Convex Invariant Refinement by Control Node Splitting: a Heuristic Approach

Vivien Maisonneuve[1]

CRI, Mathématiques et systèmes
MINES ParisTech
Fontainebleau, France

Abstract

To improve the accuracy of invariants found when analyzing a transition system, we introduce an original rewriting heuristic of control flow graphs, based on a control node splitting algorithm. The transformation preserves the program behaviors, whilst allowing a finer analysis.
We have carried out experiments with PIPS, a source-to-source compiler, and Aspic, an abstract interpretation tool, using 71 test cases published by Gonnord, Gulwani, Halbwachs, Jeannet & al. The number of invariants found by these tools goes up from 28 to 69 for PIPS and from 44 to 62 for Aspic when our heuristics is used as a preprocessing step. The total execution time of PIPS is only marginally increased, going up from 76 to 103 s, thus showing the practical interest of our optimization technique.

Keywords: model checking, transformer, program analysis, CFG restructuring, automatic invariant detection

1 Introduction

The standard state-based model checking problem is to characterize the set of all the states of a transition system, modeling some program. Most of the usual techniques consist in starting from a set of supposed predicates about a particular position in the transition system, and then propagating it to other positions by evaluating the effect of each transition on the predicates.

An alternative approach consists in computing state transformers [20], i.e., transfer functions, instead of state predicates. Each program command, elementary or compound statement or procedure call, is approximated by an affine transformer. Each function is analyzed once and its transformer is

[1] Email: vivien.maisonneuve@cri.mines-paristech.fr.

reused at each call site. Preconditions are then propagated using the transformers. Such an approach is useful in order to obtain a modular analyzer and to limit analysis times. Affine transformers are commonly used, as they offer good compromise between accuracy and analysis complexity. A transformer-based representation of programs is described in Section 2.

When analyzing a program using transformers, several factors can cause loss in precision. An important one is the computation of loop effects. When using affine transformers, another significant inaccuracy cause is the computation of effects of parallel paths between two points of a program, i.e., paths that have the same origin and destination: indeed, computing the convex union of each path transformer is required in order to obtain a convex result (see Section 3). The issue is worsened on systems with multiple, parallel loops on the same program point, since the approximation factors are combined.

We propose to address this issue with control graph restructuration. In Section 4, a general control node splitting algorithm is introduced and we discuss how to use it to refine invariants found by convex analysis. The results of our experiments are shown in Section 5, while related work is discussed in Section 6.

2 Transformer Automata

In this section, we introduce the data structure used to represent transition systems with transformers, we call it *transformer automaton*.

Let Var be a finite set of n typed variables (e.g., integer variables). A valuation is a function mapping each variable to a possible value of its type. The set of valuations on Var is noted Val.

2.1 General Definition

Definition 2.1 A *transformer* T is a relation transition from Val to Val: $T \subseteq \mathrm{Val} \times \mathrm{Val}$.

Intuitively, a transformer represents the possible changes performed by a piece of program on the program variables. Given two valuations $v, v' \in \mathrm{Val}$ and some piece of code represented by the transformer T, the boolean $T(v, v')$ means that the code, called with variables initially equal to v, may result in a memory state where variables are equal to v'.

Let T_1, T_2 be two transformers, we say that T_2 *overapproximates* T_1 (or simply *approximates* T_1) if $T_1 \subset T_2$, i.e. $\forall v, v' \in \mathrm{Val}, T_1(v, v') \Rightarrow T_2(v, v')$.

For instance, consider an integer variable x and the instruction

$$c = \texttt{\{if (}x\texttt{>0) }x\texttt{++;\}}.$$

Then the transformer of c is $T_c = \{(k, k+1) \mid k \in \mathbb{N}^*\}$ while the transformer $T'_c = \{(k, l) \mid k, l \in \mathbb{N}^*, l > k\}$ approximates T_c. We omit here the domain of valuations, representing the function that maps x to k by k.

Definition 2.2 A *transformer automaton* is a triplet $\alpha = (K, k_{\text{ini}}, \text{Trans})$ where

- K is a finite set of control points;
- $k_{\text{ini}} \in K$ is the initial control point;
- Trans is a finite set of *transitions*, i.e. of triplets (k, T, k') with $k, k' \in K$ and T is a transformer.

