HAL
open science

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

Peter Schrammel, Bertrand Jeannet

To cite this version:

Peter Schrammel, Bertrand Jeannet. Logico-Numerical Abstract Acceleration and Application to the Verification of Data-Flow Programs. Static Analysis Symposium, Sep 2011, Venice, Italy. pp.233-248, 10.1007/978-3-642-23702-7_19 . hal-00750762

HAL Id: hal-00750762

https://inria.hal.science/hal-00750762

Submitted on 12 Nov 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

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

Peter Schrammel and Bertrand Jeannet

INRIA Rhône-Alpes, Grenoble, France
{peter.schrammel,bertrand.jeannet}@inria.fr

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.

Keywords: Verification, Static Analysis, Abstract Interpretation, Abstract Acceleration, Control Flow Graph Partitioning.

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

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

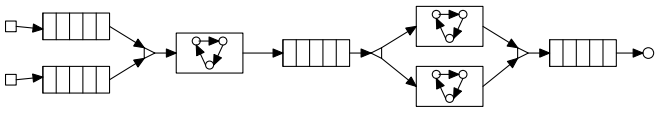


Fig. 1: Example of a production line with buffers, machines, and splitting and combining material flows.

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*. 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 $\mathcal{A}(\boldsymbol{b}, \boldsymbol{\beta}, \mathcal{C}) \rightarrow \begin{pmatrix} \boldsymbol{b}' \\ \boldsymbol{x}' \end{pmatrix} = \begin{pmatrix} \boldsymbol{f}^b(\boldsymbol{b}, \boldsymbol{\beta}, \mathcal{C}) \\ \boldsymbol{f}^x(\boldsymbol{b}, \boldsymbol{\beta}, \mathcal{C}, \boldsymbol{x}, \boldsymbol{\xi}) \end{pmatrix}$.
Numerical transition functions are written as a disjunction of guarded actions: $\boldsymbol{f}^x(\boldsymbol{b}, \boldsymbol{\beta}, \mathcal{C}, \boldsymbol{x}, \boldsymbol{\xi}) = \bigvee_i \big(g_i(\boldsymbol{b}, \boldsymbol{\beta}, \mathcal{C}) \rightarrow \boldsymbol{a}_i^x(\boldsymbol{x}, \boldsymbol{\xi})\big)$ 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 $S = S^0 \cup \mathit{post}(S), S \in \wp(E)$ in an abstract domain and to solve it iteratively, using widening to ensure convergence.

We consider the domain $A = \wp(\mathbb{B}^m) \times \mathit{Pol}(\mathbb{R}^n)$ 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 $\mathit{true} \wedge x \leq 4$.

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

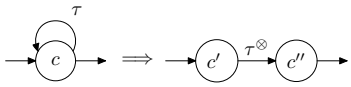


Fig. 2: Self-loop transition (left) and accelerated transition (right).

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*:

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

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

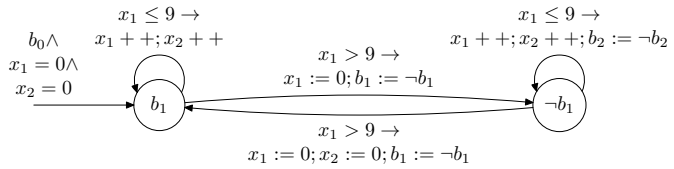


Fig. 3: Self-loop ready to be accelerated (left). Acceleration not applicable (right).

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.

Numerical acceleration can be applied to self-loops where the numerical state evolves while the Boolean state does not: see Fig. 3 for an example and a counterexample. The tool Aspic [12] is based on the enumeration of the Boolean state space which trivially yields a CFG that fulfills this requirement.

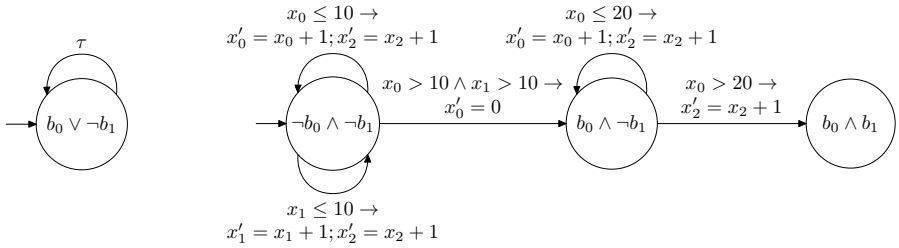
Example 1. We will try to infer invariants on the following running example:

