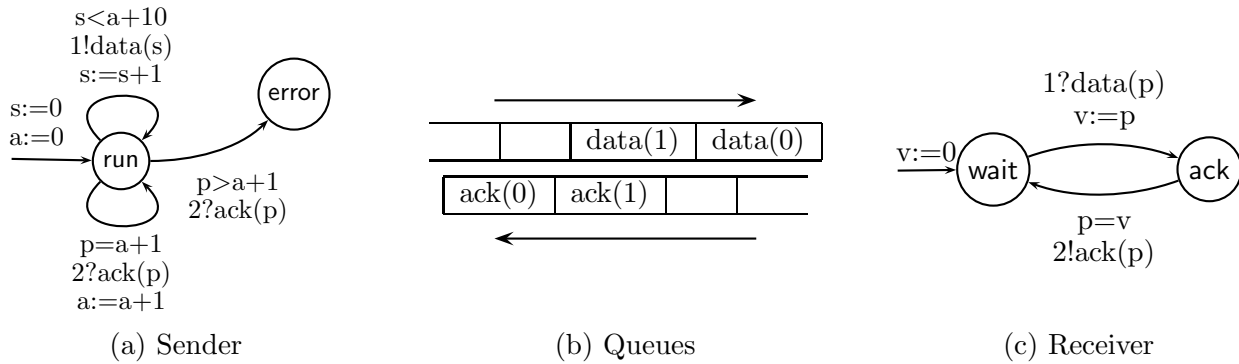


Figure 1: A simplistic protocol

Lattice automata. This motivates the introduction of *lattice automata*, which recognize (sort of) regular languages on infinite alphabets, called *lattice-based regular languages* (LBRL). In both LBRL and lattice automata, letters belonging to a finite alphabet are replaced by atomic elements of a lattice L . The idea is that a language like

$$Y = \sum_{n \geq 0} data(0) \cdot \dots \cdot data(n)$$

can be represented (with some approximations) by the LBRL

$$X = data(0) + data(0) \cdot data(1) + data(0) \cdot data(1) \cdot (data([2, +\infty]))^*$$

where the lattice $L = \{data\} \times \mathcal{I}(\mathbb{Z})$ is isomorphic to the lattice of integer intervals $\mathcal{I}(\mathbb{Z})$ ordered by inclusion. Here $data(0) \cdot data(1) \cdot (data([2, +\infty]))^*$ represents the set of words $data(0) \cdot data(1) \cdot (\sum_{n=2}^{\infty} data(n))^*$.

Contributions. The first contribution of this paper is to define with lattice automata an effective and canonical representation of languages on infinite alphabet, equipped with well-defined operations (union, intersection, concatenation, ...). Although the normal form we propose induces in general some approximations, it is a robust notion in the sense that the normalization operator is an upper closure operator which returns the best upper-approximation of a language in the set of normalized languages. The resulting abstract domain allows to lift any *atomic* abstract domain A for $\wp(S)$ to an abstract domain $\text{Reg}(A)$ for $\wp(S^*)$, the set of finite words defined on the alphabet S .

The second contribution of the paper is to demonstrate the use of this representation for the analysis of symbolic communicating machines (SCM). It appears indeed that this representation needs to be exploited in a clever way in order to be able to prove even simple properties, like in the example that messages in the queue 1 are always indexed by numbers smaller than the variable s . Our analysis shows that $a \leq s \leq a + 10$ (complete results are in Sect. 6).

A third contribution is to show that lattice automata are also adapted to the abstraction of call-stacks in imperative programs, and allows to design potentially very precise interprocedural analysis.

Outline Sect. 3 reminds some definitions about abstract interpretation and finite automata. In Sect. 4, we recall the method of [23] for analyzing CFSMs using regular languages, which is our departure point. The core contribution of the paper is Sect. 5 where we define lattice automata and their operations, which allows to manipulate languages on infinite alphabet. In Sect. 6 we exploit this representation for the abstract interpretation of Symbolic Communicating Machines. Sect. 7 discuss the use of lattice automata for interprocedural analysis, where they allows to abstract call-stacks of imperative programs.

2 Related work

The initial motivation of this work was the analysis of communicating machines. Many techniques have been devoted to the analysis of Communicating Finite-State Machines (CFSM), where both the machines and the alphabet of messages are *finite*. Reachability of CFSM is undecidable [11]. Most approaches for the verification of CFSM are based on exact but semi-decidable acceleration techniques [5, 4, 8, 2, 21]. A few attack the problem with abstract interpretation techniques [41, 29]. None of these works deals with a potentially infinite alphabet of messages.

More generally, there are several lines of work aiming at extending classical finite automata (resp. tree automata) representations for regular sets of words (resp. trees) for verification purposes. Mauborgne has proposed efficient representations for a class of sets of trees which strictly includes regular trees [36, 35]. Another approach is to focus on the decidability of some logic like the first order logic or the monadic second order logic when the model is a word with data or a tree with data (model of a XML document) [39]. New kind of automata were introduced, like register automata [32], pebble automata [38] or data automata [6], with the idea that a word with data satisfies the logical formula if it is recognized by the corresponding automata. This approach cannot be applied as is to our problem, as such a logical approach does not take into account any specific logical interpretation for data (in other word, the data domain is unspecified) and there is no notion of approximation.

They have been recently a lot of works devoted to shape analysis which can be relevant to the analysis of queues or stacks. A FIFO queue or a stack can indeed be represented by a list, which is most often the easiest data type to abstract in shape analysis techniques. However, most shape analysis focus on the structure (“the shape”) of the memory and ignores data values held by the memory cells [46, 48, 17, 9]. It is sometimes possible to handle finite data values, but mainly by brute force enumeration, which is algorithmically expensive in such a context, certainly more than the above-cited approaches based on finite automata. [24] is a pioneering work for taking into account data values in memory cells. It uses a global polyhedron to relate the numeric contents of each abstract memory cell in an abstract shape graph. The resulting abstraction is not comparable to our proposal, as it is based on very different principles. In particular, the information used for abstraction in shape graphs is mostly attached to nodes instead of edges as in automata. The involved algorithms are also probably more expensive than in our solution.

Conversely, it should be noted that our abstract domain could be applied to shape analysis, although this deserves yet a full study.

Work related to the application of lattice automata to stack abstraction and interprocedural analysis is delayed to Sect. 7.

3 Preliminaries

Finite automata. A *finite automaton* is a quintuple $\mathcal{A} = (Q, \Sigma, Q_0, Q_f, \delta)$ where Q is a finite set of states, Σ a finite alphabet of letters, $Q_0 \subseteq Q$ (resp. $Q_f \subseteq Q$) the subset of initial (resp. final) states, and $\delta \subseteq Q \times \Sigma \times Q$ the transition relation. A word $\sigma = \sigma_0 \dots \sigma_n \in \Sigma^*$ is recognized by \mathcal{A} if there exists a sequence $q_0 \dots q_{n+1} \in Q^*$ such that $q_0 \in Q_0$, $\forall i \leq n : (q_i, \sigma_i, q_{i+1}) \in \delta$, and $q_{n+1} \in Q_f$. The set of words recognized by \mathcal{A} is a regular language denoted by $L(\mathcal{A})$.

Let \approx be an equivalence relation on the set of states Q . The equivalence class of a state $q \in Q$ w.r.t. \approx is denoted by \tilde{q} . The *quotient automaton* $\mathcal{A}/\approx = \langle \tilde{Q}, \Sigma, \tilde{Q}_0, \tilde{Q}_f, \tilde{\delta} \rangle$ is defined by

- $\tilde{Q} = Q/\approx$, the set of equivalence classes;
- $\tilde{Q}_0 = \{\tilde{q}|q \in Q_0\}$ and $\tilde{Q}_f = \{\tilde{q}|q \in Q_f\}$;
- $(\tilde{q}, a, \tilde{q}') \in \tilde{\delta} \iff \exists q_0 \in \tilde{q}, \exists q'_0 \in \tilde{q}' : (q_0, a, q'_0) \in \delta$

For any equivalence relation \approx , we have $L(\mathcal{A}) \subseteq L(\mathcal{A}/\approx)$.

Given an equivalence relation \simeq on Q , and $k \geq 0$, the k -depth bisimulation equivalence relation \approx_k based on \simeq is defined inductively by

- $\approx_0 = \simeq$;
- $q \approx_{k+1} q' \iff \begin{cases} \forall(\sigma, q_1) \in \Sigma \times Q : \delta(q, \sigma, q_1) \Rightarrow \exists q'_1 : \delta(q', \sigma, q'_1) \wedge q_1 \approx_k q'_1 \\ \forall(\sigma, q'_1) \in \Sigma \times Q : \delta(q', \sigma, q'_1) \Rightarrow \exists q_1 : \delta(q, \sigma, q_1) \wedge q_1 \approx_k q'_1 \end{cases}$

The largest bisimulation relation contained in \simeq is defined as $\approx = \bigcap_{k \geq 0} \approx_k$. The equivalence bisimulation relation is defined as the largest bisimulation relation contained in the relation \simeq separating final states from other states: $q \simeq q' \triangleq (q \in Q_f \wedge q' \in Q_f) \vee (q \notin Q_f \wedge q' \notin Q_f)$. Two states are equivalent if they are related by the equivalence bisimulation relation.

\mathcal{A} is *deterministic* if there is a unique initial state and if $\delta(q, \sigma, q') \wedge \delta(q, \sigma, q'') \implies q' = q''$. Any finite automaton can be transformed into a deterministic automaton recognizing the same language, using the subset construction. \mathcal{A} is a minimal deterministic automaton (MDA) if it is deterministic and if there exists no deterministic automaton with fewer states recognizing the same language. Given a regular language L , there exists a unique (up to isomorphism) MDA $\mathcal{A}(L)$ recognizing it. Any deterministic automaton \mathcal{A} can be minimized by removing unreachable states and quotienting it w.r.t. the largest bisimulation relation separating final states from non final states.

Lattices. A partially ordered set (Λ, \sqsubseteq) is a *lattice* if it admits a smallest element \perp , a greatest element \top , and if any finite set of elements $X \subseteq \Lambda$ admits a *greatest lower bound (glb)* $\sqcap X$ and a *least upper bound (lub)* $\sqcup X$. A lattice is *complete* if the *glb* and *lub* operators are defined for any (possibly infinite) subset of Λ . An element $x \in L$ of a lattice Λ is an *atom* if it is minimal i.e. $\perp \sqsubset x \wedge \forall y \in \Lambda : \perp \sqsubset y \sqsubseteq x \implies y = x$. The set of atoms of L is denoted by $At(\Lambda)$. A lattice is *atomic* if any element $x \in L$ is the least upper bound of atoms smaller than itself : $x = \bigsqcup \{a \mid a \in At(\Lambda) \wedge a \sqsubseteq x\}$. For instance, the set of convex polyhedra in \mathbb{R}^n ordered by inclusion is an atomic lattice but is not a complete lattice. However, the set of convex sets in \mathbb{R}^n is an atomic and complete lattice. A finite partition of a lattice is a finite set $(\lambda_i)_{0 \leq i < n}$ of elements such that $\forall i \neq j : \lambda_i \sqcap \lambda_j = \perp$ and $\forall \lambda \in \Lambda : \lambda = \bigsqcup_{i=0}^{n-1} (\lambda \sqcap \lambda_i)$. If the lattice is atomic, there is an isomorphism between an element $\lambda \in \Lambda$ and its projection $\langle \lambda \sqcap \lambda_0, \dots, \lambda \sqcap \lambda_{n-1} \rangle$ on the partition. A (finite) *partitioning function* $\pi : \Sigma \rightarrow \Lambda$ is a function such that $(\pi(\sigma))_{\sigma \in \Sigma}$ is a (finite) partition of Λ .

Abstract Interpretation. Most program analysis problems come down to solving a fix-point equation $x = F(x)$ where x belongs to an ordered domain C (typically, the powerset $\wp(S)$ of the state space of a program). Abstract interpretation [15] is a general method to find approximate solutions of such fix-point equations. Its principles are to perform successively two kind of approximations:

1. The static approximation consists in substituting to the concrete domain C , a simpler abstract domain A . The abstract lattice A is linked to the concrete lattice C by a concretization function $\gamma : A \rightarrow C$ which is monotone. The concrete fix-point equation is transposed into the abstract domain A , so that to solve a new equation

$$y = F^\sharp(y), y \in A \text{ with } \gamma \circ F^\sharp \sqsupseteq F \circ \gamma$$

This ensures the soundness of the method, in the sense that $\gamma(\text{lfp}(F^\sharp)) \sqsupseteq \text{lfp}(F)$.

2. The iterative resolution of the new fix-point equation can still involve infinite iterations. Although one can choose abstract lattices A that satisfy the ascending chain condition, better results can be obtained by removing this restriction [16] and resorting to an additional dynamic approximation. It consists in using a widening operator ∇ as follows: the original sequence

$y_0 = \perp, y_{n+1} = F(y_n)$ is replaced by $z_0 = \perp, z_{n+1} = z_n \nabla F(z_n)$. Under technical assumptions on ∇ , the iterative computation of the sequence $(z_n)_{n \geq 0}$ is guaranteed to converge after a finite number of steps to some upper-approximation $z \in A$ of the least fix-point of F^\sharp (more precisely, a post-fix-point of F^\sharp).

This principles can be applied for instance to define an *interval analysis* of a program with integer variables, in which the concrete domain $C = \wp(\mathbb{Z}^n)$ of possible valuations of variables is abstracted by a product $(\mathcal{I}(\mathbb{Z}))^n$ of intervals on integers. An abstract value thus associates to each variable its possible range of values. This is an upper-approximation in the sense that what is guaranteed is that the values outside the interval cannot be taken by the variable.

Given two abstractions $\gamma_i : A_i \rightarrow C, i = 1, 2$ for the same concrete domain C , A_2 *refines* A_1 if there exists a (concretization) function $\gamma_{12} : A_1 \rightarrow A_2$ such that $\gamma_1 = \gamma_2 \circ \gamma_{12}$. This means that any concrete property $c \in C$ representable by A_1 (ie., $\exists a_1 \in A_1 : c = \gamma_1(a_1)$) is representable by A_2 (ie., $\exists a_2 \in A_2 : c = \gamma_2(a_2)$) by taking $a_2 = \gamma_{12}(a_1)$.

4 Analysis of CFSM using abstract interpretation

Our method for the analysis of *Symbolic Communicating Machine* (SCM) reuse most of the principles we have developed in [23] for the analysis of the simpler model of Communicating Finite-State Machines. As a consequence, we first describe this method in this section, which serves as an introduction to the next sections.

4.1 The CFSM Model

A *Communicating Finite-State Machine* (CFSM) models a system of finite-state machines sending or receiving messages via n unbounded FIFO queues, modeling communication channels.

Definition 1 (CFSM) *A communicating finite-state machine is given by a tuple (C, Σ, c_0, Δ) where:*

- C is a finite set of locations (control states)
- $\Sigma = \Sigma_1 \cup \Sigma_2 \cup \dots \cup \Sigma_n$ is a finite alphabet of messages, where Σ_i denotes the alphabet of messages that can be stored in queue i ;
- $c_0 \in C$ is the initial location;
- $\Delta \subseteq C \times A \times C$ is a finite set of transitions, where $A = \bigcup_i \{i\} \times \{!, ?\} \times \Sigma_i$ is the set of actions. An action can be
 - either an output $i!m$: “the message m is sent through the queue i ”;
 - or an input $i?m$: “the message m is received from the queue i ”.

In the examples, we define CFSMs in terms of an asynchronous product of finite state machines (FSMs) reading and writing on queues.

