International Journal of Foundations of Computer Science

A Generic Approach to the Static Analysis of Concurrent Programs with Procedures*

Ahmed Bouajjani
LIAFA, University of Paris 7, 2 place Jussieu, 75251 Paris cedex 5, France
abou@liafa.jussieu.fr

and

Javier Esparza
Institute for Formal Methods in Computer Science
University of Stuttgart, Universitätstr. 38, 70569 Stuttgart, Germany
esparza@informatik.uni-stuttgart.de

and

Tayssir Touili
LIAFA, University of Paris 7, 2 place Jussieu, 75251 Paris cedex 5, France
touili@liafa.jussieu.fr

Received (received date)
Revised (revised date)
Communicated by Editor's name

ABSTRACT

We present a generic aproach to the static analysis of concurrent programs with procedures. We model programs as communicating pushdown systems. It is known that typical dataflow problems for this model are undecidable, because the emptiness problem for the intersection of context-free languages, which is undecidable, can be reduced to them. In this paper we propose an algebraic framework for defining abstractions (upper approximations) of context-free languages. We consider two classes of abstractions: finite-chain abstractions, which are abstractions whose domains do not contain any infinite chains, and commutative abstractions corresponding to classes of languages that contain a word if and only if they contain all its permutations. We show how to compute such approximations by combining automata theoretic techniques with algorithms for solving systems of polynomial inequations in Kleene algebras.

Keywords: Concurrent programs with procedures, pushdown systems, Kleene algebras, abstraction, static analysis, verification.

1. Introduction

*An earlier version of this paper was presented at the 30th Annual ACM Symposium on Principles of Programming Languages, POPL 2003.

Multi-threading is widely used in modern software, but it also enables new kinds of programming errors. Developing static analysis techniques able to detect some of these errors is becoming increasingly important, and has been the subject of a number of recent papers (see [16] for a recent survey). Developing these techniques is also a very challenging task, especially in the interprocedural case, i.e., when the programming language may contain not only synchronization primitives, but also procedures. In this case, a simple result [14] states that even the basic problem of deciding if a given control point can ever be reached is undecidable. Notice that this is the case even if, as usual in a data-flow setting, all if-then-else instructions and loop guards are replaced by nondeterminism; the analysis must only be sensitive to procedure calls and synchronizations.

Due to this negative result, every analysis algorithm must necessarily work with an upper approximation of the set of possible program paths. Such algorithms have been presented in [4, 1]. These techniques approximate both the effect of procedures and synchronization. In other words, they remain approximate both in the intraprocedural case (no procedures but synchronization) and in the synchronization-free case (no synchronization but procedures). In this paper, we present an analysis technique based on our previous work on model-checking and interprocedural analysis of sequential programs [2, 7, 6]. Our technique exhibits two main features with respect to previous work. First, it only approximates the effect of synchronizations; procedures are handled in an exact way. Second (and arguably more important), it presents a generic algorithm which allows to perform analysis of different precision and cost.

In order to explain the basic idea of the approach, let us briefly recall the result of [14]. Consider a sequential program with possibly recursive procedures and let c be one of its configurations. (A configuration is a pair consisting of a control point and a stack of activation records containing the information about the procedures which have been called but whose execution is not yet finished.) If we look at a possible execution path of the program as a sequence of statements, we can identify the set of all possible program paths reaching c from the initial configuration c_0 with a context-free language $L(c_0, c)$ (and every context-free language can be obtained this way). E.g. take a program that either terminates immediately or executes a statement a, calls itself recursively, executes a statement b, and terminates; let c be the configuration corresponding to termination; we have $L(c_0, c) = \{a^n b^n \mid n \geq 0\}$.

Consider now a concurrent system consisting of two sequential programs communicating with each other by rendezvous, and a configuration (c_1, c_2) of the system. The problem is if (c_1, c_2) is reachable from (c_{01}, c_{02}) when synchronizations are taken into account. A first necessary condition is that c_i be reachable from c_{0i} when synchronizations are not taken into account. We now interpret the execution paths reaching c_1 and c_2 as sequences of *synchronization* statements (i.e. we hide all statements that can be executed by one of the two programs independently of its partner), and obtain in this way two context-free languages $L(c_{01}, c_1), L(c_{02}, c_2)$. Then (c_1, c_2) is reachable if and only if $L(c_{01}, c_1) \cap L(c_{02}, c_2)$ is nonempty, i.e., if there exist paths of both components offering the same sequence of communications to its

partner. Unfortunately, since emptiness of the intersection of context-free languages is undecidable, forward reachability of (c_1, c_2) from (c_{01}, c_{02}) is also undecidable. In general, forward or backward reachability between configurations is undecidable.

Our attack on this problem is based on our previous work, which shows how to compute, given an arbitrary *regular* set C of configurations of a sequential program, the sets $pre^*(C)$ and $post^*(C)$ of predecessors and successors of C. We address the following problem: given two sets $C_1 \times C_2$, $C_1' \times C_2'$ of configurations of the concurrent system, is (some configuration of) $C_1 \times C_2$ reachable from (some configuration of) $C_1' \times C_2'$? This amounts to deciding if $(C_1' \times C_2') \cap pre^*(C_1 \times C_2) \neq \emptyset$ or $post^*(C_1' \times C_2') \cap (C_1 \times C_2) \neq \emptyset$. Let us consider the first possibility for this discussion. Our approach consists of computing an *abstraction* of the path language $L(C_i', C_i) = \bigcup_{c \in C_i, c' \in C_i} L(c, c')$, i.e., a set $A(C_i', C_i)$ satisfying $A(C_i', C_i) \supseteq L(C_i', C_i)$. Clearly, emptiness of $A(C_1', C_1) \cap A(C_2', C_2)$ implies emptiness of $L(C_1', C_1) \cap A(C_2', C_2)$. So we check if $A(C_1', C_1) \cap A(C_2', C_2) = \emptyset$, and if so we conclude that $C_1 \times C_2$ is unreachable from $C_1' \times C_2'$.

The problem is to find computable abstract path languages for which emptiness of intersection is decidable. In this paper, we provide a generic algorithm for the computation of these approximations. This algorithm is based on the resolution of polynomial inequalities in abstract domains. We consider two classes of abstractions: (1) Finite-chain abstractions, which are abstractions whose domain do not contain any infinite chain. In this case the inequalities can be solved by an iterative fixpoint computation. (2) Commutative abstractions, which are abstractions that forget the order between the different actions, meaning that if a word belongs to $A(C', C)$ then all its permutations also belong to $A(C', C)$. In this case, we solve the polynomial inequalities using the very elegant machinery of [10] for systems of equations in commutative Kleene algebras. We give several examples of useful abstractions leading to analysis algorithms of different precision and cost.

The paper is organized as follows. Section 2 presents our program model and communicating pushdown systems, our formal model. Section 3 gives examples of abstractions and defines the classes of abstractions considered in the paper. Section 4 reduces the computation of abstract path languages to the computation of certain sets of predecessors/successors. Section 5 presents the generic algorithm for computing predecessors and discusses its complexity. A similar algorithm that computes the successors is described in Section 6. Finally, section 7 contains conclusions and discusses related work.

2. The model

2.1. Program model: Flow Graph Systems

Our program model is very similar to the trace flow graph of [5] or [13]. A sequential program consists of a set of procedures. The program has a (possibly empty) set of global variables, and each procedure has its own (possibly empty) set of local variables, which are reincarnated each time the procedure is called. We allow

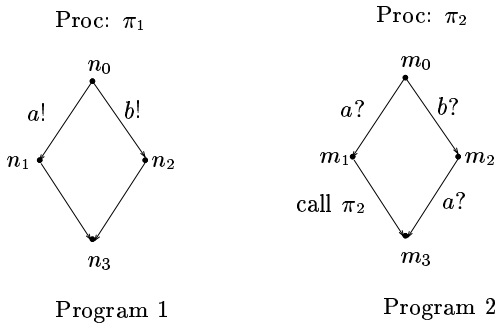


Figure 1: An example

arbitrary recursion, even mutual procedure calls, between procedures; however, as usual, we require that infinite data types have been abstracted into finite types using standard techniques of abstract interpretation (in the classical dataflow framework, all datatypes are abstracted away). Abstraction usually leads to non-deterministic control flow, which is explicitly allowed.

We represent a sequential program by a *system of flow graphs*, one for each procedure. The nodes of a flow graph correspond to control points in the procedure, and its edges are labelled with statements. Statements are assignments, calls to other procedures of the same sequential program, or communications with another sequential program. An unlabelled edge corresponds to a noop. A concurrent program is represented by a tuple of flow graph systems, one for each sequential component.

Communication takes place in rendezvous style by means of unidirectional point-to-point channels. A send statement $ch!x$ sends the value of x along channel ch, and a receive statement $ch?y$ waits for a value to be received along channel ch, and binds it to the variable y. We also allow communication of synchronization signals by means of statements $ch!$ and $ch?$. Figure 1 shows the representation as system of flow graphs of two sequential programs, each one consisting of one procedure, which exchange signals through channels a and b.

2.2. Formal model: Pushdown systems

A *pushdown system* (PDS) is a five-tuple $\mathcal{P} = (P, Act, \Gamma, c_0, \Delta)$ where P is a finite set of *control locations*, Act is a finite set of *actions*, Γ is a finite *stack alphabet*, and $\Delta \subseteq (P \times \Gamma) \times Act \times (P \times \Gamma^*)$ is a finite set of *transition rules*. If $((p,\gamma), a, (p', w)) \in \Delta$ then we write $\langle p, \gamma\rangle \overset{a}{\hookrightarrow} \langle p', w\rangle$. A *configuration* of $\mathcal{P}$ is a pair $\langle p, w\rangle$ where $p \in P$ is a control location and $w \in \Gamma^*$ is a *stack content*. c_0 is called the *initial configuration* of $\mathcal{P}$. We assume w.l.o.g. that all the rules of Δ are of the form $\langle p, \gamma\rangle \overset{a}{\hookrightarrow} \langle p', w\rangle$ such that $|w| \leq 2$ (in fact, as we shall see below, the PDSs obtained from programs already satisfy this constraint). The set of all configurations is denoted by $\mathcal{C}$. A set C of configurations is *regular* if for each control location $p \in P$ the language $\{w \in \Gamma^* \mid \langle p, w\rangle \in C\}$ is regular.

For each action a, we define a relation $\xrightarrow{a} \subseteq \mathcal{C} \times \mathcal{C}$ as follows: if $\langle q, \gamma \rangle \hookrightarrow^{a} \langle q', w \rangle$, then $\langle q, \gamma v \rangle \xrightarrow{a} \langle q', wv \rangle$ for every $v \in \Gamma^*$; we say that $\langle q, \gamma v \rangle$ is an *immediate predecessor* of $\langle q', wv \rangle$, and $\langle q', wv \rangle$ an *immediate successor* of $\langle q, \gamma v \rangle$.

The predecessor function $pre^*\colon 2^{\mathcal{C}} \to 2^{\mathcal{C}}$ of $\mathcal{P}$ is defined as follows: c belongs to $pre^*(C)$ if some successor of c belongs to C. We define $post^*(C)$ similarly.

A *communicating pushdown system* (CPDS) is a tuple $CP = (\mathcal{P}_1, \ldots, \mathcal{P}_n)$ of pushdown systems over the same set of actions Act. In order to model communication, we assume that Act contains a special action τ that represents internal actions, and that every action in $Lab = Act \setminus \{\tau\}$ is a synchronization action. The elements of Lab are called *labels*.

A *global configuration* of CP is a tuple $g = (c_1, \ldots, c_n)$ of configurations of $\mathcal{P}_1, \ldots \mathcal{P}_n$. We extend the relations $\xrightarrow{a}$ to pairs of global configurations as follows. Let $g = (c_1, \ldots, c_n)$ and $g' = (c'_1, \ldots, c'_n)$ be global configurations:

- $g \xrightarrow{\tau} g'$ if there is $1 \leq i \leq n$ such that $c_i \xrightarrow{\tau} c'_i$ and $c'_j = c_j$ for each $j \neq i$;
- $g \xrightarrow{a} g'$ if there are indices $i \neq j$ such that $c_i \xrightarrow{a} c'_i$, $c_j \xrightarrow{a} c'_j$, and $c'_k = c_k$ for every $i \neq k \neq j$.

Given a set G of global configurations, we define $pre^*(G)$ and $post^*(G)$ in the obvious way.

2.3. From the program model to the formal model

Given a tuple of flow graph systems (our model of concurrent programs), we define a corresponding CPDS. Each flow-graph system is assigned a PDS; the CPDS is just the tuple of these PDSs.