Remark 2.3 Val is isomorphic to $\mathbb{Z}^n$, which allows to represent a valuation $v \in \text{Val}$ of variables with a vector $X_v \in \mathbb{Z}^n$. The i-th component of X_v, noted $X_v[i]$, represents the value of the i-th variable in the valuation v. Similarly, a transformer is isomorphic to a relation on $\mathbb{Z}^n$.

Example 2.4 Figure 1a shows an example of transformer automaton $\alpha = (K, k_1, \text{Trans})$ where the set of controls is $K = \{k_1, k_2\}$, k_1 is the initial state and $\text{Trans} = \{(k_1, T_{\text{ini}}, k_2), (k2, T_1, k2), (k_2, T_2, k_2)\}$. Code of the corresponding program is given in Figure 1b, the notation (?) representing a boolean, nondeterministic choice.

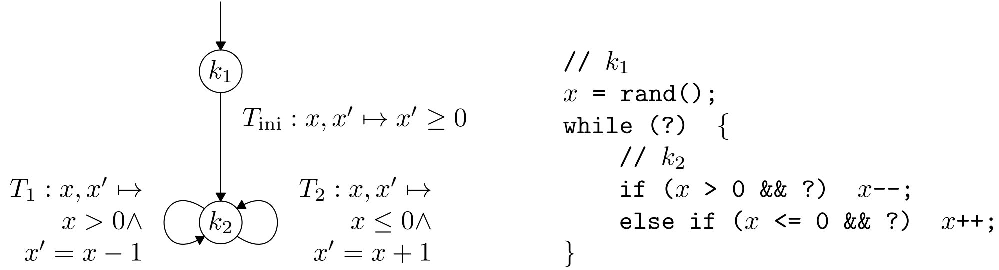

Figure 1. A transformer automaton (a) and the corresponding program code (b)

Semantics

Here is the semantic of transformer automata, in terms of transition systems.

Let $\alpha = (K, k_{\text{ini}}, \text{Trans})$ be a transformer automaton. It is associated with a transition system $(Q, Q_{\text{ini}}, \rightarrow)$ as follows:

- $Q = K \times \text{Val}$: a state is a couple formed by a control point $k \in K$ and a valuation $v \in \text{Val}$.
- $Q_{\text{ini}} = \{k_{\text{ini}}\} \times \text{Val}$: initial states have their first components equal to k_{ini}.
- The transition relation: $(k, v) \rightarrow (k', v')$ if and only if there exists a transition $(k, T, k') \in \text{Trans}$, verifying $T(v, v')$.

Q, Q_{ini} and $\rightarrow$ respectively defines the *states*, the *initial states* and the *state transition relation* of α.

In a transformer automaton α, a *trace* t is a list of states $t = q_0, \ldots, q_m$ such that $q_0 \in Q_{\text{ini}}$ and $\forall i \in [1, m], q_{i-1} \rightarrow q_i$. A state q is *reachable* if there exists a trace t so that q is the last state of t. For instance, in Figure 1a, the state $(k_2, 2)$ is reachable through the trace $(k_1, 0) \rightarrow (k_2, 4) \rightarrow (k_2, 3) \rightarrow (k_2, 2)$, while the state $(k_2, -1)$ is not reachable.

2.2 Affine Case

In this section, we define *affine transformer automata*, a particular class of transformer automata whose transformers are convex polyhedrons.

Definition 2.5 An *affine transformer* T is a finite set of affine inequalities on $2n$ variables, say $x_1, \ldots, x_n, x'_1, \ldots, x'_n$. The associated affine relation $\widetilde{T}$ is the relation verifying

$$\forall X_v, X_{v'} \in \mathbb{Z}^n, \widetilde{T}(X_v, X_{v'}) \text{ iff the valuation } \left\{ \forall i, \begin{array}{c} x_i \mapsto X_v[i] \\ x'_i \mapsto X_{v'}[i] \end{array} \right\} \text{ satisfies } T.$$

By abuse of language, we call *affine transformer* and note T the affine relation $\widetilde{T}$.

Using this latter definition, affine transformers are a particular class of transformers. Note that since the universal transformer $T_\Omega = \text{Val} \times \text{Val}$ is affine, any transformer can be approximated by an affine transformer.

An *affine transformer automaton* is a transformer automaton whose all transformers are affine. For instance, the transformer automaton represented in Figure 1a is affine. Any transformer automaton can be approximated by an affine transformer automaton.