Example. The connexion/disconnection protocol between two machines is the following (Fig. 2): the client can open a session by sending the `open` message to the server. Once a session is open, the client may close it on its own by sending the `close` message or on the demand of the server if it receives the `disconnect` message. The server can read the request messages `open` and `close`, and ask for a session closure. \square

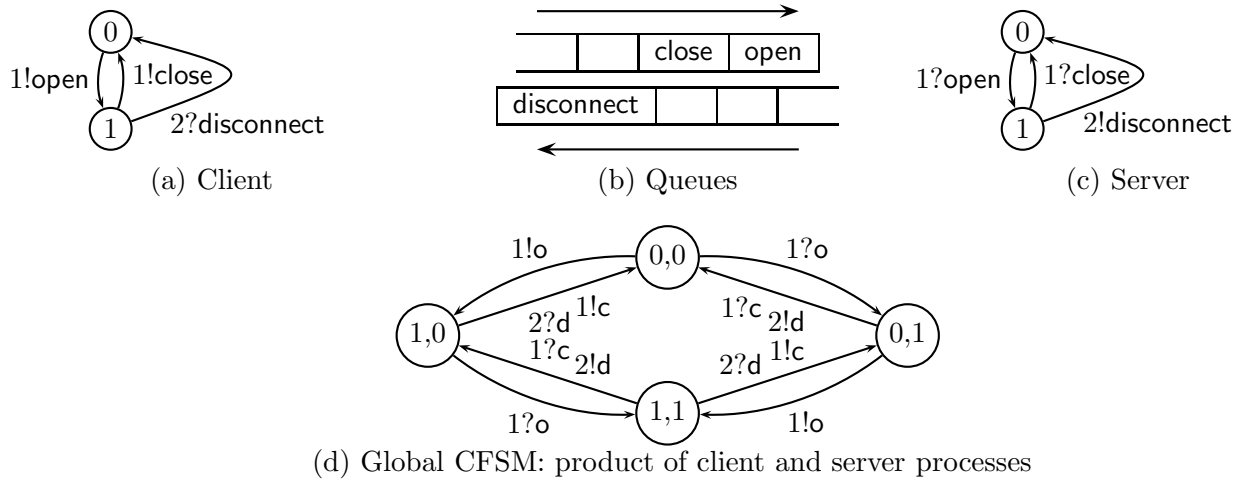


Figure 2: The connexion/disconnection protocol

The operational semantics of a CFSM (C, Σ, c_0, Δ) is given as an infinite transition system $\langle Q, Q_0, \rightarrow \rangle$ where

- $Q = C \times \Sigma_1^* \times \dots \times \Sigma_n^*$ is the set of states;
- $Q_0 = \{ \langle c_0, \varepsilon, \dots, \varepsilon \rangle \}$ is the set of initial states;
- \rightarrow is defined by the two rules:

$$\frac{(c_1, i!m, c_2) \in \Delta \quad w'_i = w_i \cdot m}{\langle c_1, w_1, \dots, w_i, \dots, w_n \rangle \longrightarrow \langle c_2, w_1, \dots, w'_i, \dots, w_n \rangle}$$

$$\frac{(c_1, i?m, c_2) \in \Delta \quad w_i = m \cdot w'_i}{\langle c_1, w_1, \dots, w_i, \dots, w_n \rangle \longrightarrow \langle c_2, w_1, \dots, w'_i, \dots, w_n \rangle}$$

A global state of a CFSM is thus a tuple $\langle c, w_1, \dots, w_n \rangle \in C \times \Sigma_1^* \times \dots \times \Sigma_n^*$ where c is the current location and w_i is a finite word on Σ_i representing the content of queue i . At the beginning, all queues are empty, so the *initial state* is $\langle c_0, \varepsilon, \dots, \varepsilon \rangle$.

4.2 Reachability analysis of a CFSM

The forward collecting semantics defines the semantics of a CFSM in terms of its set of reachable states. A set of states $X \in \wp(Q) = \wp(C \times \Sigma_1^* \times \dots \times \Sigma_n^*)$ can be viewed as a map $X : C \rightarrow \wp(\Sigma_1^* \times \dots \times \Sigma_n^*)$ associating a control state c with a language $X(c)$ representing all possible contents of the queues when being in the control state c . The forward semantics of actions $\llbracket a \rrbracket : \wp(\Sigma_1^* \times \dots \times \Sigma_n^*) \rightarrow \wp(\Sigma_1^* \times \dots \times \Sigma_n^*)$ is defined as:

$$\llbracket i!m \rrbracket(L) = \{ \langle w_1, \dots, w_i \cdot m, \dots, w_n \rangle \mid \langle w_1, \dots, w_i, \dots, w_n \rangle \in L \} \quad (1)$$

$$\llbracket i?m \rrbracket(L) = \{ \langle w_1, \dots, w_i, \dots, w_n \rangle \mid \langle w_1, \dots, m \cdot w_i, \dots, w_n \rangle \in L \} \quad (2)$$

$\llbracket i!m \rrbracket$ (resp. $\llbracket i?m \rrbracket$) associates to a set of queues contents the possible queues contents after the output (resp. the input) of the message m on the queue i , according to the operational semantics of CFSM. Using the inductive definition of reachability — a state is reachable either because it is initial, or because it is the immediate successor of a reachable state —, the reachability set RS is defined as the least solution of the fix-point equation

$$\forall c \in C, X(c) = X_0(c) \cup \bigcup_{(c', a, c) \in \Delta} \llbracket a \rrbracket(X(c')) \quad (3)$$

where Q_0 is the initial set of states.

As there is no general algorithm that can compute exactly such a reachability set [11], we propose in [23] an approximate analysis method based on abstract interpretation. The problem here is that we need to abstract the concrete domain of queues $\wp(\Sigma_1^* \times \dots \times \Sigma_n^*)$ which does not enjoy a finite representation.

4.3 Regular languages as an abstract domain for queues

Abstracting a single queue. For the sake of simplicity, we first consider the case of a single queue. Observe that the set $\wp(\Sigma^*)$ is actually the set of languages $\mathcal{L}(\Sigma)$ defined on the alphabet Σ . The solution we propose in [23] is to abstract the set of languages $\mathcal{L}(\Sigma)$ by the set of regular languages $\mathcal{R}(\Sigma)$. This simple solution presents two nice properties:

- $\mathcal{R}(\Sigma)$ is closed under union, intersection, negation and semantic transformers $\llbracket !m \rrbracket$ (corresp. to concatenation) and $\llbracket ?m \rrbracket$ (corresp. to the derivative operator of [12]). Moreover, $Q_0 = \{\langle c_0, \epsilon \rangle\}$ is regular, so that all operators involved in Eq. (3) can be transposed to $\mathcal{R}(\Sigma)$ without loss of information.
- From a computational point of view, regular languages have as a standard canonical representation the minimal deterministic automaton (MDA) recognizing them.

As a consequence, we only have to define a suitable widening operator to ensure convergence of iterative resolutions of fix-point equations on $\mathcal{R}(\Sigma)$. Indeed, the lattice $\mathcal{R}(\Sigma)$ does not satisfy the ascending chain condition and is even not complete. [23] adapts a widening operator for regular languages first mentioned in [20].

This widening operator is based on an extensive and idempotent operator $\rho_k : \mathcal{R}(\Sigma) \rightarrow \mathcal{R}(\Sigma)$ (*i.e.* $\rho_k(X) \supseteq X$ and $\rho_k \circ \rho_k = \rho_k$), where $k \in \mathbb{N}$ is a parameter. ρ_k will induce a widening operator defined by $X_1 \nabla_k X_2 = \rho_k(X_1 \cup X_2)$. Thus, the proposed widening does not work by extrapolating a difference as usual in abstract interpretation, but by simplifying the regular languages generated during the iterative resolution.

Now $\rho_k(X)$ is defined by quotienting the MDA $\mathcal{A}(X)$ recognizing X by the k -depth bisimulation relation based on the partition of the states Q of $\mathcal{A}(X)$ into $Q_0 \cap Q_f$, $Q_0 \setminus Q_f$, $Q_f \setminus Q_0$, and $Q \setminus (Q_0 \cup Q_f)$. Fig. 3 illustrates the effect of this operator. As the number of states of the MDA of $\rho_k(X)$ is bounded by $4^{|\Sigma|^{k+1}} \times 2^{|\Sigma|^k}$, the co-domain of ρ_k is finite for a k and Σ fixed. Hence the defined operator ∇_k satisfies the technical definition of a widening operator.

An upper-approximation of the least solution of Eq. 3 can now be computed by computing the sequence

$$\begin{aligned} \forall c \in C \quad X^{(0)}(c) &= X_0(c) \\ \forall c \in C \quad X^{(n+1)}(c) &= X^{(n)}(c) \nabla_k \bigcup_{(c', a, c) \in \Delta} \llbracket a \rrbracket(X^{(n)}(c')) \end{aligned} \quad (4)$$

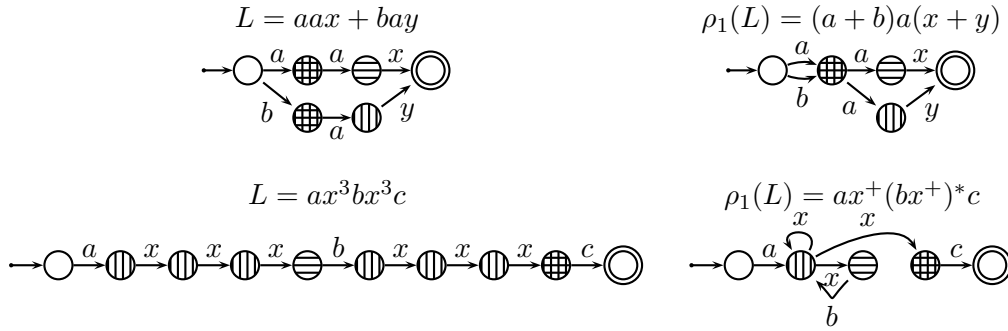
Abstracting several queues. When the system involves several queues, the simplest solution is to abstract each queue independently, leading thus to a *non-relational* or *attribute-independent* analysis. It consists in taking

$$A^{nr} = \mathcal{R}(\Sigma_1) \times \dots \times \mathcal{R}(\Sigma_n)$$

as an abstract lattice, ordered component-wise. The meaning function $\gamma^{nr} : A^{nr} \rightarrow \wp(\Sigma_1^* \times \dots \times \Sigma_n^*)$ is defined by

$$\gamma^{nr}(\langle L_1, \dots, L_n \rangle) = \gamma(L_1) \times \dots \times \gamma(L_n)$$

We can however view a configuration of the queues, which is a vector of words $(w_1, \dots, w_n) \in \Sigma_1^* \times \dots \times \Sigma_n^*$, as a single word $w_1 \# \dots \# w_n$ obtained by concatenation and the addition of a separation

Figure 3: The extensive and idempotent operator ρ_k on regular languages

letter \sharp . One can then apply the abstraction for a single queue on such concatenated word. This leads to the *relational* abstraction

$$A^r = \mathcal{R}(\Sigma \cup \{\sharp\})$$

$$\gamma^r(X) = \{\langle w_1, \dots, w_n \rangle \in \Sigma_1^* \times \dots \times \Sigma_n^* \mid w_1 \sharp \dots \sharp w_n \in X\}$$

where a single automaton is used to represent the possible contents of all queues. One just has to adapt to concatenated words the output and input operations, as well as the widening (by modifying the initial partition on which ρ_k is based).

This abstraction is strictly more precise, as A^r can represent exactly all the elements of A^{nr} , plus many more.

Example. The connexion/disconnection protocol depicted in Fig. 2 illustrates both kind of abstractions and illustrates the usefulness of a more precise relational analysis:

Relational Analysis		Non-Relational Analysis		
Client/Server	Queue 1 # Queue 2	Client/Server	Queue 1	Q.2
0/0	$(co)^*(oc)^*\#\varepsilon + c(oc)^*\#d$	0/0	$o^* + (o^*c)^+(\varepsilon + o^+ + o^+c)$	d^*
1/0	$(co)^*(oc)^*o\#\varepsilon + (co)^*\#d$	1/0	$(o^*c)^*o^+$	d^*
0/1	$c(oc)^*\#\varepsilon$	0/1	$o^* + (o^*c)^+(\varepsilon + o^+ + o^+c)$	d^*
1/1	$(co)^*\#\varepsilon$	1/1	$o^+ + o^*(co^+)^+$	d^*

The result given by the relational analysis happens to be the exact reachability set, unlike the non-relational one. The non-relational analysis misses the fact that there is at most one d in the second queue, which induces many approximations. \square

However, this operator ensures the convergence if and only if the alphabet Σ is finite. Thus this work cannot be applied for communicating systems with an infinite alphabet of messages, e.g. a protocol that can send integer values as messages, like the example of Sect. 1.

If we want to deal with such protocols, we must find a way to combine the previous work (on regular languages) with more classical abstractions of infinite domains (intervals, polyhedra, etc...).

5 Lattice automata

In this section we define with *lattice automata* an abstract representation for languages on infinite alphabets. The principle of lattice automata is to use elements of an atomic lattice for labeling the transitions of a finite automaton, and to use a partition of this lattice in order to define a projected finite automaton which acts as a guide for defining extensions of the classical finite automata operations.

We motivate our choices and show that they lead to a robust notion of approximation, in the sense that normalization is an upper-closure operation and can be seen as a best upper-approximation in the lattice of normalized lattice automata.

Lattice automata are finite automata, the transitions of which are labeled by elements of an atomic lattice (Λ, \sqsubseteq) instead of elements of a finite and unstructured alphabet. They recognize languages on atomic elements of this lattice. For instance, the interval automaton of Fig. 4 recognizes all sequences of rational numbers $x_0 \dots x_{n-1}$ where n is odd, $x_{2i} \in [0, 1]$ and $x_{2i+1} \in [1, 2]$. Such an automaton can be used to represent possible contents of a queue containing rational numbers.

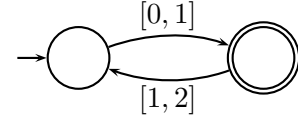


Figure 4: an interval automaton

However, the classical operations on finite automata cannot be extended as is on lattice automata. It is in particular the case of the determinization and minimization operations, because some lattice automata cannot be determinized in the usual way (this would require the ability to complement elements). We need these two operations which provide a normal form in the finite case, and which allows to define in a simple way other operations such as union and intersection. Consequently, we propose another notion of determinization which is in some sense an optimal approximation of the classical operation. We first formally define lattice automata, and we advocate the idea of using a partition of the lattice in order to define a projected finite automaton which acts as a guide for defining determinization, minimization operations. We then define the other classical operations such as union and intersection, before defining a suitable widening operator which allows to consider lattice automata as a fully equipped abstract domain.

5.1 Basic definition and discussion

Definition 2 (Lattice automaton) A lattice automaton is a tuple $\langle \Lambda, Q, Q_0, Q_f, \delta \rangle$ where :

- Λ is an atomic lattice, the order of which is denoted by \sqsubseteq ;
- Q is a finite set of states;
- $Q_0 \subseteq Q$ and $Q_f \subseteq Q$ are the sets of initial and final states;
- $\delta \subseteq Q \times (\Lambda \setminus \{\perp\}) \times Q$ is a finite transition relation.¹

A finite word $w = a_0 \dots a_n \in At(\Lambda)^*$ is accepted by the lattice automaton if there exists a sequence q_0, q_1, \dots, q_{n+1} such that $q_0 \in Q_0$, $q_{n+1} \in Q_f$, and $\forall i \leq n, \exists (q_i, \lambda_i, q_{i+1}) \in \delta : a_i \sqsubseteq \lambda_i$.

The set of words recognized by a lattice automaton \mathcal{A} is denoted by $L_{\mathcal{A}}$. The inclusion relation between languages induces a partial order on lattice automata :

Definition 3 (Partial order \sqsubseteq on lattice automata) The partial order \sqsubseteq between lattice automata is defined as $\mathcal{A} \sqsubseteq \mathcal{A}'$ iff $L_{\mathcal{A}} \subseteq L_{\mathcal{A}'}$.

Remark 1 (Downward closure of transitions.) With the Definition 2, if there exists a transition (q, λ, q') in δ , then any (q, λ', q') with $\lambda' \sqsubseteq \lambda$ may be added without modifying the recognized language. Hence, such transitions are redundant.

From now on, we assume that all transitions of a lattice automaton are maximal in the following sense : $(q, \lambda, q') \in \delta \wedge (q, \lambda', q') \in \delta$ implies that λ and λ' are not comparable.

¹No transition is labeled with the bottom element \perp .

