Logico-Numerical Abstract Acceleration and Application to the Verification of Data-Flow Programs

Peter Schrammel, Bertrand Jeannet

ISSN 0249-6399 ISRN INRIA/RR--7630-v2--FR+ENG

RESEARCH REPORT
N° 7630-v2
Nov 2012
Project-Teams POP ART

Logico-Numerical Abstract Acceleration and Application to the Verification of Data-Flow Programs

Peter Schrammel*, Bertrand Jeannet

Project-Teams POP ART

Research Report n° 7630-v2 — Nov 2012 — 22 pages

Abstract: Acceleration methods are commonly used for speeding up the convergence of loops in reachability analysis of counter machine models. Applying these methods to synchronous data-flow programs with Boolean and numerical variables, *e.g.*, LUSTRE programs, requires the enumeration of the Boolean states in order to obtain a control flow graph (CFG) with numerical variables only. Our goal is to apply acceleration techniques to data-flow programs without resorting to this exhaustive enumeration. To this end, we present (1) *logico-numerical abstract acceleration methods* for CFGs with Boolean and numerical variables and (2) partitioning techniques that make logical-numerical abstract acceleration effective. Experimental results show that incorporating these methods in a verification tool based on abstract interpretation provides not only significant advantage in terms of accuracy, but also a gain in performance in comparison to standard techniques.

Key-words: Verification, Static Analysis, Abstract Interpretation, Abstract Acceleration, Control Flow Graph Partitioning.

* This work was supported by the INRIA large-scale initiative SYNCHRONICS

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Accélération abstraite logico-numérique et application à la vérification de programme flot de données

Résumé : Les méthodes d'accélération sont utilisées pour faire converger les boucles dans l'analyse d'accéssibilité de machines à compteurs. L'application de ces méthodes aux programmes synchrones flot de données avec des variables booléennes et numériques, *e.g.*, des programmes en LUSTRE, exige l'enumeration des états booléens pour obtenir un graphe de contrôle purement numérique. Notre but consiste en l'application de méthodes d'accélération aux programmes flot de données sans enumeration exhaustive : nous proposons (1) des méthodes d'accélération abstraite logico-numérique pour des graphes de contrôle avec des variables booléennes et numériques et (2) des techniques de partitionnement pour rendre efficace l'accélération abstraite logico-numérique. Nos résultats expérimentaux montrent que l'intégration de ces méthodes dans un outil basé sur l'interprétation abstraite améliore non seulement la précision, mais elle représente aussi un gain en performance par rapport au techniques standard.

Mots-clés : vérification, analyse statique, interprétation abstraite, accélération abstraite, partitionnement de graphe de contrôle.

Table of Contents

1 Introduction

This paper deals with the *verification of safety properties* about *logico-numerical* data-flow programs, *i.e.*, programs manipulating Boolean and numerical variables. Verification of such properties amounts to checking whether the reachable state space stays within the invariant specified by the property.

Classical applications are safety-critical controllers as found in modern transport systems, as well as static checking of high-level simulation models, *e.g.* a model of a production line as depicted in Fig. 1. In such systems the properties to be proved, like throughput and workload, depend essentially on the relationships between the numerical variables of the system. Yet, there is an important observation that we are going to exploit: In many of these control systems large parts of the program simply count time or events, or, more generally, they perform rather regular linear arithmetic operations. Hence, it is appropriate to take advantage of a specialized analysis method that exploits this regularity in order to improve verification performance and precision. In this paper, we will consider abstract acceleration [1] for this purpose, which aims at computing in one step the effect of an unbounded number of loop iterations. However, at the same time, we are confronted with a huge Boolean state space in the applications we want to verify. Our contribution is therefore to extend abstract acceleration from purely numerical programs to logico-numerical programs in an efficient way.

Verifying logico-numerical data-flow programs by abstract interpretation. The reachability problem is not decidable for this class of programs, so analysis methods are incomplete. Abstract interpretation [2] is a classical method with guaranteed termination for the price of an approximate analysis result. The key idea is to approximate sets of states S by elements $S^\sharp$ of an *abstract domain.*

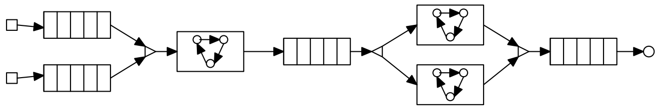


Fig. 1: Example of a production line with buffers, machines, and splitting and combining material flows.

A classical abstract domain for numerical invariants in $\wp(\mathbb{R}^n)$ is the domain of convex polyhedra $Pol(\mathbb{R}^n)$ [3]. An approximation $S^\sharp$ of the reachable set S is then computed by iteratively solving the fixed point equation characterizing S *in the abstract domain.* To ensure termination when the abstract domain contains infinitely increasing chains, one applies an extrapolation operator called *widening*, which induces additional approximations.

Since the analysis with a single abstract value gives only coarse results, it is usually conducted over a *control flow graph* (CFG) of the program. In the case of imperative programs, such a control graph can be obtained easily by associating control points with programming constructs as if-then-else or while. Data-flow programs do not have such constructs; yet, one can use finite-type variables such as Booleans to generate a control structure. Thus, the classical approach is to explicitly unfold the Boolean control structure by *enumerating* the Boolean state space and to analyze the numerical variables on the obtained CFG using a numerical abstract domain. The problem is that the analysis becomes intractable with larger programs because the number of control locations grows exponentially with the number of Boolean states.

Jeannet [4] proposed a method for iteratively refining the control structure and analyzing the system using a *logico-numerical abstract domain*, making it possible to deal with Boolean variables symbolically. We want to complement this approach with new partitioning techniques and analysis methods.

Abstract acceleration. *Acceleration* [5] refers to a set of techniques aiming at exactly computing the effects of loops in numerical transition systems like counter machines, and ultimately at computing the exact reachability set of such systems, usually using Presburger arithmetic. *Abstract acceleration* [1] reformulates these concepts within an abstract interpretation approach: it aims at computing the best correct approximation of the effect of loops in a given abstract domain (currently only convex polyhedra have been considered).

These techniques can analyze only purely numerical programs with a given CFG, of which the size often becomes prohibitively large. Furthermore, they do not consider numerical inputs. In a previous paper [6], we already extended abstract acceleration to numerical inputs.

Contributions. The missing link in the application of abstract acceleration to logico-numerical programs, such as LUSTRE programs, is an efficient method for (i) building an appropriate CFG without resorting to Boolean state space enumeration, and (ii) analyzing it using abstract acceleration. Our methods allow us to treat these two problems independently of each other.

Our contributions can be summarized as follows:

1. We propose methods for *accelerating self-loops* in the CFG of *logico-numerical* data-flow programs.
2. We define *Boolean partitioning heuristics* that favor the applicability of abstract acceleration and enable a reasonably precise reachability analysis.
3. We provide *experimental results* on the use of abstract acceleration enhancing the analysis of logico-numerical programs.

Compared to other approaches, the partitioning heuristics that we propose are based on structural properties of the program, namely the numerical transitions, and thus, they are complementary to most common techniques based on abstract or concrete counter-example refinement. In this paper we consider only partitions of the Boolean state space, in contrast to the tool NBAC [4], which in addition partitions according to numerical constraints.

Organisation of the article. §2 gives an introduction to the abstract interpretation of logico-numerical programs, partitioning, and abstract acceleration. §3 and §4 describe our contributions on logico-numerical abtract acceleration methods, §5 presents our experimental results, and finally §6 discusses related work and concludes.

2 Analysis of Logico-Numerical Programs

Program model. We consider programs modeled as a symbolic transition system $\begin{cases}\mathcal{I}(\boldsymbol{s}) \\ \mathcal{A}(\boldsymbol{s},\boldsymbol{i}) \rightarrow \boldsymbol{s}' = \boldsymbol{f}(\boldsymbol{s},\boldsymbol{i})\end{cases}$ where (1) $\boldsymbol{s}$ and $\boldsymbol{i}$ are vectors of state and input variables, that are either Boolean or numerical; (2) $\mathcal{I}(\boldsymbol{s})$ is an initial condition on state variables; (3) $\mathcal{A}(\boldsymbol{s},\boldsymbol{i})$ is an *assertion* constraining input variables depending on state variables, and typically modeling the environment of the program; (4) $\boldsymbol{f}$ is the vector of transition functions. An example of such a program is

$$\begin{cases} \mathcal{I}(b,x) = \neg b \wedge (x = 0) \\ 1 \leq \xi \leq 3 \rightarrow \begin{pmatrix} b' \\ \\ x' \end{pmatrix} = \begin{pmatrix} (b \wedge x \leq 5) \vee \beta \\ \begin{cases} x+\xi & \text{if } b \wedge x \leq 5 \\ 0 & \text{otherwise} \end{cases} \end{pmatrix} \end{cases}$$

An execution of such a system is a sequence $\boldsymbol{s^0} \xrightarrow{\boldsymbol{i^0}} \boldsymbol{s^1} \xrightarrow{\boldsymbol{i^1}} \ldots \boldsymbol{s^k} \xrightarrow{\boldsymbol{i^k}} \ldots$ such that $\mathcal{I}(\boldsymbol{s^0})$ and for any $k \geq 0$, $\mathcal{A}(\boldsymbol{s^k},\boldsymbol{i^k}) \wedge \boldsymbol{s^{k+1}} = \boldsymbol{f}(\boldsymbol{s^k},\boldsymbol{i^k})$).