3 Affine Transformer Automaton Analysis

When analyzing a structured program, each program structure is represented by a transformer. Preconditions are then propagated to each control point using the transformers. Unstructured programs can be turned into equivalent, structured programs: see for example [1].

We present here an iterative approach relying on affine transformers, used for example by the program analyzer PIPS [26]. We suppose elementary instructions have been turned into transformers, and just show how control structures are handled.

Sequence

A sequence of affine transformers "T_1 followed by T_2" is overapproximated

by the union of constraints in T_1 (on variables $x_1, \ldots, x_n, x''_1, \ldots, x''_n$) with constraints in T_2 (on variables $x''_1, \ldots, x''_n, x'_1, \ldots, x'_n$), then projected on $x_1, \ldots, x_n$, $x'_1, \ldots, x'_n$ to eliminate the "intermediate" variables $x''_1, \ldots, x''_n$. We note this operation $T_2 \circ T_1$.

Choice

The effect of a choice "T_1 or T_2" is the transformer $T_1 \cup T_2$, which is not affine in the general case (the union of two convex polyhedrons is not a convex polyhedron). The best convex approximation is the convex union $T_1 \sqcup T_2$. This is a lossy operation.

Loop

Given an affine transformer T, an affine transformer T^* representing the effect of any number of iterations of T can be computed, for example with the Affine Derivative Closure algorithm [2] used in PIPS, but other algorithms exist [3,29].

This operation is also cause of inaccuracy. First, because the possible effects of an arbitrary number of iterations of an affine transformer T cannot be encoded as an affine transformer in the general case. For example, a loop whose body is $T : (x, x') \mapsto x' = x + 2$, cannot be associated to a more precise affine transformer than $T^* : (x, x') \mapsto x' \geq x$. Second, because a heuristic to compute the effects of an unbounded number of iterations should perform a transitive closure approximation at a given point.

Example 3.1 With this iterative approach, the structure represented in Figure 2a can be approximated by the single affine transformer $(T_3 \circ T_2^* \circ T_1) \sqcup (T_5 \circ T_4)$.

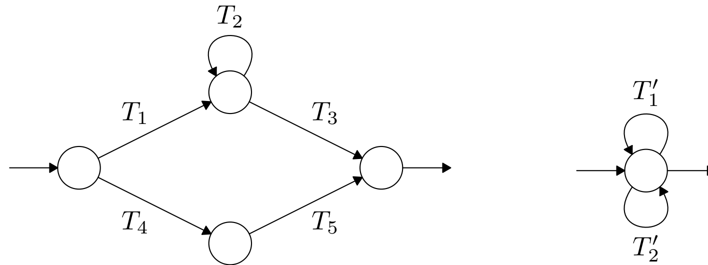

Figure 2. Transformer automaton structures

4 Improving Parallel Loop Handling

The two main sources of imprecision that appeared in Section 3 were the computation of loops ($*$) and parallel paths ($\sqcup$). These imprecisions are accumulated if there are structures in the automaton with two or more loops involved in the same control (Figure 2b, approximated by $(T'_1 \sqcup T'_2)^*$).

4.1 Control Node Splitting Algorithm

If such a structure is met during analysis, there are two ideas to improve accuracy. The first one is to refine transformers involved in loops, so that both the transitive closure approximation and the convex union might be more precise. The second is to reduce the number of parallel loops. Both require a restructuration of the transformer automaton.

The *Control Node Splitting Algorithm* allow us to play on loop transformers and layout.

Control Node Splitting Algorithm

Let $\alpha = (K, k_{\text{ini}}, \text{Trans})$ be a transformer automaton, $k \in K \setminus \{k_{\text{ini}}\}$ a control of α and $\text{Part} = P_1 \uplus \cdots \uplus P_m$ a partition of the domain of valuations Val. The transformer automaton $\alpha_{\text{Part}/k}$ is obtained from α by performing the following steps:

(i) Delete control k.
Add fresh controls $k_{P_1}, \ldots, k_{P_m}$.

(ii) Delete each transition (k, c, k') leaving k $(k' \neq k)$.
For all $i \in [1, m]$, add the transition (k_{P_i}, c_o, k') with $c_o(v, v') = c(v, v') \wedge v \in P_i$.

(iii) Delete each transition (k', c, k) entering in k $(k' \neq k)$.
For all $j \in [1, m]$, add the transition (k', c_i, k_{P_j}) with $c_i(v, v') = c(v, v') \wedge v' \in P_j$.