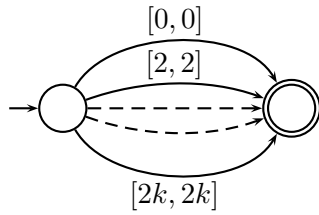


Figure 5: A family of interval automata \mathcal{A}_k with unbounded branching degree

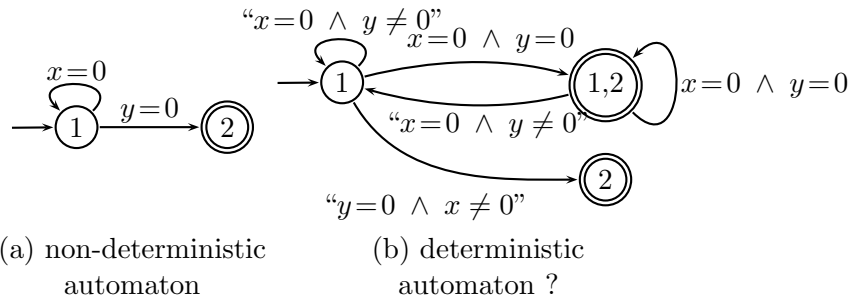


Figure 6: Attempt to determinize a lattice automaton on the lattice of affine equalities

Remark 2 (Words on atoms) We restrict the recognized words of a lattice automaton to be composed of atoms because we want the two automata of Fig. 8 to be equivalent in terms of their recognized languages. If words were composed of any elements of the lattice, the word composed of the single element $\{0 \leq x \leq 3, 0 \leq y \leq 1\}$ would be recognized by the first automaton but not the second one. In the same spirit, we require that the lattice Λ is atomic, so that any element $\lambda \in \Lambda$ labeling a transition is isomorphic to the set of atoms it dominates.

Remark 3 (Status of \perp) \perp cannot label a transition, not only as a consequence of the previous remark, but also because this corresponds to the intuition that \perp does not represent any element and denotes a notion of emptiness.

Definition 4 A lattice-based regular language is a language recognized by a lattice automaton \mathcal{A} . We denote by $\text{Reg}(\Lambda)$ the set of lattice-based regular languages.

Definition 2 raises however a number of problems. We expose them before introducing our solution to them. The first problem is related to the bounded branching degree property: in a deterministic finite automaton, there are at most $|\Sigma|$ transitions outgoing from a state. However, with definition 2, the branching degree of lattice automata is not bounded, as shown in Fig. 5.

The second problem is related to the notion of determinism. The classical notion of determinism can be extended in a straightforward way as follows:

Definition 5 (Deterministic lattice automaton) A lattice automaton $\langle \Lambda, Q, Q_0, Q_f, \delta \rangle$ is deterministic if it has a unique initial state and if $(q, \lambda_1, q_1) \in \delta \wedge (q, \lambda_2, q_2) \in \delta \implies \lambda_1 \sqcap \lambda_2 = \perp$.

Can now any lattice automaton be made deterministic while still recognizing the same language? The answer is negative, as illustrated by the example of Fig. 6, which uses the lattice of affine equalities [33]². The problem here is that elements of an atomic lattice cannot be complemented in general. With some lattices, like the lattice of intervals where an element can be complemented by a finite union of intervals, this problem can be overcome by splitting transitions, cf. Fig. 7, but we are back then to the branching degree problem.

Fig. 7 illustrates a third annoying aspect: we have given to the alphabet a lattice structure (instead of a weaker partial ordered structure) with the idea of exploiting the lattice operations offered by abstract domains. But in Fig. 7(c) we cheat by using several transitions instead of using the least upper bound operator. Moreover, using such explicit unions may be problematic to define minimization and canonical form. For instance, how to choose between the two minimal automata of Fig. 8? The problem here is that there is no canonical representation for unions of convex polyhedra.

²The lattice of affine equalities, which could also be called the lattice of affine subspaces, is the lattice formed by the conjunctions of affine equalities on the space \mathbb{R}^n .

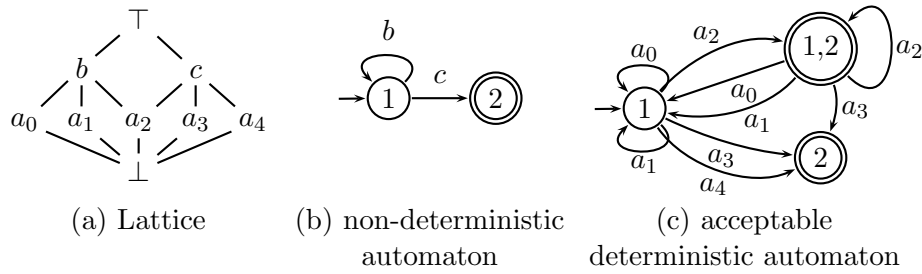


Figure 7: Attempt to determinize a lattice automaton

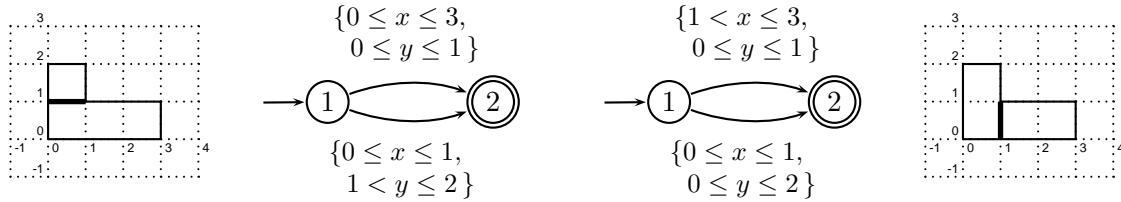


Figure 8: Two deterministic convex polyhedra automata that are equivalent

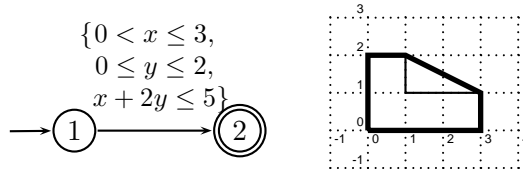


Figure 9: A merged PLA with the one-element partition

The solution we propose to fix these problems is to use a finite partition of the lattice Λ , which allows to decide when two transitions should be merged using the least upper bound operator. The fusion of transitions will induce in general an over-approximation, controlled by the fineness of the partition. The gain is that the projection of labels onto their equivalence classes produces a finite automaton on which we can reuse classical notions.

Definition 6 (Partitioned lattice automaton (PLA)) A partitioned lattice automaton (PLA) \mathcal{A} is a lattice automaton $\mathcal{A} = \langle \Lambda, \pi, Q, Q_0, Q_f, \delta \rangle$ equipped with a partitioning function $\pi : \Sigma \rightarrow \Lambda$ such that Σ is a finite alphabet and the transition relation δ satisfies:

$$\forall (q, \lambda, q') \in \delta, \exists \sigma \in \Sigma : \lambda \sqsubseteq \pi(\sigma)$$

A PLA is merged³ if:

$$(q, \lambda_1, q') \in \delta \wedge (q, \lambda_2, q') \in \delta \implies \pi^{-1}(\lambda_1) \cap \pi^{-1}(\lambda_2) = \emptyset$$

Definition 7 (Shape automaton and Shape equivalence) Given a PLA $\mathcal{A} = \langle \Lambda, \pi, Q, Q_0, Q_f, \delta \rangle$, its shape automaton $\text{shape}(\mathcal{A})$ is a finite automaton $(\Sigma, Q, Q_0, Q_f, \rightarrow)$ obtained by projecting the transition relation δ onto the equivalence classes: $(q, \lambda, q') \in \delta \implies q \xrightarrow{\pi^{-1}(\lambda)} q'$.

Two PLA are shape-equivalent if their two shapes recognize the same language.

Two transitions of a PLA labeled by elements belonging to different equivalence classes cannot be merged and are always kept separate, whereas they might be merged in the opposite case. Determin-

³The term “merged” comes from the merging operation needed to build a merged PLA from a given PLA.

istic merged PLA have the finite branching degree property: its states can have at most $|\Sigma|$ outgoing transitions.

From an expressiveness point of view, PLA are as expressive as lattice automata.

Proposition 1 (Equivalence between lattice automata and PLA) *Given a lattice automaton $\mathcal{A} = \langle \Lambda, Q, Q_0, Q_f, \delta \rangle$ and a partitioning function $\pi : \Sigma \rightarrow \Lambda$, there exists a partitioned lattice automaton $\mathcal{A}' = \langle \Lambda, \pi, Q, Q_0, Q_f, \delta \rangle$ recognizing the same language (this relies on the atomic lattice assumption).*

Proof: \mathcal{A}' is obtained from \mathcal{A} by replacing each transition (q, λ, q') of \mathcal{A} by at most $|\Sigma|$ transitions (q, λ_i, q') , where $\lambda_i = \lambda \sqcap \pi(\sigma_i)$ if $\lambda_i \neq \perp$. \square

However, for a given partition, merged PLA are strictly less expressive, as shown on Fig. 8 and 9 with the trivial partition of size 1. Moreover, if one considers the lattice of affine equalities, which can be partitioned only with the trivial partition of size 1, the automaton of Fig. 6(a) shows that in general merged PLA are strictly less expressive than PLA.

Proposition 2 *Let \mathcal{A} be a PLA. If $\text{shape}(\mathcal{A})$ is deterministic, then \mathcal{A} is deterministic.*

Proof: Let (q, λ_1, q_1) and (q, λ_2, q_2) be two transitions of \mathcal{A} , and (q, σ_1, q_1) and (q, σ_2, q_2) be the two corresponding transitions of $\text{shape}(\mathcal{A})$. Since $\text{shape}(\mathcal{A})$ is deterministic, $\pi^{-1}(\lambda_1) \cap \pi^{-1}(\lambda_2) = \emptyset$. Since π is a partitioning function, we have :

1. $\lambda_i \sqsubseteq \pi(\sigma_i)$
2. $\pi(\sigma_1) \sqcap \pi(\sigma_2) = \perp$

So $\lambda_1 \sqcap \lambda_2 = \perp$. And then \mathcal{A} is deterministic. \square

The converse is false in general. This property leads us to define a stronger notion of determinism :

Definition 8 (Strong determinism) *A PLA \mathcal{A} is strongly deterministic if $\text{shape}(\mathcal{A})$ is deterministic.*

This notion of determinism is useful since it is defined on the well-known finite automata, and is a guideline for the definition of a determinization algorithm. Therefore, in the sequel, “deterministic” means “strongly deterministic”.

With all those definitions, we will be able to define a normalized form for lattice-based regular language L . This normalization will exploit the following lemma :

Lemma 1 (Testing language inclusion) *Let $\mathcal{A} = \langle \Lambda, \pi, Q, Q_0, Q_f, \delta \rangle$ and $\mathcal{A}' = \langle \Lambda, \pi, Q', Q'_0, Q'_f, \delta' \rangle$ be two PLAs with the same partitioning function. Then $\mathcal{A} \sqsubseteq \mathcal{A}'$ iff there is a simulation relation $\mathcal{R} \subseteq \wp(Q) \times \wp(Q')$ verifying :*

1. $Q_0 \mathcal{R} Q'_0$
2. $\forall X \subseteq Q, \forall Y \subseteq Q' : X \mathcal{R} Y \implies \left[(X \cap Q_f \neq \emptyset) \implies (Y \cap Q'_f \neq \emptyset) \right]$
3. *Let $X \subseteq Q$ and $Y \subseteq Q'$ such that $X \mathcal{R} Y$. For any atom $a \in \text{At}(\Lambda)$, there are two sets $X_a \subseteq Q$ and $Y_a \subseteq Q'$ such that $X_a \mathcal{R} Y_a$ and:*

$$(\exists (q_x, \lambda_x, q'_x) \in \delta : q_x \in X \wedge a \sqsubseteq \lambda_x) \implies (\exists (q_y, \lambda_y, q'_y) \in \delta' : q_y \in Y \wedge a \sqsubseteq \lambda_y \wedge q'_x \in X_a \wedge q'_y \in Y_a)$$

Algorithm: Inclusion test for two PLAs
Input: two PLAs $\mathcal{A} = \langle \Lambda, \pi : \Sigma \rightarrow \Lambda, Q, Q_0, Q_f, \delta \rangle$ and $\mathcal{A}' = \langle \Lambda, \pi : \Sigma \rightarrow \Lambda, Q', Q'_0, Q'_f, \delta' \rangle$
Output: a boolean `is_included`
begin
 `is_included := true ;`
 $\mathcal{R} := \{(Q_0, Q'_0)\};$
 $ToDo := \{(Q_0, Q'_0)\};$
 while $ToDo \neq \emptyset \wedge \text{is_included}$ **do**
 $(X, Y) := \text{pickAndRemoveElement}(ToDo);$
 for all $a \in \text{At}(\Lambda)$ **do** ¹
 $X_a := \emptyset; Y_a := \emptyset;$
 for all $(q_x, \lambda_x, q'_x) \in \delta$ such that $q_x \in X$ and $a \sqsubseteq \lambda_x$ **do**
 $X_a := X_a \sqcup \{q'_x\};$
 endfor
 for all $(q_y, \lambda_y, q'_y) \in \delta$ such that $q_y \in Y$ and $a \sqsubseteq \lambda_y$ **do**
 $Y_a := Y_a \cup \{q'_y\};$
 endfor
 if $(Y_a = \emptyset \implies X_a = \emptyset)$ **then**
 `is_included := is_included \wedge $(X_a \cap Q_f \neq \emptyset \implies Y_a \cap Q'_f \neq \emptyset)$;`
 if $(X_a, Y_a) \notin \mathcal{R}$ **then**
 $\mathcal{R} := \mathcal{R} \cup (X_a, Y_a) ;$
 $ToDo := ToDo \cup (X_a, Y_a) ;$
 endif
 endif
 endfor
 endwhile
 return `is_included ;`
end

¹ The number of atoms may be infinite. So an implementation would manipulate elements $\lambda_x \sqcap \lambda_y$ for all outgoing transitions $(q_x, \lambda_x, q'_x) \in \delta, q_x \in X$ and $(q_y, \lambda_y, q'_y) \in \delta, q_y \in Y$.

Figure 10: Inclusion test algorithm for a merged PLA

Proof: On one hand, if there is a simulation like this one, it is obvious that $L_{\mathcal{A}} \subseteq L_{\mathcal{A}'}$, because of the definition of a language recognized by a lattice automaton.

On the other hand, if $L_{\mathcal{A}} \subseteq L_{\mathcal{A}'}$, we can build \mathcal{R} starting from $Q_0 \mathcal{R} Q'_0$ and using the same algorithm as the “inclusion test algorithm” (Fig. 10). And the condition : $\forall X \subseteq Q, Y \subseteq Q', XRY \implies [(X \cap Q_f \neq \emptyset) \implies (Y \cap Q'_f \neq \emptyset)]$ is always true, because $L_{\mathcal{A}} \subseteq L_{\mathcal{A}'}$. This algorithm works if all states of the two automata are reachable from an initial state and coreachable from a final state. Note that this is an implicit hypothesis for all propositions and algorithms of the present paper. \square

Note that the algorithm for inclusion test does not need any determinization step. This determinization is implicitly performed on the fly during the algorithm.

5.2 Normalization of PLAs

Normalization of partitioned lattice automata will be obtained by merging, determinizing and minimizing PLA. These operations provide a canonical form for PLA. Although this normalization induces an upper-approximation of the recognized language, this is a robust notion in the sense that the approximation is optimal.

