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

Ahmed Bouajjani* Javier Esparza† Tayssir Touili*

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.

Categories and Subject Descriptors

D.2.4. [Software/Program Verification], F.1.1 [Models of Computation], F.3.1. [Specifying and Verifying and Reasoning about Programs], F.4.2. [Grammars and Other Rewriting Systems].

General Terms

Algorithms, Reliability, Theory, Verification.

Keywords

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

*Liafa, University of Paris 7, Email: {Ahmed.Bouajjani,Tayssir.Touili}@liafa.jussieu.fr

†Lab. for Foundations of Computer Science,University of Edinburgh. Email: jav@dcs.ed.ac.uk

POPL'03, January 15–17, 2003, New Orleans, Louisiana, USA.

1. INTRODUCTION

Multithreading 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 [14] 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 basic result [13] 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 [13]. Consider a sequential program with possibly recursive procedures and one of its configurations c. Here, 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 sequences 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 [9] for solving systems of equations in commutative Kleene algebras. We give several examples of useful abstractions leading to analysis algorithms of different precision and cost.

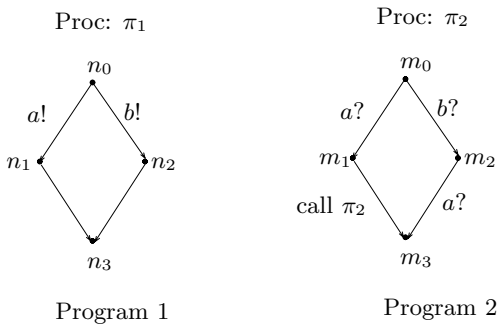

Figure 1: An example

The paper is organised 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. Finally, section 6 contains conclusions and discusses related work. For lack of space, all proofs are omitted and can be found in the full version of the paper.

2. THE MODEL

2.1 Program model: Flow Graph Systems

Our program model is very similar to the trace flow graph of [5] or [12]. 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 annotated with statements. An unlabelled statement corresponds to a noop. We allow 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. A concurrent program is represented by a tuple of flow graph systems, one for each sequential component.

Statements are assignments, calls to other procedures of the same sequential program, or communications with another sequential program. 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 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 $\overset{a}{\longrightarrow} \subseteq \mathcal{C} \times \mathcal{C}$ as follows: if $\langle q, \gamma\rangle \overset{a}{\hookrightarrow} \langle q', w\rangle$, then $\langle q, \gamma v\rangle \overset{a}{\longrightarrow} \langle q', wv\rangle$ 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. These actions in Lab will be 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 $\overset{a}{\longrightarrow}$ to pairs of global configurations as follows. Let $g = (c_1, \ldots, c_n)$ and $g' = (c'_1, \ldots, c'_n)$ be global configurations:

- $g \overset{\tau}{\longrightarrow} g'$ if there is $1 \leq i \leq n$ such that $c_i \overset{\tau}{\longrightarrow} c'_i$ and $c'_j = c_j$ for each $j \neq i$;
- $g \overset{a}{\longrightarrow} g'$ there are indices $i \neq j$ such that $c_i \overset{a}{\longrightarrow} c'_i$, $c_j \overset{a}{\longrightarrow} c'_j$, 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 simplicity, we assume that all procedures of the flow-graph system have the same local variables. The corresponding PDS $\mathcal{P} = (P, Act, \Gamma, c_0, \Delta)$ is defined as follows. P is the set of all possible valuations of the local 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. We take $c_0 = \langle glob_0, (n_0, loc_0)\rangle$, 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

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

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

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

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 elements. The PDS for the sequential program on the right-hand-side has one single control location p and rules $r_1 : \langle p, m_0\rangle \overset{a}{\hookrightarrow} \langle p, m_1\rangle$, $r_2 : \langle p, m_0\rangle \overset{b}{\hookrightarrow} \langle p, m_2\rangle$, $r_3 : \langle p, m_1\rangle \overset{\tau}{\hookrightarrow} \langle p, m_0 m_3\rangle$, $r_4 : \langle p, m_2\rangle \overset{a}{\hookrightarrow} \langle p, m_3\rangle$, and $r_5 : \langle p, m_3\rangle \overset{\tau}{\hookrightarrow} \langle p, \epsilon\rangle$.