(iv) Delete each transition (k, c, k) looping in k.
For all $i, j \in [1, m]$, add the transition (k_{P_i}, c_l, k_{P_j}) with $c_l(v, v') = c(v, v') \wedge v \in P_i \wedge v' \in P_j$.

Controls that are not accessible or not coaccessible are not created, neither are the related transitions. Transitions whose transformer is unsatisfiable are not created either.

This algorithm can be applied to any transformer automaton. If run on an affine automaton, and if every partition element P_i of Part is convex, then the resulting automaton $\alpha_{\text{Part}/k}$ is affine.

4.2 Correctness Theorems

We consider a general transformer automaton α and its image $\alpha_{\text{Part}/k}$ obtained through the control node splitting algorithm.

Theorem 4.1 *For all $i \in [1, m]$, for all $v \in Val$, if $q = (k_{P_i}, v)$ is a reachable state of $\alpha_{\text{Part}/k}$ then $v \in P_i$.*

In other words, we have a guarantee that in every control k_{P_i} of $\alpha_{\text{Part}/k}$, the invariant given by P_i holds.

Given two controls k_1 in α, k_2 in $\alpha_{\text{Part}/k}$, we introduce a "state equivalence" relation $\sim_{\text{St.}}$ between states of α and $\alpha_{\text{Part}/k}$, defined by: $(k_1, v) \sim_{\text{St.}} (k_2, v)$ iff

- Either $k_1 = k$ and $k_2 = k_{P_i}$ where $i \in [1, m]$ satisfies: $v \in P_i$.
- Either $k_1 = k_2 \neq k$.

We also define a relation $\sim_{\text{Tr.}}$ between traces of α and $\alpha_{\text{Part}/k}$: $t_1 \sim_{\text{Tr.}} t_2$ iff states in t_1 and t_2 pairwise satisfy $\sim_{\text{St.}}$. Both relations $\sim_{\text{St.}}$ and $\sim_{\text{Tr.}}$ are bijective, so notions of image and inverse image by $\sim_{\text{St.}}$ and $\sim_{\text{Tr.}}$ are defined.

Then the following two theorems hold:

Theorem 4.2 *For all trace t_1 of α, there exists a trace t_2 of $\alpha_{\text{Part}/k}$ such that $t_1 \sim_{\text{Tr.}} t_2$.*

Theorem 4.3 *For all trace t_2 of $\alpha_{\text{Part}/k}$, there exists a trace t_1 of α such that $t_1 \sim_{\text{Tr.}} t_2$.*

These two theorems show that control node splitting preserves reachable states:

Corollary 4.4 *Let q_1 a state of α and q_2 a state in $\alpha_{\text{Part}/k}$ such that $q_1 \sim_{\text{St.}} q_2$. q_1 is reachable in α if and only if q_2 is reachable in $\alpha_{\text{Part}/k}$.*

Equivalence results 4.2, 4.3 and 4.4 give correspondences between automata α and $\alpha_{\text{Part}/k}$ in terms of traces and reachable states. Thus, safety and liveness properties on α can be translated into equivalent properties on $\alpha_{\text{Part}/k}$, and conversely. This allows us to use the automaton $\alpha_{\text{Part}/k}$ instead of α to prove properties on programs, if it turns out to be easier.

4.3 Partition Choice

It is well known that the choice of a control structure affects result accuracy. Thereby, when dealing with a given transformer automaton α, an important question is to determine the control points that should be split as well as the partitions they should be split along, in order to make the analysis more accurate.

We have seen previously that most of the accuracy losses arise from the analysis of control points with parallel loops. These control points are candidates to be split. Concerning the partition choices, a trade-off should be found between different criteria.

First, the automaton structure should be kept as simple as possible, in terms of control and, even more, of transition number. In the general case, partitioning a control k within m components not only adds m control states to the automaton, but also up to m^2 transitions between the newly created controls — from any control to any control; the larger size and more complex structure will increase the analysis complexity. To avoid these issues,

the number m of partition components must be bounded and the partition, carefully chosen so that some of the created states are not reachable or not co-reachable, or that some of the created transitions are not satisfiable, as they will not be present in $\alpha_{\mathrm{Part}/k}$. If possible, the priority is to eliminate transitions involved in loops or in cycles, for the same reasons as above.

Also, the resulting transition transformers should be as precise as possible, especially the ones involved in loops or parallel paths, still with the aim to limit accuracy losses due to approximations.