For the sake of simplicity, we assume that all procedures of a flow-graph system have the same local variables. We assign to a flow graph system the PDS $\mathcal{P} = (P, Act, \Gamma, c_0, \Delta)$ defined as follows. P is the set of all possible valuations of the global variables, or a singleton if the program has no variables or all variables have been abstracted away. Act contains the action τ and an action $ch(n)$ for each channel ch and each possible value n that can be transmitted through it. If only signals are transmitted, then the action is ch. Γ is the set of all pairs (n, v), where n is a node of a flow graph, and v is a valuation of the local variables. The initial configuration c_0 is defined as $c_0 = \langle glob_0, (n_0, loc_0) \rangle$, where $glob_0$ and loc_0 are the initial values of the global and local variables, and n_0 is the initial node of the main procedure. Δ contains a set of rules for each program statement, defined next. Let s be the statement labelling an edge of the flow graph system from node n_1 to node n_2. If s is an *assignment*, then it is translated into a set of rules of the form

$$\langle glob, (n_1, loc) \rangle \hookrightarrow^{\tau} \langle glob', (n_2, loc') \rangle.$$

where $glob$ and $glob'$ (loc and loc') are the values of the global (local) variables before and after the assignment. If s is a procedure call, then it is translated into a set of rules of the form

$$\langle glob, (n_1, loc) \rangle \hookrightarrow^{\tau} \langle glob, (m_0, loc')\,(n_2, loc'') \rangle$$

$\mathcal{P} = (\{p\}, \{a, b, \tau\}, \{m_0, m_1, m_2, m_3\}, \langle p, m_0\rangle, \{r_1, \ldots r_5\})$ where

$$r_1 : \langle p, m_0\rangle \overset{a}{\hookrightarrow} \langle p, m_1\rangle \qquad r_4 : \langle p, m_2\rangle \overset{a}{\hookrightarrow} \langle p, m_3\rangle$$
$$r_2 : \langle p, m_0\rangle \overset{b}{\hookrightarrow} \langle p, m_2\rangle \qquad r_5 : \langle p, m_3\rangle \overset{\tau}{\hookrightarrow} \langle p, \epsilon\rangle$$
$$r_3 : \langle p, m_1\rangle \overset{\tau}{\hookrightarrow} \langle p, m_0 m_3\rangle$$

Figure 2: A pushdown system

where m_0 is the start node of the called procedure, loc' denotes the initial values of its local variables, and loc'' saves the local variables of the calling procedure. An input $ch?y$ is translated into a set of rules of the form

$$\langle glob, (n_1, loc)\rangle \overset{ch(v)}{\hookrightarrow} \langle glob', (n_2, loc')\rangle.$$

where $glob'$ and loc' is the result of assigning the value v to y in $glob$ or loc (depending on whether y is global or local). An output $ch!x$ is translated into a set of rules of the form

$$\langle glob, (n_1, loc)\rangle \overset{ch(v)}{\hookrightarrow} \langle glob, (n_2, loc)\rangle.$$

where v is the value of x in $glob$ or loc. Finally, a node n without output edges, corresponding to the return instruction of a procedure, is translated into rules of the form

$$\langle glob, (n, loc)\rangle \overset{\tau}{\hookrightarrow} \langle glob, \varepsilon\rangle$$

Procedures which return values can be simulated by introducing an additional global variable and assigning the return value to it.

Notice that, since channels are unidirectional and point to point, we can only have $g \xrightarrow{ch(v)} g'$ if one of the components executes an input and another one an output.

Figure 1 yields a CPDS with two PDSs. The one for the sequential program on the right-hand-side is shown in Figure 2.

3. Abstracting path languages

3.1. Reachability Analysis and Path Languages

Let $(\mathcal{P}_1, \ldots, \mathcal{P}_n)$ be a CPDS and consider the problem of checking whether a set of configurations $C_1 \times \cdots \times C_n$ is reachable from $C_1' \times \cdots \times C_n'$. This problem can be reduced to the problem of checking the emptiness of the intersection of context-free languages. Indeed, let us assume w.l.o.g. that the set of labels (visible actions) Lab is a disjoint union of sets $Lab_{i,j}$ corresponding to synchronization actions between each pair of systems $\mathcal{P}_i$ and $\mathcal{P}_j$. (Any system can be transformed in order to satisfy this condition by duplicating and relabeling transitions corresponding to synchronization actions which are common to different pairs of systems.) Then, let

$L_i = L(C_i', C_i)$ be the set of paths leading in $\mathcal{P}_i$ from a configuration in C_i' to a configuration in C_i, and let

$$\widehat{L_i} = L_i \sqcup\!\sqcup (\bigcup_{j,k \neq i} Lab_{j,k})^*$$

where $\sqcup\!\sqcup$ is the shuffle (interleaving) operator, which means that we insert everywhere in the paths of $\mathcal{P}_i$ labels corresponding to synchronization actions between pairs of other processes. In other words, we extend each pushdown system $\mathcal{P}_i$ by self loops on each of its control states labeled with synchronization actions between pairs of other processes. Notice that this extension is done only when $n \geq 3$ (for $n = 2$, we have $\widehat{L_i} = L_i$).

Since τ actions are hidden in the path languages L_i's, it is easy to see that $C_1 \times \cdots \times C_n$ is reachable from $C_1' \times \cdots \times C_n'$ if and only if

$$\widehat{L_1} \cap \ldots \cap \widehat{L_n} \neq \emptyset$$

As mentioned in the introduction, our approach for tackling this problem (which is undecidable) is based on computing abstractions $A(C', C)$ of path languages $L(C', C)$ of pushdown systems, for given source and target sets of configurations C' and C. Once abstractions $A(C_i', C_i)$ of the path languages $\widehat{L_i}$ have been computed, we check if their intersection is empty, and if the answer is affirmative we conclude that $C_1 \times \cdots \times C_n$ is not reachable from $C_1' \times \cdots \times C_n'$.

3.2. Examples of Path Language Abstractions

We consider hereafter four examples of abstract path languages. Notice that in order to apply our approach described above, we must give a finite representation of $A(C', C)$. So, in fact, we compute an *abstract object* representing $A(C', C)$. For the moment we do this informally, and delay the formal presentation, which uses a standard abstract interpretation framework, to the next subsection.

3.2.1. First occurrence ordering

The abstract object representing $A(C', C)$ is a set W of words w such that for every $a \in Lab$, $|w|_a \leq 1$ ($|w|_a$ denotes the number of occurrences of the letter a in w). A word $w = a_1 \cdots a_n$ belongs to W if there is a path in $L(C', C)$ such that the set of letters occurring in it is precisely $\{a_1, \ldots, a_n\}$, and moreover, the first occurrences of these letters in the path occur in the order defined by w (i.e., for every $i < j$, a_i occurs for the first time before a_j). Checking emptiness of the intersection of the languages represented by S_1 and S_2 is equivalent to checking emptiness of their intersection as sets.

We can use this abstraction to conclude that no global configuration of the program shown in Figure 3 having n_3 and m_3 as control locations is reachable from the initial configuration. Since the two sequential programs have no variables, their corresponding PDSs have one control location each, say p_{01}, p_{02}. The stack alphabets are $\Gamma_1 = \{n_0, \ldots, n_3\}$, $\Gamma_2 = \{m_0, \ldots m_3\}$. The initial configurations are

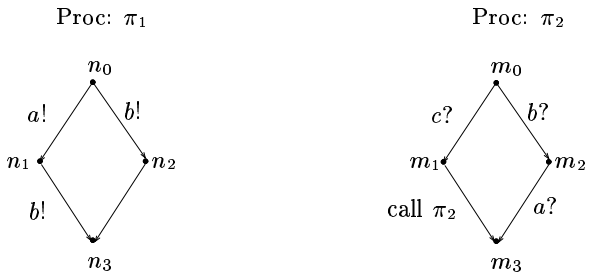


Figure 3: Another example

$c_{01} = \{(p_{01}, n_0)$, $c_{02} = (p_{02}, m_0)$. Let us denote by $W(n_3)$ and $W(m_3)$ the sets of words corresponding to $L(c_{01}, \{\langle p_{01}, n_3 w\rangle \mid w \in \Gamma_1^*\})$ and $L(c_{02}, \{\langle p_{02}, m_3 w\rangle \mid w \in \Gamma_2^*\})$. We have $W(n_3) = \{ab, b\}$, while $W(m_3) = \{cba, ba\}$. Since $W(n_3) \cap W(m_3) = \emptyset$, there is no communication sequence common to the two sequential programs allowing to reach the control locations n_3 and m_3.

3.2.2. Label bitvectors

Let us forget about the order in which the first occurrence of each letter appears. The abstract object representing representing $A(C', C)$ is now a set B of bitvectors $Lab \to \mathbb{B}$. A bitvector b belongs to B if there is a sequence in $L(C', C)$ such that $b(a) = 1$ if a occurs in the sequence and $b(a) = 0$ otherwise.

In Figure 1 we can use this abstraction to conclude that no global configuration with n_3 and m_3 as control locations is reachable. Define $B(n_3)$ and $B(m_3)$ as in the previous abstraction, *mutatis mutandis*. We have $B(n_3) = \{\binom{1}{0}, \binom{0}{1}\}$, while $B(m_3) = \{\binom{1}{1}\}$ (the components of the bitvectors correspond to the labels a, b).

3.2.3. Forbidden and required sets

The abstract object is a pair $[F, R]$, where $F, R \subseteq Lab$. F, the *forbidden* set, contains the labels a that do not occur in any sequence of $L(C', C)$. R, the *required* set, contains the labels a that appear in all sequences of $L(C', C)$. $[F, R]$ represents the language of all sequences containing no occurrence of letters in F and at least one occurrence of each letter in R. It is easy to see that the languages represented by $[F_1, R_1]$ and $[F_2, R_2]$ have empty intersection if and only if $F_1 \cap R_2 \neq \emptyset$ or $R_1 \cap F_2 \neq \emptyset$.

In Figure 4 we can use this abstraction to conclude that no global configuration with n_1 and m_2 as control locations is reachable: The action a is required to reach n_1, but forbidden if we wish to reach m_2.

3.2.4. Parikh images

The abstract object representing $A(C', C)$ is a (possibly infinite, but, as we shall see in the next section, finitely representable) set M of integer vectors $Lab \to \mathbb{N}$. A

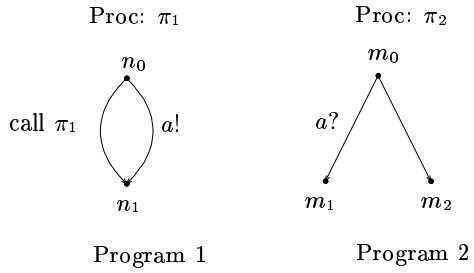


Figure 4: Yet Another example

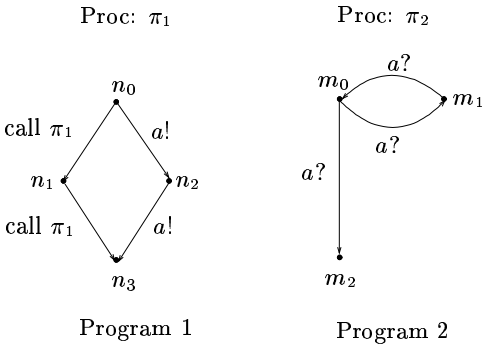


Figure 5: A last example

mapping m belongs to M if $L(C', C)$ contains a sequence in which each label a occurs exactly $m(a)$ times. We call m the *Parikh image* of the sequence. M represents the language of all sequences whose Parikh-images belong to M. Checking emptiness of the intersection of M_1 and M_2 seen as languages is equivalent to checking emptiness of their intersection as sets.

In Figure 5 we can conclude that no global configuration with n_3 and m_2 as control location is reachable: In order to reach n_3, a must occur an even number of times; however, in order to reach m_3, it must occur an odd number of times.

There are many other examples. For instance, it is possible to mix label bitvectors and Parikh images. For some actions, we retain the information whether they occur in a sequence or not, while for others we count the number of occurrences. Also, in some cases, like the one of Figure 5, it is useful to count modulo a number.

3.3. A formal framework

Let $\mathcal{L}$ be the complete lattice of languages over *Lab*, i.e.,

$$\mathcal{L} = (2^{Lab^*}, \subseteq, \cup, \cap, \emptyset, Lab^*).$$

Formally, an abstraction of $\mathcal{L}$ consists of an *abstract lattice* $\mathcal{D} = (D, \leq, \sqcup, \sqcap, \bot, \top)$, where D is some domain called the *abstract domain*, and a *Galois connection* (α, γ)

between $\mathcal{L}$ and $\mathcal{D}$, i.e., a pair of mappings $\alpha : \mathcal{L} \to \mathcal{D}$ and $\gamma : \mathcal{D} \to \mathcal{L}$ such that

$$\forall x \in \mathcal{L},\ \forall y \in \mathcal{D}.\ \alpha(x) \leq y \iff x \subseteq \gamma(y)\ .$$

Our purpose is to define abstractions of $\mathcal{L}$ such that (i) the abstract path language $\alpha(L(C_i', C_i))$ is computable when C_i and C_i' are regular, and (*ii*) emptiness of an intersection is decidable in the domain D. To that aim, we consider a class of abstractions, which we call *Kleene abstractions*, in which the lattice $\mathcal{D}$ has a rich algebraic structure. In the rest of this section we give the abstract definition and some general intuition. The next section gives concrete examples. In fact, it shows that the four examples of section 3.2 are Kleene abstractions.

