ALICe: A Framework to Improve Affine Loop Invariant Computation

Vivien Maisonneuve, Olivier Hermant, François Irigoin

ALICe: A Framework to Improve Affine Loop Invariant Computation*

Vivien Maisonneuve, Olivier Hermant and François Irigoin

MINES ParisTech, France

vivien.maisonneuve@cri.mines-paristech.fr, olivier.hermant@cri.mines-paristech.fr, francois.irigoin@cri.mines-paristech.fr

Abstract

A crucial point in program analysis is the computation of loop invariants. Accurate invariants are required to prove properties on a program but they are difficult to compute. Extensive research has been carried out but, to the best of our knowledge, no benchmark has ever been developed to compare algorithms and tools.

We present ALICe, a toolset to compare automatic computation techniques of affine loop scalar invariants. It comes with a benchmark that we built using 102 test cases which we found in the loop invariant bibliography, and interfaces with three analysis programs, that rely on different techniques: Aspic, ISL and PIPS. Conversion tools are provided to handle format heterogeneity of these programs.

Experimental results show the importance of model coding and the poor performances of PIPS on concurrent loops. To tackle these issues, we use two model restructurations techniques whose correctness is proved in Coq, and discuss the improvements realized.

1 Introduction

The standard state-based model checking problem is to characterize the set of all reachable states of a transition system modeling some program. This information is generally used to check safety properties on the system, ensuring that “bad” configurations cannot be reached. The accuracy of computed invariants is very important, as it plays an essential role in the success of the program analysis.

Dealing with potentially infinite-state models requires to overapproximate invariants into a mathematical model (or *abstract domain*) whose representation is finite and that allows to make the required computations. Many such domains exist in the literature, but we focus on the domain of *affine* invariants, first introduced by N. Halbwachs [13, 25], which offers a good trade-off between invariant accuracy and computational complexity.

Most of the usual analysis techniques consist in starting from a set of supposed predicates about a particular control position in the transition system, and then propagating it to other positions by evaluating the effect of each transition on the predicates. This is pretty straightforward, except in the case of loops that needs special treatment. This lead to intensive research, with many approaches based either on abstract interpretation [32, 16, 38], that is, using widening operations, or on direct computation [2]. The case of concurrent loops, i.e. when there are different possible loops on the same control point, is particularly challenging.

As of today, several tools for linear relation analysis use a vast pattern of algorithms and heuristics to handle loops, aiming to maximum accuracy. We propose a toolset that compares these tools on a common set of small-scale, previously published test cases and to test the sensibility of these tools to different encoding schemes. In Section 2, we introduce our toolset,

*We thank Laure Gonnord and Sven Verdoolaege who helped us to use their tools Aspic and ISL.

ALICe, and present the set of test cases and of analysis programs that we use. The different encodings are discussed in Section 3, after which the results of our experiments are shown.

2 The ALICe Benchmark

The ALICe benchmark project aims to provide tools and a standardized set of test cases to compare as fairly as possible different polyhedral analysis techniques and softwares. ALICe is a free software distributed under GPLv3, and is available at `http://alice.cri.mines-paristech.fr/`.

There are several motivations behind the ALICe project. First, we want to use it as a tool to compare polyhedral invariant computation tools on a common ground, using a set of published test cases issued from various sources, instead of evaluating their performances on *ad hoc* examples only, in the spirit of the TPTP library [44] and the CASC competition [43, 39]. ALICe also gives the opportunity to evaluate the benefits of model-to-model restructurations prior to analysis (Section 3).

2.1 Program Model

Test cases in ALICe are particular interpreted automata called *models*, and traditionally represented as graphs. In a nutshell, a model in ALICe is a transition system constituted by a set of control points (nodes), connected by guarded commands acting on integer variables (edges), and comes with initial and error states. An example of model in shown in Figure 1.

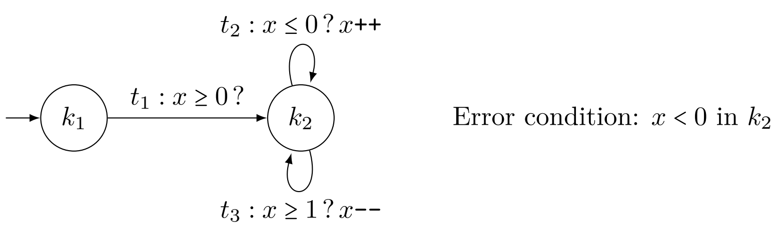


Figure 1: An example of model

Definitions Let X be a finite set of integer variables (boolean variables, if any, are cast to integers) and K be a finite set, whose elements are called *control points*. In our example, $X = \{x\}$ and $K = \{k_1, k_2\}$.