Choosing the partition can be done manually, considering the system behavior. We propose below an automatic heuristic technique too, which allows to find the expected invariant on 69 small scale test cases published in the related papers (see Section 5).

Guard-Based State Partitioning

Let α be an affine transformer automaton. On every control k with parallel cycles, let $T_1, \dots, T_p$ be the affine transformers of transitions looping on k. We recall that every affine transformer T_i, $i \in [1, p]$ is a set of affine inequalities on variables $x_1, \dots, x_n, x'_1, \dots, x'_n$. For all i, let G_i be the projection of T_i on variables $x_1, \dots, x_n$, i.e. the "guard" of T_i.

Let $\overline{G}_i = \mathrm{Val} \setminus G_i$. As $\overline{G}_i$ is the complementary of a convex polyhedron, it is a polyhedron itself so it can be partitioned into a finite set of convex polyhedrons $\overline{G}_{i,1}, \dots, \overline{G}_{i,j_i}$ [7]. So $P_i = \{G_i, \overline{G}_{i,1}, \dots, \overline{G}_{i,j_i}\}$ is a partition of Val. The partition taken on control point k is:

$$\mathrm{Part}_k = P_1 \otimes \cdots \otimes P_p$$

where $\otimes$ is the "product partition" operation defined by: $\forall E_1, \dots, E_n$, $\forall F_1, \dots, F_m$, $\{E_1, \dots, E_n\} \otimes \{F_1, \dots, F_m\} = \{E_i \cap F_j \mid i \in [1, n], j \in [1, m]\}$.

In a nutshell, controls are created to explicitly allow or disallow each transition. The key idea behind this heuristic is that most of parallel loops have at least partly disjuncts guards. In this case, there are seldom controls with many parallel loops, if at all, and the memory state on these controls is well known (Theorem 4.1). This assumption proved reasonable in most of our test cases. Despite this, the main drawback of this technique is the important number of created controls and transitions, which limits its application field to transition systems whose average number of transitions per control node is limited.

Example 4.5 Let us consider the system in Figure 1a. This system contains two parallel loops on the control k_1, whose transformers are

$$T_1 = (x, x' \mapsto x > 0 \wedge x' = x - 1) \quad \text{and} \quad T_2 = (x, x' \mapsto x \leq 0 \wedge x' = x + 1).$$

Their convex hull is

$$T_1 \sqcup T_2 = (x, x' \mapsto x - 1 \leq x' \leq x + 1)$$

so the invariant on control k_2 is given by $(T_1 \sqcup T_2)^* = T_\Omega$, which prevents us to have any information about the value of the variable x in k_2.

If we project the constraints in T_1 and T_2 on x, removing the variable x', we obtain respectively the guards $\{x > 0\}$ and $\{x \leq 0\}$ which define a partition Part of Val. Using this partition, we split the control k_2 into k'_2, k''_2. Transitions are rewritten accordingly to the algorithm described in Section 4.1. The result is shown on Figure 3. Using this restructured automaton, we are able to check the invariant $(x \geq 0)$ in k_2 (actually, in k'_2 and k''_2).

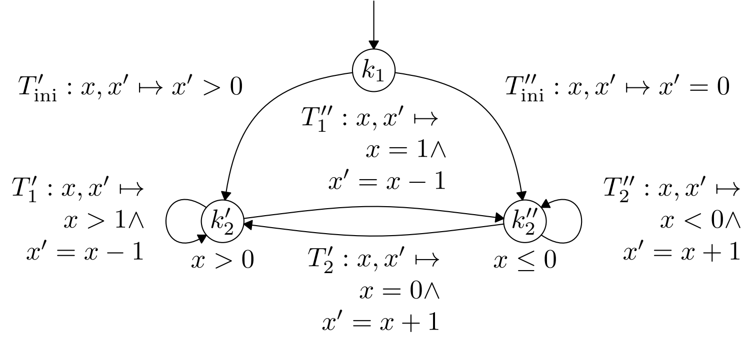

Figure 3. Example of Figure 1a, restructured

5 Experimental Results

The algorithms described in this paper have been tested with the program analyzers PIPS [2] [20,26] and Aspic [3] [9], on a set of 71 previously published test cases [4], most of them taken from references given below in Section 6, a few other from Henzinger & al. [19,4], N. Halbwachs [16,18,17], and some protocol descriptions [25,23,6]. As results depend for a part from the control structure, we decided to reduce bias factors related to this encoding choice by coding every case as a two-control automaton, formed by an initial state leading to a looping state. Still, the algorithm could apply on any type of transition system.