3.3.1. Kleene algebras

An *idempotent semiring* is a tuple $\mathcal{K} = (K, \oplus, \odot, \bar{0}, \bar{1})$, where $\oplus$ is an associative, commutative, and idempotent ($a \oplus a = a$) operation, and $\odot$ is an associative operation. $\bar{0}$ and $\bar{1}$ are neutral elements for $\oplus$ and $\odot$, respectively, $\bar{0}$ is an annihilator for $\odot$ ($a \odot \bar{0} = \bar{0} \odot a = \bar{0}$), and $\odot$ distributes over $\oplus$. $\mathcal{K}$ *is a Lab-semiring* if K is generated by $\bar{0}$, $\bar{1}$, and an element v_a for each $a \in Lab$. A semiring is *closed* if $\oplus$ can be extended to an operator over countably infinite sets (i.e., countably infinite sums are allowed), and this operator has the same properties as $\oplus$ (it is associative, commutative, idempotent, and $\odot$ distributes over it). In closed semirings we can define define $a^0 = \bar{1}$, $a^{n+1} = a \odot a^n$, and $a^\star = \bigoplus_{n \geq 0} a^n$. Adding the $\star$-operation to $\mathcal{K}$ transforms it into a *Kleene algebra*.

Convention 3.1 *In the sequel, we reserve the word Kleene algebra for the Kleene algebras induced by closed idempotent Lab-semirings.*

Let $\mathcal{K}$ be a Kleene algebra with carrier K. For technical reasons, we need to introduce the notion of the reverse x^R of an element x of K. This notion is defined inductively as follows:

- $v_a^R = v_a$,
- $(x \oplus y)^R = x^R \oplus y^R$,
- $(x \odot y)^R = y^R \odot x^R$, and
- $(x^\star)^R = (x^R)^\star$.

It follows that for every element x of K, we have:

$$(x^R)^R = x. \tag{1}$$

3.3.2. Kleene abstractions

An abstract lattice $\mathcal{D} = (D, \leq, \sqcup, \sqcap, \bot, \top)$ *is compatible with* a Kleene algebra $\mathcal{K} = (K, \oplus, \odot, \star, \bar{0}, \bar{1})$ if the order, bottom element and join operation of the abstract lattice are given by $x \leq y$ if $x \oplus y = y$, $\bot = \bar{0}$, and $\sqcup = \oplus$, respectively.

A *Kleene abstraction* is an abstraction in which the abstract lattice $\mathcal{D}$ is compatible with a Kleene algebra $\mathcal{K} = (K, \oplus, \odot, \bar{0}, \bar{1})$, and the Galois connection is given by $\alpha : 2^{Lab^*} \to K$ and $\gamma : K \to 2^{Lab^*}$ such that

$$\begin{aligned} \alpha(L) &= \bigoplus_{a_1 \cdots a_n \in L} v_{a_1} \odot \cdots \odot v_{a_n} \\ \gamma(x) &= \{a_1 \cdots a_n \in 2^{Lab^*} \mid v_{a_1} \odot \cdots \odot v_{a_n} \leq x\} \end{aligned}$$

It can easily be checked that (α, γ) is indeed a Galois connection.

The intuition behind a Kleene abstraction is the following. The abstract operations $\oplus$, $\odot$, and $\star$ of $\mathcal{K}$ correspond to union, concatenation, and Kleene closure of languages in the lattice $\mathcal{L}$. $\bar{0}$ and $\bar{1}$ are the abstract objects corresponding to the empty language and to $\{\epsilon\}$, respectively. The element v_a is the abstract object for the language $\{a\}$. The top element $\top \in K$ is the abstract object for Lab^*, and the meet operation $\sqcap$ corresponds to language intersection on the lattice $\mathcal{L}$. If we wish to compute $\alpha(L)$ for some language L, we just "abstract" each word of $a_1 \ldots a_n$ by the object $v_{a_1} \odot \cdots \odot v_{a_n}$, and take the union of these objects.

3.3.3. Properties of Kleene abstractions

In a Kleene abstraction we have $\alpha(\emptyset) = \bot$ and $\gamma(\bot) = \emptyset$, which implies that

$$\forall L_1, \ldots, L_n.\ \alpha(L_1) \sqcap \ldots \sqcap \alpha(L_n) = \bot \ \Rightarrow\ L_1 \cap \ldots \cap L_n = \emptyset$$

This property is necessary for our approach: In order to check the emptiness of the intersection of context-free languages, it suffices to check the emptiness of the intersection of their abstractions.

Moreover, we get straightforwardly from the definition of α given above that for every $L \in 2^{Lab^*}$,

$$\alpha(L^R) = \big(\alpha(L)\big)^R, \tag{2}$$

where L^R denotes the reverse of the language L (in the usual sense).

3.3.4. Particular Kleene abstractions

In this paper, we consider two families of Kleene abstractions. A *finite-chain abstraction* is an abstraction such that the semilattice $(K, \oplus)$ has no infinite ascending chains. Particular cases of such abstractions are *finite abstractions* where the abstract domain K is finite. A *commutative abstraction* is an abstraction where $\odot$ is commutative. Intuitively, this means that $\odot$ "forgets the order" of labels. Thus we get that $(K, \oplus, \odot, \star, \bar{0}, \bar{1})$ is a *commutative Kleene algebra*. We show that both families of abstractions are computable.

3.4. Instances of abstractions

We show that the four examples of section 3.2 are finite-chain or commutative Kleene abstractions. We define the corresponding Kleene algebras by giving the

carrier, the definitions of $\oplus$ and $\odot$, and the elements $\overline{1}$ and $\overline{0}$. Notice that by convention 3.1 the $\star$ operation is determined by $\oplus$ and $\odot$.

3.4.1. First occurrence ordering

Let $W = \{w \in Lab^* \mid \forall a \in Lab, |w|_a \leq 1\}$, i.e., the set of words where each letter occurs at most once. We consider the Kleene algebra $\mathcal{K}$ given by: (1) $K = 2^W$; (2) $\oplus = \cup$, (3) $U \odot V$ is the set of words $u_1 \cdot v_2'$ such that $u_1 \in U$ and there is $v_2 \in V$ such that v_2' is the projection of v_2 onto the set of letters that do not occur in u_1, (4) $\overline{0} = \emptyset$, and (5) $\overline{1} = \{\epsilon\}$. It is easy to see that this is indeed a Kleene algebra generated by the elements $v_a = \{a\}$ for each a in Lab.

The abstract lattice compatible with $\mathcal{K}$ is obtained by taking $\top = W$, and $\sqcap = \cap$. Observe that this is a finite-chain abstraction, since K is finite.

3.4.2. Label bitvectors

We consider the Kleene algebra $\mathcal{K}$ given by: (1) $K = 2^{[Lab \to \mathbb{B}]}$, where $[Lab \to \mathbb{B}]$ is the set of bitvectors indexed by Lab, (2) $\oplus = \cup$, (3) $\odot = \vee$ (extended to vectors componentwise, and to sets of vectors), (4) $\overline{0} = \emptyset$, and (5) $\overline{1} = \{(\mathit{false}, \ldots, \mathit{false})\}$. Again, it is easy to see that this is a Kleene algebra generated by the elements v_a for each a in Lab, where v_a is the singleton $\{b_a\}$ given by $b_a(a) = \mathit{true}$, and $b_a(a') = \mathit{false}$ for every $a' \neq a$, The abstract lattice is obtained by taking $\top = 2^{[Lab \to \mathbb{B}]}$, and $\sqcap = \cap$. This abstraction is both finite-chain and commutative.

3.4.3. Forbidden and required sets

Abstract elements are no longer sets of vectors, but pairs of sets. The Kleene algebra $\mathcal{K}$ is defined as follows:

- $K = \{\overline{0}\} \cup \{[F, R] \in 2^{Lab} \times 2^{Lab} \mid F \cap R = \emptyset\}$,
- $[F_1, R_1] \oplus [F_2, R_2] = [F_1 \cap F_2, R_1 \cap R_2]$,
- $[F_1, R_1] \odot [F_2, R_2] = [F_1 \cap F_2, R_1 \cup R_2]$,
- $\overline{0}$ is by definition neutral for $\oplus$ and annihilator for $\odot$,
- $\overline{1} = [Lab, \emptyset]$.

In this case, K is generated by the elements v_a for each $a \in Lab$, where $v_a(a) = [Lab \setminus \{a\}, \{a\}]$. The abstract lattice is obtained by taking $\top = [\emptyset, \emptyset]$, and defining $\sqcap$ by:

- $[F_1, R_1] \sqcap [F_2, R_2] = [F_1 \cup F_2, R_1 \cup R_2]$ if $(F_1 \cap R_2) \cup (F_2 \cap R_1) = \emptyset$,
- $[F_1, R_1] \sqcap [F_2, R_2] = \overline{0}$ otherwise,
- $\overline{0} \sqcap x = x \sqcap \overline{0} = \overline{0}$.

Like the previous one, this abstraction is both finite-chain and commutative.

3.4.4. Parikh images

Abstract elements are semilinear sets of integer vectors in $[Lab \to \mathbb{N}]$. We recall that semilinear sets are finite unions of sets of the form $\{\vec{b}+k_1\vec{p}_1+\ldots k_n\vec{p}_n \mid k_1 \ldots k_n \in \mathbb{N}\}$, where $\vec{b}, \vec{p}_1, \ldots, \vec{p}_n \in [Lab \to \mathbb{N}]$ ($\vec{b}$ is the basis, and the $\vec{p}_i$'s are the periods). We consider the commutative algebra $\mathcal{K}$ given by: (1) K is the set of all semilinear sets, (2) $\oplus = \cup$, (3) $S_1 \cdot S_2 = \{\vec{u}_1 + \vec{u}_2 \mid \vec{u}_1 \in S_1, \vec{u}_2 \in S_2\}$, (4) $\bar{0} = \emptyset$, and (5) $\bar{1}$ is a singleton set containing the vector that associates 0 to every label a.

It is easy to see that $\mathcal{K}$ is a Kleene algebra generated by the elements $v_a = \{\vec{u}_a\}$ for each $a \in Lab$, where $\vec{u}_a(a) = 1$ and $\vec{u}_a(b) = 0$ for every $b \neq a$.

The abstract lattice is obtained by taking $\top$ as the set of all vectors in $[Lab \to \mathbb{N}]$, and defining $\sqcap$ as intersection of semilinear sets. We use here that semilinear sets are closed under intersection [9].

We represent a semilinear set as a set of pairs $(\vec{b}, \{\vec{p}_1, \ldots, \vec{p}_n\})$. It is easy to compute the representation of $S_1 \oplus S_2$, $S_1 \cdot S_2$, and $S_1^\star$ from the representations of S_1 and S_2. The theory of semilinear sets shows that the representation of $S_1 \sqcap S_2$ is also effectively computable, and that the question $S_1 \overset{?}{\leq} S_2$ is decidable (see again [9]).

4. Computing abstract path languages

Given a Kleene abstraction, the abstract path language $\alpha(L(C', C))$ is nothing but a particular element of the corresponding Kleene algebra. We give a generic procedure for computing this element in terms of the basic elements v_a that generate the algebra. Throughout the section we use the PDS of Figure 2 as running example.

4.1. K-predecessors and K-successors

We introduce hereafter a notion of K-configuration and K-transition relation for pushdown systems.

Let us fix from now on a pushdown automaton $\mathcal{P} = (P, Act, \Gamma, c_0, \Delta)$ and a Kleene algebra $(K, \oplus, \odot, \star, \bar{0}, \bar{1})$ corresponding to some Kleene abstraction. The set of path-expressions Π_K is the subset of K obtained by taking $\bar{1}$ and the basic elements v_a for each $a \in Lab$, and closing under $\odot$. Given a path-expression π, we denote by $|\pi|$ (the *length* of π), the number of occurrences of basic elements in π.

A K-configuration of $\mathcal{P}$ is a pair (c, π), where c is a configuration of $\mathcal{P}$ and π is a path-expression over K. We extend the pushdown transition relation $\xrightarrow{a}$ to K-configurations as follows: if $c \xrightarrow{a} c'$ for some label $a \in Lab$, then $(c, \pi) \to_K (c', v_a \odot \pi)$ and $(c', \pi) \leftarrow_K (c, v_a \odot \pi)$ for every $\pi \in \Pi_K$; moreover, if $c \xrightarrow{\tau} c'$, then $(c, \pi) \to_K (c', \pi)$ and $(c, \pi) \leftarrow_K (c', \pi)$ for every $\pi \in \Pi_K$. We say that $(c', v_a \odot \pi)$ (respectively (c', π)) is an immediate K-successor of (c, π) and $(c, v_a \odot \pi)$ (respectively (c, π)) is an immediate K-predecessor of (c', π). The *forward (backward) reachability relation over* K $\Rightarrow_K$ ($\Leftarrow_K$) is the reflexive and transitive closure of $\to_K$ ($\leftarrow_K$).