The front-end compilation of synchronous data-flow programs, like LUSTRE, produces such a program model, that also includes various models of counter automata (by emulating locations using Boolean variables) [5].

We will use the following notations:

$\boldsymbol{s} = (\boldsymbol{b},\boldsymbol{x})$: state variable vector, with $\boldsymbol{b}$ Boolean and $\boldsymbol{x}$ numerical subvectors
$\boldsymbol{i} = (\boldsymbol{\beta},\boldsymbol{\xi})$: input variable vector, with $\boldsymbol{\beta}$ Boolean and $\boldsymbol{\xi}$ numerical subvectors
$\mathcal{C}(\boldsymbol{x},\boldsymbol{\xi})$: constraints over numerical variables, seen as a vector of Boolean decisions (for short $\mathcal{C}$)

Transitions are written in the form $\mathcal{A}(\boldsymbol{b},\boldsymbol{\beta},\mathcal{C}) \to \begin{pmatrix} \boldsymbol{b}' \\ \boldsymbol{x}' \end{pmatrix} = \begin{pmatrix} \boldsymbol{f}^b(\boldsymbol{b},\boldsymbol{\beta},\mathcal{C}) \\ \boldsymbol{f}^x(\boldsymbol{b},\boldsymbol{\beta},\mathcal{C},\boldsymbol{x},\boldsymbol{\xi}) \end{pmatrix}$. Numerical transition functions are written as a disjunction of guarded actions: $\boldsymbol{f}^x(\boldsymbol{b},\boldsymbol{\beta},\mathcal{C},\boldsymbol{x},\boldsymbol{\xi}) = \bigvee_i \big(g_i(\boldsymbol{b},\boldsymbol{\beta},\mathcal{C}) \to \boldsymbol{a}_i^x(\boldsymbol{x},\boldsymbol{\xi})\big)$ with $\neg(g_i \wedge g_j)$ for $i \neq j$. The program example above conforms to these notations.

2.1 Abstract interpretation

The state space induced by logico-numerical programs has the structure $E = \mathbb{B}^m \times \mathbb{R}^n$. As mentioned in the introduction, we adopt the abstract interpretation framework so as to abstract the equation $S = S^0 \cup post(S), S \in \wp(E)$ in an abstract domain and to solve it iteratively, using widening to ensure convergence.

We consider the domain $A = \wp(\mathbb{B}^m) \times Pol(\mathbb{R}^n)$ of *convex states* [7], which approximates a set of states coarsely by a conjunction of a Boolean formula and a single convex polyhedron. For instance the formula $(b \wedge x \leq 2) \vee (\neg b \wedge x \leq 4)$ is abstracted by $true \wedge x \leq 4$.

Partitioning the state space. We use state space partitioning to obtain a CFG in which each equivalence class of the partition corresponds to a location.

Definition 1. *A symbolic control flow graph (CFG) of a symbolic transition system is a directed graph* $\langle \Pi, \Pi_0, \rightsquigarrow \rangle$ *where*

- Π *is the set of locations; each location* $\ell \in \Pi$ *is characterized by its location invariant* $\varphi_\ell(\boldsymbol{s})$*, such that* $\{\varphi_\ell(\boldsymbol{s}) \mid \ell \in \Pi\}$ *forms a partition of* E*.*
- Π_0 *is the set of initial locations with* $\mathcal{I}(\boldsymbol{s}) = \bigvee_{\ell \in \Pi_0} \varphi_\ell(\boldsymbol{s})$
- $\rightsquigarrow$ *defines arcs between locations according to the transition relation:* $\exists \boldsymbol{s}, \boldsymbol{i} : \varphi_\ell(\boldsymbol{s}) \wedge \mathcal{A}(\boldsymbol{s},\boldsymbol{i}) \wedge \boldsymbol{s}' = \boldsymbol{f}(\boldsymbol{s},\boldsymbol{i}) \wedge \varphi_{\ell'}(\boldsymbol{s}') \Rightarrow \ell \rightsquigarrow \ell'$

There are several ways to define a partition inducing such a CFG. In predicate abstraction for instance, the partition is generated by considering the truth value of a finite set of predicates [8]. Here, we consider partitions defined by equivalence relations on Boolean state variables. For example, the fully partitioned CFG obtained by *enumerating* all Boolean states is characterized by the relation $\boldsymbol{b}_1 \sim \boldsymbol{b}_2 \Leftrightarrow \boldsymbol{b}_1 = \boldsymbol{b}_2$.

Simplifying a CFG. In practice, partitioning is done by incrementally dividing the locations. Furthermore arcs between locations that are proved to be *infeasible* are removed. This can be done, *e.g.* by checking the satisfiability of the transition relation, *e.g.* using an SMT solver.

At last, transition functions are simplified by *partial evaluation* (using a generalized cofactor operator, see [9]).

Analyzing a CFG. In the context of analysis by abstract interpretation, considering a CFG allows to apply widening in a more restrictive way, *e.g.* on loop heads only [10]. Also the information loss due to the convex union is limited, because we assign an abstract value to each location: We consider the compound abstract domain $(\Pi \to A)$ where the concrete states S are connected to their abstract counterparts $S^\sharp$ by the *Galois connection*:

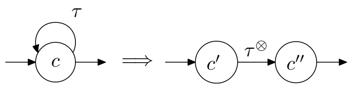


Fig. 2: Self-loop transition (left) and accelerated transition (right).

$$S^\sharp = \alpha(S) = \lambda\ell \, . \, \alpha(S \sqcap \varphi_\ell) \qquad\qquad S = \gamma(S^\sharp) = \bigcup_{\ell \in \Pi} \gamma(S^\sharp_\ell)$$

Analyzing the partitioned system amounts to computing the least fixed point $S^\sharp = S^{\sharp,0} \sqcup \lambda\ell \, . \bigsqcup_{\ell' \in \Pi} \Big(post(S^\sharp_{\ell'}) \sqcap \varphi_\ell \Big)$ where $S^\sharp, S^{\sharp,0} \in (\Pi \to A)$.

2.2 Abstract acceleration

As mentioned in the introduction, *acceleration* [5] aims at computing exactly (or precisely in the case of abstract acceleration [1, 11]) the effect of a self-loop. The basic idea is to replace a loop transition by its transitive closure (Fig. 2) by providing a formula $\tau^\otimes(X)$ computing $\tau^*(X) = \bigcup_{k \geq 0} \tau^k(X)$.

Basic concepts. A loop transition τ has the structure: $g \to a$ meaning "while guard g do action a". Our extension of abstract acceleration to numerical inputs [6] deals with loop transitions of the form

$$\underbrace{\begin{pmatrix} \mathbf{A} & \mathbf{L} \\ 0 & \mathbf{J} \end{pmatrix} \begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{\xi} \end{pmatrix} \leq \begin{pmatrix} \boldsymbol{v} \\ \boldsymbol{k} \end{pmatrix}}_{\mathbf{A}\boldsymbol{x} + \mathbf{L}\boldsymbol{\xi} \leq \boldsymbol{v} \,\wedge\, \mathbf{J}\boldsymbol{\xi} \leq \boldsymbol{k}} \to \boldsymbol{x}' = \underbrace{\begin{pmatrix} \mathbf{C} & \mathbf{T} \end{pmatrix} \begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{\xi} \end{pmatrix} + \boldsymbol{u}}_{\mathbf{C}\boldsymbol{x} + \mathbf{T}\boldsymbol{\xi} + \boldsymbol{u}} \tag{1}$$

Existing acceleration methods can deal with transitions where the matrix $\mathbf{C}$ is a diagonal matrix with zeros and ones only or when it is periodic ($\exists p > 0, l > 0 : \mathbf{C}^{p+l} = \mathbf{C}^p$). Throughout this paper, we will call such numerical transition functions *accelerable*, whereas we regard general affine transformations (with an arbitrary $\mathbf{C}$) as *non-accelerable*.

Widening and acceleration. Acceleration gives us a formula for computing the transitive closure of accelerable loop transitions. Widening is still needed in the case of non-accelerable transitions, outer loops of nested loops and to guarantee convergence when there are multiple self-loops in the same control location (see the concept of *flat systems* in [5]). The main advantages of abstract acceleration *in comparison with widening* result from two properties:

- *Idempotency* ($\tau^\otimes(X) = \tau^\otimes(\tau^\otimes(X))$), which simplifies the fixed point computation (widening usually requires more than one step to stabilize);
- *Monotonicity* $X_1 \sqsubseteq X_2 \Rightarrow \tau^\otimes(X_1) \sqsubseteq \tau^\otimes(X_2)$, that makes the analysis more robust and predictible (whereas widening operators are not monotonic).