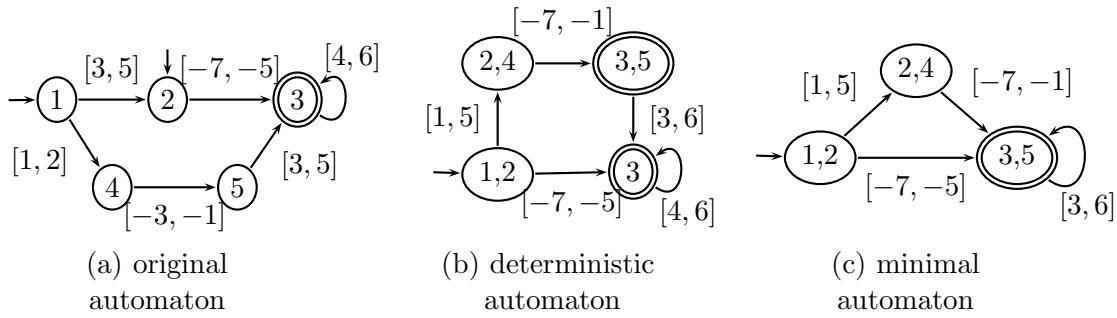


Figure 11: Determinization and minimization of an interval automaton with the partition $] -\infty, 0] \sqcup [0, +\infty[$

5.2.1 Merging.

Any PLA $\mathcal{A} = \langle \Lambda, \pi, Q, Q_0, Q_f, \delta \rangle$ recognizing a language L can be transformed into a merged PLA $\mathcal{A}_m = \langle \Lambda, \pi, Q, Q_0, Q_f, \delta_m \rangle \sqsupseteq \mathcal{A}$ recognizing a language $L_m \supseteq L$ by merging transitions as follows:

$$\frac{q, q' \in Q \quad \sigma \in \Sigma \quad \lambda_m = \bigsqcup \{ \lambda \sqcap \pi(\sigma) \mid (q, \lambda, q') \in \delta \}}{(q, \lambda_m, q') \in \delta_m}$$

The upper-approximation on the recognized language comes from the use of the *lub* operator. For instance, with a single equivalence class on \mathbb{R}^2 , the merged PLA associated to any automata depicted on Fig. 8 is depicted on Fig. 9.

5.2.2 Determinization

The determinization of a PLA mimics the determinization of its shape automaton using the subset construction on states and is illustrated on Fig. 11. The difference is that the transitions are merged in the course of the algorithm when they are labeled with values belonging to the same equivalence class. The resulting algorithm is given in Fig. 12.

Proposition 3 (Determinising PLA is a best upper-approximation) *Let \mathcal{A} be a PLA and \mathcal{A}' the PLA obtained with the algorithm of Fig. 12. Then \mathcal{A}' is the best upper-approximation of \mathcal{A} as a merged and deterministic⁴ PLA:*

1. $\mathcal{A} \sqsubseteq \mathcal{A}'$;
2. For any merged and deterministic PLA \mathcal{A}'' based on the same partition as \mathcal{A} , $\mathcal{A} \sqsubseteq \mathcal{A}'' \implies \mathcal{A}' \sqsubseteq \mathcal{A}''$.

Proof: The result is a merged and deterministic PLA because the algorithm determinizes the shape and performs merging of new transitions. It is also clear that the resulting automaton recognizes a greater language than the original one. We must prove the third point of the proposition.

Let $\mathcal{A} = \langle \Lambda, \pi, Q, Q_0, Q_f, \delta \rangle$ be the initial PLA, $\mathcal{A}' = \langle \Lambda, \pi, \mathcal{X}, X_0, X_f, \Delta \rangle$ the result of our algorithm and $\mathcal{A}'' = \langle \Lambda, \pi, Q'', Q_0'' = \{q_0''\}, Q_f'', \delta'' \rangle$ a merged and deterministic PLA such that $L_{\mathcal{A}} \subseteq L_{\mathcal{A}''}$. As usual, we suppose that, in these automata, all states are reachable from the initial states and coreachable from the final states.

We can build two simulation relations \mathcal{R}' on $\wp(Q) \times Q'$ and \mathcal{R}'' on $\wp(Q) \times Q''$ defined as :

- At the beginning, $(Q_0, q_0') \in \mathcal{R}'$ and $(Q_0, q_0'') \in \mathcal{R}''$,

⁴Remember that “deterministic” means strongly deterministic.

Algorithm: Determinization of a PLA
Input: a PLA $\mathcal{A} = \langle \Lambda, \pi : \Sigma \rightarrow \Lambda, Q, Q_0, Q_f, \delta \rangle$ whose states are co-reachable
Output: a merged deterministic PLA $\mathcal{A}' = \langle \Lambda, \pi, X, X_0, X_f, \Delta \rangle$
begin
 $X := \{Q_0\}; X_0 := \{Q_0\}; X_f := \emptyset;$
 $\Delta := \emptyset;$
 $ToDo := \{Q_0\};$
while $ToDo \neq \emptyset$ **do**
 $x := \text{pickAndRemoveElement}(ToDo);$
for all $\sigma \in \Sigma$ **do**
 $\lambda_u := \perp; x' := \emptyset;$
for all $(q, \lambda, q') \in \delta$ such that $q \in x$ and $\lambda \sqcap \pi(\sigma) \neq \perp$ **do**
 $\lambda_u := \lambda_u \sqcup (\lambda \sqcap \pi(\sigma)); x' := x' \cup \{q'\};$
endfor
if $\lambda_u \neq \perp$ **then**
 $\Delta := \Delta \cup \{(x, \lambda_u, x')\};$
if $x' \notin X$ **then**
 $X := X \cup \{x'\};$
if $x' \cap Q_f \neq \emptyset$ **then** $X_f := X_f \cup \{x'\};$
 $ToDo := ToDo \cup \{x'\};$
endif
endif
endfor
endwhile
end

Figure 12: Determinization algorithm for a merged PLA

- if we have $(Q_x, q_x) \in \mathcal{R}''$ then for all $\sigma \in \Sigma$, we consider $Q_y = \{q' \in Q \mid \exists (q, \lambda, q') \in \delta \wedge q \in Q_x \wedge \lambda \sqsubseteq \pi(\sigma)\}$. Since $L_{\mathcal{A}} \subseteq L_{\mathcal{A}''}$ and \mathcal{A}'' is deterministic, there is a unique $q_y'' \in Q''$ verifying $(q_x, \lambda'', q_y'') \in \delta'' \wedge \lambda'' \sqsubseteq \pi(\sigma)$, thus we add $(Q_y, q_y'') \in \mathcal{R}''$. Note that we have $\lambda' = \bigsqcup \{\lambda \mid \exists (q, \lambda, q') \in \delta \wedge q \in Q_x \wedge \lambda \sqsubseteq \pi(\sigma)\} \sqsubseteq \lambda''$. Since $L_{\mathcal{A}} \subseteq L_{\mathcal{A}''}$ and \mathcal{A}'' is deterministic, there is a unique $q_y'' \in Q''$ verifying $(q_x, \lambda', q_y'') \in \delta' \wedge \lambda' \sqsubseteq \pi(\sigma)$, so we can also add $(Q_y, q_y'') \in \mathcal{R}'$.

By construction, the two simulation relations define a simulation relation $\mathcal{R} \subseteq Q' \times Q'' : (q', q'') \in \mathcal{R} = \exists X \in \wp(Q), (X, q') \in \mathcal{R}' \wedge (X, q'') \in \mathcal{R}''$ so $L_{\mathcal{A}'} \subseteq L_{\mathcal{A}''}$. \square

Corollary 1 (The determinisation operation is an upper-closure operation) *The operation $\text{det} : \text{PLA} \rightarrow \text{PLA}$ is an upper-closure operation: it is (i) extensive: $\text{det}(\mathcal{A}) \sqsupseteq \mathcal{A}$; (ii) monotone: for any $\mathcal{A}, \mathcal{A}'$ defined on the same partition, $\mathcal{A} \sqsubseteq \mathcal{A}' \Rightarrow \text{det}(\mathcal{A}) \sqsubseteq \text{det}(\mathcal{A}')$; and (iii) idempotent: $\text{det}(\text{det}(\mathcal{A})) = \text{det}(\mathcal{A})$.*

Proof: The extensivity has already been shown. We have $\mathcal{A} \sqsubseteq \mathcal{A}' \sqsubseteq \text{det}(\mathcal{A}')$. $\text{det}(\mathcal{A})$ being the best approximation of \mathcal{A} as a deterministic merged PLA, $\text{det}(\mathcal{A}) \sqsubseteq \text{det}(\mathcal{A}')$, which proves the monotonicity. The idempotence is a trivial consequence of Prop. 3. \square

[30] also studies the determinization of extended automata, but their extended automata represent programs and have a different semantics. Transitions are labeled not only by guards (which could be identified to our labels) but also assignments. In this context, the proposed determinization algorithm is exact but may not terminate (it is only a semi-algorithm).

5.2.3 Minimization

We use for PLA a notion of minimization based on its shape automaton, in the same spirit as for the notion of determinism.

Definition 9 *A PLA is minimal or normalized if it is merged and if its shape automaton is minimal and deterministic. A normalized PLA will be also called a NLA (normalized lattice automaton).*

The algorithm to minimize a PLA consists in removing its unconnected states, determinizing it according to the previous algorithm and quotienting it according to the equivalence bisimulation relation as defined on the states of its shape automaton (*cf.* Sect. 3). However, when quotienting the states of a PLA, transitions labeled with elements belonging to the same equivalence class are merged, which may induce an over-approximation of the recognized language, as shown on Fig. 11.

Definition 10 (Quotient PLA) *Given a merged PLA $\langle \Lambda, \pi : \Sigma \rightarrow \Lambda, Q, Q_0, Q_f, \delta \rangle$ and an equivalence relation \approx on the set of states Q , the quotient automaton $\mathcal{A}/\approx = \langle \Lambda, \pi, \tilde{Q}, \tilde{Q}_0, \tilde{Q}_f, \tilde{\delta} \rangle$ is defined by*

- $\tilde{Q} = Q/\approx$, the set of equivalence classes;
- $\tilde{Q}_0 = \{\tilde{q}|q \in Q_0\}$ and $\tilde{Q}_f = \{\tilde{q}|q \in Q_f\}$;
- $\tilde{\delta}$ is defined by the rule
$$\frac{\sigma \in \Sigma \quad \lambda_u = \bigsqcup \{\lambda \sqsubseteq \pi(\sigma) \mid \exists q_0 \in \tilde{q}, \exists q'_0 \in \tilde{q}' : (q_0, \lambda, q'_0) \in \delta\}}{(\tilde{q}, \lambda_u, \tilde{q}') \in \tilde{\delta}}$$

Note that the quotient automaton is a merged PLA.

Theorem 1 (Minimizing PLA is a best upper-approximation) *For any PLA \mathcal{A} , there is a unique (up to isomorphism) NLA \mathcal{A}' based on the same partition π such that*

1. $\mathcal{A} \sqsubseteq \mathcal{A}'$
2. for any NLA \mathcal{A}'' based on the partition π , $\mathcal{A} \sqsubseteq \mathcal{A}'' \implies \mathcal{A}' \sqsubseteq \mathcal{A}''$.

Proof: The minimization algorithm consists first in determinizing the automaton, and then, in quotienting the result by the largest bisimulation relation on its states induced by its shape automaton. The result is merged and deterministic and the quotient operation ensures that its shape is minimal and deterministic. Thus it is minimal according to Def. 9.

Then we show that if two automata \mathcal{A}_1 and \mathcal{A}_2 recognize the same language L then $\text{minimize}(\mathcal{A}_1)$ is isomorph to $\text{minimize}(\mathcal{A}_2)$. We have $L \subseteq L_{\text{det}(\mathcal{A}_2)}$ and $\text{det}(\mathcal{A}_2)$ is merged and deterministic. According to the properties of $\text{det}(\mathcal{A}_1)$, we have $L_{\text{det}(\mathcal{A}_1)} \subseteq L_{\text{det}(\mathcal{A}_2)}$. The same thing holds when switching \mathcal{A}_1 and \mathcal{A}_2 , so $\text{det}(\mathcal{A}_1)$ and $\text{det}(\mathcal{A}_2)$ are two deterministic automata recognizing the same language. Thus the two automata are bisimilar according to a variant of Lemma 1 and we can conclude that $L_{\text{minimize}(\mathcal{A}_1)} = L_{\text{minimize}(\mathcal{A}_2)}$. Moreover $\text{shape}(\text{minimize}(\mathcal{A}_1))$ and $\text{shape}(\text{minimize}(\mathcal{A}_2))$ are isomorphic, and we can extend this isomorphism to $\text{minimize}(\mathcal{A}_1)$ and $\text{minimize}(\mathcal{A}_2)$ thanks to the previous language equality. \square

Definition 11 *Let fix a partitioning function $\pi : \Sigma \rightarrow \Lambda$. For any language $L \in \text{Reg}(\Lambda)$ recognized by a lattice automaton \mathcal{A} , \hat{L} will denote the language recognized by the unique NLA $\hat{\mathcal{A}}$ verifying the properties of Thm. 1.*

Corollary 2 (The normalisation operation is an upper-closure operation) *The function $\hat{\cdot} : \text{PLA} \rightarrow \text{NLA} \subseteq \text{PLA}$ is an upper-closure operator: it is extensive, monotone (given a fixed partition), and idempotent.*

Algorithm	Complexity
inclusion test	$O(2^n)$
merging	$O(m \cdot p)$
determinization	$O(2^n)$
quotienting	$O(n) + \text{merging (if } \approx \text{ is known)}$
minimization	$O(n \log n) + \text{quotienting}$
normalization	determinization + minimization

Table 1: Complexity of the basis operations on PLAs

Proof: Based on the same principle as Corollary 1. \square

Corollary 3 For any languages L_1 and L_2 ,

$$\widehat{L}_1 \cup \widehat{L}_2 \subseteq \widehat{L_1 \cup L_2} \quad (5)$$

$$\widehat{L_1 \cap L_2} \subseteq \widehat{L}_1 \cap \widehat{L}_2 \quad (6)$$

The inclusion is strict, because of the use of least upper bound during normalization. To see this, consider the lattice of intervals on rationals, partitioned with the trivial partition of size 1. Take $L_1 = 0$, $L_2 = 2$. One has $\widehat{L}_1 = L_1$, $\widehat{L}_2 = L_2$, $\widehat{L_1 \cup L_2} = 0 + 2$, but $\widehat{L_1 \cup L_2} = \sum_{x \in [0,2]} x$. Take now $L_1 = 0 + 2$ and $L_2 = 1 + 3$. One has $\widehat{L}_1 = \sum_{x \in [0,2]} x$, $\widehat{L}_2 = \sum_{x \in [1,3]} x$, $\widehat{L_1 \cap L_2} = \sum_{x \in [1,2]} x$, but $\widehat{L_1 \cap L_2} = \emptyset$.

Complexity. Let \mathcal{A} be a lattice automaton with n states, m transitions and a partition of size p (i.e. $|\Sigma| = p$). The complexity of the previous algorithms is given on Tab. 1, where the operations on Λ are considered as atomic.

Thm. 1 defines a normalization for languages recognized by PLA. For any language $L \in \text{Reg}(\Lambda)$ recognized by a PLA \mathcal{A} , \widehat{L} will denote the language recognized by the unique NLA verifying the properties of Thm. 1. The set of NLA defined on Λ with the partition π will be denoted by $\text{Reg}(\Lambda, \pi)$, which denotes also the corresponding set of recognized languages.

Having defined a normal form for PLA, we can now specify classical operations on languages recognized by NLA by defining them on their automata.

5.2.4 Refinement of the partitioning function

In the previous paragraphs, the partitioning function $\pi : \Sigma \rightarrow \Lambda$ was fixed. The precision of the approximations made during the merging, determinization and minimization operations depends on the fineness of the partitioning function. For example, all outgoing transitions from a given state would be merged during the determinization algorithm employed with the trivial partition of size 1.

Definition 12 A partitioning function $\pi_2 : \Sigma_2 \rightarrow \Lambda$ refines a partitioning function $\pi_1 : \Sigma_1 \rightarrow \Lambda$ if :

$$\forall \sigma_2 \in \Sigma_2, \exists \sigma_1 \in \Sigma_1 : \pi_2(\sigma_2) \sqsubseteq \pi_1(\sigma_1)$$

Let $\mathcal{A}_1 = \langle \Lambda, \pi_1 : \Sigma_1 \rightarrow \Lambda, Q, Q_0, Q_f, \delta_1 \rangle$ be a PLA. The automaton $\mathcal{A}_2 = \langle \Lambda, \pi_2 : \Sigma_2 \rightarrow \Lambda, Q, Q_0, Q_f, \delta_2 \rangle$ refines \mathcal{A}_1 if π_2 refines π_1 and the transitions of δ_2 are obtained by :

$$\frac{(q, \lambda_1, q') \in \delta_1 \quad \sigma_2 \in \Sigma_2 \quad \lambda_2 = \lambda_1 \sqcap \pi(\sigma_2)}{(q, \lambda_2, q') \in \delta_2}$$

Refining an automaton does not modify immediately the recognized language, but leads to a more precise upper-approximation in merging, determinization and minimization operations.