Given a set of configurations C, we define $pre^*_K(C)$ ($post^*_K(C)$) as the set of K-configurations (c,π) such that $(c',\bar{1}) \Leftarrow_K (c,\pi)$ ($(c',\bar{1}) \Rightarrow_K (c,\pi)$) for some $c' \in C$. Intuitively, a K-predecessor (c',π) of a configuration c memorizes in π information about the sequence of labels that lead from c to c' (if the sequence is $a_1 \ldots a_n$, then the information stored is $v_{a_1} \odot \ldots \odot v_{a_n}$), and a K-successor (c',π) of a configuration c is such that $\pi = v_{a_n} \odot \ldots \odot v_{a_1}$ if the sequence $a_1 \ldots a_n$ leads from c to c'.

In other words, we have

$$\begin{aligned} pre^*_K(c) &= \{(c',\pi) \mid c' \in pre^*(c),\ \pi \in \alpha(L(c',c))\} \\ post^*_K(c) &= \{(c',\pi) \mid c' \in post^*(c),\ \pi \in \alpha\big(L(c,c')^R\big)\} \end{aligned}$$

4.2. K-automata

The algorithmic approach we propose for computing $\alpha(L(C',C))$ is based on constructing automata representations of the pre^*_K and $post^*_K$-images of regular sets of configurations.

In [2, 6] automata are used to represent possibly infinite sets of configurations. We now extend this notion to K-automata in order to manipulate regular sets of K-configurations.

Definition 1 *A K-automaton for the PDS $\mathcal{P}$ is a tuple $\mathcal{A} = (\Gamma, Q, \delta, P, F)$ where Q is a finite set of states, $\delta \subseteq Q \times \Gamma \times K \times Q$ is a set of transitions, $P \subseteq Q$ is the set of initial states, and $F \subseteq Q$ is the set of final states.*

We define the transition relation $\longrightarrow \subseteq Q \times \Gamma^* \times K \times Q$ *as the smallest relation satisfying:*

- *if $(q,\gamma,e,q') \in \delta$ then $q \xrightarrow{(\gamma,e)} q'$,*
- *$q \xrightarrow{(\epsilon,\bar{1})} q$ for every $q \in Q$, and*
- *if $q \xrightarrow{(w_1,e_1)} q''$ and $q'' \xrightarrow{(w_2,e_2)} q'$ then $q \xrightarrow{(w_1 w_2, e_1 \odot e_2)} q'$.*

A run of $\mathcal{A}$ is a sequence $p \xrightarrow{(\gamma_1,e_1)} q_1 \cdots \xrightarrow{(\gamma_n,e_n)} q_n$. The label of this run is $(\gamma_1 \cdots \gamma_n, e_1 \odot \cdots \odot e_n)$. $\mathcal{A}$ accepts a pair $\big(\langle p,w\rangle,\pi\big)$ if $p \xrightarrow{(w,\pi \oplus e)} q$ for some $q \in F$ and some $e \in K$.

Ordinary finite automata can be seen as a special case of K-automata in which all transitions are labelled by $\bar{1}$. Equivalently, they can be obtained by dropping all the references to K in the definition of K-automata. Figure 6 shows an automaton for the PDS of Figure 2 accepting the set of configurations $\langle p, m_3\Gamma^*\rangle = \{\langle p, m_3 w\rangle \mid w \in (m_0 + \ldots + m_3)^*\}$.

Given an automaton $\mathcal{A}$, we denote by $Conf(\mathcal{A})$ the set of configurations recognized by $\mathcal{A}$. Throughout the paper we use the symbols p, p', p'', p_i, etc. to denote initial states of (K-)automata. Arbitrary states are denoted by q, q', q'', q_i, etc.

4.3. Reduction to computing pre^*_K and $post^*_K$ images

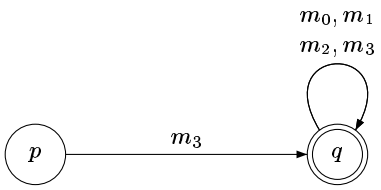


Figure 6: An automaton accepting $\langle p, m_3\Gamma^*\rangle = \{\langle p, m_3 w\rangle \mid w \in (m_0 + \ldots + m_3)^*\}$

Let C and C' be two regular sets of configurations represented by means of automata, and consider the problem of computing $\alpha(L(C', C))$. Assume that we are able to compute a K-automaton $\mathcal{A}_{pre^*_K}$ which recognizes the set $pre^*_K(C)$. Then, we can construct a K-automaton $\mathcal{A}^{C'}_{pre^*_K}$ over $\Gamma \times K$ which is the restriction of $\mathcal{A}_{pre^*_K}$ to configurations in C'. This automaton can be straightforwardly constructed by a product between $\mathcal{A}_{pre^*_K}$ and the automaton recognizing C'. [a]

Let us assume w.l.o.g. that all configurations in C' have the same control location, say p (the generalization to several control locations is straightforward). Then, $\alpha(L(C', C))$ is the *K-language* of $\mathcal{A}^{C'}_{pre^*_K}$, defined as the element of K given by:

$$\bigoplus\{e_1 \odot \cdots \odot e_n \mid \mathcal{A}^{C'}_{pre^*_K} \text{ has an accepting run of the form } p \xrightarrow{(w, e_1 \odot \cdots \odot e_n)} q\ \}.$$

A symmetrical approach can be adopted by computing an automaton $\mathcal{A}_{post^*_K}$ recognizing $post^*_K(C')$ and then restricting it to the set of configurations C, obtaining a K-automaton $\mathcal{A}^{C}_{post^*_K}$. The *$K$-language* of $\mathcal{A}^{C}_{post^*_K}$ is equal to $\alpha(L(C', C)^R)$. Then $\alpha(L(C', C))$ is obtained by reversing this K-language since by equations (1) and (2) we have that $\alpha(L(C', C)) = \big(\alpha(L(C', C)^R)\big)^R$.

Therefore, we have reduced our problem of computing $\alpha(L(C', C))$ to (i) constructing $\mathcal{A}_{pre^*_K}$ (or $\mathcal{A}_{post^*_K}$), and (ii) constructing a finite representation of the K-language of $\mathcal{A}_{pre^*_K}$ (or $\mathcal{A}_{post^*_K}$).

Problem (ii) can be solved using standard techniques for solving linear equations in Kleene algebras (e.g., using Floyd-Warshall's algorithm, or Gauss elimination). Indeed, this problem is a straightforward generalization of the problem of building regular expressions associated with finite automata.

Hence, the main problem to solve is how to compute K-automata recognizing $post^*_K$ and pre^*_K-images. We show that we can reduce this problem to solving polynomial (in)equations. In the following, we consider only the case of computing pre^*_K-images.

5. Computing pre^*_K images

Given a regular set of configurations C recognized by an automaton $\mathcal{A}$, we

[a] We consider the smallest set of transitions such that if $q_1 \xrightarrow{(a,e)} q'_1$ is in $\mathcal{A}_{pre^*_K}$ and $q_2 \xrightarrow{a} q'_2$ is in $\mathcal{A}_{C'}$ then $(q_1, q_2) \xrightarrow{(a,e)} (q'_1, q'_2)$ is in $\mathcal{A}^{C'}_{pre^*_K}$.

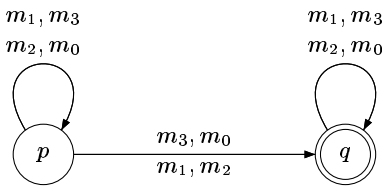


Figure 7: The automaton representing $pre^*(\langle p, m_3\Gamma^*\rangle)$

construct a K-automaton $\mathcal{A}_{pre^*_K}$ that recognizes $pre^*_K(C)$. We assume w.l.o.g. that $\mathcal{A}$ has no transition leading to an initial state.

5.1. A generic procedure

We proceed in two steps. First, we construct an automaton $\mathcal{A}_{pre^*}$ recognizing the set $pre^*(C)$. Then, we tag the transitions of $\mathcal{A}_{pre^*}$ with adequate elements of K.

For the first step we use the procedure of [2, 6], which consists of adding new transitions to $\mathcal{A}$ according to the following *saturation rule:*

> If $\langle p, \gamma\rangle \overset{a}{\hookrightarrow} \langle p', w\rangle$ and $p' \overset{w}{\longrightarrow} q$ in the current automaton, add a transition (p, γ, q).

For instance, the saturation rule allows us to add a new transition transition $p \xrightarrow{m_3} p$ to the automaton of Figure 6, because $\langle p, m_3\rangle \overset{\tau}{\hookrightarrow} \langle p, \epsilon\rangle$ (rule r_5) and $p \xrightarrow{\epsilon} p$. Then, we can add $p \xrightarrow{m_2} p$ (rule r_4), $p \xrightarrow{m_0} p$ (rule r_2), $p \xrightarrow{m_1} q$ (rule r_3, together with $p \xrightarrow{m_0 m_3} q$) etc. The final result is the automaton shown in Figure 7.

The second step consists of tagging the transitions t of $\mathcal{A}_{pre^*}$ with elements $l(t) \in K$, defined as the smallest elements of K (with respect to $\leq$) satisfying the following inequalities (where $v_\tau = \bar{1}$):

(α_1) if $t = (q, \gamma, q')$ was already a transition of $\mathcal{A}$, then

$$\bar{1} \leq l(t),$$

(α_2) for every rule $\langle p, \gamma\rangle \overset{a}{\hookrightarrow} \langle p', \gamma'\rangle$, and every $q \in Q$,

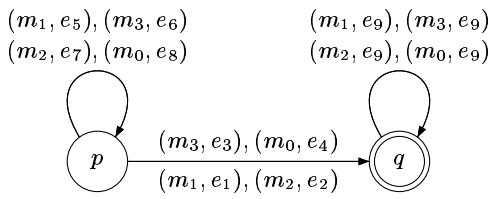
$$v_a \odot l\big((p', \gamma', q)\big) \leq l\big((p, \gamma, q)\big)$$

(α_3) for every rule $\langle p, \gamma\rangle \overset{a}{\hookrightarrow} \langle p', \epsilon\rangle$,

$$v_a \leq l\big((p, \gamma, p')\big)$$

(α_4) for every rule $\langle p, \gamma\rangle \overset{a}{\hookrightarrow} \langle p', \gamma_1\gamma_2\rangle$, and every $q \in Q$,

$$\bigoplus_{q' \in Q} \Big(v_a \odot l\big((p', \gamma_1, q')\big) \odot l\big((q', \gamma_2, q)\big)\Big) \leq l\big((p, \gamma, q)\big)$$



$$
\begin{aligned}
(x_4 \odot x_9) \oplus (x_8 \odot x_3) &\leq x_1 \\
v_a \odot x_3 &\leq x_2 \\
\bar{1} &\leq x_3 \\
(v_a \odot x_1) \oplus (v_b \odot x_2) &\leq x_4 \\
x_8 \odot x_6 &\leq x_5 \\
\bar{1} &\leq x_6 \\
v_a \odot x_6 &\leq x_7 \\
(v_b \odot x_7) \oplus (v_a \odot x_5) &\leq x_8 \\
\bar{1} &\leq x_9
\end{aligned}
$$

Figure 8: System of inequalities for the K-automaton representing $pre^*_K(\langle p, m_3\Gamma^*\rangle)$

Formally, if $\mathcal{A}_{pre^*} = (\Gamma, Q, \delta, P, F)$ is the automaton recognizing $pre^*\big(Conf(\mathcal{A})\big)$ constructed by the procedure of [2, 6], then $\mathcal{A}_{pre^*_K} = (\Gamma, Q, \delta', P, F)$ where $\delta' = \{(q, \gamma, e, q') \mid (q, \gamma, q') \in \delta, e = l\big((q, \gamma, q')\big)\}$.

In the rest of this section we show that $\mathcal{A}_{pre^*_K}$ indeed recognizes $pre^*_K\big(Conf(\mathcal{A})\big)$. In the next section we show how to compute the tags $l(p, \gamma, q)$. (Notice that the definition we have given is not effective.) Before presenting the formal proof, let us briefly explain the intuition behind (α_1)–(α_4), with the help of our running example.

- Rule (α_1) expresses that if c is a configuration from $Conf(\mathcal{A})$, then $(c, \bar{1})$ has to be accepted by $\mathcal{A}_{pre^*_K}$,
- Rules (α_2), (α_3), and (α_4) express that for every rule $r = (p, \gamma) \overset{a}{\hookrightarrow} (p', u)$, and every $q \in Q$, if w is such that $q \xrightarrow{w} q'$ for some final state q', and if $(\langle p', uw\rangle, \pi)$ is a K-predecessor of $Conf(\mathcal{A})$, then so is $(\langle p, \gamma w\rangle, v_a \odot \pi)$.

Figure 8 shows the tagging we obtain for the automaton of Figure 7. Let us explain briefly some of these inequalities. The inequality $v_a \odot x_3 \leq x_2$ expresses that if π is a path-expression of K such that $(\langle p, m_3\rangle, \pi)$ is a K-predecessor of $(\langle p, m_3\Gamma^*\rangle, \bar{1})$, then so is $(\langle p, m_2\rangle, v_a \odot \pi)$ (rule r_4). Since the transitions $p \xrightarrow{m_3} q$ and $p \xrightarrow{m_2} q$ are respectively tagged with e_3 and e_2, it follows that

$$\pi \leq e_3 \Rightarrow v_a \odot \pi \leq e_2.$$

Therefore, $v_a \odot e_3 \leq e_2$.