- A *valuation* on X is a function $v : X \to \mathbb{Z}$ mapping each variable to an integer value. Let $V = \mathbb{Z}^X$ be the set of valuations on X.
- A (*global*) *state* of the model is a pair $q = (k, v) \in K \times V$, for instance: $(k_2, 0)$. The set of global states is noted Q.
- A *guard* is a function $g : V \to \mathbb{B} = \{\bot, \top\}$, giving the value of a logic formula on a valuation v. In the example model, the guard in t_2 is:

  $$g_2 : v \mapsto v(x) \le 0, \quad \text{simply noted “}x \le 0\text{”.}$$

  In practice, the guard is assimilated to its truth set $\{v \,|\, g(v) = \top\}$. The set of guards is noted G.

- An *action* a is a binary relation on valuations: $a \subseteq V \times V$ and represent possible valuation changes; it is not necessarily a function. In t_2, the corresponding action is:

$$a_2 = \{(v, v') \in V \times V \,|\, v'(x) = v(x) + 1\}, \quad \text{simply noted "}x\texttt{++}\text{"}$$

  with $X = \{x\}$. The set of actions is noted A.

- A *transition* t is a quadruplet $t = (k, g, a, k') \in K \times G \times A \times K$. Intuitively, when the automaton is at control point k, and if the current valuation v satisfies the guard g (i.e. $g(v) = \top$), the transition execution moves to the control point k', with a valuation v' such that $(v, v') \in a$. The transition $t = (k, g, a, k')$ is usually noted:

$$t : k \xrightarrow{g\,?\,a} k'$$

  Transitions are not necessarily deterministic.

  In the example model, t_2 is a transition from control point k_2 to control point k_2 with guard g_2 and action a_2, that is $t_2 : k_2 \xrightarrow{x \leq 0\,?\,x\texttt{++}} k_2$. The set of transitions is noted T.

Formalism Using these definitions, a *model* can now be formally defined as a tuple $m = (X, K, T, Q_{\text{init}}, Q_{\text{err}})$ where

- X is a finite set of variables;
- K is a finite set of control points;
- T is a finite set of transitions on X and K;
- Q_{init} and Q_{err} are subsets of Q, called respectively *initial states* and *error states* of m.

  In the example model of Figure 1, there is an initial control point k_1 with no valuation constraint so $Q_{\text{init}} = \{k_1\} \times V$. The error region is at control point k_2 with $x < 0$, that is: $Q_{\text{err}} = \{k_2\} \times (x < 0)$.

The set of models is noted $\mathcal{M}$.

Semantics The semantics of a model is defined in terms of transition systems. The model $m = (X, K, T, Q_{\text{init}}, Q_{\text{err}})$ is associated to the transition system $(Q, \to, Q_{\text{init}})$, which transition relation obeys to:

$$(k, v) \to (k', v') \iff \exists (k, g, a, k') \in T, g(v) = \top \wedge (v, v') \in a. \tag{1}$$

Accessibility and Verification A safety property expresses that "something bad never happens". We consider *accessibility properties*, a subset of safety properties, expressing that the set of error states Q_{err} — representing "something bad" — cannot be reached. Their verification involves the computation of accessible states from the initial states Q_{init}.

Let R be a subset of states Q. We note

$$\text{Post}(R) = \{q' \in Q \,|\, \exists q \in R, q \to q'\}$$

the set of successor states of all states in R, and

$$\text{Acc}(R) = \bigcup_{n \geq 0} \text{Post}^n(R)$$

the set of all accessible states from some state in R (Acc is the transitive closure of Post).

The model m is *correct* if and only if:

$$\mathrm{Acc}(Q_{\mathrm{init}}) \cap Q_{\mathrm{err}} = \varnothing. \tag{2}$$

The test cases available in ALICe are all correct models.

Challenging a Tool In general, it is not possible to compute exactly the set of accessible states $\mathrm{Acc}(Q_{\mathrm{init}})$. Instead, analysis tools tested by ALICe compute supersets Q' of $\mathrm{Acc}(Q_{\mathrm{init}})$: if a given Q' does not intersect with Q_{err}, then the tool that generated this Q' has verified that states in Q_{err} are unreachable (Figure 2), thus the test case is considered a success for that tool. However, if the intersection $Q' \cap Q_{\mathrm{err}}$ is not empty, the tool user cannot conclude whether the property is violated or if the overapproximation is too inaccurate: this corresponds to a failure for the tool.