Proposition 4 *Let \mathcal{A}_1 be a PLA and \mathcal{A}_2 a PLA refining \mathcal{A}_1 . Then $L_{\mathcal{A}_1} = L_{\mathcal{A}_2}$, $\text{merge}(\mathcal{A}_1) \sqsupseteq \text{merge}(\mathcal{A}_2)$, $\text{det}(\mathcal{A}_1) \sqsupseteq \text{det}(\mathcal{A}_2)$ and $\widehat{\mathcal{A}}_1 \sqsupseteq \widehat{\mathcal{A}}_2$.*

Proof:

1. Obvious
2. For two states $q, q' \in Q$ and for any $\sigma_1 \in \Sigma_1, \sigma_2 \in \Sigma_2$ such that $\pi_2(\sigma_2) \sqsubseteq \pi_1(\sigma_1)$, we have :

$$\lambda_{m_2} = \bigsqcup \{ \lambda \sqcap \pi_2(\sigma_2) \mid (q, \lambda, q') \in \delta_1 \} \sqsubseteq \lambda_{m_1} = \bigsqcup \{ \lambda \sqcap \pi_1(\sigma_1) \mid (q, \lambda, q') \in \delta_1 \}$$

where λ_{m_i} labels the transition obtained by merging the transitions between q_1, q_2 labeled with elements in $\pi_i(\sigma_i)$.

3. Idea of the proof : during each iteration of the **while** loop, the determinization of the automaton \mathcal{A}_2 merges less states and less transitions than the determinization of the automaton \mathcal{A}_1 .
4. Idea of the proof : the equivalence class of the bisimulation relation (for the quotienting algorithms) are also more precise with π_2 , so both determinization and quotienting merge less states and transitions of the original automaton.

□

Refining the partition π makes the class of normalized languages $\text{Reg}(\Lambda, \pi)$ more expressive.

Proposition 5 *Let π_1 and π_2 two partitioning functions for Λ , with π_2 refining π_1 . Then $\text{Reg}(\Lambda, \pi_1) \subseteq \text{Reg}(\Lambda, \pi_2)$.*

Proof: Let $\mathcal{A}_1 \in \text{Reg}(\Lambda, \pi_1)$ be a NLA. Its refinement \mathcal{A}_2 according to π_2 (cf. Def. 12) is normalized, and thus $\mathcal{A}_2 \in \text{Reg}(\Lambda, \pi_2)$. □

Choosing an adequate partitioning function is thus important. For the analysis of SCM, where data messages are usually composed of a message type and some parameters, the type of the message defines a natural partition. When this standard partition is not sufficient for the analysis, it can be refined to a more adequate partition. In this sense, the abstraction refinement techniques based on partitioning (see for instance [28, 37]) are applicable to lattice automata.

5.3 Operations on PLA and their recognized languages

After the definition of a robust normalization concept, we can now define the classical operations on languages (union, intersection, ...) by defining them on lattice automata.

5.3.1 Set operations

Inclusion test. Two PLAs that are not normalized can be compared for language inclusion using a simulation relation taking into account the partial order of the lattice, cf. lemma 1. If they are normalized, one can first compare for inclusion their shape automata, and in case of inclusion, one can compare the labels of matching transitions.

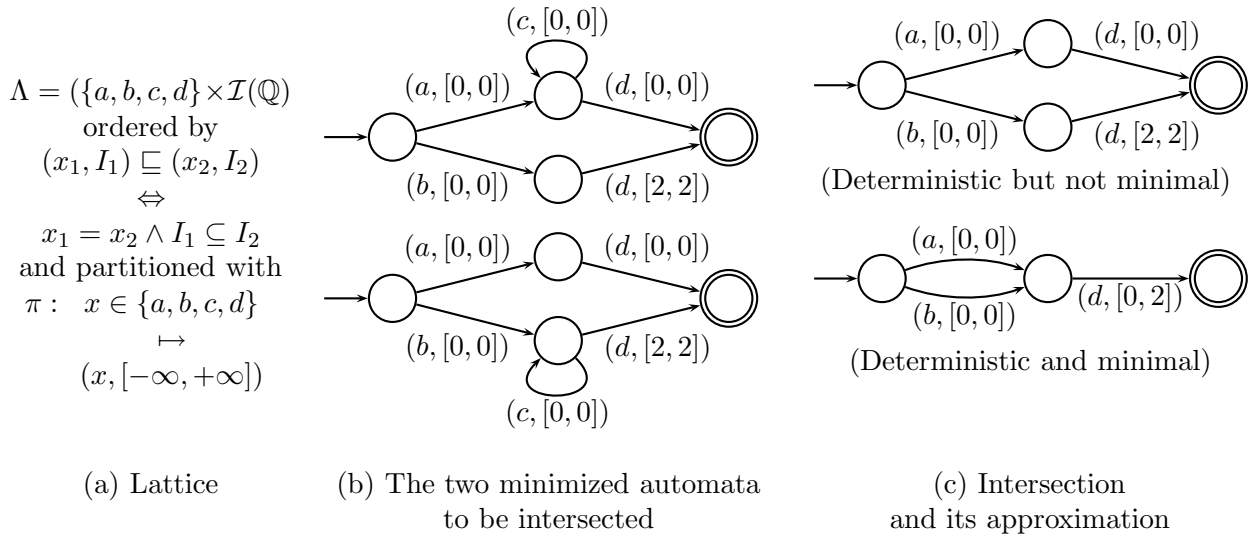


Figure 13: Intersection of normalized PLA and its upper-approximation

Union and least upper bound. The exact union of two PLAs $\mathcal{A}_i = \langle \Lambda, \pi, Q^i, Q_0^i, Q_f^i, \delta^i \rangle$ can be computed very simply as the disjoint union of the two PLA; this produces the PLA

$$\mathcal{A} = \langle \Lambda, \pi, Q^1 \cup Q^2, Q_0^1 \cup Q_0^2, Q_f^1 \cup Q_f^2, \delta^1 \cup \delta^2 \rangle$$

which is not deterministic.

Normalizing \mathcal{A} transforms the exact union operation in an upper bound operator \sqcup defined as follows: $\mathcal{A}_1 \sqcup \mathcal{A}_2 = \widehat{\mathcal{A}_1 \cup \mathcal{A}_2}$. As a corollary of Theorem 1, this upper bound operator is actually a *least upper bound operator* on the set of NLA ordered by language inclusion.

Intersection and its normalized upper-approximation. The exact intersection of two PLAs $\mathcal{A}_i = \langle \Lambda, \pi, Q^i, Q_0^i, Q_f^i, \delta^i \rangle$ can be computed as a product of the two PLA; this produces a PLA

$$\mathcal{A} = \langle \Lambda, \pi, Q^1 \times Q^2, Q_0^1 \times Q_0^2, Q_f^1 \times Q_f^2, \delta \rangle$$

where δ is defined by

$$\frac{(q_1, \lambda_1, q'_1) \in \delta_1 \quad (q_2, \lambda_2, q'_2) \in \delta_2 \quad \lambda_1 \sqcap \lambda_2 \neq \perp}{((q_1, q_2), \lambda_1 \sqcap \lambda_2, (q'_1, q'_2)) \in \delta}$$

We implicitly remove unconnected states in \mathcal{A} . If the input automata are normalized, \mathcal{A} is merged and deterministic, but not necessarily minimal. Minimizing it may induce an upper-approximation, as shown by Fig. 13. Thus, NLA are not closed under (exact) intersection.

These operations allow to equip the set of NLA with a join semilattice structure.

Proposition 6 (NLA as a join semilattice) *The set of NLA defined on an atomic lattice Λ with a fixed partition π , ordered by language inclusion, is a join semilattice: it has bottom and top elements, and a least upper bound operator.*

If the standard lattice operations on Λ are computable, so are the corresponding operations on the join semilattice of NLA.

Proof: It is clear that the bottom and top elements of this semilattice are respectively the empty automaton $\langle \Lambda, \pi, Q = \emptyset, Q_0 = \emptyset, Q_f = \emptyset, \delta = \emptyset \rangle$ recognizing no words, and the universal automaton $\langle \Lambda, \pi, Q = \{q\}, Q_0 = Q, Q_f = Q, \delta = \bigcup_{\sigma \in \Sigma} (q, \pi(\sigma), q) \rangle$ recognizing any word on the alphabet $At(\Lambda)$. The least upper bound operator is the operator \sqcup on NLA defined in Sect. 5.3. Last, we have also given in Sect. 5.3 algorithms for testing inclusion and for computing the least upper bound. \square

Remark 4 (Deterministic merged PLA as a full lattice) *If we consider the set of deterministic and merged (but not necessarily minimal) PLA, then this set partially ordered by language inclusion has a true lattice structure, instead of just the join semilattice structure of NLA. Indeed, Proposition 3 may be used to prove that the determinization of the exact union is a least upper bound operation, and as the exact intersection of 2 deterministic merged PLA is a deterministic merged PLA, this intersection is trivially a greatest lower bound.*

One could manipulate deterministic merged PLA instead of NLA. But there may be several deterministic merged PLA recognizing the same language, hence the isomorphism between languages and automata is lost. The other drawbacks of such a choice is that testing inclusion is more expensive (one needs to use simulation) and that one cannot extend the widening operation on finite automata defined in Sect. 4.3.

5.3.2 Other language operations

Language concatenation. Language concatenation on deterministic PLA is performed exactly as for finite automata, by substituting to the final states of the first automaton a copy of the initial state of the second automaton. The obtained automaton is non-deterministic in general and requires normalization. Language concatenation can also be computed on non deterministic PLA, by using ϵ transitions and ϵ -closure to remove them.

Left derivation. In the finite case, the left derivation [12] of a finite automaton is performed w.r.t. a letter. The corresponding operation would consists in deriving a PLA according to an equivalence class. However, the partition is a way to control the precision of the approximations performed by the various operations on PLA, and it does not have a semantic meaning.

As a consequence, we define a more general left derivation L/λ operator as follows:

$$\begin{aligned} \cdot/\cdot & : \text{Reg}(\Lambda) \times \Lambda \rightarrow \text{Reg}(\Lambda) \\ (L, \lambda) & \mapsto \{\omega \in \text{At}(\Lambda)^* \mid \exists a \sqsubseteq \lambda : a \cdot \omega \in L\} \end{aligned}$$

We have clearly the identity $L/\lambda = \sum_{a \in \text{At}(\lambda)} L/a$, from which we can deduce the following proposition:

Proposition 7 *Let $\lambda \in \Lambda$ and $(\lambda_\sigma)_{\sigma \in \Sigma}$ be the projection of λ on the partition defined by $\pi : \Sigma \rightarrow \Lambda$. Then $L/\lambda = \sum_{\sigma \in \Sigma} L/\lambda_\sigma$.*

The left derivation can be implemented exactly on a deterministic PLA $\mathcal{A} = \langle \Lambda, \pi, Q, Q_0 = \{q_0\}, Q_f, \delta \rangle$ as follows:

$$\begin{aligned} \cdot/\cdot & : \text{PLA} \times \Lambda \rightarrow (\Sigma \rightarrow \text{PLA}) \\ (L, \lambda) & \mapsto (\sigma \mapsto \mathcal{A}/(\lambda \sqcap \pi(\sigma))) \end{aligned}$$

where $\mathcal{A}/(\lambda \sqcap \pi(\sigma))$ is the empty automaton if $\lambda \sqcap \pi(\sigma) = \perp$, and the deterministic PLA $\langle \Lambda, \pi, Q, Q_0 = \{\delta(q_0, \sigma)\}, Q_f, \delta \rangle$ otherwise. One obtains a finite set of deterministic automata instead of a single one. This set can be of course upper-approximated by the least upper bound deterministic PLA, denoted by $\widehat{\mathcal{A}/\lambda} = \bigsqcup_{\sigma \in \Sigma} (\mathcal{A}/\lambda)(\sigma)$.

“First” and “Pop” operations. In addition to the left derivation, it may also be useful to extract the letters beginning the words of a language as follows:

$$\begin{aligned} \text{first} & : \text{Reg}(\Lambda) \rightarrow \wp(\text{At}(\Lambda)) \\ L & \mapsto \{a \in \text{At}(\Lambda) \mid a \cdot \omega \in L\} \end{aligned}$$

NLA operation	complexity	resulting PLA	exact operation ?
inclusion test	$O(n \cdot \log n)$	—	yes
union	$O(1)$	non-deterministic	yes
intersection	$O(n^2 + m^2)$	deterministic, non minimal	yes
language concatenation	$O(n \cdot p)$	non-deterministic	yes
letter right-concatenation	$O(n)$	non-deterministic	yes
letter left-concatenation	$O(1)$	normalized	yes
left derivation	$O(p)$	non-deterministic	yes
first	$O(p)$	—	yes
pop	$O(p)$	non-deterministic	yes
normalized union (\sqcup)	$O(2^n)$	normalized	no
normalized intersection (\sqcap)	$O(n^2 \log n)$	normalized	no
normalized pop	$O(p \cdot 2^n)$	normalized	no

Table 2: Complexity of various operations on NLA. We assume NLA with n states, m transitions and a partition of size p (i.e. $|\Sigma| = p$)

The equivalent (algorithmic) definition on a deterministic PLA $\mathcal{A} = \langle \Lambda, \pi, Q, Q_0 = \{q_0\}, Q_f, \delta \rangle$ is :

$$\begin{aligned} \text{first} & : \text{PLA} \rightarrow (\Sigma \rightarrow \Lambda) \\ \mathcal{A} & \mapsto (\sigma \mapsto \text{first}(\mathcal{A})(\sigma)) \end{aligned}$$

where for each $\sigma \in \Sigma$, $\text{first}(\mathcal{A})(\sigma)$ is the label of the unique transition $(q_0, \lambda, q) \in \delta$ with $\lambda \sqsubseteq \pi(\sigma)$, if it exists, and \perp otherwise. It is clear that the set of atoms covered by $\text{first}(\mathcal{A})$ is exactly $\text{first}(L_{\mathcal{A}})$:

$$\bigcup_{\sigma \in \Sigma} \{a \in \text{At}(\Lambda) \mid a \sqsubseteq \text{first}(\mathcal{A})(\sigma)\} = \text{first}(L_{\mathcal{A}})$$

The *first* operation can be combined with the left derivation in order to implement the *pop* operation, which extracts the first letters and derives the corresponding “residue” automaton :

$$\begin{aligned} \text{pop} & : \text{PLA} \times \Lambda \rightarrow (\Sigma \rightarrow \Lambda \times \text{PLA}) \\ (\mathcal{A}, \lambda) & \mapsto \left(\sigma \mapsto (\lambda \sqcap \text{first}(\mathcal{A})(\sigma), \mathcal{A}/(\lambda \sqcap \pi(\sigma))) \right) \end{aligned}$$

The complexity of the previous operations is summarized on Tab. 2.

5.4 Widening on NLA

The widening on NLA we define here combines the widening on finite automata of [23] reminded in Sect. 4.3 with the standard widening operator $\nabla_{\Lambda} : \Lambda \times \Lambda \rightarrow \Lambda$ that we assume for the atomic lattice Λ . If a widening operator is not strictly required for Λ (because Λ satisfies the ascending chain condition), then ∇_{Λ} can be defined as the least upper bound \sqcup .

We first extend the ρ_k operator defined on finite automata in Sect 4.3.

Definition 13 (Operator ρ_k on NLA) Let $\mathcal{A} = (\Lambda, \pi, Q, Q_0, Q_f, \Delta)$ be a NLA, $\text{shape}(\mathcal{A}) = (\Sigma, Q, Q_0, Q_f, \delta)$ its shape automaton. Let $k \geq 0$ be an integer and \approx_k the k -depth bisimulation relation on Q induced by the partition of Q into $Q_0 \cap Q_f$, $Q_0 \setminus Q_f$, $Q_f \setminus Q_0$ and $Q \setminus (Q_0 \cup Q_f)$ and the transition relation $\delta \subseteq Q \times \Sigma \times Q$. We define $\rho_k(\mathcal{A})$ as the quotient PLA \mathcal{A}/\approx_k .