Similarly, the inequality $(x_4 \odot x_9) \oplus (x_8 \odot x_3) \leq x_1$ expresses that if π is a path-expression of K such that $(\langle p, m_0 m_3 \rangle, \pi)$ is a K-predecessor of $(\langle p, m_3 \Gamma^* \rangle, \bar{1})$, then so is $(\langle p, m_1 \rangle, \pi)$ (rule r_3). Since there are two paths leading from p to q by $m_0 m_3$ (which are respectively tagged with $e_4 \odot e_9$ and $e_8 \odot e_3$), it follows that if $\pi \leq e_8 \odot e_3$ or $\pi \leq e_4 \odot e_9$, then $\pi \leq e_1$, which means that $e_8 \odot e_3 \leq e_1$ and $e_4 \odot e_9 \leq e_1$, i.e., $(e_4 \odot e_9) \oplus (e_8 \odot e_3) \leq e_1$.

We are now ready to state and prove our theorem:

Theorem 1 *Given a PDS $\mathcal{P}$ and a regular set of configurations recognized by an automaton $\mathcal{A}$, the K-automaton $\mathcal{A}_{pre^*_K}$ described above recognizes $pre^*_K\big(Conf(\mathcal{A})\big)$.*

For the proof, we introduce the following notation:

Notation 5.1 *Let ρ be a run of $\mathcal{A}_{pre^*_K}$ leading from p to q and having (w, e) as label. Abusing notation we denote this by $\rho = p \xrightarrow{(w,e)} q$. We denote the expression e by $\lambda\big((p', w, q)_\rho\big)$. If $w = \gamma \in \Gamma$, i.e., if ρ corresponds to one single transition, then we simply write $e = \lambda\big((p', \gamma, q)\big)$.*

Theorem 1 is an immediate consequence of Lemma 1 and Lemma 2 that we prove hereafter.

Lemma 1 *For every configuration $\langle p, v \rangle \in Conf(\mathcal{A})$, if $\big(\langle p, v \rangle, \bar{1}\big) \Leftarrow_K \big(\langle p', w \rangle, \pi\big)$ for some path-expression $\pi \in \Pi_K$, then there exists $e \in K$ such that $\pi \leq e$ and $p' \xrightarrow{(w,e)} q$ for some final state q of $\mathcal{A}_{pre^*_K}$.*

Proof:

To prove the statement it suffices to show that if $\big(\langle p, v \rangle, \bar{1}\big) \Leftarrow_K \big(\langle p', w \rangle, \pi\big)$ for some path-expression $\pi \in \Pi_K$, then there exists a final state q such that $\rho = p' \xrightarrow{w} q$ is a path of $\mathcal{A}_{pre^*}$ and $\pi \leq \lambda\big((p', w, q)_\rho\big)$.

Let $\big(\langle p, v \rangle, \bar{1}\big) \overset{k}{\Leftarrow}_K \big(\langle p', w \rangle, \pi\big)$ mean that $\big(\langle p', w \rangle, \pi\big)$ is a K-predecessor of $\big(\langle p, v \rangle, \bar{1}\big)$ after k steps. We show the statement by induction on k:

- $k = 0$. Then $p' = p$, $w = v$, and $\pi = \bar{1}$. Since $\langle p, v \rangle \in Conf(\mathcal{A})$, then there exists a final state q such that $\rho = p \xrightarrow{v} q$ is a path of $\mathcal{A}$ (and hence of $\mathcal{A}_{pre^*}$). Therefore, the rule (α_1) imply that $\bar{1} \leq \lambda\big((p, v, q)_\rho\big)$.

- $k > 0$. Let then $p'' \in P, u \in \Gamma^*, \pi' \in \Pi_K$ such that

$$\big(\langle p, v \rangle, \bar{1}\big) \overset{k-1}{\Leftarrow}_K \big(\langle p'', u \rangle, \pi'\big) \overset{1}{\Leftarrow}_K \big(\langle p', w \rangle, \pi\big)$$

  By induction hypothesis, there exists a final state q such that $\rho = p'' \xrightarrow{u} q$ is a run in $\mathcal{A}_{pre^*}$ and $\pi' \leq \lambda\big((p'', u, q)_\rho\big)$.

  Since $\big(\langle p'', u \rangle, \pi'\big) \overset{1}{\Leftarrow}_K \big(\langle p', w \rangle, \pi\big)$, there are $\gamma \in \Gamma, w_1, u_1 \in \Gamma^*$ and a rule $r = (p', \gamma) \overset{a}{\hookrightarrow} (p'', u_1)$ of Δ such that $w = \gamma w_1$, $u = u_1 w_1$, and $\pi = v_a \odot \pi'$. Let then $q' \in Q$ be such that $\rho = p'' \xrightarrow{u_1} q' \xrightarrow{w_1} q$. Let $\rho_1 = p'' \xrightarrow{u_1} q'$ be the first part of ρ, and $\rho_2 = q' \xrightarrow{w_1} q$ be its second part. It follows that $\pi' \leq \lambda\big((p'', u_1, q')_{\rho_1}\big) \odot \lambda\big((q', w_1, q)_{\rho_2}\big)$. From the saturation rule, it follows that

there is a transition $p' \xrightarrow{\gamma} q'$ in $\mathcal{A}_{pre^*}$, which means that $\rho' = p' \xrightarrow{\gamma} q' \xrightarrow{w_1} q$ is a path in $\mathcal{A}_{pre^*}$. From rules $(\alpha_2), (\alpha_3)$, and (α_4), it follows that

$$v_a \odot \lambda\big((p'', u_1, q')_{\rho_1}\big) \leq \lambda\big((p', \gamma, q')\big)$$

Therefore, $\rho' = p' \xrightarrow{w} q$ holds in $\mathcal{A}_{pre^*}$ (since $\gamma w_1 = w$), where q is a final state, and

$$\begin{aligned} \pi &= v_a \odot \pi' \\ &\leq v_a \odot \lambda\big((p'', u_1, q')_{\rho_1}\big) \odot \lambda\big((q', w_1, q)_{\rho_2}\big) \\ &\leq \lambda\big((p', \gamma, q')\big) \odot \lambda\big((q', w_1, q)_{\rho_2}\big) \\ &= \lambda\big((p', w, q)_{\rho'}\big) \end{aligned}$$

This ends the proof of the lemma.

□

Lemma 2 *Let $p \xrightarrow{(w,e)} q$ be a path of $\mathcal{A}_{pre^*_K}$, then for every path-expression $\pi \leq e$, there exists a configuration $\langle p', w' \rangle$ such that $p' \xrightarrow{w'} q$ is a path in $\mathcal{A}$, and $\big(\langle p', w' \rangle, \bar{1}\big) \Leftarrow_K \big(\langle p, w \rangle, \pi\big)$.*

Proof: We will show by induction on $|\pi|$ that if $\pi \leq \lambda\big((p, w, q)_\rho\big)$ for some run $\rho = p \xrightarrow{w} q$ of $\mathcal{A}_{pre^*}$, then there exists a configuration $\langle p', w' \rangle$ such that $p' \xrightarrow{w'} q$ is a path in $\mathcal{A}$ and $\big(\langle p', w' \rangle, \bar{1}\big) \Leftarrow_K \big(\langle p, w \rangle, \pi\big)$.

Let $\gamma \in \Gamma$, $w_1 \in \Gamma^*$, and $q' \in Q$ be such that $w = \gamma w_1$ and

$$\rho = p \xrightarrow{\gamma} q' \xrightarrow{w_1} q.$$

Let π be a path-expression such that

$$\pi \leq \lambda\big((p, w, q)_\rho\big).$$

- $|\pi| = 0$, i.e., $\pi = \bar{1}$. Then necessarily, according to the inequalities (α_1), $p \xrightarrow{w} q$ already holds in $\mathcal{A}$. Therefore, the statement holds with $p' = p$ and $w' = w$.

- $|\pi| > 0$. Then π satisfies $\pi \leq \lambda\big((p, \gamma, q')\big) \odot \lambda\big((q', w_1, q)_{\rho_1}\big)$, where $\rho_1 = q' \xrightarrow{w_1} q$ is the last part of ρ. Since $\pi \neq \bar{1}$, $\lambda\big((p, \gamma, q')\big) \neq \bar{1}$ (notice that the saturation procedure adds transition rules starting from initial states, and the initial automaton does not contain edges leading to initial states), and there exist a rule $r = (p, \gamma) \xhookrightarrow{a} (p_1, u)$ of Δ and two path-expressions π_1 and π_2 such that $\rho' = p_1 \xrightarrow{u} q' \xrightarrow{w_1} q$ is a run of $\mathcal{A}_{pre^*}$, $\pi = v_a \odot \pi_1 \odot \pi_2$, $v_a \odot \pi_1 \leq \lambda\big((p, \gamma, q')\big)$, $\pi_1 \leq \lambda\big((p_1, u, q')\big)$, and $\pi_2 \leq \lambda\big((q', w_1, q)_{\rho_1}\big)$.

  It follows that

$$\begin{aligned} \pi_1 \odot \pi_2 &\leq \lambda\big((p_1, u, q')\big) \odot \lambda\big((q', w_1, q)_{\rho_1}\big) \\ &= \lambda\big((p_1, uw_1, q)_{\rho'}\big) \end{aligned}$$

Since $|\pi_1 \odot \pi_2| < |\pi|$, the induction hypothesis implies that there exists a configuration $\langle p', w' \rangle$ such that $p' \xrightarrow{w'} q$ is a path in $\mathcal{A}$, and $\big(\langle p', w' \rangle, \bar{1}\big) \Leftarrow_K \big(\langle p_1, uw_1 \rangle, \pi_1 \odot \pi_2\big)$. Moreover, by applying r we get that $\big(\langle p_1, uw_1 \rangle, \pi_1 \odot \pi_2\big) \Leftarrow_K \big(\langle p, \gamma w_1 \rangle, v_a \odot \pi_1 \odot \pi_2\big)$. Therefore, since $w = \gamma w_1$ and $\pi = v_a \odot \pi_1 \odot \pi_2$, we get

$$\big(\langle p', w' \rangle, \bar{1}\big) \Leftarrow_K \big(\langle p, w \rangle, \pi\big)$$

where $\langle p', w' \rangle$ is such that $p' \xrightarrow{w'} q$ is a path in $\mathcal{A}$. This ends the proof.

□

5.2. Solving the system of inequations

In this section we consider the problem of computing the tag $l(t)$ for each transition t, i.e., of solving the system of inequations given by (α_1)–(α_4).

Let $t_1, t_2, \ldots, t_m$ be an arbitrary numbering of the transitions of $\mathcal{A}_{pre^*}$, and let $x_1, \ldots, x_m$ be associated variables. Then, $l(t_1), \ldots, l(t_m)$ is the smallest solution of a system of inequalities of the form

$$f_i(x_1, \ldots, x_m) \leq x_i, \qquad 1 \leq i \leq m \tag{3}$$

where the f_i's are monomials in $K[x_1, \ldots, x_m]$, the Kleene algebra of polynomials in indeterminates $x_1, \ldots, x_m$ over K. (It suffices to observe that two different inequalities of the form $e_1 \leq l\big((p, \gamma, q)\big)$ and $e_2 \leq l\big((p, \gamma, q)\big)$ can be replaced by the inequality $e_1 \oplus e_2 \leq l\big((p, \gamma, q)\big)$.)

We show how to solve the system for the two classes of abstractions we consider in this paper.

5.2.1. Finite-chain abstractions

Let $\mathbf{X} = (x_1, \ldots, x_m)$, and F be the function

$$F(\mathbf{X}) = \big(f_1(x_1, \ldots, x_m), \ldots, f_m(x_1, \ldots, x_m)\big).$$

The least solution of (3) is the least pre-fixpoint of F. It is easy to show that F is monotonic and $\oplus$-continuous. Therefore, by Tarski's theorem, the least pre-fixpoint of F exists and is equal to its least fixpoint, and by Kleene's theorem this fixpoint is equal to:

$$\bigoplus_{i \geq 0} F^i(\bar{\mathbf{0}})$$

Then, since $(K, \oplus)$ does not contain any ascending infinite chain, an iterative computation of the least fixpoint always terminates. Notice that the length of the ascending chains is not necessarily bounded. When a maximal length exists, e.g., in the case of finite abstractions, it gives an upper bound on the number of iterations of the algorithm.

5.2.2. Commutative abstractions

In this case $(K, \oplus, \odot, \star, \bar{0}, \bar{1})$ is a commutative Kleene algebra. The system of equations (3) can be solved using Gauss elimination, plus the elegant formula given by Hopkins and Kozen in [10] for solving an equation in one single variable. This is the most adequate procedure for an implementation. However, [10] also provides a fixpoint algorithm to directly solve (3). Since this is more adequate for a complexity estimate, we recall it briefly here. We need some preliminaries.