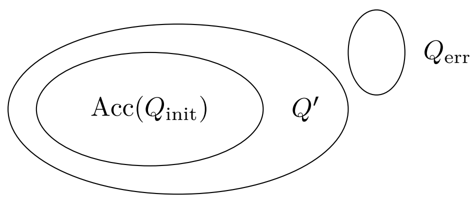


Figure 2: Model checking

Note that we could use *backward analysis* instead: starting from error states Q_{err}, computing iteratively the set of coaccessible states and testing the intersection with Q_{init}. But as all analysis tools used within ALICe rely on forward analysis (see Section 2.3), this is not explored further.

2.2 Test Cases

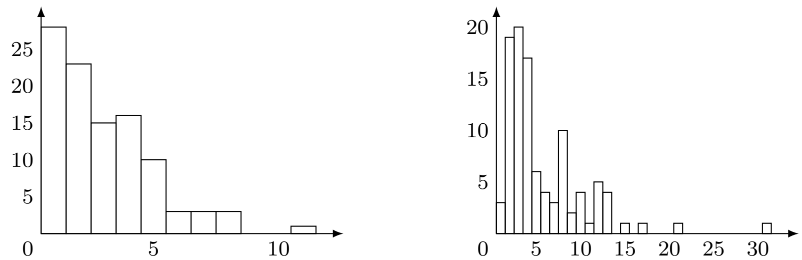
The benchmark itself consists in 102 previously published test cases, including work from L. Gonnord [16, 1], S. Gulwani [8, 21, 22, 23, 24], N. Halbwachs [26, 28, 27, 25], B. Jeannet [32] et al. The comprehensive list of model sources can be found in the bibliography [3, 36, 5, 6, 8, 9, 10, 11, 12, 18, 19, 20, 29, 30, 34, 38, 40, 41], as well as on the ALICe website. They come mostly from works on loop invariant computation, loop bound analysis and, to a lesser extent, protocol verification. Test cases are usually relatively small: typically 1 to 10 states and 2 to 15 transitions. Histograms showing distribution of test cases according to their sizes are shown in Figure 3.

These test cases come in many forms and had to be hand-encoded to a common format, described in Section 2.4.

2.3 Three Supported Tools

For now, ALICe is interfaced with three invariant computation tools: Aspic, ISL and PIPS.

- Aspic [17], a polyhedral invariant generator developed by L. Gonnord. Aspic relies on classic linear relation analysis, improved with *abstract accelerations*, identifying classes of loops in the abstract polyhedral domain whose effect can be computed directly instead of using widening operations, thus granting better accuracy.



(a) Number of control states (b) Number of transitions

Figure 3: Distribution of test cases

- ISL: "the Integer Set Library" [46], developed by S. Verdoolaege. ISL is a library for manipulating sets S and relations R on integer tuples bounded by affine constraints in the following form:

$$S(s) = \left\{x \in \mathbb{Z}^d \,\middle|\, \exists z \in \mathbb{Z}^e : Ax + Bs + Dz \geq c\right\},$$
$$R(s) = \left\{(x_1, x_2) \in \mathbb{Z}^{d_1} \times \mathbb{Z}^{d_2} \,\middle|\, \exists z \in \mathbb{Z}^e : A_1 x_1 + A_2 x_2 + Bs + Dz \geq c\right\}.$$

  These definitions allow greater expressiveness than polyhedral constraints: in terms of logical expressiveness, they are equivalent to Presburger arithmetic constraints.

- PIPS [7], an interprocedural source-to-source compiler framework for C and Fortran programs, initiated at MINES ParisTech, that relies on a polyhedral abstraction of program behavior. Unlike other tools, PIPS performs a two-step analysis. First, the program is abstracted: each program command instruction is associated to an affine transformer representing its underlying transfer function. This is a bottom-up procedure, starting from elementary instructions, then working on compound statements and up to function definitions. Second, polyhedral invariants are propagated along instructions, using transformers previously computed.

We do not consider the Omega+ [42] library, as it appears to be superseded by ISL [45]. It would be very interesting to be able to use more tools, such as FASTer [35, 4] or NBAC [33, 31], but adding support is a time-consuming task, as explained below, and we have not been able to do it up to now.

2.4 Heterogeneity of Tools

ALICe test cases are written in the fsm format. This is a simple language that directly represents models, originally used in FAST [35] and then in Aspic. An example of fsm program for the model in Figure 1 is given in Listing 1.