2.3 Classical application of abstract acceleration

We describe now the classical way to apply abstract acceleration to the analysis of logico-numerical programs, for which this paper proposes major enhancements.

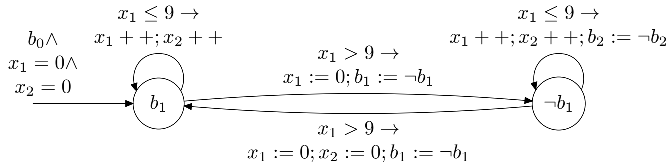
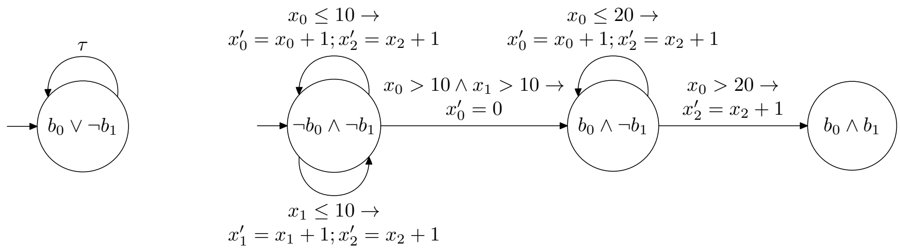


Fig. 3: Self-loop ready to be accelerated (left). Acceleration not applicable (right).



(a) Initial CFG

(b) CFG after Boolean enumeration and removal of the Boolean inputs. (Identity transition functions are implicit.)

Fig. 4: Transformation of the program of Example 1. τ is the global transition. The guards are already convex in the obtained CFG.

Example 1. We will try to infer invariants on the following running example:
$\mathcal{I}(\boldsymbol{b}, \boldsymbol{x}) = \neg b_0 \wedge \neg b_1 \wedge x_0 = 0 \wedge x_1 = 0 \wedge x_2 = 0$

$$
\mathit{true} \to \begin{cases}
b_0' = b_0 \quad \vee \quad (\neg b_0 \wedge x_0 > 10 \wedge x_1 > 10) \\
b_1' = b_1 \quad \vee \quad (\neg b_1 \wedge x_0 > 20) \\
x_0' = \begin{cases} x_0 + 1 & \text{if } (\neg b_0 \wedge \neg b_1 \wedge x_0 \leq 10 \wedge \beta) \quad \vee \quad (b_0 \wedge \neg b_1 \wedge x_0 \leq 20) \\ 0 & \text{if } \neg b_0 \wedge \neg b_1 \wedge x_0 > 10 \wedge x_1 > 10 \\ x_0 & \text{otherwise} \end{cases} \\
x_1' = \begin{cases} x_1 + 1 & \text{if } \neg b_0 \wedge \neg b_1 \wedge x_1 \leq 10 \wedge \neg\beta \\ x_1 & \text{otherwise} \end{cases} \\
x_2' = \begin{cases} x_2 + 1 & \text{if } (\neg b_0 \wedge \neg b_1 \wedge (x_0 \leq 10 \wedge \beta \vee x_1 \leq 10 \wedge \neg\beta)) \quad \vee \quad (b_0 \wedge \neg b_1) \\ x_2 & \text{otherwise} \end{cases}
\end{cases}
$$

The counting patterns of this example (see Fig. 4b) are representative of the production line benchmarks presented in Section 5.

Numerical acceleration can be applied to self-loops where the numerical state evolves while the Boolean state does not, *i.e.* the Boolean part of the transition function is the identity (see Fig. 3 for an example and a counterexample):

Definition 2 (Accelerable logico-numerical transition). *A transition τ is accelerable if it has the form* $g^b(\boldsymbol{b}, \boldsymbol{\beta}) \wedge g^x(\mathcal{C}) \to \begin{pmatrix} \boldsymbol{b}' \\ \boldsymbol{x}' \end{pmatrix} = \begin{pmatrix} \boldsymbol{b} \\ \boldsymbol{a}(\boldsymbol{x}, \boldsymbol{\xi}) \end{pmatrix}$, *where* $g^x(\mathcal{C}) \to \boldsymbol{x}' = \boldsymbol{a}(\boldsymbol{x}, \boldsymbol{\xi})$ *is accelerable.*

Generating a numerical CFG. At first, one performs a Boolean reachability analysis in order to reduce the state space of interest ($b_0 \vee \neg b_1$ in the case of our running example). Starting from the most simple CFG of the program consisting of a single location with a self-loop (see Fig. 4a), standard techniques are used for (1) *enumerating* the Boolean state space and (2) *simplifying the transitions* by source and destination location using partial evaluation. Afterwards, (3) the *Boolean input variables* are replaced by explicit non-deterministic transitions (see Fig. 4b). This CFG is purely numerical, but the guards of the loop transitions might still be non-convex. Transforming the guard into a minimal DNF and splitting the transition into several transitions, one for each conjunct, yields a CFG with self-loops compatible with the transition scheme of §2.2. A single self-loop like in location $b_0 \wedge \neg b_1$ in Fig. 4b can now be "flattened" into a transitive closure transition (cf. Fig. 2).

Multiple self-loops. However, the obtained CFG usually contains multiple self-loops like in location $\neg b_0 \wedge \neg b_1$ in Fig. 4b. In this case a simple "flattening" as in Fig. 2 is not possible: For the fixed point computation we must take into account all sequences of self-loop transitions in this location. Actually, the idempotency of accelerated transitions can be exploited in order reduce these sequences to those where the same transition is never taken twice successively: For the two accelerable loops we have to compute:
$\tau_1^{\otimes}(X) \sqcup \tau_2^{\otimes}(X) \sqcup \tau_2^{\otimes} \circ \tau_1^{\otimes}(X) \sqcup \tau_1^{\otimes} \circ \tau_2^{\otimes}(X) \sqcup \tau_1^{\otimes} \circ \tau_2^{\otimes} \circ \tau_1^{\otimes}(X) \sqcup \tau_2^{\otimes} \circ \tau_1^{\otimes} \circ \tau_2^{\otimes}(X) \sqcup \ldots$
This infinite sequence may not converge, thus in general, widening is necessary to guarantee termination. However, in practice the sequence often converges after the first few elements (see [5]).

The technique implemented in Aspic consists in expanding multiple self-loops into a graph of which the paths represent these sequences, as shown in Fig. 5 in the case of three self-loops, and to solve iteratively the fixed point equations induced by the CFG as sketched in §2.1, using widening if necessary. Moreover, Aspic implements methods to accelerate circuits of length greater than one.

3 Logico-Numerical Abstract Acceleration

Our goal is to exploit abstract acceleration techniques *without resorting to a Boolean state space enumeration* in order to overcome the limitations of current tools (e.g. [12]) w.r.t. the analysis of logico-numerical programs.

In this section, we will first discuss some related issues in order to motivate our approach, before presenting methods that make abstract acceleration applicable to a CFG that now may contain loops with operations on both, Boolean and numerical, variables.

3.1 Motivations for Our Approach

A first observation is that identifying self-loops is more complex when Boolean state variables are not fully encoded in the CFG. Indeed, if a CFG contains a *"syntactic"* self-loop (ℓ, τ, ℓ) with $\tau : g(\boldsymbol{b}, \boldsymbol{x}, \boldsymbol{\xi}) \rightarrow (\boldsymbol{b}', \boldsymbol{x}') = \boldsymbol{f}(\boldsymbol{b}, \boldsymbol{x}, \boldsymbol{\xi})$, there is an *"effective"* self-loop only for those Boolean states $\boldsymbol{b} \in \varphi_\ell$ in location ℓ such

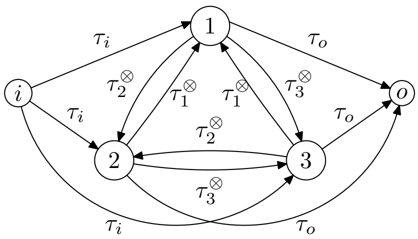


Fig. 5: Computation of three accelerable self-loops τ_1, τ_2 and τ_3. τ_i and τ_o are the incoming resp. outgoing transitions of the location.

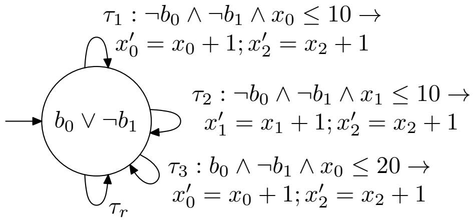


Fig. 6: Acceleration of Ex. 1 in a CFG with a single location: The upper three self-loops are accelerable. The rest of the system is summarized in the transition τ_r where the Boolean equations are not the identity.

that $g(\boldsymbol{b}, \boldsymbol{x}, \boldsymbol{\xi}) \wedge \boldsymbol{b} = \boldsymbol{f}^b(\boldsymbol{b}, \boldsymbol{x}, \boldsymbol{\xi})$ is satisfiable[1]. For instance, the self-loop around location $\neg b_1$ in Fig. 3 is not an "effective" self-loop.