$$\mathcal{I}(\boldsymbol{b}, \boldsymbol{x}) = \neg b_0 \wedge \neg b_1 \wedge x_0 = 0 \wedge x_1 = 0 \wedge x_2 = 0$$

$$\textit{true} \rightarrow \begin{cases} b'_0 = b_0 \ \vee \ (\neg b_0 \wedge x_0 > 10 \wedge x_1 > 10) \\ b'_1 = b_1 \ \vee \ (\neg b_1 \wedge x_0 > 20) \\ x'_0 = \begin{cases} x_0 + 1 & \text{if } (\neg b_0 \wedge \neg b_1 \wedge x_0 \leq 10 \wedge \beta) \ \vee \ (b_0 \wedge \neg b_1 \wedge x_0 \leq 20) \\ 0 & \text{if } \neg b_0 \wedge \neg b_1 \wedge x_0 > 10 \wedge x_1 > 10 \\ x_0 & \text{otherwise} \end{cases} \\ x'_1 = \begin{cases} x_1 + 1 & \text{if } \neg b_0 \wedge \neg b_1 \wedge x_1 \leq 10 \wedge \neg\beta \\ x_1 & \text{otherwise} \end{cases} \\ x'_2 = \begin{cases} x_2 + 1 & \text{if } (\neg b_0 \wedge \neg b_1 \wedge (x_0 \leq 10 \wedge \beta \vee x_1 \leq 10 \wedge \neg\beta)) \ \vee \ (b_0 \wedge \neg b_1) \\ x_2 & \text{otherwise} \end{cases} \end{cases}$$

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

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,



(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.

(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.

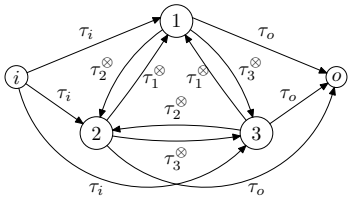


Fig. 5: Computation of three accelerable self-loops τ_1, τ_2 and τ_3. τ_i and τ_o are the incoming resp. outgoing transitions of the location.

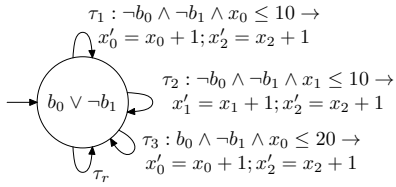


Fig. 6: Acceleration of Ex. 1 in a CFG with a single location: The upper three self-loops are accelerable. The rest of the system is summarized in the transition τ_r where the Boolean equations are not the identity.

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, which 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 symbolic 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 $\boldsymbol{b} \in \varphi_l$ such 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 *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 strongly limits in practice the length of circuits that can be reduced to self-loops and accelerated. In this paper, we will not deal with such circuits, and we consider only self-loops.

We give a definition for a logico-numerical self-loop which can be accelerated by the known methods, because the Boolean part of the transition function is the identity:

Definition 2 (Accelerable logico-numerical transition). *A transition τ is said to be accelerable if it has the form $g^b(\boldsymbol{b}, \boldsymbol{\beta}) \wedge g^x(\mathcal{C}) \rightarrow \begin{pmatrix} \boldsymbol{b}' \\ \boldsymbol{x}' \end{pmatrix} = \begin{pmatrix} \boldsymbol{b} \\ \boldsymbol{a}(\boldsymbol{x}, \boldsymbol{\xi}) \end{pmatrix}$, where $g^x(\mathcal{C}) \rightarrow \boldsymbol{x}' = \boldsymbol{a}(\boldsymbol{x}, \boldsymbol{\xi})$ is accelerable according to §2.2.*

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

[1] We assume here that Bool. inputs β have been encoded by non-determinism, see §2.3.

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 behaviour of self-loops in order to benefit from precise acceleration techniques.

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 loops where the numerical transition function can be accelerated and the Boolean transition is the identity and a last 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. We will use a dedicated partitioning technique to find an appropriate CFG in order to render effective our logico-numerical acceleration method.