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 a set of configurations $C'_1 \times \cdots \times C'_n$. 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

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

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

$$\widehat{L_1} \cap \ldots \cap \widehat{L_n} \neq \emptyset \tag{1}$$

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. Therefore, 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 S 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

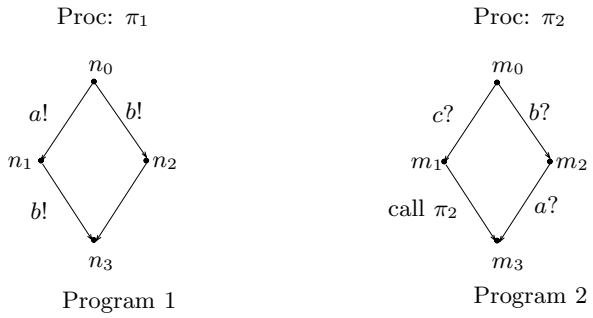

Figure 2: Another example

$w = a_1 \cdots a_n$ belongs to S if there is a path in $L(C', C)$ such that the set of letters occurring in this path is precisely $\{a_1, \ldots, a_n\}$, and moreover, the first occurrences of these letters in the path occur in the ordering 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.

In Figure 2 we can conclude that no global configuration with n_3 and m_3 as control locations is reachable: We have $S(n_3) = \{ab, b\}$, while $S(m_3) = \{cba, ba\}$.

3.2.2 Label bitvectors

Let us forget about the order in which the first occurrence of each letter appears. The abstract object representing $A(C', C)$ is now a set S of bitvectors $Lab \to \mathbb{B}$. A bitvector b belongs to S 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 conclude that no global configuration with n_3 and m_3 as control locations is reachable: We have $S(n_3) = \{\binom{1}{0}, \binom{0}{1}\}$, while $S(m_3) = \{\binom{1}{1}\}$ (the components of the bitvectors correspond to the labels a, b).

3.2.3 Forbidden and required sets

The abstract object representing $A(C', C)$ 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 3 we can use this information 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 repre-

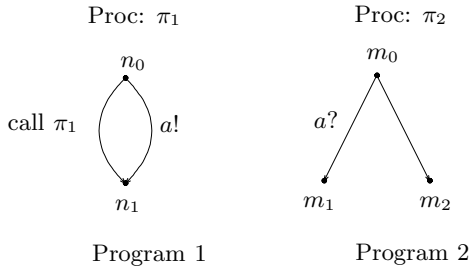

Figure 3: Yet Another example

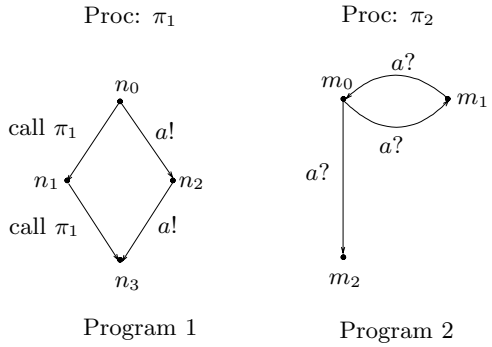

Figure 4: A last example

sentable) set M of integer vectors $Lab \to \mathbb{N}$. A 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.[1]

In Figure 4 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 4, 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 requires an *abstract lattice* $\mathcal{D} = (D, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$, where D is some 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

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

Our purpose is to define abstractions 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 abstractions where the carrier D of the lattice is an idempotent closed semiring. An *idempotent semiring* is a structure $\mathcal{K} = (K, \oplus, \odot, \bar{0}, \bar{1})$, 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 $\oplus$. 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). As usual, we 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*. The semiring of an abstraction is generated by $\bar{0}$, $\bar{1}$ and an element v_a for each $a \in Lab$.

Intuitively, the abstract operations $\oplus$ and $\odot$ of $\mathcal{K}$ correspond to union and concatenation 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 corresponds to the language $\{a\}$.

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. Intuitively, the top element $\top \in K$ and the meet operation $\sqcap$ correspond in the lattice $\mathcal{L}$ to Lab^* and to language intersection, respectively.