This observation also applies to circuits, where, moreover, *numerical inputs have to be duplicated*: If there is a circuit (ℓ, τ_1, ℓ') and (ℓ', τ_2, ℓ) with $\tau_i : g_i(\boldsymbol{s}, \boldsymbol{\xi}) \rightarrow \boldsymbol{s}' = \boldsymbol{f_i}(\boldsymbol{s}, \boldsymbol{\xi})$ for $i = 1, 2$, the composed transition has the form $\tau : g(\boldsymbol{s}, \boldsymbol{\xi}, \boldsymbol{\xi}') \rightarrow \boldsymbol{s}'' = \boldsymbol{f}(\boldsymbol{s}, \boldsymbol{\xi}, \boldsymbol{\xi}')$. This limits in practice the length of circuits that can be accelerated. Here, we will not deal with such circuits and we focus on self-loops.

A naive approach to our problem could be to partition the system into sufficiently many locations, until we get self-loops that correspond to Def. 2. This approach is simple-minded for two reasons: (i) There might be no such Boolean states in the program at all; (ii) in the case of Fig. 3, simply ignoring the Boolean variable b_2 would make the (syntactic) self-loop accelerable without impacting the precision. More generally, it may pay off to slightly abstract the behavior of self-loops in order to benefit from precise acceleration techniques rather than relying on widening which may lose much more information in the end.

Another important remark is that we do not necessarily need to partition the system into locations to apply acceleration: it is sufficient to decompose the self-loops: Starting from the basic CFG with a single location and a single self-loop, we could split the loop into self-loops where the numerical transition function can be accelerated and the Boolean transition is the identity and a last self-loop where this is not the case. Fig. 6 shows the result of the application of this idea to our running example of Fig. 4.

This allows us to *separate the issue of accelerating self-loops in a symbolic CFG*, addressed in this section, *from the issue of finding a suitable CFG*, addressed in §4.

[1] We assume here that Boolean inputs β have been encoded by non-determinism.

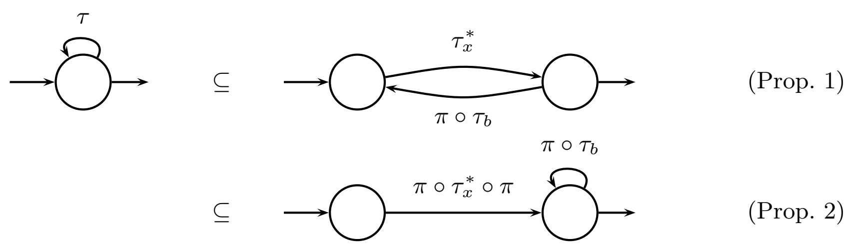


Fig. 7: Decoupling numerical and Boolean transitions

3.2 Decoupling Numerical and Boolean Transition Functions

We consider self-loops (ℓ, τ, ℓ) with $\tau : g(\boldsymbol{s}, \boldsymbol{i}) \rightarrow \begin{pmatrix} \boldsymbol{b}' \\ \boldsymbol{x}' \end{pmatrix} = \begin{pmatrix} \boldsymbol{f}^b(\boldsymbol{s}, \boldsymbol{i}) \\ \boldsymbol{f}^x(\boldsymbol{s}, \boldsymbol{i}) \end{pmatrix}$ without any restriction on $\boldsymbol{f}^b$. We use the abstraction $\wp(E) = \wp(\mathbb{B}^m \times \mathbb{R}^n) \underset{\pi}{\overset{id}{\leftrightarrows}} \wp(\mathbb{B}^m) \times \wp(\mathbb{R}^n)$, where π is the function that approximates a set $S \in E$ by a Cartesian product, *e.g.* $\pi((B_1 \times X_1) \cup (B_2 \times X_2)) = (B_1 \cup B_2) \times (X_1 \cup X_2)$. If τ is accelerable in the sense of abstract acceleration, then $\pi \circ \tau^* \subseteq \tau^{\otimes}$.

Our logico-numerical abstract acceleration method relies on *decoupling* the numerical and Boolean parts of the transition function τ with

$$\tau_b : g(\boldsymbol{s}, \boldsymbol{i}) \rightarrow \begin{pmatrix} \boldsymbol{b}' \\ \boldsymbol{x}' \end{pmatrix} = \begin{pmatrix} \boldsymbol{f}^b(\boldsymbol{s}, \boldsymbol{i}) \\ \lambda(\boldsymbol{s}, \boldsymbol{i}).\, \boldsymbol{x} \end{pmatrix} \text{ and } \tau_x : g(\boldsymbol{s}, \boldsymbol{i}) \rightarrow \begin{pmatrix} \boldsymbol{b}' \\ \boldsymbol{x}' \end{pmatrix} = \begin{pmatrix} \lambda(\boldsymbol{s}, \boldsymbol{i}).\, \boldsymbol{b} \\ \boldsymbol{f}^x(\boldsymbol{s}, \boldsymbol{i}) \end{pmatrix}.$$

We can approximate τ^* as follows (Fig. 7):

Proposition 1. $\tau^* \subseteq (\pi \circ \tau_b \circ \tau_x^*)^*$.

Proof. We prove first $\tau \subseteq \pi \circ \tau_b \circ (id \cup \tau_x)$:
Let $S = \{(\boldsymbol{b}, \boldsymbol{x})\}$, then $\tau_b(S) = \{(\boldsymbol{b}', \boldsymbol{x})\}$, $\tau_x(S) = \{(\boldsymbol{b}, \boldsymbol{x}')\}$, and $\tau(S) = \{(\boldsymbol{b}', \boldsymbol{x}')\}$:

$$\begin{array}{lll} \{(\boldsymbol{b}, \boldsymbol{x}), (\boldsymbol{b}, \boldsymbol{x}')\} & \subseteq (id \cup \tau_x)(S) = S' & \\ \Rightarrow \{(\boldsymbol{b}', \boldsymbol{x}), (\boldsymbol{b}'', \boldsymbol{x}')\} & \subseteq \tau^b(S') \quad = S'' & \text{with } \{(\boldsymbol{b}'', \boldsymbol{x}')\} = \tau^b(\{(\boldsymbol{b}, \boldsymbol{x}')\}) \\ \Rightarrow \{(\boldsymbol{b}', \boldsymbol{x}')\} & \subseteq \pi(S'') \quad = S''' & \end{array}$$

The graphical intuition of these steps is depicted in Fig. 8.
We conclude by

$$\begin{array}{ll} \tau & \subseteq \pi \circ \tau_b \circ (id \cup \tau_x) \\ \Rightarrow \tau & \subseteq \pi \circ \tau_b \circ \tau_x^* \qquad \text{(because of } (id \cup \tau_x) \subseteq \tau_x^*) \\ \Rightarrow \tau^* & \subseteq (\pi \circ \tau_b \circ \tau_x^*)^* \end{array}$$

Now, we assume that τ_x is accelerable in the sense of Def. 2, which implies that $g(\boldsymbol{s}, \boldsymbol{i}) = g^b(\boldsymbol{b}, \boldsymbol{\beta}) \wedge g^x(\boldsymbol{x}, \boldsymbol{\xi})$ and $\boldsymbol{f}^x(\boldsymbol{s}, \boldsymbol{i}) = \boldsymbol{a}(\boldsymbol{x}, \boldsymbol{\xi})$. By applying Prop. 1, we obtain that $(\pi \circ \tau_b \circ \tau_x^{\otimes})^*$ is a sound over-approximation of τ^*. However, this formula still involves Kleene iteration and thus widening is required.

But, there exists an alternative in which numerical and Boolean parts are computed in sequence, so that numerical acceleration is applied only once (Fig. 7):

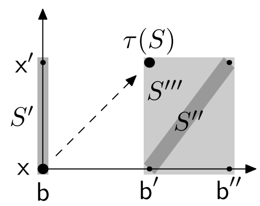


Fig. 8: Illustration of the proof of Prop. 1

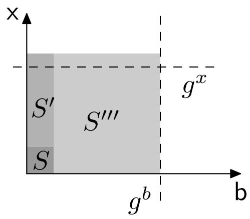


Fig. 9: Illustration of the proof of Prop. 2 ($S \subseteq S' \subseteq S'''$)

Proposition 2 (Decoupling Boolean and numerical transition functions). *If τ_x is accelerable, then*

(1) $(\pi \circ \tau_b)^* \circ \pi \circ \tau_x^* \circ \pi$ *is idempotent, and*
(2) $\tau^* \subseteq (\pi \circ \tau_b)^* \circ \pi \circ \tau_x^* \circ \pi$