Among these 71 cases, PIPS was able to provide the expected cycle invariants directly for 28. Using our restructuration techniques, we were able to run

[2] Revision 19448.
[3] Version 3.1.
[4] All our test cases are available at https://svn.cri.ensmp.fr/svn/validation/trunk/Semantics-New/NSAD_2011.sub/.

41 additional cases without suffering accuracy loss on the first group of cases. Finally, it failed on two cases, both with and without restructuration.

Aspic was able to compute the expected cycle invariants for 44 out of our 71 test cases using direct encoding. Unsurprisingly, these results are better than those obtained with PIPS since Aspic is a program devoted to polyhedral invariant computation. 21 more test cases are properly analyzed using partitioning techniques, while Aspic still failed on 6 cases.

After restructuration, the resulting systems have typically between 2 and 10 control states, which is in the scope of our analyzers. The subway example [18], more complex, grew much bigger (up to 23 controls) and its analysis, while correct, is very slow compared to what could be achieved with manual control restructuration (10 seconds vs. 0.5 seconds using 5 control states).

As for PIPS execution time, the analysis of the 28 directly working cases took 16.13 seconds on our machine [5] with direct encoding, against 20.47 seconds using control node splitting techniques (27 % slower). Analyzing the whole set of examples took 76.49 seconds when encoded as single state transition systems, against 102.77 seconds with restructuration (34 % slower). Thus, it appears that the increase in time due to restructurations does not necessarily lead to an exponential blowup as feared by Laure Gonnord [10], at least for loops with no more than five cycles.

6 Related Work

Partitioning techniques are a well established bunch of methods to improve the precision of analyses. Gulwani & al. [13,12,14,15] introduce several techniques to compute complexity bounds on procedures, including a semantics transformation on loops called *control-flow refinement*.This technique is not used in the same context as ours: it is specially devoted to bound analysis, is applied to structured programs and not to control graphs, and uses a fundamentally different transformation. Bertrand Jeannet [22,21] proposes a *dynamic partitioning* approach which tackles the problem of the control structure complexification due to partitioning. The main idea is that the final control structure is chosen dynamically, depending on the property to be proved. This does not exactly match our goal, which is to refine control invariants on a system independently of any property proof. Partitioning techniques are also used in abstract interpretation: dynamic partitioning à la Bourdoncle [5], trace partitioning à la Rival-Mauborgne [27], etc.

On the other hand, several techniques improve the computation of loop invariants without any control flow restructuration. The work of Kelly & al.

[5] All experiments were performed on an Intel Core i7 machine at 2.8 GHz running Debian Linux 2.6.32-5 with 8 GB of memory, using PIPS revision 19448.

[24] introduced many concepts and algorithms in the computation of transitive closures. However, they consider a different set of applications and focus mostly on the underapproximation of transitive closures instead of overappromations. Laure Gonnord improves results given by Linear Relation Analysis [8] [16] by identifying categories of loops whose effects can be computed exactly, via *abstract accelerations* [11,10]. Also, Sven Verdoolaege & al. [29] have developed techniques to compute more accurately the transitive closures of a class of parametrized relations that captures Presburger arithmetic and affine transformers on integer variables, with a special focus on the case of parallel paths.

7 Conclusion

We present a simple algorithm to split control nodes to refine a program control flow graph, over a partition of the set of variable valuations (Section 4.1). Theorems are given about the invariants of created control nodes, and to ensure the restructuration does not change the behavior of the program (Section 4.2).

We also give heuristics concerning the choice of partitions in the case of convex transition systems, based on the transition guards (Section 4.3). This approach is tested on a set of previously published examples, yielding encouraging results (69 examples worked out of 71, Section 5).

Future work will address performance issues. The restructuration given by our heuristics tends to create a large number of controls and transitions, which limits its scope to systems whose number of parallel loops is limited. This complexity is useless in several cases because the same invariant accuracy is obtained with much simpler, manually restructured systems. We were also able to design transition systems on which the proposed partition is not suited, leading to the analyzer failure, while a better, simpler partition choice would have worked. As part of future work, we therefore want to design better partition strategies to handle a wider range of transition systems.