The Galois connection of an abstraction is given by $\alpha : 2^{Lab^*} \to K$ and $\gamma : K \to 2^{Lab^*}$ such that

$$\alpha(L) = \bigoplus_{a_1 \cdots a_n \in L} v_{a_1} \odot \cdots \odot v_{a_n}$$

$$\gamma(x) = \{a_1 \cdots a_n \in 2^{Lab^*} \mid v_{a_1} \odot \cdots \odot v_{a_n} \leq x\}$$

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

In this paper, we consider two families of abstractions:

3.3.1 Finite-chain 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.

[1] It is well-known that Parikh-images of context-free languages are semi-linear sets of integer vectors (i.e., unions of sets of the form $\{\vec{u} + k_1\vec{u}_1 + \cdots + k_n\vec{u}_n \mid k_1, \ldots, k_n \in \mathbb{N}\}$), or equivalently, Presburger arithmetics definable sets.

3.3.2 Commutative abstractions

A *commutative abstraction* is an abstraction where $\odot$ is commutative. Intuitively, this means that $\odot$ "forgets the order" of labels. Thus we get that $(K, \oplus, \odot, \star, \overline{0}, \overline{1})$ is a *commutative Kleene algebra*.

3.4 Instances of abstractions

We show that the four examples of section 3.2 fall in our classes of abstractions.

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 semiring $\mathcal{K}$ given by: (1) $K = 2^W$; this set is generated by the elements v_a for each a in Lab, where $v_a = \{a\}$, (2) $\leq = \subseteq$, (3) $\oplus = \cup$, (4) $U \odot V = \{u_1 \cdot v_2' \mid u_1 \in U \text{ and } \exists v_2 \in V.\ v_2' \text{ is the projection of } v_2 \text{ on the set of letters which do not occur in } u_1\}$, (5) $\overline{0} = \emptyset$, and (6) $\overline{1} = \{\epsilon\}$. The abstract lattice is obtained by taking $\top = W$, and $\sqcap = \cap$. Observe that this is a finite abstraction since K is finite.

3.4.2 Label bitvectors

We consider the semiring $\mathcal{K}$ given by: (1) $K = 2^{[Lab \to \mathbb{B}]}$, where $[Lab \to \mathbb{B}]$ is the set of bitvectors indexed by Lab. K is 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 $a' \neq a$, (2) $\leq = \subseteq$, (3) $\oplus = \cup$, (4) $\odot = \vee$ (extended to vectors componentwise, and to sets of vectors), (5) $\overline{0} = \emptyset$, and (6) $\overline{1} = \{(\mathit{false}, \ldots, \mathit{false})\}$. The abstract lattice is obtained by taking $\top = 2^{[Lab \to \mathbb{B}]}$, and $\sqcap = \cap$. Notice that this abstraction is both finite and commutative.

3.4.3 Forbidden and required sets

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

- $K = \{\overline{0}\} \cup \{[F, R] \in 2^{Lab} \times 2^{Lab} \mid F \cap R = \emptyset\}$, i.e., the set of all pairs of sets of actions generated by the elements v_a for each $a \in Lab$, where $v_a(a) = [Lab \setminus \{a\}, \{a\}]$, augmented with a special element $\overline{0}$,
- $[F_1, R_1] \leq [F_2, R_2]$ iff $F_1 \supseteq F_2$ and $R_1 \supseteq R_2$,
- $[F_1, R_1] \oplus [F_2, R_2] = [F_1 \cap F_2, R_1 \cap R_2]$,
- $[F_1, R_1] \odot [F_2, R_2] = [F_1 \cap F_2, R_1 \cup R_2]$,
- $\overline{0}$ is by definition neutral for $\oplus$ and annihilator for $\odot$,
- $\overline{1} = [Lab, \emptyset]$.