Proof. The intuition for (1) is the following: If the guard $g^x \wedge g^b$ is satisfied, *i.e.*, the transition can be taken, then we compute first the transitive closure w.r.t. the numerical states before saturating the Boolean states. The application of τ_b does not enable "more" behavior of the numerical variables; hence, re-applying the τ_x^* has no effect.
We first compute $(\pi \circ \tau_b)^* \circ \pi \circ \tau_x^* \circ \pi(S)$:

$$\begin{aligned}
\pi(S) &= B \times X \\
\tau_x^* \circ \pi(S) &= (B \times X) \cup \big((B \cap (\exists \boldsymbol{\beta} : g^b)) \times X'\big) \\
&\quad \text{with } X' \text{ s.t. } (\exists \boldsymbol{\xi} : \boldsymbol{a}((X \cup X') \cap g^x)) \subseteq X' \quad \text{(i)} \\
S' = \pi \circ \tau_x^* \circ \pi(S) &= B \times (X \cup X') \\
(\pi \circ \tau_b)^* \circ \pi \circ \tau_x^* \circ \pi(S) &= S' \cup \overbrace{B' \times ((X \cup X') \cap (\exists \boldsymbol{\xi} : g^x))}^{S''} \\
&\quad \text{where } B' = \textstyle\bigcup_{k \geq 1} \tau_b^k(B, (X \cup X') \cap (\exists \boldsymbol{\xi} : g^x)) \\
&\quad \text{and with the property } (\pi \circ \tau_b)(S' \cup S'') \subseteq S'' \quad \text{(ii)} \\
S''' = (\pi \circ \tau_b)^* \circ \pi \circ \tau_x^* \circ \pi(S) &= (B \cup B') \times (X \cup X')
\end{aligned}$$

Fig. 9 illustrates the sets S, S', and S'''. S''' is obviously stable by application of π. We show that it is also stable by application of τ_x and $\pi \circ \tau_b$, which allows to conclude that $(\pi \circ \tau_b)^* \circ \pi \circ \tau_x^* \circ \pi(S''') = S'''$, hence the idempotency of the function:

$$\begin{aligned}
\tau_x(S''') &= ((B \cup B') \cap (\exists \boldsymbol{\beta} : g^b)) \times X'' \\
&\quad \text{with } X'' \subseteq X' \text{ because of property (i) above, hence} \\
\tau_x(S''') &\subseteq S''', \text{ and} \\
\pi \circ \tau_x^*(S''') &= S''' \\
\pi \circ \tau_b(S''') &= \pi \circ \tau_b(S''' \cap (\mathbb{B}^m \times (\exists \boldsymbol{\xi} : g^x))) \\
&= \pi \circ \tau_b((B \cup B') \times ((X \cup X') \cap (\exists \boldsymbol{\xi} : g^x))) \\
&\subseteq \pi \circ \tau_b(S' \cup S'') \\
&\subseteq S''' \text{ according to property (ii).}
\end{aligned}$$

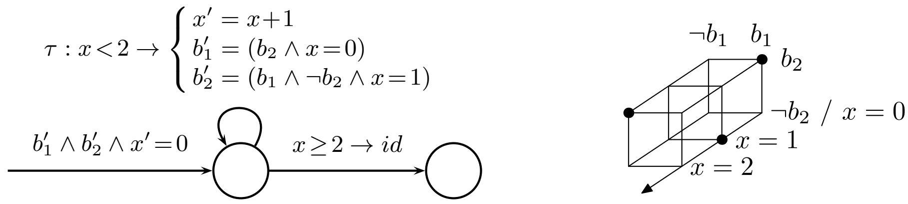


Fig. 10: CFG of a counterexample (left-hand side) to show why the Boolean iterations cannot be computed exactly: the state $(\mathit{false}, \mathit{true}, 2)$ contained in $\tau^*(\{(\mathit{true}, \mathit{true}, 0)\}) = \{(\mathit{true}, \mathit{true}, 0), (\mathit{true}, \mathit{false}, 1), (\mathit{false}, \mathit{true}, 2)\}$ (dots in the right-hand side figure) is not part of $\pi \circ \tau_b^* \circ \pi \circ \tau_x^* \circ \pi(\{(\mathit{true}, \mathit{true}, 0)\}) = \{(\mathit{true}, \mathit{true}), (\mathit{true}, \mathit{false}), (\mathit{false}, \mathit{false})\} \times \{0, 1, 2\}$.

Now, we can prove (2): from Prop. 1 follows

$$\begin{aligned} \tau^* &\subseteq (\pi \circ \tau_b \circ \tau_x^*)^* \\ &= ((\pi \circ \tau_b)^* \circ \tau_x^*)^* \\ &\subseteq ((\pi \circ \tau_b)^* \circ \pi \circ \tau_x^* \circ \pi)^* \\ &= (\pi \circ \tau_b)^* \circ \pi \circ \tau_x^* \circ \pi. \end{aligned}$$

For the last step, we use the idempotency of the function and the fact that it includes the identity. □

Remark 1 (Why not τ_b^*?). We cannot compute Boolean iterations exactly using τ_b^* instead of $(\pi \circ \tau_b)^*$. Fig. 10 gives a counterexample where $\tau^* \not\subseteq \pi \circ \tau_b^* \circ \pi \circ \tau_x^* \circ \pi$, which shows that using exact iterations τ_b^* would not give a sound decoupling.

The following theorem implements Prop. 2 in the logico-numerical product domain A with the Galois connection $\wp(\mathbb{B}^m \times \mathbb{R}^n) \underset{\alpha}{\overset{id}{\leftrightarrows}} \wp(\mathbb{B}^m) \times \mathit{Pol}(\mathbb{R}^n)$.

Theorem 1 (Logico-numerical abstract acceleration). *If a transition τ is such that τ_x is accelerable, then τ^* can be approximated in the logico-numerical product domain A with*

$$\tau^{\otimes}(B, X) = \left(\left(\tau_b^b\left[X^{\otimes}\right]\right)^*(B)\ ,\ X^{\otimes}\right)$$

where
- $X^{\otimes} = (\tau_x^x)^{\otimes}(X)$
- $(\tau_x^x)^{\otimes}$ *is the abstract acceleration of* $\tau_x^x : g^x(\boldsymbol{x}, \boldsymbol{\xi}) \rightarrow \boldsymbol{x}' = \boldsymbol{a}(\boldsymbol{x}, \boldsymbol{\xi})$
- $\tau_b^b[X](B) = \tau_b(B, X^{\otimes} \sqcap^g (\exists \boldsymbol{\xi} : g^x))$
- $(\tau_b^b[X])^*(B) = \mathit{lfp}(\lambda B' . B \cup \tau_b^b[X](B'))$.

$(\tau_b^b[X])^*$ *converges in a finite number of iterations as it is the least fixed point of a monotonic function in the finite lattice* $\wp(\mathbb{B}^m)$.

In other words, we compute the reflexive and transitive closure $X^{\otimes}$ of τ_x using numerical abstract acceleration and saturate τ_b partially evaluated over $X^{\otimes}$.

Proof. We prove that $\tau^{\otimes}(S)$ over-approximates the result of the formula of Prop. 2, *i.e.* $((\pi \circ \tau_b)^* \circ \pi \circ \tau_x^* \circ \pi)(S) \subseteq \tau^{\otimes} \circ \alpha(S)$ with $S \subseteq \wp(\mathbb{B}^m \times \mathbb{R}^n)$. This over-approximation is due to the convex approximations by the numerical abstract domain $Pol(\mathbb{R}^n)$.

$$\begin{array}{l}
(\pi \circ \tau_b)^* \circ \pi \circ \tau_x^* \circ \pi(S) \\
= (\bigcup_{k \geq 0} \tau_b^k(B, (X \cup X') \cap (\exists \boldsymbol{\xi} : g^x)) \ , \ X \cup X') \\
\hfill \text{with } \pi(S) = (B, X) \\
\hfill \text{and } X' \text{ s.t. } (\exists \boldsymbol{\xi} : \boldsymbol{a}((X \cup X') \cap g^x)) \subseteq X' \text{ (see proof of Prop. 2)} \\
\subseteq \lambda(B, X). \left(\bigcup_{k \geq 0} \tau_b^k(B, X^{\otimes} \sqcap^g (\exists \boldsymbol{\xi} : g^x)) \ , \ X^{\otimes} \right) \circ \alpha(S) \\
\hfill \text{with } \tau_x^x = \lambda X. \exists \boldsymbol{\xi} : \boldsymbol{a}(X \cap g^x) \\
\hfill \text{due to the soundness of numerical abstract acceleration:} \\
\hfill (X \cup X') \subseteq (\tau_x^x)^{\otimes} \circ \alpha(X) = X^{\otimes} \\
= \lambda(B, X).((\tau_b^b[X^{\otimes}](B))^* \ , \ X^{\otimes}) \circ \alpha(S) \\
\hfill \text{with the notation } \tau_b^b[X](B) = \tau_b(B, X \sqcap^g (\exists \boldsymbol{\xi} : g^x))
\end{array}$$

Then, by $\tau^*(S) \subseteq ((\pi \circ \tau_b)^* \circ \pi \circ \tau_x^* \circ \pi)(S)$ (Prop. 2), we conclude $\tau^*(S) \subseteq \tau^{\otimes} \circ \alpha(S)$. □

Mind that, due to the convex approximation of the numerical sets, $\tau^{\otimes}$ is not idempotent in general (cf. [13]).

