Accelerated Data-Flow Analysis

Jérôme Leroux, Grégoire Sutre

Accelerated Data-flow Analysis

Jérôme Leroux and Grégoire Sutre

LaBRI, Université de Bordeaux, CNRS
Domaine Universitaire, 351, cours de la Libération, 33405 Talence, France
{leroux, sutre}@labri.fr

Abstract. Acceleration in symbolic verification consists in computing the exact effect of some control-flow loops in order to speed up the iterative fix-point computation of reachable states. Even if no termination guarantee is provided in theory, successful results were obtained in practice by different tools implementing this framework. In this paper, the acceleration framework is extended to data-flow analysis. Compared to a classical widening/narrowing-based abstract interpretation, the loss of precision is controlled here by the choice of the abstract domain and does not depend on the way the abstract value is computed. Our approach is geared towards precision, but we don't loose efficiency on the way. Indeed, we provide a cubic-time acceleration-based algorithm for solving interval constraints with full multiplication.

1 Introduction

Model-checking safety properties on a given system usually reduces to the computation of a precise enough invariant of the system. In traditional symbolic verification, the set of all reachable (concrete) configurations is computed iteratively from the initial states by a standard fix-point computation. This reachability set is the most precise invariant, but quite often (in particular for software systems) a much coarser invariant is sufficient to prove correctness of the system. Data-flow analysis, and in particular abstract interpretation [CC77], provides a powerful framework to develop analysis for computing such approximate invariants.

A data-flow analysis of a program basically consists in the choice of a (potentially infinite) complete lattice of data properties for program variables together with transfer functions for program instructions. The merge over all path (MOP) solution, which provides the most precise abstract invariant, is in general over-approximated by the minimum fix-point (MFP) solution, which is computable by Kleene fix-point iteration. However the computation may diverge and *widening/narrowing operators* are often used in order to enforce convergence at the expense of precision [CC77, CC92]. While often providing very good results, the solution computed with widenings and narrowings may not be the MFP solution. This may lead to abstract invariants that are too coarse to prove safety properties on the system under check.

Techniques to help convergence of Kleene fix-point iterations have also been investigated in symbolic verification of infinite-state systems. In these works, the

objective is to compute the (potentially infinite) reachability set for automata with variables ranging over unbounded data, such as counters, clocks, stacks or queues. So-called *acceleration* techniques (or *meta-transitions*) have been developped [BW94, BGWW97, CJ98, FIS03, FL02] to speed up the iterative computation of the reachability set. Basically, acceleration consists in computing in one step the effect of iterating a given loop (of the control flow graph). Accelerated symbolic model checkers such as LASH [Las], TREX [ABS01], and FAST [BFLP03] successfully implement this approach.

Our contribution. In this paper, we extend acceleration techniques to data-flow analysis and we apply these ideas to interval analysis. Acceleration techniques for (concrete) reachability set computations may be equivalently formalized "semantically" in terms of control-flow path languages [LS05] or "syntactically" in terms of control-flow graph unfoldings [BFLS05]. We extend these concepts to the MFP solution in a generic data-flow analysis framework, and we establish several links between the resulting notions. It turns out that, for data-flow analysis, the resulting "syntactic" notion, based on graph *flattenings*, is more general that the resulting "semantic" notion, based on restricted regular expressions. We then propose a generic flattening-based semi-algorithm for computing the MFP solution. This semi-algorithm may be viewed as a generic template for applying acceleration-based techniques to constraint solving.

We then show how to instantiate the generic flattening-based semi-algorithm in order to obtain an efficient constraint solver[1] for integers, for a rather large class of constraints using addition, (monotonic) multiplication, factorial, or any other *bounded-increasing* function. The intuition behind our algorithm is the following: we propagate constraints in a breadth-first manner as long as the least solution is not obtained, and variables involved in a "useful" propagation are stored in a graph-like structure. As soon as a cycle appears in this graph, we compute the least solution of the set of constraints corresponding to this cycle. It turns out that this acceleration-based algorithm always terminates in cubic-time.

As the main result of the paper, we then show how to compute in cubic-time the least solution for interval constraints with full addition and multiplication, and intersection with a constant. The proof uses a least-solution preserving translation from interval constraints to the class of integer constraints introduced previously.

Related work. In [Kar76], Karr presented a polynomial-time algorithm that computes the set of all affine relations that hold in a given control location of a (numerical) program. Recently, the complexity of this algorithm was revisited in [MOS04] and a fine upper-bound was presented. For interval constraints with affine transfer functions, the exact least solution may be computed in cubic-time [SW04]. Strategy iteration was proposed in [CGG$^+$05] to speed up Kleene fix-point iteration with better precision than widenings and narrowings, and

[1] By solver, we mean an algorithm computing the least solution of constraint systems.

this approach has been developped in [TG07] for interval constraint solving with full addition, multiplication and intersection. Strategy iteration may be viewed as an instance of our generic flattening-based semi-algorithm. The class of interval constraints that we consider in this paper contains the one in [SW04] (which does not include interval multiplication) but it is more restrictive than the one in [TG07]. We are able to maintain the same cubic-time complexity as in [SW04], and it is still an open problem whether interval constraint solving can be performed in polynomial-time for the larger class considered in [TG07].

Outline. The paper is organized as follows. Section 2 presents our acceleration-based approach to data-flow analysis. We then focus on interval constraint-based data-flow analysis. We present in section 3 a cubic-time algorithm for solving a large class of constraints over the integers, and we show in section 4 how to translate interval constraints (with multiplication) into the previous class of integer constraints, hence providing a cubic-time algorithm for interval constraints. Section 5 presents some ideas for future work. Please note that most proofs are only sketched in the paper, but detailed proofs are given in appendix. This paper is the long version of our SAS 2007 paper.

2 Acceleration in Data Flow Analysis

This section is devoted to the notion of acceleration in the context of data-flow analysis. Acceleration techniques for (concrete) reachability set computations [BW94, BGWW97, CJ98, FIS03, FL02, LS05, BFLS05] may be equivalently formulated in terms of control-flow path languages or control-flow graph unfoldings. We shall observe that this equivalence does not hold anymore when these notions are lifted to data-flow analysis. All results in this section can easily be derived from the definitions, and they are thus presented without proofs.

2.1 Lattices, words and graphs

We respectively denote by $\mathbb{N}$ and $\mathbb{Z}$ the usual sets of nonnegative integers and integers. For any set S, we write $\mathbb{P}(S)$ for the set of subsets of S. The *identity* function over S is written $\mathbb{1}_S$, and shortly $\mathbb{1}$ when the set S is clear from the context.

Recall that a *complete lattice* is any partially ordered set $(A, \sqsubseteq)$ such that every subset $X \subseteq A$ has a *least upper bound* $\bigsqcup X$ and a *greatest lower bound* $\bigsqcap X$. The *supremum* $\bigsqcup A$ and the *infimum* $\bigsqcap A$ are respectively denoted by $\top$ and $\bot$. A function $f \in A \to A$ is *monotonic* if $f(x) \sqsubseteq f(y)$ for all $x \sqsubseteq y$ in A. Recall that from Knaster-Tarski's Fix-point Theorem, any monotonic function $f \in A \to A$ has a *least fix-point* given by $\bigsqcap\{a \in A \mid f(a) \sqsubseteq a\}$. For any monotonic function $f \in A \to A$, we denote by f^* the monotonic function in $A \to A$ defined by $f^*(x) = \bigsqcap\{a \in A \mid (x \sqcup f(a)) \sqsubseteq a\}$, in other words $f^*(x)$ is the least post-fix-point of f greater than x.

For any complete lattice $(A, \sqsubseteq)$ and any set S, we also denote by $\sqsubseteq$ the partial order on $S \to A$ defined as the point-wise extension of $\sqsubseteq$, i.e. $f \sqsubseteq g$ iff $f(x) \sqsubseteq g(x)$ for all $x \in S$. The partially ordered set $(S \to A, \sqsubseteq)$ is also a complete lattice, with lub $\bigsqcup$ and glb $\bigsqcap$ satisfying $(\bigsqcup F)(s) = \bigsqcup\{f(s) \mid f \in F\}$ and $(\bigsqcap F)(s) = \bigsqcap\{f(s) \mid f \in F\}$ for any subset $F \subseteq S \to A$. Given any integer $n \geq 0$, we denote by A^n the set of n-tuples over A. We identify A^n with the set $\{1, \ldots, n\} \to A$, and therefore A^n equipped with the point-wise extension of $\sqsubseteq$ also forms a complete lattice.

Let Σ be an *alphabet* (a finite set of *letters*). We write Σ^* for the set of all (finite) *words* $l_0 \cdots l_n$ over Σ, and ε denotes the empty word. Given any two words x and y, we denote by $x \cdot y$ (shortly written xy) their *concatenation*. A subset of Σ^* is called a *language*.