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 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 $\{\vec{u} + k_1\vec{v}_1 + \ldots k_n\vec{v}_n \mid k_1 \ldots k_n \in \mathbb{N}\}$, where $\vec{u}, \vec{v}_1, \ldots, \vec{v}_n \in [Lab \to \mathbb{N}]$ ($\vec{u}$ is the basis, and the $\vec{v}_i$'s are the periods). We represent semilinear sets as sets of pairs $(\vec{u}, \{\vec{v}_1, \ldots, \vec{v}_n\})$, and we consider the commutative abstraction where:

- K is the set of sets of pairs $(\vec{u}, \Omega) \subseteq [Lab \to \mathbb{N}] \times 2^{[Lab \to \mathbb{N}]}$ generated by the elements $v_a = \{(\vec{u}_a, \emptyset)\}$, for each $a \in Lab$, where $\vec{u}_a(a) = 1$ and $\vec{u}_a(b) = 0$ for every $b \neq a$.
- $\oplus = \cup$,
- $(\vec{u}_1, \Omega_1) \odot (\vec{u}_2, \Omega_2) = (\vec{u}_1 + \vec{u}_2, \Omega_1 \cup \Omega_2)$, generalized to sets in the straightforward way,
- $\overline{0} = \emptyset$,
- $\overline{1} = \{(\vec{0}, \emptyset)\}$, where $\vec{0}$ is the vector associating 0 to every label a,
- $\{(\vec{u}_1, \Omega_1), \ldots, (\vec{u}_n, \Omega_n)\}^\star = \{(\vec{0}, \{\vec{u}_1, \ldots, \vec{u}_n\} \cup \bigcup_{i=1}^n \Omega_i)\}$.

The top element $\top$ corresponds to the set of all vectors. It is represented by $\top = \{(\vec{0}, \{\vec{u}_a \mid a \in Lab\})\}$. The join operation corresponds to set intersection of semilinears. We omit here its definition in terms of the representations above since this is not essential for the aim of the paper (it is well-known that semilinear sets are closed under intersection).

4. COMPUTING ABSTRACT PATH LANGUAGES

We define structures for representing $\alpha(L(C', C))$ and show how their construction can be reduced to a predecessor/successor computation.

4.1 K-predecessors and K-successors

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

Let us fix from now on a pushdown automaton

$$\mathcal{P} = (P, Act, \Gamma, c_0, \Delta)$$

and a Kleene algebra $(K, \oplus, \odot, \star, \overline{0}, \overline{1})$ corresponding to some abstraction. The set of path-expressions Π_K is the smallest subset of K containing $\overline{1}$ and all its basic elements, and closed under $\odot$. 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 action a, then $(c,\pi) \rightarrow_K (c', v_a \odot \pi)$ and $(c, v_a \odot \pi) \leftarrow_K (c', \pi)$ for every $\pi \in \Pi_K$; we say that $(c', v_a \odot \pi)$ is an immediate K-successor of (c,π) and $(c, v_a \odot \pi)$ an immediate K-predecessor of (c', v_a). The *forward (backward) reachability relation over* K $\Rightarrow_K$ ($\Leftarrow_K$) is the reflexive and transitive closure of $\rightarrow_K$ ($\leftarrow_K$).

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-successor (c',π) of a configuration c memorizes in π a sequence of actions that lead from c to c' (from c' to c for K-predecessors). In other words, we have

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

4.2 K-multi-automata

The algorithmic approach we propose is based on constructing automata representations of pre^*_K and $post^*_K$-images of regular sets of configurations.

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

DEFINITION 4.1. *A K-multi-automaton for the PDS $\mathcal{P}$ (K-MA for short) 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 $(\langle p, w\rangle, \pi)$ if $p \xrightarrow{(w, \pi \oplus e)} q$ for some $q \in F$ and some $e \in K$.

Multi-automata can be seen as a special case of K-multi-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-multi-automata. Given a MA $\mathcal{A}$, we denote by $Conf(\mathcal{A})$ the set of configurations recognized by $\mathcal{A}$. Throughout the paper, we will use the symbols p, p', p'', p_i, etc. to denote initial states of (K-)MAs. Arbitrary states will be denoted by q, q', q'', q_i, etc.

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

Let C and C' be two regular sets of configurations represented by means of multi-automata, and consider the problem of computing $\alpha(L(C', C))$. Assume that we are able to compute a K-multi-automaton $\mathcal{A}_{pre^*_K}$ which recognizes the set $pre^*_K(C)$. Then, we can construct a K-multi-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 multi-automaton recognizing C'. [2]

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 element of K defined by:

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

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.