3.3 Discussion

At the first glance, the approximations induced by this partial decoupling seem to be rather coarse. However, this is not severe in our context for two reasons:

1. The relations between Boolean and numerical variables that are lost by our method are mostly not representable in the abstract domain A anyway. For example, consider the loop $x \leq 4 \rightarrow (b' = \neg b; x' = x+1)$, where b could be the least significant bit of a binary counter for instance: starting from $(b, x) \in \{(\mathit{true}, 0)\}$ the exact reachable set is $\{\mathit{true}\} \times \{0, 2, 4\} \cup \{\mathit{false}\} \times \{1, 3, 5\}$; its abstraction in A is $\{\top\} \times \{0 \leq x \leq 5\}$. Hence, these relations will also be lost in a standard analysis merely relying on widening. Yet, due to numerical acceleration, we can even expect a better precision with our method.
2. We will apply this method to CFGs (see §4) in which the Boolean states defining a location exhibit the same numerical behavior and thus, decoupling is supposed not to seriously affect the precision.

Until now we studied the case of a single self-loop. In the presence of multiple self-loops, we expand the graph in the same way as with purely numerical transitions, *e.g.*, as shown in Fig. 5, and we apply Thm. 1 to each loop. As in the purely numerical case, widening must be applied in order to guarantee convergence.

Example 2. We give the results obtained for Ex. 1: Analyzing the enumerated CFG in Fig. 4b using abstract acceleration gives $0 \leq x_0 \leq 21 \wedge \ 0 \leq x_1 \leq 11 \wedge$

$x_0 + x_1 \leq x_2 \leq 44$ bounding all variables[2]. Analyzing the system on a CFG *with a single location* using decoupling and abstract acceleration still bounds two variables $(0 \leq x_0 \leq 21 \wedge 0 \leq x_1 \leq 11 \wedge x_0 + x_1 \leq x_2)$, whereas, even on the enumerated CFG a standard analysis does not find any upper bound at all: $0 \leq x_0 \wedge 0 \leq x_1 \wedge x_0 + x_1 \leq x_2$.

3.4 Variants

Decoupling accelerable from non-accelerable and Boolean transition functions. Theorem 1 applies only if the numerical transition functions are accelerable. If this is not the case, we can reuse the idea of Prop. 1, but now by decoupling the *accelerable numerical* functions (marked by the sub/superscript a) from *Boolean and non-accelerable numerical* functions (sub/superscripts b and n respectively):

$$\tau_a : g(\boldsymbol{s},\boldsymbol{i}) \rightarrow \begin{pmatrix} \boldsymbol{b}' \\ \boldsymbol{x}'_n \\ \boldsymbol{x}'_a \end{pmatrix} = \begin{pmatrix} \lambda(\boldsymbol{s},\boldsymbol{i}).\, \boldsymbol{b} \\ \lambda(\boldsymbol{s},\boldsymbol{i}).\, \boldsymbol{x}_n \\ \boldsymbol{a}(\boldsymbol{x},\boldsymbol{\xi}) \end{pmatrix} \;,\; \tau_{n,b} : g(\boldsymbol{s},\boldsymbol{i}) \rightarrow \begin{pmatrix} \boldsymbol{b}' \\ \boldsymbol{x}'_n \\ \boldsymbol{x}'_a \end{pmatrix} = \begin{pmatrix} \boldsymbol{f}^b(\boldsymbol{s},\boldsymbol{i}) \\ \boldsymbol{f}^n(\boldsymbol{s},\boldsymbol{i}) \\ \lambda(\boldsymbol{s},\boldsymbol{i}).\, \boldsymbol{x}_a \end{pmatrix}$$

Proposition 3 (Decoupling accelerable and non-accelerable transitions).

$$\tau^* \subseteq (\pi \circ \tau_{n,b} \circ \tau_a^*)^* \subseteq (\pi \circ \tau_{n,b} \circ \tau_a^{\otimes})^*$$

However, we cannot remove the Kleene iteration as in Prop. 2, because the function τ_a depends on non-accelerated numerical variables updated by $\tau_{n,b}$, and hence, widening is needed.

Using inputization techniques. Inputization (see [14] for instance) is a technique that treats some state variables as input variables. This method is useful to cut dependencies between the state variables, and thus, to remove loops. For example, it can be employed to reduce $\big((\pi \circ \tau_b)^* \circ \pi\big)$ to $\big(\pi \circ \tau'_b \circ \pi\big)$ in Prop. 2, where τ'_b is obtained by inputizing in τ_b those Boolean state variables that have a transition function that is neither the identity nor constant.

Example 3 (Inputization). The loop τ_b can be approximated by the transition τ'_b where β_0 and β_2 correspond to b_0 and b_2 manipulated as Boolean inputs:

$$\tau_b : \left| \begin{array}{l} b'_0 = \neg b_0 \\ b'_1 = b_1 \\ b'_2 = b_2 \wedge x \geq 0 \end{array} \right. \qquad \tau'_b : \left| \begin{array}{l} b'_0 = \beta_0 \\ b'_1 = b_1 \\ b'_2 = \beta_2 \wedge x \geq 0 \end{array} \right.$$

In our experiments (§5) we observed that the speed-up gained often pays off in comparison to the approximations it brings about.

[2] This is an over-approximated result: the actual polyhedron has more constraints.

4 Partitioning Techniques

The logico-numerical acceleration method described in the previous section can be applied to any CFG. However, in order to make it effective we apply it to a CFG obtained by a partitioning technique that aims at alleviating the impact of decoupling Boolean and numerical transition functions on the precision. This section proposes such partitioning techniques that generate CFGs in which those Boolean states that exhibit the same numerical behavior ("numerical modes") are grouped in the same locations.

Basic technique. In order to implement this idea we generate a CFG that is characterized by the following equivalence relation:

Definition 3 (Numerical modes).

$$\boldsymbol{b}_1 \sim \boldsymbol{b}_2 \Leftrightarrow \begin{cases} \quad \forall \boldsymbol{\beta}_1, \mathcal{C} : \mathcal{A}(\boldsymbol{b}_1, \boldsymbol{\beta}_1, \mathcal{C}) \Rightarrow \\ \qquad\qquad \exists \boldsymbol{\beta}_2 : \mathcal{A}(\boldsymbol{b}_2, \boldsymbol{\beta}_2, \mathcal{C}) \wedge \boldsymbol{f}^x(\boldsymbol{b}_1, \boldsymbol{\beta}_1, \mathcal{C}) = \boldsymbol{f}^x(\boldsymbol{b}_2, \boldsymbol{\beta}_2, \mathcal{C}) \\ \wedge \forall \boldsymbol{\beta}_2, \mathcal{C} : \mathcal{A}(\boldsymbol{b}_2, \boldsymbol{\beta}_2, \mathcal{C}) \Rightarrow \\ \qquad\qquad \exists \boldsymbol{\beta}_1 : \mathcal{A}(\boldsymbol{b}_1, \boldsymbol{\beta}_1, \mathcal{C}) \wedge \boldsymbol{f}^x(\boldsymbol{b}_1, \boldsymbol{\beta}_1, \mathcal{C}) = \boldsymbol{f}^x(\boldsymbol{b}_2, \boldsymbol{\beta}_2, \mathcal{C}) \end{cases}$$

The intuition of this heuristics is to make equivalent the Boolean states that can execute the same set of *numerical actions*, guarded by the same numerical constraints.

Example 4 (Numerical modes). We illustrate the application of this method to Example 1. We first factorize the numerical transition functions by actions:

$$(x_0', x_1', x_2') = \begin{cases} (x_0+1, \ x_1, \ x_2+1) & \text{if } (\neg b_0 \wedge \neg b_1 \wedge x_0 \leq 10) \ \vee \ (b_0 \wedge \neg b_1 \wedge x_0 \leq 20) \\ (x_0, \ x_1+1, \ x_2+1) & \text{if } \neg b_0 \wedge \neg b_1 \wedge x_1 \leq 10 \\ (0, \ x_1, \ x_2) & \text{if } \neg b_0 \wedge \neg b_1 \wedge x_0 > 10 \wedge x_1 > 10 \\ (x_0, \ x_1, \ x_2+1) & \text{if } b_0 \wedge \neg b_1 \wedge x_0 > 20 \\ (x_0, \ x_1, \ x_2) & \text{otherwise} \end{cases}$$

Then by applying Def. 3, we get the equivalence classes $\{\neg b_0 \wedge \neg b_1, b_0 \wedge \neg b_1, b_0 \wedge b_1\}$: the obtained CFG is the one of Fig. 4b.

In the worst case, as in Ex. 4 above, a different set of actions can be executed in each Boolean state, thus the Boolean states will be enumerated. In the other extreme case, in all Boolean states the same set of actions can be executed, which induces a single equivalence class. Both cases are unlikely to occur in larger, real systems.