As we wish to analyze these test cases with different tools and compare results, we have to convert input and output formats. Basically, each tool uses different input and output formats:

- Aspic uses the fsm format both as input and as output;
- ISL uses a custom format to describe both the input model as a relation on states with conditions on variables, and the output invariant, given as a map from states to domains;

```
model m {
    var x;
    states k1, k2;
    transition t1 {
        from := k1;
        to := k2;
        guard := x >= 0;
        action := ;
    }
    transition t2 {
        from := k2;
        to := k2;
        guard := x <= 0;
        action := x' = x + 1;
    }
    transition t3 {
        from := k2;
        to := k2;
        guard := x >= 1;
        action := x' = x - 1;
    }
}
strategy s {
    Region init := {state = k1};
    Region bad := {x < 0};
}
```

Listing 1: Source code in fsm format

- PIPS processes a structured C program according to a script written in a custom scripting language, called tpips, while resulting invariants are given as comments surrounding instructions in the output C code.

To check whether an analyzer works successfully on a test cases, we follow these steps:

1. if necessary, convert the model (originally in fsm) into the analyzer's input format;
2. run the analyzer and get the computed model invariant;
3. if necessary, convert the model invariant into ISL format;
4. use ISL to check whether the intersection of the model error region and the invariant computed by the analyzer is empty, i.e. whether the analyzer is able to solve the test case.

Those steps are illustrated in Figure 4.

Implementing these steps required to develop several translation programs, from and to Aspic, ISL and PIPS formats. In particular, the tools fsm2c and fsm2isl respectively convert a fsm model into C code or ISL relation. There is also an export tool for fsm in dot format that allows visualization, fsm2dot. They are available within ALICe.

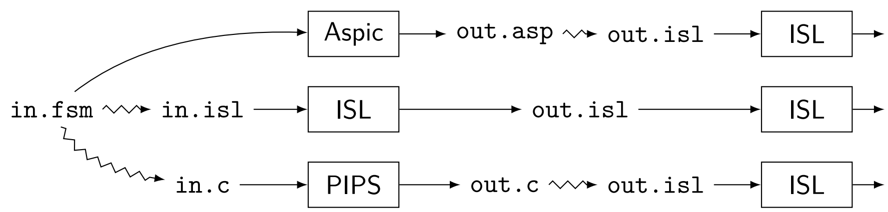


Figure 4: Analysis steps within ALICe

Conversely, it is possible to use ALICe to analyze simple models written in C by first translating them into fsm automata, using the c2fsm utility developed by Paul Feautrier [14, 15].

2.5 Results for the Raw, Hand-Encoded Test Cases

In this section, we present experimental results obtained with ALICe. Benchmarks were run on a computer with a Quad-Core AMD Opteron Processor 2380 at 2.4 GHz and 16 GB of memory, using the following versions of analyzers (latest versions at the time of writing):

- Aspic version 3.1;
- ISL from Barvinok version 0.36;
- PIPS revision 22 105 (April 2014).

Results obtained by these three tools are displayed in Table 1, along with the corresponding execution times. Aspic stands out as the winner in this comparison, with ISL coming second and PIPS, which is not primarily targeted at invariant analysis, being placed last, both in terms of success rate and of execution time.

	Aspic	**ISL**	**PIPS**
Successes	75	63	43
Time (s.)	10.9	35.5	46.2

Table 1: Benchmark results

This ranking is lessened by by the fact that no tool is *strictly better* than another: for each tool, there is at least one model that is successfully analyzed only by this tool, as shown in Figure 5; and it appears in Table 2 that there is no clear trend as for the *quality* of generated invariants, in terms of invariant inclusion.

A closer analysis shows that ISL performs comparatively well on test cases encoded with *concurrent loops* (several loops on a single control point, similar to what is shown Figure 7), unlike PIPS whose results are particularly bad. On the other hand, ISL can be quite slow on test cases that display a large, intricate control structure. Finally, despite its successes, Aspic has greater difficulty to deal with transitions featuring complex formulas, that it is not able to accelerate. This leads us to wonder if the tools are sensitive to the encoding, since a problem can be presented under many guises. This is the topic of the next section, and the other original contribution of this paper.

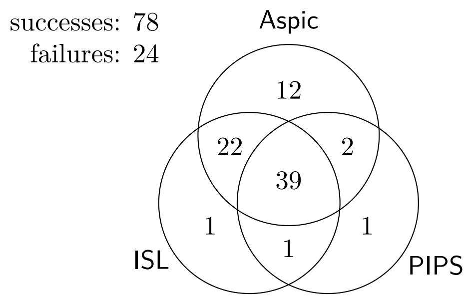


Figure 5: Venn diagram of successes for each tool

$\supseteq$	Aspic	ISL	PIPS
Aspic	–	21	23
ISL	49	–	54
PIPS	33	23	–

Table 2: Invariant inclusions

3 Model-to-Model Restructurations: Sensitivity to Encoding

We mentioned above the possibility to use ALICe to test the efficiency and relevance of model-to-model restructurations. An initial motivation was to try to improve results in PIPS by playing on the model structure.