A (directed) *graph* is any pair $G = (V, \to)$ where V is a set of *vertices* and $\to$ is a binary relation over V. A pair (v, v') in $\to$ is called an *edge*. A (finite) *path* in G is any (non-empty) sequence $v_0, \ldots, v_k$ of vertices, also written $v_0 \to v_1 \cdots v_{k-1} \to v_k$, such that $v_{i-1} \to v_i$ for all $1 \leq i \leq k$. The nonnegative integer k is called the *length* of the path, and the vertices v_0 and v_k are respectively called the *source* and *target* of the path. A *cycle* on a vertex v is any path of non-zero length with source and target equal to v. A cycle with no repeated vertices other than the source and the target is called *elementary*. We write $\overset{*}{\to}$ for the reflexive-transitive closure of $\to$. A *strongly connected component* (shortly *SCC*) in G is any equivalence class for the equivalence relation $\overset{*}{\leftrightarrow}$ on V defined by: $v \overset{*}{\leftrightarrow} v'$ if $v \overset{*}{\to} v'$ and $v' \overset{*}{\to} v$. We say that an SCC is *cyclic* when it contains a unique elementary cycle up to cyclic permutation.

2.2 Programs and data-flow solutions

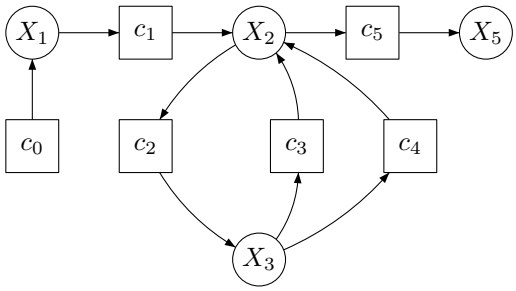
For the rest of this section, we consider a complete lattice $(A, \sqsubseteq)$. In our setting, a program will represent an instance (for some concrete program) of a data-flow analysis framework over $(A, \sqsubseteq)$. To simplify the presentation, we will consider programs given as unstructured collections of commands (this is not restrictive as control-flow may be expressed through variables).

Formally, assume a finite set $\mathcal{X}$ of *variables*. A *command* on $\mathcal{X}$ is any tuple $\langle X_1, \ldots, X_n; f; X\rangle$, also written $X := f(X_1, \ldots, X_n)$, where $n \in \mathbb{N}$ is an *arity*, $X_1, \ldots, X_n \in \mathcal{X}$ are pairwise disjoint *input variables*, $f \in A^n \to A$ is a monotonic *transfer function*, and $X \in \mathcal{X}$ is an *output variable*. Intuitively, a command $X := f(X_1, \ldots, X_n)$ assigns variable X to $f(X_1, \ldots, X_n)$ and lets all other variables untouched. A *valuation* on $\mathcal{X}$ is any function ρ in $\mathcal{X} \to A$. The *data-flow semantics* $[\![c]\!]$ of any command $c = \langle X_1, \ldots, X_n; f; X\rangle$ on $\mathcal{X}$ is the monotonic function in $(\mathcal{X} \to A) \to (\mathcal{X} \to A)$ defined by $[\![c]\!](\rho)(X) = f(\rho(X_1), \ldots, \rho(X_n))$ and $[\![c]\!](\rho)(Y) = \rho(Y)$ for all $Y \neq X$.

A *program* over $(A, \sqsubseteq)$ is any pair $\mathcal{P} = (\mathcal{X}, C)$ where $\mathcal{X}$ is a finite set of *variables* and C is a finite set of commands on $\mathcal{X}$.

Example 2.1. Consider the C-style source code given on the left-hand side below, that we want to analyse with the complete lattice $(\mathcal{I}, \sqsubseteq)$ of intervals of $\mathbb{Z}$. The corresponding program $\mathcal{E}$ is depicted graphically on the right-hand side below.

```
1  x = 1;
2  while (x ≤ 100) {
3    if (x ≥ 50) x = x−3;
4    else x = x+2;
5  }
```



Formally, the set of variables of $\mathcal{E}$ is $\{X_1, X_2, X_3, X_5\}$, representing the value of the variable x at program points 1, 2, 3 and 5. The set of commands of $\mathcal{E}$ is $\{c_0, c_1, c_2, c_3, c_4, c_5\}$, with:

$$\begin{array}{ll} c_0 : \ X_1 := \top & c_3 : \ X_2 := (X_3 \sqcap [50, +\infty]) - \{3\} \\ c_1 : \ X_2 := (\{0\} \,.\, X_1) + \{1\} & c_4 : \ X_2 := (X_3 \sqcap]-\infty, 49]) + \{2\} \\ c_2 : \ X_3 := X_2 \sqcap]-\infty, 100] & c_5 : \ X_5 := X_2 \sqcap [101, +\infty[\end{array}$$

We will use language-theoretic terminology and notations for traces in a program. A *trace* in $\mathcal{P}$ is any word $c_1 \cdots c_k$ over C. The empty word ε denotes the empty trace and C^* denotes the set of all traces in $\mathcal{P}$. The data-flow semantics is extended to traces in the obvious way: $[\![\varepsilon]\!] = \mathbb{1}$ and $[\![c \cdot \sigma]\!] = [\![\sigma]\!] \circ [\![c]\!]$. Observe that $[\![\sigma \cdot \sigma']\!] = [\![\sigma']\!] \circ [\![\sigma]\!]$ for every $\sigma, \sigma' \in C^*$. We also extend the data-flow semantics to sets of traces by $[\![L]\!] = \bigsqcup_{\sigma \in L} [\![\sigma]\!]$ for every $L \subseteq C^*$. Observe that $[\![L]\!]$ is a monotonic function in $(\mathcal{X} \to A) \to (\mathcal{X} \to A)$, and moreover $[\![L_1 \cup L_2]\!] = [\![L_1]\!] \sqcup [\![L_2]\!]$ for every $L_1, L_2 \subseteq C^*$.

Given a program $\mathcal{P} = (\mathcal{X}, C)$ over $(A, \sqsubseteq)$, the *minimum fix-point solution* (*MFP-solution)* of $\mathcal{P}$, written $\Lambda_{\mathcal{P}}$, is the valuation defined as follows:

$$\Lambda_{\mathcal{P}} \ = \ \bigsqcap \ \{\rho \in \mathcal{X} \to A \mid [\![c]\!](\rho) \sqsubseteq \rho \text{ for all } c \in C\}$$

Example 2.2. The MFP-solution of the program $\mathcal{E}$ from Example 2.1 is the valuation:

$$\Lambda_{\mathcal{E}} \ = \ \{X_1 \mapsto \top, \ X_2 \mapsto [1, 51], \ X_3 \mapsto [1, 51], \ X_5 \mapsto \bot\}$$

Recall that we denote by $[\![C]\!]^*(\rho)$ the least post-fix-point of $[\![C]\!]$ greater than ρ. Therefore it follows from the definitions that $\Lambda_{\mathcal{P}} = [\![C]\!]^*(\bot)$. In our framework, the *merge over all paths solution* (*MOP-solution)* may be defined as the valuation $[\![C^*]\!](\bot)$, and the following proposition recalls well-known links between the MOP-solution, the MFP-solution and the ascending Kleene chain.

Proposition 2.3. *For any program $\mathcal{P} = (\mathcal{X}, C)$ over a complete lattice $(A, \sqsubseteq)$, we have:*

$$[\![C^*]\!](\bot) \quad \sqsubseteq \quad \bigsqcup_{k \in \mathbb{N}} [\![C]\!]^k(\bot) \quad \sqsubseteq \quad [\![C]\!]^*(\bot) \quad = \quad \Lambda_{\mathcal{P}}$$

2.3 Accelerability and flattening

We now extend notions from accelerated symbolic verification to this data-flow analysis framework. Acceleration in symbolic verification was first introduced semantically, in the form of *meta-transitions* [BW94, BGWW97], which basically simulate the effect of taking a given control-flow loop arbitrarily many times. This leads us to the following proposition and definition.

Proposition 2.4. *Let $\mathcal{P} = (\mathcal{X}, C)$ denote a program over $(A, \sqsubseteq)$. For any languages $L_1, \ldots, L_k \subseteq C^*$, we have $(\llbracket L_k \rrbracket^* \circ \cdots \circ \llbracket L_1 \rrbracket^*)(\bot) \sqsubseteq \Lambda_{\mathcal{P}}$.*

Definition 2.5. *A program $\mathcal{P} = (\mathcal{X}, C)$ over a complete lattice $(A, \sqsubseteq)$ is called* MFP-accelerable *if $\Lambda_{\mathcal{P}} = (\llbracket \sigma_k \rrbracket^* \circ \cdots \circ \llbracket \sigma_1 \rrbracket^*)(\bot)$ for some words $\sigma_1, \ldots, \sigma_k \in C^*$.*

The following proposition shows that any program $\mathcal{P}$ for which the ascending Kleene chain stabilizes after finitely many steps is MFP-accelerable.