From an algorithmic point, we vectorize the numerical actions and factorize their common guards, which is equivalent to computing the product of the MtBdds representing the numerical transition functions:

$$\boldsymbol{f}^x(\boldsymbol{b}, \boldsymbol{\beta}, \mathcal{C}, \boldsymbol{x}, \boldsymbol{\xi}) = \bigvee_{1 \leq i \leq m} \big(g_i(\boldsymbol{b}, \boldsymbol{\beta}, \mathcal{C}) \rightarrow \boldsymbol{a}_i(\boldsymbol{x}, \boldsymbol{\xi})\big)$$

Then we eliminate the Boolean inputs β, and we decompose the results as follows

$$(\exists \boldsymbol{\beta} : g_i(\boldsymbol{b}, \boldsymbol{\beta}, \mathcal{C})) = \bigvee_{1 \leq j \leq n_i} g_{ij}^b(\boldsymbol{b}) \wedge g_{ij}^x(\mathcal{C})$$

where $g^x_{ij}(\mathcal{C})$ may be non-convex. The equivalence relation $\sim$ of Def. 3 can be reformulated as

$$\boldsymbol{b}_1 \sim \boldsymbol{b}_2 \Longleftrightarrow \forall i \forall j : g^b_{ij}(\boldsymbol{b}_1) \Leftrightarrow g^b_{ij}(\boldsymbol{b}_2).$$

This last formulation reflects the fact that, in the resulting CFG, the numerical function $\boldsymbol{f}^x$ specialized on a location ℓ does not depend any more on $\boldsymbol{b}$, and hence, the precision loss is supposed to be limited.

Reducing the size of the partition. An option for having a less discriminating equivalence relation is to make equivalent the Boolean states that can execute the same set of numerical actions *regardless of the numerical constraints guarding them.*

Definition 4 (Numerical modes (forgetting numerical guards)).

$$\boldsymbol{b}_1 \approx \boldsymbol{b}_2 \Leftrightarrow \begin{cases} \forall \boldsymbol{\beta}_1, \mathcal{C}_1 : \mathcal{A}(\boldsymbol{b}_1, \boldsymbol{\beta}_1, \mathcal{C}_1) \Rightarrow \\ \qquad \exists \boldsymbol{\beta}_2, \mathcal{C}_2 : \mathcal{A}(\boldsymbol{b}_2, \boldsymbol{\beta}_2, \mathcal{C}_2) \wedge \boldsymbol{f}^x(\boldsymbol{b}_1, \boldsymbol{\beta}_1, \mathcal{C}_1) = \boldsymbol{f}^x(\boldsymbol{b}_2, \boldsymbol{\beta}_2, \mathcal{C}_2) \\ \quad \textit{and vice versa} \end{cases}$$

We clearly have $\sim \subseteq \approx$. For example, if we have two guarded actions $b \wedge x \leq 10 \rightarrow x' = x+1$ and $\neg b \wedge x \leq 20 \rightarrow x' = x+1$, $\sim$ will separate the Boolean states satisfying resp. b and $\neg b$, whereas $\approx$ will keep them together.

Another option is to consider only a subset of the numerical actions, that is, we ignore the transition functions of some numerical variables in Defs. 3 or 4. One can typically focus only on variables involved in the property. According to our experiments, this method is very efficient, but it relies on manual intervention.

Refining the partition by backward bisimulation. Given a partition, it can be refined by *Boolean backward bisimulation* (cf. [15]).

Definition 5 (Boolean backward bisimulation). *Given an equivalence relation $\sim$, its backward Boolean bisimulation equivalence $\sim_\infty$ is defined by*

$$\boldsymbol{b}_1 \sim_0 \boldsymbol{b}_2 \Leftrightarrow \boldsymbol{b}_1 \sim \boldsymbol{b}_2 \wedge (\mathcal{I}(\boldsymbol{b}_1) = \mathcal{I}(\boldsymbol{b}_2))$$

$$\boldsymbol{b}_1 \sim_{k+1} \boldsymbol{b}_2 \Leftrightarrow \begin{cases} \forall \boldsymbol{\beta}_1, \mathcal{C}_1, \boldsymbol{b}'_1 \textit{ such that } \mathcal{A}(\boldsymbol{b}'_1, \boldsymbol{\beta}_1, \mathcal{C}_1) \wedge \boldsymbol{b}_1 = \boldsymbol{f}^b(\boldsymbol{b}'_1, \boldsymbol{\beta}_1, \mathcal{C}_1) : \\ \quad \exists \boldsymbol{b}'_2, \boldsymbol{\beta}_2, \mathcal{C}_2 : \mathcal{A}(\boldsymbol{b}'_2, \boldsymbol{\beta}_2, \mathcal{C}_2) \wedge \boldsymbol{b}_2 = \boldsymbol{f}^b(\boldsymbol{b}'_2, \boldsymbol{\beta}_2, \mathcal{C}_2) \wedge \boldsymbol{b}'_1 \sim_k \boldsymbol{b}'_2 \\ \textit{and vice versa} \end{cases}$$

The rationale behind this refinement is that partitioning the state space with Def. 3 ($\sim$) and stabilizing it by backward bisimulation yields a CFG with locations that group together states that are reachable (in the graph sense) by the same sequence of numerical actions from an initial state.

5 Experimental Evaluation

We have implemented the proposed methods in our experimentation tool ReaVer[3] on the basis of the logico-numerical abstract domain library BddApron.

[3] A first version of the tool which implemented only logico-numerical abstract acceleration methods was called nbACCel [16].

Benchmarks. Besides some small, but difficult benchmarks, we used primarily benchmarks that are simulations of *production lines* (see Fig. 1), as modeled with the library QUEST for the LCM language[4], for evaluating scalability. These models consist of building blocks like sources, buffers, machines, routers for splitting and combining flows of material and sinks, that synchronize via handshakes. The properties we want to prove depend on numerical variables, *e.g.* (1) maximal throughput time of the first element passing the production line, or (2) minimal throughput of the production line. Inputs could model non-deterministic processing and arrival times, but we did not choose benchmarks with numerical inputs in order to enable a comparison with ASPIC [12].

Results. We compared our tool REAVER with NBAC [17, 7, 4] and ASPIC. The results are summarized in Table 1. The tools where launched with the default options; for REAVER we use the partitioning heuristics of Def. 4 and the inputization technique of §3.4. We do not need the decoupling technique of Prop. 3 for our examples.

Discussion. The experimental comparison gives evidence about the advantages of abstract acceleration, but also some potential for future improvement:
– REAVER can prove a lot of examples where NBAC fails: this is due to the fact that abstract acceleration improves precision, especially in nested loops where the innermost loop can be "flattened", which makes it possible to recover more information in descending iterations.
– REAVER seems to scale better than NBAC: First, abstract acceleration reduces the number of iterations and fixed point checks. Second, our heuristics generates a partition that is well-suited for analysis – though, for some of the larger benchmarks, *e.g.* LCM quest 4-1, the dynamic partitioning of NBAC starts to pay off, whereas our static partition is more fine-grained than necessary, which makes us waste time during analysis.
– Once provided with an enumerated CFG, ASPIC is very fast on the smaller benchmarks. However, the current version (3.1) cannot deal with CFGs larger than a few hundred locations. We were surprised that some of the small examples were not proved by ASPIC. We found out that this is due to our fixed point iteration strategy that uses a higher widening delay in strongly connected components resulting from the unfolding of multiple self-loops.
– The analysis using logico-numerical acceleration proved twice as many benchmarks and turned out to be 20% faster than a standard analysis of the same CFG with widening with delay 2 and two descending iterations.
– Applying the more refined partition of Def. 3 to our benchmarks had only a minor influence on performance and precision, and not applying inputization had no impact on the verification of properties, but it slowed down the analysis by 25% on average.
– Generally, for the benchmarks LCM quest 1 to 4, property 2 was not proved by the tools. Here, the combination of our heuristics with dynamic partitioning for further refining the critical parts of the CFG could help.

[4] `http://www.3ds.com`

6 Conclusions

We proposed techniques for accelerating logico-numerical transitions, that allow us to benefit from the precision gain by numerical abstract acceleration as used in the tool Aspic, while tackling the Boolean state space explosion problem encountered when analyzing logico-numerical programs.

Experimentally, our tool ReaVer is often able to prove properties for the larger benchmarks, unlike the two other tools we tested – and this on CFGs that are ten times smaller than the CFGs obtained by enumeration of the reachable Boolean state space.

Although our method is based on the partial decoupling of the Boolean and numerical transitions, the experiments confirm our intuition that our method generally improves the precision. We attribute this to the following observations: first, numerical abstract acceleration reduces the need for widening; second, the information that we might lose by decoupling would often not be captured by the abstract domain anyway; and at last, the CFG obtained by our partitioning method particularly favors the application of our logico-numerical acceleration method.

Perspectives. Regarding abstract acceleration, the acceleration of multiple self-loops deserves additional investigation: We explained that, in case of nested loops, we have to resort to widening in order to guarantee convergence. Multiple self-loops, *i.e.* several self-loops around one location, are a special case of nested loops. Gonnord et al [1, 11] developed abstract acceleration formulas for some special cases of multiple self-loops. In the general case, they apply a graph transformation method based on a partial unfolding (see §2.3) in order to compute more precise fixed points of multiple self-loops. However, this graph transformation is costly since it transforms 1 location with n self-loops into n locations with $n(n-1)$ arcs.