Formally, a *sound model restructuration* is a function $\rho : \mathcal{M} \to \mathcal{M}$ that maps a model m_1 to a model m_2 such that: if m_2 is correct (i.e., its error region is not reachable), then m_1 is also correct:

$$\forall m_1, m_2 \in \mathcal{M}, m_2 = \rho(m_1) \wedge \text{correct}(m_2) \Longrightarrow \text{correct}(m_1). \tag{3}$$

Thus, given a model m_1 and a restructuration ρ, it is sufficient to prove the correctness of $m_2 = \rho(m_1)$ to deduce that m_1 is also correct. In addition, the restructuration ρ can be equivalent (m_1 is correct if and only if m_2 is correct), although we are not interested in proving such properties in our toolchain. So, for instance, the restructuration that to any model associates the same trivial, inconsistent model is sound (but not very interesting).

For analysis purposes, a model that fails to be checked can be rewritten into another one, hopefully easier to analyze. Within ALICe, we have implemented two restructurations on model states, to test analysis tools on a wider range of models and on specific model schemes, and explore the impact of model encoding. Model restructurations are performed at the very beginning of ALICe execution, just before Step 1., as an additional, preliminary stage to Figure 4. Both these restructurations were proved sound in Coq, using a trace-equivalence scheme: considering an arbitrary, possible state trace with transitions

$$\theta_1 = (k_0, v_0) \xrightarrow{t_1} (k_1, v_1) \xrightarrow{t_2} (k_2, v_2) \xrightarrow{t_3} \cdots$$

in the original model m_1, we show that for any corresponding trace θ_2 in the transformed model m_2 (corresponding to the same behavior, once the model has been transformed), then:

$$(\forall q_2 \in \theta_2, q_2 \notin Q_{\text{err}2}) \Longrightarrow (\forall q_1 \in \theta_1, q_1 \notin Q_{\text{err}1}),$$

thus ensuring (3).

3.1 Control-Point Splitting Heuristic

The first restructuration we use is a heuristic to split control nodes that contain several self loops. The global idea is to get rid of such nodes, that are usually the most difficult to automatically analyze, by splitting them with respect to the guards of the transitions, and adjusting the initial and error regions accordingly. This heuristic was initially designed for PIPS and is presented in details in [37].

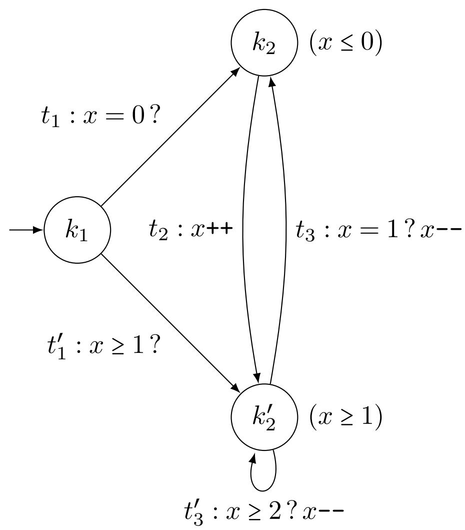


Figure 6: Model of Figure 1 transformed by the splitting heuristic

In Figure 6, we show the effect of this heuristic applied on the same model as in Figure 1, where the control point k_2 is split into two components with respect to the guards of transitions t_2 and t_3. The initial state set is unchanged. Proving that the new error state set $\{k_2, k'_2\} \times (x < 0)$ cannot be reached is easier than in the original model: it is true by construction for the control point k'_2, and can be easily deduced by looking at guards of entering transitions for control point k_2.

3.2 Reduction to a Unique Control Point

The other model restructuration reduces the set of controls to a unique control point ℓ, with all transitions turned into loops on ℓ, adding an extra boolean variable x_k for each original control point k in both guards and actions (assuming symbols ℓ and $\{x_k\}$ are not bound in the original model), to represent the corresponding control point of the original model. We assure that, in every state of the system, exactly one x_k is set to 1. Formally, a transition t between control states k_i, k_j, with guard g and action a

$$t : k_i \xrightarrow{g\,?\,a} k_j$$

is turned into the transition

$$t' : \ell \xrightarrow{g \wedge x_i = 1 \wedge \bigwedge_{k \neq i} x_k = 0\ ?\ a \wedge x'_j = 1 \wedge \bigwedge_{k \neq j} x'_k = 0} \ell.$$

Again, the initial and error set states are adjusted accordingly.

A more direct approach is to encode control information on a unique integer variable instead of several boolean variables, by numbering the control points in the initial model. We did not adopt it because it introduces encoding issues. Indeed, some relations on control points might or might not be expressed in terms of affine constraints, depending on the control-point encoding. For instance, being in k_1 or k'_2 in the model of Figure 6 cannot be expressed if the third control point k_2 is encoded with a value between those of k_1 and k'_2. This issue is avoided with our boolean variable scheme.