The widening operator we suggest consists in applying the operator ρ_k when the two argument automata have a different shape automaton, and to apply the widening operator of the lattice Λ on their matching transitions when the two argument automata have the same shape automaton, as illustrated by Fig. 14.

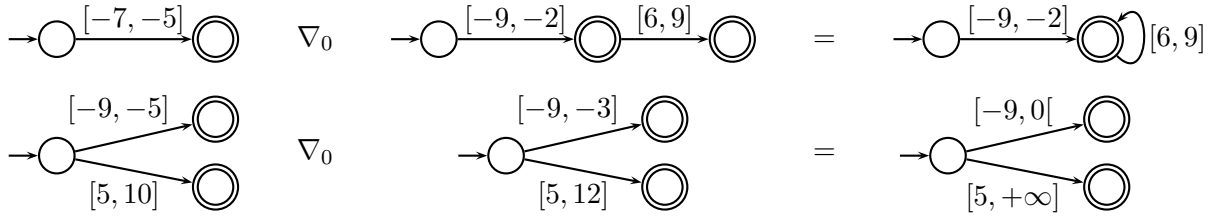


Figure 14: Widening on interval PLA, with a partition $[-\infty, 0] \sqcup [0, +\infty]$ and $k = 0$

Definition 14 (Widening on NLA) Let \mathcal{A}_1 and \mathcal{A}_2 be two NLA defined on the same partition π with $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$. The widening operator ∇_k is defined as :

$$\mathcal{A}_1 \nabla_k \mathcal{A}_2 = \begin{cases} \widehat{\rho_k(\mathcal{A}_2)} & \text{if } \text{shape}(\mathcal{A}_1) \neq \text{shape}(\rho_k(\mathcal{A}_2)) \\ \mathcal{A}_1 \nearrow \mathcal{A}_2 & \text{otherwise (which implies } \text{shape}(\mathcal{A}_1) = \text{shape}(\mathcal{A}_2)) \end{cases}$$

where $\mathcal{A}_1 \nearrow \mathcal{A}_2$ is the NLA \mathcal{A} which has the same set of states as \mathcal{A}_1 and \mathcal{A}_2 and the set of transitions δ defined by the rule:

$$\frac{\sigma \in \Sigma \quad (q, \lambda_1, q') \in \delta_1 \quad (q, \lambda_2, q') \in \delta_2 \quad \lambda_1, \lambda_2 \sqsubseteq \pi(\sigma)}{(q, (\lambda_1 \nabla_\Lambda \lambda_2) \sqcap \pi(\sigma), q') \in \delta}$$

If $\mathcal{A}_1 \not\sqsubseteq \mathcal{A}_2$, then $\mathcal{A}_1 \nabla_k \mathcal{A}_2 \triangleq \mathcal{A}_1 \nabla_k (\mathcal{A}_1 \sqcup \mathcal{A}_2)$.

Notice that with the other hypothesis, the condition $\text{shape}(\mathcal{A}_1) \neq \text{shape}(\rho_k(\mathcal{A}_2))$ is equivalent to $\text{shape}(\mathcal{A}_1) \subsetneq \text{shape}(\rho_k(\mathcal{A}_2))$, and its negation $\text{shape}(\mathcal{A}_1) = \text{shape}(\rho_k(\mathcal{A}_2)) \supseteq \text{shape}(\mathcal{A}_2)$ implies $\text{shape}(\mathcal{A}_1) = \text{shape}(\mathcal{A}_2)$.

Theorem 2 ∇_k is a proper widening operator:

1. for any NLA $\mathcal{A}_1, \mathcal{A}_2$ defined on the same partition such that $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$, $\mathcal{A}_2 \sqsubseteq \mathcal{A}_1 \nabla_k \mathcal{A}_2$;
2. If there is an increasing chain of NLA $\mathcal{A}_0 \sqsubseteq \mathcal{A}_1 \sqsubseteq \dots \sqsubseteq \mathcal{A}_n \sqsubseteq \dots$, the chain $\mathcal{A}'_0 \sqsubseteq \mathcal{A}'_1 \sqsubseteq \dots \sqsubseteq \mathcal{A}'_n \sqsubseteq \dots$ defined as $\mathcal{A}'_0 = \mathcal{A}_0$ and $\mathcal{A}'_{i+1} = \mathcal{A}'_i \nabla_k (\mathcal{A}'_i \sqcup \mathcal{A}_{i+1})$ is not strictly increasing.

Proof:

1. We have $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$ by hypothesis. If $\text{shape}(\mathcal{A}_1) \neq \rho_k(\text{shape}(\mathcal{A}_2)) = \text{shape}(\rho_k(\mathcal{A}_2))$, $\mathcal{A}_1 \nabla_k \mathcal{A}_2 = \widehat{\rho_k(\mathcal{A}_2)} \supseteq \mathcal{A}_2$. If $\text{shape}(\mathcal{A}_1) = \text{shape}(\mathcal{A}_2) = \rho_k(\text{shape}(\mathcal{A}_2))$, $\mathcal{A}_2 \sqsubseteq \mathcal{A}_1 \nearrow \mathcal{A}_2$ because for each pair of matching transitions (q, λ_1, q') and (q, λ_2, q') with $\lambda_1, \lambda_2 \sqsubseteq \pi(\sigma)$, we have $\lambda_2 \sqsubseteq (\lambda_1 \nabla_\Lambda \lambda_2) \sqcap \pi(\sigma)$ (as ∇_Λ is a widening operator on Λ).
2. $(\mathcal{A}'_i)_{i \geq 0}$ is an increasing chain of NLA because of the first property. Thus, $(S'_i = \text{shape}(\mathcal{A}'_i))_{i \geq 0}$ is an increasing chain of finite automata. Moreover, by definition of ∇_k on NLA, $S'_{i+1} = S'_i \nabla_k (S'_i \sqcup S_{i+1}) = \rho_k(S'_i \sqcup S_{i+1})$. As ∇_k as defined on finite automata is a widening operator, the chain $(S'_i)_{i \geq 0}$ becomes stationary at some rank N . The chain $(\mathcal{A}'_i)_{i \geq N}$ is thus an increasing sequence of NLA with identical shape automaton S . For each transition $(q, \sigma, q') \in \delta^S$, one can extract the corresponding transition sequence $(q, \lambda'_i, q') \in \delta^{\mathcal{A}'_i}$ with $\lambda'_i \sqsubseteq \pi(\sigma)$. The sequence $(\lambda'_i)_{i \geq N}$ converges after a finite number of steps, because ∇_Λ is a widening operator. As there is only a finite number of transitions in the shape automaton S , the sequence $(\mathcal{A}'_i)_{i \geq N}$ also converges after a finite number of steps.

□

5.5 NLA as an abstract domain for languages, stacks and queues.

Normalized lattice automata allows to define an *abstract domain functor* which lifts abstract domains for some set to abstract domains for languages on this set. More precisely, given an *atomic* abstract lattice $A \xrightarrow{\gamma_A} \wp(S)$ for some set S with the concretization function γ_A , and a partitioning function π for A , $\text{Reg}(A, \pi)$ can be viewed as an abstract domain for $\mathcal{L}(S) = \wp(S^*)$, the languages on elements of S , with the concretization function

$$\begin{aligned} \gamma : \text{Reg}(A, \pi) &\rightarrow \wp(S^*) \\ \mathcal{A} &\mapsto \{s_0 \dots s_n \in S^* \mid a_0 \dots a_n \in L_{\mathcal{A}} \wedge \forall i : s_i \in \gamma_A(a_i)\} \end{aligned}$$

and the widening operator of Def. 14. $\text{Reg}(A, \pi)$ is a non-complete join semilattice of infinite height.

We have in addition the following monotonicity result for $\text{Reg}(A, \pi)$ seen as a functor.

Theorem 3 *Let $\gamma_i : A_i \rightarrow \wp(S), i = 1, 2$ two abstract domains for $\wp(S)$ such that A_2 refines A_1 , with $\gamma_{12} : A_1 \rightarrow A_2$ such that $\gamma_1 = \gamma_2 \circ \gamma_{12}$ (cf. Sect. 3). Let π_1 a partitioning function for A_1 , and π_2 a partitioning function for A_2 refining $\gamma_{12} \circ \pi_1$. The abstract domain $\text{Reg}(A_2, \pi_2)$ refines $\text{Reg}(A_1, \pi_1)$.*

Proof: Let $\Gamma_i : \text{Reg}(A_i, \pi_i) \rightarrow \wp(S^*)$ the two concretization functions. Let $\mathcal{A}_1 \in \text{Reg}(A_1, \pi_1)$ be a NLA. We can build a NLA $\mathcal{A}_2 \in \text{Reg}(A_2, \pi_2)$ such that $\gamma_2(\mathcal{A}_2) = \gamma_1(\mathcal{A}_1)$.

One first replace in \mathcal{A}_1 transitions labeled by $\lambda_1 \in \pi_1(\sigma_1)$ by transitions labeled by $\lambda_2 = \gamma_{12}(\lambda_1) \in \gamma_{12} \circ \pi_1(\sigma_1)$. The set of elements in S “covered” by the two labels is clearly the same. The resulting automaton \mathcal{A}'_2 thus satisfies $\Gamma_2(\mathcal{A}'_2) = \Gamma_1(\mathcal{A}_1)$. Moreover, \mathcal{A}'_2 is a NLA belonging to $\text{Reg}(A_2, \gamma_{12} \circ \pi_2)$. One now define \mathcal{A}_2 as the refinement of \mathcal{A}'_2 w.r.t. the refined partition π_2 , which recognizes the same language according to Prop. 4.

This transformation from \mathcal{A}_1 to \mathcal{A}_2 defines a function $\Gamma_{12} : \text{Reg}(A_1, \pi_1) \rightarrow \text{Reg}(A_2, \pi_2)$ such that $\Gamma_1 = \Gamma_2 \circ \Gamma_{12}$. \square

Most languages operations can be efficiently abstracted in $\text{Reg}(A, \pi)$. This is in particular the case of languages operations corresponding to the operations offered by the *FIFO queue* or *stack* abstract datatypes. Hence, $\text{Reg}(A, \pi)$ is a suitable abstract domain for FIFO queues or stacks on elements of S .

6 Application to the abstract interpretation of SCM

We illustrate in this section the application of the abstract domain $\text{Reg}(A, \pi)$ defined in the previous section to the analysis of symbolic (or extended) Communicating Machines. This application is actually the initial motivation for the study of lattice automata.

6.1 Symbolic Communicating Machines

Symbolic Communicating Machines (SCM) are Communicating Finite-State Machine extended with a finite set of variables V , the values of which can be sent into FIFO queues, cf. Fig. 1. A transition is triggered when a condition on the value of the variables is satisfied. In such a case, the transition can first emit or receive values from a FIFO queue, then modify the values of variables, and last make the control jump to the destination location. This model is similar to other models like Extended Communicating Finite-State Machines [26] or Parametrized Communicating Extended Finite-State Machines [34].

Definition 15 *A SCM with N queues is defined by a tuple $\langle C, V, c_0, \Theta_0, P, \Delta \rangle$ where :*

- C is a nonempty finite set of locations (control states).
- $V = \{v_1, \dots, v_n\}$ is a nonempty, finite set of variables. The domain of values of a variable v is denoted by \mathcal{D}_v , and the set of valuations of all variables in V by \mathcal{D}_V .
- $c_0 \in C$ is the initial control state, and $\Theta_0 \subseteq \mathcal{D}_V$, a predicate on V , is the initial condition.
- $P = \{p_1, \dots, p_n\}$ is a nonempty, finite set of formal parameters that are used to send/receive values to/from FIFO queues. We assume that all queues use the same set of parameters \mathcal{D}_P .
- Δ is a finite set of transitions. A transition δ is either an input $\langle c_1, G, i?\vec{p}, A, c_2 \rangle$ or an output $\langle c_1, G, i!\vec{p}, A, c_2 \rangle$ where :
 1. c_1 and c_2 are resp. the origin and destination locations;
 2. $i \in [1..N]$ is a queue number
 3. \vec{p} is the vector of formal parameters, which holds the values sent or received to/from the queue i ;
 4. $G(\vec{v}, \vec{p}) \subseteq \mathcal{D}_V \times \mathcal{D}_P$ is a predicate on the variables and the formal parameters (also called guard).
 5. A is an assignment of the form $\vec{v}' := A(\vec{v}, \vec{p})$, where $A : \mathcal{D}_V \times \mathcal{D}_P \rightarrow \mathcal{D}_V$, which defines the values of the variables after the transition.

Compared to the definition of *Extended Communicating Finite-State Machines* [26], the values sent to queues may be of any type. Indeed, our model is closer to the *Parametrized Communicating Extended Finite-State Machines* model [34] (although there is no major difference between the three models). As a consequence, the alphabet of the queue contents is \mathcal{D}_P . This alphabet is not finite if \mathcal{D}_P is not finite.

The operational semantics of a SCM $\langle C, V, c_0, \Theta_0, P, \Delta \rangle$ is given as an infinite transition system $\langle Q, Q_0, \rightarrow \rangle$ where

- $Q = C \times \mathcal{D}_V \times ((\mathcal{D}_P)^*)^N$ is the set of states;
- $Q_0 = \{\langle c_0, \vec{v}, \varepsilon, \dots, \varepsilon \rangle \mid \vec{v} \in \Theta_0\}$ is the set of initial states;
- \rightarrow is defined by the two rules:

$$\frac{(c_1, G, i!\vec{p}, A, c_2) \in \Delta \quad w'_i = w_i \cdot \vec{p} \quad G(\vec{v}, \vec{p}) \quad \vec{v}' = A(\vec{v}, \vec{p})}{\langle c_1, \vec{v}, w_1, \dots, w_i, \dots, w_N \rangle \rightarrow \langle c_2, \vec{v}', w_1, \dots, w'_i, \dots, w_N \rangle}$$

$$\frac{(c_1, G, i?\vec{p}, A, c_2) \in \Delta \quad w_i = \vec{p}.w'_i \quad G(\vec{v}, \vec{p}) \quad \vec{v}' = A(\vec{v}, \vec{p})}{\langle c_1, \vec{v}, w_1, \dots, w_i, \dots, w_N \rangle \rightarrow \langle c_2, \vec{v}', w_1, \dots, w'_i, \dots, w_N \rangle}$$

A global state of a SCM is thus a tuple $\langle c, \vec{v}, w_1, \dots, w_N \rangle \in C \times \mathcal{D}_V \times (\mathcal{D}_P)^* \times \dots \times (\mathcal{D}_P)^*$ where c is a control state, \vec{v} is the current value of the variables and w_i is a finite word on \mathcal{D}_P representing the content of queue i .

The concrete collecting semantics of a SCM depicted on Tab. 4 is deduced from the operational semantics as it was done in Sect. 4.2. It is a bit more complex because the model involves symbolic operations.

6.2 SCM with a single queue: a simple approach

There are classical solution for the abstraction of scalar variables. Consequently, the main issue of applying abstract interpretation techniques on this model is the abstraction of the queue contents. Whereas the queue contents in the CFSM model were abstracted by regular languages, queue contents in the SCM model will be abstracted using the abstract domain of lattice automata.

If there is a single queue, the concrete set of states associated to each control point $c \in C$ has the structure $\wp(\mathcal{D}_V \times (\mathcal{D}_P)^*)$: one associates to each control point the set of possible configurations for the variables of the communicating machine and its FIFO queue

The concrete lattice $\wp(\mathcal{D}_V \times (\mathcal{D}_P)^*)$ can be abstracted with $\wp(\mathcal{D}_V) \times \mathcal{L}(\mathcal{D}_P)$ by projecting the two components. This allows to abstract $\wp(\mathcal{D}_V)$ using classical abstractions for variables of the environment, and to use lattice automata for abstracting the domain of queue contents $\mathcal{L}(\mathcal{D}_P)$.

For the sake of simplicity, and as an example, we will assume in the sequel that all variables and parameters are of rational type, and that sets of valuations are abstracted using the lattice of convex polyhedra $V^{(k)} = \text{Pol}(\mathbb{Q}^k)$. We have the abstraction

$$\wp(\mathbb{Q}^n \times (\mathbb{Q}^p)^*) \iff \wp(\mathbb{Q}^n) \times \mathcal{L}(\mathbb{Q}^p) \longleftarrow V^{(n)} \times \text{Reg}(V^{(p)})$$

As the atoms of the lattice $V^{(p)} = \text{Pol}(\mathbb{Q}^p)$ are precisely the elements of \mathbb{Q}^p , the concretization function of $\text{Reg}(V^{(p)})$ is just the identity. The induced abstract semantics is given on Tab. 5.