In the context of applying Presburger-based acceleration to program parallelization, Beletska et al. [18] propose an interesting approach which avoids this problem: for multiple self-loops $\tau_i = g_i \rightarrow \boldsymbol{a}_i$, they do not compute $(\bigcup_i \tau_i(X))^*$, but they use the more efficient, though less precise formula: $\bigcup_i \left(\tau_i \left(\bigcup_j \boldsymbol{a}_j(X)\right)^*\right)$. This computation scheme resembles the one of the derivative closure method of Ancourt et al. [19]. This approach could be considered to improve the abstract acceleration of multiple self-loops.

Concerning partition refinement, the combination of our approach with dynamic partitioning *à la* [4] seems to be worth pursuing. In particular, partitioning according to numerical constraints is mandatory for proving properties relying on non-convex invariants – our partitioning techniques partition only the Boolean state space. Such improvements should allow a wider range of benchmarks to be tackled.

Related Work. To our knowledge there is no work about the application of abstract acceleration to logical-numerical data-flow programs, but there is work on related methods that we tailored to fit our purpose. In §2 we already discussed in detail the concepts of abstract acceleration [1, 11], on which our work is based, and that we extended in [6].

Jeannet [4] uses in the tool NBAC partitioning heuristics that are based on the property being analyzed in order to cut paths between initial and bad states. The tool interleaves partitioning steps with analysis (*dynamic partitioning*), thus the "dangerous" state space is reduced in each step. Bouajjani et al. [15] describe a partition refinement algorithm for the LUSTRE compiler using bisimulation. We think that we could exploit it to refine our CFG, when we fail to prove the property.

Alternative approaches for verifying properties about data-flow programs rely on bounded model-checking or k-induction techniques, which both exploit the efficiency of modern SMT solvers. Hagen and Tinelli [20] describe the application of these two approaches to the verification of LUSTRE programs. Another example is the HYSAT tool [21], a bounded model-checker for hybrid systems with piecewise linear behavior – our methods allow to analyze discretizations of such systems. HYSAT relies on the integration of linear constraint solving with SAT solving. The interesting point is that they deal implicitly with large Boolean control structures by encoding them into linear pseudo-Boolean constraints.

References

1. Gonnord, L., Halbwachs, N.: Combining widening and acceleration in linear relation analysis. In: Static Analysis Symposium, SAS'06. Volume 4134 of LNCS. (2006) 144–160
2. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Principles of Programming Languages, POPL'77, ACM Press (1977) 238–252
3. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Principles of Programming Languages, POPL'78, ACM Press (1978) 84–97
4. Jeannet, B.: Dynamic partitioning in linear relation analysis. application to the verification of reactive systems. Formal Methods in System Design **23** (2003) 5–37
5. Bardin, S., Finkel, A., Leroux, J., Petrucci, L.: Fast: acceleration from theory to practice. Software Tools for Technology Transfer **10** (2008) 401–424
6. Schrammel, P., Jeannet, B.: Extending abstract acceleration to data-flow programs with numerical inputs. In: Numerical and Symbolic Abstract Domains, NSAD'10. Volume 267 of ENTCS. (2010) 101–114
7. Jeannet, B.: Partitionnement Dynamique dans l'Analyse de Relations Linéaires et Application à la Vérification de Programmes Synchrones. Thèse de doctorat, Grenoble INP (2000)
8. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: Computer Aided Verification, CAV'97. Volume 1254 of LNCS. (1997) 72–83
9. Coudert, O., Berthet, C., Madre, J.C.: Verification of synchronous sequential machines based on symbolic execution. In: Automatic Verification Methods for Finite State Systems. Volume 407 of LNCS. (1989)
10. Bourdoncle, F.: Efficient chaotic iteration strategies with widenings. In: Formal Methods in Programming and their Applications. Volume 735 of LNCS. (1993) 128–141

11. Gonnord, L.: Accélération abstraite pour l'amélioration de la précision en Analyse des Relations Linéaires. Thèse de doctorat, Université Joseph Fourier, Grenoble (2007)
12. Gonnord, L.: The ASPIC tool: Accelerated symbolic polyhedral invariant computation. `http://laure.gonnord.org/pro/aspic/aspic.html` (2009)
13. Leroux, J., Sutre, G.: Acceleration in convex data-flow analysis. In: Foundations of Software Technology and Theoretical Computer Science. Volume 4855 of LNCS., Springer (2007) 520–531
14. Yannis Bres, Gérard Berry, A.B., Sentovich, E.M.: State abstraction techniques for the verification of reactive circuits. In: Designing Correct Circuits, DCC'02. (2002)
15. Bouajjani, A., Fernandez, J.C., Halbwachs, N.: Minimal model generation. In: Computer Aided Verification, CAV'91. Volume 531 of LNCS. (1991) 197–203
16. Schrammel, P., Jeannet, B.: Logico-numerical abstract acceleration and application to the verification of data-flow programs. In: Static Analysis Symposium. Volume 6887 of LNCS., Springer (2011) 233–248
17. Jeannet, B., Halbwachs, N., Raymond, P.: Dynamic partitioning in analyses of numerical properties. In: Static Analysis Symposium, SAS'99. Volume 1694 of LNCS. (1999)
18. Beletska, A., Barthou, D., Bielecki, W., Cohen, A.: Computing the transitive closure of a union of affine integer tuple relations. In: Combinatorial Optimization and Applications. Volume 6508 of LNCS., Springer (2009) 98–109
19. Ancourt, C., Coelho, F., Irigoin, F.: A modular static analysis approach to affine loop invariants detection. In: Numerical and Symbolic Abstract Domains. Volume 267 of ENTCS., Elsevier (2010) 3–16
20. Hagen, G., Tinelli, C.: Scaling up the formal verification of Lustre programs with SMT-based techniques. In: Formal Methods in Computer-Aided Design, FM-CAD'08, IEEE (2008)
21. Fränzle, M., Herde, C.: Hysat: An efficient proof engine for bounded model checking of hybrid systems. Formal Methods in System Design **30** (2007) 179–198

		Aspic		ReaVer		Nbac	
	vars	size	time	size	time	size	time
Gate 1	4/4/2	7	?	5	**0.73**	24	?
Escalator 1	5/4/2	12	**0.14 (0.04)**	9	0.49	22	?
Traffic 1	4/6/0	18	**0.14 (0.01)**	16	0.19	5	3.49
Traffic 2	4/8/0	18	?	16	**0.35**	28	?
LCM Quest 0a-1	7/2/0	7	**0.04 (0.01)**	5	**0.04**	5	0.05
LCM Quest 0a-2	7/3/0	6	**0.05 (0.01)**	4	**0.05**	8	0.19
LCM Quest 0b-1	10/3/0	19	**0.08 (0.01)**	12	**0.08**	9	?
LCM Quest 0b-2	10/4/0	17	**0.09 (0.01)**	11	0.20	33	?
LCM Quest 0c-1	15/4/0	28	0.17 (0.01)	16	**0.16**	8	0.86
LCM Quest 0c-2	15/5/0	25	**0.20 (0.05)**	14	0.24	50	14.8
LCM Quest 1-1	16/5/0	114	1.99 (0.48)	42	**0.92**	6	2.45
LCM Quest 1-2	16/6/0	100	?	34	?	>156	>
LCM Quest 1b-1	16/5/0	55	0.92 (0.04)	29	**0.37**	15	?
LCM Quest 1b-2	16/5/0	45	0.76 (0.12)	23	**0.47**	61	?
LCM Quest 2-1	17/6/0	247	c	82	**7.84**	9	12.8
LCM Quest 2-2	17/7/0	198	>	62	?	>76	>
LCM Quest 3-1	25/5/0	483	26.5 (14.4)	58	8.49	12	**3.76**
LCM Quest 3-2	25/6/0	481	c	54	?	>1173	>
LCM Quest 3b-1	26/6/0	1724	>	170	43.8	14	**19.1**
LCM Quest 3b-2	26/7/0	1710	>	162	>	>32	>
LCM Quest 3c-1	26/6/0	1319	>	130	**34.2**	9	?
LCM Quest 3c-2	26/7/0	1056	c	98	>	>70	>
LCM Quest 3d-1	26/6/0	281	>	81	**5.43**	49	?
LCM Quest 3d-2	26/7/0	266	c	73	?	446	?
LCM Quest 3e-1	27/7/0	638	>	140	**20.6**	49	?
LCM Quest 3e-2	27/8/0	514	>	110	**6.46**	>28	>
LCM Quest 4-1	27/7/0	4482	>	386	186	9	**50.1**
LCM Quest 4-2	27/8/0	3586	>	290	>	>6	>

vars : Boolean state variables / numerical state variables / Boolean inputs
size : number of locations of the CFG
time : in seconds (Aspic: total time (time for analysis))
? : "don't know" (property not proved)
> : timed out after 600s
c : out of memory or crashed

Table 1: Experimental comparison between Aspic, ReaVer and Nbac.

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399