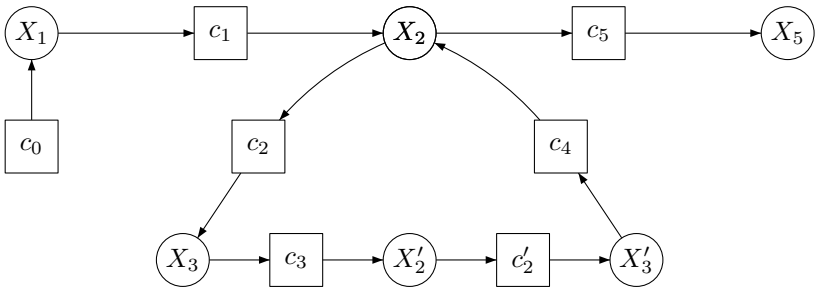
Proposition 2.6. *Let $\mathcal{P} = (\mathcal{X}, C)$ denote a program over $(A, \sqsubseteq)$. If we have $\llbracket C \rrbracket^k(\bot) = \Lambda_{\mathcal{P}}$ for some $k \in \mathbb{N}$, then $\mathcal{P}$ is MFP-accelerable.*

Acceleration in symbolic verification was later expressed syntactically, in terms of flat graph unfoldings. When lifted to data-flow analysis, this leads to a more general concept than accelerability, and we will show that these two notions coincide for "concrete" programs (as in symbolic verification). We say that a program $\mathcal{P}$ is *single-input* if the arity of every command in $\mathcal{P}$ is at most 1.

Given a program $\mathcal{P} = (\mathcal{X}, C)$ over $(A, \sqsubseteq)$, an *unfolding* of $\mathcal{P}$ is any pair $(\mathcal{P}', \kappa)$ where $\mathcal{P}' = (\mathcal{X}', C')$ is a program and $\kappa \in \mathcal{X}' \to \mathcal{X}$ is a variable *renaming*, and such that $\langle \kappa(X'_1), \ldots, \kappa(X'_n); f; \kappa(X') \rangle$ is a command in C for every command $\langle X'_1, \ldots, X'_n; f; X' \rangle$ in C'. The renaming κ induces a Galois surjection $(\mathcal{X}' \to A, \sqsubseteq) \underset{\overrightarrow{\kappa}}{\overset{\overleftarrow{\kappa}}{\leftrightarrows}} (\mathcal{X} \to A, \sqsubseteq)$ where $\overleftarrow{\kappa}$ and $\overrightarrow{\kappa}$ are defined as expected by $\overleftarrow{\kappa}(\rho) = \rho \circ \kappa$ and $\overrightarrow{\kappa}(\rho')(X) = \bigsqcup_{\kappa(X') = X} \rho'(X')$.

We associate a bipartite graph to any program in a natural way: vertices are either variables or commands, and edges denote input and output variables of commands. Formally, given a program $\mathcal{P} = (\mathcal{X}, C)$, the *program graph* of $\mathcal{P}$ is the labeled graph $G_{\mathcal{P}}$ where $\mathcal{X} \cup C$ is the set of vertices and with edges (c, X) and (X_i, c) for every command $c = \langle X_1, \ldots, X_n; f; X \rangle$ in C and $1 \leq i \leq n$. We say that $\mathcal{P}$ is *flat* if there is no SCC in $G_{\mathcal{P}}$ containing two distinct commands with the same output variable. A *flattening* of $\mathcal{P}$ is any unfolding $(\mathcal{P}', \kappa)$ of $\mathcal{P}$ such that $\mathcal{P}'$ is flat.

Example 2.7. A flattening of the program $\mathcal{E}$ from Example 2.1 is given below. Intuitively, this flattening represents a possible unrolling of the while-loop where the two branches of the inner conditional alternate.



Lemma 2.8. *Let $\mathcal{P} = (\mathcal{X}, C)$ denote a program over $(A, \sqsubseteq)$. For any unfolding $(\mathcal{P}', \kappa)$ of $\mathcal{P}$, with $\mathcal{P}' = (\mathcal{X}', C')$, we have $\overrightarrow{\kappa} \circ [\![C']\!]^* \circ \overleftarrow{\kappa} \sqsubseteq [\![C]\!]^*$.*

Proposition 2.9. *Let $\mathcal{P} = (\mathcal{X}, C)$ denote a program over $(A, \sqsubseteq)$. For any unfolding $(\mathcal{P}', \kappa)$ of $\mathcal{P}$, we have $\overrightarrow{\kappa}(\Lambda_{\mathcal{P}'}) \sqsubseteq \Lambda_{\mathcal{P}}$.*

Definition 2.10. *A program $\mathcal{P} = (\mathcal{X}, C)$ over a complete lattice $(A, \sqsubseteq)$ is called* MFP-flattable *if $\Lambda_{\mathcal{P}} = \overrightarrow{\kappa}(\Lambda_{\mathcal{P}'})$ for some flattening $(\mathcal{P}', \kappa)$ of $\mathcal{P}$.*

Observe that any flat program is trivially MFP-flattable. The following proposition establishes links between accelerability and flattability. As a corollary to the proposition, we obtain that MFP-accelerability and MFP-flattability are equivalent for single-input programs.

Proposition 2.11. *The following relationships hold for programs over $(A, \sqsubseteq)$:*

i) MFP-accelerability implies MFP-flattability.
ii) MFP-flattability implies MFP-accelerability for single-input programs.

Proof (Sketch). To prove *i*), we use the fact that for every words $\sigma_1, \ldots, \sigma_k \in C^*$, there exists a finite-state automaton $\mathcal{A}$ without nested cycles recognizing $\sigma_1^* \cdots \sigma_k^*$. The "product" of any program $\mathcal{P}$ with $\mathcal{A}$ yields a flattening that "simulates" the effect of $\sigma_1^* \cdots \sigma_k^*$ on $\mathcal{P}$. To prove *ii*), we observe that for any flat single-input program $\mathcal{P}$, each non-trivial SCC of $G_{\mathcal{P}}$ is cyclic. We pick a "cyclic" trace (which is unique up to circular permutation) for each SCC, and we arrange these traces to prove that $\mathcal{P}$ is accelerable. Backward preservation of accelerability under unfolding concludes the proof. □

Remark 2.12. For any labeled transition system $\mathcal{S}$ with a set S of states, the forward collecting semantics of $\mathcal{S}$ may naturally be given as a single-input program $\mathcal{P}_{\mathcal{S}}$ over $(\mathbb{P}(S), \subseteq)$. With respect to this translation (from $\mathcal{S}$ to $\mathcal{P}_{\mathcal{S}}$), the notion of flattability developped for accelerated symbolic verification of labeled transition systems coincide with the notions of MFP-accelerability and MFP-flattability defined above.

Recall that our main goal is to compute (exact) MFP-solutions using acceleration-based techniques. According to the previous propositions, flattening-based computation of the MFP-solution seems to be the most promising approach, and we will focus on this approach for the rest of the paper.

2.4 Generic flattening-based constraint solving

It is well known that the MFP-solution of a program may also be expressed as the least solution of a constraint system, and we will use this formulation for the rest of the paper. We will use some new terminology to reflect this new formulation, however notations and definitions will remain the same. A command $\langle X_1, \ldots, X_n; f; X \rangle$ will now be called a *constraint*, and will also be written $X \sqsupseteq f(X_1, \ldots, X_n)$. A program over $(A, \sqsubseteq)$ will now be called a *constraint system* over $(A, \sqsubseteq)$, and the MFP-solution will be called the *least solution*. Among all acceleration-based notions defined previously, we will only consider MFP-flattability for constraint systems, and hence we will shortly write *flattable* instead of MFP-flattable.

Given a constraint system $\mathcal{P} = (\mathcal{X}, C)$ over $(A, \sqsubseteq)$, any valuation $\rho \in \mathcal{X} \to A$ such that $\rho \sqsubseteq [\![C]\!](\rho)$ (resp. $\rho \sqsupseteq [\![C]\!](\rho)$) is called a *pre-solution* (resp. a *post-solution*). A post-solution is also shortly called a *solution*. Observe that the least solution $\Lambda_{\mathcal{P}}$ is the greatest lower bound of all solutions of C.