Let $\mathbf{x} = (\mathbf{x_1}, \ldots, \mathbf{x_n})$ be a vector of indeterminates, let $\mathbf{a} = (\mathbf{a_1}, \ldots, \mathbf{a_n})$ be a vector of elements of K, and let $\mathbf{f} = \mathbf{f_1}, \ldots, \mathbf{f_l}$ be a vector of polynomials with indeterminates $\mathbf{x}$ and coefficients in K. We write $\mathbf{f}(\mathbf{a})$ for the value of $\mathbf{f}$ evaluated at $\mathbf{a}$. The *jacobian* of $\mathbf{f}$, $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}(\mathbf{x})$, is the $l \times n$ matrix whose i, j^{th} element is $\frac{\partial f_i}{\partial x_j}(\mathbf{x})$, where $\frac{\partial}{\partial x_i}$ is the *differential operator* given by

- $\frac{\partial x_i}{\partial x_i} = \bar{1}$, $\frac{\partial x_j}{\partial x_i} = \bar{0}$ for $i \neq j$, and $\frac{\partial a}{\partial x_i} = \bar{0}$ for $a \in K$.
- $\frac{\partial}{\partial x_i}(f \oplus g) = \frac{\partial f}{\partial x_i} \oplus \frac{\partial g}{\partial x_i}$
- $\frac{\partial}{\partial x_i}(f \odot g) = (f \odot \frac{\partial g}{\partial x_i}) \oplus (\frac{\partial f}{\partial x_i} \odot g)$
- $\frac{\partial}{\partial x_i}(f^\star) = f^\star \odot \frac{\partial f}{\partial x_i}$

Then, the least solution of (3) is the fixpoint of the chain $\mathbf{a_0} \leq \mathbf{a_1} \leq \mathbf{a_2} \ldots$[b] given by

$$\begin{aligned} \mathbf{a_0} &= \mathbf{f}(\bar{\mathbf{0}}) \\ \mathbf{a_{k+1}} &= \frac{\partial \mathbf{f}}{\partial \mathbf{x}}(\mathbf{a_k})^\star \odot \mathbf{a_k}, \end{aligned}$$

That is, to compute $\mathbf{a_{k+1}}$ we evaluate the jacobian at the vector $\mathbf{a_k}$, obtaining a matrix over K. Then, we compute the Kleene closure of this matrix[c] using e.g. Floyd-Warshall's algorithm. Finally, we compute the product of the resulting matrix and the vector $\mathbf{a_k}$.

5.3. Example

We give the solution of the system of inequalities of Figure 8 in the case where $(K, \oplus, \odot, \star, \bar{0}, \bar{1})$ is a commutative Kleene algebra. We apply the algorithm provided by [10], where, for example, $f_1(\mathbf{x})$ is the monomial $(x_4 \odot x_9) \oplus (x_8 \odot x_3)$, and $f_4(\mathbf{x})$ is the monomial $(v_a \odot x_1) \oplus (v_b \odot x_2)$. The jacobian $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$ is a 9×9 matrix whose elements are either 0, v_a, v_b, or x_i. For example, the $(1,3)^{th}$ element of this matrix is x_8 (since $\frac{\partial f_1}{\partial x_3} = x_8$), and the $(2,3)^{th}$ element is v_a (since $\frac{\partial f_2}{\partial x_3} = v_a$). The least solution, given by $e_1 = e_5 = e_8 = v_a^\star \odot v_a \odot v_b$, $e_3 = e_6 = e_9 = \bar{1}$, $e_2 = e_7 = v_a$, and

$$e_4 = (v_a \odot v_a^\star \odot v_a \odot v_b) \oplus (v_a \odot v_b)\,,$$

[b] That this is in fact a chain follows easily from the axioms of a Kleene algebra.
[c] The Kleene closure of a matrix A over K is defined by $A^* = \bigoplus_{i \geq 0} A^i$.

is obtained after one iteration.

It follows that the configuration $\langle p, m_0 \rangle$ is tagged with e_4, meaning that the language $L(\langle p, m_0 \rangle, \langle p, m_3 \Gamma^* \rangle)$ is approximated by $e_4 = (v_a \odot v_a^\star \odot v_a \odot v_b) \oplus (v_a \odot v_b)$.

For the three commutative abstractions we have considered, we get:

- In the "label bitvectors" framework, $v_a = \binom{1}{0}$, $v_b = \binom{0}{1}$, and $e_4 = \binom{1}{1}$. This means that we have to execute at least one a and one b to go from m_0 to m_3.
- In the "forbidden and required sets" framework, $v_a = [\{b\}, \{a\}]$ and $v_b = [\{a\}, \{b\}]$, which means that $e_4 = [\emptyset, \{a, b\}]$. This shows that starting from m_0, both a and b are required to reach the point m_3.
- In the "Parikh images" framework, $v_a = \binom{1}{0}$, $v_b = \binom{0}{1}$, and $e_4 = \{\binom{k}{1} \mid k \geq 1\}$. This expresses that the paths leading from m_0 to m_3 contain an arbitrary number (≥ 1) of a's, and exactly one b.

5.4. Complexity of the procedure

It follows easily from (α_1)–(α_4) that the system of inequalities has $O(|\Delta||Q|+|\delta|)$ inequations on $O(|\Delta||Q| + |\delta|)$ variables, where Q and δ are the sets of states and transitions of $\mathcal{A}$, and Δ is the set of transitions of the pushdown automaton.

5.4.1. The finite-chain case

In the former version of this paper, presented at POPL 2003, we gave what turns out to be a very pessimistic estimate of the running time of the tagging procedure in the finite-chain case. Since then, the same problem has been independently studied by Reps, Schwoon, and Jha in [15], where a fast algorithm is presented. In fact, the algorithm even merges the computation of $\mathcal{A}_{pre^*}$ and the tagging phase into one. The algorithm is remarkably similar to the algorithm of [6] for the computation of $\mathcal{A}_{pre^*}$, and has a complexity of $O(h|\Delta||Q|^2 c)$, where h is the maximal height of a chain in the abstract lattice, and c is the cost of a lattice operation.

Let us now study in more details the complexities of the different finite-chain abstractions we have considered:

Label bitvectors: In this case the order $\leq$ is simply set inclusion, and so the longest chain in K has $2^{|Lab|}$ elements. An operation takes $O(2^{|Lab|})$ time. Therefore, the running time of the iterative computation is $2^{O(|Lab|)} \cdot |\Delta||Q|^2$.

First occurrence ordering: As in the previous case, the order $\leq$ is set inclusion. The longest chain in K has $O(2^{|Lab|!})$ elements and an operation takes $O(2^{|Lab|!})$ time. Therefore, the running time of the iterative computation is $2^{O(|Lab|!)} \cdot |\Delta||Q|^2$.

Forbidden and required sets: Since we have $[F_1, R_1] \leq [F_2, R_2]$ if and only if $F_2 \subseteq F_1$ and $R_2 \subseteq R_1$, and since for every $[F, R] \in K$ we have $F, R \subseteq Lab$ and $F \cap R = \emptyset$, the longest possible chain in K has at most length $|Lab|$. If sets of labels are implemented as bitvectors, the $\oplus$ and $\odot$ operations are computed using bitwise

OR and AND, whose cost is constant or $O(|Lab|)$, depending on the application. So the algorithm runs in $O(|Lab|^2 \cdot |\Delta||Q|^2)$ or $O(|Lab| \cdot |\Delta||Q|^2)$ time.

5.4.2. The commutative case

The complexity is dominated by the Hopkins-Kozen algorithm of [10], and we only give a rough estimate. The jacobian matrix has dimension $m \times m$, where m is the number of variables of the system of inequalities. As observed above, $m \in O(|\Delta||Q| + \delta)$. The Kleene closure of $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}(\mathbf{a_k})$ can thus be computed in $O(m^3)$ $\oplus / \odot / \star$-operations using, say, Floyd-Warshall's algorithm.

It is proved in [10] that for an *arbitrary* commutative Kleene algebra the fixpoint of the chain is reached after at most $O(3^m)$ iterations. Since each iteration requires to compute a Kleene closure, the size of the expressions can grow exponentially at each iteration. So the maximal size of an expression can be double exponential in m, and so the cost of an operation is also double exponential in m. This is also the dominating factor in the overall complexity.

5.4.3. Conclusion

As a result of this analysis, we conclude that the forbidden-required analysis has low cost, the first occurrence ordering and label bitvectors analysis have acceptable cost if there are few synchronization statements, and the Parikh-image analysis may have extremely high cost in the worst case. It only seems of practical interest when there are few synchronization statements in loops, since those are responsible for the explosion in the size of the solution.

Note that since the label bitvectors and the forbidden and required sets abstractions are both finite and commutative, we can apply either the Reps-Schwoon-Jha algorithm [15], or the Hopkins-Kozen algorithm. It remains to be seen if the latter can be implemented in such a way that it matches the complexity of the algorithm of the former in these cases. This would yield an algorithm that could be applied to any commutative abstraction without loss of efficiency. It is also worth mentioning that the number of iterations of the Hopkins-Kozen algorithm is always bounded whereas in the algorithm of [15] this number depends on the maximum height of the chains. The algorithm of [10] performs a kind of fixpoint acceleration by introducing $\star$-operations after each iteration, which can be seen as acceleration steps. These accelerations can be useful for finite-chain domains where the chains can be very long.

6. Computing $post^*_K$ images

We present in this section a procedure to compute $post^*_K$ images. It is sligtly more complicated than the procedure for pre^* images, but it uses the main ideas. Even if the complexity of computing $post^*_K$-images is somewhat larger, they turn out to be more useful than pre^*_K images in many practical cases [17]. In particular, the $post^*_K$-image of the initial configuration provides an abstraction of all the execution paths of the program, which is useful in many cases.

Given a regular set of configurations C recognized by an automaton $\mathcal{A} = (\Gamma, Q_0, \delta_0, P, F)$, we construct a K-automaton $\mathcal{A}_{post^*_K}$ which recognizes $post^*_K(C)$. As in the previous section, we assume w.l.o.g. that $\mathcal{A}$ has no transition leading to an initial state.

Again, our algorithm is in two steps. First we construct an automaton $\mathcal{A}_{post^*}$ recognizing $post^*(C)$, and then we tag the transitions of this automaton with elements of K in order to obtain $\mathcal{A}_{post^*_K}$.

For the first step we apply the procedure of [6]. This procedure constructs an automaton with ϵ-transitions $\mathcal{A}_{post^*}$, by computing iteratively the finite sequence $\mathcal{A}'_i$ defined as follows:

- $\mathcal{A}'_0$ is the automaton obtained from $\mathcal{A}$ by adding for each transition rule $r \in \Delta$ of the form $(p,\gamma) \overset{a}{\hookrightarrow} (p', \gamma'\gamma'')$ a new state r and a new transition (p', γ', r),
- for every rule $(p,\gamma) \overset{a}{\hookrightarrow} (p', \epsilon)$ and every state q such that $p \overset{\gamma}{\rightsquigarrow}_i q$, add the new transition $p' \xrightarrow{\epsilon}_{i+1} q$,
- for every rule $(p,\gamma) \overset{a}{\hookrightarrow} (p', \gamma')$ and every state q such that $p \overset{\gamma}{\rightsquigarrow}_i q$, add the new transition $p' \xrightarrow{\gamma'}_{i+1} q$,
- for every rule $(p,\gamma) \overset{a}{\hookrightarrow} (p', \gamma'\gamma'')$ and every state q such that $p \overset{\gamma}{\rightsquigarrow}_i q$, add the new transition $r \xrightarrow{\gamma''}_{i+1} q$,

where $\rightarrow_i$ is the transition relation of $\mathcal{A}'_i$, and $\overset{\gamma}{\rightsquigarrow}$ denotes the relation $\xrightarrow{\epsilon}^* \circ \xrightarrow{\gamma} \circ \xrightarrow{\epsilon}^*$. Observe that the initial automaton $\mathcal{A}$ does not contain transitions leading to an initial state, and the procedure described above does not add such transitions. Moreover, all the ϵ-transitions added by this procedure start from an initial state. Therefore, $\overset{\gamma}{\rightsquigarrow} = \xrightarrow{\gamma} \cup \xrightarrow{\epsilon} \circ \xrightarrow{\gamma}$.