3.2 Decoupling numerical and Boolean transition functions

We consider self-loops (ℓ, τ, ℓ) with $\tau : \mathcal{A}(\boldsymbol{s}, \boldsymbol{i}) \to \begin{pmatrix} \boldsymbol{b}' \\ \boldsymbol{x}' \end{pmatrix} = \begin{pmatrix} \boldsymbol{f}^b(\boldsymbol{s}, \boldsymbol{i}) \\ \boldsymbol{f}^x(\boldsymbol{s}, \boldsymbol{i}) \end{pmatrix}$. We use the abstractions $\wp(E) = \wp(\mathbb{B}^m \times \mathbb{R}^n) \underset{\pi}{\overset{id}{\leftrightarrows}} \wp(\mathbb{B}^m) \times \wp(\mathbb{R}^n) \underset{\alpha}{\overset{id}{\leftrightarrows}} A = \wp(\mathbb{B}^m) \times Pol(\mathbb{R}^n)$ discussed in §2.1, 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

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

We can approximate τ^* as follows:

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

See [13] for details of the proof. Briefly, we prove first $\tau \subseteq \pi \circ \tau_b \circ (id \cup \tau_x)$. Then, with $(id \cup \tau_x) \subseteq \tau_x^*$ we conclude $\tau^* \subseteq (\pi \circ \tau_b \circ \tau_x^*)^*$.

Now, we assume that τ_x is accelerable in the sense of Def. 2, which means that $\mathcal{A}(\boldsymbol{s}, \boldsymbol{i}) = g^b(\boldsymbol{b}, \boldsymbol{\beta}) \wedge g^x(\boldsymbol{x}, \boldsymbol{\xi})$ 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 approximation of τ^*. Although we could prove that the involved Kleene iteration is bounded and converges without applying widening, there exists a more efficient alternative in which numerical and Boolean parts are computed in sequence, so that numerical acceleration is applied only once.

Proposition 2. *If τ_x is accelerable, then*

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

See [13] for details of the 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, we saturate the numerical dimensions first; then we saturate the Boolean ones. The point is now, that the application of τ_b does not enable "more" behavior of the numerical variables. Thus, re-applying the function has no effect.
Then we can prove (2): from Prop. 1 follows $\tau^* \subseteq (\pi \circ \tau_b \circ \tau_x^*)^* = ((\pi \circ \tau_b)^* \circ \tau_x^*)^* \subseteq ((\pi \circ \tau_b)^* \circ \pi \circ \tau_x^* \circ \pi)^* = (\pi \circ \tau_b)^* \circ \pi \circ \tau_x^* \circ \pi$ (for the last step, we use (1) and the fact that the function includes the identity).

The following theorem implements Prop. 2 in the abstract domain A. We use the notation $h = f \downarrow X$ ("f partially evaluated on the convex polyhedron X"), to denote any (simpler) formula h such that $X(\boldsymbol{x}, \boldsymbol{\xi}) \Rightarrow (h(\boldsymbol{b}, \boldsymbol{\beta}, \boldsymbol{\mathcal{C}}) = f(\boldsymbol{b}, \boldsymbol{\beta}, \boldsymbol{\mathcal{C}}))$.

Theorem 1. *If a transition τ is such that τ_x is accelerable, then τ^* can be approximated in A with* $\tau^{\otimes} : \quad A \quad \to A$
$$(B, X) \mapsto \left(\left(\tau_b^b \left[X^{\otimes}\right]\right)^* (B) \ , \ X^{\otimes} \right)$$
where
- $X^{\otimes} = (\tau_x^x)^{\otimes}(X)$
- $(\tau_x^x)^{\otimes}$ *is the abstract acceleration of* $\tau_x^x : g^x(\boldsymbol{x}, \boldsymbol{\xi}) \to \boldsymbol{x}' = \boldsymbol{a}(\boldsymbol{x}, \boldsymbol{\xi})$
- $\tau_b^b[X](B) = \left\{ \left(\boldsymbol{f}^b \downarrow (X \sqcap g^x)\right)(\boldsymbol{b}, \boldsymbol{\beta}, \boldsymbol{\mathcal{C}}) \mid \boldsymbol{b} \in B \wedge g^b(\boldsymbol{b}, \boldsymbol{\beta}) \right\}$
- $(\tau_b^b[X])^*(B) = \mathit{lfp}(\lambda B' \,.\, B \cup \tau_b^b[X](B'))$.

Moreover, $(\tau_b^b[X])^*$ (and thus $\tau^{\otimes}$) can be computed in bounded time as the least fixed point of a monotonic function in the finite lattice $\wp(\mathbb{B}^m)$.