We now present a generic flattening-based semi-algorithm for constraint solving. Intuitively, this semi-algorithm performs a propagation of constraints starting from the valuation $\bot$, but at each step we extract a flat "subset" of constraints (possibly by duplicating some variables) and we update the current valuation with the least solution of this flat "subset" of constraints.

```
1  Solve(P = (X, C) : a constraint system)
2  ρ ← ⊥
3  while [[C]](ρ) ⋢ ρ
4        construct a flattening (P', κ) of P, where P' = (X', C')
5        ρ' ← ρ ∘ κ
6        ρ'' ← [[C']]*(ρ')                    { κ⃗(ρ'') ⊑ [[C]]*(ρ) from Lemma 2.8 }
7        ρ ← ρ ⊔ κ⃗(ρ'')
8  return ρ
```

The Solve semi-algorithm may be viewed as a generic template for applying acceleration-based techniques to constraint solving. The two main challenges are (1) the construction of a suitable flattening at line 4, and (2) the computation of the least solution for flat constraint systems (line 6). However, assuming that all involved operations are effective, this semi-algorithm is *correct* (i.e. if it terminates then the returned valuation is the least solution of input constraint system), and it is *complete* for flattable constraint systems (i.e. the input constraint system is flattable if and only if there exists choices of flattenings at line 4 such that the while-loop terminates). We will show in the sequel how to instantiate the Solve semi-algorithm in order to obtain an efficient constraint solver for integers and intervals.

3 Integer Constraints

Following [SW04, TG07], we first investigate integer constraint solving in order to derive in the next section an interval solver. This approach is motivated by the encoding of an interval by two integers.

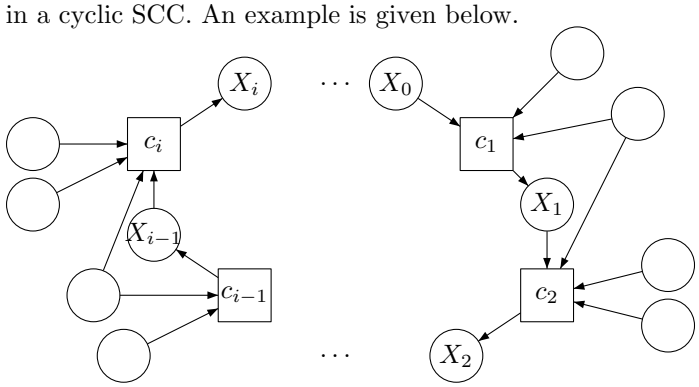
The *complete lattice of integers* $\mathcal{Z} = \mathbb{Z} \cup \{-\infty, +\infty\}$ is equipped with the natural order:

$$-\infty < \cdots < -2 < -1 < 0 < 1 < 2 < \cdots < +\infty$$

Observe that the least upper bound $x \vee y$ and the greatest lower bound $x \wedge y$ respectively correspond to the maximum and the minimum. Addition and multiplication functions are extended from $\mathbb{Z}$ to $\mathcal{Z}$ as in [TG07]:

$$\begin{array}{rcrcrcrcrcl}
x.0 & = & 0.x & = & 0 & & x + (-\infty) = (-\infty) + x & = & -\infty & & \text{for all } x \\
x.(+\infty) & = & (+\infty).x & = & +\infty & & x.(-\infty) & = & (-\infty).x & = & -\infty \quad \text{for all } x > 0 \\
x.(+\infty) & = & (+\infty).x & = & -\infty & & x.(-\infty) & = & (-\infty).x & = & +\infty \quad \text{for all } x < 0 \\
x + (+\infty) & = & (+\infty) + x & = & +\infty & & & & & & \text{for all } x > -\infty
\end{array}$$

A constraint system $\mathcal{P} = (\mathcal{X}, C)$ is said *cyclic* if the set of constraints C is contained in a cyclic SCC. An example is given below.



Observe that a cyclic constraint system is flat. A *cyclic flattening* $(\mathcal{P}', \kappa)$ where $\mathcal{P}' = (\mathcal{X}', C')$ can be naturally *associated* to any cycle $X_0 \to c_1 \to X_1 \cdots \to c_n \to X_n = X_0$ of a constraint system $\mathcal{P}$, by considering the set $\mathcal{X}'$ of variables obtained from $\mathcal{X}$ by adding n new copies $Z_1, \ldots, Z_n$ of $X_1, \ldots, X_n$ with the corresponding renaming κ that extends the identity function over $\mathcal{X}$ by $\kappa(Z_i) = X_i$, and by considering the set of constraints $C' = \{c'_1, \ldots, c'_n\}$ where c'_i is obtained from c_i by renaming the output variable X_i by Z_i and by renaming the input variable X_{i-1} by Z_{i-1} where $Z_0 = Z_n$.

In section 3.1, we introduce an instance of the generic Solve semi-algorithm that solves constraint systems that satisfy a property called *bounded-increasing*. This class of constraint systems is extended in section 3.2 with test constraints allowing a natural translation of *interval* constraint systems to contraint systems in this class.

3.1 Bounded-increasing constraint systems

A monotonic function $f \in \mathcal{Z}^k \to \mathcal{Z}$ is said *bounded-increasing* if for any $x_1 < x_2$ such that $f(\bot) < f(x_1)$ and $f(x_2) < f(\top)$ we have $f(x_1) < f(x_2)$. Intuitively f is increasing over the domain of $x \in \mathcal{Z}^k$ such that $f(x) \notin \{f(\bot), f(\top)\}$.

Example 3.1. The guarded identity $x \mapsto x \wedge b$ where $b \in \mathcal{Z}$, the addition $(x, y) \mapsto x + y$, the two multiplication functions mul_+ and mul_- defined below, the power by two $x \mapsto 2^{x \vee 0}$, the factorial $x \mapsto !(x \vee 1)$ are bounded-increasing. However the minimum and the maximum functions are not bounded-increasing.

$$\mathsf{mul}_+(x,y) = \begin{cases} x.y & \text{if } x, y \geq 0 \\ 0 & \text{otherwise} \end{cases} \qquad \mathsf{mul}_-(x,y) = \begin{cases} -x.y & \text{if } x, y < 0 \\ 0 & \text{otherwise} \end{cases}$$

A *bounded-increasing constraint* is a constraint of the form $X \geq f(X_1, \ldots, X_k)$ where f is a bounded-increasing function. Such a constraint is said *upper-saturated* (resp. *lower-saturated*) by a valuation ρ if $\rho(X) \geq f(\top)$ (resp. $\rho(X) \leq f(\bot)$). Given a constraint system $\mathcal{P} = (\mathcal{X}, C)$ and a bounded-increasing constraint $c \in C$ upper-saturated by a valuation ρ_0, observe that $[\![C]\!]^*(\rho_0) = [\![C']\!]^*(\rho_0)$ where $C' = C \backslash \{c\}$. Intuitively, an upper-saturated constraint for ρ_0 can be safely removed from a constraint system without modifying the least solution greater than ρ_0. The following lemma will be useful to obtain upper-saturated constraints.

Lemma 3.2. *Let $\mathcal{P}$ be a cyclic bounded-increasing constraint system. If ρ_0 is a pre-solution of $\mathcal{P}$ that does not lower-saturate any constraint, then either ρ_0 is a solution or $[\![C]\!]^*(\rho_0)$ upper-saturates a constraint.*

Proof. (Sketch). Let $X_0 \to c_1 \to X_1 \to \cdots \to c_n \to X_n = X_0$ be the unique (up to a cyclic permutation) cycle in the graph associated to $\mathcal{P}$. Consider a pre-solution ρ_0 of $\mathcal{P}$ that is not a solution. Let us denote by $(\rho_i)_{i \geq 0}$ the sequence of valuations defined inductivelly by $\rho_{i+1} = \rho_i \vee [\![C]\!](\rho_i)$. There are two cases:

- either there exists $i \geq 0$ such that ρ_i upper-saturates a constraint c_j. Since $\rho_i \leq [\![C]\!]^*(\rho_0)$, we deduce that $[\![C]\!]^*(\rho_0)$ upper-saturates c_j.
- or $c_1, \ldots, c_n$ are not upper-saturated by any of the ρ_i. As these constraints are bounded-increasing, the sequence $(\rho_i)_{i \geq 0}$ is strictly increasing. Thus $(\bigvee_{i \geq 0} \rho_i)(X_j) = +\infty$ for any $1 \leq j \leq n$. Since $\bigvee_{i \geq 0} \rho_i \leq [\![C]\!]^*(\rho_0)$, we deduce that $[\![C]\!]^*(\rho_0)$ upper-saturates $c_1, \ldots, c_n$.