References

[1] Zahira Ammarguellat. A control-flow normalization algorithm and its complexity. *IEEE Trans. Software Eng.*, 18(3):237–251, 1992.

[2] Corinne Ancourt, Fabien Coelho, and François Irigoin. A modular static analysis approach to affine loop invariants detection. *Electr. Notes Theor. Comput. Sci.*, 267(1):3–16, 2010.

[3] Anna Beletska, Denis Barthou, Wlodzimierz Bielecki, and Albert Cohen. Computing the transitive closure of a union of affine integer tuple relations. In *COCOA*, pages 98–109, 2009.

[4] Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Path invariants. In *PLDI*, pages 300–309, 2007.

[5] François Bourdoncle. Abstract interpretation by dynamic partitioning. *J. Funct. Program.*, 2(4):407–423, 1992.

[6] Tevfik Bultan, Richard Gerber, and William Pugh. Symbolic model checking of infinite state systems using presburger arithmetic. In *CAV*, pages 400–411, 1997.

[7] Bernard Chazelle. Convex partitions of polyhedra: A lower bound and worst-case optimal algorithm. *SIAM J. Comput.*, 13(3):488–507, 1984.

[8] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96, 1978.

[9] Laure Gonnord. Aspic, 2005–2010.

[10] Laure Gonnord. *Accélération abstraite pour l'amélioration de la précision en Analyse des Relations Linéaires.* PhD thesis, Université Joseph-Fourier - Grenoble I, 10 2007.

[11] Laure Gonnord and Nicolas Halbwachs. Combining widening and acceleration in linear relation analysis. In *SAS*, pages 144–160, 2006.

[12] Sumit Gulwani. Speed: Symbolic complexity bound analysis. In *CAV*, pages 51–62, 2009.

[13] Sumit Gulwani, Sagar Jain, and Eric Koskinen. Control-flow refinement and progress invariants for bound analysis. In *PLDI*, pages 375–385, 2009.

[14] Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. Speed: precise and efficient static estimation of program computational complexity. In *POPL*, pages 127–139, 2009.

[15] Sumit Gulwani and Florian Zuleger. The reachability-bound problem. In *PLDI*, pages 292–304, 2010.

[16] Nicolas Halbwachs. *Détermination automatique de relations linéaires vérifiées par les variables d'un programme.* PhD thesis, Institut National Polytechnique de Grenoble - INPG; Université Joseph-Fourier - Grenoble I, 03 1978. Universités : Université scientifique et médicale de Grenoble et Institut national polytechnique de Grenoble SUDOC-004907809 ; MI2S-tu985.

[17] Nicolas Halbwachs. Linear relation analysis: Principles and recent progress, 12 2010. Presentation at Second French Compiler Research Meeting.

[18] Nicolas Halbwachs, Yann-Erick Proy, and Patrick Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, 1997.

[19] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.

[20] François Irigoin, Pierre Jouvelot, and Rémi Triolet. Semantical interprocedural parallelization: an overview of the PIPS project. In *ICS*, pages 244–251, 1991.

[21] Bertrand Jeannet. *Partitionnement dynamique dans l'analyse de relations linéaires et application à la vérification de programmes synchrones.* PhD thesis, Institut National Polytechnique Grenoble, sep 2000.

[22] Bertrand Jeannet, Nicolas Halbwachs, and Pascal Raymond. Dynamic partitioning in analyses of numerical properties. In *SAS*, pages 39–50, 1999.

[23] Randy H. Katz, Susan J. Eggers, David A. Wood, C. L. Perkins, and R. G. Sheldon. Implementing a cache consistency protocol. In *ISCA*, pages 276–283, 1985.

[24] Wayne Kelly, William Pugh, Evan Rosser, and Tatiana Shpeisman. Transitive closure of infinite graphs and its applications. In *LCPC*, pages 126–140, 1995.

[25] Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.

[26] MINES ParisTech. PIPS, 1989–2011. Open source, under GPLv3.

[27] Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.*, 29(5), 2007.

[28] Peter Schrammel and Bertrand Jeannet. Extending abstract acceleration methods to data-flow programs with numerical inputs. *Electr. Notes Theor. Comput. Sci.*, 267(1):101–114, 2010.

[29] Sven Verdoolaege, Albert Cohen, and Anna Beletska. Transitive Closures of Affine Integer Tuple Relations and their Overapproximations. Research Report RR-7560, INRIA, 03 2011.