Therefore, we have reduced our problem of computing $\alpha(L(C', C))$ to (*i*) constructing K-multi-automata recognizing pre^*_K or $post^*_K$-images, and then (*ii*) constructing elements in K representing the K-languages recognized by finite K-automata.

The problem (ii) can be solved using standard techniques for solving linear equations in closed semi-rings (e.g., using Floyd-Warshall algorithm, or Gauss elimination). Indeed, this problem is similar to the problem of building regular expressions associated with finite-state automata.

Hence, the main problem to solve is how to compute representations for $post^*_K$ 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. A similar procedure which computes $post^*_K$-images can be found in the full version of the paper.

5. COMPUTING PRE^*_K IMAGES

Given a regular set of configurations C recognized by a MA $\mathcal{A}$, we construct a K-MA $\mathcal{A}_{pre^*_K}$ that recognizes $pre^*_K(C)$. We assume w.l.o.g. that $\mathcal{A}$ has no transition leading to an initial state.

[2] 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 $\mathcal{A}^{C'}_{pre^*_K}$.

5.1 A generic procedure

We proceed in two steps. First, we construct a MA $\mathcal{A}_{pre^*}$ 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 $\langle p,\gamma\rangle \xhookrightarrow{a} \langle p',w\rangle$ and $p' \xrightarrow{w} q$ in the current automaton, add a transition (p,γ,q).

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 $\langle p,\gamma\rangle \xhookrightarrow{a} \langle p',\gamma'\rangle$, 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 $\langle p,\gamma\rangle \xhookrightarrow{a} \langle p',\epsilon\rangle$,
$$v_a \leq l\big((p,\gamma,p')\big)$$

(α_4) for every rule $\langle p,\gamma\rangle \xhookrightarrow{a} \langle p',\gamma_1\gamma_2\rangle$, and every $q \in Q$,
$$\bigoplus_{q'\in Q} \Big(v_a \odot l\big((p',\gamma_1,q')\big) \odot l\big((q',\gamma_2,q)\big)\Big) \leq l\big((p,\gamma,q)\big)$$

Before showing how to compute these tags, let us briefly explain the intuition behind the rules.

- Rules (α_1) express 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 $r = (p,\gamma) \xhookrightarrow{a} (p',u)$, 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 $(\langle p,\gamma w\rangle, v_a \odot \pi)$.

We have the following theorem, proved in the full paper.

THEOREM 5.1. *Given a PDS $\mathcal{P}$ and a regular set of configurations recognized by a MA $\mathcal{A}$, the K-MA $\mathcal{A}_{pre^*_K}$ described above recognizes precisely $pre^*_K\big(Conf(\mathcal{A})\big)$.*

It remains to show how to compute the tagging $l(t)$ for each transition t. For that 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, by (α_1)–(α_4), $l(t_1), \ldots, l(t_m)$ is the smallest solution of a system of inequalities of the form

$$f_i(x_1,\ldots,x_m) \leq x_i, \quad 1 \leq i \leq m \tag{2}$$

where the f_i's are monomials in $K[x_1,\ldots,x_m]$. (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 $e_1 \oplus e_2 \leq l\big((p,\gamma,q)\big)$.)

Solving these inequations depends on the kind of abstraction we are considering:

5.1.1 Finite-chain abstraction

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 (2) is the least pre-fixpoint of F. We can 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.

5.1.2 Commutative abstraction

In this case $(K,\oplus,\odot,\star,\bar{0},\bar{1})$ is a commutative Kleene algebra. The system of equations (2) can be solved using Gauss elimination, plus the elegant formula of [9] for solving an equation in one single variable. This is the most adequate procedure for an implementation. However, [9] also provides a fixpoint algorithm to directly solve (2). Since this is more adequate for a complexity estimate, we recall it briefly here.