In both cases, $[\![C]\!]^*(\rho_0)$ upper-saturates at least one constraint. □

```
1  CyclicSolve (P = (X, C) : a cyclic bounded−increasing constraint system,
2               ρ0 : a valuation)
3  let  X0 → c1 → X1 ··· → cn → Xn = X0 be the ''unique'' elementary cycle
4  ρ ← ρ0
5  for i = 1 to n do
```

```
6        ρ ← ρ ∨ [[c_i]](ρ)
7    for i = 1 to n do
8        ρ ← ρ ∨ [[c_i]](ρ)
9    if ρ ≥ [[C]](ρ)
10       return ρ
11   for i = 1 to n do
12       ρ(X_i) ← +∞
13   for i = 1 to n do
14       ρ ← ρ ∧ [[c_i]](ρ)
15   for i = 1 to n do
16       ρ ← ρ ∧ [[c_i]](ρ)
17   return ρ
```

Proposition 3.3. *The algorithm* CyclicSolve *returns* $[\![C]\!]^*(\rho_0)$ *for any cyclic constraint system* $\mathcal{P}$ *and for any valuation* ρ_0*.*

Proof. (Sketch). The first two loops (lines 5–8) propagate the valuation ρ_0 along the cycle two times. If the resulting valuation is not a solution at this point, then it is a pre-solution and no constraint is lower-saturated. From Lemma 3.2, we get that $[\![C]\!]^*(\rho_0)$ upper-saturates some constraint. Observe that the valuation ρ after the third loop (lines 11–12) satisfies $[\![C]\!]^*(\rho_0) \sqsubseteq \rho$. The descending iteration of the last two loops yields (at line 17) $[\![C]\!]^*(\rho_0)$. □

We may now present our cubic time algorithm for solving bounded-increasing constraint systems. The main loop of this algorithm first performs $|C|+1$ rounds of Round Robin iterations and keeps track for each variable of the last constraint that updated its value. This information is stored in a partial function λ from $\mathcal{X}$ to C. The second part of the main loop checks whether there exists a cycle in the subgraph induced by λ, and if so it selects such a cycle and calls the CylicSolve algorithm on it.

```
1    SolveBI(P = (X, C) : a bounded−increasing constraint system,
2                ρ_0 : an initial valuation)
3    ρ ← ρ_0 ∨ [[C]](ρ_0)
4    while [[C]](ρ) ⋢ ρ
5        λ ← ∅                          { λ is a partial function from X to C }
6        repeat |C| + 1 times
7            for each c ∈ C
8                if ρ ⋡ [[c]](ρ)
9                    ρ ← ρ ∨ [[c]](ρ)
10                   λ(X) ← c, where X is the input variable of c
11       if there exists an elementary cycle X_0 → λ(X_1) → X_1 ··· λ(X_n) → X_0
12           construct the corresponding cyclic flattening (P', κ)
13           ρ' ← ρ ∘ κ
14           ρ'' ← CyclicSolve(P', ρ')
```

15 $\rho \leftarrow \rho \vee \overrightarrow{\kappa}(\rho'')$

16 `return` ρ

Note that the SolveBI algorithm is an instance of the Solve semi-algorithm where flattenings are deduced from cycles induced by the partial function λ. The following proposition 3.4 shows that this algorithm terminates.

Proposition 3.4. *The algorithm SolveBI returns the least solution $[\![C]\!]^*(\rho_0)$ of a bounded-increasing constraint system $\mathcal{P}$ greater than a valuation ρ_0. Moreover, the number of times the while loop is executed is bounded by one plus the number of constraints that are upper-saturated for $[\![C]\!]^*(\rho_0)$ but not for ρ_0.*

Proof. (Sketch). Observe that initially $\rho = \rho_0 \vee [\![C]\!](\rho_0)$. Thus, if during the execution of the algorithm $\rho(X)$ is updates by a constraint c then necessary c is not lower-saturated. That means if $\lambda(X)$ is defined then $c = \lambda(X)$ is not lower-saturated.

Let ρ_0 and ρ_1 be the values of ρ respectively before and after the execution of the first two nested loops (line 5-9) and let ρ_2 be the value of ρ after the execution of line 14.

Observe that if there does not exist an elementary cycle satisfying the condition given in line 11, the graph associated to $\mathcal{P}$ restricted to the edges (X, c) if $c = \lambda(X)$ and the edges (X_i, c) if X_i is an input variable of c is acyclic. This graph induces a natural partial order over the constraints c of the form $c = \lambda(X)$. An enumeration $c_1, \ldots, c_m$ of this constraints compatible with the partial order provides the relation $\rho_1 \leq [\![c_1 \ldots c_m]\!](\rho_0)$. Since the loop 6-9 is executed at least $m + 1$ times, we deduce that ρ_1 is a solution of $\mathcal{C}$.

Lemma 3.2 shows that if ρ_1 is not a solution of $\mathcal{P}$ then at least one constraint is upper-saturated for ρ_2 but not for ρ_0. We deduce that the number of times the while loop is executed is bounded by one plus the number of constraints that are upper-saturated for $[\![C]\!]^*(\rho_0)$ but not for ρ_0. □

3.2 Integer constraint systems

A *test function* is a function $\theta_{>b}$ or $\theta_{\geq b}$ with $b \in \mathcal{Z}$ of the following form:

$$\theta_{\geq b}(x, y) = \begin{cases} y & \text{if } x \geq b \\ -\infty & \text{otherwise} \end{cases} \qquad \theta_{>b}(x, y) = \begin{cases} y & \text{if } x > b \\ -\infty & \text{otherwise} \end{cases}$$

A *test constraint* is a constraint of the form $X \geq \theta_{\sim b}(X_1, X_2)$ where $\theta_{\sim b}$ is a test function. Such a constraint c is said *active* for a valuation ρ if $\rho(X_1) \sim b$. Given a valuation ρ such that c is active, observe that $[\![c]\!](\rho)$ and $[\![c']\!](\rho)$ are equal where c' is the bounded-increasing constraint $X \geq X_2$. This constraint c' is called the *active form* of c and denoted by $\mathsf{act}(c)$.

In the sequel, an *integer constraint* either refers to a bounded-increasing constraint or a test-constraint.

```
1   SolveInteger (P = (X, C) : an integer constraint system)
2   ρ ← ⊥
3   C_t ← set of test constraints in C
4   C' ← set of bounded−increasing constraints in C
5   while [[C]](ρ) ⋢ ρ
6         ρ ← SolveBI((X, C'), ρ)
7         for each c ∈ C_t
8               if c is active for ρ
9                     C_t ← C_t\{c}
10                    C' ← C' ∪ {act(c)}
11  return ρ
```

Theorem 3.5. *The algorithm SolveInteger computes the least solution of an integer constraint system $\mathcal{P} = (\mathcal{X}, C)$ by performing $O((|\mathcal{X}| + |C|)^3)$ integer comparisons and image computation by some bounded-increasing functions.*

Proof. Let us denote by n_t be the number of test constraints in C. Observe that if during the execution of the while loop, no test constraints becomes active (line 7-10) then ρ is a solution of $\mathcal{P}$ and the algorithm terminates. Thus this loop is executed at most $1+n_t$ times. Let us denote by $m_1, \ldots, m_k$ the integers such that m_i is equal to the number of times the while loop of SolveBI is executed. Since after the execution there is $m_i - 1$ constraints that becomes upper-saturated, we deduce that $\sum_{i=1}^{k}(m_i - 1) \leq n$ and in particular $\sum_{i=1}^{k} m_i \leq n + k \leq 2.|C|$. Thus the algorithm SolveInteger computes the least solution of an integer constraint system $\mathcal{P} = (\mathcal{X}, C)$ by performing $O((|\mathcal{X}|+|C|)^3)$ integer comparisons and image computation by some bounded-increasing functions. □

Remark 3.6. We deduce that any integer constraint system is MFP-flattable.

4 Interval Constraints

In this section, we provide a cubic time constraint solver for intervals. Our solver is based on the usual [SW04, TG07] encoding of intervals by two integers in $\mathcal{Z}$. The main challenge is the translation of an interval constraint system with full multiplication into an integer constraint system.

An *interval* I is subset of $\mathbb{Z}$ of the form $\{x \in \mathbb{Z};\ a \leq x \leq b\}$ where $a, b \in \mathcal{Z}$. We denote by $\mathcal{I}$ the complete lattice of intervals partially ordered with the inclusion relation $\sqsubseteq$. The *inverse* $-I$ of an interval I, the *sum* and the *multiplication* of two intervals I_1 and I_2 are defined as follows:

$$-I = \{-x;\ x \in I\} \qquad \begin{array}{l} I_1 + I_2 = \{x_1 + x_2;\ (x_1, x_2) \in I_1 \times I_2\} \\ I_1\ .\ I_2 = \bigsqcup\{x_1.x_2;\ (x_1, x_2) \in I_1 \times I_2\} \end{array}$$

We consider interval constraints of the following forms where $I \in \mathcal{I}$:

$$X \sqsupseteq -X_1 \quad X \sqsupseteq I \quad X \sqsupseteq X_1 + X_2 \quad X \sqsupseteq X_1 \sqcap I \quad X \sqsupseteq X_1.X_2$$

Observe that we allow arbitrary multiplication between intervals, but we restrict intersection to intervals with a constant interval.