Example. We consider the protocol presented in the introduction (Fig. 1). For the purpose of the analysis, we make the asynchronous product of the two automata. The result is a control structure with four states, labeled 00, 01, 10 and 11, with the following convention : the SCM is in the “xy” state if the sender is in the “x” state and the receiver is in the “y” one.

The example was modeled by a system of equation and we solved the reachability analysis using a generic fix-point calculator. We used the polyhedra abstract lattice as implemented in the APRON library[1]. No partitioning of the alphabet lattice was employed in this example. We obtained the result of Tab. 6. \square

The result of this analysis is disappointing: one is not able to prove that the messages contained in the queues are indexed by integers that are lower than the variable s . This is not due to the queue abstraction, nor due to the variables abstraction, but to the coupling of the two abstractions. The following section proposes a solution to this problem..

6.3 SCM with a single queue: linking message and state variables

The idea to improve on the previous abstraction is to use an augmented semantics to link the message variables contained in queues with the state variables of the machines.

The proposal of this section is to put into queues not only the queue content itself, but also (a subset of) the environment. This allows not only to establish relations between messages in queues and the current environment, but also to indirectly establish relations between the messages contained in different letters. For instance, the abstract value

$$\left(s \in [8, 9], \text{data}(\{s - p = 3\}) \cdot \text{data}(\{s - p = 2\}) \cdot \text{data}(\{s - p = 1\}) \right)$$

will represent the 2 concrete states ($s = 8, \text{data}(5) \cdot \text{data}(6) \cdot \text{data}(7)$) and ($s = 9, \text{data}(6) \cdot \text{data}(7) \cdot \text{data}(8)$). This is to be compared with the standard abstraction of these 2 states:

$$\left(s \in [8, 9], \text{data}(p \in [5, 6]) \right) \cdot \text{data}(p \in [6, 7]) \cdot \text{data}(p \in [7, 8])$$

which represents $2^4 = 16$ concrete states.

Parameterized Domains with $k \geq 0$	
Concrete domain	$C^{(k)} = \wp(\mathbb{Q}^k \times (\mathbb{Q}^p)^*)$
Abstract domain for variables	$V^{(k)} = \text{Pol}(\mathbb{Q}^k)$
Abstract domain for one FIFO queue	$L = \text{Reg}(V^{(p)})$
Abstract domain	$A^{(k)} = V^{(k)} \times L$

Table 3: Semantic domains

Concrete collecting semantics		
guard	$G(\vec{v}, \vec{p})$	$\llbracket G \rrbracket : \wp(\mathbb{Q}^n) \rightarrow \wp(\mathbb{Q}^{n+p})$ by extension $\llbracket G \rrbracket : C^{(n)} \rightarrow C^{(n+p)}$
assignment	$A(\vec{v}, \vec{p})$	$\llbracket A \rrbracket : \wp(\mathbb{Q}^{n+p}) \rightarrow \wp(\mathbb{Q}^n)$ by extension $\llbracket A \rrbracket : C^{(n+p)} \rightarrow C^{(n)}$
output	$1!\vec{p}$	$\llbracket 1!\vec{p} \rrbracket : C^{(n+p)} \rightarrow C^{(n+p)}$ $X \mapsto \{(\vec{v}, \vec{p}, \omega \cdot \vec{p}) \mid (\vec{v}, \vec{p}, \omega) \in X\}$
input	$1?\vec{p}$	$\llbracket 1?\vec{p} \rrbracket : C^{(n+p)} \rightarrow C^{(n+p)}$ $X \mapsto \{(\vec{v}, \vec{p}, \omega) \mid (\vec{v}, \vec{p}, \vec{p} \cdot \omega) \in X\}$
transition	t	$\llbracket t \rrbracket : C^{(n)} \rightarrow C^{(n)}$ $X \mapsto \begin{cases} \llbracket A \rrbracket \circ \llbracket G \rrbracket & \text{if } t = (G, --, A) \\ \llbracket A \rrbracket \circ \llbracket 1!\vec{p} \rrbracket \circ \llbracket G \rrbracket & \text{if } t = (G, 1!\vec{p}, A) \\ \llbracket A \rrbracket \circ \llbracket 1?\vec{p} \rrbracket \circ \llbracket G \rrbracket & \text{if } t = (G, 1?\vec{p}, A) \end{cases}$

The semantics of a transition is defined as follows: first the queue variables \vec{p} are introduced, and then constrained by the guard (semantics of guards). A push or pop operation is possibly performed. Last, the assignment updates the value of state variables. For push and pop, the value \vec{p} is defined by a kind of unification between the guard and the queue content.

Table 4: Concrete collecting semantics

Standard abstract semantics		
guard	$G(\vec{v}, \vec{p})$	$\llbracket G \rrbracket^\sharp : V^{(n)} \rightarrow V^{(n+p)}$ by extension $\llbracket G \rrbracket^\sharp : A^{(n)} \rightarrow A^{(n+p)}$
assignment	$A(\vec{v}, \vec{p})$	$\llbracket A \rrbracket^\sharp : V^{(n+p)} \rightarrow V^{(n)}$ by extension $\llbracket A \rrbracket^\sharp : A^{(n+p)} \rightarrow A^{(n)}$
output	$1!\vec{p}$	$\llbracket 1!\vec{p} \rrbracket^\sharp : V^{(n+p)} \times L \rightarrow V^{(n+p)} \times L$ $(Y, F) \mapsto (Y, F \cdot (\exists \vec{v} : Y))$
input	$1?\vec{p}$	$\llbracket 1?\vec{p} \rrbracket^\sharp : V^{(n+p)} \times L \rightarrow V^{(n+p)} \times L$ $(Y, F) \mapsto \bigsqcup_{\sigma} (Y \sqcap \text{Embed} \circ \text{first}(F)(\sigma), (F/(\exists \vec{v} : Y))(\sigma))$ with $\text{Embed} : V^{(p)} \rightarrow V^{(n+p)}$ a canonical embedding function
transition	t	$\llbracket t \rrbracket^\sharp : A^{(n)} \rightarrow A^{(n)}$ $X \mapsto \begin{cases} \llbracket A \rrbracket^\sharp \circ \llbracket G \rrbracket^\sharp & \text{if } t = (G, --, A) \\ \llbracket A \rrbracket^\sharp \circ \llbracket 1!\vec{p} \rrbracket^\sharp \circ \llbracket G \rrbracket^\sharp & \text{if } t = (G, 1!\vec{p}, A) \\ \llbracket A \rrbracket^\sharp \circ \llbracket 1?\vec{p} \rrbracket^\sharp \circ \llbracket G \rrbracket^\sharp & \text{if } t = (G, 1?\vec{p}, A) \end{cases}$

Table 5: Standard abstract semantics

Non-standard abstract semantics. The formalization of this technique requires slightly heavier notations. We assume that we put all the environment in the queue. The abstract lattice is then

$$A^{(n)} = V^{(n)} \times L \quad \text{with} \quad L = \text{Reg}(V^{(n+p)})$$

with the concretization function

$$\gamma(Y, F) = \{(\vec{v}, \omega = \omega_0 \dots \omega_k) \in \mathbb{Q}^n \times (\mathbb{Q}^p)^* \mid \vec{v} \in Y \wedge (\vec{v}, \omega_0) \dots (\vec{v}, \omega_k) \in F\}$$

Control	Abstract Value
00	$\llbracket s \geq 0; a \geq 0 \rrbracket$ $(d, \llbracket p \geq 0 \rrbracket)^*$ $(a, \llbracket p \geq 0 \rrbracket)^*$
01	$\llbracket v \geq 0; s \geq 0; a \geq 0 \rrbracket$ $(d, \llbracket p \geq 0 \rrbracket)^*$ $(a, \llbracket p \geq 0 \rrbracket)^*$
10	$\llbracket s \geq 0; a - 2 \geq 0 \rrbracket$ $(d, \llbracket p \geq 0 \rrbracket)^*$ $(a, \llbracket p \geq 0 \rrbracket)^*$
11	$\llbracket v \geq 0; s \geq 0; a - 2 \geq 0 \rrbracket$ $(d, \llbracket p \geq 0 \rrbracket)^*$ $(a, \llbracket p \geq 0 \rrbracket)^*$

Table 6: Analysis of the example with the standard approach

Control	Abstract Value
00	$\llbracket 0 \leq a \leq s \leq a + 10 \rrbracket$ $(d, \llbracket 0 \leq a \leq s - 1 \leq a + 9; 0 \leq p \leq s - 1 \rrbracket)^*$ $(a, \llbracket 0 \leq a \leq s \leq a + 10; 0 \leq p \leq s - 1; 0 \leq v \leq s - 1 \rrbracket)^*$
01	$\llbracket 0 \leq a \leq s \leq a + 10; 0 \leq v \leq s - 1 \rrbracket$ $(d, \llbracket 0 \leq a \leq s - 1 \leq a + 9; 0 \leq p \leq s - 1 \rrbracket)^*$ $(a, \llbracket 0 \leq a \leq s \leq a + 10; 0 \leq p \leq s - 1; 0 \leq v \leq s - 1 \rrbracket)^*$
10	$\llbracket 0 \leq a \leq s - 3 \leq a + 7; 0 \leq v \leq s - 1 \rrbracket$ $(d, \llbracket 0 \leq a \leq s - 1 \leq a + 9; 0 \leq p \leq s - 1 \rrbracket)^*$ $(a, \llbracket 0 \leq a \leq s \leq a + 10; 0 \leq p \leq s - 1; 0 \leq v \leq s - 1 \rrbracket)^*$
11	$\llbracket 0 \leq a \leq s - 3 \leq a + 7; 0 \leq v \leq s - 1 \rrbracket$ $(d, \llbracket 0 \leq a \leq s - 1 \leq a + 10; 0 \leq p \leq s - 1 \rrbracket)^*$ $(a, \llbracket 0 \leq a \leq s \leq a + 10; 0 \leq p \leq s - 1; 0 \leq v \leq s - 1 \rrbracket)^*$

Table 7: Analysis of the example with the non-standard approach

The partitioning function $\pi : \Sigma \rightarrow V^{(p)}$ used in PLA is implicitly extended to $\pi : \Sigma \rightarrow V^{(n+p)}$. $A^{(n)}$ may be seen as a reduced product of the two interacting components $V^{(n)}$ and $\text{Reg}(V^{(n+p)})$.

Abstract operations. The abstract semantics of guards now involves both the abstract environment and the queue:

$$\begin{aligned} \llbracket G \rrbracket^r : \quad & V^{(n)} \times \text{Reg}(V^{(n+p)}) \rightarrow V^{(n+p)} \times \text{Reg}(V^{(n+p)}) \\ & (Y, F = \langle \Lambda, \pi, Q, Q_0, Q_f, \delta \rangle) \mapsto (\llbracket G \rrbracket^\sharp(Y), F' = \langle \Lambda, \pi, Q, Q_0, Q_f, \delta' \rangle) \end{aligned}$$

with $\delta' = \{ (q_1, \lambda', q_2) \mid (q_1, \lambda, q_2) \in \delta \wedge \lambda' = \lambda \cap (\exists \vec{p}' : G) \}$.

The abstract semantics of assignments is more complex:

$$\begin{aligned} \llbracket A \rrbracket^r : \quad & V^{(n+p)} \times \text{Reg}(V^{(n+p)}) \rightarrow V^{(n)} \times \text{Reg}(V^{(n+p)}) \\ & (Y, F = \langle \Lambda, \pi, Q, Q_0, Q_f, \delta \rangle) \mapsto (Y', F' = \langle \Lambda, \pi, Q, Q_0, Q_f, \delta' \rangle) \end{aligned}$$

where $Y' = \llbracket A \rrbracket^\sharp(Y)$ and $\delta' = \{ (q_1, \lambda', q_2) \mid (q_1, \lambda, q_2) \in \delta \wedge \lambda' \text{ defined as below } \}$. $\lambda'(\vec{v}, \vec{p}_0)$ is obtained from $\lambda(\vec{v}, \vec{p}_0)$ by the following operations:

We build $\phi(\vec{v}, \vec{p}, \vec{v}', \vec{p}_0) = \lambda(\vec{v}, \vec{p}_0) \wedge v' = A(\vec{v}, \vec{p}) \wedge Y(\vec{v}, \vec{p})$, where \vec{v}' is the value of the state variables after the assignment, \vec{v} and \vec{p} the current value of state and message variables, and \vec{p}_0 the value of the message variable in the transition of the lattice automaton.

We then build $\phi'(\vec{v}', \vec{p}_0) = \exists \vec{v} \exists \vec{p} : \phi$ by eliminating variables in the current environment;

Last we perform a renaming: $\lambda'(\vec{v}, \vec{p}_0) = \phi'(\vec{v}', \vec{p}_0)[\vec{v} \leftarrow \vec{v}']$

The application of the assignment to a NLA produces a NLA; in particular, as the partitioning function involves only the message parameters, and $\llbracket A \rrbracket^\sharp$ modifies only the state variables, the lattice automaton remains well-partitioned.

The semantics of message outputs and inputs are in some way simpler than with the previous abstract semantics:

$$\begin{aligned} \llbracket !\vec{p} \rrbracket^r(Y, F) &= (Y, F \cdot Y) \\ \llbracket 1?\vec{p} \rrbracket^r(Y, F) &= (Y \sqcap \text{first}(F), \widehat{F/Y}) \end{aligned}$$

The semantics of transitions remains similar to that of Tab. 5.

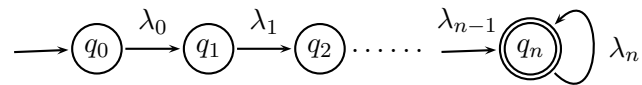
Example. We performed a new analysis based on these non-standard semantics on the same example as before. A ‘widening up to’ operator was used instead of the standard widening on polyhedra [25]. We obtained the better result of Tab. 7. \square

This analysis is quite accurate, since it shows that :

- $0 \leq a \leq s \leq a + 10$
- $0 \leq p \leq s - 1$
- $0 \leq v \leq s - 1$

Remark 5 (Efficiency of the approach.) *It should be noted that the abstraction of this section is much more expensive than the abstraction of Sect. 6.2, because guards and assignments do not apply only on the abstract value representing state variables, but also the abstract values labeling the transitions of the automata. This suggests the use of a small partition π of the lattice Λ and the use of a small parameter k in the widening, in order to keep the lattice automata reasonably small. However, the gain in precision brought by this approach worth it, although experimental results on a significative set of examples is still lacking.*

Remark 6 (The particular case of a partition of size 1) *In our analysis examples, we did not exploit a partition of the alphabet lattice. In such a case, normalized lattice automata have a very simple, linear structure of the form:*



If such an automaton is the normalized image of the widening operator with parameter k , then $n \leq k + 2$.

In general, the most natural choice for the analysis of SCM is to partition the alphabet according to the different kinds of messages exchanged (transmission, retransmission, acknowledgment, ...).

6.4 SCM with several queues

As in Sect. 4.3, there are mainly two ways to analyze systems with several queues. One can follow :

- a non-relational, attribute-independent method, in which an abstract configuration is defined by a polyhedron representing the value of the state variables and N lattice automata, each representing a queue content,
- or a relational, attribute-dependent method, in which a single lattice automaton recognizes concatenated words $w = w_1\#w_2\#\dots\#w_N$, where each subword w_i represents the content of the queue i and $\#$ is a special separating character [4].

The extension to lattice automata of these two variants discussed in Sect. 4.3 for finite automata is straightforward. In the previous subsections, the analyzed example, which has two FIFO queues, was implicitly analyzed with a non-relational approach.