The resulting model of this restructuration applied on the initial model of Figure 1 is shown in Figure 7. The corresponding initial state set is: $Q_{\text{init}} = \{\ell\} \times \{b_1 = 1 \wedge b_2 = 0\}$. The error state set is: $Q_{\text{err}} = \{\ell\} \times \{b_1 = 0 \wedge b_2 = 1 \wedge x < 0\}$.

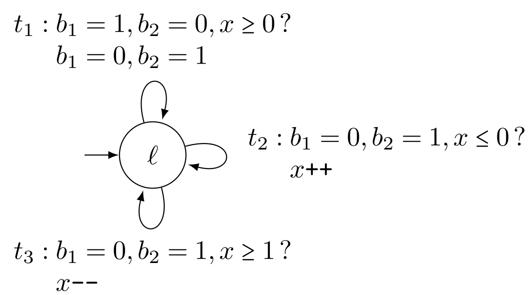


Figure 7: Model of Figure 1 reduced to a unique control point

This restructuration has three purposes. First, it stresses the tool with more difficult test cases. It also reduces bias factors related to encoding choices. Finally, if used before the control-point splitting heuristic, merging increases the effect of splitting by generating the control-point structure to which it applies.

3.3 Combining Restructurations

These model restructurations can be used independently or together: first the model is reduced to a unique control point, then this control point is split, widening the scope of the splitting heuristic. Therefore, ALICe works with four version of each model:

- The original version, with no restructuration (noted "direct" in the tables below);
- With all control points merged into a unique one, as described in Section 3.2 ("merged");
- Using the control-point splitting heuristic presented in Section 3.1 ("split");
- Combining both approaches ("merged-split").

3.4 Impact of Restructurations on Experimental Results

The results obtained using these restructuration schemes are displayed in Table 3.

We notice that the control state splitting heuristic leads to improved results for all tools, as shown by the comparison of Tables 3a and 3c on the one hand, and Tables 3b and 3d on the other. These results also confirm that ISL is significantly better in the treatment of concurrent loops, as previously noticed in Section 2.5: it outperforms the other tools on merged models (Table 3b). Aspic has lower scores than ISL on split models, with or without merging (Tables 3c and 3d), because it cannot accelerate transitions that are generated by control node merging. PIPS is the worst performer in all cases, but we managed to increase its success rate by about

	Aspic	**ISL**	**PIPS**
Successes	75	63	43
Time (s.)	10.9	35.5	46.2

(a) Direct

	Aspic	**ISL**	**PIPS**
Successes	59	70	40
Time (s.)	16.7	26.2	50.0

(b) Merged

	Aspic	**ISL**	**PIPS**
Successes	79	72	50
Time (s.)	12.8	43.0	61.7

(c) Split

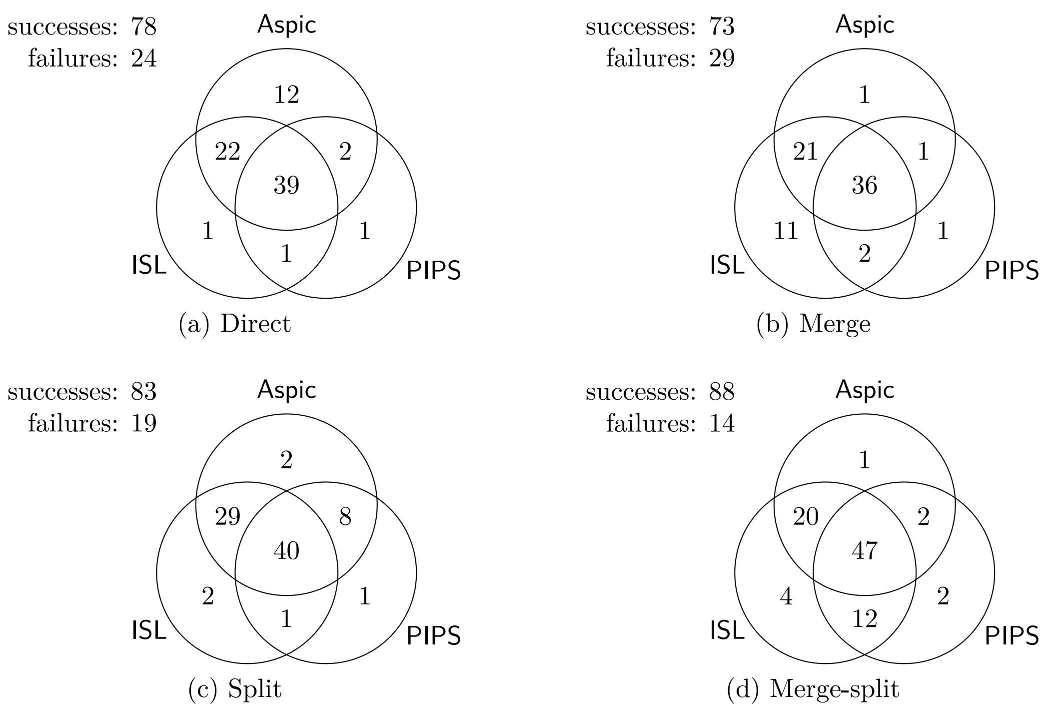
	Aspic	**ISL**	**PIPS**
Successes	70	83	63
Time (s.)	11.3	40.8	59.5