We say that an interval constraint system $\mathcal{P} = (\mathcal{X}, C)$ has the positive-multiplication property if for any constraint $c \in C$ of the form $X \sqsupseteq X_1.X_2$, the intervals $\Lambda_{\mathcal{P}}(X_1)$ and $\Lambda_{\mathcal{P}}(X_2)$ are included in $\mathbb{N}$. Given an interval constraint system $\mathcal{P} = (\mathcal{X}, C)$ we can effectively compute an interval constraint system $\mathcal{P}' = (\mathcal{X}', C')$ satisfying this property and such that $\mathcal{X} \subseteq \mathcal{X}'$ and $\Lambda_{\mathcal{P}}(X) = \Lambda_{\mathcal{P}'}(X)$ for any $X \in \mathcal{X}$. This constraint system $\mathcal{P}'$ is obtained from $\mathcal{P}$ by replacing the constraints $X \sqsupseteq X_1.X_2$ by the following constraints:

$$\begin{array}{ll} X \sqsupseteq X_{1,u}.X_{2,u} & X_{1,u} \sqsupseteq X_1 \sqcap \mathbb{N} \\ X \sqsupseteq X_{1,l}.X_{2,l} & X_{2,u} \sqsupseteq X_2 \sqcap \mathbb{N} \\ X \sqsupseteq -X_{1,u}.X_{2,l} & X_{1,l} \sqsupseteq (-X_1) \sqcap \mathbb{N} \\ X \sqsupseteq -X_{1,l}.X_{2,u} & X_{2,l} \sqsupseteq (-X_2) \sqcap \mathbb{N} \end{array}$$

Intuitively $X_{1,u}$ and $X_{2,u}$ corresponds to the positive parts of X_1 and X_2, while $X_{1,l}$ and $X_{2,l}$ corresponds to the negative parts.

Let us provide our construction for translating an interval constraint system $\mathcal{P} = (\mathcal{X}, C)$ having the positive multiplication property into an integer constraint system $\mathcal{P}' = (\mathcal{X}', C')$. Since an interval I can be naturally encoded by two integers $I^-, I^+ \in \mathcal{Z}$ defined as the least upper bound of respectively $-I$ and I, we naturally assume that $\mathcal{X}'$ contains two integer variable X^- and X^+ encoding each interval variable $X \in \mathcal{X}$. In order to extract from the least solution of $\mathcal{P}'$ the least solution of $\mathcal{P}$, we are looking for an integer constraint system $\mathcal{P}'$ satisfying $(\Lambda_{\mathcal{P}}(X))^- = \Lambda_{\mathcal{P}'}(X^-)$ and $(\Lambda_{\mathcal{P}}(X))^+ = \Lambda_{\mathcal{P}'}(X^+)$ for any $X \in \mathcal{X}$.

As expected, a constraint $X \sqsupseteq -X_1$ is converted into $X^+ \geq X_1^-$ and $X^- \geq X_1^+$, a constraint $X \sqsupseteq I$ into $X^+ \geq I^+$ and $X^- \geq I^-$, and a constraint $X \sqsupseteq X_1 + X_2$ into $X^- \geq X_1^- + X_2^-$ and $X^- \geq X_1^- + X_2^-$. However, a constraint $X \sqsupseteq X_1 \sqcap I$ cannot be simply translated into $X^- \geq X_1^- \wedge I^-$ and $X^+ \geq X_1^+ \wedge I^+$. In fact, these constraints may introduce imprecision when $\Lambda_{\mathcal{P}}(X) \cap I = \emptyset$. We use test functions to overcome this problem. Such a constraint is translated into the following integer constraints:

$$\begin{array}{rcl} X^- & \geq & \theta_{\geq -I^+}(X_1^-, \theta_{\geq -I^-}(X_1^+, X_1^- \wedge I^-)) \\ X^+ & \geq & \theta_{\geq -I^-}(X_1^+, \theta_{\geq -I^+}(X_1^-, X_1^+ \wedge I^+)) \end{array}$$

For the same reason, the constraint $X \sqsupseteq X_1.X_2$ cannot be simply converted into $X^- \geq \mathsf{mul}_-(X_1^-, X_2^-)$ and $X^+ \geq \mathsf{mul}_+(X_1^+, X_2^+)$. Instead, we consider the following constraints:

$$\begin{array}{rcl} X^- & \geq & \theta_{>-\infty}(X_1^-, \theta_{>-\infty}(X_1^+, \theta_{>-\infty}(X_2^-, \theta_{>-\infty}(X_2^+, \mathsf{mul}_-(X_1^-, X_2^-))))) \\ X^+ & \geq & \theta_{>-\infty}(X_1^+, \theta_{>-\infty}(X_1^-, \theta_{>-\infty}(X_2^+, \theta_{>-\infty}(X_2^-, \mathsf{mul}_+(X_1^+, X_2^+))))) \end{array}$$

Observe in fact that $X^- \geq \mathsf{mul}_-(X_1^-, X_2^-)$ and $X^+ \geq \mathsf{mul}_+(X_1^+, X_2^+)$ are precise constraint when the intervals $I_1 = \Lambda_{\mathcal{P}}(X_1)$ and $I_2 = \Lambda_{\mathcal{P}}(X_2)$ are non empty.

Since, if this condition does not hold then $I_1.I_2 = \emptyset$, the previous encoding consider this case by testing if the values of X_1^-, X_1^+, X_2^-, X_2^+ are strictly greater than $-\infty$.

Now, observe that the integer constraint system $\mathcal{P}'$ satisfies the equalities $(\Lambda_{\mathcal{P}}(X))^+ = \Lambda_{\mathcal{P}'}(X^+)$ and $(\Lambda_{\mathcal{P}}(X))^- = \Lambda_{\mathcal{P}'}(X^-)$ for any $X \in \mathcal{X}$. Thus, we have proved the following theorem.

Theorem 4.1. *The least solution of an interval constraint system $\mathcal{P} = (\mathcal{X}, C)$ with full multiplication can by computed in time $O((|\mathcal{X}| + |C|)^3)$ with integer manipulations performed in $O(1)$.*

Remark 4.2. We deduce that any interval constraint system is MFP-flattable.

5 Conclusion and Future Work

In this paper we have extended the acceleration framework from symbolic verification to the computation of MFP-solutions in data-flow analysis. Our approach leads to an efficient cubic-time algorithm for solving interval constraints with full addition and multiplication, and intersection with a constant.

As future work, it would be interesting to combine this result with strategy iteration techniques considered in [TG07] in order to obtain a polynomial time algorithm for the extension with full intersection. We also intend to investigate the application of the acceleration framework to other abstract domains.

References

[ABS01] A. Annichini, A. Bouajjani, and M. Sighireanu. TReX: A tool for reachability analysis of complex systems. In *Proc. 13th Int. Conf. Computer Aided Verification (CAV'2001), Paris, France, July 2001*, volume 2102 of *Lecture Notes in Computer Science*, pages 368–372. Springer, 2001.

[BFLP03] S. Bardin, A. Finkel, J. Leroux, and L. Petrucci. FAST: Fast Acceleration of Symbolic Transition systems. In *Proc. 15th Int. Conf. Computer Aided Verification (CAV'2003), Boulder, CO, USA, July 2003*, volume 2725 of *Lecture Notes in Computer Science*, pages 118–121. Springer, 2003.

[BFLS05] S. Bardin, A. Finkel, J. Leroux, and P. Schnoebelen. Flat acceleration in symbolic model checking. In *Proc. 3rd Int. Symp. Automated Technology for Verification and Analysis (ATVA'05), Taipei, Taiwan, Oct. 2005*, volume 3707 of *Lecture Notes in Computer Science*, pages 474–488. Springer, 2005.

[BGWW97] B. Boigelot, P. Godefroid, B. Willems, and P. Wolper. The power of QDDs. In *Proc. Static Analysis 4th Int. Symp. (SAS'97), Paris, France, Sep. 1997*, volume 1302 of *Lecture Notes in Computer Science*, pages 172–186. Springer, 1997.

[BW94] B. Boigelot and P. Wolper. Symbolic verification with periodic sets. In *Proc. 6th Int. Conf. Computer Aided Verification (CAV'94), Stanford, CA, USA, June 1994*, volume 818 of *Lecture Notes in Computer Science*, pages 55–67. Springer, 1994.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM Symp. Principles of Programming Languages, Los Angeles, CA, USA*, pages 238–252. ACM Press, 1977.