We need some preliminaries: Let $\mathbf{x} = (x_1,\ldots,x_n)$ be a vector of indeterminates, let $\mathbf{a} = (a_1,\ldots,a_n)$ be a vector of elements of K, and let $\mathbf{f} = f_1,\ldots,f_l$ 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 $a \in K$.
- $\frac{\partial}{\partial x_i}(f \oplus g) = \frac{\partial f}{\partial x_i} \oplus \frac{\partial g}{\partial x_i}$
- $\frac{\partial}{\partial x_i}(f \odot g) = (f \odot \frac{\partial g}{\partial x_i}) \oplus (\frac{\partial f}{\partial x_i} \odot g)$
- $\frac{\partial}{\partial x_i}(f^\star) = f^\star \odot \frac{\partial f}{\partial x_i}$

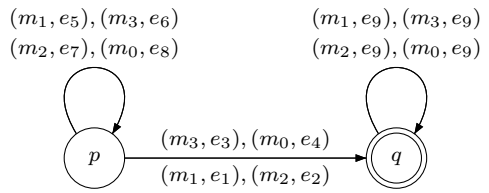

Figure 5: The K-MA representing $Pre^*_K(\langle p, m_3\Gamma^*\rangle)$

Then, the least solution of (2) is the fixpoint of the chain $\mathbf{a_0} \leq \mathbf{a_1} \leq \mathbf{a_2} \ldots$[3] 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 transitive closure of this matrix using e.g. Floyd-Warshall's algorithm. Finally, we compute the product of the resulting matrix and the vector $\mathbf{a_k}$.

5.2 Example

Let us consider as example the program presented on the right of Figure 1. This program is represented by the push-down system $\mathcal{P} = (\{p\}, \{a, b, \tau\}, \{m_0, m_1, m_2, m_3\}, \langle p, m_0\rangle, \Delta)$, where Δ is the set of rules given at the end of section 2.

Let $(K, \oplus, \odot, \star, \bar{0}, \bar{1})$ be a Kleene algebra associated to $\mathcal{P}$. We compute $pre^*_K(\langle p, m_3\Gamma^*\rangle)$. We start with the MA $\mathcal{A} = (\{m_0, m_1, m_2, m_3\}, \{p, q\}, \{(p, m_3, q), (q, m_0, q), \ldots, (q, m_3, q)\}, \{p\}, \{q\})$ recognizing $\langle p, m_3\Gamma^*\rangle$.

Our algorithm yields the K-MA depicted in Figure 5, where the labels $e_1, \ldots, e_9$ are the least solution of the following inequalities:

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

[3]That this is in fact a chain follows easily from the axioms of a Kleene algebra.

Let us explain briefly 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 $(\langle p, m_0m_3\rangle, \pi)$ 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_0m_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 give the solution of these inequalities in the case where $(K, \oplus, \odot, \star, \bar{0}, \bar{1})$ is a commutative Kleene algebra. We apply the algorithm provided by [9], 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

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

is obtained after one iteration.

It follows that the configuration $\langle p, m_0\rangle$ is tagged with e_4, meaning that $L(\langle p, m_0\rangle, \langle p, m_3\Gamma^*\rangle)$ 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.

5.3 Complexity of the procedure

5.3.1 The finite-chain case

Let n_F be the maximal number of $\oplus/\odot$-operations in the monomials f_i. ¿From the definition of the f_i's we have that $n_F = O(|\Delta| \cdot |Q|)$. Therefore, each iteration step has a cost $O(m \cdot |\Delta| \cdot |Q|)$, where m is the number of transitions of $\mathcal{A}_{pre^*}$. It is shown in [6] that $\mathcal{A}_{pre^*}$ has $O(|Q||\Delta| + |\delta|)$ transitions. In typical applications the automaton $\mathcal{A}$ has constant size (two or three states for pure dataflow analysis). So we can consider that m is in $O(|\Delta|)$, and therefore that each iteration step computes $O(|\Delta|^2)$ operations. The total cost of the algorithm depends on the number of iterations which is known to be finite but not necessarily bounded for all inputs. In the case where this number is bounded, e.g., when the abstract domain is finite, let ℓ be the length of the longest chain in K. In this case, we can have at most $m\ell$ iterations, since we are computing a vector of m elements. Therefore, the running time of the iterative fixpoint evaluation algorithm is in $O(\ell \cdot |\Delta|^3 \cdot c)$, where c is the cost of an operation.