(d) Merged-split

Table 3: Benchmark results with different encodings

50 % (from Table 3a to Table 3d). The merge-splitting strategy gives the best results for ISL and PIPS.

Once again, detailed results are more contrasted. There is still no inclusion relation between successful test cases for different tools, whatever the restructuration scheme is chosen (Figure 8). The same holds for invariant sharpness.



(a) Direct (b) Merge (c) Split (d) Merge-split

Figure 8: Venn diagrams of successes for each tool

Globally, the merge-split restructuration leads to the best results with 88 out of 102 test cases correctly solved.

4 Conclusion and Future Work

In this paper, we described ALICe, a benchmark for automatic tools that compute affine loop invariants. The benchmark consisted in 102 test cases taken from previously published papers from the relevant literature, and helper tools that allowed to compare three invariant generators, Aspic, ISL and PIPS, handling their different formats. We also presented two model restructurations, splitting and merging, that can be used separately or together to improve results and highlight the differences between tools. Finally, we gave some insight on the benchmark results.

The ALICe toolset provides a framework to improve current results, either by working more deeply on the model restructurations, or improving the loop computation algorithms used in PIPS or other tools to deal with weaknesses in concurrent loop computation.

There are two ways to further pursue this work. The benchmark can be expanded, either by adding more test cases, which is relatively easy but time-consuming, or by integrating other analysis tools, such as FASTer [35, 4] or NBAC [33, 31]. Before adding new tools, it may be interesting to equip the models with a "minimum invariant", of which computed invariants should be supersets to be considered correct. This would reduce the problems posed by the presence of a buggy or cheating tool, that could generate wrong, too narrow invariants and would apparently pass all test cases. Adding deliberately incorrect models could also address this issue.

References

[1] Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Radhia Cousot, and Matthieu Martel, editors, *Static Analysis*, volume 6337, pages 117–133. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[2] Corinne Ancourt, Fabien Coelho, and François Irigoin. A modular static analysis approach to affine loop invariants detection. *Electronic Notes in Theoretical Computer Science*, 267(1):3–16, October 2010.

[3] Sébastien Bardin. *Vers un Model Checking avec accélération plate des systèmes hétérogènes*. PhD thesis, École normale supérieure de Cachan, 2005.

[4] Sébastien Bardin, Alain Finkel, and Jérôme Leroux. FASTer acceleration of counter automata in practice. In Gerhard Goos, Juris Hartmanis, Jan Leeuwen, Kurt Jensen, and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988, pages 576–590. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[5] Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Path invariants. page 300. ACM Press, 2007.

[6] Tevfik Bultan, Richard Gerber, and William Pugh. Symbolic model checking of infinite state systems using presburger arithmetic. In Gerhard Goos, Juris Hartmanis, Jan Leeuwen, and Orna Grumberg, editors, *Computer Aided Verification*, volume 1254, pages 400–411. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

[7] Centre de Recherche en Informatique, MINES ParisTech. PIPS, 2013.

[8] Aziem Chawdhary, Byron Cook, Sumit Gulwani, Mooly Sagiv, and Hongseok Yang. Ranking abstractions. In Sophia Drossopoulou, editor, *Programming Languages and Systems*, volume 4960, pages 148–162. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[9] Michael A. Colón and Henny B. Sipma. Synthesis of linear ranking functions. In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031, pages 67–81. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[10] Michael A. Colón and Henny B. Sipma. Practical methods for proving program termination. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification*, volume 2404, pages 442–454. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[11] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. *ACM SIGPLAN Notices*, 41(6):415, June 2006.

[12] Patrick Cousot. Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, and Radhia Cousot, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 3385, pages 1–24. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[13] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. pages 84–96. ACM Press, 1978.

[14] Paul Feautrier. c2fsm, 2010.