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

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

1. The correlations 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 $x \leq 4 \to (b' = \neg b; x' = x+1)$, 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, this information 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 our running example: 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 standard analysis does not find any upper bound at all: $0 \leq x_0 \wedge 0 \leq x_1 \wedge x_0 + x_1 \leq x_2$.

3.3 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 from Boolean and non-accelerable numerical functions:

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

Proposition 3. $\tau^* \subseteq (\pi \circ \tau_{n,b} \circ \tau_a^*)^* \subseteq (\pi \circ \tau_{n,b} \circ \tau_a^{\otimes})^*$

However, Prop. 2 does not apply any more, because the function τ_a depends on non-accelerated numerical variables updated by $\tau_{n,b}$. Moreover, widening is required because $\tau_{n,b}^*$ is not guaranteed to converge in a bounded number of iterations.

3.4 Using inputization techniques

Inputization (see [14] for instance) is a technique that treats state variables as input variables. This method is useful to cut dependencies. 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. 1, where τ'_b is computed by inputizing in τ_b the Boolean state variables having a transition function which is neither the identity nor constant.

Example 3. 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:

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

Our experiments show that this technique is quite useful: the speed-up gained by removing loops often pays off in comparison to the approximations it brings about.

4 Partitioning Techniques for Logico-Numerical Acceleration

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

[2] Over-approximated result: the actual polyhedron has more constraints.

CFG obtained by a partitioning technique that aims at alleviating the impact of decoupling on the precision. This section proposes such partitioning techniques that generate CFGs in which the Boolean states that exhibit the same numerical behavior are grouped in the same locations, so that it is likely that the numerical transition functions in loops do not depend on Boolean state variables.

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

Definition 3. *(Boolean states with same set of guarded numerical actions)*

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

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. We illustrate the application of this method to Example 1. We first factorize the numerical transition functions by actions:

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

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 of view, we represent all our functions with BDDs and MTBDDs [15], and we proceed as follows: We factorize the numerical transition functions by the numerical actions (trivial with MTBDDs):

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

Then we eliminate the Boolean inputs β, and we decompose the results into

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

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

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

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 $\boldsymbol{b}$. Hence, the information 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. *(Boolean states with same set of numerical actions)*

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

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.

5 Experimental Evaluation

Our experimentation tool nbACCel implements the proposed methods on the basis of the logico-numerical abstract domain library BddApron [16].

Benchmarks. Besides some small, but difficult benchmarks, we used primarily benchmarks that are simulations of *production lines* as modeled with the library Quest for the LCM language[3] (see Fig. 1) 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 serve modeling nondeterministic 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 nbACCel with NBac [4] and Aspic. The results are summarized in Table 1. The tools where launched with the default options; for nbACCel we use the partitioning heuristics of Def. 4 and the inputization technique of §3.4. We do not need the technique of §3.3 for our examples.

Discussion. The experimental comparison gives evidence about the advantages of abstract acceleration, but also some potential for future improvement:

- nbACCel 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.
- nbACCel seems to scale better than NBac: First, the idempotency of 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.

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

		Aspic		nbACCel		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

(Benchmarks on `http://pop-art.inrialpes.fr/people/schramme/nbaccel/`)

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

- 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 proven by Aspic. We suspect that this is due to some information loss in widening.
- 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.

6 Conclusion and Related Work

We propose 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 analysing logico-numerical programs. Experimentally, our tool nbACCel 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.

This work raises interesting perspectives: Regarding abstract acceleration, the acceleration of multiple self-loops deserves additional investigation in relation with partitioning techniques. 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 inductive invariants. Such improvements should allow to tackle a wider range of benchmarks.

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. [17] 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 [18] describe the application of these two approaches to the verification of Lustre programs. Another example is the HySat tool [19], 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. Schrammel, P., Jeannet, B.: Logico-numerical abstract acceleration and application to the verification of data-flow programs. Technical Report 7630, INRIA (2011)
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. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. on Computers **35** (1986)
16. Jeannet, B.: Bddapron: A logico-numerical abstract domain library. `http://pop-art.inrialpes.fr/~bjeannet/bjeannet-forge/bddapron/` (2009)
17. Bouajjani, A., Fernandez, J.C., Halbwachs, N.: Minimal model generation. In: Computer Aided Verification, CAV'91. Volume 531 of LNCS. (1991) 197–203
18. Hagen, G., Tinelli, C.: Scaling up the formal verification of Lustre programs with SMT-based techniques. In: Formal Methods in Computer-Aided Design, FMCAD'08, IEEE (2008)
19. 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