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

First occurrence ordering and label bitvectors: In this case the order $\leq$ is simply set inclusion. So the longest chain in K has $2^{|Lab|}$ elements. In this case, one operation takes $O(2^{|Lab|})$ time. Therefore, the running time of the iterative computation is $2^{O(|Lab|)} \cdot |\Delta|^3$.

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 $O(|Lab|^2 \cdot |\Delta|^3)$ or $O(|Lab| \cdot |\Delta|^3)$time.

5.3.2 The commutative case

The complexity is dominated by the fixpoint algorithm of [9]. Since the manipulated matrices have dimension $m \times m$, each iteration of this algorithm requires to compute $O(m^3)$ $\oplus/\odot/\star$-operations. Recall that m is the number of transitions of $\mathcal{A}_{pre^*}$. As said above, we can consider that m is in $O(|\Delta|)$, and hence, the running time is in $O(t \cdot |\Delta|^3 \cdot c)$, where t is the number of iterations, and c is the cost of an operation.

It is proved in [9] that for an *arbitrary* Kleene algebra the fixpoint of the chain is reached after at most $O(3^m)$ iterations. Since each iteration requires to compute a transitive 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.

However, for particular algebras the running time can be much lower. Let us compute it for our three commutative abstractions.

Label bitvectors: If the maximal $\leq$-chain in the abstract lattice has length ℓ, then the number of iterations of the fixpoint algorithm is obviously bounded by $m\ell$. In this case the longest chain in K has $2^{|Lab|}$ elements. Since the cost of an operation takes $O(2^{|Lab|})$ time, we get $2^{O(|Lab|)} \cdot |\Delta|^4$ time.

Forbidden and required sets: In this case the longest possible chain in K has at most length $|Lab|$. So the fixpoint algorithm terminates after at most $m|Lab|$ iterations. Since the cost of an operation is constant or $O(|Lab|)$ ($\star$ is the identity), the algorithm runs in $O(|Lab|^2 \cdot |\Delta|^4)$ or $O(|Lab| \cdot |\Delta|^4)$time.

Parikh images: In this case the complexity is the same as for arbitrary algebras.

As a result of this analysis, we conclude that the forbidden-required analysis has acceptable cost, the first occurrence ordering and label bitvectors analysis have acceptable cost if there are few synchronization statements, and the Parikh-image analysis is likely to have too high cost in most cases. 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 iterative fixpoint evaluation algorithm, or the algorithm of [9]. The analysis above shows that the first algorithm has a worst case better complexity for these abstractions. However, it is worth mentioning that the number of iterations in the algorithm of [9] is always bounded whereas this number depends on the considered domain in the case of the fixpoint evaluation algorithm. In fact the algorithm of [9] performs a kind of fixpoint acceleration. Indeed, this algorithm introduces $\star$-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 longer than the 3^m bound given in [9].

6. 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 [13], 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, and for which emptyness is decidable. We have shown how to compute a context-free algebra for the context-free language of program paths between two arbitrary regular sets of configurations. Using algebraic results we have provided two generic algorithms for the com-

putation 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 [15], 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 [14] is a good 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. In this case, 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 [11], 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 [12] 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 explicitely study concurrency analysis in the presence of procedures 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.

This 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 alphabet 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 analyised in recent papers [1, 16]. 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, 10], and we plan to investigate this in the future.

Acknowledgements

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

7. 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*, pages 36–48. 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, pages 62–75. 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*, pages 1–11. ACM Press, 2000.

[9] M. Hopkins and D. Kozen. Parikh's Theorem in Commutative Kleene Algebra. In *Proc. IEEE Conf. Logic in Computer Science (LICS'99)*. IEEE, 1999.

[10] D. Lugiez and P. Schnoebelen. The Regular Viewpoint on PA-processes. In *TCS*, volume 274(1-2), 2002.

[11] N. Mercouroff. An algorithm for analyzing communicating processes. In *Mathematical Foundations of Programming Semantics*, volume 598 of *LNCS*, pages 312–25, 1991.

[12] 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*, pages 24–34. ACM Press, 1998.

[13] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22:416–430, 2000.

[14] 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.

[15] R. Taylor. A general-purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, 26:362–376, 1983.

[16] E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. *ACM SIGPLAN Notices*, 36(3):27–40, 2001.