Then, the second step consists in tagging the transitions t of $\mathcal{A}_{post^*}$ with elements $h(t) \in K$ defined as the smallest elements satisfying the following inequalities:

(β_1) if $t = (q, \gamma, q')$ is already a transition of $\mathcal{A}'_0$, then $\bar{1} \leq h(t)$,

(β_2) for every rule $r = (p,\gamma) \overset{a}{\hookrightarrow} (p', \gamma')$ and every $q \in Q$,

$$v_a \odot h'\big((p,\gamma,q)\big) \leq h\big((p',\gamma',q)\big)$$

(β_3) for every rule $r = (p,\gamma) \overset{a}{\hookrightarrow} (p', \epsilon)$ and every $q \in Q$,

$$v_a \odot h'\big((p,\gamma,q)\big) \leq h\big((p',\epsilon,q)\big)$$

(β_4) for every rule $r = (p,\gamma) \overset{a}{\hookrightarrow} (p', \gamma'\gamma'')$ and every $q \in Q$,

$$v_a \odot h'\big((p,\gamma,q)\big) \leq h\big((r,\gamma'',q)\big)$$

where $h'\big((p,\gamma,q)\big)$ denotes the summary of the labels of all the runs of $\mathcal{A}_{post^*}$ of the form $p \overset{\gamma}{\rightsquigarrow} q$, i.e. (since $\overset{\gamma}{\rightsquigarrow} = \xrightarrow{\gamma} \cup \xrightarrow{\epsilon} \circ \xrightarrow{\gamma}$),

$$h'\big((p,\gamma,q)\big) = h\big((p,\gamma,q)\big) \oplus \bigoplus_{q' \in Q} \Big(h\big((p,\epsilon,q')\big) \odot h\big((q',\gamma,q)\big) \Big)$$

Intuitively, the inequalities defining the mapping h express that, for every rule $r = (p,\gamma) \overset{a}{\hookrightarrow} (p',u)$, and every $q \in Q$, if w is such that $q \xrightarrow{w} q'$ for some final state q', and if $(\langle p,\gamma w\rangle, \pi)$ is a K-successor of $Conf(\mathcal{A})$, then so is $(\langle p',\gamma w\rangle, v_a \odot \pi)$.

To compute the tagging $h(t)$ for each transition t, we proceed as in the previous section. We consider an arbitrary numbering of the transitions of $\mathcal{A}_{post^*}$, $t_1, t_2, \ldots, t_m$, and let $x_1, \ldots, x_m$ be associated variables. Then, by (β_1)–(β_4), we have that $l(t_1), \ldots, l(t_m)$ is the smallest solution of a system of inequalities of the form

$$g_i(x_1, \ldots, x_m) \leq x_i, \qquad 1 \leq i \leq m \tag{4}$$

where the g_i's are monomials in $K[x_1, \ldots, x_m]$. These inequations can be solved in the cases of finite-chain and commutative abstractions in the same way as we did for the case of pre^*-images. Then we get the following:

Theorem 2 *Given a PDS $\mathcal{P}$ and a regular set of configurations recognized by an automaton $\mathcal{A}$, the procedure above constructs a K-automaton $\mathcal{A}_{post^*_K}$ which recognizes $post^*_K\big(Conf(\mathcal{A})\big)$.*

We show that if $\mathcal{A}_{post^*} = (\Gamma, Q, \delta, P, F)$ is the automaton defined in [6] recognizing $post^*\big(Conf(\mathcal{A})\big)$, then $\mathcal{A}_{post^*_K} = (\Gamma, Q, \delta', P, F)$ where

$$\delta' = \{(q,\gamma,e,q') \mid (q,\gamma,q') \in \delta, e = h\big((q,\gamma,q')\big)\}.$$

Notation 6.1 *Let ρ be a run of $\mathcal{A}_{post^*_K}$ leading from p to q and having (w,e) as label. Abusing notation we denote this by $\rho = p \xrightarrow{(w,e)} q$. We denote the expression e by $\mu\big((p',w,q)_\rho\big)$. If $w = \gamma \in \Gamma$, i.e., if ρ corresponds to one single transition, then we simply write $e = \mu\big((p',\gamma,q)\big)$.*

Theorem 2 follows from the next two lemmas:

Lemma 3 *For every configuration $\langle p,v\rangle \in Conf(\mathcal{A})$, if $\big(\langle p,v\rangle, \bar{1}\big) \Rightarrow_K \big(\langle p',w\rangle, \pi\big)$ then there exists $e \in K$ such that $\pi \leq e$ and $p' \xrightarrow{(w,e)} q$ for some final state q of $\mathcal{A}_{post^*_K}$.*

Proof: Let $\langle p,v\rangle \in Conf(\mathcal{A})$. We will show that if $\big(\langle p,v\rangle, \bar{1}\big) \Rightarrow_K \big(\langle p',w\rangle, \pi\big)$ then there exists a final state q such that $\rho = p' \xrightarrow{w} q$ is a path of $\mathcal{A}_{post^*}$ and $\pi \leq \mu\big((p',w,q)_\rho\big)$.

Let $\big(\langle p,v\rangle, \bar{1}\big) \overset{k}{\Rightarrow}_K \big(\langle p',w\rangle, \pi\big)$ mean that $\big(\langle p',w\rangle, \pi\big)$ is a K-successor of $\big(\langle p,v\rangle, \bar{1}\big)$ after k steps. We show the statement by induction on k.

- $k = 0$. Then $p' = p$, $w = v$, and $\pi = \bar{1}$. Since $\langle p, v\rangle \in \mathit{Conf}(\mathcal{A})$, then there exists a final state q such that $\rho = p \xrightarrow{v} q$ is already a path of $\mathcal{A}$ (and hence of $\mathcal{A}_{post^*}$). Therefore, rules (β_1) imply that $\bar{1} \leq \mu\big((p, v, q)_\rho\big)$.

- $k > 0$. Let then $p'' \in P, u \in \Gamma^*, \pi' \in \Pi_K$ such that

$$\big(\langle p, v\rangle, \bar{1}\big) \stackrel{k-1}{\Rightarrow}_K \big(\langle p'', u\rangle, \pi'\big) \stackrel{1}{\Rightarrow}_K \big(\langle p', w\rangle, \pi\big)$$

  By induction hypothesis there exists a path $\rho' = p'' \xrightarrow{u} q$ of $\mathcal{A}_{post^*}$ such that q is a final state and $\pi' \leq \mu\big((p'', u, q)_{\rho'}\big)$.

  Since $\big(\langle p'', u\rangle, \pi'\big) \stackrel{1}{\Rightarrow}_K \big(\langle p', w\rangle, \pi\big)$, there are $\gamma \in \Gamma, w_1, u_1 \in \Gamma^*$ and a rule $r = (p'', \gamma) \stackrel{a}{\hookrightarrow} (p', w_1)$ of Δ such that $u = \gamma u_1$, $w = w_1 u_1$, and $\pi = v_a \odot \pi'$. Let then $q' \in Q$ such that $\rho' = p'' \stackrel{\gamma}{\rightsquigarrow} q' \xrightarrow{u_1} q$. Let $\rho_1 = p'' \stackrel{\gamma}{\rightsquigarrow} q'$ and $\rho_2 = q' \xrightarrow{u_1} q$ be the two parts of ρ'. There are two cases depending on whether $|w_1| < 2$ or $|w_1| = 2$:

  - $|w_1| < 2$. From the saturation rules, it follows that there is a transition $p' \xrightarrow{w_1} q'$ in $\mathcal{A}_{post^*}$, which means that $\rho = p' \xrightarrow{w_1} q' \xrightarrow{u_1} q$ is a path in $\mathcal{A}_{post^*}$. From rules (β_2) and (β_3), it follows that

$$v_a \odot \mu\big((p'', \gamma, q')_{\rho_1}\big) \leq \mu\big((p', w_1, q')\big)$$

    Therefore, since $w_1 u_1 = w$, $\rho = p' \xrightarrow{w} q$ is an accepting path of $\mathcal{A}_{post^*}$, and

$$\begin{aligned}
\pi &= v_a \odot \pi' \\
&\leq v_a \odot \mu\big((p'', u, q)_{\rho'}\big) \\
&= v_a \odot \mu\big((p'', \gamma, q')_{\rho_1}\big) \odot \mu\big((q', u_1, q)_{\rho_2}\big) \\
&\leq \mu\big((p', w_1, q')\big) \odot \mu\big((q', u_1, q)_{\rho_2}\big) \\
&= \mu\big((p', w, q)_\rho\big)
\end{aligned}$$

  - $|w_1| = 2$, let then $w_1 = \gamma'\gamma''$. It follows that there is a path

$$p' \xrightarrow{\gamma'} r \xrightarrow{\gamma''} q'$$

    in $\mathcal{A}_{post^*}$, which means that $\rho = p' \xrightarrow{\gamma'} r \xrightarrow{\gamma''} q' \xrightarrow{u_1} q$ is a path in $\mathcal{A}_{post^*}$. Rules (β_1) imply that $\bar{1} \leq \mu\big((p', \gamma', r)\big)$, and from rules (β_4) it follows that

$$v_a \odot \mu\big((p'', \gamma, q')_{\rho_1}\big) \leq \mu\big((r, \gamma'', q')\big)$$

    Therefore, $\rho = p' \xrightarrow{w} q$ is an accepting path of $\mathcal{A}_{post^*}$ (since $\gamma'\gamma''u_1 =$

$w_1u_1 = w$) and π satisfies:

$$\begin{aligned}
\pi &= \bar{1} \odot v_a \odot \pi' \\
&\leq \mu\big((p', \gamma', r)\big) \odot v_a \odot \mu\big((p'', u, q)_{\rho'}\big) \\
&= \mu\big((p', \gamma', r)\big) \odot v_a \odot \mu\big((p'', \gamma, q')_{\rho_1}\big) \odot \mu\big((q', u_1, q)_{\rho_2}\big) \\
&\leq \mu\big((p', \gamma', r)\big) \odot \mu\big((r, \gamma'', q')\big) \odot \mu\big((q', u_1, q)_{\rho_2}\big) \\
&= \mu\big((p', w, q)_{\rho}\big)
\end{aligned}$$

□

Lemma 4 *Let $p \xrightarrow{(w,e)} q$ be a path of $\mathcal{A}_{post^*_K}$ such that $q \in Q_0$, then for every path-expression $\pi \leq e$, there exists a configuration $\langle p', w' \rangle$ such that $p' \xrightarrow{w'} q$ is a path in $\mathcal{A}$, and $\big(\langle p', w' \rangle, \bar{1}\big) \Rightarrow_K \big(\langle p, w \rangle, \pi\big)$.*

Proof: We will show by induction on $|\pi|$ that if $\pi \leq \mu\big((p, w, q)_\rho\big)$ for some run $\rho = p \xrightarrow{w} q$ of $\mathcal{A}_{post^*}$ such that $q \in Q_0$, then there exists a configuration $\langle p', w' \rangle$ such that $p' \xrightarrow{w'} q$ is a path in $\mathcal{A}$ and $\big(\langle p', w' \rangle, \bar{1}\big) \Rightarrow_K \big(\langle p, w \rangle, \pi\big)$.

Let $\rho = p \xrightarrow{w} q$ be a path of $\mathcal{A}_{post^*}$ such that $q \in Q_0$, and let π be a path-expression such that $\pi \leq \mu\big((p, w, q)_\rho\big)$.

- $|\pi| = 0$, i.e., $\pi = \bar{1}$. Then necessarily, according to the inequalities (β_1), $\rho = p \xrightarrow{w} q$ is a run of $\mathcal{A}'_0$ and hence, it is already a run of $\mathcal{A}$ since $\mathcal{A}'_0$ contains the transitions of $\mathcal{A}$ together with transitions of the form $p \xrightarrow{\gamma} r$ where $r \notin Q_0$. It follows that in $\mathcal{A}'_0$ there is no run leading to a state in Q_0 that passes through a state r. The path ρ considers then only transitions of $\mathcal{A}$. Therefore, the statement holds with $p' = p$ and $w' = w$.

- $|\pi| > 0$. There are three cases:

  - There exist $\gamma \in \Gamma$, $w_1 \in \Gamma^*$, and $q' \in Q_0$ $(q' \neq r)$, $w = \gamma w_1$ and

    $$\rho = p \xrightarrow{\gamma} q' \xrightarrow{w_1} q.$$

    π satisfies $\pi \leq \mu\big((p, \gamma, q')\big) \odot \mu\big((q', w_1, q)_{\rho_1}\big)$, where $\rho_1 = q' \xrightarrow{w_1} q$. From the fact that $\pi \neq \bar{1}$, and according to the inequalities (β_2), there exists a rule $r = (p_1, \gamma_1) \xhookrightarrow{a} (p, \gamma)$ of Δ, a run $\rho' = p_1 \overset{\gamma_1}{\rightsquigarrow} q' \xrightarrow{w_1} q$ of $\mathcal{A}_{post^*}$, and two path-expressions π_1 and π_2 such that $\pi = v_a \odot \pi_1 \odot \pi_2$, $v_a \odot \pi_1 \leq \mu\big((p, \gamma, q')\big)$, $\pi_1 \leq \mu\big((p_1, \gamma_1, q')_{\rho_2}\big)$, and $\pi_2 \leq \mu\big((q', w_1, q)_{\rho_1}\big)$, where ρ_2 is a path of the form $\rho_2 = p_1 \overset{\gamma_1}{\rightsquigarrow} q'$.
    It follows that

    $$\begin{aligned}
    \pi_1 \odot \pi_2 &\leq \mu\big((p_1, \gamma_1, q')_{\rho_2}\big) \odot \mu\big((q', w_1, q)_{\rho_1}\big) \\
    &= \mu\big((p_1, \gamma_1 w_1, q)_{\rho'}\big)
    \end{aligned}$$

Sine $|\pi_1 \odot \pi_2| < |\pi|$, the induction hypothesis implies that there exists a configuration $\langle p', w' \rangle$ such that $p' \xrightarrow{w'} q$ is a run of $\mathcal{A}$ and

$$\big(\langle p', w' \rangle, \bar{1}\big) \Rightarrow_K \big(\langle p_1, \gamma_1 w_1 \rangle, \pi_1 \odot \pi_2\big).$$

Moreover, by applying r we get that

$$\big(\langle p_1, \gamma_1 w_1 \rangle, \pi_1 \odot \pi_2\big) \Rightarrow_K \big(\langle p, \gamma w_1 \rangle, v_a \odot \pi_1 \odot \pi_2\big).$$

Therefore, since $w = \gamma w_1$ and $\pi = v_a \odot \pi_1 \odot \pi_2$, we get $\big(\langle p', w' \rangle, \bar{1}\big) \Rightarrow_K \big(\langle p, w \rangle, \pi\big)$, where $\langle p', w' \rangle$ is such that $p' \xrightarrow{w'} q$ is a run of $\mathcal{A}$. i.e. $\big(\langle p', w' \rangle, \bar{1}\big) \Rightarrow_K \big(\langle p, w \rangle, \pi\big)$, since $w = \gamma w_1$ and $\pi = v_a \odot \pi_1 \odot \pi_2$.