$Prog = (P_i)_{0 \leq i < n}$: A program is defined by a set of procedures
$LVar_i, \vec{l}_i$: Set of local variables of procedure P_i
\vec{fp}_i : Tuple of formal call parameters of procedure P_i $\vec{fp}_i \subseteq LVar_i$
\vec{fr}_i : Tuple of formal return parameters of procedure P_i $\vec{fr}_i \subseteq LVar_i$
$G_i = \langle K_i, I_i \rangle$: Flow graph of the procedure P_i
K_i : Control points of the procedure P_i
$s_i, e_i \in K_i$: start and exit points of procedure P_i

Table 8: Syntactic domains for imperative programs

$v \in Value$: values of expressions and variables
$\epsilon_i \in LEnv_i = LVar_i \rightarrow Value$: local environments for procedure/function P_i
$\epsilon \in LEnv = \bigcup_i LEnv_i$: local environments for any procedure/function
$\langle c, \epsilon \rangle \in Act = K \times LEnv$: activation record
$\Gamma \in Act^*$: stacks (sequences) of activation records = program states

Table 9: Semantic domains for imperative programs

7 Application to interprocedural analysis

The analysis of communicating machines was the initial motivation for the study of lattice automata in Sect. 5. However there is an interesting application in precise interprocedural analysis, where one can use lattice automata for abstracting call-stacks. This would allow both to simplify and to improve the abstraction proposed in [29]. One can also see this application as an extension to infinite state programs of [19] which uses pushdown automata to model finite-state recursive programs and finite automata for representing (co)reachable sets of configurations.

Program model. Tab. 8 sums up our notations and assumptions on recursive programs. Notice that we assume no global variables, and parameter passing by value. We use a symmetrical notion of effective and formal call parameters for return parameters in an instruction $(y_1, y_2) = P(x_1, x_2)$ where y_1, y_2 are effective return parameters assigned from formal return parameters \vec{fr} in P . To integrate the call-string and functional approach, we make an important assumption, which is that formal parameters are not modified in a procedure. This is not restrictive, as one can always copy such variables into local variables. Tab. 9 defines the semantic domains used for these programs. We assume that the top of a stack $\Gamma \in Act^+$ is on the left side of the word Γ , but the other convention is also possible.⁵

We do not detail more the program model, as we will focus on the abstraction of call-stacks and we will not detail the induced abstraction of the instructions and procedure calls and returns.

Well-formed call-stacks. Reachable call-stacks should reflect the equality of effective parameters in a caller procedure and formal callee parameters in the callee procedure. Thus *well-formed* call-stacks $\Gamma = \langle c_n, \epsilon_n \rangle \dots \langle c_1, \epsilon_1 \rangle \dots \langle c_0, \epsilon_0 \rangle$ satisfy : $\forall i < n : \epsilon_{i+1}(\vec{fp}) = \epsilon_i(\vec{x})$ where \vec{x} denotes the effective parameters at the call point c_i and \vec{fp} the formal call parameters in the corresponding callee procedure. We will implicitly focus on such *well-formed* stacks.

Using lattice automata for interprocedural analysis. In the concrete semantics of programs, it is standard to partition the state-space according to the current control point. Thus, we use the identity $Act^+ = K \rightarrow LEnv \times Act^*$. Now, given an abstraction $\wp(LEnv) \iff \Lambda$ for environments, we

⁵This choice is important insofar as the lattice automata abstraction does not commute with the reverse operation on words.

can use the following abstraction:

$$\wp(\text{Act}^+) \xleftrightarrow[\alpha]{\gamma} K \rightarrow \Lambda \times \text{Reg}(K \times \Lambda)$$

The most natural partitioning for the lattice $K \times \Lambda$ is of course the finite partitioning function $\pi(k) = \{k\} \times \top_\Lambda$.

In the forward abstract (operational) semantics induced by this abstraction, an intraprocedural instruction modifies only Y in an abstract configuration $(Y, \Gamma) \in \Lambda \times \text{Reg}(K \times \Lambda)$. A procedure call from an abstract configuration (Y, F) at point c results in a configuration of the form $(Y', (\{c\}, Y) \cdot F)$ at start point s of the callee. A procedure return from an abstract configuration (Y, F) at exit point e to the return point c consists in computing $Y' = \text{first}(F)(c)$ and in modifying it using the return value in Y . This is where it is useful to exploit the well-formedness of call-stacks, by unifying Y' with Y using the equality $Y(\vec{fp}) = Y'(\vec{x})$ before assigning the formal return parameters in Y to the effective return parameters in Y' .

One can define similarly a backward abstract semantics, and one can also intersect the resulting backward analysis with the result of a forward analysis, as described in [29]

Interprocedural analysis by explicit representation of the call-stacks. The call-string approach of [47], generalized by tokens in [31] corresponds to a very strong abstraction of the call-stacks in which the value of variables is largely ignored. It is more suitable to compilation-oriented dataflow analysis than to program verification. A different line of work consists in modeling recursive programs using pushdown automata, and exploiting the property that the set of (co)reachable stacks of pushdown automata is regular [13] and its computation has a polynomial complexity [18, 7, 22]. The application of this approach to verification is mainly restricted either to programs manipulating finite-state variables, or to programs first abstracted to such programs [19], although [43] removes some restrictions using weights on transitions. The approach sketched in this section can be seen as an extension of this line of work to more expressive programs, where undecidability is overcome by resorting to approximations.

The lattice automata abstraction may be seen as both an improvement and a simplification of [29], which derives and unifies two classical techniques for interprocedural analysis (namely the call-string and functional approaches identified by [47]) by abstract interpretation of the operational semantics of imperative programs. Compared to the lattice automata abstraction, the abstract domain is roughly a set of environments labeled with call-strings: $K^* \rightarrow \Lambda$ (instead of $\text{Reg}(K \times \Lambda)$). The concretization function uses again the well-formedness property to rebuild stacks from such abstract values. In the context of interprocedural shape analysis, [44] represents explicitly the call-stack using the same abstract representation than for the memory configurations.

There are some advantages in having an explicit representation of call-stacks in interprocedural analysis. As long as data variables are not abstracted, the more classical functional approach is as precise, but as soon as they are, such an explicit representation allows to recover some loss of information. Moreover, the stack abstraction approach allows to describe naturally transformations such as procedures inlining in the course of an analysis. This is discussed in more details in [29]. Last, some applications *require* information on the stack. This is the case for instance of analysis related to stack inspection mechanisms for ensuring security properties in JAVA and .NET architecture [3]. Another example is the test selection technique proposed in [14], in the context of conformance testing of reactive systems w.r.t. an interprocedural specification.

8 Conclusion

We defined in this paper an abstract domain for languages on infinite alphabets, which are represented by *lattice automata*. Our main motivation was the analysis of symbolic (or extended) communicating

machines which exchange infinite data through FIFO queues, but this abstract domain may also be used for abstracting call-stacks in the analysis of recursive programs.

The principle of lattice automata is to use elements of an atomic lattice for labeling the transitions of a finite automaton, and to use a partition of this lattice in order to define a projected finite automaton which acts as a guide for defining determinization, minimization and widening operations. We motivate our choices and show that they lead to a robust notion of approximation, in the sense that normalization is an upper-closure operation and can be seen as a best upper-approximation in the join semilattice of normalized lattice automata.

The resulting abstract domain allows to lift any atomic abstract domain A for $\wp(S)$ to an abstract domain $\text{Reg}(A)$ for $\wp(S^*)$. It is also *parametric*: it is parametrized first by the underlying alphabet lattice, then by a partition of the alphabet lattice, and last by the parameter of the widening operator. Its precision may be improved by adjusting these parameters.

We illustrate the use of lattice automata for the verification of symbolic communicating machines, and we show the need for a non-standard semantics to couple the abstraction of the state variables of the machines with the contents of the FIFO queues. To our knowledge, this is the first technique able to prove the specified properties on our example without manual transformation of the model, despite its relative simplicity. We also explore the applicability of lattice automata to interprocedural analysis and compare this solution to related work.

As a result, this work extends both analysis techniques dedicated to communicating machines, and interprocedural analysis based on an explicit representation of the call-stacks.

Future work includes first a deeper study of the experimental relevance of lattice automata to the analysis of SCM. The challenge we would like to take up is the verification of the SSCOP communication protocol, which is a sliding window protocol from which our running example is extracted. Previous verification attempts that we are aware of are either based on enumerated state-space exploration techniques [10], or on the partial use of theorem proving [45]. It would be interesting to study the application of lattice automata to shape analysis, in the spirit of [9]. A last direction which could be explored is the generalization of lattice automata recognizing languages on infinite alphabets to tree automata recognizing trees on infinite sets of symbols.

References

- [1] The APRON numerical abstract domain library. <http://apron.cri.enscm.fr/library/>.
- [2] Sébastien Bardin, Alain Finkel, Jérôme Leroux, and Laure Petrucci. FAST: Fast Acceleration of Symbolic Transition systems. In *Computer Aided Verification (CAV'03)*, volume 2725 of *LNCS*, July 2003.
- [3] Frederic Besson, Thomas Jensen, Daniel Le Métayer, and Tommy Thorn. Model checking security properties of control flow graphs. *Journal of Computer Security*, 9, 2001.
- [4] Bernard Boigelot and Patrice Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs. *Formal Methods in System Design*, 14(3), 1997.
- [5] Bernard Boigelot, Patrice Godefroid, Bernard Willems, and Pierre Wolper. The power of QDDs (extended abstract). In *Static Analysis Symposium, SAS'97*, 1997.
- [6] Mikolaj Bojanczyk, Anca Muscholl, Thomas Schwentick, Luc Segoufin, and Claire David. Two-variable logic on words with data. In *Symposium on Logic in Computer Science, LICS '06*, 2006.

- [7] Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata: Application to model checking. In *Int. Conf. on Concurrency Theory, CONCUR'97*, volume 1243 of *LNCS*, 1997.
- [8] Ahmed Bouajjani and Peter Habermehl. Symbolic reachability analysis of FIFO-channel systems with nonregular sets of configurations. *Theoretical Computer Science*, 221(1-2), 1999.
- [9] Ahmed Bouajjani, Peter Habermehl, Adam Rogalewicz, and Thomas Vojnar. Abstract tree regular model checking of complex dynamic data structures. In *Static Analysis Symposium, SAS'06*, volume 4218 of *LNCS*, 2006.
- [10] Marius Bozga, Jean-Claude Fernandez, Lucian Ghirvu, Claude Jard, Thierry Jéron, Alain Kerbrat, Pierre Morel, and Laurent Mounier. Verification and test generation for the SSCOP protocol. *Scientific Computer Programming*, 36(1), 2000.
- [11] Daniel Brand and Pitro Zafropulo. On communicating finite-state machines. *Journal of ACM*, 30(2), 1983.
- [12] Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4), 1964.
- [13] Didier Caucal. On the regular structure of prefix rewriting. *Theoretical Computer Science*, 106, 1992.
- [14] Camille Constant, Bertrand Jeannet, and Thierry Jéron. Automatic test generation from interprocedural specifications. Technical Report PI 1835, IRISA, 2007. Submitted to TESTCOM/FATES conference.
- [15] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages, POPL '77*, 1977.
- [16] Patrick Cousot and Radhia Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *Symposium on Programming Language Implementation and Logic Programming*, 1992.
- [17] Dino Distefano, Peter O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS'06*, volume 3920 of *LNCS*, 2006.
- [18] Javier Esparza and Jens Knoop. An automata-theoretic approach to interprocedural data-flow analysis. In *Int. Conf. on Foundations of Software Science and Computation Structure (FoSSaCS '99)*, volume 1578 of *LNCS*, March 1999.
- [19] Javier Esparza and Stefan Schwoon. A BDD-based model checker for recursive programs. In *Computer Aided Verification, CAV'01*, volume 2102 of *LNCS*, 2001.
- [20] Jérôme Feret. Abstract interpretation-based static analysis of mobile ambients. In *Eighth Int. Static Analysis Symposium (SAS'01)*, number 2126 in *LNCS*, 2001.
- [21] Alain Finkel, S. Purushothaman Iyer, and Grégoire Sutre. Well-abstracted transition systems: application to FIFO automata. *Information and Computation*, 181(1), 2003.
- [22] Alain Finkel, Bernard Willems, and Pierre Wolper. A direct symbolic approach to model checking pushdown systems. *Electronic Notes on Theoretical Computer Science*, 9, 1997.

- [23] Tristan Le Gall, Bertrand Jeannet, and Thierry Jéron. Verification of communication protocols using abstract interpretation of FIFO queues. In *Algebraic Methodology and Software Technology, AMAST '06*, volume 4019 of *LNCS*, July 2006.
- [24] Denis Gopan, Frank DiMaio, Nurit Dor, Thomas Reps, and Mooly Sagiv. Numeric domains with summarized dimensions. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS'04*, volume 2988 of *LNCS*, 2004.
- [25] Nicolas Halbwachs, Yann-Erick Proy, and Patrick Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2), August 1997.
- [26] Masahiro Higuchi, Osamu Shirakawa, Hiroyuki Seki, Mamoru Fujii, and Tadao Kasami. A verification procedure via invariant for extended communicating finite-state machines. In *Computer Aided Verification, CAV '92*, volume 663 of *LNCS*, 1993.
- [27] ITU-TS. *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC)*, 1999.
- [28] Bertrand Jeannet. Dynamic partitioning in linear relation analysis. application to the verification of reactive systems. *Formal Methods in System Design*, 23(1), July 2003.
- [29] Bertrand Jeannet and Wedelin Serwe. Abstracting call-stacks for interprocedural verification of imperative programs. In *Int. Conf. on Algebraic Methodology and Software Technology, AMAST'04*, volume 3116 of *LNCS*, July 2004.
- [30] Thierry Jéron, Hervé Marchand, and Vlad Rusu. Symbolic determinisation of extended automata. In *IFIP Int. Conf. on Theoretical Computer Science*, IFIP book series, 2006.
- [31] Neil D. Jones and Steven S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Symposium on Principles of Programming Languages (POPL'82)*, 1982.
- [32] Michael Kaminski and Nissim Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2), 1994.
- [33] Michael Karr. Affine relationships among variables of a program. *Acta Informatica*, 6, 1976.
- [34] David Lee, K. K. Ramakrishnan, W. Melody Moh, and Udaya Shankar. Protocol specification using parameterized communicating extended finite state machines - a case study of the atm abr rate control scheme. In *Int. Conf. on Network Protocols (ICNP '96)*, 1996.
- [35] Laurent Mauborgne. *Representation of sets of trees for abstract interpretation*. PhD thesis, École Polytechnique, 1999.
- [36] Laurent Mauborgne. Tree schemata and fair termination. In *Static Analysis Symposium, SAS'00*, volume 1824 of *LNCS*, 2000.
- [37] Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. In *European Symposium on Programming, ESOP'05*, volume 3444 of *LNCS*, 2005.
- [38] Tova Milo, Dan Suciu, and Victor Vianu. Typechecking for XML transformers. In *Symposium on Principles of Database Systems*, 2000.
- [39] Frank Neven, Thomas Schwentick, and Victor Vianu. Towards regular languages over infinite alphabets. volume 2136 of *LNCS*, 2001.
- [40] Mogens Nielsen, Gordon Plotkin, and Glynn Winskel. Petri nets, event structures and domains, part 1. *Theoretical Computer Science*, 13, 1981.

- [41] Wuxu Peng and S. Puroshothaman. Data flow analysis of communicating finite state machines. *ACM Trans. Program. Lang. Syst.*, 13(3), 1991.
- [42] Michel Reniers and Sjouke Mauw. High-level message sequence charts. In *SDL Forum*, Sep 1997.
- [43] Thomas Reps, Stefan Schwoon, Somesh Jha, and David Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Science of Computer Programming*, 58(1–2), 2005.
- [44] Noam Rinetzky and Mooly Sagiv. Interprocedural shape analysis for recursive programs. In *Compiler Construction, CC'01*, volume 2027 of *LNCS*, 2001.
- [45] Vlad Rusu. Combining formal verification and conformance testing for validating reactive systems. *Journal of Software Testing, Verification, and Reliability*, 13(3), September 2003.
- [46] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *Symposium on Principles of Programming Languages (POPL'99)*, January 1999.
- [47] Micha Sharir and Amir Pnueli. Semantic foundations of program analysis. In *Program Flow Analysis: Theory and Applications*, chapter 7. 1981.
- [48] Tuba Yavuz-Kahveci and Tefvik Bultan. Automated verification of concurrent linked lists with counters. In *Static Analysis Symposium, SAS'02*, volume 2477 of *LNCS*, 2002.