[CC92] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *Proc. 4th Int. Symp. Programming Language Implementation and Logic Programming (PLILP'92), Leuven, Belgium, Aug. 1992*, volume 631 of *Lecture Notes in Computer Science*, pages 269–295. Springer, 1992.

[CGG+05] A. Costan, S. Gaubert, E. Goubault, M. Martel, and S. Putot. A policy iteration algorithm for computing fixed points in static analysis of programs. In *In Proc. 7th Int. Conf. on Computer Aided Verification (CAV'05), Edinburgh, Scotland, UK, July 2005*, Lecture Notes in Computer Science, pages 462–475. Springer, 2005.

[CJ98] H. Comon and Y. Jurski. Multiple counters automata, safety analysis and Presburger arithmetic. In *Proc. 10th Int. Conf. Computer Aided Verification (CAV'98), Vancouver, BC, Canada, June-July 1998*, volume 1427 of *Lecture Notes in Computer Science*, pages 268–279. Springer, 1998.

[FIS03] A. Finkel, S. P. Iyer, and G. Sutre. Well-abstracted transition systems: Application to FIFO automata. *Information and Computation*, 181(1):1–31, 2003.

[FL02] A. Finkel and J. Leroux. How to compose Presburger-accelerations: Applications to broadcast protocols. In *Proc. 22nd Conf. Found. of Software Technology and Theor. Comp. Sci. (FST&TCS'2002), Kanpur, India, Dec. 2002*, volume 2556 of *Lecture Notes in Computer Science*, pages 145–156. Springer, 2002.

[Kar76] M. Karr. Affine relationship among variables of a program. *Acta Informatica*, 6:133–141, 1976.

[Las] LASH homepage. `http://www.montefiore.ulg.ac.be/~boigelot/research/lash/`.

[LS05] J. Leroux and G. Sutre. Flat counter automata almost everywhere! In *Proc. 3rd Int. Symp. Automated Technology for Verification and Analysis (ATVA'05), Taipei, Taiwan, Oct. 2005*, volume 3707 of *Lecture Notes in Computer Science*, pages 489–503. Springer, 2005.

[MOS04] M. Müller-Olm and H. Seidl. A note on Karr's algorithm. In *Proc. 31st Int. Coll. on Automata, Languages and Programming (ICALP'04), Turku, Finland, July 2004*, Lecture Notes in Computer Science, pages 1016 – 1028. Springer, 2004.

[SW04] Z. Su and D. Wagner. A class of polynomially solvable range constraints for interval analysis without widenings and narrowings. In *Proc. 10th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04), Barcelona, Spain, Mar.-Apr. 2004*, volume 2988 of *Lecture Notes in Computer Science*, pages 280–295. Springer, 2004.

[TG07] H. Seidl T. Gawlitza. Precise fixpoint computation through strategy iteration. In *Proc. 16th European Symp. on Programming (ESOP'2007), Braga, Portugal, April 2007*, volume 4421 of *Lecture Notes in Computer Science*, pages 300–315. Springer, 2007.

A Proof of Lemma 3.2

Lemma 3.2. *Let $\mathcal{P}$ be a cyclic bounded-increasing constraint system. If ρ_0 is a pre-solution of $\mathcal{P}$ that does not lower-saturate any constraint, then either ρ_0 is a solution or $[\![C]\!]^*(\rho_0)$ upper-saturates a constraint.*

Proof. Let $X_0 \to c_1 \to X_1 \to \cdots \to c_n \to X_n = X_0$ be the unique (up to a cyclic permutation) cycle in the graph associated to $\mathcal{P}$.

Let us prove that for any pre-solution ρ that is not a solution and that does not lower-saturate any constraint, there exists a constraint $c \in C$ such that $\rho' = [\![c]\!](\rho)$ is a pre-solution satisfying $\rho' > \rho$ that either upper-saturates a constraint or that is not a solution. Since ρ is not a solution, there exists a constraint c_{i-1} such that the valuation $\rho' = [\![c_{i-1}]\!](\rho)$ satisfies $\rho' \not\leq \rho$. As c_i only modifies the value of X_{i-1}, we get $\rho'(X_{i-1}) > \rho(X_{i-1})$. Observe that if ρ' upper-saturates c_{i-1} we are done. Let us assume that ρ' does not upper-saturate c_{i-1}. Let us show that ρ' is not a solution of C. As ρ is a pre-solution and c_i is the unique variable that modifies X_i, we have $\rho(X_i) \leq [\![c_i]\!](\rho)(X_i)$. Since $\rho(X_{i-1}) < \rho'(X_{i-1})$ and c_i is neither upper-saturate nor lower-saturate for ρ' and ρ we get $[\![c_i]\!](\rho)(X_i) < [\![c_i]\!](\rho')(X_i)$ from $\rho < \rho'$. The relations $\rho(X_i) = \rho'(X_i)$, $\rho(X_i) \leq [\![c_i]\!](\rho)(X_i)$ and $[\![c_i]\!](\rho)(X_i) < [\![c_i]\!](\rho')(X_i)$ provide the relation $[\![c_i]\!](\rho')(X_i) > \rho'(X_i)$. Thus ρ' is not a solution.

Assume by contradiction that $[\![C]\!]^*(\rho_0)$ does not upper-saturate a constraint. Since ρ_0 is a pre-solution that is not a solution and such that any constraint $c \in C$ is not lower-saturated, from the previous paragraph, we get an infinite sequence $\rho_0 < \ldots < \rho_k < \ldots$ of valuations satisfying $\rho_k \leq [\![C]\!]^*(\rho_0)$. We deduce that there exists a variable X_i such that $\vee_{k\geq 0}\rho_k(X_i) = +\infty$. Thus $[\![C]\!]^*(\rho_0)(X_i) = +\infty$ and we have proved that $[\![C]\!]^*(\rho_0)$ upper-saturates c_{i+1}. This contradiction proves that $[\![C]\!]^*(\rho_0)$ upper-saturates at least one constraint in C. □

3 Proof of Proposition 3.3

Proposition 3.3. *The algorithm* CyclicSolve *returns $[\![C]\!]^*(\rho_0)$ for any cyclic constraint system $\mathcal{P}$ and for any valuation ρ_0.*

Proof. Let ρ_1, ρ_2, ρ_3, ρ_4, and ρ_5 be the value of ρ just after the 1st, the 2sd, the 3th, the 4th and the 5th loops.

Let us first show that if the i_0th iteration of the second loop does not modify the valuation ρ then ρ_2 is a solution of $\mathcal{P}$. Observe that the iterations $i_0, \ldots, n$ of the first loop and the iterations $1, \ldots, i_0-1$ of the second loop provide a valuation ρ such that $\rho(X_i) \geq [\![c_i]\!](\rho)(X_i)$ for any $i \neq i_0$. As the i_0th iteration of the second loop does not modify ρ we deduce that $\rho(X_{i_0}) \geq [\![c_{i_0}]\!](X_{i_0})$. Therefore ρ is a solution. We deduce that ρ remains unchanged during the remaining iterations $i_0, \ldots, n$ of the second loop. Thus ρ_2 is a solution of $\mathcal{P}$.

Assume that ρ_2 is not a pre-solution of $\mathcal{P}$. There exists i_0 such that $[\![c_{i_0}]\!](\rho)(X_{i_0}) \not\geq \rho(X_{i_0})$. We deduce that the value of ρ has not been modified at the i_0th iteration of the 2sd loop. Thus, from the previous paragraph, ρ_2 is a solution.

Next, assume that a constraint c_{i_0} is lower-saturated by ρ_2. Since after the i_0-iteration of the first loop we have $\rho(X_{i_0}) \geq [\![c_{i_0}]\!](\bot)$, we deduce that the i_0th iteration of the second loop does not modify ρ. From the first paragraph we also deduce that ρ_2 is a solution of $\mathcal{P}$.

As the line 9 of the algorithm detects if ρ_2 is a solution, we can assume that ρ_2 is not a solution. From the two previous paragraph we deduce that ρ_2 is a pre-solution of $\mathcal{P}$ and the constraints are not lower-saturated. From Lemma 3.2 we deduce that $[\![C]\!]^*(\rho_2)$ upper-saturates at least one constraint denoted by c_{i_0}. Observe that $[\![C]\!]^*(\rho_2) = [\![C]\!]^*(\rho_0)$.

