ALICe: A Framework to Improve Affine Loop Invariant Computation
Vivien Maisonneuve, Olivier Hermant, François Irigoin

To cite this version:
Vivien Maisonneuve, Olivier Hermant, François Irigoin. ALICe: A Framework to Improve Affine Loop Invariant Computation. the 5th International Workshop on Invariant Generation (WING 2014), Jul 2014, Vienne, Austria. hal-01086957

HAL Id: hal-01086957
https://minesparis-psl.hal.science/hal-01086957
Submitted on 25 Nov 2014

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.
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 is shown in Figure 1.

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 $v : X \rightarrow \mathbb{Z}$ 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 $g : V \rightarrow \mathbb{B} = \{\bot, \top\}$, giving the value of a logic formula on a valuation $v$. In the example model, the guard in $t_2$ is:

$$g_2 : v \mapsto v(x) \leq 0,$$

simply noted “$x \leq 0$”.

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

Figure 1: An example of model
• 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:

$$a_2 = \{(v, v') \in V \times V \mid v'(x) = v(x) + 1\},$$

simply noted “$x++$”

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 $t_2 : k_2 \xrightarrow{x \leq 0 \? x++} k_2$. 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 $K$;
- $Q_{\text{init}}$ and $Q_{\text{err}}$ 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 $(Q, \rightarrow, Q_{\text{init}})$, which transition relation obeys to:

$$(k, v) \rightarrow (k', v') \iff \exists (k, g, a, k') \in T, g(v) = \top \land (v, v') \in a. \quad (1)$$

**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 $Q_{\text{err}}$ — representing “something bad” — cannot be reached. Their verification involves the computation of accessible states from the initial states $Q_{\text{init}}$.

Let $R$ be a subset of states $Q$. We note

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

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:

$$\text{Acc}(Q_{\text{init}}) \cap Q_{\text{err}} = \emptyset.$$  \hfill (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 $\text{Acc}(Q_{\text{init}})$. Instead, analysis tools tested by ALICe compute supersets $Q'$ of $\text{Acc}(Q_{\text{init}})$: if a given $Q'$ does not intersect with $Q_{\text{err}}$, then the tool that generated this $Q'$ has verified that states in $Q_{\text{err}}$ are unreachable (Figure 2), thus the test case is considered a success for that tool. However, if the intersection $Q' \cap Q_{\text{err}}$ 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](image)

Note that we could use backward analysis instead: starting from error states $Q_{\text{err}}$, computing iteratively the set of coaccessible states and testing the intersection with $Q_{\text{init}}$. 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.
• 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:

\[
S(s) = \{ x \in \mathbb{Z}^d \mid \exists z \in \mathbb{Z}^e : Ax + Bs + Dz \geq c \},
\]

\[
R(s) = \{ (x_1, x_2) \in \mathbb{Z}^{d_1} \times \mathbb{Z}^{d_2} \mid \exists z \in \mathbb{Z}^e : A_1 x_1 + A_2 x_2 + Bs + Dz \geq c \}.
\]

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 \texttt{fsm} format. This is a simple language that directly represents models, originally used in FAST [35] and then in Aspic. An example of \texttt{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 \texttt{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;
Listing 1: Source code in fsm format

```plaintext
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};
}
```

- 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 ALIe.
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 22105 (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.

<table>
<thead>
<tr>
<th></th>
<th>Aspic</th>
<th>ISL</th>
<th>PIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Successes</td>
<td>75</td>
<td>63</td>
<td>43</td>
</tr>
<tr>
<td>Time (s.)</td>
<td>10.9</td>
<td>35.5</td>
<td>46.2</td>
</tr>
</tbody>
</table>

Table 1: Benchmark results

This ranking is lessened 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.
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 $\rho : M \rightarrow M$ 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:

$$\forall m_1, m_2 \in M, m_2 = \rho(m_1) \land \text{correct}(m_2) \implies \text{correct}(m_1).$$

Thus, given a model $m_1$ and a restructuration $\rho$, 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 $\rho$ 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 $\theta_2$ in the transformed model $m_2$ (corresponding to the same behavior, once the model has been transformed), then:

$$(\forall q_2 \in \theta_2, q_2 \notin Q_{err2}) \implies (\forall q_1 \in \theta_1, q_1 \notin Q_{err1}),$$

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].

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\} × (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 $\ell$, with all transitions turned into loops on $\ell$, adding an extra boolean variable $x_k$ for each original control point $k$ in both guards and actions (assuming symbols $\ell$ and \{x\} 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 $a$

$$ t : k_1 \xrightarrow{g ? a} k_j $$

is turned into the transition

$$ t' : \ell \xrightarrow{g \land x_i = 1 \land \neg x_j \land x_k = 0 \land a \land x'_j = 1 \land \neg x_k \land x'_k = 0} \ell. $$

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: $Q_{\text{init}} = \{\ell\} \times \{b_1 = 1 \land b_2 = 0\}$. The error state set is: $Q_{\text{err}} = \{\ell\} \times \{b_1 = 0 \land b_2 = 1 \land x < 0\}$.

$$
t_1 : b_1 = 1, b_2 = 0, x \geq 0 \?
\quad b_1 = 0, b_2 = 1
\quad \ell
\quad t_2 : b_1 = 0, b_2 = 1, x \leq 0 \?
\quad x++
\quad t_3 : b_1 = 0, b_2 = 1, x \geq 1 \?
\quad x--
$$

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
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.

![Venn diagrams of successes for each tool](image)

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


[28] Nicolas Halbwachs, Yann-Erick Proy, and Patrick Roumanoff. Verification of real-time systems