- The case where $\rho = p \xrightarrow{\epsilon} q' \xrightarrow{w_1} q$ is similar. In this case, r is of the form $r = (p_1, \gamma_1) \overset{a}{\hookrightarrow} (p, \epsilon)$.
- There exist $\gamma_1, \gamma_2 \in \Gamma$, $w_1 \in \Gamma^*$, a rule r, and $q' \in Q$ such that $w = \gamma_1 \gamma_2 w_1$ and
$$\rho = p \xrightarrow{\gamma_1} r \xrightarrow{\gamma_2} q' \xrightarrow{w_1} q.$$
π satisfies $\pi \leq \mu\big((p, \gamma_1, r)\big) \odot \mu\big((r, \gamma_2, q')\big) \odot \mu\big((q', w_1, q)_{\rho_1}\big)$, where $\rho_1 = q' \xrightarrow{w_1} q$. According to the inequalities (β_1) and (β_4), r is of the form $r = (p_1, \gamma) \overset{a}{\hookrightarrow} (p, \gamma_1\gamma_2)$ of Δ, and there exist two path-expressions π_1 and π_2 such that $\pi = \bar{1} \odot v_a \odot \pi_1 \odot \pi_2$, $v_a \odot \pi_1 \leq \mu\big((r, \gamma_2, q')\big)$, $\pi_1 \leq \mu\big((p_1, \gamma, q')_{\rho_2}\big)$, and $\pi_2 \leq \mu\big((q', w_1, q)_{\rho_1}\big)$, where ρ_2 is a path of the form $\rho_2 = p_1 \overset{\gamma}{\rightsquigarrow} q'$. Indeed, the inequalities infer that $\mu\big((p, \gamma_1, r)\big) = \bar{1}$. Let $\rho' = p_1 \overset{\gamma}{\rightsquigarrow} q' \xrightarrow{w_1} q$ be the corresponding run of $\mathcal{A}_{post^*}$. It follows that

$$\begin{array}{rcl} \pi_1 \odot \pi_2 & \leq & \mu\big((p_1, \gamma, q')_{\rho_2}\big) \odot \mu\big((q', w_1, q)_{\rho_1}\big) \\ & = & \mu\big((p_1, \gamma w_1, q)_{\rho'}\big) \end{array}$$

Sine $|\pi_1 \odot \pi_2| < |\pi|$, the induction hypothesis implies that there exists a configuration $\langle p', w' \rangle$ such that $p' \xrightarrow{w'} q$ is a run of $\mathcal{A}$ and

$$\big(\langle p', w' \rangle, \bar{1}\big) \Rightarrow_K \big(\langle p_1, \gamma w_1 \rangle, \pi_1 \odot \pi_2\big).$$

Moreover, by applying r we get that

$$\big(\langle p_1, \gamma w_1 \rangle, \pi_1 \odot \pi_2\big) \Rightarrow_K \big(\langle p, \gamma_1\gamma_2 w_1 \rangle, v_a \odot \pi_1 \odot \pi_2\big)$$

i.e. $\big(\langle p', w' \rangle, \bar{1}\big) \Rightarrow_K \big(\langle p, w \rangle, \pi\big)$, since $w = \gamma_1\gamma_2 w_1$ and $\pi = v_a \odot \pi_1 \odot \pi_2$.

□

6.1. Complexity of the procedure

In the finite-chain case, and using again the ideas of [15], an algorithm that merges the computation of $\mathcal{A}_{post^*}$ and the tagging phase into one can be designed. The algorithm is a generalization of the algorithm of [6] for the computation of $\mathcal{A}_{post^*}$, and has a complexity of $O(h \cdot (|P||\Delta|(|Q| + |\Delta|) + |P||\delta|) \cdot c)$, where h is the maximal height of a chain in the abstract lattice, P and Δ are the sets of control states and transitions rules of the PDS, Q and δ are the sets of states and transitions of $\mathcal{A}$, and c is the cost of a lattice operation.

In the commutative case essentially the same analysis as before applies.

7. Conclusions and related work

We have presented a generic approach to the static analysis of concurrent programs with procedures. The effect of the procedures is determined exactly, while the constraints on program paths induced by synchronizations is only approximated. Due to the undecidability result of [14], approximation techniques must be used. The gist of our technique is to approximate context-free languages representing all program paths between two regular sets of configurations by languages closed under intersection, what allows to obtain an upper approximation of the sequences of communication actions executed by the program. We have shown how to compute a system of inequalities for (an abstraction of) the context-free language of program paths between two arbitrary regular sets of configurations. If the abstract and concrete lattices coincide, then the system of inequalities is nothing but a context-free grammar in disguise. Using algebraic results we have provided two generic algorithms for the computation of the approximations, and we have shown four possible instantiations leading to analysis of different precision and cost.

Work on static analysis of concurrent programs goes back to Taylor's seminal paper [18], which showed that many problems are NP-complete even in the intraprocedural case (other problems are even PSPACE-complete) and proposed a general analysis algorithm for different problems. Since then, the intraprocedural case has received considerable attention, and [16] is a recent survey. In our framework, this case corresponds to communicating finite automata instead of communicating pushdown systems. The set of program paths of each component is regular, and, since regular languages are closed under intersection, so is the overall set of program paths. Approximations are used to reduce the computational cost. The approach of Corbett [3], in which program paths are approximated using integer linear programming, is less precise than our Parikh image approximation, but computationally more efficient. In [12], Mercouroff proposes an analysis based on counting the number of communications that have occurred in the different channels of the system. The computed approximations are less precise than ours since they are based on a forward analysis which terminates due to rough extrapolation techniques. The approach of [13] is especially interesting for us. It avoids an exponential state based analysis by means of what can be seen as the definition of a weak product of finite automata. This technique is orthogonal to the ones shown here, and can be adapted to our pushdown automata model.

The only paper to explicitly study concurrency analysis in the presence of proce-

dures seems to be [4]. The paper provides an approximate analysis for determining which statements can be concurrently executed. If statements a and b can be concurrently executed, then it is possible to execute a before b and b before a. The analysis of [4] computes for each statement a a set of statements B such that for every $b \in B$ and every program path, either every occurrence of b appears before every occurrence of a, or every occurrence of a appears before every occurrence of b. We have then that a cannot be concurrently executed with any occurrence of b.

A very similar analysis can also be carried out in our framework, and we sketch how. First, we say that $a_1 \dots a_n$ is a subword of a language L if L contains a word of the form

$$u_0 a_1 u_1 \dots u_{n-1} a_n u_n.$$

Our abstract lattice has as elements pairs (Lab', D), where $Lab' \subseteq Lab$ and $D\colon Lab' \times Lab' \to \{\mathbf{none}, \mathbf{12}, \mathbf{21}, \mathbf{both}\}$. Given a path language $L(C', C)$, we have $\alpha(L(C', C)) = (Lab', D)$, where Lab' is the set of actions that occur in $L(C', C)$, and

- $D(a, b) = \mathbf{none}$ if neither ab nor ba are subwords of $L(C', C)$;
- $D(a, b) = \mathbf{12}$ if ab is a subword of $L(C', C)$, but ba is not;
- $D(a, b) = \mathbf{21}$ if ba is a subword of $L(C', C)$, but ab is not;
- $D(a, b) = \mathbf{both}$ if both ab and ba are subwords of $L(C', C)$.

It is easy to define operations $\oplus$ and $\odot$ corresponding to union and concatenation of languages, as well as the operation $\sqcap$ corresponding to intersection. Since the lattice is finite, the approximation can be effectively computed. If the approximation says that $D(a, b) \neq \mathbf{both}$, then we know that a and b cannot be concurrently executed in any program path from C' to C. Notice that in this case we are not interested in determining if the intersection is empty, but in actually computing it.

The main advantage of our approach over that of [4] is its genericity. For instance, we can combine the approximation of [4] with our first occurrence approximation to obtain a more precise analysis. Also, we do not need to prove the correctness of dataflow equations. Notice, however, that in [4] dynamic task creation is allowed, which is not possible in our current setting. In fact, concurrency analysis with task creation has been further analyzed in recent papers [1, 19]. While procedures can be approximated by considering them as new tasks, the analysis is not sensitive to multiple incarnations of the procedure. A possibility to add dynamic creation to our setting is to use the technique of [8, 11], and we plan to investigate this in the future.

Acknowledgments

We thank Luca Aceto for helpful discussions on commutative semirings. Sriram Rajamani and Stefan Schwoon provided very valuable comments that greatly helped to improve the paper.

References

1. J. Blieberger, B. Burgstaller, and B. Scholz. Symbolic Data Flow Analysis for Detecting Deadlocks in Ada Tasking Programs. In *Proc. of the Ada-Europe International Conference on Reliable Software Technologies*, Potsdam, Germany, 2000.
2. A. Bouajjani, J. Esparza, and O. Maler. Reachability Analysis of Pushdown Automata: Application to Model Checking. In *CONCUR'97*, volume 1243 of *LNCS*, 1997.
3. J. C. Corbett. Automated Formal Analysis Methods for Concurrent and Real-Time Software. PhD thesis, University of Massachusetts at Amherst, 1992.
4. E. Duesterwald and M. Soffa. Concurrency analysis in the presence of of procedures using a data-flow framework. In *Proc. of the Symposium on Testing, Analysis, and Verification, Victoria, Canada*, ACM Press, 1991.
5. M. B. Dwyer and L. A. Clarke. Data flow analysis for verifying properties of concurrent programs. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, ACM Press, 1994.
6. J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithm for model checking pushdown systems. In *CAV'00*, volume 1885 of *LNCS*, 2000.
7. J. Esparza and J. Knoop. An automata-theoretic approach to interprocedural data-flow analysis. In W. Thomas, editor, *Proc of Foundations of Software Science and Computation Structure, FoSSaCS'99*, volume 1578 of *Lecture Notes in Computer Science*. Springer, 1999.
8. J. Esparza and A. Podelski. Efficient algorithms for pre* and post* on interprocedural parallel flow graphs. In *Proceedings of the 27th ACM SIGPLAN-SIGACT on Principles of Programming Languages, POPL 2000*, ACM Press, 2000.
9. S. Ginsburg and E.H. Spanier. Semigroups, Presburger formulas and languages In Pacific Journal of Mathematics, volume 16, 1966.
10. M. Hopkins and D. Kozen. Parikh's Theorem in Commutative Kleene Algebra. In *Proc. IEEE Conf. Logic in Computer Science (LICS'99)*. IEEE, 1999.
11. D. Lugiez and P. Schnoebelen. The Regular Viewpoint on PA-processes. In *TCS*, volume 274(1-2), 2002.
12. N. Mercouroff. An algorithm for analyzing communicating processes. In *Mathematical Foundations of Programming Semantics*, volume 598 of *LNCS*, 1991.
13. G. Naumovich and G. Avrunin. A conservative data flow algorithm for detecting all pairs of statement that may happen in parallel. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM Press, 1998.
14. G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, volume 22, 2000.
15. T. Reps, S. Schwoon and S. Jha. Weighted Pushdown Systems and Their Application to Interprocedural Dataflow Analysis To appear in: *Proc. of the 10th International Symposium on Static Analysis, SAS 2003*, 2003.
16. M. Rinard. Analysis of multithreaded programs. In P. Cousot, editor, *Proc. of the 8th International Symposium on Static Analysis, SAS 2001*, volume 2126 of *LNCS*, 2001.
17. S. Schwoon. Model Checking Pushdown Systems. Ph.D. Thesis, Technical University of Munich, 2002.
18. R. Taylor. A general-purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, volume 26, 1983.

19. E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. *ACM SIGPLAN Notices*, volume 36(3), 2001.