Let us show that $[\![c]\!]([\![C]\!]^*(\rho_0)) = [\![C]\!]^*(\rho_0)$ for any constraint $c \in C$. Since ρ_2 is a pre-solution we get $[\![C]\!]([\![C]\!]^*(\rho_2)) = [\![C]\!]^*(\rho_2)$. Moreover, as the output variables of two distinct constraints are distinct, we deduce that $[\![c]\!]([\![C]\!]^*(\rho_2)) = [\![C]\!]^*(\rho_2)$ for any constraint $c \in C$. As $[\![C]\!]^*(\rho_0) = [\![C]\!]^*(\rho_2)$ we get the property.

We deduce that the valuation $\rho' = \rho \wedge [\![c]\!](\rho)$ satisfies $[\![C]\!]^*(\rho_0) \leq \rho'$ for any valuation ρ such that $[\![C]\!]^*(\rho_0) \leq \rho$ and for any constraint $c \in C$.

After the 3th loop of the algorithm, we have $[\![C]\!]^*(\rho_0) \leq \rho_3$, the previous paragraph proves that $[\![C]\!]^*(\rho_0) \leq \rho$ is an invariant of the remaining of the program. Observe that at the i_0-th iteration of the 4th loop, we have $\rho(X_{i_0}) = [\![C]\!]^*(\rho_0)(X_{i_0})$. Thanks to the remaining iterations $i_0 + 1, \ldots, n$ of the 4th loop and the first iterations $1, \ldots i_0 - 1$ of the 5th loop, we get $\rho(X_i) = [\![C]\!]^*(\rho_0)(X_i)$ for any i since $[\![c]\!]([\![C]\!]^*(\rho_0)) = [\![C]\!]^*(\rho_0)$ for any constraint $c \in C$. Thus at this point of the execution we have $\rho = [\![C]\!]^*(\rho_0)$. Observe that ρ is unchanged during the remaining iterations of the 5th loop. Thus, the algorithm returns $[\![C]\!]^*(\rho_0)$. □

3 Proof of Proposition 3.4

Proposition 3.4. *The algorithm* SolveBI *returns the least solution* $[\![C]\!]^*(\rho_0)$ *of a bounded-increasing constraint system* $\mathcal{P}$ *greater than a valuation* ρ_0*. Moreover, the number of times the while loop is executed is bounded by one plus the number of constraints that are upper-saturated for* $[\![C]\!]^*(\rho_0)$ *but not for* ρ_0*.*

Proof. Note that λ is a partially defined function from $\mathcal{X}$ to C. At the beginning of the while loop this function is empty. Then, it is updated when the algorithm replaces a valuation ρ by $\rho \vee [\![c]\!](\rho)$. Denoting by X the output variable of c, the value $\lambda(X)$ becomes equal to c. That means λ keeps in memory the last constraint that have modified a variable. Observe also that initially $\rho = \rho_0 \vee [\![C]\!](\rho_0)$. Thus, if during the execution of the algorithm $\rho(X)$ is updates by a constraint c then necessary c is not lower-saturated. That means if $\lambda(X)$ is defined then $c = \lambda(X)$ is not lower-saturated.

Let ρ_0 and ρ_1 be the values of ρ respectively before and after the execution of the first two nested loops (line 5-9) and let ρ_2 be the value of ρ after the

execution of line 14.

We are going to prove that if the sets of upper-saturated constraints for ρ_0 and ρ_1 are equal and if there does not exist a cycle satisfying the condition given line 10, then ρ_1 is a solution of $\mathcal{P}$. Let us consider the subset set of constraints $C' = \{\lambda(X);\ X \in \mathcal{X}\}$ and let us consider the graph G' associated to the constraint system $(\mathcal{X}, C')$. We construct the graph G_1 obtained from G by keeping only the transitions (X, c) if $c = \lambda(X)$ and the transitions (X_i, c). Observe that G_1 is acyclic. Thus, there exists an enumeration $c_1, \ldots, c_m$ of the set of constraints C' such that if there exists a path from c_{i_1} to c_{i_2} in G_1 then $i_1 \leq i_2$. Let us denote by X_i the output variable of c_i.

Let us prove by induction over i that for any $j \in \{1, \ldots, i\}$ we have $\rho_1(X_j) \leq [\![c_1 \ldots c_j]\!](\rho_0)(X_j)$. The rank $i = 0$ is immediate since in this case $\{1, \ldots, i\}$ is empty. Let us assume that rank $i - 1 < n$ is true and let us prove the rank i. Since $\lambda(X_i) = c_i$ we deduce that the valuation $\rho_1(X_i)$ has been modified thanks to c_i. Thus, denoting by ρ the valuation in the algorithm just before this update, we deduce that $\rho_1(X_i) = [\![c_i]\!](\rho)(X_i)$ and $\rho_0 \leq \rho \leq \rho_1$. Let us prove that $\rho(X_{i,j}) \leq [\![c_1 \ldots c_{i-1}]\!](\rho_0)(X_{i,j})$ for any input variable $X_{i,j}$ of c_i. Observe that if $X_{i,j} \in \mathcal{X}'$ then $\rho_1(X_{i,j}) = \rho(X_{i,j}) = \rho_0(X_{i,j})$ by construction of λ and in particular $\rho(X_{i,j}) \leq [\![c_1 \ldots c_{i-1}]\!](\rho_0)(X_{i,j})$ since $c_1, \ldots, c_{i-1}$ do not modify the variable $X_{i,j}$. Otherwise, if $X_{i,j} \in \mathcal{X}'$, there exists $i' < i$ satisfying $X_{i,j} = X_{i'}$. By induction hypothesis, we have $\rho_1(X_{i'}) \leq [\![c_1 \ldots c_{i'}]\!](\rho_0)(X_{i'})$. Since $c_1, \ldots, c_m$ have distinct output variables, we deduce that $[\![c_1 \ldots c_{i'}]\!](\rho_0)(X_{i'}) = [\![c_1 \ldots c_{i-1}]\!](\rho_0)(X_{i'})$. Thus $\rho_1(X_{i,j}) \leq [\![c_1 \ldots c_{i-1}]\!](\rho_0)(X_{i,j})$ and from $\rho \leq \rho_1$, we get $\rho(X_{i,j}) \leq [\![c_1 \ldots c_{i-1}]\!](\rho_0)(X_{i,j})$ for any input variable $X_{i,j}$. Therefore $[\![c_i]\!](\rho)(X_i) \leq [\![c_1 \ldots c_i]\!](\rho_0)(X_i)$. From $\rho_1(X_i) = [\![c_i]\!](\rho)(X_i)$, we get $\rho_1(X_i) \leq [\![c_1 \ldots c_i]\!](\rho_0)(X_i)$ and we have proved the induction.

We deduce the relation $\rho_1 \leq [\![c_1 \ldots c_m]\!](\rho_0)$ since c_1, ..., c_m have distinct output variables. Observe that after the first execution of the loop 6-9, we get $\rho \geq [\![c_1]\!](\rho_0)$, after the second $\rho \geq [\![c_1.c_2]\!](\rho_0)$. By induction, after m executions we get $\rho \geq [\![c_1 \ldots c_m]\!](\rho_0) \geq \rho_1$. Since $m \leq |C|$, this loop is executed at least one more time. Note that after this execution, we have $\rho \geq [\![c]\!](\rho_1)$ for any $c \in C$. Since $\rho_1 \geq \rho$, we have proved that $\rho_1 \geq [\![C]\!](\rho_1)$. Therefore ρ_1 is a solution of C.

Next, assume that there exists a cycle $X_0 \to c_1 \to X_1 \cdots c_n \to X_n = X_0$ that satisfies $c_i = \lambda(X_i)$. From the first paragraph we deduce that $c_1, \ldots, c_n$ are not lower-saturated. Let us prove that there exists a constraint upper-saturated for ρ_2 that is not upper-saturated for ρ_0. Naturally, if there exists a constraints upper-saturated from ρ_1 that is not upper-saturated for ρ_0, since $\rho_1 \leq \rho_2$, we are done. Thus, we can assume that the constraints c_1, ..., c_n are not upper-saturated for ρ_1. By definition of λ, we get $\rho_i(X_i) \leq [\![c_i]\!](\rho)$. Thus ρ' is a pre-solution of $\mathcal{P}'$. Let X_i be the last variable amongst $X_0, \ldots, X_{n-1}$ that have been updated. Since c_{i+1} is not upper-saturated and not lower-saturated for ρ_1 and since the value of X_i has increased when this last update appeared, we deduce

that $\rho'(X_{i+1}) \not\geq [\![c_{i+1}]\!](\rho')(X_{i+1})$. Thus ρ' is not a solution and from lemma 3.2 we deduce that ρ'' upper-saturates at least one constraint c_i. Thus ρ_2 upper-saturates a constraints that is not upper-saturated by ρ_1.

Finally, note that each time the while loop is executed at least one bounded-increasing constraint becomes upper-saturated. As every upper-saturated constraint remains upper-saturated, we are done. □