[15] Paul Feautrier and Laure Gonnord. Accelerated invariant generation for c programs with aspic and c2fsm. *Electronic Notes in Theoretical Computer Science*, 267(2):3–13, October 2010.

[16] Laure Gonnord. *Accélération abstraite pour l'amélioration de la précision en Analyse des Relations Linéaires*. PhD thesis, Université Joseph-Fourier - Grenoble 1, 2007.

[17] Laure Gonnord. Aspic, 2010.

[18] Denis Gopan and Thomas Reps. Lookahead widening. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, Thomas Ball, and Robert B. Jones, editors, *Computer Aided Verification*, volume 4144, pages 452–466. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[19] Denis Gopan and Thomas Reps. Guided static analysis. In Hanne Riis Nielson and Gilberto Filé, editors, *Static Analysis*, volume 4634, pages 349–365. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[20] Bhargav S. Gulavani, Thomas A. Henzinger, Yamini Kannan, Aditya V. Nori, and Sriram K. Rajamani. SYNERGY: a new algorithm for property checking. page 117. ACM Press, 2006.

[21] Sumit Gulwani. SPEED: symbolic complexity bound analysis. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Ahmed Bouajjani, and Oded Maler, editors, *Computer Aided Verification*, volume 5643, pages 51–62. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[22] Sumit Gulwani, Sagar Jain, and Eric Koskinen. Control-flow refinement and progress invariants for bound analysis. *ACM SIGPLAN Notices*, 44(6):375, May 2009.

[23] Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. SPEED: precise and efficient static estimation of program computational complexity. page 127. ACM Press, 2008.

[24] Sumit Gulwani and Florian Zuleger. The reachability-bound problem. page 292. ACM Press, 2010.

[25] Nicolas Halbwachs. *Détermination automatique de relations linéaires vérifiées par les variables d'un programme*. PhD thesis, Grenoble INP, 1979.

[26] Nicolas Halbwachs. Delay analysis in synchronous programs. In G. Goos, J. Hartmanis, and Costas Courcoubetis, editors, *Computer Aided Verification*, volume 697, pages 333–346. Springer Berlin Heidelberg, Berlin, Heidelberg, 1993.

[27] Nicolas Halbwachs. Linear relation analysis principles and recent progress, 2010.

[28] 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, August 1997.

[29] Julien Henry. Static analysis by abstract interpretation, path focusing, August 2011.

[30] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. pages 58–70. ACM Press, 2002.

[31] B. Jeannet. Dynamic partitioning in linear relation analysis: Application to the verification of reactive systems. *Form. Methods Syst. Des.*, 23(1):5–37, July 2003.

[32] Bertrand Jeannet. *Partitionnement dynamique dans l'Analyse de Relation Linéaire et application à la vérification de programmes synchrones*. PhD thesis, Grenoble INP, 2000.

[33] Bertrand Jeannet. NBAC, 2010.

[34] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon. Implementing a cache consistency protocol. *ACM SIGARCH Computer Architecture News*, 13(3):276–283, June 1985.

[35] Laboratoire Spécification et Vérification, ÉNS Cachan. FAST - fast acceleration of symbolic transition systems, 2006.

[36] Chin Soon Lee. Program termination analysis in polynomial time. In Don Batory, Charles Consel, and Walid Taha, editors, *Generative Programming and Component Engineering*, volume 2487, pages 218–235. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[37] Vivien Maisonneuve. Convex invariant refinement by control node splitting: a heuristic approach. *Electronic Notes in Theoretical Computer Science*, 288:49–59, December 2012.

[38] David Merchat. *Réduction du nombre de variables en analyse de relations linéaires*. PhD thesis, Université Joseph-Fourier - Grenoble 1, 2005.

[39] F.J. Pelletier, G. Sutcliffe, and C.B. Suttner. The development of CASC. *AI Communications*, 15(2-3):79–90, 2002.

[40] Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In Gerhard Goos, Juris Hartmanis, Jan Leeuwen, Bernhard Steffen, and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 2937, pages 239–251. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[41] Corneliu Popeea and Wei-Ngan Chin. Inferring disjunctive postconditions. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Mitsu Okada, and Ichiro Satoh, editors, *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues*, volume 4435, pages 331–345. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[42] William Pugh. The omega project, 2011.

[43] G. Sutcliffe and C. Suttner. The state of CASC. *AI Communications*, 19(1):35–48, 2006.

[44] Geoff Sutcliffe. The TPTP problem library and associated infrastructure: The FOF and CNF parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, December 2009.

[45] Sven Verdoolaege. isl: An integer set library for the polyhedral model. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama, editors, *Mathematical Software – ICMS 2010*, volume 6327, pages 299–302. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[46] Sven Verdoolaege. isl - integer set library, 2013.